

# The **xintexpr** and allied packages

JEAN-FRANÇOIS BURNOL

jfbu (at) free (dot) fr

Package version: 1.4o (2025/09/06); documentation date: 2025/09/06.

From source file xint.dtx. Time-stamp: <07-09-2025 at 00:47:30 CEST>.

## Part I. The **xintexpr** package

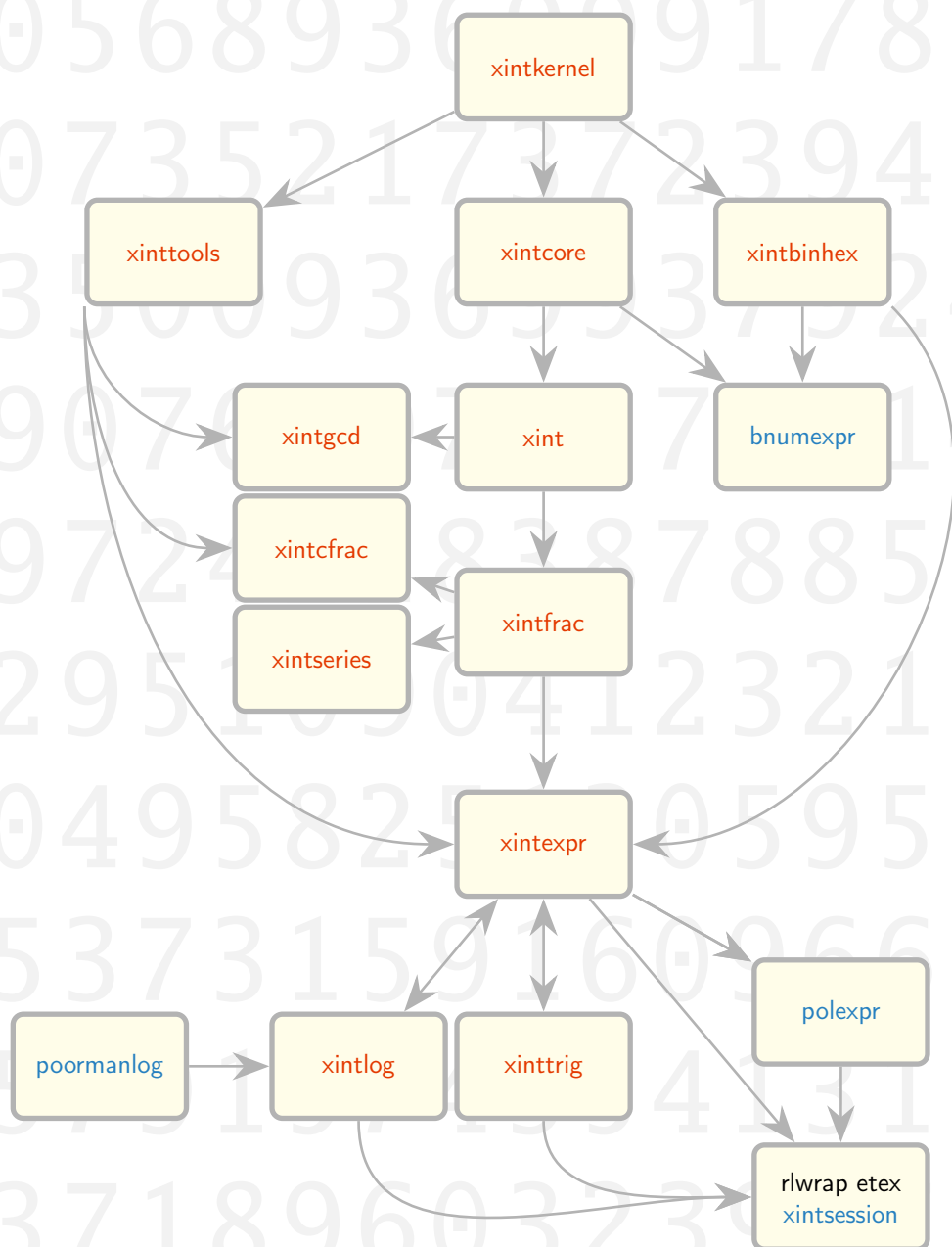
<b>1</b>	<b>Introduction</b>	3
<b>1.1</b>	Compatible engines and formats	3
<b>1.2</b>	Usage	4
<b>1.3</b>	xintsession	4
<b>1.4</b>	poexpr	5
<b>1.5</b>	bnumexpr	5
<b>1.6</b>	Printing big numbers on the page	6
<b>1.7</b>	Repository	6
<b>1.8</b>	License and installation instructions	6
<b>2</b>	<b>Syntax reference and user guide</b>	7
<b>2.1</b>	The three parsers	7
<b>2.2</b>	Output customization	10
<b>2.3</b>	Built-in operators and their precedences	14
	<i>Table of precedence levels of operators</i>	15
<b>2.4</b>	Built-in functions	19
	<i>Table of functions in expressions</i>	19
<b>2.5</b>	Generators of arithmetic progressions	36
<b>2.6</b>	Python slicing and indexing of one-dimensional sequences	37
<b>2.7</b>	NumPy like nested slicing and indexing for	
	arbitrary oples and nutples	38
<b>2.8</b>	Tacit multiplication	38
<b>2.9</b>	User defined variables	39
<b>2.10</b>	User defined functions	43
<b>2.11</b>	Examples of user defined functions	51
<b>2.12</b>	Links to some (old) examples within this document	54
<b>2.13</b>	Oples and nutples: the 1.4 terminology	55
<b>2.14</b>	Expansion (for geeks only)	59
<b>2.15</b>	Known bugs/features (last updated at 1.4n)	60
<b>3</b>	<b>The macros of <b>xintexpr</b> (ancient documentation, mostly)</b>	63
<b>4</b>	<b>The <b>xinttrig</b> package</b>	80
<b>5</b>	<b>The <b>xintlog</b> package</b>	84
<b>6</b>	<b>Macros of the <b>xinttools</b> package</b>	87
<b>7</b>	<b>Additional (old) examples with <b>xinttools</b> or <b>xintexpr</b> or both</b>	108

## Part II. The macro layer for expandable computations: **xintcore**, **xint**, **xintfrac**, and some extras

<b>8</b>	<b>The <b>xint</b> bundle</b>	127
<b>9</b>	<b>Macros of the <b>xintkernel</b> package</b>	142
<b>10</b>	<b>Macros of the <b>xintcore</b> package</b>	146
<b>11</b>	<b>Macros of the <b>xint</b> package</b>	151
<b>12</b>	<b>Macros of the <b>xintfrac</b> package</b>	163
<b>13</b>	<b>Macros of the <b>xintbinhex</b> package</b>	193
<b>14</b>	<b>Macros of the <b>xintgcd</b> package</b>	198
<b>15</b>	<b>Macros of the <b>xintseries</b> package</b>	200
<b>16</b>	<b>Macros of the <b>xintcfrac</b> package</b>	216

## Part III. The **xintexpr** and allied packages source code

<b>17</b>	<b>An introduction and a brief timeline</b>	233
<b>18</b>	<b>Package <b>xintkernel</b> implementation</b>	236
<b>19</b>	<b>Package <b>xinttools</b> implementation</b>	259
<b>20</b>	<b>Package <b>xintcore</b> implementation</b>	302
<b>21</b>	<b>Package <b>xint</b> implementation</b>	359
<b>22</b>	<b>Package <b>xintbinhex</b> implementation</b>	400
<b>23</b>	<b>Package <b>xintgcd</b> implementation</b>	420
<b>24</b>	<b>Package <b>xintfrac</b> implementation</b>	430
<b>25</b>	<b>Package <b>xintseries</b> implementation</b>	525
<b>26</b>	<b>Package <b>xintcfrac</b> implementation</b>	534
<b>27</b>	<b>Package <b>xintexpr</b> implementation</b>	557
<b>28</b>	<b>Package <b>xinttrig</b> implementation</b>	689
<b>29</b>	<b>Package <b>xintlog</b> implementation</b>	712
<b>30</b>	<b>Cumulative line and macro count</b>	750



Dependency graph for the **xint bundle** components. Modules pointed to by arrows **automatically** import the module from which the arrow originates.

**bnumexpr** is a  $\text{\LaTeX}$  package by the author which uses (by default) **xintcore** as its mathematical engine. To use it under Plain  $\text{eTeX}$  issue first `\input miniltx.tex` then `\input bnumexpr.sty`.

**polexpr** handles definitions and algebraic operations on one-variable polynomials, as well as root localization to arbitrary precision. It works both with Plain  $\text{TeX}$  and with  $\text{\LaTeX}$ .

**xinttrig** and **xintlog** are loaded automatically by **xintexpr**; they should not be loaded directly via a separate `\usepackage` (in  $\text{\LaTeX}$ ).

**poormanlog** is a  $\text{TeX}$  and  $\text{\LaTeX}$  package by the author which is loaded automatically by **xintlog**.

**xintsession** is invoked on the command line as `etex xintsession` (or, much better if available: `rlwrap etex xintsession`).

# Part I.

## The **xintexpr** package

1	Introduction .....	3
2	Syntax reference and user guide .....	7
3	The macros of <b>xintexpr</b> (ancient documentation, mostly) .....	63
4	The <b>xinttrig</b> package .....	80
5	The <b>xintlog</b> package .....	84
6	Macros of the <b>xinttools</b> package .....	87
7	Additional (old) examples with <b>xinttools</b> or <b>xintexpr</b> or both .....	108

## 1. Introduction

.1	Compatible engines and formats .....	3	.5	bnumexpr .....	5
.2	Usage .....	4	.6	Printing big numbers on the page .....	6
.3	xintsession .....	4	.7	Repository .....	6
.4	poexpr .....	5	.8	License and installation instructions .....	6

JÜRGEN GILG's interest into what he called "**XINT**" was instrumental in keeping the author motivated over the years. We exchanged on many topics extending beyond  $\TeX$  and often reacted similarly to private and public events. I knew he was a very kind and devoted person, who took care of the needs of others prior to his own, although he never mentioned it. Jürgen suffered a sudden, unexpected, and deadly stroke in May 2022. I will miss his friendship profoundly.

### 1.1. Compatible engines and formats

The components of the **xint bundle** can be used indifferently with Plain  $\TeX$  (and other formats, as mentioned next) or with  $\LaTeX$ . The sole difference being that with the latter the loading must be done by `\usepackage` whereas with any non- $\LaTeX$  format it has to be via `\input` (using **.sty** filename extension, not **.tex**).

The engine can be PDF $\TeX$ , Xe $\TeX$ , or Lua $\TeX$ .

With release **1.4n** you can also use the packages with Con $\TeX$ t (only latest one, with LuaMeta $\TeX$  engine), and Op $\TeX$ .

You can't use the **xint bundle** with Knuth's original **tex** binary, because its functionalities require `\numexpr` and other e- $\TeX$  extensions as well as the more recent `\expanded` engine primitive (and `\pdfstrcomp` or `\strcmp`).

**xintexpr** will be probably the main entry point, and it actually automatically loads most other components. The aim of **xintexpr** is to provide expandable parsers of numerical expressions, either floating point numbers or fractions.

Here is an example:

```
\xintfloateval{cos(3Pi/17)*sin(1)^0.123 + log(3.42e5)}
```

13.57492307809003

```
\xintSetDigits*{32}
\xintfloateval{\cos(3\pi/17)*\sin(1)^{0.123} + \log(3.42e5)}
```

And with still more digits (the `[-2]` rounds away the two least significant digits of the result):

```
\xintSetDigits*{62}
\xintfloateval[-2]{cos(3Pi/17)*sin(1)^0.123 + log(3.42e5)}
13.5749230780900311479955540398179965151694656991408063951975
```

```
\xinteval{reduce(add(1/i^3, i=1..25))}
```

And two examples with large integers:

```
$2^{1000}=\printnumber{\xinteval{2^1000}}$. \newline
$100!^3=\printnumber{\xintiieval{100!^3}}$.
```

[illegible]

The table of [built-in functions](#) and the one of [built-in operators](#) will give a quick overview of the available syntax.

The simplest way<sup>1</sup> to test the syntax is to work interactively on the command line (this feature is available since April 2021, the version of `xintsession` used here is `1.3a`). Beware though that ill-formed inputs will trigger  $\text{\TeX}$  famously antiquated error handling, from which it is hard to recover, although hitting `S` may sometimes miraculously bring you back to the `xintsession` prompt.

```
rlwrap etex xintsession
[...welcome banner...]
  Magic words: `&pause' (or `;'), `&help', `&bye',
               `&exact', `&fp', `&int', `&pol'.

  \jobname is xintsession
  Transcript will go to log and to xintsession-210609_12h00.tex
  Starting in exact mode (floating point evaluations use 16 digits)
>>> 2^100;
@_1      1267650600228229401496703205376
>>> cos(1);
```

4

```

@_2      0.5403023058681397
>>> &fp=32
(/usr/local/texlive/2021/texmf-dist/tex/generic/xint/xintlog.sty)
(/usr/local/texlive/2021/texmf-dist/tex/generic/xint/xinttrig.sty)
fp mode (log and trig reloaded at Digits=32)
>>> cos(1);
@_3      0.54030230586813971740093660744298
>>> 3^1000;
@_4      1.3220708194808066368904552597521e477
>>> &exact
exact mode (floating point evaluations use 32 digits)
>>> 3^1000;

@_5      13220708194808066368904552597521443659654220327521481676649203682268285
9734670489954077831385060806196390977769687258235595095458210061891186534272525
7953674027620225198320803878014774228964841274390400117588618041128947815623094
4380615661730540866744905061781254803444055470543970388958174653682549161362208
3026856377858229022841639830788789691855640408489893760937324217184635993869551
6765018940588109060426089671438864102814350385648747165832010614366132173102768
902855220001
>>> &bye
Did I say something wrong?
Session transcript written on xintsession-210609_12h00.tex
)
No pages of output.
Transcript written on xintsession.log.

```

## 1.4. `polexpr`

The package `polexpr` enriches the `\xinteval` syntax (but not the one of `\xintfloateval`) with a polynomial type with associated constructor `pol([c0,c1,...])`, and polynomial specific functions such as `polgcd(pol1, pol2, ...)`.

Full usage of polynomials (and algebraic notations `c_0 + c_1 x + c_2 x^2 + ...` for input and also output) goes through a dedicated `\poldef` parser which is based upon `\xintdefvar`/`\xintdeffunc` and is a necessary step to then access via a dedicated macro interface operations such as identifying all rational roots and isolating all real roots to arbitrary precision.

The simplest manner to experiment with `polexpr` is via the `&pol` mode of `xintsession`.

## 1.5. `bnumexpr`

This ~~TeX~~ package loads `xintcore` and `xintbinhex` and provides `\bnumeval` which is a scaled-down `\xintiieval` (omitting support for nested structures, functions, variables, boolean branching, etc...). It can be used with Plain eTeX, thanks to `miniltx`. For this, use `\input miniltx.tex` followed by `\input bnumexpr.sty` (remark: `miniltx` is not needed for `xintexpr`).

`\bnumeval` is thus a boosted `\inteval` which addition of support for arbitrarily large integers, powers with `**` and `^`, rounded division with `/`, floored division with `//` and associated modulo `/:`, factorials via `!` postfix operator, comma separated multi-item expressions.

It also supports as `\xintiieval` does the `'`, `"`, `0b`, `0o` and `0x` input prefixes and the optional arguments `[b]`, `[o]`, or `[h]`.

Further, it provides an interface (which does not exist with `xintexpr`) to let all operations be done by macros of one's own choosing, as replacement for some or all of the operations by default implemented via the help of `xintcore` and `xintbinhex`. It even makes it possible to add to the syntax extra infix or postfix operators and to modify the precedence levels of those already defined.

## 1.6. Printing big numbers on the page

When producing very long numbers there is the question of printing them on the page, without going beyond the page limits. In this document, I have most of the time made use of a `\printnumber` macro, which is not provided by the package. A primitive form would be:

```
\def\allowsplits #1{\ifx #1\relax \else #1\hskip 0pt plus 1pt\relax
\expandafter\allowsplits\fi}%
\def\printnumber #1{\expandafter\allowsplits \romannumeral-`0#1\relax }%
```

This macro triggers `f-expansion` of its argument (and indeed `\xinteval` and friends expand completely under such trigger), then it goes through the computation result character by character inserting TeX potential break points in-between them. It is ineffective in math mode, one would need to add some `\allowbreak`'s. The `\printnumber` used for building this documentation uses slightly different and more sophisticated mechanisms and can be found in the source file `xint.dtx`.

## 1.7. Repository

It is at <https://github.com/jfbu/xint>. At this stage, it does not record real-time development status but only actual successive CTAN releases since 2013. Use it to report issues. Don't forget to include @jfbu in the ticket else I will not be pinged.

A front page at <https://jfbu.github.io/xint> provides, in addition to the present `xint.pdf` and to `README.md` a file `CHANGES.html`, which contains the complete list of changes relevant to user level since the initial release of the package:

<https://jfbu.github.io/xint/CHANGES.html>

Its version `xintchanges.md` in Markdown format is included in the CTAN upload,  
`texdoc xintchanges.md`

Warning: I don't have the time to maintain perfectly such large documentation. It combines old documentation which never really got updated and may be locally obsolete with more recent stuff mostly written on occasion of the 1.4 release of January 2020 and the 1.4e one of May 2021, and the intervening changes might also have made some of it not completely accurate, despite my best efforts.

## 1.8. License and installation instructions

The `xint bundle` components are made available under the [LaTeX Project Public License 1.3c](#). They are included in all major TeX distributions, thus there is probably no need for a custom install: just use the package manager to update if necessary the `xint bundle` components to the latest version available.

Else, CTAN access provides `xint.tds.zip` which has all source code and documentation in a TDS-compliant archive, only waiting to be `unzip -d <DIR>` into some suitable hierarchical structure.

See <https://jfbu.github.io/xint> for how to build from the CTAN `xint.dtx` source file.

## 2. Syntax reference and user guide

.1	The three parsers.....	7	arbitrary oples and nutples .....	38
.2	Output customization .....	10	.8	Tacit multiplication.....
.3	Built-in operators and their precedences ..	14	.9	User defined variables.....
<i>Table of precedence levels of operators .....</i>		15	.10	User defined functions .....
.4	Built-in functions.....	19	.11	Examples of user defined functions .....
<i>Table of functions in expressions .....</i>		19	.12	Links to some (old) examples within this document.....
.5	Generators of arithmetic progressions .....	36	.13	Oples and nutples: the 1.4 terminology...
.6	Python slicing and indexing of one-dimensional sequences.....	37	.14	Expansion (for geeks only) .....
.7	NumPy like nested slicing and indexing for		.15	Known bugs/features (last updated at 1.4n)
				60

**WARNING:** this documentation goes sometimes into too much details, and does need some improvements. But there is no time for that at 1.4n. Although people do not believe me when I say that, there is ample intellectual reward in actually reading the documentation, and it would be nice if at least, at last someone on Earth did, once (as JÜRGEN GILG some years back).

### 2.1. The three parsers

**xintexpr** provides three numerical expression parsers corresponding to these three respective tasks:

**\xintfloateval:** evaluations with floating point numbers; the default precision is with 16 digits, it can be set via **\xintSetDigits\***,  
**\xinteval:** exact evaluations with fractions, decimal fixed point numbers, numbers in scientific notation, with no size limitation,  
**\xintiieval:** evaluations allowing only integers with no size limitation,

and two secondary ones which act like the exact evaluator then round the output to a given number of fractional digits, or convert them to **false** or **true** according to whether they vanish or do not vanish.

Changed  
at 1.4m!

Please note the following:

- If you find that **\xintfloateval** is too much of a mouthful, you create an alias named, for example, **\fpeval**. Oops, no, that is ~~TeX~~<sup>TeX</sup> very efficient floating point engine. It is faster at its (unchangeable) precision of 16 decimal digits than **\xintfloateval** due to various reasons, one of them being that **xintexpr** birth was related to big integers only, and floating point support in arbitrary precision was added on top of that, via some expedients which have never been refactored, in view of the massive work that this would entail by now. But for example you could do **\let\evalfp\xintfloateval** if you want a shorter name. By the way **\xintfloateval** could very well have been christened **\xintfpeval** but when the author wrote the first release in 2013 he was barely if at all aware of existence of ~~TeX~~<sup>TeX</sup>, and of its **l3fp** component.
- Although **\xinteval** manipulates arbitrarily long integers or fractions it also accepts scientific notation on input, as well as all the mathematical functions (evaluated using the prevailing digits precision), and (depending on customization) can thus produce also scientific notation on output.
- So far, individual operations and the printing routine of **\xinteval** do not automatically reduce fractions to their lowest terms.

- `\xinteval{expression}` handles integers, decimal numbers, numbers in scientific notation and fractions. The algebraic computations are done *exactly*, and in particular `/` simply constructs fractions. Use `//` for floored division.

The output in this specific example came out irreducible. In general one needs a `reduce()` wrapper for an irreducible output:

Arbitrarily long numbers are allowed in the input. The space character (contrarily to the situation inside `\numexpr`) and also the underscore character (as allowed in Python too) can serve to separate groups of digits for better readability. But the package currently provides no macros to let the output be formatted with such separators.

New with  
1.4n

Hexadecimal, octal and binary (with fractional part allowed) can be input using suitable pre-  
fixes: respectively " or **0x**, ' or **0o**, and **0b**:

- `\xintieval[D]` is the same parser as `\xinteval`, i.e. accepts the same inputs and does all computations exactly in the same manner, but it then rounds its final result to the nearest integer, or, in case there is an optional argument `[D]`, to:
  - if `D>0`: the nearest fixed point number with `D` digits after the decimal mark,
  - if `D=0`: the nearest integer (as for `\xinteval` with no optional argument),
  - if `D<0`: the rounded quotient by  $10^{(-D)}$ .

The optional argument [*D*] can also be located *within* the braces at the start of the expression (this was actually the legacy syntax until 1.4k).

- `\xintiieval{⟨expression⟩}` executes computations on (big) integers only. It is (only slightly) faster than `\xinteval` for the same expression.

Attention: the forward slash `/` does the *rounded* integer division to match behaviour of `\numexpr`. The `//` operator does floored division as in `\xinteval`. The `/:` is the associated modulo operator (we could easily let the catcode `12 %` character be an alias, but using such an unusual percent character would be a bit cumbersome in a  $\text{\TeX}$  workflow, if only for matters of syntax highlighting in  $\text{\TeX}$ -aware text editors).

New with  
1.4n

An optional argument **[h]**, **[o]**, or **[b]** says to convert the output to hexadecimal, octal or binary:

8



- `\xintfloateval[⟨Q⟩]{⟨expression⟩}` does floating point computations with a given precision, which defaults to 16. The precision `P` can be set using `\xintDigits*:=P\relax` or `\xintSetDigits*{P}` syntaxes.

Its optional argument `[Q]`, if present, means to do a *final* float rounding to a mantissa of `Q` digits (this thus makes sense only if  $Q < P$ ).

A negative `Q` is allowed and means to round to  $P+Q$  digits only.

Prior to 1.4k the optional argument `[⟨Q⟩]` had to be located *within* the braces at the start of the expression. The legacy syntax is and will keep being allowed.

The infix operator `/` will compute the correct rounding of the exact fraction. The operator `//` is floored division and `/:` is its associated modulo (see also `divmod()`).

```
\begingroup
\xintDigits:=64\relax
\xintfloateval{sqrt(3)}
\endgroup
1.732050807568877293527446341505872366942805253810380628055806979
```

The four basic operations and the square root achieve *correct rounding*.<sup>2</sup>

On output, `\xintfloateval` uses `\xintPFloat` for each numeric leaf. This can be modified (cf. `\xintfloatexprPrintOne`).

There is a core syntax:

- `\xintexpr⟨expression⟩\relax`,
- `\xinttiexpr⟨expression⟩\relax`,
- `\xintiexpr⟨expression⟩\relax`,
- `\xintfloatexpr⟨expression⟩\relax`,
- `\xintboolexpr⟨expression⟩\relax`.

`\xintboolexpr⟨expression⟩\relax` does all computations like `\xintexpr` then converts all (non-empty) leaves<sup>3</sup> to `true` or `false` (cf. `\xintboolexprPrintOne`). There is no `\xintbooleval`.<sup>4</sup>

Changed  
at 1.4m!

Formerly the `\xintexpr... \relax` legacy syntax had to be prefixed by `\xintthe` if in typesetting context, else an error was raised (deliberately). The `\xintthe` prefix was made optional at 1.4.

In an `\edef` these constructs expand to some braced nested data, all computations having been completely done, which is prefixed with some `\protected` “typesetter” macros.

In an `\edef`, `\xinteval` (in contrast to `\xintexpr`), or `\xintfloateval` (in contrast to `\xintfloatexpr`) expand the “typesetting macros” and the final complete expansion consists of explicit digits and other characters such as those of scientific notation or square brackets.<sup>5</sup>

In  $\TeX$  it is possible to use the core syntax `\xintexpr⟨expression⟩\relax` also in so-called moving arguments, because when written out to a file the final expansion outcome uses only standard catcodes and thus will get retokenized and expand as expected if it has been written to an external file which is then reloaded.

One needs `\xinteval` et al. only if one really wants the final digits (and other characters), for example in a context where  $\TeX$  expects a number or a dimension.

As alternative to `\xinteval{⟨expression⟩}`, an equivalent is `\xintthe\xintexpr⟨expression⟩\relax`. Similarly `\xintthe` can prefix all other core parsers. And one can also use `\xinttheexpr` as shortcut for `\xintthe\xintexpr`.

Doing exact computations with fractions leads very quickly to very big results (and furthermore one needs to use explicitly the `reduce()` function to convert the fractions into smallest terms). Thus most probably what you want is `\xintfloateval` and `\xintfloatexpr`.

<sup>2</sup> when the inputs are already floating point numbers with at most `P`-digits mantissas. <sup>3</sup> Currently, empty leaves are output using `\xintexprEmptyItem`, i.e. default to `[]`. This may change. <sup>4</sup> This was `True` and `False` prior to 1.4m. <sup>5</sup> `\xinteval` and `\xintexpr` both expand completely in exactly two steps. And `\xintexpr` expands fully under *f-expansion* (of the `\romannumeral 0` or `-`0` type). As per `\xinteval` attention that it may expand to nothing, then naturally *f-expansion* propagates to tokens following up in the input stream.

## 2.2. Output customization

[source](#)

### 2.2.1. `\xintfloatexprPrintOne` et al. for numerical values

The package provides only minimal facilities for formatting the output from `\xinteval` or `\xintfloateval` or... And this output may well consist of comma separated values, even nested ones with, by default, square brackets. First we explain how to influence the handling of individual “leaves”.

Here are the default definitions to this effect:

```
\def\xintexprEmptyItem{[]}% (all parsers)
\def\xintexprPrintOne #1{\xintFracToSci{#1}}% \xinteval
\def\xintiexprPrintOne #1{\xintDecToString{#1}}% \xintieval
\def\xintiiexprPrintOne#1{#1}% \xintiiieval
\def\xintfloatexprPrintOne [#1]#2{\xintPFloat[#1]{#2}}% \xintfloateval
\def\xintboolexprPrintOne#1{\xintiifNotZero{#1}{true}{false}}
```

They can be re-defined to one's wishes. If configured to do anything non expandable they must be `\protected`.  $\TeX$  users will want to use `\RenewDocumentCommand` for this.

$\TeX$ -hackers note:

- Actually, the defaults are more done in the style

```
\let\xintexprPrintOne\xintFracToSci
```

thus sparing grabbing the argument `#1`. And one can do

```
\def\xintexprPrintOne{\xintFracToSci}
```

too.

- `\xintexprPrintOne` defaults in truth to some private variant of `\xintFracToSci` with exactly the same output but able to understand only certain limited types of inputs as used internally.
- `\xintiiexprPrintOne` is used with `\xintiiieval`. But it gets replaced with `\xintiiexprPrintOneHex`, `\xintiiexprPrintOneOct`, or `\xintiiexprPrintOneOct` if the optional argument `[h]`, `[o]`, or `[b]` is used. These macros default to respectively `\xintDecToHex`, `\xintDecToOct` and `\xintDecToBin`.
- `\xintfloatexprPrintOne` defaults in fact to a private variant of `\xintPFloat` which assumes the optional argument `[P]` is present as it will be the case always in this context. This optional argument `[P]` is the optional argument `[Q]` of `\xintfloateval` (or `Digits+Q` if `Q<0`).
- The typesetter for `\xintiiexpr` simply prints “as is”, but this may change in future, if some internal format is used requiring a conversion step.

New with  
1.4n

Here is a possibly not up-to-date list of macros of interest, whose documentations you might consider reading (the first two require math mode):

- `\xintTeXFromSci`,
- `\xintTeXFrac`,
- `\xintDecToString`,
- `\xintPRaw`,
- `\xintFracToSci`,
- `\xintFracToDecimal`,
- `\xintPFloat`,
- and `\xintFloatToDecimal`.

Naming scheme, as one can see, has been pretty much incoherent, apologies.

Among packages providing macros formatting numeric values, there are `numprint` and its macro `\np` (or `\numprint` without the option `np`), and `siunitx` and its `\num`, and possibly more packages not known to the author.<sup>6</sup> These macros are suitable in combination with `\xintFloat` as in the example below to customize the `\xintfloateval` output. Numerical output from `\xinteval` is more challenging as individual values may naturally contain the `/` character for fractions which the above mentioned packages will not know how to handle, as far as I know.

<sup>6</sup> There does not seem to be yet a  $\TeX$  user level interface to the `l3str-format` package, part of `l3experimental`, which provides an implementation of the Python `format` function.

Here an example, with  $\TeX$  and `\num` from [siunitx](#):

```
\RenewDocumentCommand\xintfloatexprPrintOne{o m}{\num{\xintFloat[#1]{#2}}}
```

We could have used here simply `\def` with delimited parameters `[#1]#2` because:

- the optional argument will always be present at time of use,
- `\num` is a `\protected` macro.

Note that when using only `\def` for the definition, the argument of `\num` is getting to be expanded first, but `\num` would have done that anyhow.

With `numprint`, one can similarly do:

```
\RenewDocumentCommand\xintfloatexprPrintOne{o m}{\numprint{\xintPFloat[#1]{#2}}}
```

This used `\xintPFloat` rather than `\xintFloat` as with `\num`. This is because (in my limited testing) `\numprint` with not silently remove a zero scientific exponent but it will typeset it, for example as  $1.5 \cdot 10^0$ . So we use our own `\xintPFloat` poor man “prettifier”.

Maybe you want to use a macro which is unable to have `\xintPFloat[#1]{#2}` as argument because it needs to see only a number in scientific notation and nothing else. If that macro is `\protected`, do the definition with `\def`. If it is not `\protected` and not purely expandable either, one can do this:

```
\protected\def\myfoo{\foo}%<<<--- with options perhaps
\def\xintfloatexprPrintOne[#1]#2{\myfoo{\xintPFloat[#1]{#2}}}
```

Then when `\myfoo` finally expands, its argument has been expanded already.

The current behaviour of `\xintfloateval` corresponds to this set-up:

```
\def\xintfloatexprPrintOne [#1]#2{\xintPFloat[#1]{#2}}
```

and to this default configuration of `\xintPFloat`:

```
\def\xintPFloatE{e}
\def\xintPFloatZero{0}
\def\xintPFloatIntSuffix{}
\def\xintPFloatLengthOneSuffix{}
\def\xintPFloatNoSciEmax{5}
\def\xintPFloatNoSciEmin{-4}
\def\xintPFloatMinTrimmed{4}
```

With the custom replacement

```
\def\xintfloatexprPrintOne{\xintFloatToDecimal}
```

the `\xintfloateval` output will use decimal fixed point notation, i.e. no scientific exponents, and as many zeros as are needed (but no more, as trailing zeros will be removed from the significant digits). Here is an example comparing outputs from the default configuration and custom ones:

```
\xintfloateval{exp(-32.456)/2000} (default, i.e. PFloat)\newline
\def\xintfloatexprPrintOne{\xintFloatToDecimal}%
\xintfloateval{exp(-32.456)/2000} (FloatToDecimal)\newline
\def\xintfloatexprPrintOne[#1]#2{\xintTeXFromSci{\xintFloat[#1]{#2}}}%
$\xintfloateval{exp(-32.456)/2000}$ (TeXFromSci on Float)\par % math mode required
```

4.013361680161317e-18 (default, i.e. PFloat)

0.00000000000000000004013361680161317 (FloatToDecimal)

4.013361680161317 · 10<sup>-18</sup> (TeXFromSci on Float)

Some examples showing now the effect of sensible customizations on `\xinteval`:

```
\xinteval{exp(-32.456)/2000} (default, i.e. FracToSci)\newline
\def\xintexprPrintOne{\xintFracToDecimal}%
\xinteval{exp(-32.456)/2000} (FracToDecimal)\newline
\def\xintexprPrintOne#1{\xintTeXFromSci{\xintFracToSci{#1}}}%
$\xinteval{exp(-32.456)/2000}$ (TeXFromSci on FracToSci)\par % math mode required
```

8.026723360322633e-15/2000 (default, i.e. FracToSci)

0.0000000000000000008026723360322633/2000 (FracToDecimal)

8.026723360322633 · 10<sup>-15</sup> · 2000<sup>-1</sup> (TeXFromSci on FracToSci)

Notice that the `/2000` denominator remains “as is” in the output, in conformity with the docu-

A slightly more costly typesetter could be for example:

Then

- the fraction (inclusive of its power of ten part) will be reduced to lowest terms (see `\xint-Irr`),
- next the trailing zeros will be moved as an exponent (positive or negative) to the numerator,
- this numerator with a power of ten part will be printed in decimal fixed point notation, with as few zeros as are needed,
- and finally the denominator **B**, which has been trimmed of trailing zeros, will be printed as `/B` or not at all if **B=1**.

$$0.000000000000000000008026723360322633/2$$

```
\def\xintexprPrintOne#1{\xintDecToStringREZ{\xintPIrr{#1}}}
```

One can also consider this for math mode:

$$\frac{0.0000000000000000008026723360322633}{2}$$

See our hesitations about what `\xintTeXFromSci` should do with denominators.

**TeX-hackers note:** The macro used as customization of `\xintexprPrintOne` (whose default is a private variant of `\xintFracToSci` with exactly same output) must understand the internal `xintfrac` format `A/B[N]`, but with the `/B` and `[N]` parts being only optional. This is not a problem when using for this task (nested) macros of `xintfrac`, as they of course accept such inputs as argument and in fact much more general ones.

In particular one can benefit from `\xintRaw`, or `\xintRawBraced`, to convert the argument into a well defined shape (`A/B[N]` for the former and `{N}{A}{B}` for the latter) and then work from there.

The macro used by `\xintfloatexprPrintOne` has the guarantee that the `[P]` will be always present at expansion time.

The customization should be compatible with being exposed to `\expanded` (which is like expansion in an `\edef`), either from being completely expandable or at the opposite from being `\protected`. `\TeX` commands defined via `\newcommand` as macros with one optional parameter are not compatible with this requirement.

Attention! The interface requirements described above for the macros customizing the behaviours of `\xintexprPrintOne` and `\xintfloatexprPrintOne` may change at any release... as they depend on some internal structures and it is not certain backwards compatibility will be maintained systematically in case of evolution.

### 2.2.2. `\xintthealign` for output of general oples

```
\xintthealign\xintexpr ndseq(1/(i+j), i = 1..10; j=1..10)\relax
```

```
[[ 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, 1/10, 1/11 ],
[ 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, 1/10, 1/11, 1/12 ],
[ 1/4, 1/5, 1/6, 1/7, 1/8, 1/9, 1/10, 1/11, 1/12, 1/13 ],
[ 1/5, 1/6, 1/7, 1/8, 1/9, 1/10, 1/11, 1/12, 1/13, 1/14 ],
[ 1/6, 1/7, 1/8, 1/9, 1/10, 1/11, 1/12, 1/13, 1/14, 1/15 ],
[ 1/7, 1/8, 1/9, 1/10, 1/11, 1/12, 1/13, 1/14, 1/15, 1/16 ],
[ 1/8, 1/9, 1/10, 1/11, 1/12, 1/13, 1/14, 1/15, 1/16, 1/17 ],
[ 1/9, 1/10, 1/11, 1/12, 1/13, 1/14, 1/15, 1/16, 1/17, 1/18 ],
[ 1/10, 1/11, 1/12, 1/13, 1/14, 1/15, 1/16, 1/17, 1/18, 1/19 ],
[ 1/11, 1/12, 1/13, 1/14, 1/15, 1/16, 1/17, 1/18, 1/19, 1/20 ]]
```

Attention, this `\xintthealign` must be a prefix to `\xintexpr`, or `\xintfloatexpr` etc..., but there will be low-level  $\TeX$  errors if it is used to prefix `\xinteval` et al. or `\xinttheexpr` et al.

It is possible to customize the behaviour of `\xintthealign`. For example:

```
\protected\def\xintexpralignbegin      {\halign\bgroup\tabskip2ex\hfil##&&##\hfil\cr}%
\def\xintexpralignend                  {\crcr\egroup}% removed \protected at 1.4c
\protected\def\xintexpralignlinesep    {\,\cr}% separates "lines"
\protected\def\xintexpralignleftsep    {\&}% at left of first item in a "line"
                                         % (after "left bracket")
\protected\def\xintexpraligninnersep    {\,&}% at the left of non-first items
\protected\def\xintexpralignrightsep    {\&}% at right of last item in a "line"
                                         % (before "right bracket")
\protected\def\xintexpralignleftbracket {\[%
\protected\def\xintexpralignrightbracket{]}%
```

The above definitions use `\protected` with no strong reason, as the replacement tokens are not expanding anyhow, but the idea is that this allows to execute a computation via an `\edef` and later one can change the meaning of the auxiliary macros depending on what one wants to do with the expansion result.

**$\TeX$ -hackers note:** `\xintexpralignend` is expanded once, after the body has been submitted to exhaustive expansion (`\expanded` induced), and prior to the expansion of `\xintexpralignbegin`.

**Unstable!** Although we will try to keep stable the way “regular arrays” as in the above example are rendered by default, the `\xintthealign` macro (and its associated customizability) is to be considered work-in-progress and may experience breaking changes.

Use for example this for outputting to a file or a terminal:<sup>7</sup>

```
% Better here without \protected.
% We assume here \newlinechar has the LaTeX setting.
\def\xintexpralignbegin      {}%
\def\xintexpralignend        {}%
\def\xintexpralignlinesep    {\,^}% separates "lines"
\def\xintexpralignleftsep    { }% at left of first item in a "line" (after brackets)
\def\xintexpraligninnersep    { , }% at the left of non-first items
\def\xintexpralignrightsep    { }% at right of last item in a "line" (before brackets)
\def\xintexpralignleftbracket {\[%
\def\xintexpralignrightbracket{]}%
```

In the  $\TeX$  example next using a `pmatrix` environment, `\noexpand` rather than `\protected` is used. This environment will not break across pages, contrarily to the display produced by the default `\xintthealign` configuration which uses  $\TeX$ 's `\halign`.

```
\[
\def\xintexpralignbegin      {\begin{pmatrix}}%
\def\xintexpralignend        {\end{pmatrix}}%
\def\xintexpralignlinesep    {\noexpand\\}% needed to counteract an internal \expanded
```

<sup>7</sup> With the `xetex` engine this will need its `-8bit` option else the `^` in `\xintexpralignlinesep` will be printed literally instead of being converted into a line separator in the file or terminal output.

```
\def\xintexpraligninnersep    {&}%
\let\xintexpralignleftbracket\empty \let\xintexpralignleftsep\empty
\let\xintexpralignrightbracket\empty \let\xintexpralignrightsep\empty
% by default amsmath matrices can have 10 columns at most
% (cf amsmath documentation for what to do to allow more)
l.c.m.=\xintthealign\xintiexpr ndmap(lcm, 1..12; 1..10)\relax
\]
```

$$l.c.m. = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 2 & 6 & 4 & 10 & 6 & 14 & 8 & 18 & 10 \\ 3 & 6 & 3 & 12 & 15 & 6 & 21 & 24 & 9 & 30 \\ 4 & 4 & 12 & 4 & 20 & 12 & 28 & 8 & 36 & 20 \\ 5 & 10 & 15 & 20 & 5 & 30 & 35 & 40 & 45 & 10 \\ 6 & 6 & 6 & 12 & 30 & 6 & 42 & 24 & 18 & 30 \\ 7 & 14 & 21 & 28 & 35 & 42 & 7 & 56 & 63 & 70 \\ 8 & 8 & 24 & 8 & 40 & 24 & 56 & 8 & 72 & 40 \\ 9 & 18 & 9 & 36 & 45 & 18 & 63 & 72 & 9 & 90 \\ 10 & 10 & 30 & 20 & 10 & 30 & 70 & 40 & 90 & 10 \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 & 110 \\ 12 & 12 & 12 & 12 & 60 & 12 & 84 & 24 & 36 & 60 \end{pmatrix}$$

## 2.3. Built-in operators and their precedences

The parser implements precedence rules based on concepts which are summarized below (only for binary infix operators):

- an infix operator has two associated precedence levels, say **L** for left and **R** for right,
- the parser proceeds from left to right, pausing each time it has found a new number and an operator following it,
- the parser compares the left-precedence **L** of the new found operator to the right-precedence **R<sub>last</sub>** of the last delayed operation (which already has one argument and would like to know if it can use the new found one): if **L** is at most equal to it, the delayed operation is now executed, else the new-found operation is kept around to be executed first, once it will have gathered its arguments, of which only one is known at this stage.

This means for example in the case of the multiplication **\*** and the division operators **/**, **//**, **/:** that they are parsed in a left-associative way because they all share the same (left and right) precedence level. This is the case with the analogous operators from the Python language, as well.

At 1.4g the power operators were changed to act in a right associative way. Again, this matches the behaviour of e.g. Python:

```
\xinteval{2^3^4}
1/2417851639229258349412352
```

The entries of [Table 1](#) are hyperlinked to the more detailed discussion at each level. In these entries the number within parentheses indicates the right-precedence, if it differs from the left.

∞ At this highest level of precedence, one finds:

**functions and variables** Functions (even the logic functions **!**() and **?**() whose names consist of a single non-letter character) must be used with parentheses. These parentheses may arise from expansion after the function name is parsed (there are exceptions which are documented at the relevant locations.)

- \* Python-like “unpacking” prefix operator. Sometimes one needs to use it as function **\***() (but I can't find an example right now) but most of the time parentheses are unneeded.

$\infty$ : at this top level the syntax elements whose execution is done prior to operators preceding them: <ul style="list-style-type: none"> <li>• <a href="#">built-in</a> or <a href="#">user-defined</a> functions,</li> <li>• <a href="#">variables</a>,</li> <li>• the <code>*</code> unpacking operator,</li> <li>• and intrinsic constituents of numbers: decimal mark <code>.</code>, <code>e</code> and <code>E</code> of scientific notation, hexadecimal prefix <code>0x</code></li> </ul>	
Precedence	``Operators'' at this level
20	postfix <code>!</code> and branching <code>?</code> , <code>??</code> operators
-	minus sign as unary operator inherits the right-precedence of the infix operator it follows, if that precedence is higher than the one of binary <code>+</code> and <code>-</code> , else it inherits the latter
18 (17)	<code>^</code> and <code>**</code> are synonymous; they act in a right-associative way ( <b>Changed at 1.4g!</b> )
16 (14)	<a href="#">Tacit multiplication</a> has an elevated (left) precedence
14	<code>*</code> , <code>/</code> , <code>//</code> (floored division), and <code>/:</code> (associated modulo, alias <code>'mod'</code> )
12	<code>+</code> , <code>-</code>
10	<code>&lt;</code> , <code>&gt;</code> , <code>==</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>!=</code> (they can be chained)
8	Boolean conjunction <code>&amp;&amp;</code> and its alias <code>'and'</code>
6	Boolean disjunction <code>  </code> and its alias <code>'or'</code> . Also <code>'xor'</code> and <code>..</code> , <code>..[</code> , <code>]..</code> , and <code>:</code> have this precedence
4	the brackets for slicers and extractors <code>[</code> , <code>]</code>
3	the comma <code>,</code>
2	the bracketers <code>[</code> , <code>]</code> construct nestable "arrays"
1	the parentheses <code>(</code> , <code>)</code> , and the semi-colon <code>;</code> in <code>iter()</code> , <code>rseq()</code> , and further structures
<ul style="list-style-type: none"> <li>• Binary operators have a left and a right precedence, which for most coincide. The right precedence is indicated within parentheses.</li> <li>• <a href="#">Tacit multiplication</a> has an elevated left precedence level: <code>(1+2)/(3+4)5</code> is computed as <code>(1+2)/((3+4)*5)</code> and <code>x/2y</code> is interpreted as <code>x/(2*y)</code> when using variables.</li> </ul>	

Table 1: Precedence levels



- . is decimal mark; the number scanner treats it as an inherent, optional and unique component of a being formed number. `\xintexpr 0.^2+2^.0\relax` is interpreted as  $0^2+2^0$  and thus produces 1.

Since release 1.2 an isolated decimal mark is illegal input in the `xintexpr` parsers (it remains legal as argument to the macros of `xintfrac`).

e scientific notation.

E scientific notation. For output, see `\xintPFloatE`.

" prefix for hexadecimal input. Only uppercase letters, and one optional . separating integer and fractional hexadecimal parts.

```
\xintexpr "FEDCBA9876543210\relax\newline
\xintexpr ".FEDCBA9876543210\relax\newline
\xintexpr 16^5-("F75DE.0A8B9+"8A21.F5746+16^5)\relax
18364758544493064720
0.99555555555555555555555555555559410496613281793543137609958648681640625
0
```

It is possible that in future the " prefix could be dropped in favour of `0x` prefix. This would free " to be used for input of "string"-like entities.

20 The postfix operators ! and the branching conditionals ?, ??.

! computes the factorial of an integer.

? is used as `(stuff)?{yes}{no}`. It evaluates `stuff` and chooses the `yes` branch if the result is non-zero, else it executes `no`. After evaluation of `stuff` it acts as a macro with two mandatory arguments within braces, chooses the correct branch *without evaluating the wrong one*. Once the braces are removed, the parser scans and expands the uncovered material.

?? is used as `(stuff)??{<0}{=0}{>0}`, where `stuff` is anything, its sign is evaluated and depending on the sign the correct branch is un-braced, the two others are discarded with no evaluation of their contents.

- As unary operator, the minus sign inherits as precedence the minimum of 12 (which is the precedence for addition and subtraction) and of the (right-) precedence of the operators preceding it (if any).

```
\xintexpr -3-4*-5^7, (-3)-(4*(-(5^(-7))))\relax\newline
\xintexpr -3^4*-5-7, (-((3^(-4))*(-5)))-7\relax\newline
|2^-10| gives \xintexpr 2^-10\relax\space
-234371/78125, -234371/78125
-562/81, -562/81
2^-10 gives 1/1024 and is thus perfectly legal, no need for parentheses.
```

The + character as prefix unary operator is simply ignored during input parsing.

18

^

\*\* Both compute powers. They act in a right associative way.

```
\xintiexpr 2^3^4\relax
2417851639229258349412352
```

16 see [Tacit multiplication](#).

14

\* multiplication



/ division:

- in `\xinteval`: exact division in the field of rational numbers (not automatically reduced to lowest terms),
- in `\xintfloateval`: correct rounding of the exact division; the two operands are, if necessary, float-rounded before the fraction is evaluated and rounded (to obtain the correctly rounded  $A/B$  without prior rounding of  $A$  and  $B$  see `qfloat()`),
- in `\xintiieval`: for compatibility with the legacy behaviour of `/` in `\numexpr`, it rounds the exact fraction with half-integers going towards the infinity of the same sign.

The division is left-associative. Example:

```
\xintexpr reduce(100/50/2)\relax
1
```

// floored division (and thus produces an integer, see `divmod()` for details)

/: the associated modulo (see `divmod()` and `mod()`)

Left-associativity applies to the division operators:

```
\xintexpr 100000/:13, 100000 'mod' 13\relax, \xintexpr 100000/:13/13\relax
4, 4, 4/13
```

Nothing special needs to be done in contexts such as `\TeX \ExplSyntaxOn` where `:` is of cat-code letter, but if `:` is an active character one needs to use input such as `/\string :` (or replace it with usage of the function `mod()`).

Bulky workarounds such as `/\string :` are unneded if activation is due to Babel. See also `\xintexprSafeCatcodes` and a framed note found in [subsection 3.1](#).

'mod' is same as `/:`.

Attention: with `polexpr` loaded, which allows `'` in variable and function names, `'mod'` can not follow a variable name. Add parentheses around the variable, or use `/:`.

12

+ addition

- subtraction. According to the general left-associativity rule in case of equal precedence, it is left associative:

```
\xintiexpr 100-50-2\relax
48
```

10 Comparison operators are (as in Python) all at the same level of precedence, use parentheses for disambiguation.

`< a<b` evaluates to 1 if the strict inequality holds to 0 if not.

`> a>b` evaluates to 1 if the strict inequality holds to 0 if not.

`== a==b` evaluates to 1 if equality holds to 0 if not.

`<= a<=b` evaluates to 1 if left hand side is at most equal to right hand side, to 0 if not.

`>= a>=b` evaluates to 1 if left hand side is at least equal to right hand side, to 0 if not.

`!= a!=b` evaluates to 1 if they differ, to 0 if not.

Comparisons can be chained arbitrarily, e.g., `x < y <= z != t` is equivalent to `x < y 'and' y <= z 'and' z != t` (and also to `all(x<y, y<=z, z!=t)`), except that if `y` and `z` involve computations, they are evaluated only once. Currently there is no short-circuit here, i.e. even if some intermediate comparison turns out false (in fact 0), all the remaining conditionals will still be evaluated.

New with  
1.4n



```
\xintifboolexpr{1<=2!=3<4>1}{true}{\error},
\xintifboolexpr{1<=2>=3<4>1}{\error}{false},
\xintifboolexpr{3 != 3! == 6 != 4! == 24}{true}{\error}
```

true, false, true

8

**&&** logical conjunction. Evaluates to **1** if both sides are non-zero, to **0** if not.

'and' same as **&&**. See also the [all\(\)](#) multi-arguments function.



**Attention:** with [polexpr](#) loaded, which allows ' in variable and function names, 'and' can not follow a variable name. Add parentheses around the variable, or use **&&**.

6

**||** logical (inclusive) disjunction. Evaluates to **1** if one or both sides are non-zero, to **0** if not.

'or' same as **||**. See also the [any\(\)](#) multi-arguments function.



**Attention:** with [polexpr](#) loaded, which allows ' in variable and function names, 'or' can not follow a variable name. Add parentheses around the variable, or use **||**.

'xor' logical (exclusive) disjunction.



**Attention:** with [polexpr](#) loaded, which allows ' in variable and function names, 'xor' can not follow a variable name. Add parentheses around the variable, or use the [xor\(\)](#) function syntax.

..

..[

].. Syntax for arithmetic progressions. See [subsection 2.5](#).

: This is a separator involved in **[a:b]** Python-like slicing syntax.

4

[

] Involved in Python-like slicing **[a:b]** and extracting **[N]** syntax. And its extension à la NumPy **[a:b,N,c:d,...,:]**. Ellipsis **...** is not yet implemented. The "step" parameter as in **[a:b:step]** is not yet implemented.

3

, The comma separates expressions (or function arguments).<sup>8</sup>

```
\xintiexpr 2^3,3^4,5^6\relax
```

8, 81, 15625

2

[

] The bracketers construct nestable "array-like" structures. Arbitrary (heterogeneous) nesting is allowed. For output related matters see [\xintthealign](#) (its usage is optional, without it rendering is "one-dimensional"). Output shape of non-homogeneous arrays is to be considered unstable at this time.

1

(

<sup>8</sup> The comma is really like a binary operator, which may be called "join". It has lowest precedence of all (apart the parentheses) because when it is encountered all postponed operations are executed in order to finalize its *first* operand; only a new comma or a closing parenthesis or the end of the expression will finalize its *second* operand.

- ) The parentheses serve as mandatory part of the syntax for functions, and to disambiguate precedences.<sup>9</sup> They do not construct any nested structure.
- ; The semi-colon as involved as part of the syntax of [iter\(\)](#), [rseq\(\)](#), [ndseq\(\)](#), [ndmap\(\)](#) has the same precedence as a closing parenthesis.

**\relax** This is the expression terminator for [\xintexpr](#) et al. It may arise from expansion during the parsing itself. As alternative to [\xintexpr](#) (et al.) use [\xinteval](#) (et al.) which have the usual macro interface (with one mandatory argument).

The **;** also serves as syntax terminator for [\xintdefvar](#) and [\xintdeffunc](#). It can in this rôle not arise from expansion as the expression body up to it is fetched by a delimited macro. But this is done in a way which does not require any specific hiding for inner semi-colons as involved in the syntax of [iter\(\)](#), etc...

## 2.4. Built-in functions

See [Table 2](#) whose elements are hyperlinked to the corresponding definitions.

Functions are at the same top level of priority. All functions even [?\(\)](#) and [!\(\)](#) require parentheses around their arguments.

<a href="#">!()</a>	<a href="#">atan2()</a>	<a href="#">first()</a>	<a href="#">iter()</a>	<a href="#">num()</a>	<a href="#">rbit()</a>	<a href="#">subs()</a>
<a href="#">?()</a>	<a href="#">atan2d()</a>	<a href="#">flat()</a>	<a href="#">iterr()</a>	<a href="#">nuple()</a>	<a href="#">reduce()</a>	<a href="#">subsm()</a>
<a href="#">*'</a>	<a href="#">binomial()</a>	<a href="#">float()</a>	<a href="#">inv()</a>	<a href="#">odd()</a>	<a href="#">reversed()</a>	<a href="#">subsn()</a>
<a href="#">+'()</a>	<a href="#">bool()</a>	<a href="#">float_dgt()</a>	<a href="#">last()</a>	<a href="#">pArg()</a>	<a href="#">round()</a>	<a href="#">tan()</a>
<a href="#">abs()</a>	<a href="#">ceil()</a>	<a href="#">floor()</a>	<a href="#">lcm()</a>	<a href="#">pArgd()</a>	<a href="#">rrseq()</a>	<a href="#">tand()</a>
<a href="#">add()</a>	<a href="#">cos()</a>	<a href="#">frac()</a>	<a href="#">len()</a>	<a href="#">pfactorial()</a>	<a href="#">rseq()</a>	<a href="#">tg()</a>
<a href="#">all()</a>	<a href="#">cosd()</a>	<a href="#">gcd()</a>	<a href="#">log()</a>	<a href="#">pow()</a>	<a href="#">sec()</a>	<a href="#">togl()</a>
<a href="#">any()</a>	<a href="#">cot()</a>	<a href="#">if()</a>	<a href="#">log10()</a>	<a href="#">pow10()</a>	<a href="#">secd()</a>	<a href="#">trunc()</a>
<a href="#">acos()</a>	<a href="#">cotd()</a>	<a href="#">ifint()</a>	<a href="#">max()</a>	<a href="#">preduce()</a>	<a href="#">seq()</a>	<a href="#">unpack()</a>
<a href="#">acosd()</a>	<a href="#">cotg()</a>	<a href="#">ifone()</a>	<a href="#">min()</a>	<a href="#">qfloat()</a>	<a href="#">sgn()</a>	<a href="#">xor()</a>
<a href="#">Arg()</a>	<a href="#">csc()</a>	<a href="#">ifsgn()</a>	<a href="#">mod()</a>	<a href="#">qfrac()</a>	<a href="#">sin()</a>	<a href="#">zip()</a>
<a href="#">Argd()</a>	<a href="#">cscd()</a>	<a href="#">ilog10()</a>	<a href="#">mul()</a>	<a href="#">qint()</a>	<a href="#">sinc()</a>	
<a href="#">asin()</a>	<a href="#">divmod()</a>	<a href="#">iquo()</a>	<a href="#">ndmap()</a>	<a href="#">qrand()</a>	<a href="#">sind()</a>	
<a href="#">asind()</a>	<a href="#">even()</a>	<a href="#">irem()</a>	<a href="#">ndseq()</a>	<a href="#">qraw()</a>	<a href="#">sqr()</a>	
<a href="#">atan()</a>	<a href="#">exp()</a>	<a href="#">isint()</a>	<a href="#">ndfillraw()</a>	<a href="#">random()</a>	<a href="#">sqrt()</a>	
<a href="#">atand()</a>	<a href="#">factorial()</a>	<a href="#">isone()</a>	<a href="#">not()</a>	<a href="#">randrange()</a>	<a href="#">sqrtr()</a>	

Table 2: Functions (click on names)

.4.1	Functions with no argument	20
.4.2	Functions with one argument	21
.4.3	Functions with an alphanumeric argument	24
.4.4	Functions with one mandatory and a second but optional argument	25
.4.5	Functions with two arguments	26
.4.6	Functions with 3 or 4 arguments	28
.4.7	Functions with an arbitrary number of arguments	29
.4.8	Functions requiring dummy variables	31

Miscellaneous notes:

- since release **1.3d** [gcd\(\)](#) and [lcm\(\)](#) are extended to apply to fractions too, and do NOT require the loading of [xintgcd](#),

<sup>9</sup> It is not apt to describe the opening parenthesis as an operator, but the closing parenthesis is analogous to a postfix unary operator. It has lowest precedence which means that when it is encountered all postponed operations are executed to finalize its operand. The start of this operand was decided by the opening parenthesis.



- The randomness related functions `random()`, `grand()` and `randrange()` require that the  $\TeX$  engine provides the `\uniformdeviate` or `\pdfuniformdeviate` primitive. This is currently the case for `pdf $\TeX$` , `(u)pt $\TeX$` , `lua $\TeX$` , and also for `xet $\TeX$`  since  $\TeX$ Live 2019.
- `togl()` is provided for the case `etoolbox` package is loaded,
- `bool()`, `togl()` use delimited macros to fetch their argument and the closing parenthesis must be explicit, it can not arise from on the spot expansion. The same holds for `qint()`, `qfrac()`, `qfloat()`, `qraw()`, `random()` and `grand()`.
- Also [functions with dummy variables](#) use delimited macros for some tasks. See the relevant explanations there.
- Functions may be called with `oples` as arguments as long as the total length is the number of arguments the function expects.

#### 2.4.1. Functions with no argument

`random()` returns a random float  $x$  verifying  $0 \leq x < 1$ . It obeys the prevailing precision as set by `\xintDigits`: i.e. with  $P$  being the precision the random float multiplied by  $10^P$  is an integer, uniformly distributed in the  $0..10^P-1$  range.

This description implies that if  $x$  turns out to be  $< 0.1$  then its (normalized) mantissa has  $P-1$  digits and a trailing zero, if  $x < 0.01$  it has  $P-2$  digits and two trailing zeros, etc... This is what is observed also with Python's `random()`, of course with 10 replaced there by radix 2.

```
\pdfsetrandomseed 12345
\xintDigits:=37\relax
\xintthefloatexpr random()\relax\newline
\xintthefloatexpr random()\relax\par
0.2415544817596207455547929850209500042
0.2584863529993996627285461554203021352
```

`grand()` returns a random float  $0 \leq x < 1$  using 16 digits of precision (i.e.  $10^{16}x$  is an integer). This is provided when speed is a at premium as it is optimized for precision being precisely 16.

```
% still with 37 digits as prevailing float precision
\xintthefloatexpr grand(), random()\relax\newline
\xintDigits:=16\relax
\xintthefloatexpr grand(), random()\relax\par
0.4883568991327765, 0.09165461826072383107532471669335645230
0.9069127435402274, 0.9106687541716861
```

One can use both `grand()` and `random()` inside the `\xintexpr` parser too. But inside the integer only `\xintiexpr` parser they will cause some low-level error as soon as they get involved in any kind of computation as they use an internal format not recognized by the integer-only parser.

See further `randrange()`, which generates random integers.

Currently there is no `uniform()` function<sup>10</sup> but it can be created by user:

```
\xintdeffloatfunc uniform(a, b):= a + (b-a)*random();
\romannumeral\xintreplicate{10}%
{%
\xintthefloatexpr uniform(123.45678, 123.45679)\relax\newline
}%
```

<sup>10</sup> Because I am not sure how to handle rounding issues: should the computation proceed exactly and a rounding be done only at very end?

**rbit()** returns a random 0 or 1.

**num(x)** truncates to the nearest integer (truncation towards zero). It has the same sign as **x**, except of course with  $-1 < x < 1$  as then **num(x)** is zero.

**frac(x)** fractional part. For all numbers  $x = \text{num}(x) + \text{frac}(x)$ , and  $\text{frac}(x)$  has the same sign as  $x$  except when  $x$  is an integer, as then  $\text{frac}(x)$  vanishes.

**reduce(x)** reduces a fraction to smallest terms

Recall that this is NOT done automatically, for example when adding fractions.

**abs(x)** absolute value

**inv(x)** inverse.

**floor(x)** floor function.

**ceil(x)** ceil function.

**sqr(x)** square.

**ilog10(x)** in `\xintiexpr` the integer exponent  $a$  such that  $10^a \leq \text{abs}(x) < 10^{a+1}$ ; returns (this may evolve in future) `-2147450880` if  $x$  vanishes (i.e. `0x7fff8000`).

```
\xintiieval{ilog10(1), ilog10(-1234567), ilog10(-123456789123456789), ilog10(2**31)}\par
0, 6, 17, 9
```

See `ilog10()` for the behaviour in `\xintexpr`-essions.

**sqrt(x)** in `\xintiexpr`, truncated square root; in `\xintexpr` or `\xintfloatexpr` this is the floating point square root, and there is an optional second argument for the precision. See `sqrt()`.

**sqrtr(x)** available only in `\xintiexpr`, rounded square root.

**factorial(x)** factorial function (like the post-fix `!` operator.) When used in `\xintexpr` or `\xintfloatexpr` there is an optional second argument. See `factorial()`.

**?(x)** is the truth value, 1 if non zero, 0 if zero. Must use parentheses.

**!(x)** is logical not, 0 if non zero, 1 if zero. Must use parentheses.

**not(x)** logical not.

**even(x)** is the evenness of the truncation `num(x)`.

```
\xintthefloatexpr [3] seq((x,even(x)), x=-5/2..[1/3]..+5/2)\relax
-2.50, 1, -2.17, 1, -1.83, 0, -1.50, 0, -1.17, 0, -0.833, 1, -0.500, 1, -0.167, 1, 0.167, 1,
0.500, 1, 0.833, 1, 1.17, 0, 1.50, 0, 1.83, 0, 2.17, 1, 2.50, 1
```

**odd(x)** is the oddness of the truncation `num(x)`.

```
\xintthefloatexpr [3] seq((x,odd(x)), x=-5/2..[1/3]..+5/2)\relax
-2.50, 0, -2.17, 0, -1.83, 1, -1.50, 1, -1.17, 1, -0.833, 0, -0.500, 0, -0.167, 0, 0.167, 0,
0.500, 0, 0.833, 0, 1.17, 1, 1.50, 1, 1.83, 1, 2.17, 0, 2.50, 0
```

**isint(x)** evaluates to 1 if  $x$  is an integer, to 0 if not. See `ifint()`.

```
$\xinttheexpr -5/3..[1/3]..+5/3\relax
\rightarrow \xinttheexpr seq(isint(x), x=-5/3..[1/3]..+5/3)\relax$
-5/3, -4/3, -3/3, -2/3, -1/3, 0, 1/3, 2/3, 3/3, 4/3, 5/3 → 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0
```

**isone(x)** evaluates to 1 if  $x$  is 1, to 0 if not. See `ifone()`.

```
\xintthefloatexpr subs(((x-1)/x, x/x, (x+1)/x), x=2**30)\relax
\rightarrow
\xintthefloatexpr seq(isone(y), y=subs(((x-1)/x, x/x, (x+1)/x), x=2**30))\relax$
0.9999999990686774, 1, 1.000000000931323 → 0, 1, 0
```

**qint(x)** belongs with `qfrac()`, `qfloat()`, `qraw()` to a special category:

1. They require the closing parenthesis of their argument to be immediately visible, it can not arise from expansion.
2. They grab the argument and store it directly; the format must be compatible with what is expected at macro level.
3. And in particular the argument can not be a variable, it has to be numerical.

`qint()` achieves the same result as `num`, but the argument is grabbed as a whole without expansion and handed over to the `\xintiNum` macro. The `q` stands for ``quick'', and `qint` is thought out for use in `\xintiexpr...\relax` with integers having dozens of digits.

Testing showed that using `qint()` starts getting advantageous for inputs having more (or *expanding* to more) than circa 20 explicit digits. But for hundreds of digits the input gain becomes a negligible proportion of (for example) the cost of a multiplication.

Leading signs and then zeroes will be handled appropriately but spaces will not be systematically stripped. They should cause no harm and will be removed as soon as the number is used with one of the basic operators. This input mode *does not accept decimal part or scientific part*.

```
\def\x{...many many many ... digits}\def\y{...also many many many digits...}
\xinttheiexpr qint(\x)*qint(\y)+qint(\y)^2\relax\par
```

`qfrac(x)` does the same as `qint` except that it accepts fractions, decimal numbers, scientific numbers as they are understood by the macros of package `xintfrac`. Thus, it is for use in `\xintexpr...\relax`. It is not usable within an `\xintiexpr`-ession, except if hidden inside functions such as `round` or `trunc` which then produce integers acceptable to the integer-only parser. It has nothing to do with `frac` (sigh...).

`qfloat(x)` does the same as `qfrac` and then converts to a float with the precision given by the setting of `\xintDigits`. This can be used in `\xintexpr` to round a fraction as a float with the same result as with the `float()` function (whereas using `\xintfloatexpr A/B\relax` inside `\xintexpr...\relax` would first round `A` and `B` to the target precision); or it can be used inside `\xintfloatexpr...\relax` as a faster alternative to wrapping the fraction in a sub-`\xintexpr`-ession. For example, the next two computations done with 16 digits of precision do not give the same result:

```
\xintthefloatexpr qfloat(12345678123456785001/12345678123456784999)-0.5\relax\newline
\xintthefloatexpr 12345678123456785001/12345678123456784999-0.5\relax\newline
\xintthefloatexpr 1234567812345679/1234567812345678-0.5\relax\newline
\xintthefloatexpr \xintexpr12345678123456785001/12345678123456784999\relax-0.5\newline
```

```
0.5
0.50000000000000010
0.50000000000000010
0.5
```

because the second is equivalent to the third, whereas the first one is equivalent to the fourth one. Equivalently one can use `qfrac` to the same effect (the subtraction provoking the rounding of its two arguments before further processing.)

Note that if the input needs no special rounding, the internal form of the output keeps a short mantissa (it does not add padding zeros to make it of length equal to the float precision). For example `qfloat(2[20])` would keep internally the input format.

`float_dgt(x)` is like `float()` and avoids `float()`'s check whether it used with its second optional argument. This is useful in the context of converting function definitions done via `\xintdeffunc` (see explanations there) to functions usable in `\xintfloateval`.

**Do not use! (1.4)** `nuple(x)` is currently same as `[...]`. Reserved for possible alternative meaning in future.

```
\xinteval{nuple(1,2,3)}
[1, 2, 3]
```

`unpack(x)` is alternative for `*` unpacking operator.

```
\xinteval{unpack([1,2,3])}
```

1, 2, 3

**flat(ople)** removes all nesting to produce a (non-bracketed) ople having the same leaves (some possibly empty) but located at depth 1.

```
\xinteval{flat([[[[1,[],3],[4,[[[5,6,[]],[8,9],[[],11]],12],[13,14]]], [[[],16]]], [])}
1, [], 3, 4, 5, 6, [], 8, 9, [], 11, 12, 13, 14, [], 16, []
```

unstable?

I almost delayed indefinitely release because I was hesitating on the name: perhaps better with **flattened()**, but long names add (negligible, but still) overhead compared to short names. For this reason, consider that name may change.

### 2.4.3. Functions with an alphanumeric argument

**bool(name)** returns 1 if the  $\TeX$  conditional `\ifname` would act as `\iftrue` and 0 otherwise. This works with conditionals defined by `\newif` (in  $\TeX$  or  $\LaTeX$ ) or with primitive conditionals such as `\ifmmode`. For example:

```
\xintifboolexpr{25*4-if(bool(mmode),100,75)}{YES}{NO}
```

will return NO if executed in math mode (the computation is then  $100-100=0$ ) and YES if not (the `if()` conditional is described below; the `\xintifboolexpr` test automatically encapsulates its first argument in an `\xintexpr` and follows the first branch if the result is non-zero (see [subsection 3.14](#))).

The alternative syntax `25*4-\ifmmode100\else75\fi` could have been used here, the usefulness of **bool(name)** lies in the availability in the `\xintexpr` syntax of the logic operators of conjunction `&&`, inclusive disjunction `||`, negation `!` (or `not`), of the multi-operands functions `all`, `any`, `xor`, of the two branching operators `if` and `ifsgn` (see also `?` and `??`), which allow arbitrarily complicated combinations of various **bool(name)**.

**togl(name)** returns 1 if the  $\LaTeX$  package `etoolbox`<sup>11</sup> has been used to define a toggle named **name**, and this toggle is currently set to `true`. Using `togl` in an `\xintexpr... \relax` without having loaded `etoolbox` will result in an error from `\iftoggle` being a non-defined macro. If `etoolbox` is loaded but `togl` is used on a name not recognized by `etoolbox` the error message will be of the type `ERROR: Missing \endcsname inserted.`, with further information saying that `\protect` should have not been encountered (this `\protect` comes from the expansion of the non-expandable `etoolbox` error message).

When **bool** or **togl** is encountered by the `\xintexpr` parser, the argument enclosed in a parenthesis pair is expanded as usual from left to right, token by token, until the closing parenthesis is found, but everything is taken literally, no computations are performed. For example `togl(2+3)` will test the value of a toggle declared to `etoolbox` with name `2+3`, and not `5`. Spaces are gobbled in this process. It is impossible to use `togl` on such names containing spaces, but `\iftoggle{name with spaces}{1}{0}` will work, naturally, as its expansion will pre-empt the `\xintexpr` scanner.

There isn't in `\xintexpr...` a `test` function available analogous to the `test{\ifsometest}` construct from the `etoolbox` package; but any expandable `\ifsometest` can be inserted directly in an `\xintexpr`-ession as `\ifsometest10` (or `\ifsometest{1}{0}`), for example `if(\ifsometest{1}{0},YES,NO)` (see the `if` operator below) works.

A straight `\ifsometest{YES}{NO}` would do the same more efficiently, the point of `\ifsometest10` is to allow arbitrary boolean combinations using the (described later) `&&` and `||` logic operators: `\ifsometest10 && \ifsomeothertest10 || \ifsomeotherthertest10`, etc... YES or NO above stand for material compatible with the `\xintexpr` parser syntax.

See also `\xintifboolexpr`, in this context.

<sup>11</sup> <https://ctan.org/pkg/etoolbox>



#### 2.4.4. Functions with one mandatory and a second but optional argument

**round(x[, n])** Rounds its first argument to an integer multiple of  $10^{(-n)}$  (i.e. it quantizes). The case of negative **n** is new with 1.4a. Positive **n** corresponds to conversion to a fixed point number with **n** digits after decimal mark.

```
\xinteval{round(-2^30/3^5,12), round(-2^30/3^5,-3)}
-4418690.633744855967, -4419e3
```

**trunc(x[, n])** Truncates its first argument to an integer multiple of  $10^{(-n)}$ . The case of negative **n** is new with 1.4a.

```
\xinteval{trunc(-2^30/3^5,12), trunc(-2^30/3^5,-3)}
-4418690.633744855967, -4418e3
```

**float(x[, n])** Rounds its first argument to a floating point number, with a precision given by the second argument, which must be positive.

```
\xinteval{float(-2^30/3^5,12), float(-2^30/3^5, 1)}
-4.41869063374e6, -4e6
```

For this example and earlier ones if the parser had been `\xintfloateval`, not `\xinteval`, the first argument (here  $2^{30}/3^5$ ) would already have been computed as floating point number with numerator and denominator rounded separately first to the prevailing precision. To avoid that, use `\xintexpr...\relax` wrapper. Then the rounding or truncation will be applied to an exact fraction.

**sfloat(x[, n])** It is the same as `float()`, but in case of a short (non-fractional) input it gets stored internally without adding zeros to make the mantissa have the `\xinttheDigits` length. One may wonder then what is the utility of `sfloat()`? See for an example of use the documentation of `\xintdeffunc`. Notice however that this is a bit experimental and may evolve in future when `xint` gets a proper internal data structure for floating point numbers. The non-normalized format is useful for multiplication or division, but float additions and subtractions usually convert their arguments to a normalized mantissa.

**ilog10(x[, n])** If there is an optional argument **n**, returns the (relative) integer **a** such that  $10^a \leq \text{abs}(\text{float}(x, n)) < 10^{a+1}$ . In absence of the optional argument:

- in `\xintexpr`, it returns the exponent **a** such that  $10^a \leq \text{abs}(x) < 10^{a+1}$ .
- in `\xintfloatexpr`, the input is first rounded to `\xinttheDigits` float precision, then the exponent **a** is evaluated.

```
\xintfloateval{ilog10(99999999/10000000, 8), ilog10(-99999999/10000000, 8),
ilog10(-99999999/10000000, 9)}\newline
\xinteval{ilog10(-99999999/10000000), ilog10(-99999999/10000000, 8)}
0, 1, 0
0, 1
```

If the input vanishes the function outputs `-2147450880` (i.e. `-0x7fff8000` which is near the minimal TeX number `-0x7fffffff`). This is also subject to change.

The **integer-only** variant for `\xintiexpr` admits no optional argument.

**sqrt(x[, n])** in `\xintexpr...\relax` and `\xintfloatexpr...\relax` it achieves the precision given by the optional second argument. For legacy reasons the `sqrt` function in `\xintiexpr` truncates (to an integer), whereas `sqrt` in `\xintfloatexpr...\relax` (and in `\xintexpr...\relax` which borrows it) rounds (in the sense of floating numbers). There is `sqrtr` in `\xintiexpr` for rounding to nearest integer.

```
\xinttheexpr sqrt(2,31)\relax\ and \xinttheiexpr sqrt(num(2e60))\relax
```

1.41421356237309504880168872421 and 1414213562373095048801688724209

There is an [integer only](#) variant for `\xintiiexpr`.

**factorial(x[, n])** when the second optional argument is made use of inside `\xintexpr...\relax`, this switches to the use of the float version, rather than the exact one.

```
\xinttheexpr factorial (100,32)\relax, {\xintDigits:=32\relax \xintthefloatexpr
factorial (100)\relax}\newline
\xinttheexpr factorial (50)\relax\newline
\xinttheexpr factorial (50, 32)\relax
9.3326215443944152681699238856267e157, 9.3326215443944152681699238856267e157
30414093201713378043612608166064768844377641568960512000000000000
3.0414093201713378043612608166065e64
```

The [integer only variant](#) of course has no optional second argument.

**randrange(A[, B])** when used with a single argument **A** returns a random integer  $0 \leq x < A$ , and when used with two arguments **A** and **B** returns a random integer  $A \leq x < B$ . As in Python it is an “empty range” error in first case if **A** is zero or negative and in second case if **B**  $\leq$  **A**.

Attention that the arguments are first converted to integers using `\xintNum` (i.e. truncated towards zero).

The function can be used in all three parsers. Of course the size is not limited (but in the float parser, the integer will be rounded if involved in any operation).

```
\pdfsetrandomseed 12345
\xinttheiexpr randrange(10**20)\relax\newline
\xinttheiexpr randrange(1234*10**16, 1235*10**16)\relax\newline
\printnumber{\xinttheiexpr randrange(10**199,10**200)\relax}\par
12545314555479298502
12341249468233524155
3872427149656655225094489636677708166243633082496887337312033225820004454949709978664331
9106687541716861906912743540227448009165461826072383107532471669335645234883568991327765
395258486352999399662728
```

For the support macros see `\xintRandomDigits`, `\xintiiRandRange`, `\xintiiRandRangeAtoB`. For some details regarding how `xint` uses the engine provided generator of pseudo-random numbers, see `\xintUniformDeviate`.

## 2.4.5. Functions with two arguments

**iquo(m, n)** Only available in `\xintiiexpr/\xintiieval` context. Computes the Euclidean quotient. Matches with the remainder defined in next item. See `\xintiiQuo`.

**irem(m, n)** Only available in `\xintiiexpr/\xintiieval` context. Computes the Euclidean remainder. Attention that, following mathematical definition, it is always non-negative. See `\xint-iiRem`.

**mod(f, g)** computes  $f - g \cdot \text{floor}(f/g)$ . Hence its output is a general fraction or floating point number or integer depending on the used parser. If non-zero, it has the same sign as **g**.

Prior to 1.2p it computed  $f - g \cdot \text{trunc}(f/g)$ .

The `/:` and `'mod'` infix operators are both mapped to the same underlying macro as this `mod(f, g)` function. At 1.3 this macro produces smaller denominators when handling fractions than formerly.

```
\xinttheexpr mod(11/7,1/13), reduce(((11/7)/(1/13))*1/13+mod(11/7,1/13)),
```

```
mod(11/7,1/13)- (11/7)/(1/13), (11/7)/(1/13)\relax\newline
\xintthefloatexpr mod(11/7,1/13)\relax\par
3/91, 11/7, 0, 20
0.03296703296703260
```

Attention: the precedence rules mean that  $29/5 /: 3/5$  is handled like  $((29/5)/(3))/5$ . This is coherent with behaviour of Python language for example:

```
>>> 29/5 % 3/5, 11/3 % 17/19, 11/57
(0.5599999999999999, 0.19298245614035087, 0.19298245614035087)
>>> (29/5) % (3/5), (11/3) % (17/19), 5/57
(0.4, 0.08771929824561386, 0.08771929824561403)
```

For comparison (observe on the last lines how `\xintfloatexpr` is more accurate than Python!):

```
\noindent\xinttheexpr 29/5 /: 3/5, 11/3 /: 17/19\relax\newline
\xinttheexpr (29/5) /: (3/5), (11/3) /: (17/19)\relax\newline
\xintthefloatexpr 29/5 /: 3/5, 11/3 /: 17/19, 11/57\relax\newline
\xintthefloatexpr (29/5) /: (3/5), (11/3) /: (17/19), 5/57\relax\newline
5/57 = \xinttheexpr trunc(5/57, 20)\relax\dots\newline
14/25, 11/57
2/5, 5/57
0.56, 0.1929824561403509, 0.1929824561403509
0.4, 0.08771929824561420, 0.08771929824561404
5/57 = 0.08771929824561403508...
```

Regarding some details of behaviour in `\xintfloatexpr`, see discussion of `divmod` function next.

**divmod(f, g)** computes the two mathematical values  $\text{floor}(f/g)$  and  $\text{mod}(f,g)=f - g*\text{floor}(f/g)$  and produces them as a bracketed pair in other terms it is analogous to the Python `divmod` function. Its output is equivalent to using  $f//g$ ,  $f/:g$  but its implementation avoids doing twice the needed division.

In `\xintfloatexpr...\relax` the modulo is rounded to the prevailing precision. The quotient is like in the other parsers an exact integer. It will be rounded as soon as it is used in further operations, or via the global output routine of `\xintfloatexpr`. Those examples behave as in 1.3f because assignments to multiple variables tacitly unpack if this is necessary.

```
\xintdefvar Q, R := divmod(3.7, 1.2);%
\xinttheexpr Q, R, 1.2Q + R\relax\newline
\xintdefiivar Q, R := divmod(100, 17);%
\xinttheiexpr Q, R, 17Q + R\relax\newline
\xintdeffloatvar Q, R := divmod(100, 17e-20);%
\xintthefloatexpr Q, R, 17e-20 * Q + R\relax\newline
% show Q exactly, although defined as float it can be used in iiexpr:
\xinttheiexpr Q\relax\ (we see it has more than 16 digits)\par
\xintunassignvar{Q}\xintunassignvar{R}%
3, 0.1, 3.7
5, 15, 100
5.882352941176471e20, 9e-20, 100
5882352941176471[5] (we see it has more than 16 digits)
```

Again:  $f//g$  or the first item output by `divmod(f, g)` is an integer  $q$  which when computed inside `\xintfloatexpr...\relax` is not yet rounded to the prevailing float precision; the second item  $f-q*g$  is the rounding to float precision of the exact mathematical value evaluated with this exact  $q$ . This behaviour may change in future major release; perhaps  $q$  will be rounded and



$f-q*g$  will correspond to usage of this rounded  $q$ .

As `\xintfloatexpr` rounds its global result, or rounds operands at each arithmetic operation, it requires special circumstances to show that the  $q$  is produced unrounded. Either as in the above example or this one with comparison operators:

```
\xintDigits := 4\relax
\xintthefloatexpr if(12345678//23==537000, 1, 0), 12345678//23\relax\newline
\xintthefloatexpr if(float(12345678//23)==537000, 1, 0)\relax\par
\xintDigits := 16\relax
0, 537000
1
```

In the first line, the comparison is done with `floor(12350000/23)=536957` (notice in passing that `12345678//23` was evaluated as `12350000//23` because the operands are first rounded to 4 digits of floating point precision), hence the conditional takes the "False" branch. In the second line the `float` forces rounding of the output to 4 digits, and the conditional takes the "True" branch.

This example shows also that comparison operators in `\xintfloatexpr.\relax` act on unrounded operands.

**binomial(x, y)** computes binomial coefficients. It returns zero if  $y < 0$  or  $x < y$  and raises an error if  $x < 0$  (or if  $x > 99999999$ .)

```
\xinttheexpr seq(binomial(20, i), i=0..20)\relax
1, 20, 190, 1140, 4845, 15504, 38760, 77520, 125970, 167960, 184756, 167960, 125970, 77520,
38760, 15504, 4845, 1140, 190, 20, 1

\printnumber{\xintthefloatexpr seq(binomial(100, 50+i), i=-5..+5)\relax}%
6.144847121413618e28, 7.347099819081500e28, 8.441348728306404e28, 9.320655887504988e28, 9
.891308288780803e28, 1.008913445455642e29, 9.891308288780803e28, 9.320655887504988e28, 8.
441348728306404e28, 7.347099819081500e28, 6.144847121413618e28
```

The arguments must be (expand to) short integers.

**pfactorial(a, b)** computes partial factorials i.e. `pfactorial(a,b)` evaluates the product  $(a+1) \dots b$ .

```
\xinttheexpr seq(pfactorial(20, i), i=20..30)\relax
1, 21, 462, 10626, 255024, 6375600, 165765600, 4475671200, 125318793600, 3634245014400,
109027350432000
```

The arguments must (expand to) short integers. See [subsection 11.36](#) for the behaviour if the arguments are negative.

**ndfillraw(TeX-macro, n-uple)** The second argument is  $[N_1, N_2, \dots, N_k]$ . The construct fills an  $N_1 \times N_2 \times \dots \times N_k$  hyperrectangular nested list by evaluating the given `macro` as many times as needed. The expansion result goes directly into internal data and must thus comply with what is expected internally for an individual numeric leaf (at 1.4, `xintfrac` raw format worked for `\xintexpr` or `\xintfloatexpr`, but not `\xintiexpr`, and this may have changed since). This is an experimental function serving to generate either constant or random arrays. Attention that **TeX-macro** stands here for any expandable TeX macro, and an `\xintexpr`-ession at this location thus requires an explicit `\xinteval` wrapping.

Do not  
use!

## 2.4.6. Functions with 3 or 4 arguments

**if(cond,yes,no)** (twofold-way conditional)

checks if **cond** is true or false and takes the corresponding branch. Any non zero number or fraction is logical true. The zero value is logical false. Both ``branches'' are evaluated (they are not really branches but just numbers). See also the [?](#) operator.

**ifint(x,yes,no)** (twofold-way conditional)

checks if **x** is an integer and in that case chooses the ``yes'' branch.

See also [isint\(\)](#).

**ifone(x,yes,no)** (twofold-way conditional)

checks if **x** is equal to one and in that case chooses the ``yes'' branch.

Slightly more efficient than **if(x==1,...)**. See also [isone\(\)](#).

**ifsgn(cond,<0,=0,>0)** (threefold-way conditional)

checks the sign of **cond** and proceeds correspondingly. All three are evaluated. See also the [??](#) operator.

#### 2.4.7. Functions with an arbitrary number of arguments

The functions [all\(\)](#), [any\(\)](#), [xor\(\)](#), [+'\(\)](#), [\\*'](#), [max\(\)](#), [min\(\)](#), [gcd\(\)](#), [lcm\(\)](#), [first\(\)](#), [last\(\)](#), [reversed\(\)](#) and [len\(\)](#) work both with "open" and "packed" lists (aka **nutples**).

Since 1.4, when used with a single argument which is a **nutple**, it is automatically unpacked. But from 1.4 to 1.4h these functions could not be used with a single numeric argument: either they had at least two arguments, or only one and it had to be a **nutple**. At 1.4i it is again possible to use them with a lone numeric argument.

In the specific case of [reversed\(\)](#) with a **nutple** argument the output is then repacked so that the output is a **nutple** if and only if the input was one (the reversal does not propagate to deeper nested **nutple**'s, it applies only at depth one).

**Do not use!** **qraw(stuff)** It injects directly tokens to represent internally numerical data. Will break at any release modifying the internal data format specifications (which are not always documented).

**all(x, y, ...)** inserts a logical **AND** in-between its arguments and evaluates the resulting logical assertion (as with all functions, all arguments are evaluated).

```
\xinteval{all(1,1,1), all([1,0,1]), all([1,1,1])}
1, 0, 1
```

**any(x, y, ...)** inserts a logical **OR** in-between its arguments and evaluates the resulting logical assertion,

```
\xinteval{any(0,0,0), any([1,0,1]), any([0,0,0])}
0, 1, 0
```

**xor(x, y, ...)** inserts a logical **XOR** in-between its arguments and evaluates the resulting logical assertion,

```
\xinteval{xor(1,1,1), xor([1,0,1]), xor([1,1,1])}
1, 0, 1
```

**`+(x, y, ...)** adds (left ticks mandatory):

```
\xinttheexpr `+(1,3,19), `+(1**2,3**2,sqr(19)), `+([1**2,3**2,sqr(19)])\relax
23, 371, 371
```

**`\*(x, y, ...)** multiplies (left ticks mandatory):

```
\xinttheexpr `*(1,3,19), `*(1^2,3^2,19^2), `*([1^2,3^2,19^2])\relax
```

57, 3249, 3249

**max(x, y, ...)** maximum of the (arbitrarily many) arguments,

```
\xinttheexpr max(1,3,19), min([1,3,19])\relax
```

19, 1

**min(x, y, ...)** minimum of the (arbitrarily many) arguments,

```
\xinttheexpr min(1,3,19), min([1,3,19])\relax
```

1, 1

**gcd(x, y, ...)** computes the positive generator of the fractional ideal of rational numbers  $x\mathbb{Z} + y\mathbb{Z} + \dots \subset \mathbb{Q}$ . Since 1.4d the output is always in lowest terms.

This example shows how to reduce an n-uple to its primitive part:

```
\xinteval{gcd(7/300, 11/150, 13/60)}\newline
$(7/300, 11/150, 13/60)\to
(\xinteval{subsn(seq(reduce(x/D), x = L), D=gcd(L); L=7/300, 11/150, 13/60))}\newline
\xintexpr gcd([7/300, 11/150, 13/60])\relax\par
```

1/300  
 (7/300, 11/150, 13/60)  $\rightarrow$  (7, 22, 65)  
 1/300

MEMO Perhaps a future release will provide a **primpart()** function as built-in functionality.

In case of strict integers, using a **\xintiexpr... \relax** wrapper is advantageous as the integer-only **gcd()** is more efficient. As **\xintiexpr** accepts only strict integers, doing this may require wrapping the argument in **num()**.

**lcm(x, y, ...)** computes the positive generator of the fractional ideal of rational numbers  $x\mathbb{Z} \cap y\mathbb{Z} \cap \dots \subset \mathbb{Q}$ .

```
\xinttheexpr lcm([7/300, 11/150, 13/60])\relax
```

1001/30 As for **gcd()**, since 1.4d the output is always in lowest terms. For strict integers it is slightly advantageous to use a sub **\xintiexpr**-ession.

**first(x, y, ...)** first item of the list or nutple argument:

```
\xintiexpr first([last(-7..3), [58, 97..105]])\relax
```

3

**last(x, y, ...)** last item of the list or nutple argument:

```
\xintiexpr last([-7..3, 58, first(97..105)])\relax
```

97

**reversed(x, y, ...)** reverses the order of the comma separated list or inside a nutple:

```
\xintieval{reversed(reversed(1..5), reversed([1..5]))}
```

[5, 4, 3, 2, 1], 1, 2, 3, 4, 5

The above is correct as **xintexpr** functions may produce oples and this is the case here.

**len(x, y, ...)** computes the number of items in a comma separated list or inside a nutple (at first level only: it is not a counter of leaves).

```
\xinttheiexpr len(37.5), len(1..50, [101..150], 1001..1050), len([1..10])\relax
```

1, 101, 10

**zip(\*nutples)** behaves similarly to the Python function of the same name: i.e. it produces an ople of nutples, where the  $i$ -th nutple contains the  $i$ -th element from each of the argument nutples. The ople ends when the shortest input nutple is exhausted. With a single nutple argument, it returns an ople of 1-nutples. With no arguments, it returns the empty ople.

As there is no exact match in [xintexpr](#) of the concept of “iterator” object,<sup>12</sup> there is a significant difference here that (for example) the `zip(x,x,x)` Python idiom to cluster the iterator `x` into successive chunks of length 3 does not apply. Consider for this reason even the name of the function as work-in-progress, susceptible to change.

unstable?

```
\xintiieval{zip([1..9], [0, 1, 2], [11..29], [111..139])}
[1, 0, 11, 111], [2, 1, 12, 112], [3, 2, 13, 113]
```

See also [\xintthespaceseparated](#) for some possible usage in combination with `flat()`.

## 2.4.8. Functions requiring dummy variables

The pseudo-functions `subs()`, `seq()`, `subsm()`, `subsn()`, `iter()`, `add()`, `mul()`, `rseq()`, `iterr()`, `rrseq()`, `ndseq()`, `ndmap()`, `ndfillraw()` use delimited macros for some tasks:

- for all of them, whenever a `<varname>=` chunk must be parsed into a (non-assigned) variable name, then the equal sign must be visible,
- and if the syntax is with `,<varname>=` the initial comma also must be visible (spaces do not matter),
- for all of them but `ndmap()` and `ndfillraw()` the final closing parenthesis must be visible.

Although delimited macros involving commas are used to locate `,<varname>=` this is done in a way silently ignoring commas located inside correctly balanced parentheses. Thus, as the examples will show, nesting works as expected.

The semi-colons involved in the syntax may arise from expansion alone. For `rseq()`, `iter()`, `rrseq()` and `iterr()` the `,<varname>=` part may also be created from the expansion which will generate the initial comma separated values delimited by a semi-colon.

Prior to 1.4, semi-colons needed to be braced or otherwise hidden when located in an expression parsed by `\xintdefvar` or `\xintdeffunc`, to not be confused with the expression terminator.

`seq()`, `rseq()`, `iter()`, `rrseq()`, `iterr()` and also `add()`, `mul()`, but not `subs()` admit the `omit`, `abort`, and `break()` keywords. This is a new feature at 1.4 for `add()` and `mul()`.

In the case of a potentially infinite list generated by the `<integer>++` syntax, use of `abort` or of `break()` is mandatory, naturally.

All lowercase and uppercase Latin letters are pre-configured for usage as dummy variables. In Unicode engines one can use `\xintnewdummy` to turn any letter into a usable dummy variable.

And since 1.4, `\xintnewdummy` works (in all engines) to turn a multi-letter word into a dummy variable. In the descriptions, `varname` stands for such a dummy variable, either single-letter or word.

**subs(expr, varname=values)** for variable substitution.

```
\xinttheexpr subs(subs(seq(x*z,x=1..10),z=y^2),y=10)\relax\newline
100, 200, 300, 400, 500, 600, 700, 800, 900, 1000
```

Attention that `xz` generates an error, one must use explicitly `x*z`, else the parser expects a variable with name `xz`.

`subs()` is useful when defining macros for which some argument will be used more than once but may itself be a complicated expression or macro, and should be evaluated only once, for matters

<sup>12</sup> Speaking of iterators, I have some ideas about this: as `\xintexpr` does not have the global expression in its hands it is difficult to organize globally expandably the idea of iterator, but locally via syntax like the one for `seq()` this is feasible. When one thinks about it, `seq()` is closely related to the iterator idea.



of efficiency. But `subs()` is helpless in function definitions: all places where a variable is substituted will receive the complete recipe to compute the variable, rather than evaluate only once.

One should rather define auxiliary functions to compute intermediate results. Or one can use `seq()`. See the documentation of `\xintdeffunc`.

**add(expr, varname=values)** addition

```
\xintiexpr add(x^3,x=1..20), add(x(x+1), x=1,3,19)\relax\newline
\xintiexpr add(x^3, x = 1..[2]..20)\relax\newline           % add only odd cubes
\xintiexpr add((odd(x))>{x^3}{omit}, x = 1..20)\relax\par % add only odd cubes
44100, 394
19900
19900
```

At 1.4 (fixed at 1.4a), the keywords `omit` (as in example above), `abort` and `break()` are allowed. The meaning of `break()` is specific: its argument serves as last operand for the addition, not as ultimate value.

```
\xintiexpr add((x>10){break(1000)}{x}, x = 1..15)\relax
1055
```

The `@` special variable holds the so-far accumulated value. Initially its value is zero.

```
\xintiexpr add(1 + @, i=1..10)\relax % iterates x <- 2x+1
1023
```

See `'+'()` for syntax simply adding items of a list without usage of a dummy variable.

**mul(expr, varname=values)** multiplication

```
\xintiexpr mul(x^2, x = 1, 3, 19, 37..50)\relax
21718466538487411085212279802172111087206400000000
```

The `@` special variable holds the so-far accumulated value. Initially its value is one.

At 1.4 (fixed at 1.4a), the keywords `omit`, `abort` and `break()` are allowed. The meaning of `break()` is specific: its argument serves as last operand for the multiplication, not as ultimate value.

```
\xintieval{mul((i==100){break(i^4)}{i}, i = 98, 99, 100)}
970200000000
```

See `'*'` for syntax without a dummy variable.

**seq(expr, varname=values)** comma separated values generated according to a formula

```
\xintiexpr seq(x(x+1)(x+2)(x+3),x=1..10), `*`(seq(3x+2,x=1..10))\relax
24, 120, 360, 840, 1680, 3024, 5040, 7920, 11880, 17160, 1162274713600

\smallskip
\leavevmode\vbox{\xintthealign\xintiexpr [seq([seq(i^2+j^2, i=0..j)], j=0..10)]\relax}
[[ 0      ],
 [ 1,    2      ],
 [ 4,    5,    8      ],
 [ 9,   10,   13,   18      ],
 [ 16,  17,  20,  25,  32      ],
 [ 25,  26,  29,  34,  41,  50      ],
 [ 36,  37,  40,  45,  52,  61,  72      ],
 [ 49,  50,  53,  58,  65,  74,  85,  98      ],
 [ 64,  65,  68,  73,  80,  89,  100, 113, 128      ],
 [ 81,  82,  85,  90,  97,  106, 117, 130, 145, 162      ],
 [ 100, 101, 104, 109, 116, 125, 136, 149, 164, 181, 200 ]]
```



**rseq(initial value; expr, varname=values)** recursive sequence, @ for the previous value.

```
\printnumber {\xintthefloatexpr subs(rseq (1; @/2+y/2@, i=1..10),y=1000)\relax }\newline
1, 500.5, 251.2490009990010, 127.6145581634591, 67.72532736082604, 41.24542607499115, 32.7
4526934448864, 31.64201586865079, 31.62278245070105, 31.62277660168434, 31.62277660168379
```

Attention: in the example above  $y/2@$  is interpreted as  $y/(2*@)$ . With versions 1.2c or earlier it would have been interpreted as  $(y/2)*@$ .

In case the initial stretch is a comma separated list, @ refers at the first iteration to the whole list. Use parentheses at each iteration to maintain this ``nuple''. For example:

```
\printnumber{\xintthefloatexpr rseq(1,10^6;
      (sqrt(@[0]*@[1]),(@[0]+@[1])/2), i=1..7)\relax }
1, 1e6, 1000, 500000.5, 22360.69095533499, 250500.25, 74842.22521066670, 136430.4704776675,
101048.3052657827, 105636.3478441671, 103316.8617608946, 103342.3265549749, 103329.593373
4841, 103329.5941579348, 103329.5937657094, 103329.5937657095
```

Prior to 1.4 the above example had to be written with `[@]`. This is still possible (@ stands for an ople with two items, bracketing then extracting is like extracting directly), but it is leaner to drop the extra “packing”.

**iter(initial value; expr, varname=values)** is exactly like `rseq`, except that it only prints the last iteration.

`iter()` is convenient to handle compactly higher order iterations. We can illustrate its use with an expandable (!) implementation of the Brent-Salamin algorithm for the computation of  $\pi$ :

```
\xintDigits:= 87\relax % we target 84 digits, and use 3 guard digits
\xintdeffloatfunc BS(a, b, t, p):= 0.5*(a+b), sqrt(a*b), t-p*sqr(a-b),
\xintiexpr 2p\relax;

\xinteval
{trunc(% I feel truncation is better than rounding to display decimals of  $\pi$ 
  \xintfloatexpr
    iter(1, sqrt(0.5), 1, 1; % initial values
% this 43 is 84/2 + 1
    (@[0]-@[1]<2e-43)?% stopping criteria; takes into account that the
                        % exit computation (break() argument) doubles
                        % number of exact digits (roughly)
    {break(sqr(@[0]+@[1])/@[2])} % ... do final computation,
    {BS(@)}, % else do iteration
    i=1++) % This generates infinite iteration. The i is not used.
  \relax
% this 83 is 84 - 1 (there is a digit known to be 3 actually, before decimal mark)
, 83)% closing parenthesis of trunc()
}%...% some dots following end of \xinteval argument
\xintDigits:=16\relax
```

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862...  
 You can try with `\xintDigits:=1004\relax` and `2e-501` in place of `\xintDigits:=87\relax` and `2e-43`, but be patient for some seconds for the result. Of course don't truncate the final result to only 83 fractional decimal digits but 1000... and better to wrap the whole thing in `\message` or `\immediate\write128` or `\edef` because it will then run in the right margin.

Prior to 1.4 the above example had to use notation such as `[@][0]`; this would still work but `@[0]` is leaner.

**rrseq(initial values; expr, varname=values)** recursive sequence with multiple initial terms. Say, there are K of them. Then `@1`, ..., `@4` and then `@@n` up to `n=K` refer to the last K values.

Notice the difference with `rseq()` for which `@` refers to a list of items in case the initial value is a list and not a single item.<sup>13</sup> Using `rrseq()` with `@1` etc... accessors may be perhaps a bit more efficient than using `rseq()` with a list as staring value and constructs such as `@[0]`, `@[1]` (or rather `@[-1]`, `@[-2]` to mimick what `@1`, `@2`, `@3`, `@4` and `@@(integer)` do in `rrseq()`.

```
\xinttheiexpr rseq(0,1; @1+@2, i=2..30)\relax
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,
17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040
```

```
\xinttheiexpr rseq(1; 2@, i=1..10)\relax
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
```

```
\xinttheiexpr rseq(1; 2@+1, i=1..10)\relax
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047
```

```
\xinttheiexpr rseq(2; @(@+1)/2, i=1..5)\relax
2, 3, 6, 21, 231, 26796
```

```
\xinttheiexpr rrseq(0,1,2,3,4,5; @1+@2+@3+@4+@@(5)+@@(6), i=1..20)\relax
0, 1, 2, 3, 4, 5, 15, 30, 59, 116, 229, 454, 903, 1791, 3552, 7045, 13974, 27719, 54984, 109065,
216339, 429126, 851207, 1688440, 3349161, 6643338
```

I implemented an `Rseq` which at all times keeps the memory of *all* previous items, but decided to drop it as the package was becoming big.

**iterr(initial values; expr, varname=values)** same as `rrseq` but does not print any value until the last `K`.

```
\xinttheiexpr iterr(0,1; @1+@2, i=2..5, 6..10)\relax
% the iterated over list is allowed to have disjoint defining parts.
55
```

**subsm(expr, var1=value1; var2=value2; ....; varN=valueN[:])** Simultaneous substitutions. The assigned values must not involve the variables. An optional final semi-colon is allowed.

```
\xintiieval{subsm(x+2y+3z+4t, x=1; y=10; z=100; t=1000;)}
4321
```

**subsn(expr, var1=value1; var2=value2; ....; varN=valueN[:])** Simultaneous substitutions. The assigned values may involve all variables located further to its right. An optional final semi-colon is allowed.

```
\xintiieval{subsn(x+y+z+t, x=20y; y=20z; z=20t; t=1)}
8421
```

**ndmap(function, values1; values2; ....; valuesN[:])** Construction of a nested list (a priori having `N` dimensions) from function values. The function must be an `N`-variable function (or a function accepting arbitrarily many arguments), but it is not constrained to produce only scalar values. Only in the latter case is the output really an `N`-dimensional “`ndlist`” type object. An optional final semi-colon in the input before the closing parenthesis is allowed.

```
\xintdeffunc foo(a,b,c,d) = a+b+c+d;
\begin{multicols}{2}
\xintthealign\xintexpr ndmap(foo, 1000,2000,3000; 100,200,300; 10,20,30; 1,2,3)\relax
\end{multicols}
```

<sup>13</sup> Prior to 1.4, one could use `@` in `rrseq()` and `iterr()` as an alias to `@1`. This undocumented feature is dropped and `@` will break `rrseq()` and `iterr()`.

```

[[[ 1111, 1112, 1113 ],
 [ 1121, 1122, 1123 ],
 [ 1131, 1132, 1133 ]],
 [[ 1211, 1212, 1213 ],
 [ 1221, 1222, 1223 ],
 [ 1231, 1232, 1233 ]],
 [[ 1311, 1312, 1313 ],
 [ 1321, 1322, 1323 ],
 [ 1331, 1332, 1333 ]]],
 [[ 2111, 2112, 2113 ],
 [ 2121, 2122, 2123 ],
 [ 2131, 2132, 2133 ]],
 [[ 2211, 2212, 2213 ],
 [ 2221, 2222, 2223 ],
 [ 2231, 2232, 2233 ]],
 [[ 2311, 2312, 2313 ],
 [ 2321, 2322, 2323 ],
 [ 2331, 2332, 2333 ]]],
 [[ 3111, 3112, 3113 ],
 [ 3121, 3122, 3123 ],
 [ 3131, 3132, 3133 ]],
 [[ 3211, 3212, 3213 ],
 [ 3221, 3222, 3223 ],
 [ 3231, 3232, 3233 ]],
 [[ 3311, 3312, 3313 ],
 [ 3321, 3322, 3323 ],
 [ 3331, 3332, 3333 ]]]]

```

**ndseq(expr, var1=values1; var2=values2; ...; varN = valuesN[;])** Constructs a nested list (a priori having **N** dimensions) from substitutions in an expression involving **N** (dummy) variables. The expression is not constrained to produce only scalar values. Only in the latter case is the output really an **N**-dimensional “**ndlist**” type object. An optional final semi-colon in the input before the closing parenthesis is allowed.

```

\begin{multicols}{2}
\xintthealign\xintexpr ndseq(a+b+c+d, a=1000,2000,3000; b=100,200,300;
                                c=10,20,30; d=1,2,3;)\relax
\end{multicols}% in case of page break, this makes amusing zigzag rendering

```

```

[[[ 1111, 1112, 1113 ],
 [ 1121, 1122, 1123 ],
 [ 1131, 1132, 1133 ]],
 [[ 1211, 1212, 1213 ],
 [ 1221, 1222, 1223 ],
 [ 1231, 1232, 1233 ]],
 [[ 1311, 1312, 1313 ],
 [ 1321, 1322, 1323 ],
 [ 1331, 1332, 1333 ]]],
 [[ 2111, 2112, 2113 ],
 [ 2121, 2122, 2123 ],
 [ 2131, 2132, 2133 ]],
 [[ 2211, 2212, 2213 ],
 [ 2221, 2222, 2223 ],
 [ 2231, 2232, 2233 ]],
 [[ 2311, 2312, 2313 ],
 [ 2321, 2322, 2323 ],
 [ 2331, 2332, 2333 ]]],
 [[ 3111, 3112, 3113 ],
 [ 3121, 3122, 3123 ],
 [ 3131, 3132, 3133 ]],
 [[ 3211, 3212, 3213 ],
 [ 3221, 3222, 3223 ],
 [ 3231, 3232, 3233 ]],
 [[ 3311, 3312, 3313 ],
 [ 3321, 3322, 3323 ],
 [ 3331, 3332, 3333 ]]]]

```

Recursions may be nested, with **@@@(n)** giving access to the values of the outer recursion... and there is even **@@@@(n)** to access the outer outer recursion but I never tried it!

The following keywords are recognized:

**abort** it is a pseudo-variable which indicates to stop here and now.

**omit** it is a pseudo-variable which says to omit this value and go to next one.

**break(stuff)** says to abort and insert **stuff** as last value.

**<integer>++** serves to generate a potentially infinite list. In conjunction with an **abort** or **break()** this is often more efficient than iterating over a pre-established list of values.

```
\xinttheiiexpr iter(1;(@>10^40)?{break(@)}{2@},i=1++)\relax
```

10889035741470030830827987437816582766592 is the smallest power of 2 with at least forty one digits.

The `i=<integer>++` syntax (any letter is allowed in place of `i`) works only in the form `<letter>_=<integer>++`, something like `x=10,17,30++` is not legal. The `<integer>` must be a  $\TeX$ -allowable integer.

```
First Fibonacci number at least |2^31| and its index
% we use iterr to refer via @1 and @2 to the previous and previous to previous.
\xinttheiiexpr iterr(0,1; (@1>=2^31)?{break(@1, i)}{@2+@1}, i=1++)\relax
```

First Fibonacci number at least  $2^{31}$  and its index 2971215073, 47. If one also wants the previous Fibonacci number one only has to use `break(@2, @1, i)` in the above example.

## 2.5. Generators of arithmetic progressions

- `a..b` constructs the **small** integers from the ceil `[a]` to the floor `[b]` (possibly a decreasing sequence): one has to be careful if using this for algorithms that `1..0` for example is not empty or `1` but expands to `1, 0`. Again, `a..b` can not be used with `a` and `b` greater than  $2^{31} - 1$ . Also, only about at most **5000** integers can be generated (this depends upon some  $\TeX$  memory settings).

The `..` has lower precedence than the arithmetic operations.

```
\xintexpr 1.5+0.4..2.3+1.1\relax; \xintexpr 1.9..3.4\relax; \xintexpr 2..3\relax
```

2, 3; 2, 3; 2, 3

The step of replacing `a` by its ceil and `b` by its floor is a kind of silly overhead, but `a` and `b` are allowed to be themselves the result of computations and there is no notion of “int” type in `\xinteval`. The solution is, when `a` and `b` are given explicit integers to temporarily switch to the `\xintiexpr` parser:

```
\xintexpr \xintiexpr 1..10\relax\relax
```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

On the other hand integers from `\xintexpr 1..10\relax` are already in raw `xintfrac` format for example `3/1[0]` which speeds up their usage in the macros internally involved in computations... thus perhaps what one gains on one side is lost on the other side.

- `a..[d]..b` generates “real” numbers along arithmetic progression of reason `d`. It does not replace `a` by its ceil, nor `b` by its floor. The generated list is empty if `b-a` and `d` are of opposite signs; if `d=0` or if `a=b` the list expands to single element `a`.

```
\xintexpr 1.5..[1.01]..11.23\relax
```

1.5, 2.51, 3.52, 4.53, 5.54, 6.55, 7.56, 8.57, 9.58, 10.59

At 1.4, this generator behaves in `\xintfloatexpr` exactly as in `\xintexpr`, i.e. exactly. This is breaking change.

```
\xintDigits:=6;
\xintexpr\xintfloatexpr 100..[1.23456]..110\relax\relax
\xintDigits:=16;
```

100, 101.23456, 102.46912, 103.70368, 104.93824, 106.1728, 107.40736, 108.64192, 109.87648

This demonstration embedded the float expression in the exact parser only to avoid the rounding to the prevailing precision on output, thus we can see that internally additions are done exactly and not with 6 digits mantissas (in this example).

## 2.6. Python slicing and indexing of one-dimensional sequences

We denote here by *list* or *sequence* a general *opple*, either given as a variable or explicitly. In the former case the parentheses are optional.<sup>14</sup>

- `(list)[n]` returns the `n+1`th item if `n>=0`. If `n<0` it enumerates items from the tail. Items are numbered as in Python, the first element corresponding to `n=0`.

```
\xintexpr (0..10)[6], (0..10)[-1], (0..10)[23*18-22*19]\relax
6, 10, 7
```

This also works for singleton *opples* which are in fact a *number*:

```
\xintexpr (7)[0], (7)[-1], 9, (7)[-2], 9\relax
7, 7, 9, 9
```

In the example above the parentheses serve to disambiguate from the raw `xintfrac` format such as `7[-1]` which, although discouraged, is accepted on input. And we used a trick to show that `(7)[-2]` returns `nil`.

The behaviour changes for singleton *opples* which are not *numbers*. They are thus *nutples*, or equivalently they are the bracketing (bracing, packing) of another *opple*. In this case, the meaning of the syntax for item indexing is, as in Python, item extraction:

```
\xintexpr [0,1,2,3,4,5][2], [0,1,2,3,4,5][-3]\relax\newline
\xintexpr [0,[1,2,3,4,5],6][1][-1]\relax
2, 3
5
```

- `(list)[:n]` produces the first `n` elements if `n>0`, or suppresses the last `|n|` elements if `n<0`.

```
\xintiexpr (0..10)[:6]\relax\ and \xintiexpr (0..10)[: -6]\relax
0, 1, 2, 3, 4, 5 and 0, 1, 2, 3, 4
```

As above, the meaning change for *nutples* and fits with expectations from Python regarding its sequence types:

```
\xintiexpr [0..10][:6]\relax\ and \xintiexpr [0..10][: -6]\relax
[0, 1, 2, 3, 4, 5] and [0, 1, 2, 3, 4]
```

- `(list)[n:]` suppresses the first `n` elements if `n>0`, or extracts the last `|n|` elements if `n<0`.

```
\xintiexpr (0..10)[6:]\relax\ and \xintiexpr (0..10)[ -6:]\relax
6, 7, 8, 9, 10 and 5, 6, 7, 8, 9, 10
```

As above, the meaning change for *nutples* and fit with expectations from Python with *tuple* or *list* types:

```
\xintiexpr [0..10][6:]\relax\ and \xintiexpr [0..10][ -6:]\relax
[6, 7, 8, 9, 10] and [5, 6, 7, 8, 9, 10]
```

- Finally, `(list)[a:b]` also works according to the Python ``slicing'' rules (inclusive of negative indices). Notice though that stepping is currently not supported.

```
\xinttheiexpr (1..20)[6:13]\relax\ = \xinttheiexpr (1..20)[6-20:13-20]\relax
\newline
\xinttheiexpr [1..20][6:13]\relax\ = \xinttheiexpr [1..20][6-20:13-20]\relax
7, 8, 9, 10, 11, 12, 13 = 7, 8, 9, 10, 11, 12, 13
[7, 8, 9, 10, 11, 12, 13] = [7, 8, 9, 10, 11, 12, 13]
```

<sup>14</sup> Even for an "open list", if it is given as a *variable* then the indexing or slicing will not apply to its last item but to itself as an entity.

- It is naturally possible to execute such slicing operations one after the other (the syntax is simplified compared to before 1.4):

```
\xintexpr (1..50)[13:37][10:-10]\relax\newline
\xintexpr (1..50)[13:37][10:-10][-1]\relax
24, 25, 26, 27
27
```

## 2.7. NumPy like nested slicing and indexing for arbitrary oples and nutples

I will give one illustrative example and refer to the NumPy documentation for more.

Notice though that our interpretation of the syntax is more general than NumPy's concepts (of basic slicing/indexing):

- slicing and itemizing apply also to non-bracketed objects i.e. *oples*,
- the leaves do not have to be all at the same depth,
- there are never any out-of-range index errors: out-of-range indices are silently ignored.

```
\begin{multicols}{3}
\xintdefvar myArray = ndseq(a+b+c, a=100,200,300; b=40,50,60; c=7,8,9);
myArray = \xintthealign\xintexpr myArray\relax
\columnbreak
mySubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2]\relax
myExtractedSubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2][0]\relax
\columnbreak
myExtractedSubArray = \xintthealign\xintexpr myArray[0:2,0:2,0:2][0,1]\relax
\noindent
firstExtractedScalar = \xintexpr myArray[0:2,0:2,0:2][0,1,0]\relax\newline
secondExtractedScalar = \xintexpr myArray[0,1,0]\relax\par
\end{multicols}
```

myArray =	mySubArray =	myExtractedSubArray =
[[[ 147, 148, 149 ],	[[[ 147, 148 ],	[ 157, 158 ]
[ 157, 158, 159 ],	[ 157, 158 ]],	firstExtractedScalar = 157
[ 167, 168, 169 ]],	[[ 247, 248 ],	secondExtractedScalar = 157
[[ 247, 248, 249 ],	[ 257, 258 ]]]	
[ 257, 258, 259 ],	myExtractedSubArray =	
[ 267, 268, 269 ]],	[[ 147, 148 ],	
[[ 347, 348, 349 ],	[ 157, 158 ]]	
[ 357, 358, 359 ],		
[ 367, 368, 369 ]]]		

As said before, *stepping* is not yet implemented. Also the NumPy extension to Python for item selection (i.e. via a **tuple** of comma separated indices) is not yet implemented.

## 2.8. Tacit multiplication

Tacit multiplication (insertion of a *\**) applies when the parser is currently either scanning the digits of a number (or its decimal part or scientific part, or hexadecimal input), or is looking for an infix operator, and:

- (1.) encounters a count or dimen or skip register or variable or an  $\varepsilon$ -TeX expression, or
- (2.) encounters a sub-\xintexpression, or
- (3.) encounters an opening parenthesis, or

- (4.) encounters a letter (which is interpreted as signaling the start of either a variable or a function name), or  
 (5.) (of course, only when in state "looking for an operator") encounters a digit.

**!!!!ATTENTION!!!!**

Explicit digits prefixing a variable, or a function, whose name starts with an **e** or **E** will trap the parser into trying to build a number in scientific notation. So the **\*** must be explicitly inserted.

```
\xintdefiivar e := (2a+4b+6d+N)/:7;%
\xintdefiivar f := (c+11d+22*e)//451;% 22e would raise errors
I don't think I will fix this anytime soon...
```

For example, if **x**, **y**, **z** are variables all three of **(x+y)z**, **x(y+z)**, **(x+y)(x+z)** will create a tacit multiplication.

Furthermore starting with release **1.2e**, whenever tacit multiplication is applied, in all cases it always ``ties'' more than normal multiplication or division, but still less than power. Thus **x/2y** is interpreted as **x/(2y)** and similarly for **x/2max(3,5)** but **x^2y** is still interpreted as **(x^2)\*y** and **2n!** as **2\*n!**.

```
\xintdefvar x:=30;\xintdefvar y:=5;%
\xinttheexpr (x+y)x, x/2y, x^2y, x!, 2x!, x/2max(x,y)\relax
1050, 30/10, 4500, 265252859812191058636308480000000, 530505719624382117272616960000000,
30/60
```

Since **1.2q** tacit multiplication is triggered also in cases such as **(1+2)5** or **10!20!30!**.

```
\xinttheexpr (10+7)5, 4!4!, add(i, i=1..10)10, max(x, y)100\relax
85, 576, 550, 3000
```

The ``tie more'' rule applies to all cases of tacit multiplication. It impacts only situations with a division operator as the last seen operator, as multiplication is mathematically associative.

```
\xinttheexpr 1/(3)5, (1+2)/(3+4)(5+6), 2/x(10), 2/10x,
3/y\xintiexpr 5+6\relax, 1/x(y)\relax\
differ from\newline\xinttheexpr 1/3*5, (1+2)/(3+4)*(5+6), 2/x*(10), 2/10*x,
3/y*\xintiexpr 5+6\relax, 1/x*(y)\relax\par
1/15, 3/77, 2/300, 2/300, 3/55, 1/150 differ from
5/3, 33/7, 20/30, 60/10, 33/5, 5/30
```

Note that **y\xinttheiexpr 5+6\relax** would have tried to use a variable with name **y11** rather than doing **y\*11**: tacit multiplication works only in front of sub-**\xintexpressions**, not in front of **\xinttheexpressions** which are unlocked into explicit digits.

Here is an expression whose meaning is completely modified by the ``tie more'' property of tacit multiplication:

```
\xintdeffunc e(z):=1+z(1+z/2(1+z/3(1+z/4)));
will be parsed as
\xintdeffunc e(z):=1+z*(1+z/(2*(1+z/(3*(1+z/4)))));
which is not at all the presumably hoped for:
\xintdeffunc e(z):=1+z*(1+(z/2)*(1+(z/3)*(1+(z/4))));
```

## 2.9. User defined variables

Since release **1.1** it is possible to make an assignment to a variable name and let it be known to the parsers of **xintexpr**. Since **1.2p** simultaneous assignments are possible. Since **1.4** simulta-

neous assignments are possible with a right-hand-side being a **nutple** which will be automatically unpacked.

```
\xintdefvar myPi:=3.141592653589793238462643;%
$myPi = \xinteval{myPi}$\newline % (there is already built-in Pi variable)
\xintdefvar x_1, x_2, x_3 := 10, 20, 30;%
$x_1 = \xinteval{x_1}, x_2 = \xinteval{x_2}, x_3 = \xinteval{x_3}$\newline
\xintdefvar x_1, x_2, x_3 := [100, 200, 300];%
$x_1 = \xinteval{x_1}, x_2 = \xinteval{x_2}, x_3 = \xinteval{x_3}$\par
```

myPi = 3.141592653589793238462643

x<sub>1</sub> = 10, x<sub>2</sub> = 20, x<sub>3</sub> = 30

x<sub>1</sub> = 100, x<sub>2</sub> = 200, x<sub>3</sub> = 300

Simultaneous assignments with more variables than values do not raise an error but simply set the extra variables to the **nil** value.

```
\xintdefiivar a, b, c := [1, 2];% will be automatically unpacked
The value of a is \xinteval{a}, the one of b is \xinteval{b} and
the one of c is \xinteval{c}.
```

The value of a is 1, the one of b is 2 and the one of c is .

```
\xintdefiivar a, b, c := 314;%
The value of a is \xinteval{a}, the one of b is \xinteval{b} and
the one of c is \xinteval{c}.
```

The value of a is 314, the one of b is and the one of c is .

Notice that **nil** variables must be used with caution as they break arithmetic operations if used as operands to them. And they are not the same as the **None** variables, which can also be input as `[]`.

Simultaneous assignments with less variables than values do not raise an error but set the last variable to be the ope concatenating the remaining values.

```
\xintdefiivar seq := 1..10;%
\xintdefiivar a, seq := seq;%
\xintdefiivar b, seq := seq;%
\xintdefiivar c, d, seq := seq;%
The value of a is \xinteval{a}, the one of b is \xinteval{b}, the one of c is \xinteval{c},
the one of d is \xinteval{d}, the one of seq is \xinteval{seq}.
```

The value of a is 1, the one of b is 2, the one of c is 3, the one of d is 4, the one of seq is 5, 6, 7, 8, 9, 10.

In the above we define a variable **seq** but there is a built-in function **seq()**. It is indeed allowed to use the same name for both a variable and a function.<sup>15</sup> But for safety we will unassign **seq** now:

```
\xintunassignvar{a}\xintunassignvar{b}\xintunassignvar{c}\xintunassignvar{d}%
\xintunassignvar{seq}%
```

Single letter names **a..z** and **A..Z** are pre-declared by the package for use as a special type of variables called ```dummy variables''`. Unassigning them restores this initial meaning. See further `\xintunassignvar` and `\xintnewdummy`. Since 1.4 even assigned variables can be used in the call signatures of function declarations.

Regarding the manipulation of an “open list” as above, there is no way to obtain with only one use of the variable both its last item and the reduction of the variable to its truncated self. One can do rather:

```
\xintdefiivar mylist := 1..10;%
\xintdefiivar z, mylist := last(mylist), mylist[:-1];%
The value of z is \xinteval{z} and mylist is now \xinteval{mylist}.\par
```

The value of z is 10 and mylist is now 1, 2, 3, 4, 5, 6, 7, 8, 9.

This uses twice **mylist** and is about the same as doing it in two steps:

<sup>15</sup> But until a bugfix added at release 1.4i, some built-in function names (those implementing syntax with dummy variables, and the so-called “pseudo”-functions) were fragile under such overloading.



```
\xintdefiivar w := last(mylist);%
\xintdefiivar mylist := mylist[:-1];%
The value of w is \xinteval{w} and mylist is now \xinteval{mylist}.%
\xintunassignvar{z}\xintunassignvar{w}\xintunassignvar{mylist}\par
```

The value of `w` is 9 and `mylist` is now 1, 2, 3, 4, 5, 6, 7, 8.

It is recommended generally speaking to work with “closed (i.e. bracketed) lists” because only them and numbers can be arguments to functions (but see [\xintdeffunc](#) and the notion of variadic last argument). For more on the Python-like slicing used above see [subsection 2.6](#) and [subsection 2.13.4](#). For more information relative to variables versus arguments see [subsection 2.13.6](#).

- For catcodes issues (particularly, for the semi-colon used to delimit the fetched expression), see the discussion of [\xintexprSafeCatcodes](#) and some comments in the section documenting [\xintdeffunc](#).
- Both syntaxes `\xintdefvar foo := <expr>;` and `\xintdefvar foo = <expr>;` are accepted.
- Spaces in the variable name or around the equal sign are removed and are immaterial.
- The variable names are expanded in an `\edef` (and stripped of spaces). Example:

```
\xintdefvar x\xintListWithSep{, x}{\xintSeq{0}{10}} := seq(2**i, i = 0..10);%
```

This defines `x0`, `x1`, ..., `x10` for future usage.

Legal variable names are composed of letters, digits, `_` and `@` and characters. A variable name must start with a letter. Variable names starting with a `@` or `_` are reserved for internal usage.<sup>16</sup>

As `x_1x_2` or even `x_1x` are licit variable names, and as the parser does not trace back its steps, input syntax must be `x_1*x_2` if the aim is to multiply such variables.

Using `\xintdefvar`, `\xintdefiivar`, or `\xintdeffloatvar` means that the variable value will be computed using respectively `\xintexpr`, `\xintiexpr` or `\xintfloatexpr`. It can then be used in all three parsers, as long as the parser understands the format. Currently this means that variables using `\xintdefvar` or `\xintdeffloatvar` can be used freely either with `\xintexpr` or `\xintfloatexpr` but not with `\xintiexpr`, and variables defined via `\xintdefiivar` can be used in all parsers.

When defining a variable with `\xintdeffloatvar` it (or generally speaking its numerical leaves) is rounded to [\xinttheDigits](#) precision. So the variable holds the same value as would be printed via [\xintfloateval](#) for the same computation.

Prior to 1.4e, this was the case only if the variable definition actually involved some computation.

However the `\xintfloatexpr..relax` wrapper by itself induces no rounding. If it is encountered in the typesetting flow, the print-out will be rounded to [\xinttheDigits](#) precision, but this is an effet of behaving like [\xintfloateval](#) in this context.

```
% Since 1.4e, \xintdeffloatvar always rounds (to \xinttheDigits)
\xintdeffloatvar e:=2.7182818284590452353602874713526624977572470936999595749669676;%
1) \xintexpr e\relax\newline % shows the recorded value: it is rounded
2) \xintfloatexpr % when used in typesetting flow, acts like \xintfloateval:
   2.7182818284590452353602874713526624977572470936999595749669676
\relax\newline % the print-out is rounded.
3) \xintexpr
   \xintfloatexpr
   2.7182818284590452353602874713526624977572470936999595749669676
   \relax
\relax\newline
%
% but we can see via the \xintexpr wrapper all the digits were there rounding
```

<sup>16</sup> The process of variable declaration does not check that these rules are met, and breakage will arise on use, if rules are not followed. For example, prior to 1.4g, using a variable (illegally) declared with a name starting with a (normal, catcode 8) `_` triggered an infinite loop.

```
% can be forced using an extra 0+, the float() function, or the [D] option.
% tidbit: comparison operators do not pre-round, so 1.2345678 is not same as
% (1.2345678+0) in low precision.
%
\begingroup\xintDigits:=4;%
4) \xintifboolfloatexpr{1.2345 == 1.23456}
   {\error}{Different! Comparisons do not pre-round to Digits precision.}\newline
5) \xintifboolfloatexpr{1.2345 == 1.2345 + 0}
   {\error}{Different! Right hand side rounded from operation,
           left hand side not rounded.}\par
\endgroup
```

1) 2.718281828459045

2) 2.718281828459045

3) 2.7182818284590452353602874713526624977572470936999595749669676

4) Different! Comparisons do not pre-round to Digits precision.

5) Different! Right hand side rounded from operation, left hand side not rounded.

After issuing `\xintverbosetrue` the values of defined variables are written out to the log (and terminal). As in this example:

```
Package xintexpr Info: (on line 1)
  Variable myPi defined with value {3141592653589793238462643[-24]}.
Package xintexpr Info: (on line 2)
  Variable x_1 defined with value {10}.
Package xintexpr Info: (on line 2)
  Variable x_2 defined with value {20}.
Package xintexpr Info: (on line 2)
  Variable x_3 defined with value {30}.
Package xintexpr Info: (on line 3)
  Variable List defined with value {0}{1}{3}{6}{10}{15}{21}{28}{36}{45}{55}
.
Package xintexpr Info: (on line 4)
  Variable Nuple defined with value {{0}{1}{9}{36}{100}{225}{441}{784}{1296}
  }{2025}{3025}}.
Package xintexpr Info: (on line 5)
  Variable FourthPowers defined with value {{0}{1}{81}{1296}{10000}{50625}{
  194481}{614656}{1679616}{4100625}{9150625}}.
```

*source*

### 2.9.1. `\xintunassignvar`

`\xintunassignvar{⟨variable⟩}` will make the `⟨variable⟩` un-assigned. For example in the previous section we used

```
\xintdeffloatvar e := ...some value...;
```

To undo one either waits for the current scope (e.g. a  $\text{\TeX}$  environment) to expire or the impatient does:

```
\xintunassignvar{e}
```



In this special case of using `\xintunassignvar` with a single `⟨letter⟩` the effect is actually to let the `⟨letter⟩` recover the meaning of a dummy variable (i.e. it is the same as using `\xintnewdummy` documented in the next section):

```
% overwriting a dummy letter
\xintdefvar i := 3;%
\xinteval{i^3} is as expected but |\xinteval{add(i,i=1..10)}| computes
\xinteval{add(i,i=1..10)} because "i" has the fixed value 3.\newline
\xintunassignvar{i}% back to normal
After |\xintunassignvar{i}|
```

`\xinteval{add(i,i=..10)}` is evaluated with "i" acting as a dummy variable and thus outputs `\xinteval{add(i,i=..10)}.\par`

27 is as expected but `\xinteval{add(i,i=1..10)}` computes 30 because "i" has the fixed value 3. After `\xintunassignvar{i}` `\xinteval{add(i,i=..10)}` is evaluated with "i" acting as a dummy variable and thus outputs 55.

Under `\xintglobaldefstrue` regime the effect of `\xintunassignvar` is of global scope.

[source](#)

### 2.9.2. `\xintnewdummy`

Any catcode 11 character can serve as a dummy variable, via this declaration:

`\xintnewdummy{<letter>}% the <letter> must be of catcode letter!`

For example with Lua<sub>TeX</sub> or Xe<sub>TeX</sub> the following works:

% Requires a Unicode engine

`\xintnewdummy{ξ}`  
`\xinteval{add(ξ, ξ=1..10)}`

Starting with 1.4, it is allowed to use `\xintnewdummy` to define ``dummy variables'' having names with more than one letter. They can then be used as expected:

% Requires a Unicode engine

`\xintnewdummy{ατλν}`  
`\xintdefunc test(ατλν) = sqr(1 + ατλν);`  
`\xinteval{seq(test(ατλν), ατλν = 0..10)}`

Remark regarding Op<sub>TeX</sub>: it is a format using Lua<sub>TeX</sub>, but non ascii letters have catcode ``other''. So for the above example to compile with `optex`, the catcodes need to be set to ``letter'' beforehand:

% Requires a Unicode engine

% next line is required if with Op<sub>TeX</sub> (only):

`\xintfor* #1 in {ατλν}\do{\catcode`#1=11 }`  
`\xintnewdummy{ατλν}`  
`\xintdefunc test(ατλν) = sqr(1 + ατλν);`  
`\xinteval{seq(test(ατλν), ατλν = 0..10)}`

Under `\xintglobaldefstrue` regime the effect of `\xintnewdummy` is of global scope.

[source](#)

[source](#)

### 2.9.3. `\xintensuredummy`, `\xintrestorevariable`

Use

`\xintensuredummy{<character>}`  
...  
... code using the (catcode 11) character as a dummy variable  
...  
`\xintrestorevariable{<character>}`

if other parts need the letter as an assigned variable name. For example `xinttrig` being written at high level needs a few genuine dummy variables, and it uses `\xintensuredummy` to be certain everything is ok.

## 2.10. User defined functions

2.10.1 <code>\xintdefunc</code> .....	44
2.10.2 <code>\xintdefiifunc</code> .....	47
2.10.3 <code>\xintdeffloatfunc</code> .....	47
2.10.4 <code>\xintdefufunc</code> , <code>\xintdefiufunc</code> , <code>\xintdeffloatufunc</code> .....	47
2.10.5 Using the same name for both a variable and a function .....	50
2.10.6 <code>\xintunassignexprfunc</code> , <code>\xintunassigniifunc</code> , <code>\xintunassignfloatexprfunc</code> .....	50
2.10.7 <code>\ifxintverbose</code> conditional .....	50
2.10.8 <code>\ifxintglobaldefs</code> conditional .....	50

2.10.9 \xintNewFunction .....	50
-------------------------------	----

[source](#)

### 2.10.1. \xintdeffunc

Here is an example:

```
\xintdeffunc
```

```
Rump(x,y):=1335 y^6/4 + x^2 (11 x^2 y^2 - y^6 - 121 y^4 - 2) + 11 y^8/2 + x/2y;
```

(notice the numerous tacit multiplications in this expression; and that  $x/2y$  is interpreted as  $x/(2y)$ .)

- The ending semi-colon is allowed to be of active catcode, as `\xintdeffunc` temporarily resets catcodes via `\xintexprSafeCatcodes` before parsing the expression.

But this will fail if the whole thing is inside a macro definition. Then the used semi-colon must be the standard one.

In the case of a  $\TeX$  document using Babel, and a language such as French which makes the semi-colon active, it is still the standard one inside the preamble, so there is no problem there.

For a macro definition done inside the document body, which, as I understand, is sin, almost evil, either turn off locally the activation (`\string;` will not work because `\xintdeffunc` uses delimited macros to fetch all the way to the semi-colon), or define in the preamble `\MyDefFunc{#1}` to do `\xintdeffunc #1;` and use that in the `\newcommand\foo` inside the document body.

- Semi-colons used inside the expression need not be hidden inside braces. (new with 1.4)
- The colon before the equal sign is optional and its (reasonable) catcode does not matter.

Here are a few important items (bookmark this for reading again later once you have gained experience in using this interface...):

- The function names are composed of letters, digits, underscores or @ signs. A function name must start with a letter. It may be a single letter (see [subsubsection 2.10.5](#)).
- The variable names used in the function signature may be multi-letter words. It is also allowed for them to already be in use for previously declared variables. Their meanings will get restored for usage after the function declaration.
- A function can be declared with at most nine arguments. It can be declared as a function with no arguments.
- If in the function declaration the last argument is prefixed by `*`, it stands for a `nutple` which will gather all arguments of the function call beyond the first positional ones. See [subsubsection 2.13.6](#) for additional explanations on such “variadic” arguments.
- Recursive definitions are possible; for them to not generate error or fall in infinite loops, the use of the short-circuit conditionals `?` and `??` is mandatory.
- If a function is used in another definition it will check if it is applied to numerical arguments and if this is the case will expand fully.
- The previous item has an exception for functions with no arguments; they never expand immediately in other function definitions (else they would be almost like variables). This provides a way to define functions with parameters: simply let their definition use some functions with no arguments.

- ```
\xintdefloatfunc foo(x) := float_dgt(\xintexpr foo(x)\relax);
```

- And in the reverse direction one can do:

The main difficulty of `\xintdeffunc` is with the pseudo-functions `seq()`, `iter()`, etc..., which admit the keywords `omit`, `abort`, `break()`. We have no alternative for them, if the iterated over values are not entirely numerical than to postpone expansion, but this means simply storing for later a possibly big sub-expression.

At 1.4 we did some obstinate work to make this working but:

- this means that the stored function body has not been entirely parsed, parsing will happen on the fly at each execution for small or large bits,
- there remains a main stumbling-block. If the variables used in the function declaration are used only in the iterated over values or the initial values, then the mechanism may work. If however they are used not only in those values iterated over but directly in the expression which the generators map to the iterated over values, then it will break certainly. Indeed at this stage the variables are simply names, and it is impossible to transfer the mechanism which converts these names into numerical arguments for delayed usage by the declared function. Except if one is ready to basically freeze the entire thing; which then is not any different at all than using `\xintNewFunction`.

Conclusion: if some `\xintdeffunc` break, check if it does not fit the above criterion before reporting... and recall `\xintNewFunction` is your friend. It has the big advantage of declaring a function for all parsers simultaneously!

A special note on `subs()`: it is and has always been hopeless in `\xintdeffunc` context. All it does (if it works at all) after being malaxed by `\xintdeffunc` is to copy over at the indicated places the *recipe* to compute something. Thus at every location where that something is needed it will be evaluated from scratch again. Yes, this is disappointing. But... on the other hand the more general `seq()` does work, or pretends to work. Let me illustrate to make things clear. We start with this:

```
\xintverbosetrue
\xintdeffunc foo(x,y,z) = subs(S + S^2, S = x+y+z);
\xintdeffunc bar(x,y,z) = seq(S + S^2, S = x+y+z);
\xintexpr foo(100,10,1), bar(100,10,1)\relax
\xintverbosfalse
```

12432, 12432

It produces in the log:

Package xintexpr Info: (on line 2)

Function foo for \xintexpr parser associated to \XINT\_expr\_userfunc\_foo with meaning macro:#1#2#3->{\xintAdd {\xintAdd {\xintAdd {#1}{#2}}{#3}}{\xintPow {\xintAdd {\xintAdd {#1}{#2}}{#3}}{2}}}

Package xintexpr Info: (on line 3)

Function bar for \xintexpr parser associated to \XINT\_expr\_userfunc\_bar with meaning macro:#1#2#3->\expanded \bgroup \expanded {\unexpanded {\XINT\_expr\_seq:\_b {\xintbareeval S + S^2\relax !S}}{\xintAdd {\xintAdd {#1}{#2}}{#3}}^}

Even without understanding all details one sees that in the first case the `\xintAdd {\xintAdd {#1}{#2}}{#3}` appears twice, and in the second case only once. But in the second case we have a yet to evaluate expression. So the second approach is not much different in its effect than using the more simple-minded `\xintNewFunction`. Besides one gets a feeling why the function arguments can not appear in the expression but only in the iterated over values, because there is no way to understand what `x`, `y`, `z` are supposed to mean without adding extra structure showing they map to `#1`, `#2`, `#3`.

The above remarks apply to `subsm()` and `subsn()`. Even if they do work in `\xintdeffunc` context (warning, testing at 1.4 release has remained minimal), they will not bring added efficiency if the substituted values are to be used multiple times. They may still be useful to visually simplify the input of a big expression by expressing it in terms of smaller constituents.

Another workaround if one wants genuine (not “macro”-) functions for some expression where the same thing is used multiple times is to define helper functions computing the intermediate data. One can see illustrations of this in the code source of [xinttrig](#) (or in the matrix multiplication example at the end of this chapter).

[source](#)

### 2.10.2. `\xintdefiifunc`

With `\xintdeffunc` the created function is known by the `\xintexpr` parser only. For usage in the `\xintiexpr` parser, it is required to use `\xintdefiifunc`.

[source](#)

### 2.10.3. `\xintdeffloatfunc`

With `\xintdeffunc` the created function is known by the `\xintexpr` parser only. For usage in the `\xintfloatexpr` parser, it is required to use `\xintdeffloatfunc`.

Note: the optional argument `[Q]` accepted by `\xintfloatexpr` does not work with `\xintdeffloatfunc`. It is still possible to wrap the expression in `float(expression,Q)`, if it evaluates to a scalar.

[source](#)

[source](#)

[source](#)

### 2.10.4. `\xintdefufunc`, `\xintdefiufunc`, `\xintdeffloatufunc`

This allows to define so-called “Universal functions”. This is terminology borrowed from NumPy.

Here is an example:

```
\xintdefiivar Array = ndmap(lcm, 1..5; 1..10; 1..10);
Array = \xintthealign\xintiexpr Array\relax
\xintdefiufunc foo(x) = x^3;
\begin{figure}[htbp]
\caption{Output of a universal function acting on an array}\label{fig:ufunc}
\centeredline{$\vcenter{\xintthealign\xintiexpr foo(Array)\relax}$}
\end{figure}
See \autopageref{fig:ufunc} for the output.
```

Array =

```
[[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10 ],
 [ 2,  2,  6,  4, 10,  6, 14,  8, 18, 10 ],
 [ 3,  6,  3, 12, 15,  6, 21,  9, 30,  9 ],
 [ 4,  4, 12,  4, 20, 12, 28,  8, 36, 20 ],
 [ 5, 10, 15, 20,  5, 30, 35, 40, 45, 10 ],
 [ 6,  6,  6, 12, 30,  6, 42, 24, 18, 30 ],
 [ 7, 14, 21, 28, 35, 42,  7, 56, 63, 70 ],
 [ 8,  8, 24,  8, 40, 24, 56,  8, 72, 40 ],
 [ 9, 18,  9, 36, 45, 18, 63, 72,  9, 90 ],
 [10, 10, 30, 20, 10, 30, 70, 40, 90, 10 ]],
[[ 2,  2,  6,  4, 10,  6, 14,  8, 18, 10 ],
 [ 2,  2,  6,  4, 10,  6, 14,  8, 18, 10 ],
 [ 6,  6,  6, 12, 30,  6, 42, 24, 18, 30 ],
 [ 4,  4, 12,  4, 20, 12, 28,  8, 36, 20 ],
 [10, 10, 30, 20, 10, 30, 70, 40, 90, 10 ],
 [ 6,  6,  6, 12, 30,  6, 42, 24, 18, 30 ],
 [14, 14, 42, 28, 70, 42, 14, 56, 126, 70 ],
 [ 8,  8, 24,  8, 40, 24, 56,  8, 72, 40 ],
 [18, 18, 18, 36, 90, 18, 126, 72, 18, 90 ],
 [10, 10, 30, 20, 10, 30, 70, 40, 90, 10 ]],
[[ 3,  6,  3, 12, 15,  6, 21,  9, 30,  9 ],
 [ 6,  6,  6, 12, 30,  6, 42, 24, 18, 30 ],
 [ 3,  6,  3, 12, 15,  6, 21,  9, 30,  9 ]]
```



```
[ 12, 12, 12, 12, 60, 12, 84, 24, 36, 60 ],
[ 15, 30, 15, 60, 15, 30, 105, 120, 45, 30 ],
[ 6, 6, 6, 12, 30, 6, 42, 24, 18, 30 ],
[ 21, 42, 21, 84, 105, 42, 21, 168, 63, 210 ],
[ 24, 24, 24, 24, 120, 24, 168, 24, 72, 120 ],
[ 9, 18, 9, 36, 45, 18, 63, 72, 9, 90 ],
[ 30, 30, 30, 60, 30, 30, 210, 120, 90, 30 ]],
[[ 4, 4, 12, 4, 20, 12, 28, 8, 36, 20 ],
[ 4, 4, 12, 4, 20, 12, 28, 8, 36, 20 ],
[ 12, 12, 12, 12, 60, 12, 84, 24, 36, 60 ],
[ 4, 4, 12, 4, 20, 12, 28, 8, 36, 20 ],
[ 20, 20, 60, 20, 20, 60, 140, 40, 180, 20 ],
[ 12, 12, 12, 12, 60, 12, 84, 24, 36, 60 ],
[ 28, 28, 84, 28, 140, 84, 28, 56, 252, 140 ],
[ 8, 8, 24, 8, 40, 24, 56, 8, 72, 40 ],
[ 36, 36, 36, 36, 180, 36, 252, 72, 36, 180 ],
[ 20, 20, 60, 20, 20, 60, 140, 40, 180, 20 ]],
[[ 5, 10, 15, 20, 5, 30, 35, 40, 45, 10 ],
[ 10, 10, 30, 20, 10, 30, 70, 40, 90, 10 ],
[ 15, 30, 15, 60, 15, 30, 105, 120, 45, 30 ],
[ 20, 20, 60, 20, 20, 60, 140, 40, 180, 20 ],
[ 5, 10, 15, 20, 5, 30, 35, 40, 45, 10 ],
[ 30, 30, 30, 60, 30, 30, 210, 120, 90, 30 ],
[ 35, 70, 105, 140, 35, 210, 35, 280, 315, 70 ],
[ 40, 40, 120, 40, 40, 120, 280, 40, 360, 40 ],
[ 45, 90, 45, 180, 45, 90, 315, 360, 45, 90 ],
[ 10, 10, 30, 20, 10, 30, 70, 40, 90, 10 ]]]
```

See [page 49](#) for the output.

The function can be applied to any nested structure:

```
\xintiexpr foo([1, [2, [3, [4, [5, 6, 7, 8, 9, 10]]]])\relax
```

```
1, [8, [27, [64, [125, 216, 343, 512, 729, 1000]]]]
```

It must be defined as function acting on scalars, but its value type is not constrained.

```
\xintdefiivar Array = [1..10];
\xintdefiifunc foo(x) = [1..x];
\xintthealign\xintiexpr foo(Array)\relax
```

```
[[ 1 ],
[ 1, 2 ],
[ 1, 2, 3 ],
[ 1, 2, 3, 4 ],
[ 1, 2, 3, 4, 5 ],
[ 1, 2, 3, 4, 5, 6 ],
[ 1, 2, 3, 4, 5, 6, 7 ],
[ 1, 2, 3, 4, 5, 6, 7, 8 ],
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]]
```

It is even allowed to produce oples and act on oples:

```
\xintdefiivar Ople = 1..10;
\xintdefiifunc bar(x) = x, x^2, x^3;
\xintiexpr bar(Ople)\relax
```

```
1, 1, 1, 2, 4, 8, 3, 9, 27, 4, 16, 64, 5, 25, 125, 6, 36, 216, 7, 49, 343, 8, 64, 512, 9, 81, 729,
10, 100, 1000
```

Figure 1: Output of a universal function acting on an array

```

[[[ 1,      8,      27,      64,      125,      216,      343,      512,      729,      1000    ],
   [ 8,      8,      216,      64,      1000,      216,      2744,      512,      5832,      1000    ],
   [ 27,      216,      27,      1728,      3375,      216,      9261,      13824,      729,      27000    ],
   [ 64,      64,      1728,      64,      8000,      1728,      21952,      512,      46656,      8000    ],
   [ 125,      1000,      3375,      8000,      125,      27000,      42875,      64000,      91125,      1000    ],
   [ 216,      216,      216,      1728,      27000,      216,      74088,      13824,      5832,      27000    ],
   [ 343,      2744,      9261,      21952,      42875,      74088,      343,      175616,      250047,      343000    ],
   [ 512,      512,      13824,      512,      64000,      13824,      175616,      512,      373248,      64000    ],
   [ 729,      5832,      729,      46656,      91125,      5832,      250047,      373248,      729,      729000    ],
   [ 1000,      1000,      27000,      8000,      1000,      27000,      343000,      64000,      729000,      1000    ]],
 [[ 8,      8,      216,      64,      1000,      216,      2744,      512,      5832,      1000    ],
   [ 8,      8,      216,      64,      1000,      216,      2744,      512,      5832,      1000    ],
   [ 216,      216,      216,      1728,      27000,      216,      74088,      13824,      5832,      27000    ],
   [ 64,      64,      1728,      64,      8000,      1728,      21952,      512,      46656,      8000    ],
   [ 1000,      1000,      27000,      8000,      1000,      27000,      343000,      64000,      729000,      1000    ],
   [ 216,      216,      216,      1728,      27000,      216,      74088,      13824,      5832,      27000    ],
   [ 2744,      2744,      74088,      21952,      343000,      74088,      2744,      175616,      2000376,      343000    ],
   [ 512,      512,      13824,      512,      64000,      13824,      175616,      512,      373248,      64000    ],
   [ 5832,      5832,      5832,      46656,      729000,      5832,      2000376,      373248,      5832,      729000    ],
   [ 1000,      1000,      27000,      8000,      1000,      27000,      343000,      64000,      729000,      1000    ]],
 [[ 27,      216,      27,      1728,      3375,      216,      9261,      13824,      729,      27000    ],
   [ 216,      216,      216,      1728,      27000,      216,      74088,      13824,      5832,      27000    ],
   [ 27,      216,      27,      1728,      3375,      216,      9261,      13824,      729,      27000    ],
   [ 1728,      1728,      1728,      1728,      216000,      1728,      592704,      13824,      46656,      216000    ],
   [ 3375,      27000,      3375,      216000,      3375,      27000,      1157625,      1728000,      91125,      27000    ],
   [ 216,      216,      216,      1728,      27000,      216,      74088,      13824,      5832,      27000    ],
   [ 9261,      74088,      9261,      592704,      1157625,      74088,      9261,      4741632,      250047,      9261000    ],
   [ 13824,      13824,      13824,      13824,      1728000,      13824,      4741632,      13824,      373248,      1728000    ],
   [ 729,      5832,      729,      46656,      91125,      5832,      250047,      373248,      729,      729000    ],
   [ 27000,      27000,      27000,      216000,      27000,      27000,      9261000,      1728000,      729000,      27000    ]],
 [[ 64,      64,      1728,      64,      8000,      1728,      21952,      512,      46656,      8000    ],
   [ 64,      64,      1728,      64,      8000,      1728,      21952,      512,      46656,      8000    ],
   [ 1728,      1728,      1728,      1728,      216000,      1728,      592704,      13824,      46656,      216000    ],
   [ 64,      64,      1728,      64,      8000,      1728,      21952,      512,      46656,      8000    ],
   [ 8000,      8000,      216000,      8000,      8000,      216000,      2744000,      64000,      5832000,      8000    ],
   [ 1728,      1728,      1728,      1728,      216000,      1728,      592704,      13824,      46656,      216000    ],
   [ 21952,      21952,      592704,      21952,      2744000,      592704,      21952,      175616,      16003008,      2744000    ],
   [ 512,      512,      13824,      512,      64000,      13824,      175616,      512,      373248,      64000    ],
   [ 46656,      46656,      46656,      46656,      5832000,      46656,      16003008,      373248,      46656,      5832000    ],
   [ 8000,      8000,      216000,      8000,      8000,      216000,      2744000,      64000,      5832000,      8000    ]],
 [[ 125,      1000,      3375,      8000,      125,      27000,      42875,      64000,      91125,      1000    ],
   [ 1000,      1000,      27000,      8000,      1000,      27000,      343000,      64000,      729000,      1000    ],
   [ 3375,      27000,      3375,      216000,      3375,      27000,      1157625,      1728000,      91125,      27000    ],
   [ 8000,      8000,      216000,      8000,      8000,      216000,      2744000,      64000,      5832000,      8000    ],
   [ 125,      1000,      3375,      8000,      125,      27000,      42875,      64000,      91125,      1000    ],
   [ 27000,      27000,      27000,      216000,      27000,      27000,      9261000,      1728000,      729000,      27000    ],
   [ 42875,      343000,      1157625,      2744000,      42875,      9261000,      42875,      21952000,      31255875,      343000    ],
   [ 64000,      64000,      1728000,      64000,      64000,      1728000,      21952000,      64000,      46656000,      64000    ],
   [ 91125,      729000,      91125,      5832000,      91125,      729000,      31255875,      46656000,      91125,      729000    ],
   [ 1000,      1000,      27000,      8000,      1000,      27000,      343000,      64000,      729000,      1000    ]]]

```

### 2.10.5. Using the same name for both a variable and a function

It is licit to overload a variable name (all Latin letters are predefined as dummy variables) with a function name and vice versa. The parsers will decide from the context if the function or variable interpretation must be used (dropping various cases of tacit multiplication as normally applied).

```
\xintdefiifunc f(x):=x^3;
\xinttheiexpr add(f(f),f=100..120)\relax\newline
\xintdeffunc f(x,y):=x^2+y^2;
\xinttheexpr mul(f(f(f,f),f(f,f)),f=1..10)\relax
\xintunassigniexprfunc{f}\xintunassignexprfunc{f}%
28205100
186188134867578885427848806400000000
```

[source](#)

[source](#)

[source](#)

### 2.10.6. \xintunassignexprfunc, \xintunassigniexprfunc, \xintunassignfloatexprfunc

Function names can be unassigned via `\xintunassignexprfunc{<name>}`, `\xintunassigniexprfunc{<name>}`, and `\xintunassignfloatexprfunc{<name>}`.

```
\xintunassignexprfunc{e}
\xintunassignexprfunc{f}
```

Warning: no check is done to avoid undefining built-in functions...

[source](#)

### 2.10.7. \ifxintverbose conditional

With `\xintverbosetrue` the meanings of the functions (or rather their associated macros) will be written to the log. For example the **Rump** declaration above generates this in the log file:

```
Function Rump for \xintexpr parser associated to \XINT_expr_userfunc_Rump w
ith meaning macro:#1#2->{\xintAdd {\xintAdd {\xintAdd {\xintDiv {\xintMul {1335
}{\xintPow {#2}{6}}}{4}}{\xintMul {\xintPow {#1}{2}}{\xintSub {\xintSub {\xintS
ub {\xintMul {11}{\xintMul {\xintPow {#1}{2}}{\xintPow {#2}{2}}}}{\xintPow {#2}
{6}}}{\xintMul {121}{\xintPow {#2}{4}}}{2}}}{\xintDiv {\xintMul {11}{\xintPow
{#2}{8}}}{2}}{\xintDiv {#1}{\xintMul {2}{#2}}}}
```



The meanings written out to the log for more complicated functions may sometimes use the same character at different locations but with different catcodes.

It may thus be impossible to retokenize it (even after having removed the extra spaces from the added line breaks).

This is in contrast with variable values which are always output in the log in the benign way, using digits, braces and some characters of catcode 12.

[source](#)

### 2.10.8. \ifxintglobaldefs conditional

If true user defined variables (`\xintdefvar`, ...) and functions (`\xintdeffunc`, ..., `\xintNewFunction`) for the expression parsers, as well as macros obtained via `\xintNewExpr` et al. have global scope. If false (default) they have local scope.

[source](#)

### 2.10.9. \xintNewFunction

This is syntactic sugar which allows to use notation of functions for what is nothing more in disguise than a  $\TeX$  macro. Here is an example:

```
\xintNewFunction {foo}[3]{add(mul(x+i, i=#1..#2),x=1..#3)}
```

We now have a genuine function `foo()` of three variables which can be used in *all three* parsers.

```
\xintexpr seq(foo(0, 3, j), j= 1..10)\relax
```

24, 144, 504, 1344, 3024, 6048, 11088, 19008, 30888, 48048

Each time the created “macro-function” `foo()` will be encountered the corresponding replacement text will get inserted as a sub-expression (of the same type as the surrounding one), the macro parameters having been replaced with the (already evaluated) function arguments, and the parser *will then have to parse the expression*. It is very much like a macro substitution, but with parentheses and comma separated arguments (which can be arbitrary expressions themselves).

It differs fundamentally from `\xintdeffunc` as it realizes no pre-parsing whatsoever of the associated sub-expression; using it shortens the input but not the parsing time (which however is most of the time negligible compared to actual numerical computations). Use it for syntax which `\xintdeffunc` does not parse successfully.

## 2.11. Examples of user defined functions

### 2.11.1. Example with vectors and matrices

Suppose we want to manipulate 3-dimensional vectors, which will be represented as **nutples** of length 3. And let's add a bit of matrix algebra.

```
\xintdeffunc dprod(V, W) := V[0]*W[0] + V[1]*W[1] + V[2]*W[2];
\xintdeffunc cprod(V, W) := [V[1]*W[2] - V[2]*W[1],
                             V[2]*W[0] - V[0]*W[2],
                             V[0]*W[1] - V[1]*W[0]];
\xintdeffunc Det3(U, V, W) := dprod(cprod(U, V), W);
\xintdeffunc DetMat(M) = Det3(*M);
\xintdeffunc RowMat(U, V, W) := [U, V, W];
\xintdeffunc ColMat(U, V, W) := [[U[0], V[0], W[0]],
                                 [U[1], V[1], W[1]],
                                 [U[2], V[2], W[2]]];

\xintdeffunc MatMul(A, B) :=
[[A[0,0]*B[0,0]+A[0,1]*B[1,0]+A[0,2]*B[2,0],
  A[0,0]*B[0,1]+A[0,1]*B[1,1]+A[0,2]*B[2,1],
  A[0,0]*B[0,2]+A[0,1]*B[1,2]+A[0,2]*B[2,2]],
 [A[1,0]*B[0,0]+A[1,1]*B[1,0]+A[1,2]*B[2,0],
  A[1,0]*B[0,1]+A[1,1]*B[1,1]+A[1,2]*B[2,1],
  A[1,0]*B[0,2]+A[1,1]*B[1,2]+A[1,2]*B[2,2]],
 [A[2,0]*B[0,0]+A[2,1]*B[1,0]+A[2,2]*B[2,0],
  A[2,0]*B[0,1]+A[2,1]*B[1,1]+A[2,2]*B[2,1],
  A[2,0]*B[0,2]+A[2,1]*B[1,2]+A[2,2]*B[2,2]]];

\xintdefvar vec1, vec2, vec3 := [1, 1, 1], [1, 1/2, 1/4], [1, 1/3, 1/9];
\xintdefvar mat1 = RowMat(vec1, vec2, vec3);
\xintdefvar mat2 = ColMat(vec1, vec2, vec3);
\xintdefvar mat12 = MatMul(mat1,mat2);
\xintdefvar mat21 = MatMul(mat2,mat1);
Some computations (|align| executes multiple times hence we pre-computed!):
\begin{align*}
M_1 &= \vcenter{\xintthealign \xintexpr mat1\relax}&&\quad
M_2 \cdot M_1 = \vcenter{\xintthealign \xintexpr mat21\relax}\\[3\jot]
M_2 &= \vcenter{\xintthealign \xintexpr mat2\relax}&&\quad
M_1 \cdot M_2 = \vcenter{\xintthealign \xintexpr mat12\relax}
\end{align*}
$$
\det(M_1) = \xinteval{DetMat(mat1)},\quad
\det(M_1.M_2) = \xinteval{reduce(DetMat(mat12))},\quad
```

```
\det(M_2.M_1) = \xinteval{reduce(DetMat(mat21))}
$$
```

Some computations (align executes multiple times hence we pre-computed!):

$$\begin{aligned}
 M_1 &= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1/2 & 1/4 \\ 1 & 1/3 & 1/9 \end{bmatrix}, & M_2.M_1 &= \begin{bmatrix} 3 & 11/6 & 49/36 \\ 11/6 & 49/36 & 251/216 \\ 49/36 & 251/216 & 1393/1296 \end{bmatrix} \\
 M_2 &= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1/2 & 1/3 \\ 1 & 1/4 & 1/9 \end{bmatrix}, & M_1.M_2 &= \begin{bmatrix} 3 & 7/4 & 13/9 \\ 7/4 & 21/16 & 43/36 \\ 13/9 & 43/36 & 91/81 \end{bmatrix} \\
 \det(M_1) &= -1/18, & \det(M_1.M_2) &= 1/324, & \det(M_2.M_1) &= 1/324
 \end{aligned}$$

For some hair-raising experience check the `\xintverbosetrue` output in the log... here is an alternative with two (three, counting `dprod()`) helper functions:

```
% annoying that Tr also starts Trace, but Spur is available
% well Sp also starts Spectrum. Big problems.
\xintdeffunc Tr(M) :=
  [[M[0,0], M[1,0], M[2,0]],
   [M[0,1], M[1,1], M[2,1]],
   [M[0,2], M[1,2], M[2,2]]];
```

```
\xintdeffunc MatMul_a(r1, r2, r3, c1, c2, c3) :=
  [[dprod(r1, c1), dprod(r1, c2), dprod(r1, c3)],
   [dprod(r2, c1), dprod(r2, c2), dprod(r2, c3)],
   [dprod(r3, c1), dprod(r3, c2), dprod(r3, c3)]];
```

```
\xintdeffunc MatMul(A, B) := MatMul_a(*A, *Tr(B));
```

And once we have the transpose and the scalar product of vectors, we can simply use `ndmap()` for a lean syntax (this would extend to arbitrary dimension):

```
\xintdeffunc MatMul(A, B) = ndmap(dprod, *A; *Tr(B));
\xintdefvar mat1212 = MatMul(mat12, mat12);
\begin{group}
\def\xintexprPrintOne      #1{\xintTeXFrac{#1}}%
\def\xintexpralignbegin    {\begin{pmatrix}}%
\def\xintexpralignend      {\end{pmatrix}}%
\def\xintexpralignlinesep  {\noexpand\\[2\jot]}% counteract an internal \expanded
\def\xintexpraligninnersep  {\&}%
\let\xintexpralignleftbracket\empty \let\xintexpralignleftsep\empty
\let\xintexpralignrightbracket\empty \let\xintexpralignrightsep\empty
$$ \xintthealign \xintexpr mat1\relax \cdot \xintthealign \xintexpr mat2\relax \cdot
  \xintthealign \xintexpr mat1\relax \cdot \xintthealign \xintexpr mat2\relax =
  \xintthealign \xintexpr mat12\relax ^2 = \xintthealign \xintexpr mat1212\relax$$
$$ \det(M_1\cdot M_2 \cdot M_1 \cdot M_2) = \xinteval{reduce(DetMat(mat1212))}$$
\end{group}
```

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & \frac{1}{3} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{1}{4} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & \frac{1}{3} & \frac{1}{9} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2} & \frac{1}{3} \\ 1 & \frac{1}{4} & \frac{1}{9} \end{pmatrix} = \begin{pmatrix} 3 & \frac{7}{4} & \frac{13}{9} \\ \frac{7}{4} & \frac{21}{16} & \frac{43}{36} \\ \frac{13}{9} & \frac{43}{36} & \frac{91}{81} \end{pmatrix}^2 = \begin{pmatrix} \frac{18337}{1296} & \frac{48067}{5184} & \frac{93853}{11664} \\ \frac{48067}{5184} & \frac{128809}{20736} & \frac{253687}{46656} \\ \frac{93853}{11664} & \frac{253687}{46656} & \frac{501289}{104976} \end{pmatrix}$$

$$\det(M_1 \cdot M_2 \cdot M_1 \cdot M_2) = \frac{1}{104976}$$

### 2.11.2. Example with the Rump test

Let's try out our `Rump()` function:

```
\xinttheexpr Rump(77617,33096)\relax.
```

-54767/66192. Nothing problematic for an exact evaluation, naturally!

Thus to test the Rump polynomial (it is not quite a polynomial with its  $x/2y$  final term) with floats, we *must* also declare `Rump` as a function to be used there:

```
\xintdeffloatfunc
  Rump(x,y):=333.75 y^6 + x^2 (11 x^2 y^2 - y^6 - 121 y^4 - 2) + 5.5 y^8 + x/2y;
```

The numbers are scanned with the current precision, hence as here it is 16, they are scanned exactly in this case. We can then vary the precision for the evaluation.

```
\def\CR{\cr}
\halign
{\tabskiplex
\hfil\bfseries#\xintDigits:=\xintloopindex\relax
\xintthefloatexpr Rump(77617,33096)#\cr
\xintloop [8+1]
\xintloopindex &\relax\CR
\ifnum\xintloopindex<40 \repeat
}
```

```
8 7e29
9 -1e28
10 5e27
11 -3e26
12 4e25
13 3e24
14 3e23
15 -2e22
16 1e21
17 -5e20
18 1.17260394005317863
19 1.000000000000000001e18
20 -9.9999999999999998827e16
21 1.000000000000000011726e16
22 3.0000000000000001172604e15
23 -9.999999999999998827396060e13
24 -1.9999999999999988273960599e13
25 -1.999999999999998827396059947e12
26 1.1726039400531786318588349
27 -5.9999999999999988273960599468214e10
28 -9.9999999999999988273960599468213681e8
29 2.00000000117260394005317863186e8
30 1.000000011726039400531786318588e7
31 -999998.8273960599468213681411651
32 200001.17260394005317863185883490
33 -9998.82739605994682136814116509548
34 -1998.827396059946821368141165095480
35 -198.82739605994682136814116509547982
36 21.1726039400531786318588349045201837
37 -0.8273960599468213681411650954798162920
38 -0.82739605994682136814116509547981629200
39 -0.827396059946821368141165095479816292000
40 -0.8273960599468213681411650954798162919990
```

### 2.11.3. Examples of recursive definitions

Recursive definitions *require* using the short-circuit branching operators.

```
\xintdefiifunc GCD(a,b):=(b)?{GCD(b,a/:b)}{a};
```

This of course is the Euclidean algorithm: it will be here applied to variables which may be fractions. For example:

```
\xinttheexpr GCD(385/102, 605/238)\relax
```

55/714

There is already a built-in `gcd()` (which accepts arbitrarily many arguments):

```
\xinttheexpr gcd(385/102, 605/238)\relax
```

55/714

Our second example is modular exponentiation:

```
\xintdefiifunc powmod_a(x, m, n) :=
  isone(m)?
    % m=1, return x modulo n
    { x /: n }
  % m > 1 test if odd or even and do recursive call
  { odd(m)? { x*sqr(powmod_a(x, m//2, n)) /: n }
    {      sqr(powmod_a(x, m//2, n)) /: n }
  }
;
\xintdefiifunc powmod(x, m, n) := (m)?{powmod_a(x, m, n)}{1};
```

I have made the definition here for the `\xintiexpr` parser; we could do the same for the `\xintexpr`-parser (but its usage with big powers would quickly create big denominators, think `powmod(1/2, 1000, 1)` for example.)

```
\xinttheiexpr seq(powmod(x, 1000, 128), x=9, 11, 13, 15, 17, 19, 21)\relax\par
```

65, 97, 33, 1, 1, 33, 97

The function assumes the exponent is non-negative (the Python `pow` behaved the same until 3.8 release), but zealous users will add the necessary code for negative exponents, after having defined another function for modular inverse!

If function **A** needs function **B** which needs function **A** start by giving to **B** some dummy definition, define **A**, then define **B** properly. TODO: add some example here...

## 2.12. Links to some (old) examples within this document

- The utilities provided by `xinttools` (section 6), some completely expandable, others not, are of independent interest. Their use is illustrated through various examples: among those, it is shown in subsection 7.8 how to implement in a completely expandable way the [Quick Sort algorithm](#) and also how to illustrate it graphically. Other examples include some dynamically constructed alignments with automatically computed prime number cells: one using a completely expandable prime test and `\xintApplyUnbraced` (subsection 7.2), another one with `\xintFor*` (subsection 7.6).
- One has also a [computation of primes within an \edef](#) (subsection 6.15), with the help of `\xintiloop`. Also with `\xintiloop` an [automatically generated table of factorizations](#) (subsection 7.5).
- The code for the title page fun with Fibonacci numbers is given in subsection 3.18 with `\xintFor*` joining the game.
- The computations of  $\pi$  and  $\log 2$  (subsection 15.11) using `xint` and the computation of the [convergents of e](#) with the further help of the `xintcfrac` package are among further examples.
- Also included, an [expandable implementation of the Brent-Salamin algorithm](#) for evaluating  $\pi$ .



- The [subsection 7.4](#) implements expandably the Miller-Rabin pseudo-primality test.
- The functionalities of [xintexpr](#) are illustrated with various other examples, in [subsection 2.10.1](#), [Functions with dummy variables](#), [subsection 7.1](#) or [Recursive definitions](#).

## 2.13. Oples and nutples: the 1.4 terminology

*Skip this on first reading, else you will never start using the package.* SKIP THIS! (understood?)

In this section I will describe a mathematical terminology which models how the parser handles the input syntax with numbers, commas, and brackets, and how it maps internally to  $\text{\TeX}$  specific concept, particularly braces and macro arguments.

|       |                                             |    |
|-------|---------------------------------------------|----|
| .13.1 | Base terminology .....                      | 55 |
| .13.2 | Items (and sub-items) versus elements ..... | 56 |
| .13.3 | Oples as trees .....                        | 57 |
| .13.4 | Ople slicing and indexing .....             | 57 |
| .13.5 | Nested slicing of oples .....               | 58 |
| .13.6 | Function arguments versus variables .....   | 58 |
| .13.7 | Final words on leaves .....                 | 59 |
| .13.8 | Farewell, thanks for your visit! .....      | 59 |

### 2.13.1. Base terminology

We start with a set  $\mathcal{A}$  of *atoms*, which represent numeric data. In  $\text{\TeX}$  syntax such *atoms* are always braced, more precisely, currently they look like

`{raw format within  $\text{\TeX}$  braces}`

The  $\text{\TeX}$  braces are not set-theoretical braces here, they are simply used for  $\text{\TeX}$ nical reasons (one could imagine using rather some terminator token, but ultimately support macros for built-in and user defined functions rely on  $\text{\TeX}$  macros with undelimited parameters, at least so far).

Our category  $C$  of “oples” is the smallest collection of *totally ordered finite sets* verifying these properties:

1. The empty set  $\emptyset$  is an *ople*, i.e. it belongs to  $C$ .
2. Each singleton set  $\{0\}$  whose element  $0$  is either an *atom*  $a \in \mathcal{A}$  or an *ople* qualifies as an *ople*.
3.  $C$  is stable by concatenation.

Notes:

- We refer to the empty set  $\emptyset$  via the variable *nil*.<sup>17</sup>
- It is convenient to accept the empty set as being also an *atom*. If this is done, then we may refer to the original *atoms* (elements of  $\mathcal{A}$ ) as *non empty numerical data*.
- Concatenation is represented in the syntax by the comma. Thus repeated commas are like only one and *nil* is a neutral element.
- A singleton *ople*  $\{a\}$  whose single element is a (non-empty) *atom* is called a *number*.<sup>18</sup>
- The operation of constructing  $\{0\}$  from the *ople*  $0$  is called *bracing* (set theory,  $\text{\TeX}$ ), or *bracketing* ([xintexpr](#) input syntax, Python *lists*), or *packing* (as a reverse to Python's unpacking of sequence type objects). In the expression input syntax it corresponds to enclosing  $0$  within square brackets:  $[0]$ .

<sup>17</sup> There is actually a built-in variable with this name. At 1.4, `\xintexpr \relax` is legal and also generates the *nil*.  
<sup>18</sup> This has to be taken in a general sense, for example with [polexpr](#), polynomials are represented by such “numbers”.

- A braced *ople* is called a *nutple*. Among them `{nil}` (aka `{0}`) is a bit special. It is called the *none-ple*.<sup>19</sup> It is not `nil`.<sup>20</sup>

Each *ople* has a *length* which is its cardinality as set. The singleton *oples* are called *one-ples*. There are thus two types of *one-ples*:

- *numbers* `{a}`,  $a \in \mathcal{A}$ ,
- *nutples* `{0}`,  $0 \in \mathcal{C}$ .

If we consider the empty set `nil` on the same footing as `atoms`, the two types have only one common object which is the *none-ple*. As a rule arithmetic operations will either break or silently convert the *none-ple* to the zero value:

```
\xinteval{3+[], 5^[], 10*[]}
```

`3, 1, 0`. But attention that `\xintiieval` in contrast to `\xinteval` is broken by such inputs.

### 2.13.2. Items (and sub-items) versus elements

In order to illustrate these concepts, let us consider how one should interpret notation such as `3,5,7,9` when it arises in an `\xintexpression`:

**tempting vocabulary:** Each of `3`, `5`, `7`, and `9` is an *item*, or *element* of the (comma separated) *list*. In other terms we have here a list with 4 items.

**rigorous vocabulary:** each one of `3`, `5`, `7`, `9` stands for an *ople* (of the *one-ple* type) and `3,5,7,9` stands for their *concatenation*.

It is important to understand that in an `\xintexpression`, there is no difference between `3,5,7` and `3,,,,5,,,,,,7`. So the view of the comma as separator is misleading. In other terms, the comma is NOT a separator but the (associative) operator of concatenation of totally ordered sets, and the number `3` for example represents a (singleton) set.

If we want to refer to `3` or `5` or `7` or `9` as “the items of the (open) list `3,5,7,9`” (and probably this documentation already has such utterances, due to legacy reasons from the pre-1.4 internal model), we *must* realize that this clashes with using the word *item* as synonymous to *element* in the set-theoretical sense.

To repeat, any *ople* `0` is a finite totally ordered set: if not the empty set, it has *elements*  $a_1, \dots, a_k$ , and the above means that its *items* are the singleton *oples* (aka *one-ples*)  $I_1 = \{a_1\}, \dots, I_k = \{a_k\}$ . Each  $a_j$  may be an *atom*, then  $I_j$  is a *number*, or  $a_j$  is an *ople* (possibly the empty set), then  $I_j$  is a *nutple* whose depth is one more than the one of the *ople*  $a_j$ .

Thus we can refer to “items” but must then understand they are not “elements”: “items” are “singleton sub-sets”. The cardinality (aka length) of an *ople* is also the number of its items. It would be tempting to use the terminology “sub-item” to keep in mind they are “sub-sets” but this would again create confusion: a *nutple* has only one item which is itself; and we need some terminology to refer to the individual numbers in the *nutple* given in input as `[1,2,3]` for example. It is natural to refer to `1`, `2`, `3` as “sub-items” of `[1,2,3]` as the latter may be an “item” (it is in particular an “item” of itself, the unique one at that).

We distinguish the *oples* of length zero (there is only one, the empty set) or at least two as those which can never be an “item”. Those of length one, the *one-ples*, are exactly those which can be “items”. Among them some may have “sub-items”, they are the *nutples* with the exception of the *none-ple*. And the others do not have “sub-items”, they are the *numbers* and the *none-ple* (whose

<sup>19</sup> Prior to version 1.4j of this documentation it was called the *not-ple*. <sup>20</sup> There is (experimental) a pre-defined “None” variable which stands for the *none-ple*. It can also be input as `[]`.

input syntax is either `[]` or the variable `None`).<sup>21</sup>

### 2.13.3. Oples as trees

We say that the empty set `nil` and *atoms* are *leaves*.

We associate with any *ople* a tree. The root is the *ople*. In the case of the `nil` *ople*, there is nothing else than the root, which we then consider also a *leaf*. Else the children at top level are the successive *elements* (not “items”!) of the *ople*.<sup>22</sup> Among the elements some are *atoms* giving *leaves* of the tree, others are *nutples* which in turn have children. In the special case of the *none-ple* we consider it has a child, which is the empty set and this is why we consider the empty set `nil` to be also a potential *leaf*. We then proceed recursively. We thus obtain from the root *ople* a tree whose vertices are either *oples* or *leaves*. Only the empty set `nil` is both a *leaf* and an *ople*.

Considering the empty set `nil` as an *atom* fits with the `xintexpr` internal implementation based on  $\text{\TeX}$ : `nil` is an empty pair of braces `{}`, whereas an *atom* is a braced representation of a numeric value using digits and other characters. We construct *oples* by putting one after the other such constituents and bracing them, and then repeating the process recursively.

It has also an impact on the definition of the *depth* (a.k.a as *maximal dimension*) of an *ople*. For example the *ople*  $\{0A_1A_2\}$  with three elements, among them the empty set and two atoms is said to have depth 1, or to have maximal dimension 1. And  $\{\{0\}A_1A_2\}$  is of depth 2 because it has a leaf (the empty set) which is a child of a child of the *ople*. NumPy *ndarrays* have a more restricted structure for example  $\{A_{00}A_{01}\{A_{10}A_{11}\}\}$  is a 2-dimensional array, where all leaves are at the same depth. When slicing empties the array from its atoms, NumPy keeps the shape information but prints the array as `[]`. This will not be the case with `xintexpr`, which has no other way to indicate the shape than display it.

```
\xinteval{[],[]}  
[], []  
\xinteval{[[0,1],[10,11]][:,2:]}  
[], []
```

### 2.13.4. Ople slicing and indexing

“Set-theoretical” slicing of an *ople* means replacing it with one of its subsets. This applies also if it is a *number*. Then it can be sliced only to itself or to the empty set (indeed it has only one element, which is an *atom*). Similarly the *none-ple* can only be sliced to give itself or the empty set. And more generally a *nutple* is a singleton so also can only be set-sliced to either the empty set or itself.

`xintexpr` extends “Python-like” slicing to act on *oples*:

- if they are not *nutples* set-theoretical slicing applies,
- if they are *nutples* (only case having a one-to-one correspondence in Python) then the slicing happens *within brackets*: i.e. the *nutple* is unpacked then the set-theoretical slicing is applied, then the result is *repacked* to produce a new *nutple*.

<sup>21</sup> A note on the `\xintverbosetrue` regime: for a variable defined to be `3,5,7,9`, it will say that its value is  $\{3\}\{5\}\{7\}\{9\}$ , because it does not keep the external set-theoretical braces. The braces here are only  $\text{\TeX}$  braces, and  $\{3\}$  is an *atom*. The *number* would be  $\{\{3\}\}$  with the external braces being set-theoretical and also used internally as  $\text{\TeX}$  braces. From the four numbers  $\{\{3\}\}$ , ...,  $\{\{9\}\}$  concatenation gives  $\{\{3\}\{5\}\{7\}\{9\}\}$ , which is the *ople* `3,5,7,9`. But the log view drops deliberately the external braces. If the variable is defined to be the *nutple* `[3,5,7,9]`, then the log view will be  $\{\{3\}\{5\}\{7\}\{9\}\}$  (up to details on how exactly the numeric quantities are coded) and the actual internal  $\text{\TeX}$  entity will be  $\{\{\{3\}\{5\}\{7\}\{9\}\}\}$ , where the two external layers of braces are both set-theoretical and  $\text{\TeX}$ nical braces. <sup>22</sup> We could also consider a tree for which the children of the root node would be its items and recursively; in that case the leaves would be *numbers* and possibly the `None`. The tree of the `nil` would be the empty tree, the tree of `None` would have a single node and no edges. Such a tree would match the input syntax (of course applying the rule that iterated commas are like only one). The tree which is described in this section matches more directly the internal syntax, hence is more useful to the author, who is also the sole reader who extracts some benefit from reading this documentation once in a while.

With these conventions the *none-ple* for example is invariant under slicing: unpacking it gives the empty set, which has only the empty set as subset and repacking gives back the *none-ple*. Slicing a general *nutple* returns a *nutple* but now of course in general distinct from the first one.

The input syntax for Python slicing is to postfix a variable or a parenthesized *ople* with `[a:b]`. See [subsection 2.6](#) for more. There are never any out-of-range errors when slicing or indexing. All operations are licit and resolved by the *nil*, a.k.a. empty set.

“Set-theoretical” item indexing of an *ople* means reducing it to a subset which is a singleton. It is thus a special case of set-theoretical slicing (which is the general process of selecting a subset as replacement of a set).

*xintexpr* extends “Python-like” indexing to act on *oples*:

- if they are not *nutples* set-theoretical item indexing applies,
- if they are *nutples* (only case having a one-to-one correspondence in Python) then the meaning becomes *extracting*: i.e. the *nutple* is unpacked then the set-theoretical indexing is applied, but the result is *not repacked*.

For example when applied to the *none-ple* we always obtain the *nil*. Whereas as we saw slicing the *none-ple* always gives back the *none-ple*. Indexing is denoted in the syntax by postfixing by `[N]`. Thus for *nutples* (which are analogous to Python objects), there is genuine difference between the `[N]` extractor and the `[N:N+1]` slicer. But for *oples* which are either *nil*, a *number*, or of length at least 2, there is no difference.

### 2.13.5. Nested slicing of *oples*

Nested slicing is a concept from NumPy, which is extended by *xintexpr* to trees of varying depths. We have a chain of slicers and extractors. I will describe only the case of slicers and letting them act on a *nutple*. The first slicer gives back a new *nutple*. The second slicer will be applied to each of one of its remaining elements. However some of them may be *atoms* or the empty set. In the NumPy context all leaves are at the same depth thus this can happen only when we have reached beyond the last dimension (axis). This is not permitted by NumPy and generates an error. *xintexpr* does not generate an error. But any attempt to slice an *atom* or the empty set (as element of its container) removes it. Recall we call them *leaves*. We can not slice leaves. We can only slice non-leaf elements: such items are necessarily *nutples*. The procedure then applies recursively.

If we handle an extractor rather than a slicer, the procedure is similar: we can not extract out of an *atom* or the empty set. They are thus removed. Else we have a *nutple*. It is thus unpacked and replaced by the selected element. This element may be an *atom* or the empty set and any further slicer or extractor will remove them, or it is a *nutple* and the procedure applies with the next slicer/extractor.

*xintexpr* allows to apply such a `[a:b,c:d,N,e:f,...]` chain of slicing/extracting also to an *ople*, which is not a *nutple*. We simply apply the first step as has been described previously and successive steps will only get applied to either *nutples* or *leaves*, the latter getting silently removed by any attempted operation.

### 2.13.6. Function arguments versus variables

In a function declaration with `\xintdeffunc`, the call signature is parsed as a comma separated list, so here it is not true that repeated commas are like only one: repeated commas are not allowed and will break the function declaration.

When *xintexpr* parses a function call, it first constructs the *ople* which is delimited by the opening and closing parentheses, then it applies the function body, after having mapped the successive items (not the elements) of the parsed *ople* to the variables appearing in the function call signature. Hence the arguments in the call signature stand for *one-ples* (i.e. either *numbers* or *nutples*).

Let me explain why we can not define a function `foo(A,B)` of two *oples*: the function call will evaluate as an *ople* what is enclosed within the parentheses. It is then impossible in general to

split this uniquely into two oples **A** and **B**, except if for example we know a priori the length of **A**. We could imagine defining a declarative interface for a `foo(A,B)` with **A** preset to have 37 items or at least a pre-defined number of items but this is extraneous layer for a functionality no-one will use.

The alternative would be to consider that declaring `foo(A,B)` means **A** will pick-up always the first item and **B** all the remaining ones, and thus will be an ople; here, there are some  $\text{\TeX}$  implementation reasons which have dissuaded the author to do this.

In its place, a special syntax `foo(A,*B)` for the declaration of the function is available. It means that **B** stands for the **nutple** which receives as items all arguments in the function call beyond the first one already assigned to **A**.

More generally, the last positional argument in a function declaration can have the form `*⟨argname⟩`. This then means that `⟨argname⟩` represents a **nutple** which will receive as items all arguments in the function call remaining after the earlier positional arguments have been assigned. The declared function body is free to again use the syntax `*⟨argname⟩` which will unpack it and thus produce the ople concatenating all such optional arguments.

With `\xintdefvar` one can define a variable with value an **ople** of arbitrary cardinality. Such a variable can be used in a function call, it will then occupy the place of as many arguments as its cardinality (which is its number of elements, hence of its associated items). For example if function `foo` was declared as a function of 5 arguments `f(a,b,c,d,e)` it is legitimate to use it as `f(A,B)` if **A** is an ople-valued variable of length three and **B** of length two. The actual arguments `a,b,c,d,e` will be made to match the three items of **A** and the two items of **B**.

### 2.13.7. Final words on leaves

In case things were too clear, let's try to add a bit of confusion with an extra word on *leaves*. When we discuss informally (particularly to compare with NumPy) an input such as

```
[[1, 2], [3, 4]]
```

we may well refer to 1, 2, 3, and 4 as being “the leaves of the 2d array”. But obviously we have here numbers and previously we explained that a number is not a *leaf*, its *atom* is. Well, the point here is that we must make a difference between the input form as above and the actual constructed *ople* the parser will obtain out of it. In the input we do have numbers. The comma is a *concatenator*, it is not a separator for enumeration! The *ople* which corresponds to it has a  $\text{\TeX}$  representation like this:

```
{{{1}}{2}}>{{3}}{4}}
```

where we don't have the *numbers* anymore (which would look like `{{1}}, {{2}}, ...`) but numeric *atoms* `{1}, {2}, {3}, {4}` where the braces are  $\text{\TeX}$  braces and **not** set-theoretical braces (the other braces are both). Hence we should see the above as the **ople** `{{A00A01}}{A10A11}}` with atoms  $A_{00} = \{1\}$ , ..., being the leaves of the tree associated to (or which is) the *ople*.

Numbers may be called the *leaves* of the **input**, but once parsed, the input becomes an *ople* which is (morally) a tree whose leaves are *atoms* (and the empty set). This discussion can also be revisited with footnote 22 in mind.

### 2.13.8. Farewell, thanks for your visit!

I hope this is clear to everyone. If not, maybe time to say this section is not needed to understand almost all of the manual, but I needed to write it to be able to maintain in future my own software.

## 2.14. Expansion (for geeks only)

As mentioned already, the parsers are compatible with expansion-only context.

Also, they expand the expression piece by piece: the normal mode of operation of the parsers is to unveil the parsed material token by token. Unveiling is a process combining space swallowing, brace removal (one level generally), and *f-expansion*.

For example a closing parenthesis after some function arguments does not have to be immediately visible, it and the arguments themselves may arise from *f-expansion* (applied before grabbing each successive token). Even the ending `\relax` may arise from expansion. Even though the `\xinteval` user interface means that the package has at some point the entire expression in its hands, it immediately re-inserts it into token stream with an additional postfixed `\relax` and from this point on has lost any ways (a simple-minded delimited macro won't do because the expression is allowed to contain sub-`\xintexpressions`, even nested) to manipulate formally again the whole thing; it can only re-discover it one token at a time.

This general behaviour (which allows much more freedom in assembling expressions than is usually the case with familiar programming languages such as Python, although admittedly that freedom will prove useful only to power-`TEX` users and possibly does not have that many significant use cases) has significative exceptions. These exceptions are mostly related to “pseudo”-functions. A “pseudo”-function will grab some of its arguments via delimited macros. For example `subs(expr1, x=expr2)` needs to see the comma, equal sign and closing parenthesis. But it has mechanisms to allow `expr1` and `expr2` to possess their own commas and parentheses.

Inner semi-colons on the other hand currently always can originate from expansion. Defining functions or variables requires a visible semi-colon acting as delimiter of the expression, but inner semi-colons do not need to be hidden within braces or macros.

The expansion stops only when the ending `\relax` has been found (it is then removed from the token stream).

For catcode related matters see `\xintexprSafeCatcodes`.

A word of warning on the bracketed optional argument of respectively `\xintfloatexpr` and `\xintiexpr`. When defining macros which will hand over some argument to one of these two parsers, the argument may potentially start with a left square bracket `[` (e.g. argument could be `[1, 2, 3]`) and this will break the parser. The fix is to use in the macro definition `\xintfloatexpr\empty`. This extra `\empty` token will prevent the parser from thinking there is an optional argument and it will then disappear during expansion.

If comparing to other languages able to handle floating point numbers or big integers, such as Python, one should take into account that what the `xint` packages manipulate are streams of ascii bytes, one per digit. At no time (due to expandability) is it possible to store intermediate results in an arithmetic CPU register; each elementary operation via `\the\numexpr` will output digit tokens (hence as many bytes), not things such as handles to memory locations where some numbers are stored as memory words. The process can never put aside things but can only possibly permute them with upcoming tokens, to use them later, or, via combinations of `\expanded` and `\unexpanded` or some other more antiquated means grab some tokens and shift the expansion to some distant locations to later come back. The process is a never-ending one-dimensional one...

## 2.15. Known bugs/features (last updated at 1.4n)

`\xinteval{\xintLength{\par\par\par}}` complains about a Runaway argument:

`\xintLength{\par\par\par}` has no issue as `\xintLength` is a `\long` macro but this is not the case of `\xinteval`. Most macros of a non arithmetic nature in `xintkernel` and `xinttools` are declared `\long` but absolutely none in `xintexpr`, and its dependencies `xint`, etc... As a remark in passing, I could not use the `\TeX` `\item` directly:<sup>23</sup>

```
Runaway argument?
{| \xinteval { \xintLength {
! Paragraph ended before \@item was complete.
<to be read again>
\par
1.4434 \item[{| \xinteval { \xintLength { \par
\par \par } | complains about a Runawa...
```

This is the reason I guess why everything is a priori `\long` in the `\TeX`3 interface except if asked for otherwise (as far as I know).

<sup>23</sup> For those who wonder my custom `\verb` employs a `\scantokens` approach, so it can be used in the argument of a macro, for example `\footnote {\verb |\verb |}`.



Although most macros are dealing with inputs which can only be with digits and some other character tokens, it would still be quite some work to chase all top-level ones. Besides, in practice, it does help better locate ill-formed input.

**if(100>0,(100,125),(100,128)) breaks my code:** This is a feature. This is a syntax error, as the comma serves to concatenate "oples" (see [subsection 2.13](#)), and parentheses do not create analogs of "tuples", so this input is parsed the same as

```
if(100>0,100,125,100,128)
```

which is an error as `if()` requires exactly three arguments, not five. Use:

```
if(100>0,[100,125],[100,128])
```

which will expand to the "tuple" `[100,125]`.

**\xintdefunc foo(x):= gcd((x>0)?{[x,125]}{[x,128]}); creates a broken function:** Bug. Normally `gcd()` (and other multi-arguments functions) work both with open lists of arguments or bracketed lists ("nutples") and the above syntax would work perfectly fine in numerical context. But the presence of the `?` breaks in `\xintdefunc` context the flexibility of `gcd()`.

Currently working alternatives:

```
\xintdefunc foo(x) := gcd(if(x>0, [x,125], [x,128]));
\xintdefunc foo(x) := if(x>0, gcd(x,125), gcd(x,128));
\xintdefunc foo(x) := if(x>0, gcd([x,125]), gcd([x,128]));
\xintdefunc foo(x) := gcd((x>0)?{x,125}{x,128});
\xintdefunc foo(x) := (x>0)?{gcd(x,125)}{gcd(x,128)};
\xintdefunc foo(x) := (x>0)?{gcd([x,125])}{gcd([x,128])};
```

The same problem will arise with an `??` nested inside `gcd()` or similar functions, in an `\xintdefunc`.

**\xinteval{0^-0.5} says "0 raised to power -1"** Feature. Half integer exponents are handled via a square-root extraction, so here `xintexpr` wanted to first raise `0` to power `-1`, as reported.

**Comparison operator == crashes with nutples** Not yet implemented...

**I liked the "broadcasting" `[1..10]^10` syntax, but it was removed at 1.4** Patience... `seq(x^10,x=1..10)` is alternative (add external `[..]` to get a nutple).

**1e\numexpr 5+2\relax crashes** Not clear yet if bug or feature. The syntax accepted in the scientific part is limited, and failure is expected: hitting a `\numexpr` when parsing a number triggers insertion of a tacit multiplication and then `1e` is missing the scientific exponent. The same happens with `1e(2+3)`. Use syntax such as `1e\the\numexpr5+2\relax`, or `1e\xinteval{5+2}` (although here this relies on output format of `\xinteval` using integer notation with no decoration in this case).

**seq(1e-i,i=1..5) crashes** Not clear if bug or feature. Use `seq(1e\xinteval{-i},i=1..5)` or, as a possibly faster way `seq(1e\xintiieval{-i},i=\xintiexpr1..5\relax)`.

**omit/abort if nested and not last in the sub-expression cause a crash** For example `seq(subs((i)?{i}{abort},t=i)+10, i=-2, -1, 0, 1)` crashes, due to the presence of the `+10`. This is a long-standing limitation, applying ever since `omit/abort` were added to the syntax at 1.1. Even without the `+10` the nested case was broken by a 1.4 regression and got fixed only at 1.4h. The non-nested case `seq((i)?{i}{abort}+10, i=-2, -1, 0, 1)` works and the "must be last in expression if nested" limitation is currently considered a feature.

**\xintdefunc X(a,k)= add(n^k, n=1..a); creates a broken function:** Bug. Sadly `\xintdefunc` has problems. When it does not work, use `\xintNewFunction` to define the function. Examples:

```
\xintNewFunction{myX}[2]{add(n^#2, n=1..#1)}\xinteval{myX(10,7)}\newline
\xintNewFunction{myY}[2]{mul(1 - j/#1, j=1..#2-1)}\xintfloateval{myY(10,10)}
18080425
0.00036288
```

**seq([i,i\string ^2], i=1..10) crashes with Ooops, looks like we are missing a ]. Aborting!** Bug. The cause is that the square brackets do not hide the comma from `seq()` parsing. Contrarily to what happens with parentheses, there is no balancing mechanism for square brackets. Work-arounds: either use an extra pair of parentheses `seq(([i,i^2]), ...)` or hide the inner comma within braces `seq([i{,}i^2], ...)`.

**subs([x,x^2],x=3); crashes with Ooops, looks like we are missing a ]. Aborting!** Bug. Same cause as previous one. Workaround: use parentheses `subs([x,x^2]),x=3` or curly braces `subs({[x,x^2]},x=3)`.

```
\xinteval{subs({[x^10,x^20,x^30]}, x=17)}
[2015993900449, 4064231406647572522401601, 8193465725814765556554001028792218849]
```

**seq([x,2x,3x],x=3..5); crashes with Ooops, looks like we are missing a ]. Aborting!** Bug. Same cause as previous one. The workaround is again to use braces to hide the inner commas.

```
\xinteval{seq({[x^10,x^20,x^30]}, x=1, 2, 3)}
[1, 1, 1], [1024, 1048576, 1073741824], [59049, 3486784401, 205891132094649]
```

**iter([1,10^6];[sqrt(@[0]\*@[1]),(@[0]+@[1])/2], i=1..7) also complains with Ooops, looks like we are missing a ]. Aborting!** Bug. Turns out that braces would not do the job here, but parentheses do work:

```
% 32 digits and 8 iterations v----- parentheses added -----v
\xintfloateval{iter([1,10^6];([sqrt(@[0]*@[1]),(@[0]+@[1])/2]), i=1..8)}
[103329.59376570941022723837701642, 103329.59376570941022723837701642]
```

More bugs are known to the author and many more no doubt exist.



### 3. The macros of **xintexpr** (ancient documentation, mostly)

|     |                                                                                                                    |    |     |                                                                                                                                                                                 |    |
|-----|--------------------------------------------------------------------------------------------------------------------|----|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| .1  | The <code>\xintexpr</code> expressions .....                                                                       | 63 | .13 | The <code>\xintthespaceseparated</code> macro ....                                                                                                                              | 72 |
| .2  | <code>\numexpr</code> or <code>\dimexpr</code> expressions, count<br>and dimension registers and variables .....   | 66 | .14 | <code>\xintifboolexpr</code> , <code>\xintifboolfloatexpr</code> ,<br><code>\xintifbooliexpr</code> .....                                                                       | 73 |
| .3  | Catcodes and spaces .....                                                                                          | 66 | .15 | <code>\xintifsgnexpr</code> , <code>\xintifsgnfloatexpr</code> ,<br><code>\xintifsgniexpr</code> .....                                                                          | 73 |
| .4  | Expandability, <code>\xintexpro</code> .....                                                                       | 67 | .16 | The <code>\xintNewExpr</code> , <code>\xintNewIExpr</code> ,<br><code>\xintNewFloatExpr</code> , <code>\xintNewIExpr</code> , and<br><code>\xintNewBoolExpr</code> macros ..... | 73 |
| .5  | <code>\xintDigits*</code> , <code>\xintSetDigits*</code> .....                                                     | 68 | .17 | Analogies and differences of <code>\xintiexpr</code><br>with <code>\numexpr</code> .....                                                                                        | 74 |
| .6  | <code>\xintiexpr</code> , <code>\xinttheiexpr</code> .....                                                         | 68 | .18 | Chaining expressions for expandable algo-<br>rithmics .....                                                                                                                     | 75 |
| .7  | <code>\xintiexpr</code> , <code>\xinttheiexpr</code> .....                                                         | 68 | .19 | When expandability is too much .....                                                                                                                                            | 78 |
| .8  | <code>\xintboolexpr</code> , <code>\xinttheboolexpr</code> .....                                                   | 69 | .20 | Acknowledgements (2013/05/25) .....                                                                                                                                             | 79 |
| .9  | <code>\xintfloatexpr</code> , <code>\xintthefloatexpr</code> ....                                                  | 70 |     |                                                                                                                                                                                 |    |
| .10 | <code>\xinteval</code> , <code>\xintieval</code> , <code>\xintiieval</code> ,<br><code>\xintfloateval</code> ..... | 70 |     |                                                                                                                                                                                 |    |
| .11 | Using an expression parser within another<br>one .....                                                             | 71 |     |                                                                                                                                                                                 |    |
| .12 | The <code>\xintthecoords</code> macro .....                                                                        | 71 |     |                                                                                                                                                                                 |    |

The **xintexpr** package was first released with version 1.07 (2013/05/25) of the **xint bundle**. It was substantially enhanced with release 1.1 from 2014/10/28.

The 1.4 release from 2020/01/31 maintains the same general architecture but needed adapting all the code base for the switch from `\csname` to `\expanded` techniques. On this occasion the mechanism for defining functions was substantially strengthened. The parser core mechanisms were improved too.

The package loads automatically **xintfrac** and **xinttools**.

This section should be trimmed to contain only information not already covered in [section 2](#).

[source](#)

#### 3.1. The `\xintexpr` expressions

★ An **xintexpression** is a construct `\xintexpr<expandable_expression>\relax` where the expandable expression is read and completely expanded from left to right.

An `\xintexpr... \relax` must end in a `\relax` (which will be absorbed). Contrarily to a `\numexpr` expression, it is printable as is without a prefix `\the` or `\number` (don't use them with `\xintexpr` this will raise an error).

But one can use `\xintthe` prefix if one does need the explicit digits and other characters as in the final typesetted result.

As an alternative and equivalent syntax to

```
\xintexpr round(<expression>, D)\relax
```

there is

```
\xintiexpr [D] <expression> \relax
```

For  $D > 0$  this produces a decimal number with  $D$  figures after the decimal mark, which is the rounding of the expression. For  $D = 0$  the rounding to an integer is produced. For  $D < 0$  (and this was changed at 1.4f), the rounded quotient of the expression by  $1e|D|$  is produced.

- the expression may contain arbitrarily many levels of nested parenthesized sub-expressions,
- the expression may contain explicitly or from a macro expansion a sub-expression `\xintexpr... \relax`, which itself may contain a sub-expressions etc. . .
- to let sub-contents evaluate as a sub-unit it should thus be either
  1. parenthesized,
  2. or a sub-expression `\xintexpr... \relax`.

- to use an expression as argument to macros from `xintfrac`, or more generally to macros which expand their arguments, one must use the `\xinttheexpr...\relax` or `\xintthe\xintexpr...\relax` forms.
- one should not use `\xintthe\xintexpr...\relax` as a sub-constituent of another expression but only the `\xintexpr...\relax` form which is more efficient in this context.
- each `xintexpression`, whether prefixed or not with `\xintthe`, is completely expandable and obtains its result in two expansion steps.

The information now following is possibly in need of updates.

- An expression is built the standard way with opening and closing parentheses, infix operators, and (big) numbers, with possibly a fractional part, and/or scientific notation (except for `\xintiexpr` which only admits big integers). All variants work with comma separated expressions. On output each comma will be followed by a space. A decimal number must have digits either before or after the decimal mark.
- As everything gets expanded, the characters `.`, `+`, `-`, `*`, `/`, `^`, `!`, `&`, `|`, `?`, `:`, `<`, `>`, `=`, `(`, `)`, `"`, `]`, `[`, `@` and the comma `,` should not (if used in the expression) be active.

New with  
1.4n

- Babel-activated characters (for example `!`, `?`, `;` and `:` with French) are not a problem.
- If the character is active due to some other mechanism, prefix it with `\string`.
- A few syntax elements involving the comma, the equal sign and the closing parenthesis are implemented using delimited macros. They are not allowed to be catcode active (even via Babel) and `\string` will not work. Use then `\xintexprSafeCatcodes`, see next.

One can use `\xintexprSafeCatcodes` to reset all characters potentially needed by `\xintexpr` to their standard catcodes and `\xintexprRestoreCatcodes` then restores the former status.

Note that this is what `\xintdefvar` and `\xintdeffunc` do automatically. Expandable `\xintfloateval` et al. can't do that.

- Count registers and `\numexpr`-essions are accepted (LaTeX's counters can be inserted using `\value`) natively without `\the` or `\number` as prefix. Also dimen registers and control sequences, skip registers and control sequences (TeX's lengths), `\dimexpr`-essions, `\glueexpr`-essions are automatically unpacked using `\number`, discarding the stretch and shrink components and giving the dimension value in `sp` units (1/65536th of a TeX point). Furthermore, tacit multiplication is implied, when the (count or dimen or glue) register or variable, or the (`\numexpr` or `\dimexpr` or `\glueexpr`) expression is immediately prefixed by a (decimal) number. See [subsection 2.8](#) for the complete rules of tacit multiplication.

- With a macro `\x` defined like this:

```
\def\x {\xintexpr \a + \b \relax} or \edef\x {\xintexpr \a+\b\relax}
```

one may then do `\xintthe\x`, either for printing the result on the page or to use it in some other macros expanding their arguments. The `\edef` does the computation immediately but keeps it in a protected form. Naturally, the `\edef` is only possible if `\a` and `\b` are already defined. With both approaches the `\x` can be inserted in other expressions, as for example (assuming naturally as we use an `\edef` that in the 'yet-to-be computed' case the `\a` and `\b` now have some suitable meaning):

```
\edef\y {\xintexpr \x^3\relax}
```

- There is also `\xintboolexpr ... \relax` and `\xinttheboolexpr ... \relax`.
- See also `\xintifboolexpr` ([subsection 3.14](#)) and the `bool()` and `togl()` functions in [section 2](#). Here is an example. Well in fact the example ended up using only `\xintboolexpr` so it was modified to use `\xintifboolexpr`.

```

\xintdeffunc A(p,q,r) = p && (q || r) ;
\xintdeffunc B(p,q,r) = p || (q && r) ;
\xintdeffunc C(p,q,r) = xor(p, q, r) ;

\centeredline{\normalcolor
\begin{tabular}{ccrccl}
  \xintfor* #1 in {{False}}{True}} \do {%
  \xintfor* #2 in {{False}}{True}} \do {%
  \xintfor* #3 in {{False}}{True}} \do {%
    #1 &AND &(#2 &OR  &#3)&is&\textcolor[named]{OrangeRed}
      {\xintifboolexpr{A(#1,#2,#3)}{true}{false}}\\
    #1 &OR  &(#2 &AND &#3)&is&\textcolor[named]{OrangeRed}
      {\xintifboolexpr{B(#1,#2,#3)}{yes}{no}}\\
    #1 &XOR &#2 &XOR &#3 &is&\textcolor[named]{OrangeRed}
      {\xintifboolexpr{C(#1,#2,#3)}{oui}{non}}\\
  }}}
\end{tabular}%
}

```

|       |     |        |     |        |    |       |
|-------|-----|--------|-----|--------|----|-------|
| False | AND | (False | OR  | False) | is | false |
| False | OR  | (False | AND | False) | is | no    |
| False | XOR | False  | XOR | False  | is | non   |
| False | AND | (False | OR  | True)  | is | false |
| False | OR  | (False | AND | True)  | is | no    |
| False | XOR | False  | XOR | True   | is | oui   |
| False | AND | (True  | OR  | False) | is | false |
| False | OR  | (True  | AND | False) | is | no    |
| False | XOR | True   | XOR | False  | is | oui   |
| False | AND | (True  | OR  | True)  | is | false |
| False | OR  | (True  | AND | True)  | is | yes   |
| False | XOR | True   | XOR | True   | is | non   |
| True  | AND | (False | OR  | False) | is | false |
| True  | OR  | (False | AND | False) | is | yes   |
| True  | XOR | False  | XOR | False  | is | oui   |
| True  | AND | (False | OR  | True)  | is | true  |
| True  | OR  | (False | AND | True)  | is | yes   |
| True  | XOR | False  | XOR | True   | is | non   |
| True  | AND | (True  | OR  | False) | is | true  |
| True  | OR  | (True  | AND | False) | is | yes   |
| True  | XOR | True   | XOR | False  | is | non   |
| True  | AND | (True  | OR  | True)  | is | true  |
| True  | OR  | (True  | AND | True)  | is | yes   |
| True  | XOR | True   | XOR | True   | is | oui   |

- See also `\xintifsgnexpr`.
- There is `\xintfloatexpr ... \relax` where the algebra is done in floating point approximation (also for each intermediate result). Use the syntax `\xintDigits:=N\relax` to set the precision. Default: 16 digits.

```
\xintthefloatexpr 2^100000\relax: 9.990020930143845e30102
```

The square-root operation can be used in `\xintexpr`, it is computed as a float with the precision set by `\xintDigits` or by the optional second argument:

```

\xinttheexpr sqrt(2,60)\relax\newline
Here the [60] is to avoid truncation to |\xinttheDigits| of precision on output.
\newline

```

```
\printnumber{\xintthefloatexpr [60] sqrt(2,60)\relax}
```

```
1.41421356237309504880168872420969807856967187537694807317668
```

Here the [60] is to avoid truncation to `\xinttheDigits` of precision on output.

```
1.41421356237309504880168872420969807856967187537694807317668
```

Floats are quickly indispensable when using the power function, as exact results will easily have hundreds, even thousands of digits.

```
\xintDigits:=48\relax \xintthefloatexpr 2^100000\relax
```

```
9.99002093014384507944032764330033590980429139054e30102
```

Only integer and (in `\xintfloatexpr...\relax`) half-integer exponents are allowed.

- if one uses macros within `\xintexpr...\relax` one should obviously take into account that the parser will not see the macro arguments, hence one cannot use the syntax there, except if the arguments are themselves wrapped as `\xinttheexpr...\relax` and assuming the macro `f-expands` these arguments.

### 3.2. `\numexpr` or `\dimexpr` expressions, count and dimension registers and variables

Count registers, count control sequences, dimen registers, dimen control sequences (like `\parindent`), skips and skip control sequences, `\numexpr`, `\dimexpr`, `\glueexpr`, `\fontdimen` can be inserted directly, they will be unpacked using `\number` which gives the internal value in terms of scaled points for the dimensional variables: `1pt = 65536sp` (stretch and shrink components are thus discarded).

Tacit multiplication (see [subsection 2.8](#)) is implied, when a number or decimal number prefixes such a register or control sequence.  $\TeX$  lengths are skip control sequences and  $\TeX$  counters should be inserted using `\value`.

Release 1.2 of the `\xintexpr` parser also recognizes and prefixes with `\number` the `\ht`, `\dp`, and `\wd`  $\TeX$  primitives as well as the `\fontcharht`, `\fontcharwd`, `\fontchardp` and `\fontcharic`  $\varepsilon$ - $\TeX$  primitives.

In the case of numbered registers like `\count255` or `\dimen0` (or `\ht0`), the resulting digits will be re-parsed, so for example `\count255 0` is like 100 if `\the\count255` would give 10. The same happens with inputs such as `\fontdimen6\font`. And `\numexpr 35+52\relax` will be exactly as if 87 as been encountered by the parser, thus more digits may follow: `\numexpr 35+52\relax 000` is like 87000. If a new `\numexpr` follows, it is treated as what would happen when `\xintexpr` scans a number and finds a non-digit: it does a tacit multiplication.

```
\xinttheexpr \numexpr 351+877\relax\numexpr 1000-125\relax\relax{} is the same
as \xinttheexpr 1228*875\relax.
```

```
1074500 is the same as 1074500.
```

Control sequences however (such as `\parindent`) are picked up as a whole by `\xintexpr`, and the numbers they define cannot be extended extra digits, a syntax error is raised if the parser finds digits rather than a legal operation after such a control sequence.

A token list variable must be prefixed by `\the`, it will not be unpacked automatically (the parser will actually try `\number`, and thus fail). Do not use `\the` but only `\number` with a dimen or skip, as the `\xintexpr` parser doesn't understand `pt` and its presence is a syntax error. To use a dimension expressed in terms of points or other  $\TeX$  recognized units, incorporate it in `\dimexpr...\relax`.

Regarding how dimensional expressions are converted by  $\TeX$  into scaled points see also [subsection 8.7](#).

### 3.3. Catcodes and spaces

The main problems are caused by active characters, because `\xintexpr` et al. expand forward whatever comes from token stream; they apply `\string` only in a second step. For example the catcode of `&` from `&&` Boolean disjunction is not really important as long as it is not active, or comment,

or escape... or brace... or ignored... in brief, as long as it is reasonable, and in particular whether @ is of catcode letter or other does not matter.

It is always possible to insert manually the `\string` in the expression before a problematic (but reasonable) character catcode, or even to use `\detokenize` for a big chunk.

*source*

### 3.3.1. `\xintexprSafeCatcodes`

Some problems with active characters can be resolved on the fly by prefixing them by `\string` but some aspects of the parsing done by `\xintexpr` involves delimited macros which need the comma, equality sign and closing parenthesis to have their standard catcodes.

So `\xintexprSafeCatcodes` is provided as a utility to set in one go catcodes of many characters to `\xintexpr`-safely compatible values. This is a non-expandable step as it changes catcodes.

`\xintdefvar`, `\xintdeffunc`, et al., use it, and then they restore catcodes to the prior state via `\xintexprRestoreCatcodes`.

*source*

### 3.3.2. `\xintexprRestoreCatcodes`

Restores the catcodes to the state prevailing at the time of the last executed `\xintexprSafeCatcodes` (if located at the same  $\text{\LaTeX}$  environment or  $\text{\TeX}$  grouping level).

Prior to 1.4k, in a situation like the following:

```
\xintexprSafeCatcodes
...stuff possibly changing catcodes
\xintexprSafeCatcodes
...stuff possibly changing catcodes
\xintexprSafeCatcodes
...stuff possibly changing catcodes
\xintexprRestoreCatcodes
```

On exit, the catcodes recovered their status as prior to the *first* `\xintexprSafeCatcodes`. Since 1.4k, they are set to what they were prior to the *last* `\xintexprSafeCatcodes`, i.e. the mechanism is now similar to a ``last in, first out'' stack.

Note that no global assignments are made so the behaviour can be modified by usage of  $\text{\TeX}$  groups or  $\text{\LaTeX}$  environments: e.g. if an `\xintexprSafeCatcodes` is issued inside a  $\text{\LaTeX}$  environment it does not have to be paired by `\xintexprRestoreCatcodes` explicitly, the catcode scope is limited by the environment.

Spaces inside an `\xinttheexpr...\relax` should mostly be innocuous (except inside macro arguments).

`\xintexpr` and `\xinttheexpr` are for the most part agnostic regarding catcodes, but the characters in the expression should not be "active" (except on purpose) as everything is expanded along the way, and `\xintexpr` will choke on typesetting related commands. One can (in almost all cases) use `\string` to prefix a problematic character.

New with  
1.4n

Babel-activated characters are not a problem.

Digits, slash, square brackets, minus sign, in the output from an `\xinttheexpr` are all of catcode 12. For `\xintthefloatexpr` the 'e' in the output has its standard catcode ``letter''.

## 3.4. Expandability, `\xintexpr`

As is the case with all other package macros `\xintexpr` *f-expands* (in two steps) to its final (some-what protected) result; and `\xinttheexpr` *f-expands* (in two steps) to the chain of digits (and

possibly minus sign -, decimal mark ., fraction slash /, scientific e, square brackets [, ]) representing the result.

The once expanded `\xintexpr` is `\romannumeral0\xintexpr`. And there are similarly `\xintiexpr`, `\xintiipro` and `\xintfloatexpr`. For an example see [subsection 3.18](#).

An expression can only be legally terminated by a `\relax` token, which will be absorbed. This token may arise from expansion, it does not have to be immediately visible.

It is quite possible to nest expressions among themselves; for example, if one needs inside an `\xintiexpr...\relax` to do some computations with fractions, rounding the final result to an integer, one just has to insert `\xintiexpr...\relax`. The functioning of the infix operators will not be in the least affected from the fact that the outer ``environment'' is the `\xintiexpr` one.

[source](#)

[source](#)

### 3.5. `\xintDigits*`, `\xintSetDigits*`

These starred variants of `\xintDigits` and `\xintSetDigits` execute `\xintreloadxinttrig` and `\xintreloadxintlog`.

[source](#)

[source](#)

### 3.6. `\xintiexpr`, `\xinttheiexpr`

**x ★** Equivalent to doing `\xintexpr round(...)\relax` (more precisely, `round` is applied to each leaf item of the `ople` independently of its depth).

Intermediate calculations are exact, only the final output gets rounded. Half integers are rounded towards  $+\infty$  for positive numbers and towards  $-\infty$  for negative ones.

An optional parameter `D` within brackets, immediately after `\xintiexpr` is allowed: it instructs (for `D>0`) the expression to do its final rounding to the nearest value with that many digits after the decimal mark, i.e. `\xintiexpr [D] <expression>\relax` is equivalent (in case of a single expression) to `\xintexpr round(<expression>, D)\relax`.

`\xintiexpr [0] ...` is the same as `\xintiexpr ...` and rounds to an integer.

The case of negative `D` gives quantization to an integer multiple of  $1e-D$ . This was modified at [1.4f](#) and the produced value is now the rounded quotient by  $1e-D$  (i.e. no trailing zeros nor scientific exponent in the output).

If truncation rather than rounding is needed one can use `\xintexpr trunc(...)\relax` for truncation to an integer or `\xintexpr trunc(...,D)\relax` for quantization to an integer multiple or  $1eD$  (if `D>0`, for `D<0` the analog would be `trunc(...)/1e-D`). But this works only for a single scalar value.

When defining a macro doing something such as `\xintiexpr #1\relax`, it is recommended to rather use `\xintiexpr\empty #1\relax`, as the `#1` may start with a `[` which without the `\empty` would be interpreted by `\xintiexpr` as the start of the optional `[D]`.

[source](#)

[source](#)

### 3.7. `\xintiipro`, `\xinttheiipro`

**x ★** This variant does not know fractions. It deals almost only with long integers. Comma separated lists of expressions are allowed.

It maps `/` to the rounded quotient. The operator `//` is, like in `\xintexpr...\relax`, mapped to truncated division. The Euclidean quotient (which for positive operands is like the truncated quotient) was, prior to release [1.1](#), associated to `/`. The function `quo(a,b)` can still be employed.

The `\xintiipro`-essions use the ``ii'` macros for addition, subtraction, multiplication, power, square, sums, products, Euclidean quotient and remainder.

The `floor()` and `ceil()` functions are available in this integer only parser, with arguments allowed to be decimal numbers. But for example `floor(5/3)` returns the perhaps counter intuitive value `2`, because `/` does the rounded division to an integer. To force the parser to allow fractions and not interpret `/` as integer-only operator, uses `qfrac()` as in the examples below.

Also `round()` and `trunc()` are allowed in `\xintiexpr`-essions: they are mapped to `\xintiRound` and `\xintiTrunc` which explains how they behave with respect to their optional second argument.

```
\xinttheiiexpr 5/3, round(5/3,3), trunc(5/3,3), trunc(\xintDiv {5}{3},3),
trunc(\xintRaw {5/3},3)\relax{} are problematic, but
%
\xinttheiiexpr 5/3, round(qfrac(5/3),3), trunc(qfrac(5/3),3), floor(qfrac(5/3)),
ceil(qfrac(5/3))\relax{} work!
```

2, 2000, 2000, 2000, 2000 are problematic, but 2, 1667, 1666, 1, 2 work!

Scientific notation is not accepted on input for operations, but can be wrapped in the `num()` or `round()` functions. They will use as many zeroes as necessary on output.

```
% This illustrates output can only use pure integer notation:
```

[illegible]

```
% This should (as num truncates) compute 13456+10000:
```

```
\xinttheiexpr num(13.4567e3)+num(10000123e-3)\relax
```

The `reduce` function is not available and will raise an error. The `frac` function also. The `sqrt` function is mapped to `\xintiiSqrt` which gives a truncated square root. The `sqrtr` function is mapped to `\xintiiSqrtR` which gives a rounded square root.

One can use the Float macros if one is careful to use `num`, or `round` etc. . . on their output.

```
\xinttheiiexpr \xintFloatSqrt [20]{2},
               \xintFloatSqrt [20]{3}\relax % no operations
```

14142135623730950488[-19], 17320508075688772935[-19] The above went through because no actual operations were carried out. But it is dangerous because the ``printer'` for `\xinttheiiexpr` could choke on such values. By default however it does nothing.

In the next example there will be an addition. So we first apply **round** to get integers (the second argument of **round** and **trunc** tells how many digits from after the decimal mark one should keep.)

```
\xinttheiiexpr round(\xintFloatSqrt [20]{2},19) +
               round(\xintFloatSqrt [20]{3},19)\relax
```

31462643699419723423

The whole point of `\xintiiexpr` is to gain some speed in *integer-only* algorithms, and the above explanations related to how to nevertheless use fractions therein are a bit peripheral. We observed (2013/12/18) of the order of 30% speed gain when dealing with numbers with circa one hundred digits (1.2: this info may be obsolete).

*source*

*source*

### 3.8. `\xintboolexpr`, `\xinttheboolexpr`

**x ★** Equivalent to doing `\xintexpr ... \relax` and returning **true** if the result does not vanish, and **false** if the result is zero. As `\xintexpr`, this can be used on comma separated lists of expressions, and even bracketed lists.

It can be customized, one only needs to modify the following:

```
\def\xintboolexprPrintOne#1{\xintiifNotZero{#1}{true}{false}}%
```

Not only are **true** and **false** usable in input, also **True** and **False** are pre-declared variables.

There is quirk in case it is used as a sub-expression: the boolean expression needs at least one logic operation else the value is not standardized to 1 or 0, for example we get from

```
\xinttheexpr \xintboolexpr 1.23\relax\relax\newline
```



### 1.23

which is to be compared with

```
\xinttheboolexpr 1.23\relax
true
```

[source](#)

[source](#)

## 3.9. \xintfloatexpr, \xintthefloatexpr

★ `\xintfloatexpr... \relax` is a variant of `\xintexpr... \relax` which does floating point operations. The target precision for the computation is from the current setting of `\xintDigits`. Comma separated lists of expressions are allowed.

An optional parameter within brackets `[Q]` is allowed at the very start of the expression:

- if positive it instructs the macro to round the result to that many digits of precision. It thus makes sense to employ it only if this parameter is less than the `\xinttheDigits` precision.
- if negative it means to trim off that many digits (of course, in the sense of rounding the values to shorter mantissas). Don't use it to trim all digits (or more than all)!

Since 1.2f all float operations first round their arguments; a parsed number is not rounded prior to its use as operand to such a float operation.

```
\xintDigits:=36\relax
\xintthefloatexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax
0.00564487459334466559166166079096852897
\xintthefloatexpr\xintexpr (1/13+1/121)*(1/179-1/173)/(1/19-1/18)\relax\relax
0.00564487459334466559166166079096852912
```

The latter is the rounding of the exact result. The former one has its last three digits wrong due to the cumulative effect of rounding errors in the intermediate computations, as compared to exact evaluations.

I recall here from [subsection 8.2](#) that with release 1.2f the float macros for addition, subtraction, multiplication and division round their arguments first to `P` significant places with `P` the asked-for precision of the output; and similarly the power macros and the square root macro. This does not modify anything for computations with arguments having at most `P` significant places already.

When defining a macro doing something such as `\xintfloatexpr #1\relax`, it is recommended to rather use `\xintfloatexpr\empty #1\relax`, as the `#1` may start with a `[` which without the `\empty` would be interpreted by `\xintfloatexpr` as the start of the optional `[Q]`.

[source](#)

[source](#)

[source](#)

[source](#)

## 3.10. \xinteval, \xintieval, \xintiieval, \xintfloateval

★ `\xinteval` is an *f-expandable* macro which is basically defined in such a way that `\xinteval{<expression>}` behaves like `\xinttheexpr{<expression>}\relax`. It expands completely in two steps and delivers its output using digits, the dot `.` as decimal separator, the letter `e` for scientific notation, the slash `/` for fractions, as well as commas in case of multi-items expression and square brackets `[` and `]` for nesting.

★ `\xintieval` is similarly related to `\xinttheiexpr`. It admits an optional argument `[D]` which may be located in the expected location from conventions of `\TeX` macros with optional argument, but had been long constrained (until 1.4k) to be inside the braces at the start of the expression.

```
\xintieval[7]{355/113} = \xintieval{[7]355/113}
3.1415929 = 3.1415929
```



When defining a macro doing something such as `\xintieval{#1}`, it is recommended to rather use `\xintieval{\empty #1}`, as the `#1` may start with a `[` which without the `\empty` would be interpreted by `\xintieval` as the start of the optional `[D]`.

`\xintiieval` is similarly related to `\xinttheiexpr`.  
`\xintfloateval` is similarly related to `\xintthefloatexpr`. It admits an optional argument `[Q]` which may be located either outside (since 1.4k) or inside the braces.  
`\xintfloateval [7]{355/113} = \xintfloateval{[7] 355/113}`  
`3.141593 = 3.141593`

When negative, the optional argument tells how many digits to remove from the prevailing precision:

`\xintfloateval[-2]{355/113}=`  
`\xintfloateval{[-2]355/113}` has `\xinttheDigits` minus 2 digits.  
`3.1415929203540 = 3.1415929203540` has 16 minus 2 digits.

When defining a macro doing something such as `\xintfloateval{#1}`, it is recommended to rather use `\xintfloateval{\empty #1}`, as the `#1` may start with a `[` which without the `\empty` would be interpreted by `\xintfloateval` as the start of the optional `[Q]`.

### 3.11. Using an expression parser within another one

This was already illustrated before. In the following:

`\xintfloatexpr \xintexpr add(1/i, i=1234..1243)\relax ^100\relax`  
`5.136088460396579e-210`, the inner sum is computed exactly. Then it will be rounded to `\xinttheDigits` significant digits, and then its power will be evaluated as a float operation. One should avoid the `"\xintthe"` parsers in inner positions as this induces digit by digit parsing of the inner computation result by the outer parser. Here is the same computation done with floats all the way:

`\xintfloatexpr add(1/i, i=1234..1243)^100\relax`  
`5.136088460396643e-210`

Not surprisingly this differs from the previous one which was exact until raising to the 100th power.

The fact that the inner expression occurs inside a bigger one has nil influence on its behaviour. There is the limitation though that the outputs from `\xintexpr` and `\xintfloatexpr` can not be used directly in `\xinttheiexpr` integer-only parser. But one can do:

`\xintiexpr round(\xintfloatexpr 3.14^10\relax)\relax % or trunc`  
`93174`

*source*

### 3.12. The `\xintthecoords` macro

It converts (in two expansion steps) the expansion result of `\xintfloatexpr` (or `\xintexpr` or `\xintiexpr`) into the `(a, b) (c, d) ...` format for list of coordinates as expected by the `TikZ coordinates` syntax.

```
\begin{figure}[htbp]
\centering\begin{tikzpicture}[scale=10]\xintDigits:=8\relax
\clip (-1.1,-.25) rectangle (.3,.25);
\draw [blue] (-1.1,0)--(1,0);
\draw [blue] (0,-1)--(0,+1);
\draw [red] plot[smooth] coordinates {%
%% \xintthecoords converts output of next expression into the
```

```
%% (x1, y1) (x2, y2) ...
%% format
\xintthecoords\xintfloatexpr
%% This syntax -1+[0..4]/2 is currently dropped at xint 1.4
%% seq((x^2-1,mul(x-t,t=-1+[0..4]/2)),x=-1.2..[0.1]..1.2)\relax
%% Use this:
seq((x^2-1,mul(x-t,t=seq(-1+u/2, u=0..4))),x=-1.2..[0.1]..1.2)
\relax
};
\end{tikzpicture}
\caption{Coordinates with \csbxint{thecoords}.}
\end{figure}
```

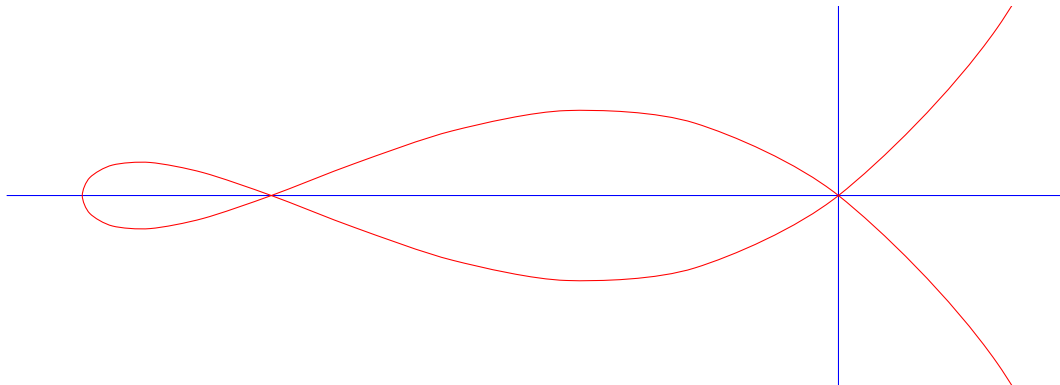


Figure 2: Coordinates with `\xintthecoords`.

**Unstable!** It is currently undecided how `\xintthecoords` should handle bracketed data. Currently, it (or `TikZ`) will break if the input contains nested structures. One can use it with `flat()` which removes all nesting. And in combination with `zip()` it is easy to plot data given by some mechanism in separate lists of x- and y-coordinates (see an example in next section)

[source](#)

### 3.13. The `\xintthespaceseparated` macro

It converts (in two expansion steps) the expansion result of `\xintfloatexpr` (or `\xintexpr` or `\xintiexpr`) into the space separated format suitable for usage with `PS-Tricks \listplot` macro.

Here is for example some syntax (the replacement text of `\foo`, which is used here only to show that indeed complete expansion is attained in two steps) which can be used as argument to `\listplot`. Using 4 fractional decimal digits is sufficient when unit is the centimeter (it gives a fixed point precision of one micron, amply enough for plots...).

```
\oodef\foo{%
\xintthespaceseparated
\xintiexpr[4]\xintfloatexpr seq((i, log10(i)), i=1..[0.5]..10)\relax\relax
}\meaning\foo
```

```
macro:->1.0000 0 1.5000 0.1761 2.0000 0.3010 2.5000 0.3979 3.0000 0.4771 3.5000 0.5441 4.0000
0.6021 4.5000 0.6532 5.0000 0.6990 5.5000 0.7404 6.0000 0.7782 6.5000 0.8129 7.0000 0.8451 7.5000
0.8751 8.0000 0.9031 8.5000 0.9294 9.0000 0.9542 9.5000 0.9777 10.0000 1.0000
```

Here we don't really need the inner `\xintfloatexpr...\relax` because the `log10()` function works the same in the exact parser `\xintexpr` but in general this is recommended.

**Unstable!** It is currently undecided how `\xintthespaceseparated` should handle bracketed data. Currently, it (or `\listplot`) will break if the input contains nested structures. One can use it with `flat()`

which removes all nesting. And in combination with `zip()` it is easy to plot data given by some mechanism in separate lists of x- and y-coordinates.

```
% let's imagine we have something like this
\def\Xcoordinates{1, 3, 5, 7, 9}
\def\Ycoordinates{1, 9, 25, 49, 81}
% then:
|\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax|
is suitable to use as argument to |\listplot|, as it expands to
\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax
\xintthespaceseparated\xintexpr flat(zip([\Xcoordinates], [\Ycoordinates]))\relax is suitable
to use as argument to \listplot, as it expands to 1 1 3 9 5 25 7 49 9 81
```

[source](#)[source](#)[source](#)

### 3.14. `\xintifboolexpr`, `\xintifboolfloatexpr`, `\xintifboolliexpr`

**xnn ★** `\xintifboolexpr{<expr>}{<YES>}{<NO>}` does `\xinttheexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non-zero or zero. Thus one can read *if bool expr* as meaning *if not zero*:

if `<expr>`-ession does not vanish do `<YES>` else do `<NO>`

The expression is not limited to using only comparison operators and Boolean logic (`<`, `>`, `==`, `!=`, `&&`, `||`, `all()`, `any()`, `xor()`, `bool()`, `togl()`, ...), it can be the most general computation.

**xnn ★** `\xintifboolfloatexpr{<expr>}{<YES>}{<NO>}` does `\xintthefloatexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non zero or zero.

**xnn ★** `\xintifboolliexpr{<expr>}{<YES>}{<NO>}` does `\xinttheiiexpr<expr>\relax` and then executes the `<YES>` or the `<NO>` branch depending on whether the outcome was non zero or zero.

The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

[source](#)[source](#)[source](#)

### 3.15. `\xintifsgnexpr`, `\xintifsgnfloatexpr`, `\xintifsgnliexpr`

**xnnn ★** `\xintifsgnexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintexpression` and chooses the branch corresponding to its sign.

**xnnn ★** `\xintifsgnfloatexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintfloatexpression` and chooses the branch corresponding to its sign.

**xnnn ★** `\xintifsgnliexpr{<expr>}{<<0>}{<=0>}{<>0>}` evaluates the `\xintiexpression` and chooses the branch corresponding to its sign.

The expression argument must be a single one, comma separated sub-expressions will cause low-level errors.

[source](#)[source](#)[source](#)[source](#)

### 3.16. The `\xintNewExpr`, `\xintNewIIExpr`, `\xintNewFloatExpr`, `\xintNewIExpr`, and `\xintNewBoolExpr` macros

`\xintNewExpr` macro is used as:

```
\xintNewExpr{<myformula>}[n]{<stuff>}, where
```

- `<stuff>` will be inserted inside `\xinttheexpr . . . \relax`,
- `n` is an integer between zero and nine, inclusive, which is the number of parameters of `\myfor` `mula`,
- the placeholders `#1`, `#2`, ..., `#n` are used inside `<stuff>` in their usual rôle,<sup>24 25</sup>
- the `[n]` is mandatory, even for `n=0`.<sup>26</sup>

<sup>24</sup> if `\xintNewExpr` is used inside a macro, the `#`'s must be doubled as usual. <sup>25</sup> the `#`'s will in practice have their usual catcode, but category code other `#`'s are accepted too. <sup>26</sup> there is some use for `\xintNewExpr[0]` compared to an `\edef` as `\xintNewExpr` has some built-in catcode protection.

- the macro `\myformula` is defined without checking if it already exists,  $\TeX$  users might prefer to do first `\newcommand*\myformula {}` to get a reasonable error message in case `\myformula` already exists,
- the protection against active characters is done automatically (as long as the whole thing has not already been fetched as a macro argument and the catcodes correspondingly already frozen).

It (if it succeeds) will be a completely expandable macro entirely built-up using `\xintAdd`, `\xintSub`, `\xintMul`, `\xintDiv`, `\xintPow`, etc. . . as corresponds to the expression written with the infix operators. Macros created by `\xintNewExpr` can thus be nested.

```
\xintNewFloatExpr \FA [2]{(#1+#2)^10}
\xintNewFloatExpr \FB [2]{sqrt(#1*#2)}
\begin{enumerate}[nosep]
  \item \FA {5}{5}
  \item \FB {30}{10}
  \item \FA {\FB {30}{10}}{\FB {40}{20}}
\end{enumerate}
```

1. `1e10`
2. `17.32050807568877`
3. `3.891379490446502e16`

The documentation is much shortened here because `\xintNewExpr` and `\xintdeffunc` are very much related one with the other.

#### ATTENTION!

The original spirit of `\xintNewExpr` was to define a (possibly very big) macro using only `xintfrac`, and this means in particular that it must be used only with arguments compatible with the `xintfrac` input format.

Thus an `\xintexpr` declared variable has no chance to work, it must be wrapped explicitly in `\xinteval{...}` to be fetched as argument to a macro constructed by `\xintNewExpr`.

They share essentially the same limitations.

Notice though that `\xintNewFloatExpr` accepts and recognizes the optional argument `[Q]` of `\xint-floatexpr`, contrarily to `\xintdeffloatfunc`. Use an `\empty` in case the contents are not known in advance.

Historical note: prior to 1.4, `xintexpr` used a `\csname..\endcsname` encapsulation technique which impacted the string pool memory. The `\xintNewExpr` was designed as a method to pre-parse the expression and produce one single, gigantic, nested usage of the relevant `xintfrac` macros. This way, only those macros were expanded which had nil impact on the  $\TeX$  string pool.

Later on it was found that this mechanism could be employed to define functions. Basically underneath 98% of `\xintNewExpr` and `\xintdeffunc` are using the same shared code.

*source*

### 3.17. Analogies and differences of `\xintiexpr` with `\numexpr`

`\xintiexpr..\relax` is a parser of expressions knowing only (big) integers. There are, besides the enlarged range of allowable inputs, some important differences of syntax between `\numexpr` and `\xintiexpr` and variants:

- Contrarily to `\numexpr`, the `\xintiexpr` parser will stop expanding only after having encountered (and swallowed) a mandatory `\relax` token.
- In particular, spaces between digits (and not only around infix operators or parentheses) do not stop `\xintiexpr`, contrarily to the situation with `numexpr`: `\the\numexpr 7 + 3 5\relax`

expands (in one step)<sup>27</sup> to `105\relax`, whereas `\xintthe\xintiexpr 7 + 3 5\relax` expands (in two steps) to `42`.<sup>28</sup>

- Inside an `\edef`, an expression `\xintiexpr...\relax` get fully evaluated, whereas `\numexpr` without `\the` or `\number` prefix would not, if not itself embedded in another `\the\numexpr` or similar context.
- (ctd.) The private format to which `\xintiexpr...\relax` (et al.) evaluates may use `\xintthe` prefix to turn into explicit digits, (for example in arguments to some macros which expand their arguments). The `\the` TeX primitive prefix would not work here.
- (ctd.) One can embed a `\numexpr...\relax` (with its `\relax!`) inside an `\xintiexpr...\relax` without `\the` or `\number`, but the reverse situation requires usage of `\xintthe` or `\xinteval` user interface,
- `\the\numexpr -(1)\relax` is illegal. In contrast `\xinttheiexpr -(1)\relax` is perfectly legal and gives the expected result (what else?).
- `\the\numexpr 2+-(1+1)\relax` is illegal. In contrast `\xinttheiexpr 2+-(1+1)\relax` is legal.
- `\the\numexpr 2\cnta\relax` is illegal (with `\cnta` a `\count` register.) In contrast `\xinttheiexpr 2\cnta\relax` is perfectly legal and will do the tacit multiplication.
- `\the\numexpr` or `\number\numexpr` expands in one step, but `\xintthe\xintiexpr` or `\xinttheiexpr` needs two steps.

### 3.18. Chaining expressions for expandable algorithmics

We will see in this section how to chain `\xintexpr`-essions with `\expandafter`'s, like it is possible with `\numexpr`. For this it is convenient to use `\romannumeral0\xintexpr` which is the once-expanded form of `\xintexpr`, as we can then chain using only one `\expandafter` each time.

For example, here is the code employed for the background of page 2. It computes (expandably, of course!) the 1250th Fibonacci number.

```
\catcode`_ 11
\def\Fibonacci #1{% \Fibonacci{N} computes F(N) with F(0)=0, F(1)=1.
  \expandafter\Fibonacci_a\expandafter
    {\the\numexpr #1\expandafter}\expandafter
    {\romannumeral0\xintiexpr 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpr 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpr 1\expandafter\relax\expandafter}\expandafter
    {\romannumeral0\xintiexpr 0\relax}}
%
\def\Fibonacci_a #1{%
  \ifcase #1
    \expandafter\Fibonacci_end_i
  \or
    \expandafter\Fibonacci_end_ii
  \else
    \ifodd #1
      \expandafter\expandafter\expandafter\Fibonacci_b_ii
    \else
      \expandafter\expandafter\expandafter\Fibonacci_b_i
```

<sup>27</sup> The `\numexpr` triggers continued expansion after the space following the `3` to check if some operator like `+` is upstream. But after having found the `5` it treats it as an end-marker. <sup>28</sup> Since 1.21 one can also use the underscore `_` to separate digits for readability of long numbers.

```

\fi
\fi {#1}%
}% * signs are omitted from the next macros, tacit multiplications
\def\Fibonacci_b_i #1#2#3{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr #1/2\expandafter}\expandafter
  {\romannumeral0\xintiexprpro sqr(#2)+sqr(#3)\expandafter\relax
    \expandafter}\expandafter
  {\romannumeral0\xintiexprpro (2#2-#3)#3\relax}%
}% end of Fibonacci_b_i
\def\Fibonacci_b_ii #1#2#3#4#5{\expandafter\Fibonacci_a\expandafter
  {\the\numexpr (#1-1)/2\expandafter}\expandafter
  {\romannumeral0\xintiexprpro sqr(#2)+sqr(#3)\expandafter\relax
    \expandafter}\expandafter
  {\romannumeral0\xintiexprpro (2#2-#3)#3\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiexprpro #2#4+#3#5\expandafter\relax\expandafter}\expandafter
  {\romannumeral0\xintiexprpro #2#5+#3(#4-#5)\relax}%
}% end of Fibonacci_b_ii
%
% code as used on title page:
%\def\Fibonacci_end_i #1#2#3#4#5{\xintthe#5}
%\def\Fibonacci_end_ii #1#2#3#4#5{\xinttheiexpr #2#5+#3(#4-#5)\relax}
%
% new definitions:
\def\Fibonacci_end_i #1#2#3#4#5{{#4}{#5}}% {F(N+1)}{F(N)} in \xintexpr format
\def\Fibonacci_end_ii #1#2#3#4#5%
  {\expandafter
    {\romannumeral0\xintiexprpro #2#4+#3#5\expandafter\relax
      \expandafter}\expandafter
    {\romannumeral0\xintiexprpro #2#5+#3(#4-#5)\relax}}% idem.
% \FibonacciN returns F(N) (in encapsulated format: needs \xintthe for printing)
\def\FibonacciN {\expandafter\xint_secondoftwo\romannumeral-`0\Fibonacci}%
\catcode_`8

```

The macro `\Fibonacci` produces not one specific value  $F(N)$  but a pair of successive values  $\{F(N), F(N+1)\}$  which can then serve as starting point of another routine devoted to compute a whole sequence  $F(N), F(N+1), F(N+2), \dots$ . Each of  $F(N)$  and  $F(N+1)$  is kept in the encapsulated internal `xintexpr` format.

`\FibonacciN` produces the single  $F(N)$ . It also keeps it in the private format; thus printing it will need the `\xintthe` prefix.

Here a code snippet which checks the routine via a `\message` of the first 51 Fibonacci numbers (this is not an efficient way to generate a sequence of such numbers, it is only for validating `\FibonacciN`).

```

\def\Fibo #1.{\xintthe\FibonacciN {#1}}%
\message{\xintloop [0+1] \expandafter\Fibo\xintloopindex.,
  \ifnum\xintloopindex<49 \repeat \xintthe\FibonacciN{50}.}

```

The way we use `\expandafter`'s to chain successive `\xintiexprpro` evaluations is exactly analogous to what is possible with `\numexpr`. The various `\romannumeral0\xintiexprpro` could very well all have been `\xintiexpr`'s but then we would have needed `\expandafter\expandafter\expandafter` each time.

There is a difference though: `\numexpr` does NOT expand inside an `\edef`, and to force its expansion we must prefix it with `\the` or `\number` or `\romannumeral` or another `\numexpr` which is itself prefixed, etc. . . .

But `\xintexpr`, `\xintiexpr`, . . . , expand fully in an `\edef`, with the completely expanded result encapsulated in a private format.

Using `\xintthe` as prefix is necessary to print the result (like `\the` or `\number` in the case

of `\numexpr`), but it is not necessary to get the computation done (contrarily to the situation with `\numexpr`).

Our `\Fibonacci` expands completely under *f-expansion*, so we can use `\fdef` rather than `\edef` in a situation such as

```
\fdef \X {\FibonacciN {100}} ,
```

but it is usually about as efficient to employ `\edef`. And if we want

```
\edef \Y {\FibonacciN{100},\FibonacciN{200}} ,
```

then `\edef` is necessary.

Allright, so let's now give the code to generate  $\{F(N)\}\{F(N+1)\}\{F(N+2)\}\dots$ , using `\Fibonacci` for the first two and then using the standard recursion  $F(N+2)=F(N+1)+F(N)$ :

```
\catcode\_ 11
\def\FibonacciSeq #1#2{%#1=starting index, #2>#1=ending index
  \expandafter\Fibonacci_Seq\expandafter
  {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2-1}%
}%
\def\Fibonacci_Seq #1#2{%
  \expandafter\Fibonacci_Seq_loop\expandafter
  {\the\numexpr #1\expandafter}\romannumeral0\Fibonacci {#1}{#2}%
}%
\def\Fibonacci_Seq_loop #1#2#3#4{% standard Fibonacci recursion
  {#3}\unless\ifnum #1<#4 \Fibonacci_Seq_end\fi
  \expandafter\Fibonacci_Seq_loop\expandafter
  {\the\numexpr #1+1\expandafter}\expandafter
  {\romannumeral0\xintiipro #2+#3\relax}{#2}{#4}%
}%
\def\Fibonacci_Seq_end\fi\expandafter\Fibonacci_Seq_loop\expandafter
  #1\expandafter #2#3#4{\fi {#3}}%
\catcode\_ 8
```

This `\FibonacciSeq` macro is completely expandable but it is not *f-expandable*.

This is not a problem in the next example which uses `\xintFor*` as the latter applies repeatedly full expansion to what comes next each time it fetches an item from its list argument. Thus `\xintFor*` still manages to generate the list via iterated full expansion.

```
\newcounter{myindex}% not "index", which would overwrite theindex environment!
% (many have probably been bitten by this trap)
\tabskip 1ex
\def\Fibxxx{\FibonacciN {30}}%
\setcounter{myindex}{30}%
\vbox{\halign{\bfseries#.\hfil&\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {30}{59}}\do
  {\themymindex &\xintthe#1 &
    \xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\vbox{\halign{\bfseries#.\hfil&\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {60}{89}}\do
  {\themymindex &\xintthe#1 &
    \xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }}%
}\vrule
\vbox{\halign{\bfseries#.\hfil&\hfil &\hfil #\cr
  \xintFor* #1 in {\FibonacciSeq {90}{119}}\do
  {\themymindex &\xintthe#1 &
```



|     |              |        |     |                     |        |      |                           |        |
|-----|--------------|--------|-----|---------------------|--------|------|---------------------------|--------|
| 30. | 832040       | 0      | 60. | 1548008755920       | 0      | 90.  | 2880067194370816120       | 0      |
| 31. | 1346269      | 514229 | 61. | 2504730781961       | 1      | 91.  | 4660046610375530309       | 514229 |
| 32. | 2178309      | 514229 | 62. | 4052739537881       | 1      | 92.  | 7540113804746346429       | 514229 |
| 33. | 3524578      | 196418 | 63. | 6557470319842       | 2      | 93.  | 12200160415121876738      | 196418 |
| 34. | 5702887      | 710647 | 64. | 10610209857723      | 3      | 94.  | 19740274219868223167      | 710647 |
| 35. | 9227465      | 75025  | 65. | 17167680177565      | 5      | 95.  | 31940434634990099905      | 75025  |
| 36. | 14930352     | 785672 | 66. | 27777890035288      | 8      | 96.  | 51680708854858323072      | 785672 |
| 37. | 24157817     | 28657  | 67. | 44945570212853      | 13     | 97.  | 83621143489848422977      | 28657  |
| 38. | 39088169     | 814329 | 68. | 72723460248141      | 21     | 98.  | 135301852344706746049     | 814329 |
| 39. | 63245986     | 10946  | 69. | 117669030460994     | 34     | 99.  | 21892299583455169026      | 10946  |
| 40. | 102334155    | 825275 | 70. | 190392490709135     | 55     | 100. | 354224848179261915075     | 825275 |
| 41. | 165580141    | 4181   | 71. | 308061521170129     | 89     | 101. | 573147844013817084101     | 4181   |
| 42. | 267914296    | 829456 | 72. | 498454011879264     | 144    | 102. | 927372692193078999176     | 829456 |
| 43. | 433494437    | 1597   | 73. | 806515533049393     | 233    | 103. | 1500520536206896083277    | 1597   |
| 44. | 701408733    | 831053 | 74. | 1304969544928657    | 377    | 104. | 2427893228399975082453    | 831053 |
| 45. | 1134903170   | 610    | 75. | 2111485077978050    | 610    | 105. | 3928413764606871165730    | 610    |
| 46. | 1836311903   | 831663 | 76. | 3416454622906707    | 987    | 106. | 6356306993006846248183    | 831663 |
| 47. | 2971215073   | 233    | 77. | 5527939700884757    | 1597   | 107. | 10284720757613717413913   | 233    |
| 48. | 4807526976   | 831896 | 78. | 8944394323791464    | 2584   | 108. | 16641027750620563662096   | 831896 |
| 49. | 7778742049   | 89     | 79. | 14472334024676221   | 4181   | 109. | 26925748508234281076009   | 89     |
| 50. | 12586269025  | 831985 | 80. | 23416728348467685   | 6765   | 110. | 43566776258854844738105   | 831985 |
| 51. | 20365011074  | 34     | 81. | 37889062373143906   | 10946  | 111. | 70492524767089125814114   | 34     |
| 52. | 32951280099  | 832019 | 82. | 61305790721611591   | 17711  | 112. | 114059301025943970552219  | 832019 |
| 53. | 53316291173  | 13     | 83. | 99194853094755497   | 28657  | 113. | 184551825793033096366333  | 13     |
| 54. | 86267571272  | 832032 | 84. | 160500643816367088  | 46368  | 114. | 298611126818977066918552  | 832032 |
| 55. | 139583862445 | 5      | 85. | 259695496911122585  | 75025  | 115. | 483162952612010163284885  | 5      |
| 56. | 225851433717 | 832037 | 86. | 420196140727489673  | 121393 | 116. | 781774079430987230203437  | 832037 |
| 57. | 365435296162 | 2      | 87. | 679891637638612258  | 196418 | 117. | 1264937032042997393488322 | 2      |
| 58. | 591286729879 | 832039 | 88. | 1100087778366101931 | 317811 | 118. | 2046711111473984623691759 | 832039 |
| 59. | 956722026041 | 1      | 89. | 1779979416004714189 | 514229 | 119. | 3311648143516982017180081 | 1      |

Some Fibonacci numbers together with their residues modulo  $F(30)=832040$

```
\xintiiRem{\xintthe#1}{\xintthe\Fibxxx}\stepcounter{myindex}\cr }%
}%
```

This produces the Fibonacci numbers from  $F(30)$  to  $F(119)$ , and computes also all the congruence classes modulo  $F(30)$ . The output has been put in a `float`, which appears above. I leave to the mathematically inclined readers the task to explain the visible patterns...;-).

### 3.19. When expandability is too much

Let's use the macros of [subsection 3.18](#) related to Fibonacci numbers. Notice that the 47th Fibonacci number is `2971215073` thus already too big for  $\text{\TeX}$  and  $\varepsilon\text{-}\text{\TeX}$ .

The `\FibonacciN` macro found in [subsection 3.18](#) is completely expandable, it is even *f-expandable*. We need a wrapper with `\xintthe` prefix

```
\def\xtheFibonacciN{\xintthe\FibonacciN}
```

to print in the document or to use within `\message` (or  $\text{\TeX}$  `typeout`) to write to the log and terminal.

The `\xintthe` prefix also allows its use it as argument to the `xint` macros: for example if we are interested in knowing how many digits  $F(1250)$  has, it suffices to issue `\xintLen {\theFibonacciN {1250}}` (which expands to `261`). Or if we want to check the formula  $\gcd(F(1859), F(1573)) = F(\gcd(1859, 1573)) = F(143)$ , we only need<sup>29</sup>

```
\$ \xintiiGCD{\theFibonacciN{1859}}{\theFibonacciN{1573}}=%
```

<sup>29</sup> The `\xintiiGCD` macro is provided by both the `xintgcd` package (since 1.0) and by the `xint` package (since 1.3d).



```
\theFibonacciN{\xintiigcd{1859}{1573}}$
```

which produces:

```
343358302784187294870275058337 = 343358302784187294870275058337
```

The `\theFibonacciN` macro expanded its `\xintiigcd{1859}{1573}` argument via the services of `\numexpr`: this step allows only things obeying the  $\TeX$  bound, naturally! (but `F(2147483648)` would be rather big anyhow...).

This is very convenient but of course it repeats the complete evaluation each time it is done. In practice, it is often useful to store the result of such evaluations in macros. Any `\edef` will break expandability, but if the goal is at some point to print something to the `dvi` or `pdf` output, and not only to the `log` file, then expandability has to be broken one day or another!

Hence, in practice, if we want to print in the document some computation results, we can proceed like this and avoid having to repeat identical evaluations:

```
\begingroup
\def\A {1859} \def\B {1573}
\edef\X {\theFibonacciN\A} \edef\Y {\theFibonacciN\B}
\edef\GCDAB {\xintiigcd\A\B}\edef\Z {\theFibonacciN\GCDAB}
\edef\GCDXY{\xintiigcd\X\Y}
The identity $\gcd(F(\A),F(\B))=F(\gcd(\A,\B))$ can be checked via evaluation
of both sides: $\gcd(F(\A),F(\B))=\gcd(\printnumber\X,\printnumber\Y)=
\printnumber{\GCDXY} = F(\gcd(\A,\B)) = F(\GCDAB) =\printnumber\Z$.par
% some further computations involving \A, \B, \X, \Y
\endgroup % closing the group removes assignments to \A, \B, ...
% or choose longer names less susceptible to overwrite something.
% Note: there is no LaTeX \newcommand which would be to \edef like
% \newcommand is to \def
```

The identity  $\gcd(F(1859), F(1573)) = F(\gcd(1859, 1573))$  can be checked via evaluation of both sides:  $\gcd(F(1859), F(1573)) = \gcd(14405827913044251198771689151504042869913161495023481014226686367010882725975754947224824377535296194597948692273576288822163093580182640808517753199742569560552943502886158524517372508867364222284929082289524558388949544219265576041299929025565979711337876105452217623490841529979811413199660087517689703410997520079993610707576019520876324584695551467505894985013610208598628752325727241, 24438419251951185733282794597776261998539902481570619232605360900784013394036743212445223278959909515869581103189177976905803274151632595307616686661013725200866754096569888951010022888016831459347310131566517721593249344798634399479371195758766544765827958909282390070313197135548122004938644531329524847747273166471511289078393) = 343358302784187294870275058337 = F(\gcd(1859, 1573)) = F(143) = 343358302784187294870275058337.$

One may legitimately ask the author: why expandability to such extremes, for things such as big fractions or floating point numbers (even continued fractions...) which anyhow can not be used directly within  $\TeX$ 's primitives such as `\ifnum`? Why insist on a concept which is foreign to the vast majority of  $\TeX$  users and even programmers?

I have no answer: it made definitely sense at the start of `xint` (see [subsection 8.13](#)) and once started I could not stop.

### 3.20. Acknowledgements (2013/05/25)

I was greatly helped in my preparatory thinking, prior to producing such an expandable parser, by the commented source of the `l3fp` package, specifically the `l3fp-parse.dtx` file (in the version of April-May 2013; I think there was in particular a text called ```roadmap''` which was helpful). Also the source of the `calc` package was instructive, despite the fact that here for `\xintexpr` the principles are necessarily different due to the aim of achieving expandability.

## 4. The **xinttrig** package

|    |                                        |    |    |                                     |    |
|----|----------------------------------------|----|----|-------------------------------------|----|
| .1 | <code>\xintreloadxinttrig</code> ..... | 80 | .4 | Important implementation notes..... | 82 |
| .2 | Constants.....                         | 80 | .5 | Some example evaluations .....      | 82 |
| .3 | Functions.....                         | 81 |    |                                     |    |

This package provides trigonometric functions for use with **xintexpr**. The sole macro is `\xintreloadxinttrig`.

This package was first included in release 1.3e (2019/04/05) of **xintexpr**. It is automatically loaded by **xintexpr**.

At 1.4e (2021/05/05) the accuracy was significantly increased: formerly the high-level user interface used to define the functions had as consequences that intermediate steps of the computations could not operate with guard digits, and as a result the last two digits were most of the time off (at least the last one). Now, computations are done internally in extended precision, and the accuracy is high up to the last digits, with faithful rounding and high probability of correct rounding. And the maximal number of digits was raised slightly to 62 digits.

At 8 digits a special, faster, mode is used, which is less accurate. But faster.

**Acknowledgements:** I finally decided to release some such functions under friendly pressure of Jürgen GILG and Thomas SÖLL, let them both be thanked here.

Jürgen passed away in 2022. I will miss our friendship which was born and grew from numerous and regular exchanges on topics not limited to this package or even the  $\TeX$  world. Let's now continue to "take care and keep motivated"!

[source](#)

### 4.1. `\xintreloadxinttrig`

The library is loaded automatically by **xintexpr** at start-up. It is then configured for 16 digits.

To work for example with 48 digits, execute `\xintSetDigits*{48}` or `\xintDigits*:=48`; (the ending `;` can be replaced by a `\relax` in case of problems due to it being active, e.g. with  $\LaTeX$  and some languages).



With the non-starred variant `\xintDigits:=48`; it is needed to issue `\xintreloadxinttrig` to recalibrate the functions provided by the library (and the exponential/logarithm functions will only be updated if also `\xintreloadxintlog` is used).

### 4.2. Constants

Their values (with more digits) get incorporated into the trigonometrical functions at the time of their definitions during loading or reloading of the package. They are left free to use, or modified, or `\xintunassignvar`'d, as this will have no impact whatsoever on the functions.

**twoPi** what could that be?

**threePiover2**

**Pi**

**Piover2**

**oneRadian** this is one radian in degrees:  $180/\pi$

**oneDegree** this is one degree in radian:  $\pi/180$

## 4.3. Functions

### 4.3.1. Direct trigonometry

With the variable in radians:

**sin(x)** sine

**cos(x)** cosine

**tan(x)** tangent

**cot(x)** cotangent

**sec(x)** secant

**csc(x)** cosecant

With the variable in degrees:

**sind(x)** sine

**cosd(x)** cosine

**tand(x)** tangent

**cotd(x)** cotangent

**secd(x)** secant

**cscd(x)** cosecant

Only available with the variable in radians:

**tg(x)** tangent

**cotg(x)** cotangent

**sinc(x)** cardinal sine  $\text{sinc}(x) = \sin(x)/x$

### 4.3.2. Inverse trigonometry

With the value in radians:

**asin(x)** arcsine

**acos(x)** arccosine

**atan(x)** arctangent

**Arg(x, y)** the main branch of the argument of the complex number  $x+iy$ , from  $-\pi$  (excluded) to  $\pi$  (included). As the output is rounded **-Pi** is a possible return value.

**pArg(x, y)** the branch of the argument of the complex number  $x+iy$  with values going from 0 (included) to  $2\pi$  (excluded). Inherent rounding makes **twoPi** a possible return value.

**atan2(y, x)** it is **Arg(x, y)**. Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

With the value in degrees:

**asind(x)** arcsine

**acosd(x)** arccosine

**atand(x)** arctangent

**Argd(x, y)** the main branch of the argument of the complex number  $x+iy$ , from  $-180$  (excluded) to  $180$  (included). Inherent rounding of output can cause  $-180$  to be returned.

**pArgd(x, y)** the branch of the argument of the complex number  $x+iy$  with values going from  $0$  (included) to  $360$  (excluded). Inherent rounding of output can cause  $360$  to be returned.

**atan2d(y, x)** it is **Argd(x, y)**. Note the reversal of the arguments, this seems to be the most frequently encountered convention across languages.

#### 4.3.3. Conversion functions (optional definitions left to user decision)

Python provides functions **degrees()** and **radians()**. But as most of the **xinttrig** functions are already defined for the two units, I felt this was not really needed. It is a oneliner to add them:

```
\xintdefloatfunc radians(x) := x * oneDegree;
\xintdefloatfunc degrees(x) := x * oneRadian;
\xintdefunc radians(x) := float_dgt(x * oneDegree);
\xintdefunc degrees(x) := float_dgt(x * oneRadian);
```

The **float\_dgt()** does a float rounding to **\xinttheDigits** precision (recall that  $*$  is mapped to exact multiplication in **\xintdefunc**).

#### 4.4. Important implementation notes

- Currently, **xint** is lacking some dedicated internal representation of floats which means that most operations re-parse the digit tokens of their arguments to count them. . . this does not contribute to efficiency (you can load the module under **\xintverbosetrue** regime and see how the nested macros look like and get an idea of how many times some rather silly re-counting of mantissa lengths will get done!)
- One should not overwrite some function names which are employed as auxiliaries; refer to **xint** [source.pdf](#).
- Floats with large exponents are integers and are multiple of **1000**; hence modulo **360** all such ``angles'' are multiple of **40** degrees. Needless to say that considering usage of the **sind()** and **cosd()** functions with such large float numbers is meaningless.
- See **xintsource.pdf** for some comments on limitations of the range reduction implementation.

#### 4.5. Some example evaluations

```
\xintDigits* := 48\relax
Digits at \xinttheDigits:\newline
$sind(17)\approx\xintfloateval{sind(17)}$\newline
$cosd(17)\approx\xintfloateval{cosd(17)}$\newline
$tand(17)\approx\xintfloateval{tand(17)}$\newline
$sind(43)\approx\xintfloateval{sind(43)}$\newline
$cosd(43)\approx\xintfloateval{cosd(43)}$\newline
$tand(43)\approx\xintfloateval{tand(43)}$\newline
$asind(0.3)\approx\xintfloateval{asind(0.3)}$\newline
$acosd(0.3)\approx\xintfloateval{acosd(0.3)}$\newline
$atand(3)\approx\xintfloateval{atand(3)}$\newline
```

Digits at 48:

Digits at 24:

```
sind(17) ≈ 0.292371704722736728097469
cosd(17) ≈ 0.956304755963035481338651
tand(17) ≈ 0.305730681458660355734542
sind(43) ≈ 0.681998360062498500442226
cosd(43) ≈ 0.731353701619170483287544
tand(43) ≈ 0.932515086137661705612186
asind(0.3) ≈ 17.4576031237220922902460
acosd(0.3) ≈ 72.5423968762779077097540
atan(3) ≈ 71.5650511770779893515722
tan(atan(7)) ≈ 7
asind(sind(25)) ≈ 25
```

## 5. The **xintlog** package

|    |                                       |    |    |                                                  |    |
|----|---------------------------------------|----|----|--------------------------------------------------|----|
| .1 | <code>\xintreloadxintlog</code> ..... | 84 | .3 | Some information on how powers are computed..... | 84 |
| .2 | Functions .....                       | 84 |    |                                                  |    |

This package provides logarithms, exponentials and fractional powers for use with **xintexpr**.

This package was first included in release **1.3e** (2019/04/05) of **xintexpr**. It is automatically loaded by **xintexpr**.

At release **1.4e** (2021/05/05) it was substantially extended to cover usage with mantissas of up to 62 digits.

At **Digits** set to 8 or less, the old faster but less accurate macros based on **poormanlog** are used. These macros compute logarithms and exponentials with about 8 or 9 nine digits of *fixed point* precision.

**T<sub>E</sub>X-hackers note:** There is thus, for **Digits=8** or less a systematic loss of rounding precision in the floating point sense for logarithms of inputs close to 1: e.g. `log10(1.0011871)` is produced as `5.15245e-4` which stands for `0.000515145` having indeed 9 correct fractional digits, but only 6 correct digits in the floating point sense. Situation is worse for `log()` as it applies a conversion factor and does not remove the trailing junk digits, which we don't have for `log10()`. Check [xintsource.pdf](#) and **poormanlog** README for more info.

*source*

### 5.1. `\xintreloadxintlog`

The library is loaded automatically by **xintexpr** at start-up. It is then configured for 16 digits.

To work for example with 48 digits, execute `\xintSetDigits*{48}` or `\xintDigits*:=48;` (the ending `;` can be replaced by a `\relax` in case of problems due to it being active, e.g. with  $\TeX$  and some languages).

With the non-starred variant `\xintDigits:=48;` it is needed to issue `\xintreloadxintlog` to recalibrate the functions provided by the library (and the trigonometric functions will only be updated if also `\xintreloadxinttrig` is used).

### 5.2. Functions

**log10(x)** logarithm in base 10

**pow10(x)** fractional powers of 10

**log(x)** natural logarithm

**exp(x)** exponential function

**pow(x, y)** computes  $x^y$  either via the formula `pow10(y*log10(x))` (applied with some internally increased accuracy), for **y** neither an integer nor an half-integer; or via the legacy `\xintFloatPower` and `\xintFloatSqrt` macros if the exponent is integer or half-integer. Integer exponents trigger an exact evaluation in `\xinteval` if the output will not exceed (or will only slightly exceed) 10000 digits (separately for numerator and denominator), else the power is computed in the floating point sense.

```
\xintfloateval{log(2), exp(1), 2^(1/3), 2^10000}
0.6931471805599453, 2.718281828459045, 1.259921049894873, 1.995063116880758e3010
```

### 5.3. Some information on how powers are computed

For powers  $a^b$  or  $a**b$  in `\xintfloateval` the following rules apply:

1. a check is made if exponent is integer or half-integer,

2. if this is the case legacy `\xintFloatPower` (combined with `\xintFloatSqrt` for half-integer case) are used to evaluate the power (and `a` can be negative if exponent is integer),
3. else the power is computed as `pow10(b*log10(a))` (but keeping some extra digits in intermediate evaluations; in particular `b` is not float-rounded, but `a` is).

The reason is that the log/exp approach loses accuracy for very big exponents (say for exponents of the order of 100000000 or more). Here is an example of a precise computation with a very large exponent (184884258895036416):

```

$\xintTeXFromSci{\xintfloateval{1.00000001^\xintiexpr 12^16\relax}}$\newline
\xintDigits:=48;%\xintreloadxintlog is not done as log10/pow10 will not be used
$\xintTeXFromSci{\xintfloateval{1.00000001^12^16}}$\newline
\xintDigits:=64;%\xintreloadxintlog is not done as log10/pow10 will not be used
$\xintTeXFromSci{\xintfloateval{1.00000001^12^16}}$\newline
\xintDigits:=80;%\xintreloadxintlog is not done as log10/pow10 will not be used
$\xintTeXFromSci{\xintfloateval{1.00000001^12^16}}$
\xintDigits:=16;%
1.879985676694948 · 10802942130
1.87998567669494838838184407480229599674641360997 · 10802942130
1.879985676694948388381844074802295996746413609968646474887080800 · 10802942130
1.8799856766949483883818440748022959967464136099686464748870808001110266973999979 · 10802942130

```

Notes:

- in the case with 16 digits precision, we ensured  $12^{16}$  got computed exactly with all its 18 digits and was not rounded to only 16 digits (and confirmation is that the result matches the second one at 48 digits),
- the 1.4g right associativity of powers is taken into account to drop parentheses.

As the legacy `\xintFloatPower` and `\xintFloatSqrt` work in arbitrary precision, the result for integer or half-integer exponents is produced with a full-size mantissa, even if `Digits` is more than 62 (as is exemplified above).

In the  $10^{(b \cdot \log_{10}(a))}$  branch the mantissa size is limited to the minimum of `Digits` and of 64. Its last digits will start being wrong if `b` becomes about (in absolute value) 100000000. If you really need to compute powers with exponents that large or larger, it is recommended to decompose the exponent as a sum of the nearest integer or half-integer and a fractional part and express the power as a product. This is not done automatically as it would add some overhead in general for some a priori very rare use cases.

In `\xinteval`, this is as in `\xintfloateval` but for one difference: integer exponents will trigger an exact evaluation, as long as:

- the exponent absolute value is at most 9999,
- it is evaluated a priori, based on the length of the input, that the output will have at most 10000 digits (or only a bit more), separately for numerator and denominator.

The check for integrality of exponent is not on its mathematical value but on its internal representation, for speed. So  $6/3$  is not recognized as being an integer exponent in `\xinteval`; but in `\xintfloateval`, the  $6/3$  will have been computed and recognized as 2. Also  $2.00$  or  $200e-2$  is recognized as an integer in both parsers. Similar remarks apply to half-integer case.

To compute exactly higher powers than  $2^{9999}$  or  $9^{9999}$  or  $99^{5000}$  or  $999^{3333}$ , etc..., use `\xint-iiieval`. See `\xintiiPow` for related comments if you don't want to melt your CPU.

If `Digits` is at most 8, logarithms are computed faster but with less accuracy; internally, using 9 fixed point fractional digits. And powers  $a^b$  lose accuracy in last digits quickly as `b` rises. Here is what was observed with some random tests:

- for `b` neither integer nor half-integer and  $1 < b < 10$ , roughly 8 correct digits for between 80% and 90% of cases and in the remaining cases only a 1ulp error.

- for  $b$  neither integer nor half-integer and  $10^e < b < 10^{e+1}$ , roughly  $8-e$  digits are correct for about 90% of cases and there is a one unit error in the last of those digits in the remaining cases.

To maintain higher accuracy, split the input as  $a^n a^h$  with  $n$  integer or half-integer nearest to  $b$ . After having considered (and implemented) the method, decision was made to not incorporate it as it would induce serious overhead generally speaking. The  $a^b$  with fractional exponent  $b$  such that  $\text{abs}(b) < 10$  are currently computed with at most 1ulp error in the vast majority of cases it seems, which is largely precise enough for plots, and then speed matters most. Larger exponents can be handled (since 1.4f) via manually implementing the splitting trick, as described above.

The documentation of the legacy macro `\xintFloatPower` (which is used for powers with integer and half-integer exponents) explains it has a guaranteed error bound of 0.52ulp, in arbitrary precision. Generally speaking, the math functions added at 1.4e target even smaller errors (but only up to 62 digits), something of the order of 0.505ulp, and in practice they seem to achieve even better than 99% of correct rounding probability (at least in their natural ranges, and it varies according to the value of `Digits`). Perhaps in future I will re-examine whether it is worthwhile to increase a bit the theoretical accuracy of `\xintFloatPower`, as I have not had the time to really measure systematically its practical accuracy, all anecdotal evidence showing it is good.



## 6. Macros of the **xinttools** package

|     |                                                                                              |    |     |                                                                                                                                                                                 |     |
|-----|----------------------------------------------------------------------------------------------|----|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| .1  | <code>\xintRevWithBraces</code> .....                                                        | 87 | .15 | <code>\xintilooop, \xintilooopindex, \xintouter-<br/>ilooopindex, \xintbreakilooop, \xint-<br/>breakilooopanddo, \xintilooopskiptonext,<br/>\xintilooopskipandredo</code> ..... | 96  |
| .2  | <code>\xintZapFirstSpaces, \xintZapLas-<br/>tSpaces, \xintZapSpaces, \xintZapSpacesB</code>  | 87 | .16 | <code>\xintApplyInline</code> .....                                                                                                                                             | 99  |
| .3  | <code>\xintCSVtoList</code> .....                                                            | 88 | .17 | <code>\xintFor, \xintFor*</code> .....                                                                                                                                          | 100 |
| .4  | <code>\xintNthElt</code> .....                                                               | 89 | .18 | <code>\xintifForFirst, \xintifForLast</code> .....                                                                                                                              | 102 |
| .5  | <code>\xintNthOnePy</code> .....                                                             | 90 | .19 | <code>\xintBreakFor, \xintBreakForAndDo</code> ....                                                                                                                             | 103 |
| .6  | <code>\xintKeep</code> .....                                                                 | 90 | .20 | <code>\xintintegers, \xintdimensions, \xin-<br/>trationals</code> .....                                                                                                         | 103 |
| .7  | <code>\xintKeepUnbraced</code> .....                                                         | 91 | .21 | <code>\xintForpair, \xintForthree, \xintFor-<br/>four</code> .....                                                                                                              | 105 |
| .8  | <code>\xintTrim</code> .....                                                                 | 91 | .22 | <code>\xintAssign</code> .....                                                                                                                                                  | 105 |
| .9  | <code>\xintTrimUnbraced</code> .....                                                         | 91 | .23 | <code>\xintAssignArray</code> .....                                                                                                                                             | 106 |
| .10 | <code>\xintListWithSep</code> .....                                                          | 92 | .24 | <code>\xintDigitsOf</code> .....                                                                                                                                                | 106 |
| .11 | <code>\xintApply</code> .....                                                                | 92 | .25 | <code>\xintRelaxArray</code> .....                                                                                                                                              | 107 |
| .12 | <code>\xintApplyUnbraced</code> .....                                                        | 93 |     |                                                                                                                                                                                 |     |
| .13 | <code>\xintSeq</code> .....                                                                  | 93 |     |                                                                                                                                                                                 |     |
| .14 | <code>\xintloop, \xintbreakloop, \xintbreak-<br/>loopanddo, \xintloopskiptonext</code> ..... | 93 |     |                                                                                                                                                                                 |     |

These utilities used to be provided within the **xint** package; since 1.09g (2013/11/22) they have been moved to an independently usable package **xinttools**, which has none of the **xint** facilities regarding big numbers. Whenever relevant release 1.09h has made the macros `\long` so they accept `\par` tokens on input.

The completely expandable utilities (up to `\xintilooop`) are documented first, then the non expandable utilities.

[section 7](#) gives additional (some quite dated) examples of use of macros of this package.

**xinttools** is automatically loaded by **xintexpr**.

[source](#)

### 6.1. `\xintRevWithBraces`

**f** ★ `\xintRevWithBraces{⟨list⟩}` first does the *f-expansion* of its argument then it reverses the order of the tokens, or braced material, it encounters, maintaining existing braces and adding a brace pair around each naked token encountered. Space tokens (in-between top level braces or naked tokens) are gobbled. This macro is mainly thought out for use on a `⟨list⟩` of such braced material; with such a list as argument the *f-expansion* will only hit against the first opening brace, hence do nothing, and the braced stuff may thus be macros one does not want to expand.

```
\edef\x{\xintRevWithBraces{12345}}
\meaning\x:macro:->{5}{4}{3}{2}{1}
\edef\y{\xintRevWithBraces\x}
\meaning\y:macro:->{1}{2}{3}{4}{5}
```

The examples above could be defined with `\edef`'s because the braced material did not contain macros. Alternatively:

```
\expandafter\def\expandafter\w\expandafter
{\romannumeral0\xintrevwithbraces{{\A}{\B}{\C}{\D}{\E}}}
\meaning\w:macro:->{\E }{\D }{\C }{\B }{\A }
```

**n** ★ The macro `\xintReverseWithBracesNoExpand` does the same job without the initial expansion of its argument.

[source](#)

[source](#)

[source](#)

[source](#)

### 6.2. `\xintZapFirstSpaces, \xintZapLastSpaces, \xintZapSpaces, \xintZapSpacesB`

**n** ★ `\xintZapFirstSpaces{⟨stuff⟩}` does not do any expansion of its argument, nor brace removal of any sort, nor does it alter `⟨stuff⟩` in anyway apart from stripping away all *leading* spaces.

This macro will be mostly of interest to programmers who will know what I will now be talking about. *The essential points, naturally, are the complete expandability and the fact that no brace removal nor any other alteration is done to the input.*

$\TeX$ 's input scanner already converts consecutive blanks into single space tokens, but `\xintZapFirstSpaces` handles successfully also inputs with consecutive multiple space tokens. However, it is assumed that  $\langle stuff \rangle$  does not contain (except inside braced sub-material) space tokens of character code distinct from 32.

It expands in two steps, and if the goal is to apply it to the expansion text of `\x` to define `\y`, then one can do: `\odef\y{\romannumeral0\expandafter\xintzapfirstspaces\expandafter{\x}}` (one can also define a wrapper macro to `\xintZapFirstSpaces` in order to expand once the argument first, but `xinttools` not being a programming layer, it provides no "Generate Variants" facilities).

Other use case: inside a macro which received a parameter `#1`, one can do `\oodef\x{\xintZapFirstSpaces{#1}}`, or, if `#1`, after leading spaces have been stripped can accept `\edef` expansion, one can do `\edef\x{\xintZapFirstSpaces{#1}}`.

```
\xintZapFirstSpaces { \a { \X } { \b \Y } }->\a { \X } { \b \Y } +++
```

**n ★** `\xintZapLastSpaces` $\langle stuff \rangle$  does not do any expansion of its argument, nor brace removal of any sort, nor does it alter  $\langle stuff \rangle$  in anyway apart from stripping away all *ending* spaces. The same remarks as for `\xintZapFirstSpaces` apply.

```
\xintZapLastSpaces { \a { \X } { \b \Y } }-> \a { \X } { \b \Y } +++
```

**n ★** `\xintZapSpaces` $\langle stuff \rangle$  does not do any expansion of its argument, nor brace removal of any sort, nor does it alter  $\langle stuff \rangle$  in anyway apart from stripping away all *leading* and all *ending* spaces. The same remarks as for `\xintZapFirstSpaces` apply.

```
\xintZapSpaces { \a { \X } { \b \Y } }->\a { \X } { \b \Y } +++
```

**n ★** `\xintZapSpacesB` $\langle stuff \rangle$  does not do any expansion of its argument, nor does it alter  $\langle stuff \rangle$  in anyway apart from stripping away all *leading* and all *ending* spaces and possibly removing one level of braces if  $\langle stuff \rangle$  had the shape `<spaces>{braced}<spaces>`. The same remarks as for `\xintZapFirstSpaces` apply.

```
\xintZapSpacesB { \a { \X } { \b \Y } }->\a { \X } { \b \Y } +++
\xintZapSpacesB { { \a { \X } { \b \Y } } }-> \a { \X } { \b \Y } +++
```

The spaces here at the start and end of the output come from the braced material, and are not removed (one would need a second application for that; recall though that the `xint` zapping macros do not expand their argument).

[source](#)

### 6.3. `\xintCSVtoList`

**f ★** `\xintCSVtoList{a,b,c...,z}` returns `{a}{b}{c}...{z}`. A *list* is by convention in this manual simply a succession of tokens, where each braced thing will count as one item ('items' are defined according to the rules of  $\TeX$  for fetching undelimited parameters of a macro, which are exactly the same rules as for  $\mathbb{TeX}$  and macro arguments [they are the same things]). The word 'list' in 'comma separated list of items' has its usual linguistic meaning, and then an 'item' is what is delimited by commas.

So `\xintCSVtoList` takes on input a 'comma separated list of items' and converts it into a ' $\TeX$  list of braced items'. The argument to `\xintCSVtoList` may be a macro: it will first be *f-expanded*. Hence the item before the first comma, if it is itself a macro, will be expanded which may or may not be a good thing. A space inserted at the start of the first item serves to stop that expansion (and disappears). The macro `\xintCSVtoListNoExpand` does the same job without the initial expansion of the list argument.

Apart from that no expansion of the items is done and the list items may thus be completely arbitrary (and even contain perilous stuff such as unmatched `\if` and `\fi` tokens).

Contiguous spaces and tab characters, are collapsed by  $\TeX$  into single spaces. All such spaces

around commas<sup>30</sup> are removed, as well as the spaces at the start and the spaces at the end of the list.<sup>31</sup> The items may contain explicit `\par`'s or empty lines (converted by the  $\TeX$  input parsing into `\par` tokens).

```
\xintCSVtoList { 1 , { 2 , 3 , 4 , 5 } , a , {b,T} U , { c , d } , { {x , y} } }
->{1}{2 , 3 , 4 , 5}{a}{b,T} U { c , d } { {x , y} }
```

One sees on this example how braces protect commas from sub-lists to be perceived as delimiters of the top list. Braces around an entire item are removed, even when surrounded by spaces before and/or after. Braces for sub-parts of an item are not removed.

We observe also that there is a slight difference regarding the brace stripping of an item: if the braces were not surrounded by spaces, also the initial and final (but no other) spaces of the enclosed material are removed. This is the only situation where spaces protected by braces are nevertheless removed.

From the rules above: for an empty argument (only spaces, no braces, no comma) the output is `{ }` (a list with one empty item), for ``<opt. spaces>{ }<opt. spaces>'` the output is `{ }` (again a list with one empty item, the braces were removed), for ``{ }'` the output is `{ }` (again a list with one empty item, the braces were removed and then the inner space was removed), for `` { }'` the output is `{ }` (again a list with one empty item, the initial space served only to stop the expansion, so this was like ``{ }'` as input, the braces were removed and the inner space was stripped), for `` { }` the output is `{ }` (this time the ending space of the first item meant that after brace removal the inner spaces were kept; recall though that  $\TeX$  collapses on input consecutive blanks into one space token), for `` , '` the output consists of two consecutive empty items `{ }{ }`. Recall that on output everything is braced, a `{ }` is an "empty" item. Most of the above is mainly irrelevant for every day use, apart perhaps from the fact to be noted that an empty input does not give an empty output but a one-empty-item list (it is as if an ending comma was always added at the end of the input).

```
\def\y{ \a,\b,\c,\d,\e} \xintCSVtoList\y->{\a }{\b }{\c }{\d }{\e }
\def\t {{\if},{\ifnum},{\ifx},{\ifdim},{\ifcat},{\ifmmode}}
\xintCSVtoList\t->{\if }{\ifnum }{\ifx }{\ifdim }{\ifcat }{\ifmmode }
```

The results above were automatically displayed using  $\TeX$ 's primitive `\meaning`, which adds a space after each control sequence name. These spaces are not in the actual braced items of the produced lists. The first items `\a` and `\if` were either preceded by a space or braced to prevent expansion. The macro `\xintCSVtoListNoExpand` would have done the same job without the initial expansion of the list argument, hence no need for such protection but if `\y` is defined as `\def\y{\a,\b,\c,\d,\e}` we then must do:

```
\expandafter\xintCSVtoListNoExpand\expandafter {\y}
```

Else, we may have direct use:

```
\xintCSVtoListNoExpand {\if,{\ifnum},{\ifx},{\ifdim},{\ifcat},{\ifmmode}}
->{\if }{\ifnum }{\ifx }{\ifdim }{\ifcat }{\ifmmode }
```

Again these spaces are an artefact from the use in the source of the document of `\meaning` (or rather here, `\detokenize`) to display the result of using `\xintCSVtoListNoExpand` (which is done for real in this document source).

f ★ For the similar conversion from comma separated list to braced items list, but without removal of spaces around the commas, there is `\xintCSVtoListNonStripped` and `\xintCSVtoListNonStripped-NoExpand`.

source

## 6.4. `\xintNthElt`

num x f ★ `\xintNthElt{x}{<list>}` gets (expandably) the `x`th item of the `<list>`. A braced item will lose one level of brace pairs. The token list is first f-expanded.

<sup>30</sup> and multiple space tokens are not a problem; but those at the top level (not hidden inside braces) *must* be of character code 32. <sup>31</sup> let us recall that this is all done completely expandably... There is absolutely no alteration of any sort of the item apart from the stripping of initial and final space tokens (of character code 32) and brace removal if and only if the item apart from initial and final spaces (or more generally multiple char 32 space tokens) is braced.

Items are counted starting at one.

```
\xintNthElt {3}{\agh}\u{zzz}\v{Z}} is zzz
\xintNthElt {3}{\agh}\u{zzz}\v{Z}} is {zzz}
\xintNthElt {2}{\agh}\u{zzz}\v{Z}} is \u
\xintNthElt {37}{\xintiiFac {100}}=9 is the thirty-seventh digit of 100!.
\xintNthElt {10}{\xintFtoCv {566827/208524}}=1457/536
is the tenth convergent of 566827/208524 (uses xintcfrac package).
\xintNthElt {7}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
\xintNthElt {0}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=9
\xintNthElt {-3}{\xintCSVtoList {1,2,3,4,5,6,7,8,9}}=7
```

If  $x=0$ , the macro returns the *length* of the expanded list: this is not equivalent to `\xintLength` which does no pre-expansion. And it is different from `\xintLen` which is to be used only on integers or fractions.

If  $x<0$ , the macro returns the  $|x|$ th element from the end of the list. Thus for example  $x=-1$  will fetch the last item of the list.

```
\xintNthElt {-5}{\agh}\u{zzz}\v{Z}} is {agh}
```

num  
x n ★

The macro `\xintNthEltNoExpand` does the same job but without first expanding the list argument:  
`\xintNthEltNoExpand {-4}{\u\v\w T\x\y\z}` is T.

If  $x$  is strictly larger (in absolute value) than the length of the list then `\xintNthElt` produces empty contents.

[source](#)

## 6.5. `\xintNthOnePy`

num  
x f ★

`\xintNthOnePy{x}{\langle list \rangle}` gets (expandably) the  $x$ th item of the  $\langle list \rangle$ , adding a brace pair if there wasn't one.

Attention, items are counted starting at zero. For negative index, behaves as `\xintNthElt`.

If the index is out of range, the empty output is returned. If the input list was empty (had no items) the empty output is returned.

[source](#)

## 6.6. `\xintKeep`

num  
x f ★

`\xintKeep{x}{\langle list \rangle}` expands the token list argument  $L$  and produces a new list, depending on the value of  $x$ :

- if  $x>0$ , the new list contains the first  $x$  items from  $L$  (counting starts at one.) *Each such item will be output within a brace pair.* Use `\xintKeepUnbraced` if this is not desired. This means that if the list item was braced to start with, there is no modification, but if it was a token without braces, then it acquires them.
- if  $x\geq\text{length}(L)$ , the new list is the old one with all its items now braced.
- if  $x=0$  the empty list is returned.
- if  $x<0$  the last  $|x|$  elements compose the output in the same order as in the initial list; as the macro proceeds by removing head items the kept items end up in output as they were in input: no added braces.
- if  $x\leq-\text{length}(L)$  the output is identical with the input.

`\xintKeepNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintKeep {17}{\xintKeep {-69}{\xintSeq {1}{100}}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{\{1\}{2\}{3\}{4\}{5\}{6\}{7\}{8\}{9\}}}\meaning\test\par
\noindent\fdef\test {\xintKeep {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintKeep {-7}{123456789}}\meaning\test\par
```

```
macro:->\{32\}\{33\}\{34\}\{35\}\{36\}\{37\}\{38\}\{39\}\{40\}\{41\}\{42\}\{43\}\{44\}\{45\}\{46\}\{47\}\{48\}
```

```
macro:->\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}
```

```
macro:->\{3\}\{4\}\{5\}\{6\}\{7\}\{8\}\{9\}
```

```
macro:->\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}
```

macro:->3456789

[source](#)

## 6.7. \xintKeepUnbraced

Same as `\xintKeep` but no brace pairs are added around the kept items from the head of the list in the case  $x > 0$ : each such item will lose one level of braces. Thus, to remove braces from all items of the list, one can use `\xintKeepUnbraced` with its first argument larger than the length of the list; the same is obtained from `\xintListWithSep{}{\langle list \rangle}`. But the new list will then have generally many more items than the original ones, corresponding to the unbraced original items.

For  $x < 0$  the macro is no different from `\xintKeep`. Hence the name is a bit misleading because brace removal will happen only if  $x > 0$ .

`\xintKeepUnbracedNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintKeepUnbraced {10}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}%
\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}%
\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintKeepUnbraced {-7}{123456789}}\meaning\test\par
```

macro:->12345678910

macro:->1234567

macro:->{3}{4}{5}{6}{7}{8}{9}

macro:->1234567

macro:->3456789

[source](#)

## 6.8. \xintTrim

num  
x *f* ★ `\xintTrim{x}{\langle list \rangle}` expands the list argument and gobbles its first  $x$  elements.

- if  $x > 0$ , the first  $x$  items from  $L$  are gobbled. The remaining items are not modified.
- if  $x = \text{length}(L)$ , the returned list is empty.
- if  $x = 0$  the original list is returned (with no added braces.)
- if  $x < 0$  the last  $|x|$  items of the list are removed. *The head items end up braced in the output.* Use `\xintTrimUnbraced` if this is not desired.
- if  $x = -\text{length}(L)$  the output is empty.

`\xintTrimNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintTrim {17}{\xintTrim {-69}{\xintSeq {1}{100}}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{{1}{2}{3}{4}{5}{6}{7}{8}{9}}}\meaning\test\par
\noindent\fdef\test {\xintTrim {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrim {-7}{123456789}}\meaning\test\par
```

macro:->{18}{19}{20}{21}{22}{23}{24}{25}{26}{27}{28}{29}{30}{31}

macro:->{8}{9}

macro:->{1}{2}

macro:->89

macro:->{1}{2}

[source](#)

## 6.9. \xintTrimUnbraced

Same as `\xintTrim` but in case of a negative  $x$  (cutting items from the tail), the kept items from the head are not enclosed in brace pairs. They will lose one level of braces. The name is a bit misleading because when  $x > 0$  there is no brace-stripping done on the kept items, because the macro works simply by gobbling the head ones.

`\xintTrimUnbracedNoExpand` does the same without first *f-expanding* its list argument.

```
\fdef\test {\xintTrimUnbraced {-90}{\xintSeq {1}{100}}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {7}{1}{2}{3}{4}{5}{6}{7}{8}{9}}%
\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{1}{2}{3}{4}{5}{6}{7}{8}{9}}%
\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {7}{123456789}}\meaning\test\par
\noindent\fdef\test {\xintTrimUnbraced {-7}{123456789}}\meaning\test\par
```

```
macro:->12345678910
```

```
macro:->{8}{9}
```

```
macro:->12
```

```
macro:->89
```

```
macro:->12
```

*source*

## 6.10. \xintListWithSep

*nf* ★ `\xintListWithSep{<sep>}{<list>}` inserts the separator `<sep>` in-between all items of the given list of braced items (or individual tokens). The items are fetched as does  $\TeX$  with undelimited macro arguments, thus they end up unbraced in output. If the `<list>` is only one (or multiple) space tokens, the output is empty.

The list argument `<list>` gets *f-expanded* first (thus if it is a macro whose contents are braced items, the first opening brace stops the expansion, and it is as if the macro had been expanded once.) The separator `<sep>` is not pre-expanded, it ends up as is in the output (if the `<list>` contained at least two items.)

*nn* ★ The variant `\xintListWithSepNoExpand` does the same job without the initial expansion of the `<list>` argument.

```
\edef\foo{\xintListWithSep{, }{123456789{10}{11}{12}}}\meaning\foo\newline
\edef\foo{\xintListWithSep{:}{\xintiiFac{20}}}\meaning\foo\newline
\oodef\FOO{\xintListWithSepNoExpand{\FOO}{\bat\baz\biz\buz}}\meaning\FOO\newline
% a braced item or a space stops the f-expansion:
\oodef\foo{\xintListWithSep{\FOO}{\bat\baz\biz\buz}}\meaning\foo\newline
\oodef\foo{\xintListWithSep{\FOO}{ \bat\baz\biz\buz}}\meaning\foo\par
```

```
macro:->1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

```
macro:->2:4:3:2:9:0:2:0:0:8:1:7:6:6:4:0:0:0:0
```

```
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
```

```
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
```

```
macro:->\bat \FOO \baz \FOO \biz \FOO \buz
```

*source*

## 6.11. \xintApply

*ff* ★ `\xintApply{\macro}{<list>}` expandably applies the one parameter macro `\macro` to each item in the `<list>` given as second argument and returns a new list with these outputs: each item is given one after the other as parameter to `\macro` which is expanded at that time (as usual, i.e. fully for what comes first), the results are braced and output together as a succession of braced items (if `\macro` is defined to start with a space, the space will be gobbled and the `\macro` will not be expanded; it is allowed to have its own arguments, the list items serve as last arguments to `\macro`). Hence `\xintApply{\macro}{1}{2}{3}` returns `{\macro{1}}{\macro{2}}{\macro{3}}` where all instances of `\macro` have been already *f-expanded*.

Being expandable, `\xintApply` is useful for example inside alignments where implicit groups make standard loops constructs usually fail. In such situation it is often not wished that the new list elements be braced, see `\xintApplyUnbraced`. The `\macro` does not have to be expandable: `\xintApply` will try to expand it, the expansion may remain partial.

The `<list>` may itself be some macro expanding (in the previously described way) to the list of tokens to which the macro `\macro` will be applied. For example, if the `<list>` expands to some



positive number, then each digit will be replaced by the result of applying `\macro` on it.

```
\def\macro #1{\the\numexpr 9-#1\relax}
\xintApply\macro{\xintiiFac {20}}=7567097991823359999
```

**fn** ★ The macro `\xintApplyNoExpand` does the same job without the first initial expansion which gave the `<list>` of braced tokens to which `\macro` is applied.

[source](#)

## 6.12. `\xintApplyUnbraced`

**ff** ★ `\xintApplyUnbraced{\macro}{<list>}` is like `\xintApply`. The difference is that after having expanded its list argument, and applied `\macro` in turn to each item from the list, it reassembles the outputs without enclosing them in braces. The net effect is the same as doing

```
\xintListWithSep {}{\xintApply {\macro}{<list>}}
```

This is useful for preparing a macro which will itself define some other macros or make assignments, as the scope will not be limited by brace pairs.

```
\def\macro #1{\expandafter\def\csname myself#1\endcsname {#1}}
\xintApplyUnbraced\macro{\elta}{\eltb}{\eltc}
\begin{enumerate}[nosep,label=(\arabic{*})]
\item \meaning\myselfelta
\item \meaning\myselfeltb
\item \meaning\myselfeltc
\end{enumerate}
```

```
(1) macro:->elta
(2) macro:->eltb
(3) macro:->eltc
```

**fn** ★ The macro `\xintApplyUnbracedNoExpand` does the same job without the first initial expansion which gave the `<list>` of braced tokens to which `\macro` is applied.


[source](#)

## 6.13. `\xintSeq`

[<sup>num</sup><sub>x</sub>] [<sup>num</sup><sub>x</sub>] [<sup>num</sup><sub>x</sub>] ★ `\xintSeq[d]{x}{y}` generates expandably `{x}{x+d}...` up to and possibly including `{y}` if `d>0` or down to and including `{y}` if `d<0`. Naturally `{y}` is omitted if `y-x` is not a multiple of `d`. If `d=0` the macro returns `{x}`. If `y-x` and `d` have opposite signs, the macro returns nothing. If the optional argument `d` is omitted it is taken to be the sign of `y-x`. Hence `\xintSeq {1}{0}` is not empty but `{1}{0}`. But `\xintSeq [1]{1}{0}` is empty.

The arguments `x` and `y` are expanded inside a `\numexpr` so they may be count registers or a `\TeX` `\value{counternam}`, or arithmetic with such things.

```
\xintListWithSep{,\hskip2pt plus 1pt minus 1pt }{\xintSeq {12}{-25}}
12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -15,
-16, -17, -18, -19, -20, -21, -22, -23, -24, -25
\xintiiSum{\xintSeq [3]{1}{1000}}
167167
```

 When the macro is used without the optional argument `d`, it can only generate up to about 5000 numbers, the precise value depends upon some `TeX` memory parameter (input save stack).  
With the optional argument `d` the macro proceeds differently (but less efficiently) and does not stress the input save stack.

[source](#)

[source](#)

[source](#)

[source](#)

## 6.14. `\xintloop`, `\xintbreakloop`, `\xintbreakloopanddo`, `\xintloopskiptonext`

☆ `\xintloop<stuff>\if<test>...\repeat` is an expandable loop compatible with nesting. However to break out of the loop one almost always need some un-expandable step. The cousin `\xintilooop` is `\xintloop` with an embedded expandable mechanism allowing to exit from the loop. The iterated macros may contain `\par` tokens or empty lines.



If a sub-loop is to be used all the material from the start of the main loop and up to the end of the entire subloop should be braced; these braces will be removed and do not create a group. The simplest to allow the nesting of one or more sub-loops is to brace everything between `\xintloop` and `\repeat`, being careful not to leave a space between the closing brace and `\repeat`.

As this loop and `\xintilooop` will primarily be of interest to experienced TeX macro programmers, my description will assume that the user is knowledgeable enough. Some examples in this document will be perhaps more illustrative than my attempts at explanation of use.

One can abort the loop with `\xintbreakloop`; this should not be used inside the final test, and one should expand the `\fi` from the corresponding test before. One has also `\xintbreakloopanddo` whose first argument will be inserted in the token stream after the loop; one may need a macro such as `\xint_afterfi` to move the whole thing after the `\fi`, as a simple `\expandafter` will not be enough.

One will usually employ some count registers to manage the exit test from the loop; this breaks expandability, see `\xintilooop` for an expandable integer indexed loop. Use in alignments will be complicated by the fact that cells create groups, and also from the fact that any encountered un-expandable material will cause the TeX input scanner to insert `\endtemplate` on each encountered `&` or `\cr`; thus `\xintbreakloop` may not work as expected, but the situation can be resolved via `\xint_2_firstofone{&}` or use of `\TAB` with `\def\tab{&}`. It is thus simpler for alignments to use rather than `\xintloop` either the expandable `\xintApplyUnbraced` or the non-expandable but alignment compatible `\xintApplyInline`, `\xintFor` or `\xintFor*`.

As an example, let us suppose we have two macros `\A{<i>}{<j>}` and `\B{<i>}{<j>}` behaving like (small) integer valued matrix entries, and we want to define a macro `\C{<i>}{<j>}` giving the matrix product (*i* and *j* may be count registers). We will assume that `\A[I]` expands to the number of rows, `\A[J]` to the number of columns and want the produced `\C` to act in the same manner. The code is very dispendious in use of `\count` registers, not optimized in any way, not made very robust (the defined macro can not have the same name as the first two matrices for example), we just wanted to quickly illustrate use of the nesting capabilities of `\xintloop`.<sup>32</sup>

```
\newcount\rowmax \newcount\colmax \newcount\summax
\newcount\rowindex \newcount\colindex \newcount\sumindex
\newcount\tmpcount
\makeatletter
\def\MatrixMultiplication #1#2#3{%
  \rowmax #1[I]\relax
  \colmax #2[J]\relax
  \summax #1[J]\relax
  \rowindex 1
  \xintloop % loop over row index i
  {\colindex 1
    \xintloop % loop over col index k
    {\tmpcount 0
      \sumindex 1
      \xintloop % loop over intermediate index j
      \advance\tmpcount \numexpr #1\rowindex\sumindex*#2\sumindex\colindex\relax
      \ifnum\sumindex<\summax
        \advance\sumindex 1
      \repeat }%
      \expandafter\edef\csname\string#3{\the\rowindex.\the\colindex}\endcsname
      {\the\tmpcount}%
    \ifnum\colindex<\colmax
      \advance\colindex 1
    }
```

<sup>32</sup> for a more sophisticated implementation of matrix multiplication, inclusive of determinants, inverses, and display utilities, with entries big integers or decimal numbers or even fractions see some code online posted from November 11, 2013.

```

\repeat }%
\ifnum\rowindex<\rowmax
\advance\rowindex 1
\repeat
\expandafter\edef\csname\string#3{I}\endcsname{\the\rowmax}%
\expandafter\edef\csname\string#3{J}\endcsname{\the\colmax}%
\def #3##1{\ifx[##1\expandafter\Matrix@helper@size
\else\expandafter\Matrix@helper@entry\fi #3{##1}}%
}%
\def\Matrix@helper@size #1#2#3{\csname\string#1{#3}\endcsname }%
\def\Matrix@helper@entry #1#2#3%
{\csname\string#1{\the\numexpr#2.\the\numexpr#3}\endcsname }%
\def\A #1{\ifx[#1\expandafter\A@size
\else\expandafter\A@entry\fi {#1}}%
\def\A@size #1#2{\ifx I#23\else4\fi}% 3rows, 4columns
\def\A@entry #1#2{\the\numexpr #1+#2-1\relax}% not pre-computed...
\def\B #1{\ifx[#1\expandafter\B@size
\else\expandafter\B@entry\fi {#1}}%
\def\B@size #1#2{\ifx I#24\else3\fi}% 4rows, 3columns
\def\B@entry #1#2{\the\numexpr #1-#2\relax}% not pre-computed...
\makeatother
\MatrixMultiplication\A\B\C \MatrixMultiplication\C\C\D
\MatrixMultiplication\C\D\E \MatrixMultiplication\C\E\F
\begin{multicols}2
\[\begin{pmatrix}
\A11&\A12&\A13&\A14\\
\A21&\A22&\A23&\A24\\
\A31&\A32&\A33&\A34
\end{pmatrix}
\times
\begin{pmatrix}
\B11&\B12&\B13\\
\B21&\B22&\B23\\
\B31&\B32&\B33\\
\B41&\B42&\B43
\end{pmatrix}
=
\begin{pmatrix}
\C11&\C12&\C13\\
\C21&\C22&\C23\\
\C31&\C32&\C33
\end{pmatrix}\]
\[\begin{pmatrix}
\C11&\C12&\C13\\
\C21&\C22&\C23\\
\C31&\C32&\C33
\end{pmatrix}^2 = \begin{pmatrix}
\D11&\D12&\D13\\
\D21&\D22&\D23\\
\D31&\D32&\D33
\end{pmatrix}\]
\[\begin{pmatrix}

```

```

\C11&\C12&\C13\\
\C21&\C22&\C23\\
\C31&\C32&\C33
\end{pmatrix}^3 = \begin{pmatrix}
\E11&\E12&\E13\\
\E21&\E22&\E23\\
\E31&\E32&\E33
\end{pmatrix}\]
\[\begin{pmatrix}
\C11&\C12&\C13\\
\C21&\C22&\C23\\
\C31&\C32&\C33
\end{pmatrix}^4 = \begin{pmatrix}
\F11&\F12&\F13\\
\F21&\F22&\F23\\
\F31&\F32&\F33
\end{pmatrix}\]
\end{multicols}

```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 20 & 10 & 0 \\ 26 & 12 & -2 \\ 32 & 14 & -4 \end{pmatrix}$$

$$\begin{pmatrix} 20 & 10 & 0 \\ 26 & 12 & -2 \\ 32 & 14 & -4 \end{pmatrix}^3 = \begin{pmatrix} 20880 & 10160 & -560 \\ 24624 & 11968 & -688 \\ 28368 & 13776 & -816 \end{pmatrix}$$

$$\begin{pmatrix} 20 & 10 & 0 \\ 26 & 12 & -2 \\ 32 & 14 & -4 \end{pmatrix}^2 = \begin{pmatrix} 660 & 320 & -20 \\ 768 & 376 & -16 \\ 876 & 432 & -12 \end{pmatrix}$$

$$\begin{pmatrix} 20 & 10 & 0 \\ 26 & 12 & -2 \\ 32 & 14 & -4 \end{pmatrix}^4 = \begin{pmatrix} 663840 & 322880 & -18080 \\ 781632 & 380224 & -21184 \\ 899424 & 437568 & -24288 \end{pmatrix}$$

## [source](#) [source](#) [source](#) [source](#) 6.15. `\xintloop`, `\xintloopindex`, `\xintouterloopindex`, `\xintbreakloop`, `\xintbreakloopanddo`, `\xintloopskiptonext`, `\xintloopskipandredo`

☆ `\xintloop[start+delta](stuff)\if<test> ... \repeat` is a completely expandable nestable loop. complete expandability depends naturally on the actual iterated contents, and complete expansion will not be achievable under a sole *f-expansion*, as is indicated by the hollow star in the margin; thus the loop can be used inside an `\edef` but not inside arguments to the package macros. It can be used inside an `\xintexpr... \relax`. The `[start+delta]` is mandatory, not optional.

This loop benefits via `\xintloopindex` to (a limited access to) the integer index of the iteration. The starting value `start` (which may be a `\count`) and increment `delta` (*id.*) are mandatory arguments. A space after the closing square bracket is not significant, it will be ignored. Spaces inside the square brackets will also be ignored as the two arguments are first given to a `\numexpr... \relax`. Empty lines and explicit `\par` tokens are accepted.

As with `\xintloop`, this tool will mostly be of interest to advanced users. For nesting, one puts inside braces all the material from the start (immediately after `[start+delta]`) and up to and inclusive of the inner loop, these braces will be removed and do not create a loop. In case of nesting, `\xintouterloopindex` gives access to the index of the outer loop. If needed one could write on its model a macro giving access to the index of the outer outer loop (or even to the *n*th outer loop).

The `\xintloopindex` and `\xintouterloopindex` can not be used inside braces, and generally speaking this means they should be expanded first when given as argument to a macro, and that this macro receives them as delimited arguments, not braced ones. Or, but naturally this will break

expandability, one can assign the value of `\xintloopindex` to some `\count`. Both `\xintloopindex` and `\xintouterloopindex` extend to the literal representation of the index, thus in `\ifnum` tests, if it comes last one has to correctly end the macro with a `\space`, or encapsulate it in a `\numexpr..\relax`.

When the repeat-test of the loop is, for example, `\ifnum\xintloopindex<10 \repeat`, this means that the last iteration will be with `\xintloopindex=10` (assuming `delta=1`). There is also `\ifnum\xintloopindex=10 \else\repeat` to get the last iteration to be the one with `\xintloopindex=10`.

One has `\xintbreakiloop` and `\xintbreakiloopanddo` to abort the loop. The syntax of `\xintbreakiloopanddo` is a bit surprising, the sequence of tokens to be executed after breaking the loop is not within braces but is delimited by a dot as in:

```
\xintbreakiloopanddo <afterloop>.etc.. etc... \repeat
```

The reason is that one may wish to use the then current value of `\xintloopindex` in `<afterloop>` but it can't be within braces at the time it is evaluated. However, it is not that easy as `\xintloopindex` must be expanded before, so one ends up with code like this:

```
\expandafter\xintbreakiloopanddo\expandafter\macro\xintloopindex.%
etc.. etc.. \repeat
```

As moreover the `\fi` from the test leading to the decision of breaking out of the loop must be cleared out of the way, the above should be a branch of an expandable conditional test, else one needs something such as:

```
\xint_afterfi{\expandafter\xintbreakiloopanddo\expandafter\macro\xintloopindex.}%
\fi etc..etc.. \repeat
```

There is `\xintloopskiptonext` to abort the current iteration and skip to the next, `\xintloopskipandredo` to skip to the end of the current iteration and redo it with the same value of the index (something else will have to change for this not to become an eternal loop...).

Inside alignments, if the looped-over text contains a `&` or a `\cr`, any un-expandable material before a `\xintloopindex` will make it fail because of `\endtemplate`; in such cases one can always either replace `&` by a macro expanding to it or replace it by a suitable `\firstofone{&}`, and similarly for `\cr`.

As an example, let us construct an `\edef\z{...}` which will define `\z` to be a list of prime numbers:

```
\begingroup
\edef\z
{\xintloop [10001+2]
 {\xintloop [3+2]
  \ifnum\xintouterloopindex<\numexpr\xintloopindex*\xintloopindex\relax
   \xintouterloopindex,
   \expandafter\xintbreakiloop
  \fi
  \ifnum\xintouterloopindex=\numexpr
    (\xintouterloopindex/\xintloopindex)*\xintloopindex\relax
  \else
  \repeat
 }% no space here
 \ifnum \xintloopindex < 10999 \repeat }%
\meaning\z\endgroup
```

```
macro:->10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103,
10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193, 10211, 10223, 10243,
10247, 10253, 10259, 10267, 10271, 10273, 10289, 10301, 10303, 10313, 10321, 10331, 10333, 10337,
10343, 10357, 10369, 10391, 10399, 10427, 10429, 10433, 10453, 10457, 10459, 10463, 10477, 10487,
10499, 10501, 10513, 10529, 10531, 10559, 10567, 10589, 10597, 10601, 10607, 10613, 10627, 10631,
10639, 10651, 10657, 10663, 10667, 10687, 10691, 10709, 10711, 10723, 10729, 10733, 10739, 10753,
10771, 10781, 10789, 10799, 10831, 10837, 10847, 10853, 10859, 10861, 10867, 10883, 10889, 10891,
```

10903, 10909, 10937, 10939, 10949, 10957, 10973, 10979, 10987, 10993, and we should have taken some steps to not have a trailing comma, but the point was to show that one can do that in an `\edef`! See also [subsection 7.3](#) which extracts from this code its way of testing primality.

Let us create an alignment where each row will contain all divisors of its first entry. Here is the output, thus obtained without any count register:

```
\begin{multicols}2
\tabskiplex \normalcolor
\halign{&\hfil#\hfil\cr
  \xintilooop [1+1]
  {\xexpandafter\bfseries\xintilooopindex &
  \xintilooop [1+1]
  \ifnum\xintouterilooopindex=\numexpr
    (\xintouterilooopindex/\xintilooopindex)*\xintilooopindex\relax
  \xintilooopindex&\fi
  \ifnum\xintilooopindex<\xintouterilooopindex\space % CRUCIAL \space HERE
  \repeat \cr }%
  \ifnum\xintilooopindex<30
  \repeat
}
\end{multicols}
```

|    |   |    |    |    |    |    |    |
|----|---|----|----|----|----|----|----|
| 1  | 1 | 16 | 1  | 2  | 4  | 8  | 16 |
| 2  | 1 | 2  | 17 | 1  | 17 |    |    |
| 3  | 1 | 3  | 18 | 1  | 2  | 3  | 6  |
| 4  | 1 | 2  | 4  | 9  | 18 |    |    |
| 5  | 1 | 5  | 19 | 1  | 19 |    |    |
| 6  | 1 | 2  | 3  | 6  | 10 | 20 |    |
| 7  | 1 | 7  | 21 | 1  | 3  | 7  | 21 |
| 8  | 1 | 2  | 4  | 8  | 22 | 1  | 2  |
| 9  | 1 | 3  | 9  | 23 | 1  | 23 |    |
| 10 | 1 | 2  | 5  | 10 | 24 | 1  | 2  |
| 11 | 1 | 11 | 25 | 1  | 5  | 25 |    |
| 12 | 1 | 2  | 3  | 4  | 6  | 8  | 12 |
| 13 | 1 | 13 | 26 | 1  | 2  | 13 | 26 |
| 14 | 1 | 2  | 7  | 14 | 27 | 1  | 3  |
| 15 | 1 | 3  | 5  | 15 | 28 | 1  | 2  |
|    |   |    |    |    | 29 | 1  | 29 |
|    |   |    |    |    | 30 | 1  | 2  |
|    |   |    |    |    |    | 3  | 5  |
|    |   |    |    |    |    | 6  | 10 |
|    |   |    |    |    |    | 15 | 30 |

We wanted this first entry in bold face, but `\bfseries` leads to unexpandable tokens, so the `\expandafter` was necessary for `\xintilooopindex` and `\xintouterilooopindex` not to be confronted with a hard to digest `\endtemplate`. An alternative way of coding:

```
\tabskiplex
\def\firstofone #1{#1}%
\halign{&\hfil#\hfil\cr
  \xintilooop [1+1]
  {\bfseries\xintilooopindex\firstofone{&}%
  \xintilooop [1+1] \ifnum\xintouterilooopindex=\numexpr
  (\xintouterilooopindex/\xintilooopindex)*\xintilooopindex\relax
  \xintilooopindex\firstofone{&}\fi
  \ifnum\xintilooopindex<\xintouterilooopindex\space % \space is CRUCIAL
  \repeat \firstofone{\cr}}%
  \ifnum\xintilooopindex<30 \repeat }
```

The next utilities are not compatible with expansion-only context.

[source](#)

## 6.16. `\xintApplyInline`

`\xintApplyInline{\macro}{\list}` works non expandably. It applies the one-parameter `\macro` to the first element of the expanded list (`\macro` may have itself some arguments, the list item will be appended as last argument), and is then re-inserted in the input stream after the tokens resulting from this first expansion of `\macro`. The next item is then handled.

This is to be used in situations where one needs to do some repetitive things. It is not expandable and can not be completely expanded inside a macro definition, to prepare material for later execution, contrarily to what `\xintApply` or `\xintApplyUnbraced` achieve.

```
\def\Macro #1{\advance\cnta #1 , \the\cnta}
\cnta 0
0\xintApplyInline\Macro {3141592653}.
```

0, 3, 4, 8, 9, 14, 23, 25, 31, 36, 39. The first argument `\macro` does not have to be an expandable macro.

`\xintApplyInline` submits its second, token list parameter to an *f-expansion*. Then, each *unbraced* item will also be *f-expanded*. This provides an easy way to insert one list inside another. *Braced* items are not expanded. Spaces in-between items are gobbled (as well as those at the start or the end of the list), but not the spaces *inside* the braced items.

`\xintApplyInline`, despite being non-expandable, does survive to contexts where the executed `\macro` closes groups, as happens inside alignments with the tabulation character `&`. This tabular provides an example:

```
\centerline{\normalcolor\begin{tabular}{ccc}
  $N$ & $N^2$ & $N^3$ \\ \hline
  \def\Row #1{ #1 & \xintiiSqr {#1} & \xintiiPow {#1}{3} \\ \hline }%
  \xintApplyInline \Row {\xintCSVtoList{17,28,39,50,61}}
\end{tabular}}\medskip
```

| N  | N <sup>2</sup> | N <sup>3</sup> |
|----|----------------|----------------|
| 17 | 289            | 4913           |
| 28 | 784            | 21952          |
| 39 | 1521           | 59319          |
| 50 | 2500           | 125000         |
| 61 | 3721           | 226981         |

We see that despite the fact that the first encountered tabulation character in the first row close a group and thus erases `\Row` from  $\text{\TeX}$ 's memory, `\xintApplyInline` knows how to deal with this.

Using `\xintApplyUnbraced` is an alternative: the difference is that this would have prepared all rows first and only put them back into the token stream once they are all assembled, whereas with `\xintApplyInline` each row is constructed and immediately fed back into the token stream: when one does things with numbers having hundreds of digits, one learns that keeping on hold and shuffling around hundreds of tokens has an impact on  $\text{\TeX}$ 's speed (make this ``thousands of tokens'' for the impact to be noticeable).

One may nest various `\xintApplyInline`'s. For example (see the [table](#) on the next page):

```
\begin{figure*}[ht]
\centering\phantomsection\label{float}
\def\Row #1{#1:\xintApplyInline {\Item {#1}}{0123456789}\ }%
\def\Item #1{#1&\xintiiPow {#1}{#2}}%
\centeredline {\begin{tabular}{ccccccccc} &0&1&2&3&4&5&6&7&8&9\\ \hline
  \xintApplyInline \Row {0123456789}
\end{tabular}}
```

`\end{figure*}`

|    | 0 | 1 | 2  | 3   | 4    | 5     | 6      | 7       | 8        | 9         |
|----|---|---|----|-----|------|-------|--------|---------|----------|-----------|
| 0: | 1 | 0 | 0  | 0   | 0    | 0     | 0      | 0       | 0        | 0         |
| 1: | 1 | 1 | 1  | 1   | 1    | 1     | 1      | 1       | 1        | 1         |
| 2: | 1 | 2 | 4  | 8   | 16   | 32    | 64     | 128     | 256      | 512       |
| 3: | 1 | 3 | 9  | 27  | 81   | 243   | 729    | 2187    | 6561     | 19683     |
| 4: | 1 | 4 | 16 | 64  | 256  | 1024  | 4096   | 16384   | 65536    | 262144    |
| 5: | 1 | 5 | 25 | 125 | 625  | 3125  | 15625  | 78125   | 390625   | 1953125   |
| 6: | 1 | 6 | 36 | 216 | 1296 | 7776  | 46656  | 279936  | 1679616  | 10077696  |
| 7: | 1 | 7 | 49 | 343 | 2401 | 16807 | 117649 | 823543  | 5764801  | 40353607  |
| 8: | 1 | 8 | 64 | 512 | 4096 | 32768 | 262144 | 2097152 | 16777216 | 134217728 |
| 9: | 1 | 9 | 81 | 729 | 6561 | 59049 | 531441 | 4782969 | 43046721 | 387420489 |

One could not move the definition of `\Item` inside the tabular, as it would get lost after the first `&`. But this works:

```
\begin{tabular}{cccccccccc}
&0&1&2&3&4&5&6&7&8&9\\ \hline
\def\Row #1{#1:\xintApplyInline {\&\xintiiPow {#1}{0123456789}}\\ }%
\xintApplyInline \Row {0123456789}
\end{tabular}
```

A limitation is that, contrarily to what one may have expected, the `\macro` for an `\xintApplyInline` can not be used to define the `\macro` for a nested sub-`\xintApplyInline`. For example, this does not work:

```
\def\Row #1{#1:\def\Item ##1{\&\xintiiPow {#1}{##1}}%
\xintApplyInline \Item {0123456789}\\ }%
\xintApplyInline \Row {0123456789} % does not work
```

But see `\xintFor`.

[source](#)

[source](#)

## 6.17. `\xintFor`, `\xintFor*`

`\xintFor` is a new kind of for loop.<sup>33</sup> Rather than using macros for encapsulating list items, its behaviour is like a macro with parameters: `#1`, `#2`, ..., `#9` are used to represent the items for up to nine levels of nested loops. Here is an example:

```
\xintFor #9 in {1,2,3} \do {%
\xintFor #1 in {4,5,6} \do {%
\xintFor #3 in {7,8,9} \do {%
\xintFor #2 in {10,11,12} \do {%
$$#9\times#1\times#3\times#2=\xintiiPrd{{#1}{#2}{#3}{#9}}$$$$}}
```

This example illustrates that one does not have to use `#1` as the first one: the order is arbitrary. But each level of nesting should have its specific macro parameter. Nine levels of nesting is presumably overkill, but I did not know where it was reasonable to stop. `\par` tokens are accepted in both the comma separated list and the replacement text.

$\TeX$ nical notes:

- The `#1` is replaced in the iterated-over text exactly as in general  $\TeX$  macros or  $\LaTeX$  commands. This spares the user quite a few `\expandafter`'s or other tricks needed with loops which have the values encapsulated in macros, like  $\LaTeX$ 's `\@for` and `\@tfor`.

<sup>33</sup> first introduced with `xint 1.09c` of 2013/10/09.



- `\xintFor` (and `\xintFor*`) isn't purely expandable: one can not use it inside an `\edef`. But it may be used, as will be shown in examples, in some contexts such as  $\TeX$ 's `tabular` which are usually hostile to non-expandable loops.
- `\xintFor` (and `\xintFor*`) does some assignments prior to executing each iteration of the replacement text, but it acts purely expandably after the last iteration, hence if for example the replacement text ends with a `\\`, the loop can be used insided a `tabular` and be followed by a `\hline` without creating the dreaded `'Misplaced \noalign'` error.
- As stated in previous item the first iteration follows some non-expandable internal dealings. This means for example that in  $\TeX$ , one can not inject a `\multicolumn` in the first iteration. Sometimes one way work around this by injecting father `&\multicolumn` or `\\ \multicolumn`.
- It does not create groups.
- It makes no global assignments.
- The iterated replacement text may close a group which was opened even before the start of the loop (typical example being with `&` in alignments).

```
\begin{tabular}{rccccc}
\hline
\xintFor #1 in {A, B, C} \do {%
  #1:\xintFor #2 in {a, b, c, d, e} \do {&($ #2 \to #1 $)}\\ }%
\hline
\end{tabular}
```

|    |         |         |         |         |         |
|----|---------|---------|---------|---------|---------|
| A: | (a → A) | (b → A) | (c → A) | (d → A) | (e → A) |
| B: | (a → B) | (b → B) | (c → B) | (d → B) | (e → B) |
| C: | (a → C) | (b → C) | (c → C) | (d → C) | (e → C) |

- There is no facility provided which would give access to a count of the number of iterations as it is technically not easy to do so it in a way working with nested loops while maintaining the `'expandable after done'` property; something in the spirit of `\xint-loopindex` is possible but this approach would bring its own limitations and complications. Hence the user is invited to update her own count or  $\TeX$  counter or macro at each iteration, if needed.
- A `\macro` whose definition uses internally an `\xintFor` loop may be used inside another `\xintFor` loop even if the two loops both use the same macro parameter. The loop definition inside `\macro` must use `##` as is the general rule for definitions done inside macros.
- `\xintFor` is for comma separated values and `\xintFor*` for lists of braced items; their respective expansion policies differ. They are described later.

Regarding `\xintFor`:

- the spaces between the various declarative elements are all optional,
- in the list of comma separated values, spaces around the commas or at the start and end are ignored,
- if an item must contain itself its own commas, then it should be braced, and the braces will be removed before feeding the iterated-over text,
- the list may be a macro, it is expanded only once,
- items are not pre-expanded. The first item should be braced or start with a space if the list is explicit and the item should not be pre-expanded,

- empty items give empty #1's in the replacement text, they are not skipped,
- an empty list executes once the replacement text with an empty parameter value,
- the list, if not a macro, must be braced.

*\*fn* Regarding `\xintFor*`:

- it handles lists of braced items (or naked tokens),
- it *f-expands* the list,
- and more generally it *f-expands* each naked token encountered before assigning the #1 values (gobbling spaces in the process); this makes it easy to simulate concatenation of multiple lists `\x`, `\y`: if `\x` expands to `{1}{2}{3}` and `\y` expands to `{4}{5}{6}` then `{\x\y}` as argument to `\xintFor*` has the same effect as `{{1}{2}{3}{4}{5}{6}}`.

For a further illustration see the use of `\xintFor*` at the end of [subsection 3.18](#).

- spaces at the start, end, or in-between items are gobbled (but naturally not the spaces inside braced items),
- except if the list argument is a macro (with no parameters), it must be braced,
- an empty list leads to an empty result.

The macro `\xintSeq` which generates arithmetic sequences is to be used with `\xintFor*` as its output consists of successive braced numbers (given as digit tokens).

```
\xintFor* #1 in {\xintSeq [+2]{-7}{+2}}\do {stuff
  with #1\xintifForLast{\par}{\newline}}
```

```
stuff with -7
stuff with -5
stuff with -3
stuff with -1
stuff with 1
```

When nesting `\xintFor*` loops, using `\xintSeq` in the inner loops is inefficient, as the arithmetic sequence will be re-created each time. A more efficient style is:

```
\edef\innersequence {\xintSeq[+2]{-50}{50}}%
\xintFor* #1 in {\xintSeq {13}{27}} \do
  {\xintFor* #2 in \innersequence \do {stuff with #1 and #2}%
  .. some other macros .. }
```

This is a general remark applying for any nesting of loops, one should avoid recreating the inner lists of arguments at each iteration of the outer loop.

When the loop is defined inside a macro for later execution the # characters must be doubled.<sup>34</sup> For example:

```
\def\T{\def\z {}%
  \xintFor* ##1 in {{u}{v}{w}} \do {%
    \xintFor ##2 in {x,y,z} \do {%
      \expandafter\def\expandafter\z\expandafter {\z\sep (##1,##2)} }%
    }%
  }%
\T\def\sep {\def\sep{, }}\z
```

```
(u,x), (u,y), (u,z), (v,x), (v,y), (v,z), (w,x), (w,y), (w,z)
```

Similarly when the replacement text of `\xintFor` defines a macro with parameters, the macro character # must be doubled.

The iterated macros as well as the list items are allowed to contain explicit `\par` tokens.

## 6.18. `\xintifForFirst`, `\xintifForLast`

*nn* ★ `\xintifForFirst` {YES branch}{NO branch} and `\xintifForLast` {YES branch}{NO branch} execute the YES or NO branch if the `\xintFor` or `\xintFor*` loop is currently in its first, respectively last, iteration.

<sup>34</sup> sometimes what seems to be a macro argument isn't really; in `\raisebox{1cm}{\xintFor #1 in {a,b,c}\do {#1}}` no doubling should be done.

Designed to work as expected under nesting (but see frame next.) Don't forget an empty brace pair `{}` if a branch is to do nothing. May be used multiple times in the replacement text of the loop.

Pay attention to these implementation features:

- if an inner `\xintFor` loop is positioned before the `\xintifForFirst` or `\xintifForLast` of the outer loop it will contaminate their settings. This applies also naturally if the inner loop arises from the expansion of some macro located before the outer conditionals.

One fix is to make sure that the outer conditionals are expanded before the inner loop is executed, e.g. this will be the case if the inner loop is located inside one of the branches of the conditional.

Another approach is to enclose, if feasible, the inner loop in a group of its own.

- if the replacement text closes a group (e.g. from a `&` inside an alignment), the conditionals will lose their ascribed meanings and end up possibly undefined, depending whether there is some outer loop whose execution started before the opening of the group.

The fix is to arrange things so that the conditionals are expanded before  $\TeX$  encounters the closing-group token.

[source](#)

[source](#)

## 6.19. `\xintBreakFor`, `\xintBreakForAndDo`

One may immediately terminate an `\xintFor` or `\xintFor*` loop with `\xintBreakFor`.

As it acts by clearing up all the rest of the replacement text when encountered, it will not work from inside some `\if...\fi` without suitable `\expandafter` or swapping technique.

Also it can't be used from inside braces as from there it can't see the end of the replacement text.

There is also `\xintBreakForAndDo`. Both are illustrated by various examples in the next section which is devoted to ```forever''` loops.

[source](#)

[source](#)

[source](#)

## 6.20. `\xintintegers`, `\xintdimensions`, `\xintrationals`

If the list argument to `\xintFor` (or `\xintFor*`, both are equivalent in this context) is `\xintintegers` (equivalently `\xintegers`) or more generally `\xintintegers[start+delta]` (the whole within braces!)<sup>35</sup>, then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of (short) integers with initial value `start` and increment `delta` (default values: `start=1`, `delta=1`; if the optional argument is present it must contains both of them, and they may be explicit integers, or macros or count registers). The `#1` (or `#2`, ..., `#9`) will stand for `\numexpr <opt sign><digits>\relax`, and the litteral representation as a string of digits can thus be obtained as `\the#1` or `\number#1`. Such a `#1` can be used in an `\ifnum` test with no need to be postfixed with a space or a `\relax` and one should not add them.

If the list argument is `\xintdimensions` or more generally `\xintdimensions[start+delta]` (within braces!), then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of dimensions with initial value `start` and increment `delta`. Default values: `start=0pt`, `delta=1pt`; if the optional argument is present it must contain both of them, and they may be explicit specifications, or macros, or dimen registers, or length macros in  $\TeX$  (the stretch and shrink components will be discarded). The `#1` will be `\dimexpr <opt sign><digits>sp\relax`, from which one can get the litteral (approximate) representation in points via `\the#1`. So `#1` can be used anywhere  $\TeX$  expects a dimension (and there is no need in conditionals to insert a

<sup>35</sup> the `start+delta` optional specification may have extra spaces around the plus sign or near the square brackets, such spaces are removed. The same applies with `\xintdimensions` and `\xintrationals`.

`\relax`, and one should not do it), and to print its value one uses `\the#1`. The chosen representation guarantees exact incrementation with no rounding errors accumulating from converting into points at each step.

If the list argument to `\xintFor` (or `\xintFor*`) is `\xintrationals` or more generally `\xintrationals[start+delta]` (within braces!), then `\xintFor` does an infinite iteration where `#1` (or `#2`, ..., `#9`) will run through the arithmetic sequence of `xintfrac` fractions with initial value `start` and increment `delta` (default values: `start=1/1`, `delta=1/1`). This loop works only with `xintfrac` loaded. If the optional argument is present it must contain both of them, and they may be given in any of the formats recognized by `xintfrac` (fractions, decimal numbers, numbers in scientific notations, numerators and denominators in scientific notation, etc...) , or as macros or count registers (if they are short integers). The `#1` (or `#2`, ..., `#9`) will be an `a/b` fraction (without a `[n]` part), where the denominator `b` is the product of the denominators of `start` and `delta` (for reasons of speed `#1` is not reduced to irreducible form, and for another reason explained later `start` and `delta` are not put either into irreducible form; the input may use explicitly `\xintIrr` to achieve that).

```
\begingroup\small
\noindent\parbox{\dimexpr\linewidth-3em}{\color[named]{OrangeRed}%
\xintFor #1 in {\xintrationals [10/21+1/21]} \do
{#1=\xintifInt {#1}
  {\textcolor{blue}{\xintTrunc{10}{#1}}}
  {\xintTrunc{10}{#1}}% display in blue if an integer
  \xintifGt {#1}{1.123}{\xintBreakFor}{, }%
}}
\endgroup\smallskip
```

10/21=0.4761904761, 11/21=0.5238095238, 12/21=0.5714285714, 13/21=0.6190476190,  
 14/21=0.6666666666, 15/21=0.7142857142, 16/21=0.7619047619, 17/21=0.8095238095,  
 18/21=0.8571428571, 19/21=0.9047619047, 20/21=0.9523809523, 21/21=1.0000000000,  
 22/21=1.0476190476, 23/21=1.0952380952, 24/21=1.1428571428

The example above confirms that computations are done exactly, and illustrates that the two initial (reduced) denominators are not multiplied when they are found to be equal. It is thus recommended to input `start` and `delta` with a common smallest possible denominator, or as fixed point numbers with the same numbers of digits after the decimal mark; and this is also the reason why `start` and `delta` are not by default made irreducible. As internally the computations are done with numerators and denominators completely expanded, one should be careful not to input numbers in scientific notation with exponents in the hundreds, as they will get converted into as many zeroes.

```
\noindent\parbox{\dimexpr.7\linewidth}{\raggedright
\xintFor #1 in {\xintrationals [0.000+0.125]} \do
{\edef\tmp{\xintTrunc{3}{#1}}%
\xintifInt {#1}
  {\textcolor{blue}{\tmp}}
  {\tmp}%
\xintifGt {#1}{2}{\xintBreakFor}{, }%
}}\smallskip
```

0, 0.125, 0.250, 0.375, 0.500, 0.625, 0.750, 0.875, 1.000, 1.125,  
 1.250, 1.375, 1.500, 1.625, 1.750, 1.875, 2.000, 2.125

We see here that `\xintTrunc` outputs (deliberately) zero as 0, not (here) 0.000, the idea being not to lose the information that the truncated thing was truly zero. Perhaps this behaviour should be changed? or made optional? Anyhow printing of fixed points numbers should be dealt with via dedicated packages such as `numprint` or `siunitx`.

[source](#)

[source](#)

[source](#)

## 6.21. `\xintForpair`, `\xintForthree`, `\xintForfour`

**on** The syntax is illustrated in this example. The notation is the usual one for **n**-uples, with parentheses and commas. Spaces around commas and parentheses are ignored.

```
{\centering\begin{tabular}{cccc}
\xintForpair #1#2 in { ( A , a ) , ( B , b ) , ( C , c ) } \do {%
\xintForpair #3#4 in { ( X , x ) , ( Y , y ) , ( Z , z ) } \do {%
  $\Biggl(\begin{tabular}{cc}
    -#1- & -#3-\ \\
    -#4- & -#2-\ \\
  \end{tabular}$\Biggr)$&\\noalign{\vskip1\jot}}%
\end{tabular}}\}
```

$$\begin{pmatrix} -A- & -X- \\ -x- & -a- \end{pmatrix} \begin{pmatrix} -A- & -Y- \\ -y- & -a- \end{pmatrix} \begin{pmatrix} -A- & -Z- \\ -z- & -a- \end{pmatrix}$$

$$\begin{pmatrix} -B- & -X- \\ -x- & -b- \end{pmatrix} \begin{pmatrix} -B- & -Y- \\ -y- & -b- \end{pmatrix} \begin{pmatrix} -B- & -Z- \\ -z- & -b- \end{pmatrix}$$

$$\begin{pmatrix} -C- & -X- \\ -x- & -c- \end{pmatrix} \begin{pmatrix} -C- & -Y- \\ -y- & -c- \end{pmatrix} \begin{pmatrix} -C- & -Z- \\ -z- & -c- \end{pmatrix}$$

`\xintForpair` must be followed by either `#1#2`, `#2#3`, `#3#4`, ..., or `#8#9` with `#1` usable as an alias for `#1#2`, `#2` as alias for `#2#3`, etc ... and similarly for `\xintForthree` (using `#1#2#3` or simply `#1`, `#2#3#4` or simply `#2`, ...) and `\xintForfour` (with `#1#2#3#4` etc...).

Nesting works as long as the macro parameters are distinct among `#1`, `#2`, ..., `#9`. A macro which expands to an `\xintFor` or a `\xintFor(pair,three,four)` can be used in another one with no constraint about using distinct macro parameters.

`\par` tokens are accepted in both the comma separated list and the replacement text.

[source](#)

## 6.22. `\xintAssign`

`\xintAssign`(*braced things*)\to(*as many cs as they are things*) defines (without checking if something gets overwritten) the control sequences on the right of `\to` to expand to the successive tokens or braced items located to the left of `\to`. `\xintAssign` is not an expandable macro.

**f-expansion** is first applied to the material in front of `\xintAssign` which is fetched as one argument if it is braced. Then the expansion of this argument is examined and successive items are assigned to the macros following `\to`. There must be exactly as many macros as items. No check is done. The macro assignments are done with removal of one level of brace pairs from each item.

After the initial **f-expansion**, each assigned (brace-stripped) item will be expanded according to the setting of the optional parameter.

For example `\xintAssign [e]...` means that all assignments are done using `\edef`. With `[f]` the assignments will be made using `\fdef`. The default is simply to make the definitions with `\def`, corresponding to an empty optional parameter `[]`. Possibilities for the optional parameter are: `[]`, `[g]`, `[e]`, `[x]`, `[o]`, `[go]`, `[oo]`, `[goo]`, `[f]`, `[gf]`. For example `[oo]` means a double expansion.

```
\xintAssign \xintiiDivision{1000000000000}{133333333}\to\Q\R
\meaning\Q\newline
\meaning\R\newline
\xintAssign {\xintiiDivision{1000000000000}{133333333}}\to\X
\meaning\X\newline
\xintAssign [oo]{\xintiiDivision{1000000000000}{133333333}}\to\X
\meaning\X\newline
\xintAssign \xintiiPow{7}{13}\to\SevenToThePowerThirteen
\meaning\SevenToThePowerThirteen\par
```

```
macro:->7500
macro:->2500
macro:->\xintiiDivision {1000000000000}{133333333}
macro:->{7500}{2500}
macro:->96889010407
```

Two special cases:

- if after this initial expansion no brace is found immediately after `\xintAssign`, it is assumed that there is only one control sequence following `\to`, and this control sequence is then defined via `\def` (or what is set-up by the optional parameter) to expand to the material between `\xintAssign` and `\to`.
- if the material between `\xintAssign` and `\to` is enclosed in two brace pairs, the first brace pair is removed, then the *f-expansion* is immediately stopped by the inner brace pair, hence `\xintAssign` now finds a unique item and thus defines only a single macro to be this item, which is now stripped of the second pair of braces.

Note: prior to release 1.09j, `\xintAssign` did an `\edef` by default for each item assignment but it now does `\def` corresponding to no or empty optional parameter.

It is allowed for the successive braced items to be separated by spaces. They are removed during the assignments. But if a single macro is defined (which happens if the argument after *f-expansion* does not start with a brace), naturally the scooped up material has all intervening spaces, as it is considered a single item. But an upfront initial space will have been absorbed by *f-expansion*.

```
\def\X{ {a} {b} {c} {d} }\def\Y { u {a} {b} {c} {d} }
\xintAssign\X\to\A\B\C\D
\xintAssign\Y\to\Z
\meaning\A, \meaning\B, \meaning\C, \meaning\D+++\\new\\line
\meaning\Z+++\\par
```

```
macro:->a, macro:->b, macro:->c, macro:->d+++
```

```
macro:->u {a} {b} {c} {d} +++
```

As usual successive space characters in input make for a single TeX space token.

*source*

## 6.23. `\xintAssignArray`

`\xintAssignArray`*(braced things)*`\to\myArray` first expands fully what comes immediately after `\xintAssignArray` and expects to find a list of braced things `{A}{B}...` (or tokens). It then defines `\myArray` as a macro with one parameter, such that `\myArray{x}` expands to give the *x*th braced thing of this original list (the argument `{x}` itself is fed to a `\numexpr` by `\myArray`, and `\myArray` expands in two steps to its output). With 0 as parameter, `\myArray{0}` returns the number *M* of elements of the array so that the successive elements are `\myArray{1}`, ..., `\myArray{M}`.

```
\xintAssignArray \xintBezout {1000}{113}\to\Bez
```

will set `\Bez{0}` to 3, `\Bez{1}` to -20, `\Bez{2}` to 177, and `\Bez{3}` to 1:  $-20 \times 1000 + 177 \times 113 = 1$ . This macro is incompatible with expansion-only contexts.

`\xintAssignArray` admits an optional parameter, for example `\xintAssignArray [e]` means that the definitions of the macros will be made with `\edef`. The empty optional parameter (default) means that definitions are done with `\def`. Other possibilities: `[]`, `[o]`, `[oo]`, `[f]`. Contrarily to `\xintAssign` one can not use the *g* here to make the definitions global. For this, one should rather do `\xintAssignArray` within a group starting with `\globaldefs 1`.

*source*

## 6.24. `\xintDigitsOf`

*fN* This is a synonym for `\xintAssignArray`, to be used to define an array giving all the digits of a given (positive, else the minus sign will be treated as first item) number.

```
\xintDigitsOf\xintiiPow {7}{500}\to\digits
```

[TOC](#)

[TOC](#), [Start here](#), [xintexpr](#), [xintexpr \(old doc\)](#), [xinttrig](#), [xintlog](#), [xinttools](#), [Examples](#), [xint bundle](#)

$7^{500}$  has `\digits{0}=423` digits, and the 123rd among them (starting from the most significant) is `\digits{123}=3`.

## 6.25. `\xintRelaxArray`

`\xintRelaxArray\myArray` (globally) sets to `\relax` all macros which were defined by the previous `\xintAssignArray` with `\myArray` as array macro.



## 7. Additional (old) examples with **xinttools** or **xintexpr** or both

|    |                                          |     |    |                                           |     |
|----|------------------------------------------|-----|----|-------------------------------------------|-----|
| .1 | More examples with dummy variables.....  | 108 | .5 | A table of factorizations .....           | 115 |
| .2 | Completely expandable prime test .....   | 109 | .6 | Another table of primes.....              | 116 |
| .3 | Another completely expandable prime test | 110 | .7 | Factorizing again .....                   | 118 |
| .4 | Miller-Rabin Pseudo-Primality expandably | 112 | .8 | The Quick Sort algorithm illustrated..... | 119 |

Note: **xintexpr.sty** automatically loads **xinttools.sty**.

The examples given here start to feel dated and are currently in need of some rewrite to better illustrate newer features of the package.

### 7.1. More examples with dummy variables

These examples were first added to this manual at the time of the 1.1 release (2014/10/29).

Prime numbers are always cool

```
\xinttheiexpr seq((seq((subs((x/:m)?{(m*m>x)?{1}{0}}{-1},m=2n+1))
??{break(0)}{omit}{break(1)},n=1++))){x}{omit},
x=10001..[2]..10200)\relax
```

Prime numbers are always cool 10007, 10009, 10037, 10039, 10061, 10067, 10069, 10079, 10091, 10093, 10099, 10103, 10111, 10133, 10139, 10141, 10151, 10159, 10163, 10169, 10177, 10181, 10193

The syntax in this last example may look a bit involved (... and it is so I admit). First `x/:m` computes `x modulo m` (this is the modulo with respect to floored division). The `(x)?{yes}{no}` construct checks if `x` (which *must* be within parentheses) is true or false, i.e. non zero or zero. It then executes either the **yes** or the **no** branch, the non chosen branch is *not* evaluated. Thus if `m` divides `x` we are in the second (`'false'`) branch. This gives a `-1`. This `-1` is the argument to a `??` branch which is of the type `(y)?{y<0}{y=0}{y>0}`, thus here the `y<0`, i.e., `break(0)` is chosen. This `0` is thus given to another `?` which consequently chooses **omit**, hence the number is not kept in the list. The numbers which survive are the prime numbers.

The first Fibonacci number beyond  $|2^{64}|$ ,  $|2^{64}|$ , and the index are respectively

```
\xinttheiexpr subs(iterr(0,1;(@1>N)?{break(@1,N,i)}{@1+@2},i=1++),N=2^64)\relax.
```

The first Fibonacci number beyond  $2^{64}$ ,  $2^{64}$ , and the index are respectively 19740274219868223167, 18446744073709551616, 94.

One more recursion:

```
\def\syr #1{\xinttheiexpr
rseq(#1; (@<=1)?{break(i)}{odd(@)?{3@+1}{@//2}},i=0++)\relax}
```

The  $3x+1$  problem: \syr{231}\par

The  $3x+1$  problem: 231, 694, 347, 1042, 521, 1564, 782, 391, 1174, 587, 1762, 881, 2644, 1322, 661, 1984, 992, 496, 248, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 127

OK, a final one:

```
\def\syrMax #1{\xinttheiexpr iterr(#1,#1;even(i)?
{(@2<=1)?{break(@1,i//2)}
{odd(@2)?{3@2+1}{@2//2}}}
{(@1>@2)?{@1}{@2}},i=0++)\relax }
```

With initial value 1161, the maximal intermediate value and the number of steps needed to reach 1 are respectively \syrMax{1161}.\par

With initial value 1161, the maximal intermediate value and the number of steps needed to reach 1 are respectively 190996, 181.

Look at the [Brent-Salamin algorithm implementation](#) for a more interesting recursion.

## 7.2. Completely expandable prime test

Let us now construct a completely expandable macro which returns 1 if its given input is prime and 0 if not:

```
\def\remainder #1#2{\the\numexpr #1-(#1/#2)*#2\relax }
\def\IsPrime #1%
{\xintANDof {\xintApply {\remainder {#1}}{\xintSeq {2}}{\xintiiSqrt{#1}}}}
```

This uses `\xintiiSqrt` and assumes its input is at least 5. Rather than `xint`'s own `\xintiiRem` we used a quicker `\numexpr` expression as we are dealing with short integers. Also we used `\xintANDof` which will return 1 only if all the items are non-zero. The macro is a bit silly with an even input, ok, let's enhance it to detect an even input:

```
\def\IsPrime #1%
{\xintiifOdd {#1}
 {\xintANDof % odd case
  {\xintApply {\remainder {#1}}
   {\xintSeq [2]{3}{\xintiiSqrt{#1}}}%
 }%
 }
 {\xintifEq {#1}{2}{1}{0}}%
 }
```

We used the `xint` expandable tests (on big integers or fractions) in order for `\IsPrime` to be *f-expandable*.

Our integers are short, but without `\expandafter`'s with `\@firstoftwo`, or some other related techniques, direct use of `\ifnum... \fi` tests is dangerous. So to make the macro more efficient we are going to use the expandable tests provided by the package [etoolbox](#)<sup>36</sup>. The macro becomes:

```
\def\IsPrime #1%
{\ifnumodd {#1}
 {\xintANDof % odd case
  {\xintApply {\remainder {#1}}{\xintSeq [2]{3}{\xintiiSqrt{#1}}}}
 {\ifnumequal {#1}{2}{1}{0}}}
```

In the odd case however we have to assume the integer is at least 7, as `\xintSeq` generates an empty list if `#1=3` or `5`, and `\xintANDof` returns 1 when supplied an empty list. Let us ease up a bit `\xintANDof`'s work by letting it work on only 0's and 1's. We could use:

```
\def\IsNotDivisibleBy #1#2%
{\ifnum\numexpr #1-(#1/#2)*#2=0 \expandafter 0\else \expandafter 1\fi}
```

where the `\expandafter`'s are crucial for this macro to be *f-expandable* and hence work within the applied `\xintANDof`. Anyhow, now that we have loaded [etoolbox](#), we might as well use:

```
\newcommand{\IsNotDivisibleBy}[2]{\ifnumequal{#1-(#1/#2)*#2}{0}{0}{1}}
```

Let us enhance our prime macro to work also on the small primes:

```
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
{\ifnumodd {#1}
 {\ifnumless {#1}{8}
  {\ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
  {\xintANDof
   {\xintApply
    {\IsNotDivisibleBy {#1}}{\xintSeq [2]{3}{\xintiiSqrt{#1}}}%
   }}% END OF THE ODD BRANCH
 {\ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
 }
```

The input is still assumed positive. There is a deliberate blank before `\IsNotDivisibleBy` to use this feature of `\xintApply`: a space stops the expansion of the applied macro (and disappears). This expansion will be done by `\xintANDof`, which has been designed to skip everything as soon as

<sup>36</sup> <http://ctan.org/pkg/etoolbox>

it finds a false (i.e. zero) input. This way, the efficiency is considerably improved.

We did generate via the `\xintSeq` too many potential divisors though. Later sections give two variants: one with `\xintilooop` (subsection 7.3) which is still expandable and another one (subsection 7.6) which is a close variant of the `\IsPrime` code above but with the `\xintFor` loop, thus breaking expandability. The `xintilooop variant` does not first evaluate the integer square root, the `xintFor variant` still does. I did not compare their efficiencies.

Let us construct with this expandable primality test a table of the prime numbers up to 1000. We need to count how many we have in order to know how many tab stops one should add in the last row.<sup>37</sup> There is some subtlety for this last row. Turns out to be better to insert a `\\` only when we know for sure we are starting a new row; this is how we have designed the `\OneCell` macro. And for the last row, there are many ways, we use again `\xintApplyUnbraced` but with a macro which gobbles its argument and replaces it with a tabulation character. The `\xintFor*` macro would be more elegant here.

```
\newcounter{primecount}
\newcounter{cellcount}
\newcommand{\NbOfColumns}{13}
\newcommand{\OneCell}[1]{%
  \ifnumequal{\IsPrime{#1}}{1}
  {\stepcounter{primecount}
   \ifnumequal{\value{cellcount}}{\NbOfColumns}
   {\setcounter{cellcount}{1}#1}
   {\&\stepcounter{cellcount}#1}%
  } % was prime
  {}% not a prime, nothing to do
}
\newcommand{\OneTab}[1]{&}
\begin{tabular}{|*{\NbOfColumns}{r}|}
\hline
2 \setcounter{cellcount}{1}\setcounter{primecount}{1}%
\xintApplyUnbraced \OneCell {\xintSeq [2]{3}{999}}%
\xintApplyUnbraced \OneTab
{\xintSeq [1]{1}{\the\numexpr\nbOfColumns-\value{cellcount}\relax}}%
\\
\hline
\end{tabular}
There are \arabic{primecount} prime numbers up to 1000.
```

The table has been put in `float` which appears on the following page. We had to be careful to use in the last row `\xintSeq` with its optional argument `[1]` so as to not generate a decreasing sequence from 1 to 0, but really an empty sequence in case the row turns out to already have all its cells (which doesn't happen here but would with a number of columns dividing 168).

### 7.3. Another completely expandable prime test

The `\IsPrime` macro from subsection 7.2 checked expandably if a (short) integer was prime, here is a partial rewrite using `\xintilooop`. We use the `etoolbox` expandable conditionals for convenience, but not everywhere as `\xintilooopindex` can not be evaluated while being braced. This is also the reason why `\xintbreakilooopanddo` is delimited, and the next macro `\SmallestFactor` which returns the smallest prime factor exemplifies that. One could write more efficient completely expandable routines, the aim here was only to illustrate use of the general purpose `\xintilooop`. A little table giving the first values of `\SmallestFactor` follows, its coding uses `\xintFor`, which is described later; none of this uses count registers.

<sup>37</sup> although a tabular row may have less tabs than in the preamble, there is a problem with the | vertical rule, if one does that.

|     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 2   | 3   | 5   | 7   | 11  | 13  | 17  | 19  | 23  | 29  | 31  | 37  | 41  |
| 43  | 47  | 53  | 59  | 61  | 67  | 71  | 73  | 79  | 83  | 89  | 97  | 101 |
| 103 | 107 | 109 | 113 | 127 | 131 | 137 | 139 | 149 | 151 | 157 | 163 | 167 |
| 173 | 179 | 181 | 191 | 193 | 197 | 199 | 211 | 223 | 227 | 229 | 233 | 239 |
| 241 | 251 | 257 | 263 | 269 | 271 | 277 | 281 | 283 | 293 | 307 | 311 | 313 |
| 317 | 331 | 337 | 347 | 349 | 353 | 359 | 367 | 373 | 379 | 383 | 389 | 397 |
| 401 | 409 | 419 | 421 | 431 | 433 | 439 | 443 | 449 | 457 | 461 | 463 | 467 |
| 479 | 487 | 491 | 499 | 503 | 509 | 521 | 523 | 541 | 547 | 557 | 563 | 569 |
| 571 | 577 | 587 | 593 | 599 | 601 | 607 | 613 | 617 | 619 | 631 | 641 | 643 |
| 647 | 653 | 659 | 661 | 673 | 677 | 683 | 691 | 701 | 709 | 719 | 727 | 733 |
| 739 | 743 | 751 | 757 | 761 | 769 | 773 | 787 | 797 | 809 | 811 | 821 | 823 |
| 827 | 829 | 839 | 853 | 857 | 859 | 863 | 877 | 881 | 883 | 887 | 907 | 911 |
| 919 | 929 | 937 | 941 | 947 | 953 | 967 | 971 | 977 | 983 | 991 | 997 |     |

There are 168 prime numbers up to 1000.

```
% clean up possible left-over mess from previous examples
\let\IsPrime\undefined \let\SmallestFactor\undefined
\newcommand{\IsPrime}[1] % returns 1 if #1 is prime, and 0 if not
{
  \ifnumodd {#1}
  {
    \ifnumless {#1}{8}
    {
      \ifnumequal{#1}{1}{0}{1}}% 3,5,7 are primes
    {
      \if
      \xintloop [3+2]
      \ifnum#1<\numexpr\xintloopindex*\xintloopindex\relax
      \expandafter\xintbreaklooppando\expandafter1\expandafter.%
      \fi
      \ifnum#1=\numexpr (#1/\xintloopindex)*\xintloopindex\relax
      \else
      \repeat 00\expandafter0\else\expandafter1\fi
    }%
  }% END OF THE ODD BRANCH
  \ifnumequal {#1}{2}{1}{0}}% EVEN BRANCH
}
\catcode`_ 11
\newcommand{\SmallestFactor}[1] % returns the smallest prime factor of #1>1
{
  \ifnumodd {#1}
  {
    \ifnumless {#1}{8}
    {
      {#1}% 3,5,7 are primes
    {
      \xintloop [3+2]
      \ifnum#1<\numexpr\xintloopindex*\xintloopindex\relax
      \xint_afterfi{\xintbreaklooppando#1.}%
      \fi
      \ifnum#1=\numexpr (#1/\xintloopindex)*\xintloopindex\relax
      \xint_afterfi{\expandafter\xintbreaklooppando\xintloopindex.}%
      \fi
      \iftrue\repeat
    }%
  }% END OF THE ODD BRANCH
  {2}% EVEN BRANCH
}
\catcode`_ 8
{\centering
```

```

\begin{tabular}{|c|*{10}c|}
\hline
\intFor #1 in {0,1,2,3,4,5,6,7,8,9}\do {\&\bfseries #1}\
\hline
\bfseries 0&--&--&2&3&2&5&2&7&2&3\
\intFor #1 in {1,2,3,4,5,6,7,8,9}\do
{\bfseries #1%
\intFor #2 in {0,1,2,3,4,5,6,7,8,9}\do
{\&\SmallestFactor{#1#2}}\}%
\hline
\end{tabular}\par
}

```

|   | 0  | 1  | 2 | 3  | 4 | 5 | 6 | 7  | 8 | 9  |
|---|----|----|---|----|---|---|---|----|---|----|
| 0 | -- | -- | 2 | 3  | 2 | 5 | 2 | 7  | 2 | 3  |
| 1 | 2  | 11 | 2 | 13 | 2 | 3 | 2 | 17 | 2 | 19 |
| 2 | 2  | 3  | 2 | 23 | 2 | 5 | 2 | 3  | 2 | 29 |
| 3 | 2  | 31 | 2 | 3  | 2 | 5 | 2 | 37 | 2 | 3  |
| 4 | 2  | 41 | 2 | 43 | 2 | 3 | 2 | 47 | 2 | 7  |
| 5 | 2  | 3  | 2 | 53 | 2 | 5 | 2 | 3  | 2 | 59 |
| 6 | 2  | 61 | 2 | 3  | 2 | 5 | 2 | 67 | 2 | 3  |
| 7 | 2  | 71 | 2 | 73 | 2 | 3 | 2 | 7  | 2 | 79 |
| 8 | 2  | 3  | 2 | 83 | 2 | 5 | 2 | 3  | 2 | 89 |
| 9 | 2  | 7  | 2 | 3  | 2 | 5 | 2 | 97 | 2 | 3  |

## 7.4. Miller-Rabin Pseudo-Primality expandably

The `isPseudoPrime(n)` is usable in `\xintiiexpr`-essions and establishes if its (positive) argument is a Miller-Rabin PseudoPrime to the bases 2, 3, 5, 7, 11, 13, 17. If this is true and  $n < 341550071728321$  (which has 15 digits) then  $n$  really is a prime number.

Similarly  $n = 3825123056546413051$  (19 digits) is the smallest composite number which is a strong pseudo prime for bases 2, 3, 5, 7, 11, 13, 17, 19 and 23. It is easy to extend the code below to include these additional tests (we could make the list of tested bases an argument too, now that I think about it.)

For more information see

[https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin\\_primality\\_test#Deterministic\\_variants\\_of\\_the\\_test](https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test) and

[http://primes.utm.edu/prove/prove2\\_3.html](http://primes.utm.edu/prove/prove2_3.html)

In particular, according to JÄESCHKE *On strong pseudoprimes to several bases*, Math. Comp., 61 (1993) 915-926, if  $n < 4,759,123,141$  it is enough to establish Rabin-Miller pseudo-primality to bases  $a = 2, 7, 61$  to prove that  $n$  is prime. This range is enough for  $\mathbb{T}_X$  numbers and we could then write a very fast expandable primality test for such numbers using only `\numexpr`. Left as an exercise...

```

% I ----- Modular Exponentiation
% Computes x^m modulo n (with m non negative).
% We will always use it with 1 < x < n
%
% With xint 1.4 we should use ? and ?? (although in the case at hand ifsgn()
% and if() would be ok but I should not say that).
%
\xintdefiifunc powmod_a(x, m, n) :=
  isone(m)?
    % m=1, return x modulo n
    { x /: n }

```

```

        % m > 1 test if odd or even and do recursive call
        {   odd(m)? {   x*sqr(powmod_a(x, m//2, n)) /: n }
                {   sqr(powmod_a(x, m//2, n)) /: n }
        }
    };
\intdefiifunc powmod(x, m, n) := (m)?{powmod_a(x, m, n)}{1};

%% Syntax used before xint 1.4:
% \intdefiifunc powmod_a(x, m, n) :=
%     ifone(m,
%         % m=1, return x modulo n
%         x /: n,
%         % m > 1 test if odd or even and do recursive call
%         if(odd(m), (x*sqr(powmod_a(x, m//2, n))) /: n,
%             sqr(powmod_a(x, m//2, n)) /: n
%         )
%     );
% \intdefiifunc powmod(x, m, n) := if(m, powmod_a(x, m, n), 1);

% II ----- Miller-Rabin compositeness witness

% n=2^k m + 1 with m odd and k at least 1

% Choose 1<x<n.
% compute y=x^m modulo n
% if equals 1 we can't say anything
% if equals n-1 we can't say anything
% else put j=1, and
% compute repeatedly the square, incrementing j by 1 each time,
% thus always we have y^{2^{j-1}}
% -> if at some point n-1 mod n found, we can't say anything and break out
% -> if however we never find n-1 mod n before reaching
%     z=y^{2^{k-1}} with j=k
%     we then have z^2=x^{n-1}.
% Suppose z is not -1 mod n. If z^2 is 1 mod n, then n can be prime only if
% z is 1 mod n, and we can go back up, until initial y, and we have already
% excluded y=1. Thus if z is not -1 mod n and z^2 is 1 then n is not prime.
% But if z^2 is not 1, then n is not prime by Fermat. Hence (z not -1 mod n)
% implies (n is composite). (Miller test)

% let's use again xintexpr indecipherable (except to author) syntax. Of course
% doing it with macros only would be faster.

% Here \intdefiifunc is not usable because not compatible with iter, break, ...
% but \intNewFunction comes to the rescue.

\intNewFunction{isCompositeWitness}[4]{% x=#1, n=#2, m=#3, k=#4
    subs((y==1)?{0}
        {iter(y;(j==#4)?{break(!(@==#2-1))}
            {(@==#2-1)?{break(0)}{sqr(@)/:#2}},j=1++)}
        ,y=powmod(#1,#3,#2))}

```

```

% added note (2018/03/07) it is possible in the above that m=#3 is never
% zero, so we should rather call powmod_a for a small gain, but I don't
% have time to re-read the code comments and settle this.

% III ----- Strong Pseudo Primes

% cf
% http://oeis.org/A014233
% http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html>
% http://mathworld.wolfram.com/StrongPseudoprime.html>

% check if positive integer <49 is a prime.
% 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
\def\IsVerySmallPrime #1%
  {\ifnum#1=1 \xintdothis0\fi
   \ifnum#1=2 \xintdothis1\fi
   \ifnum#1=3 \xintdothis1\fi
   \ifnum#1=5 \xintdothis1\fi
   \ifnum#1=\numexpr (#1/2)*2\relax\xintdothis0\fi
   \ifnum#1=\numexpr (#1/3)*3\relax\xintdothis0\fi
   \ifnum#1=\numexpr (#1/5)*5\relax\xintdothis0\fi
   \xintorthat 1}

\xintNewFunction{isPseudoPrime}[1]{% n = #1
  (#1<49)?% use ? syntax to evaluate only what is needed
% prior to 1.4 we had \xintthe#1 here but the actual tokens represented
% by this #1 when isPseudoPrime() function expands have changed and
% the correct way is now \xintiieval{#1} to hand over explicit digits to
% the \IsVerySmallPrime macro.
  {\IsVerySmallPrime{\xintiieval{#1}}}
  {(even(#1))?
   {0}
   {subs(%
    % L expands to two values m, k hence isCompositeWitness does get
    % its four variables x, n, m, k
    isCompositeWitness(2, #1, L)?
    {0}%
    isCompositeWitness(3, #1, L)?
    {0}%
    isCompositeWitness(5, #1, L)?
    {0}%
    isCompositeWitness(7, #1, L)?
    {0}%
    % above enough for N<3215031751 hence all TeX numbers
    isCompositeWitness(11, #1, L)?
    {0}%
    % above enough for N<2152302898747, hence all 12-digits numbers
    isCompositeWitness(13, #1, L)?
    {0}%
    % above enough for N<3474749660383
    isCompositeWitness(17, #1, L)?
    {0}%
  }
}

```



```
% above enough for N<341550071728321
    {1}%
    }% not needed to comment-out end of lines spaces inside
    }% \xintexpr but this is too much of a habit for me with TeX!
    }% I left some after the ? characters.
    }%
    }%
    }% this computes (m, k) such that n = 2^k m + 1, m odd, k>=1
    , L=iter(#1//2;(even(@))?{@//2}{break(@,k)},k=1++)}%
    }%
    }%
}

% if needed:
%\def\IsPseudoPrime #1{\xinttheiiexpr isPseudoPrime(#1)\relax}

\noindent The smallest prime number at least equal to 3141592653589 is
\xintiexpr
    seq(isPseudoPrime(3141592653589+n)?
        {break(3141592653589+n)}{omit}, n=0++)\relax.
% we could not use 3141592653589++ syntax because it works only with TeX numbers
\par
```

The smallest prime number at least equal to 3141592653589 is 3141592653601.

## 7.5. A table of factorizations

As one more example with `\xintilooop` let us use an alignment to display the factorization of some numbers. The loop will actually only play a minor rôle here, just handling the row index, the row contents being almost entirely produced via a macro `\factorize`. The factorizing macro does not use `\xintilooop` as it didn't appear to be the convenient tool. As `\factorize` will have to be used on `\xintilooopindex`, it has been defined as a delimited macro.

To spare some fractions of a second in the compilation time of this document (which has many many other things to do), 2147483629 and 2147483647, which turn out to be prime numbers, are not given to `\factorize` but just typeset directly; this illustrates use of `\xintilooopskipnext`.

The code next generates a `table` which has been made into a float appearing on page 117. Here is now the code for factorization; the conditionals use the package provided `\xint_firstoftwo` and `\xint_secondoftwo`, one could have employed rather  $\TeX$ 's own `\@firstoftwo` and `\@secondoftwo`, or, simpler still in  $\TeX$  context, the `\ifnumequal`, `\ifnumless` . . . , utilities from the package `etoolbox` which do exactly that under the hood. Only  $\TeX$  acceptable numbers are treated here, but it would be easy to make a translation and use the `xint` macros, thus extending the scope to big numbers; naturally up to a cost in speed.

The reason for some strange looking expressions is to avoid arithmetic overflow.

```
\catcode`_ 11
\def\abortfactorize #1\xint_secondoftwo\fi #2#3{\fi}

\def\factorize #1.{\ifnum#1=1 \abortfactorize\fi
    \ifnum\numexpr #1-2=\numexpr ((#1/2)-1)*2\relax
        \expandafter\xint_firstoftwo
    \else\expandafter\xint_secondoftwo
    \fi
    {2&\expandafter\factorize\the\numexpr#1/2.}%
    {\factorize_b #1.3.}}%
```

```
\def\factorize_b #1.#2.{\ifnum#1=1 \abortfactorize\fi
  \ifnum\numexpr #1-(#2-1)*#2<#2
    #1\abortfactorize
  \fi
  \ifnum \numexpr #1-#2=\numexpr ((#1/#2)-1)*#2\relax
    \expandafter\xint_firstoftwo
  \else\expandafter\xint_secondoftwo
  \fi
  {#2&\expandafter\factorize_b\the\numexpr#1/#2.#2.}%
  {\expandafter\factorize_b\the\numexpr #1\expandafter.%
    \the\numexpr #2+2.}%
}

\catcode`_ 8
\begin{figure*}[ht!]
\centering\phantomsection\label{floatfactorize}\normalcolor
\tabskiplex
\centeredline{\vbox{\halign {\hfil\strut#\hfil&\hfil#\hfil\cr\noalign{\hrule}
  \xintilop ["7FFFFFFE0+1]
  \expandafter\bfseries\xintilopindex &
  \ifnum\xintilopindex="7FFFFFFED
    \number"7FFFFFFED\cr\noalign{\hrule}
  \expandafter\xintilopskiptonext
  \fi
  \expandafter\factorize\xintilopindex.\cr\noalign{\hrule}
  \ifnum\xintilopindex<"7FFFFFFE
  \repeat
  \bfseries \number"7FFFFFFF&\number "7FFFFFFF\cr\noalign{\hrule}
}}}
\centeredline{A table of factorizations}
\end{figure*}
```

## 7.6. Another table of primes

As a further example, let us dynamically generate a tabular with the first 50 prime numbers after 12345. First we need a macro to test if a (short) number is prime. Such a completely expandable macro was given in [subsection 7.2](#), here we consider a variant which will be slightly more efficient. This new `\IsPrime` has two parameters. The first one is a macro which it redefines to expand to the result of the primality test applied to the second argument. For convenience we use the `etoolbox` wrappers to various `\ifnum` tests, although here there isn't anymore the constraint of complete expandability (but using explicit `\if.. \fi` in tabulars has its quirks); equivalent tests are provided by `xint`, but they have some overhead as they are able to deal with arbitrarily big integers.

```
\def\IsPrime #1#2% #1=\Result, #2=tested number (assumed >0).
{\edef\TheNumber {\the\numexpr #2}% hence #2 may be a count or \numexpr.
\ifnumodd {\TheNumber}
{\ifnumgreater {\TheNumber}{1}
{\edef\ItsSquareRoot{\xintiiSqrt \TheNumber}%
\xintFor ##1 in {\xintintegers [3+2]}\do
{\ifnumgreater {##1}{\ItsSquareRoot} % ##1 is a \numexpr.
{\def#1{1}\xintBreakFor}
}%
\ifnumequal {\TheNumber}{(\TheNumber/##1)*##1}
{\def#1{0}\xintBreakFor }
}%
}
```

|            |            |           |           |         |          |         |        |         |
|------------|------------|-----------|-----------|---------|----------|---------|--------|---------|
| 2147483616 | 2          | 2         | 2         | 2       | 2        | 3       | 2731   | 8191    |
| 2147483617 | 6733       | 318949    |           |         |          |         |        |         |
| 2147483618 | 2          | 7         | 367       | 417961  |          |         |        |         |
| 2147483619 | 3          | 3         | 23        | 353     | 29389    |         |        |         |
| 2147483620 | 2          | 2         | 5         | 4603    | 23327    |         |        |         |
| 2147483621 | 14741      | 145681    |           |         |          |         |        |         |
| 2147483622 | 2          | 3         | 17        | 467     | 45083    |         |        |         |
| 2147483623 | 79         | 967       | 28111     |         |          |         |        |         |
| 2147483624 | 2          | 2         | 2         | 11      | 13       | 1877171 |        |         |
| 2147483625 | 3          | 5         | 5         | 5       | 7        | 199     | 4111   |         |
| 2147483626 | 2          | 19        | 37        | 1527371 |          |         |        |         |
| 2147483627 | 47         | 53        | 862097    |         |          |         |        |         |
| 2147483628 | 2          | 2         | 3         | 3       | 59652323 |         |        |         |
| 2147483629 | 2147483629 |           |           |         |          |         |        |         |
| 2147483630 | 2          | 5         | 6553      | 32771   |          |         |        |         |
| 2147483631 | 3          | 137       | 263       | 19867   |          |         |        |         |
| 2147483632 | 2          | 2         | 2         | 2       | 7        | 73      | 262657 |         |
| 2147483633 | 5843       | 367531    |           |         |          |         |        |         |
| 2147483634 | 2          | 3         | 12097     | 29587   |          |         |        |         |
| 2147483635 | 5          | 11        | 337       | 115861  |          |         |        |         |
| 2147483636 | 2          | 2         | 536870909 |         |          |         |        |         |
| 2147483637 | 3          | 3         | 3         | 13      | 6118187  |         |        |         |
| 2147483638 | 2          | 2969      | 361651    |         |          |         |        |         |
| 2147483639 | 7          | 17        | 18046081  |         |          |         |        |         |
| 2147483640 | 2          | 2         | 2         | 3       | 5        | 29      | 43     | 113 127 |
| 2147483641 | 2699       | 795659    |           |         |          |         |        |         |
| 2147483642 | 2          | 23        | 46684427  |         |          |         |        |         |
| 2147483643 | 3          | 715827881 |           |         |          |         |        |         |
| 2147483644 | 2          | 2         | 233       | 1103    | 2089     |         |        |         |
| 2147483645 | 5          | 19        | 22605091  |         |          |         |        |         |
| 2147483646 | 2          | 3         | 3         | 7       | 11       | 31      | 151    | 331     |
| 2147483647 | 2147483647 |           |           |         |          |         |        |         |

A table of factorizations

```

}}
{\def#1{0}}% 1 is not prime
{\ifnumequal {\TheNumber}{2}{\def#1{1}}{\def#1{0}}}%
}

```

As we used `\xintFor` inside a macro we had to double the `#` in its `#1` parameter. Here is now the code which creates the prime table (the table has been put in a `float`, which should be found on page 118):

```

\newcounter{primecount}
\newcounter{cellcount}
\begin{figure*}[ht!]
\centering
\begin{tabular}{|*{7}c|}
\hline
\setcounter{primecount}{0}\setcounter{cellcount}{0}%
\xintFor #1 in {\xintintegers [12345+2]} \do
% #1 is a \numexpr.
{\IsPrime\Result{#1}}%

```

```
\ifnumgreater{\Result}{0}
{\stepcounter{primecount}%
\stepcounter{cellcount}%
\ifnumequal {\value{cellcount}}{7}
{\the#1 \\\setcounter{cellcount}{0}}
{\the#1 &}}
}%
\ifnumequal {\value{primecount}}{50}
{\xintBreakForAndDo
{\multicolumn {6}{l}{These are the first 50 primes after 12345.}}
}%
}\hline
\end{tabular}
\end{figure*}
```

|       |                                            |       |       |       |       |       |
|-------|--------------------------------------------|-------|-------|-------|-------|-------|
| 12347 | 12373                                      | 12377 | 12379 | 12391 | 12401 | 12409 |
| 12413 | 12421                                      | 12433 | 12437 | 12451 | 12457 | 12473 |
| 12479 | 12487                                      | 12491 | 12497 | 12503 | 12511 | 12517 |
| 12527 | 12539                                      | 12541 | 12547 | 12553 | 12569 | 12577 |
| 12583 | 12589                                      | 12601 | 12611 | 12613 | 12619 | 12637 |
| 12641 | 12647                                      | 12653 | 12659 | 12671 | 12689 | 12697 |
| 12703 | 12713                                      | 12721 | 12739 | 12743 | 12757 | 12763 |
| 12781 | These are the first 50 primes after 12345. |       |       |       |       |       |

## 7.7. Factorizing again

Here is an *f-expandable* macro which computes the factors of an integer. It uses the *xint* macros only.

```
\catcode\@ 11
\let\factorize\relax
\newcommand\Factorize [1]
{\romannumeral0\expandafter\factorize\expandafter{\romannumeral-`0#1}}%
\newcommand\factorize [1]{\xintiiifOne{#1}{ 1}{\factors@a #1.#1};}%
\def\factors@a #1.{\xintiiifOdd{#1}
{\factors@c 3.#1}%
{\expandafter\factorize@b \expandafter1\expandafter.\romannumeral0\xinthalff{#1}.}}%
\def\factorize@b #1.#2.{\xintiiifOne{#2}
{\factors@end {2, #1}}%
{\xintiiifOdd{#2}{\factors@c 3.#2.{2, #1}}%
{\expandafter\factorize@b \the\numexpr #1+\@ne\expandafter.%
\romannumeral0\xinthalff{#2}.}}%
}%
\def\factorize@c #1.#2.{%
\expandafter\factorize@d\romannumeral0\xintiividivision {#2}{#1}{#1}{#2}%
}%
\def\factorize@d #1#2#3#4{\xintiiifNotZero{#2}
{\xintiiifGt{#3}{#1}
{\factors@end {#4, 1}}% ultimate quotient is a prime with power 1
{\expandafter\factorize@c\the\numexpr #3+\tw@.#4.}}%
{\factors@e 1.#3.#1}%
}%
\def\factorize@e #1.#2.#3.{\xintiiifOne{#3}
```

```
{\factors@end {#2, #1}}%
{\expandafter\factors@f\romannumeral0\xintiidivision {#3}{#2}{#1}{#2}{#3}}%
}%
\def\factors@f #1#2#3#4#5{\xintiiifNotZero{#2}
{\expandafter\factors@c\the\numexpr #4+\tw@.#5.{#4, #3}}%
{\expandafter\factors@e\the\numexpr #3+\@ne.#4.#1.}%
}%
\def\factors@end #1;\xintlistwithsep{, }{\xintRevWithBraces {#1}}}%
\catcode`@ 12
```

The macro will be acceptably efficient only with numbers having somewhat small prime factors.

```
\Factorize{16246355912554185673266068721806243461403654781833}
```

```
16246355912554185673266068721806243461403654781833, 13, 5, 17, 8, 29, 5, 37, 6, 41, 4, 59, 6
```

It puts a little stress on the input save stack in order not be bothered with previously gathered things.<sup>38</sup>

Its output is a comma separated list with the number first, then its prime factors with multiplicity. Let's produce something prettier:

```
\catcode`_ 11
\def\ShowFactors #1{\expandafter
\ShowFactors_a\romannumeral-`0\Factorize{#1},\relax,\relax,}
\def\ShowFactors_a #1,{#1=\ShowFactors_b}
\def\ShowFactors_b #1,#2,{\if\relax#1\else#1^{#2}\expandafter\ShowFactors_b\fi}
\catcode`_ 8
$$\ShowFactors{16246355912554185673266068721806243461403654781833}$$
```

```
16246355912554185673266068721806243461403654781833 = 135178295376414596
```

If we only considered small integers, we could write pure `\numexpr` methods which would be very much faster (especially if we had a table of small primes prepared first) but still ridiculously slow compared to any non expandable implementation, not to mention use of programming languages directly accessing the CPU registers...

## 7.8. The Quick Sort algorithm illustrated

First a completely expandable macro which sorts a comma separated list of numbers.<sup>39</sup>

The `\QSx` macro expands its list argument, which may thus be a macro; its comma separated items must expand to integers or decimal numbers or fractions or scientific notation as acceptable to `xintfrac`, but if an item is itself some (expandable) macro, this macro will be expanded each time the item is considered in a comparison test! This is actually good if the macro expands in one step to the digits, and there are many many digits, but bad if the macro needs to do many computations. Thus `\QSx` should be used with either explicit numbers or with items being macros expanding in one step to the numbers (particularly if these numbers are very big).

If the interest is only in TeX integers, then one should replace the `\xintifCmp` macro with a suitable conditional, possibly helped by tools such as `\ifnumgreater`, `\ifnumequal` and `\ifnumless` from `etoolbox` (TeX only; I didn't see a direct equivalent to `\xintifCmp`.) Or, if we are dealing with decimal numbers with at most four+four digits, then one should use suitable `\ifdim` tests. Naturally this will boost consequently the speed, from having skipped all the overhead in parsing fractions and scientific numbers as are acceptable by `xintfrac` macros, and subsequent treatment.

```
% THE QUICK SORT ALGORITHM EXPANDABLY
```

<sup>38</sup> 2015/11/18 I have not revisited this code for a long time, and perhaps I could improve it now with some new techniques.

<sup>39</sup> The code in earlier versions of this manual handled inputs composed of braced items. I have switched to comma separated inputs on the occasion of (link removed) The version here is like `code 3` on (link removed) (which is about 3x faster than the earlier code it replaced in this manual) with a modification to make it more efficient if the data has many repeated values. A faster routine (for sorting hundreds of values) is provided as `code 6` at the link mentioned in the footnote, it is based on Merge Sort, but limited to inputs which one can handle as TeX dimensions. This `code 6` could be extended to handle more general numbers, as acceptable by `xintfrac`. I have also written a non expandable version, which is even faster, but this matters really only when handling hundreds or rather thousands of values.

```

% \usepackage{xintfrac} in the preamble (latex)
\makeatletter
% use extra safe delimiters
\catcode`! 3 \catcode`? 3
\def\Qsx {\romannumeral0\qxs }%
% first we check if empty list (else \qxs@finish will not find a comma)
\def\qxs #1{\expandafter\qxs@a\romannumeral-`0#1,!}%
\def\qxs@a #1{\ifx,#1\expandafter\qxs@abort\else
\expandafter\qxs@start\fi #1}%
\def\qxs@abort #1?{ }%
\def\qxs@start {\expandafter\qxs@finish\romannumeral0\qxs@b,}%
\def\qxs@finish ,#1{ #1}%
%
% we check if empty of single and if not pick up the first as Pivot:
\def\qxs@b ,#1#2,#3{\ifx?#3\xintdothis\qxs@empty\fi
\ifx!#3\xintdothis\qxs@single\fi
\xintorthat\qxs@separate {#1#2}{#3}%
\def\qxs@empty #1#2#3#4#5{ }%
\def\qxs@single #1#2#3#4#5?{, #4}%
\def\qxs@separate #1#2#3#4#5#6,%
{%
\ifx!#5\expandafter\qxs@separate@done\fi
\xintifCmp {#5#6}{#4}%
\qxs@separate@appendtosmaller
\qxs@separate@appendtoequal
\qxs@separate@appendtogreater {#5#6}{#1}{#2}{#3}{#4}%
}%
%
\def\qxs@separate@appendtoequal #1#2{\qxs@separate {#2,#1}}%
\def\qxs@separate@appendtogreater #1#2#3{\qxs@separate {#2}{#3,#1}}%
\def\qxs@separate@appendtosmaller #1#2#3#4{\qxs@separate {#2}{#3}{#4,#1}}%
%
\def\qxs@separate@done\xintifCmp #1%
\qxs@separate@appendtosmaller
\qxs@separate@appendtoequal
\qxs@separate@appendtogreater #2#3#4#5#6#7?%
{%
\expandafter\qxs@f\expandafter {\romannumeral0\qxs@b #4,!}{\qxs@b #5,!}{#3}%
}%
%
\def\qxs@f #1#2#3{#2, #3#1}%
%
\catcode`! 12 \catcode`? 12
\makeatother

% EXAMPLE
\begingroup
\edef\z {\Qsx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
\meaning\z

\def\ a {3.123456789123456789}\def\ b {3.123456789123456788}

```

```

\def\c {3.123456789123456790}\def\d {3.123456789123456787}
\oodef\z {\QSx { \a, \b, \c, \d}}%
% The space before \a to let it not be expanded during the conversion from CSV
% values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
\meaning\z
\endgroup
macro:->0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7,
1.8, 1.9, 2.0
macro:->\d , \b , \a , \c (the spaces after \d, etc... come from the use of the \meaning primitive.)

```

The choice of pivot as first element is bad if the list is already almost sorted. Let's add a variant which will pick up the pivot index randomly. The previous routine worked also internally with comma separated lists, but for a change this one will use internally lists of braced items (the initial conversion via [\xintCSVtoList](#) handles all potential spurious space problems).

```

% QuickSort expandably on comma separated values with random choice of pivots
% ===== Requires availability of \pdfuniformdeviate =====
% \usepackage{xintfrac, xinttools} in preamble
\makeatletter
\def\QSx {\romannumeral0\qsx}% This is a f-expandable macro.
% This converts from comma separated values on input and back on output.
% **** NOTE: these steps (and the other ones too, actually) are costly if input
%           has thousands of items.
\def\qsx #1{\xintlistwithsep{,}%
           {\expandafter\qsx@sort@a\expandafter{\romannumeral0\xintcsvtolist{#1}}}%
%
% we check if empty or single or double and if not pick up the first as Pivot:
\def\qsx@sort@a #1%
           {\expandafter\qsx@sort@b\expandafter{\romannumeral0\xintlength{#1}}{#1}}%
\def\qsx@sort@b #1{\ifcase #1
                  \expandafter\qsx@sort@empty
                  \or\expandafter\qsx@sort@single
                  \or\expandafter\qsx@sort@double
                  \else\expandafter\qsx@sort@c\fi {#1}}%
\def\qsx@sort@empty #1#2{ }%
\def\qsx@sort@single #1#2{#2}%
\catcode`_ 11
\def\qsx@sort@double #1#2{\xintifGt #2{\xint_exchangetwo_keepbraces}}{#2}%
\catcode`_ 8
\def\qsx@sort@c      #1#2{%
  \expandafter\qsx@sort@sep@a\expandafter
    {\romannumeral0\xintntheft{\pdfuniformdeviate #1+\@ne}{#2}}{#2}}%
\def\qsx@sort@sep@a #1{\qsx@sort@sep@loop {}{}{#1}}%
\def\qsx@sort@sep@loop #1#2#3#4#5%
{%
  \ifx?#5\expandafter\qsx@sort@sep@done\fi
  \xintifCmp {#5}{#4}%
    \qsx@sort@sep@appendtosmaller
    \qsx@sort@sep@appendtoequal
    \qsx@sort@sep@appendtogreater {#5}{#1}{#2}{#3}{#4}%
}%
%
\def\qsx@sort@sep@appendtoequal #1#2{\qsx@sort@sep@loop {#2}{#1}}%

```



```

\def\qxs@sort@sep@appendtogreater #1#2#3{\qxs@sort@sep@loop {#2}{#3{#1}}}%
\def\qxs@sort@sep@appendtosmaller #1#2#3#4{\qxs@sort@sep@loop {#2}{#3}{#4{#1}}}%
%
\def\qxs@sort@sep@done\xintifCmp #1%
    \qxs@sort@sep@appendtosmaller
    \qxs@sort@sep@appendtoequal
    \qxs@sort@sep@appendtogreater #2#3#4#5#6%
{%
    \expandafter\qxs@sort@recurse\expandafter
        {\romannumeral0\qxs@sort@a {#4}}{\qxs@sort@a {#5}}{#3}%
}%
%
\def\qxs@sort@recurse #1#2#3{#2#3#1}%
%
\makeatother

% EXAMPLES
\begingroup
\edef\z {\QSx {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
    1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}}
\meaning\z

\def\a {3.123456789123456789}\def\b {3.123456789123456788}
\def\c {3.123456789123456790}\def\d {3.123456789123456787}
\oodef\z {\QSx { \a, \b, \c, \d}}%
% The space before \a to let it not be expanded during the conversion from CSV
% values to List. The \oodef expands exactly twice (via a bunch of \expandafter's)
\meaning\z

\def\somenumbers{%
3997.6421, 8809.9358, 1805.4976, 5673.6478, 3179.1328, 1425.4503, 4417.7691,
2166.9040, 9279.7159, 3797.6992, 8057.1926, 2971.9166, 9372.2699, 9128.4052,
1228.0931, 3859.5459, 8561.7670, 2949.6929, 3512.1873, 1698.3952, 5282.9359,
1055.2154, 8760.8428, 7543.6015, 4934.4302, 7526.2729, 6246.0052, 9512.4667,
7423.1124, 5601.8436, 4433.5361, 9970.4849, 1519.3302, 7944.4953, 4910.7662,
3679.1515, 8167.6824, 2644.4325, 8239.4799, 4595.1908, 1560.2458, 6098.9677,
3116.3850, 9130.5298, 3236.2895, 3177.6830, 5373.1193, 5118.4922, 2743.8513,
8008.5975, 4189.2614, 1883.2764, 9090.9641, 2625.5400, 2899.3257, 9157.1094,
8048.4216, 3875.6233, 5684.3375, 8399.4277, 4528.5308, 6926.7729, 6941.6278,
9745.4137, 1875.1205, 2755.0443, 9161.1524, 9491.1593, 8857.3519, 4290.0451,
2382.4218, 3678.2963, 5647.0379, 1528.7301, 2627.8957, 9007.9860, 1988.5417,
2405.1911, 5065.8063, 5856.2141, 8989.8105, 9349.7840, 9970.3013, 8105.4062,
3041.7779, 5058.0480, 8165.0721, 9637.7196, 1795.0894, 7275.3838, 5997.0429,
7562.6481, 8084.0163, 3481.6319, 8078.8512, 2983.7624, 3925.4026, 4931.5812,
1323.1517, 6253.0945}%

\oodef\z {\QSx \somenumbers}% produced as a comma+space separated list
% black magic as workaround to the shrinkability of spaces in last line...
\hsize 87\fontcharwd\font`0
\lccode`~ = 32
\lowercase{\def~}{\discretionary{}{}{\kern\fontcharwd\font`0}}\catcode32 13
\noindent\phantom{00}\scantokens\expandafter{\meaning\z}\par

```

```

\endgroup
macro:->0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7,
1.8, 1.9, 2.0
macro:->\d , \b , \a , \c
macro:->1055.2154, 1228.0931, 1323.1517, 1425.4503, 1519.3302, 1528.7301, 1560.2458, 1698.3952,
1795.0894, 1805.4976, 1875.1205, 1883.2764, 1988.5417, 2166.9040, 2382.4218, 2405.1911,
2625.5400, 2627.8957, 2644.4325, 2743.8513, 2755.0443, 2899.3257, 2949.6929, 2971.9166,
2983.7624, 3041.7779, 3116.3850, 3177.6830, 3179.1328, 3236.2895, 3481.6319, 3512.1873,
3678.2963, 3679.1515, 3797.6992, 3859.5459, 3875.6233, 3925.4026, 3997.6421, 4189.2614,
4290.0451, 4417.7691, 4433.5361, 4528.5308, 4595.1908, 4910.7662, 4931.5812, 4934.4302,
5058.0480, 5065.8063, 5118.4922, 5282.9359, 5373.1193, 5601.8436, 5647.0379, 5673.6478,
5684.3375, 5856.2141, 5997.0429, 6098.9677, 6246.0052, 6253.0945, 6926.7729, 6941.6278,
7275.3838, 7423.1124, 7526.2729, 7543.6015, 7562.6481, 7944.4953, 8008.5975, 8048.4216,
8057.1926, 8078.8512, 8084.0163, 8105.4062, 8165.0721, 8167.6824, 8239.4799, 8399.4277,
8561.7670, 8760.8428, 8809.9358, 8857.3519, 8989.8105, 9007.9860, 9090.9641, 9128.4052,
9130.5298, 9157.1094, 9161.1524, 9279.7159, 9349.7840, 9372.2699, 9491.1593, 9512.4667,
9637.7196, 9745.4137, 9970.3013, 9970.4849

```

All the previous examples were with numbers which could have been handled via `\ifdim` tests rather than the `\xintifCmp` macro from `xintfrac`; using `\ifdim` tests would naturally be faster. Even faster routine is [code 6](#) at [\(link removed\)](#) which uses `\pdfescapestring` and a Merge Sort algorithm.

We then turn to a graphical illustration of the algorithm.<sup>40</sup> For simplicity the pivot is always chosen as the first list item. Then we also give a variant which picks up the last item as pivot.

```

% in LaTeX preamble:
% \usepackage{xintfrac, xinttools}
% \usepackage{color}
% or, when using Plain TeX:
% \input xintfrac.sty \input xinttools.sty
% \input color.tex
%
% Color definitions
\definecolor{LEFT}{RGB}{216,195,88}
\definecolor{RIGHT}{RGB}{208,231,153}
\definecolor{INERT}{RGB}{199,200,194}
\definecolor{INERTpiv}{RGB}{237,237,237}
\definecolor{PIVOT}{RGB}{109,8,57}
% Start of macro defintions
\makeatletter
% \catcode`? 3 % a bit too paranoid. Normal ? will do.
%
% argument will never be empty
\def\QS@cmp@a #1{\QS@cmp@b #1??}%
\def\QS@cmp@b #1{\noexpand\QS@sep@A\@ne{#1}\QS@cmp@d {#1}}%
\def\QS@cmp@d #1#2{\ifx ?#2\expandafter\QS@cmp@done\fi
\expandafter\QS@sep@A\@ne{#2}\tw@{#1}\QS@cmp@d {#1}}%
\def\QS@cmp@done #1{?}%
%
\def\QS@sep@A #1{\QSLr\QS@sep@L #1\thr@@?#1\thr@@?#1\thr@@?}%
\def\QS@sep@L #1#2{\ifcase #1{#2}\or\or\else
\expandafter\QS@sep@I@start\fi \QS@sep@L}%

```

<sup>40</sup> I have rewritten (2015/11/21) the routine to do only once (and not thrice) the needed calls to `\xintifCmp`, up to the price of one additional `\edef`, although due to the context execution time on our side is not an issue and moreover is anyhow overwhelmed by the TikZ's activities. Simultaneously I have updated the code. The variant with the choice of pivot on the right has more overhead: the reason is simply that we do not convert the data into an array, but maintain a list of tokens with self-reorganizing delimiters.

```

\def\QS@sep@I@start\QS@sep@L {\noexpand\empty?\QSIr\QS@sep@I}%
\def\QS@sep@I #1#2{\ifcase#1\or{#2}\or\else\expandafter\QS@sep@R@start\fi\QS@sep@I}%
\def\QS@sep@R@start\QS@sep@I {\noexpand\empty?\QSRr\QS@sep@R}%
\def\QS@sep@R #1#2{\ifcase#1\or\or{#2}\else\expandafter\QS@sep@done\fi\QS@sep@R}%
\def\QS@sep@done\QS@sep@R {\noexpand\empty?}%
%
\def\QS@loop {%
  \xintloop
  % pivot phase
  \def\QS@pivotcount{0}%
  \let\QSLr\DecoLEFTwithPivot \let\QSIr \DecoINERT
  \let\QSRr\DecoRIGHTwithPivot \let\QSIrr\DecoINERT
  \centerline{\QS@list}%
  % sorting phase
  \ifnum\QS@pivotcount>\z@
    \def\QSLr {\QS@cmp@a}\def\QSRr {\QS@cmp@a}%
    \def\QSIr {\QSIrr}\let\QSIrr\relax
    \edef\QS@list{\QS@list}% compare
    \let\QSLr\relax\let\QSRr\relax\let\QSIr\relax
    \edef\QS@list{\QS@list}% separate
    \def\QSLr ##1##2?{\ifx\empty##1\else\noexpand \QSLr {{##1}##2}\fi}%
    \def\QSIr ##1##2?{\ifx\empty##1\else\noexpand \QSIr {{##1}##2}\fi}%
    \def\QSRr ##1##2?{\ifx\empty##1\else\noexpand \QSRr {{##1}##2}\fi}%
    \edef\QS@list{\QS@list}% gather
    \let\QSLr\DecoLEFT \let\QSRr\DecoRIGHT
    \let\QSIr\DecoINERTwithPivot \let\QSIrr\DecoINERT
    \centerline{\QS@list}%
  \repeat }%
%
% \xintFor* loops handle gracefully empty lists.
\def\DecoLEFT #1{\xintFor* ##1 in {#1} \do {\colorbox{LEFT}{##1}}}%
\def\DecoINERT #1{\xintFor* ##1 in {#1} \do {\colorbox{INERT}{##1}}}%
\def\DecoRIGHT #1{\xintFor* ##1 in {#1} \do {\colorbox{RIGHT}{##1}}}%
\def\DecoPivot #1{\begingroup
  \color{PIVOT}\advance\fbboxsep-\fbboxrule\fbbox{#1}\endgroup}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForFirst {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}%
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForFirst {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}%
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForFirst {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}%
%
\def\QuickSort #1{% warning: not compatible with empty #1.
  % initialize, doing conversion from comma separated values
  % to a list of braced items
  \edef\QS@list{\noexpand\QSRr{\xintCSVtoList{#1}}}%
  % may \edef's are to follow anyhow
% earlier I did a first drawing of the list, here with the color of RIGHT elements,

```

```
% but the color should have been for example white, anyway I drop this first line
%\let\QSRr\DecoRIGHT
%\par\centerline{\QS@list}%
%
% loop as many times as needed
\QS@loop}%
%
% \catcode? 12 % in case we had used a funny ? as delimiter.
\makeatother
%% End of macro definitions.
%% Start of Example
\begin{group}\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%             1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
            1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\end{group}
```

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1.0 | 0.5 | 0.3 | 0.8 | 1.5 | 1.8 | 2.0 | 1.7 | 0.4 | 1.2 | 1.4 | 1.3 | 1.1 | 0.7 | 0.3 | 1.6 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.2 |
| 0.5 | 0.3 | 0.8 | 0.4 | 0.7 | 0.3 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.0 | 1.5 | 1.8 | 2.0 | 1.7 | 1.2 | 1.4 | 1.3 | 1.1 | 1.6 | 1.2 |
| 0.5 | 0.3 | 0.8 | 0.4 | 0.7 | 0.3 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.0 | 1.5 | 1.8 | 2.0 | 1.7 | 1.2 | 1.4 | 1.3 | 1.1 | 1.6 | 1.2 |
| 0.3 | 0.4 | 0.3 | 0.3 | 0.2 | 0.5 | 0.8 | 0.7 | 0.6 | 0.8 | 0.8 | 0.7 | 1.0 | 1.2 | 1.4 | 1.3 | 1.1 | 1.2 | 1.5 | 1.8 | 2.0 | 1.7 | 1.6 |
| 0.3 | 0.4 | 0.3 | 0.3 | 0.2 | 0.5 | 0.8 | 0.7 | 0.6 | 0.8 | 0.8 | 0.7 | 1.0 | 1.2 | 1.4 | 1.3 | 1.1 | 1.2 | 1.5 | 1.8 | 2.0 | 1.7 | 1.6 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.7 | 0.6 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.4 | 1.3 | 1.5 | 1.7 | 1.6 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.7 | 0.6 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.4 | 1.3 | 1.5 | 1.7 | 1.6 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |

Here is the variant which always picks the pivot as the rightmost element.

```
\makeatletter
%
\def\QS@cmp@a #1{\noexpand\QS@sep@a\expandafter\QS@cmp@d\expandafter
                {\romannumeral0\xintnthelt{-1}{#1}}#1??}%
%
\def\DecoLEFTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForLast {\DecoPivot {##1}}{\colorbox{LEFT}{##1}}}}
\def\DecoINERTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForLast {\colorbox{INERTpiv}{##1}}{\colorbox{INERT}{##1}}}}
\def\DecoRIGHTwithPivot #1{\xdef\QS@pivotcount{\the\numexpr\QS@pivotcount+\@ne}%
  \xintFor* ##1 in {#1} \do
    {\xintifForLast {\DecoPivot {##1}}{\colorbox{RIGHT}{##1}}}}
\def\QuickSort #1{%
  % initialize, doing conversion from comma separated values
  % to a list of braced items
  \edef\QS@list{\noexpand\QSLr {\xintCSVtoList{#1}}}%
  % many \edef's are to follow anyhow
  %
  % loop as many times as needed
```

```

\QS@loop }%
\makeatother
\begin{group}\offinterlineskip
\small
% \QuickSort {1.0, 0.5, 0.3, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
%             1.3, 1.1, 0.7, 1.6, 0.6, 0.9, 0.8, 0.2, 0.1, 1.9}
% \medskip
% with repeated values
\QuickSort {1.0, 0.5, 0.3, 0.8, 1.5, 1.8, 2.0, 1.7, 0.4, 1.2, 1.4,
            1.3, 1.1, 0.7, 0.3, 1.6, 0.6, 0.3, 0.8, 0.2, 0.8, 0.7, 1.2}
\end{group}

```

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1.0 | 0.5 | 0.3 | 0.8 | 1.5 | 1.8 | 2.0 | 1.7 | 0.4 | 1.2 | 1.4 | 1.3 | 1.1 | 0.7 | 0.3 | 1.6 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.2 |
| 1.0 | 0.5 | 0.3 | 0.8 | 0.4 | 1.1 | 0.7 | 0.3 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.2 | 1.2 | 1.5 | 1.8 | 2.0 | 1.7 | 1.4 | 1.3 | 1.6 |
| 1.0 | 0.5 | 0.3 | 0.8 | 0.4 | 1.1 | 0.7 | 0.3 | 0.6 | 0.3 | 0.8 | 0.2 | 0.8 | 0.7 | 1.2 | 1.2 | 1.5 | 1.8 | 2.0 | 1.7 | 1.4 | 1.3 | 1.6 |
| 0.5 | 0.3 | 0.4 | 0.3 | 0.6 | 0.3 | 0.2 | 0.7 | 0.7 | 1.0 | 0.8 | 1.1 | 0.8 | 0.8 | 1.2 | 1.2 | 1.5 | 1.4 | 1.3 | 1.6 | 1.8 | 2.0 | 1.7 |
| 0.5 | 0.3 | 0.4 | 0.3 | 0.6 | 0.3 | 0.2 | 0.7 | 0.7 | 1.0 | 0.8 | 1.1 | 0.8 | 0.8 | 1.2 | 1.2 | 1.5 | 1.4 | 1.3 | 1.6 | 1.8 | 2.0 | 1.7 |
| 0.2 | 0.5 | 0.3 | 0.4 | 0.3 | 0.6 | 0.3 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.5 | 1.4 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.5 | 0.3 | 0.4 | 0.3 | 0.6 | 0.3 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.5 | 1.4 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.5 | 0.4 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.5 | 0.4 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.5 | 0.4 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.5 | 0.4 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |
| 0.2 | 0.3 | 0.3 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 | 0.8 | 1.0 | 1.1 | 1.2 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 2.0 |

The choice of the first or last item as pivot is not a good one as nearly ordered lists will take quadratic time. But for explaining the algorithm via a graphical interpretation, it is not that bad. If one wanted to pick up the pivot randomly, the routine would have to be substantially rewritten: in particular the `\Deco.withPivot` macros need to know where the pivot is, and currently this is implemented by using either `\xintifForFirst` or `\xintifForLast`.

## Part II.

# The macro layer for expandable computations: [xintcore](#), [xint](#), [xintfrac](#), and some extras

### WARNING !

The documentation is getting old, and is in need of rewrites for many sections, particularly for examples.

We do try to keep updated the description of macros provided by the packages.

|           |                                                               |     |
|-----------|---------------------------------------------------------------|-----|
| <b>8</b>  | <b>The <a href="#">xint bundle</a></b> .....                  | 127 |
| <b>9</b>  | <b>Macros of the <a href="#">xintkernel</a> package</b> ..... | 142 |
| <b>10</b> | <b>Macros of the <a href="#">xintcore</a> package</b> .....   | 146 |
| <b>11</b> | <b>Macros of the <a href="#">xint</a> package</b> .....       | 151 |
| <b>12</b> | <b>Macros of the <a href="#">xintfrac</a> package</b> .....   | 163 |
| <b>13</b> | <b>Macros of the <a href="#">xintbinhex</a> package</b> ..... | 193 |
| <b>14</b> | <b>Macros of the <a href="#">xintgcd</a> package</b> .....    | 198 |
| <b>15</b> | <b>Macros of the <a href="#">xintseries</a> package</b> ..... | 200 |
| <b>16</b> | <b>Macros of the <a href="#">xintcfrac</a> package</b> .....  | 216 |

## 8. The [xint bundle](#)

|           |                                                |     |            |                                                  |     |
|-----------|------------------------------------------------|-----|------------|--------------------------------------------------|-----|
| <b>.1</b> | <b>Characteristics</b> .....                   | 127 | <b>.8</b>  | <b>\ifcase, \ifnum, ... constructs</b> .....     | 136 |
| <b>.2</b> | <b>Floating point evaluations</b> .....        | 129 | <b>.9</b>  | <b>No variable declarations are needed</b> ..... | 137 |
| <b>.3</b> | <b>Expansion matters</b> .....                 | 130 | <b>.10</b> | <b>Possible syntax errors to avoid</b> .....     | 137 |
| <b>.4</b> | <b>Input formats for macros</b> .....          | 132 | <b>.11</b> | <b>Error messages</b> .....                      | 138 |
| <b>.5</b> | <b>Output formats of macros</b> .....          | 134 | <b>.12</b> | <b>Package namespace, catcodes</b> .....         | 139 |
| <b>.6</b> | <b>Count registers and variables</b> .....     | 134 | <b>.13</b> | <b>Origins of the package</b> .....              | 140 |
| <b>.7</b> | <b>Dimension registers and variables</b> ..... | 135 |            |                                                  |     |

### 8.1. Characteristics

The main characteristics are:

1. exact algebra on ``big numbers'', integers as well as fractions,

2. floating point variants with user-chosen precision,
3. the computational macros are compatible with expansion-only context,
4. the bundle comes with parsers (integer-only, or handling fractions, or doing floating point computations) of infix operations implementing beyond infix operations extra features such as dummy variables.

Since 1.2 ``big numbers'' must have less than about 19950 digits: the maximal number of digits for addition is at 19968 digits, and it is 19959 for multiplication. The reasonable range of use of the package is with numbers of up to a few hundred digits.<sup>41</sup>

$\TeX$  does not know off-hand how to print on the page such very long numbers, see subsection 1.6.

Integers with only 10 digits and starting with a 3 already exceed the  $\TeX$  bound; and  $\TeX$  does not have a native processing of floating point numbers (multiplication by a decimal number of a dimension register is allowed --- this is used for example by the [pgf](#) basic math engine.)

$\TeX$  elementary operations on numbers are done via the non-expandable `\advance`, `\multiply`, and `\divide` assignments. This was changed with  $\varepsilon\text{-}\TeX$ 's `\numexpr` which does expandable computations using standard infix notations with  $\TeX$  integers. But  $\varepsilon\text{-}\TeX$  did not modify the  $\TeX$  bound on acceptable integers, and did not add floating point support.

The [bigintcalc](#) package by HEIKO OBERDIEK provided expandable macros (using some of `\numexpr` possibilities, when available) on arbitrarily big integers, beyond the  $\TeX$  bound. It does not provide an expression parser.<sup>42</sup> [xint](#) did it again using more of `\numexpr` for higher speed, and in a later evolution added handling of exact fractions, of scientific numbers, and an expression parser. Arbitrary precision floating points operations were added as a derivative, and not part of the initial design goal.

The concept of signed infinities, signed zeroes, NaN's, error traps...<sup>43</sup> have not been implemented, only the notion of 'scientific notation with a given number of significant figures'.<sup>44</sup>

The  $\mathbb{M}\TeX$ 3 project has implemented expandably floating-point computations with 16 significant figures ([l3fp](#)), including functions such as exp, log, sine and cosine.<sup>45</sup>

More directly related to the [xint bundle](#) there is the [l3bigint](#) package, also devoted to big integers and in development a.t.t.o.w (2015/10/09, no division yet). It is part of the experimental trunk of the [M \$\TeX\$ 3 Project](#) and provides an expression parser for expandable arithmetic with big integers. Its author Bruno LE FLOCH succeeded brilliantly into implementing expandably the Karatsuba multiplication algorithm and he achieves *sub-quadratic growth for the computation time*. This shows up very clearly with numbers having thousands of digits, up to the maximum which a.t.t.o.w is at 8192 digits.

The [l3bigint](#) multiplication from late 2015 is observed to be roughly 3x--4x faster than the one from `\xintiexpr` in the range of 4000 to 5000 digits integers, and isn't far from being 9x faster at 8000 digits. On the other hand `\xintiexpr`'s multiplication is found to be on average roughly

<sup>41</sup> For example multiplication of integers having from 50 to 100 digits takes roughly of the order of the millisecond on a 2012 desktop computer. I compared this to using Python3: using `timeit` module on a wrapper defined as `return w*z` with random integers of 100 digits, I observe on the same computer a computation time of roughly  $4 \cdot 10^{-7}$ s per call. And with `return str(w*z)` then this becomes more like  $16 \cdot 10^{-7}$ s per call. And with `return str(int(W)*int(Z))` where `W` and `Z` are strings, this becomes about  $26 \cdot 10^{-7}$ s (I am deliberately ignoring Python's `Decimal` module here...). Anyway, my sentence from earlier version of this documentation: *this is, I guess, at least about 1000 times slower than what can be expected with any reasonable programming language*, is about right. I then added: *nevertheless as compilation of a typical  $\mathbb{M}\TeX$  document already takes of the order of seconds and even dozens of seconds for long ones, this leaves room for reasonably many computations via `xintexpr` or via direct use of the macros of `xint/xintfrac`*.<sup>42</sup> One can currently use package [bnumexpr](#) to associate the [bigintcalc](#) macros with an expression parser. This may be unavailable in future if [bnumexpr](#) becomes more tightly associated with future evolutions or variants of [xintcore](#). EDIT: still possible as of [bnumexpr](#) 1.6 2025/09/01.<sup>43</sup> The latter exist as work-in-progress for some time in the source code.<sup>44</sup> Multiplication of two floats with `P=\xinttheDigits` digits is first done exactly then rounded to `P` digits, rather than using a specially tailored multiplication for floating point numbers which would be more efficient (it is a waste to evaluate fully the multiplication result with `2P` or `2P-1` digits.)<sup>45</sup> at the time of writing (2014/10/28) the [l3fp](#) (exactly represented) floating point numbers have their exponents limited to  $\pm 9999$ .



2.5x faster than [l3bignt](#)'s for numbers up to 100 digits and the two packages achieve about the same speed at 900 digits: but each such multiplication of numbers of 900 digits costs about one or two tenths of a second on a 2012 desktop computer, whereas the order of magnitude is rather the ms for numbers with 50--100 digits.<sup>46</sup>

Even with the superior [l3bignt](#) Karatsuba multiplication it takes about 3.5s on this 2012 desktop computer for a single multiplication of two 5000-digits numbers. Hence it is not possible to do routinely such computations in a document. I have long been thinking that without the expandability constraint much higher speeds could be achieved, but perhaps I have not given enough thought to sustain that optimistic stance.<sup>47</sup>

I remain of the opinion that if one really wants to do computations with *thousands* of digits, one should drop the expandability requirement. Indeed, as clearly demonstrated long ago by the [pi computing file](#) by D. ROEGEL one can program  $\TeX$  to compute with many digits at a much higher speed than what [xint](#) achieves: but, direct access to memory storage in one form or another seems a necessity for this kind of speed and one has to renounce at the complete expandability.<sup>48</sup>

## 8.2. Floating point evaluations

Floating point macros are provided by package [xintfrac](#) to work with a given arbitrary precision  $P$ . The default value is  $P = 16$  meaning that the significands of the produced (non-zero) numbers have 16 decimal digits. The syntax to set the precision to  $P$  is

```
\xintDigits:=P\relax
```

The value is local to the group or environment (if using  $\LaTeX$ ). To query the current value use [\xinttheDigits](#).

Most floating point macros accept an optional first argument  $[P]$  which then sets the target precision and replaces the [\xintDigits](#) assigned value (the  $[P]$  must be repeated if the arguments are themselves [xintfrac](#) macros with arguments of their own.) In this section  $P$  refers to the prevailing [\xinttheDigits](#) float precision or to the target precision set in this way as an optional argument.

[\xintfloatexpr](#) $[Q] \dots \relax$  also admits an optional argument  $[Q]$  but it has an altogether different meaning: the computations are always done with the prevailing [\xinttheDigits](#) precision and the optional argument  $Q$  is used for the final rounding. This makes sense only if  $Q < \text{\xinttheDigits}$  and is intended to clean up the result from dubious last digits (when  $Q < 0$  it indicates rather by how many digits one should reduce the mantissa lengths via a final rounding).

The IEEE 754<sup>49</sup> requirement of *correct rounding* for addition, subtraction, multiplication, division and square root is achieved (in arbitrary precision) by the macros of [xintfrac](#) hence also by the infix operators  $+$ ,  $-$ ,  $*$ ,  $/$ .

This means that for operands given with at most  $P$  significant digits (and arbitrary exponents) the output coincides exactly with the rounding of the exact theoretical result (barring overflow or underflow).

Due to a typographical oversight, this documentation (up to 1.2j) adjoined  $^$  and  $^{**}$  to the above list of infix operators. But as is explained in [subsection 12.97](#), what is guaranteed regarding integer powers is an error of at most 0.52ulp, not the correct rounding. Half-integer powers are computed as square roots of integer powers.

The rounding mode is ``round to nearest, ties away from zero''. It is not customizable.

Currently [xintfrac](#) has no notion of NaNs or signed infinities or signed zeroes, but this is intended for the future.

<sup>46</sup> I have tested this again on 2016/12/19, but the macros have not changed on the [l3bignt](#) side and barely on the [xintcore](#) side, hence I got again the same results. . .

<sup>47</sup> The [apnum](#) package implements (non-expandably) arbitrary precision fixed point algebra and (v1.6) functions exp, log, sqrt, the trigonometrical direct and inverse functions. <sup>48</sup> The  $\LaTeX$  project possibly makes endeavours such as [xint](#) appear even more insane that they are, in truth: [xint](#) is able to handle fast enough computations involving numbers with less than one hundred digits and brings this to all engines.

Since release 1.2f, square root extraction achieves correct rounding in arbitrary precision.

See [xintlog](#) for fractional powers and [xinttrig](#) for trigonometrical functions.

The maximal floating point decimal exponent is currently 2147483647 which is the maximal number handled by  $\TeX$ . The minimal exponent is its opposite. But this means that overflow or underflow are detected only via low-level `\numexpr` arithmetic overflows which are basically un-recoverable. Besides there are some border effects as the routines need to add or subtract lengths of numbers from exponents, possibly triggering the low-level overflows. In the future not only the Precision but also the maximal and minimal exponents `Emin` and `Emax` will be specifiable by the user.

Since 1.2f, the float macros round their inputs to the target precision `P` before further processing. Formerly, the initial rounding was done to `P+2` digits (and at least `P+3` for the power operation.)

The more ambitious model would be for the computing macros to obey the intrinsic precision of their inputs, i.e. to compute the correct rounding to `P` digits of the exact mathematical result corresponding to inputs allowed to have their own higher precision.<sup>50</sup> This would be feasible by [xintfrac](#) which after all knows how to compute exactly, but I have for the time being decided that for reasons of efficiency, the chosen model is the one of rounding inputs to the target precision first.

The float macros of [xintfrac](#) have to handle inputs which not only may have much more digits than the target float precision, but may even be fractions: in a way this means infinite precision.

From releases 1.08a to 1.2j a fraction input `AeM/BeN` had its numerator and denominator `A` and `B` truncated to `Q+2` digits of precision, then the substituted fraction was correctly rounded to `Q` digits of precision (usually with `Q` set to `P+2`) and then the operation was implemented on such rounded inputs. But this meant that two fractions representing the same rational number could end up being rounded differently (with a difference of one unit in the last place), if it had numerators and denominators with at least `Q+3` digits.

Starting with release 1.2k a fractional input `AeM/BeN` is handled intrinsically: the fraction, independently of its representation `AeM/BeN`, is *correctly rounded* to `P` digits during the input parsing. Hence the output depends only on its arguments as mathematical fractions and not on their representatives as quotients.

Notice that in float expressions, the `/` is treated as operator, and is applied to arguments which are generally already `P`-floats, hence the above discussion becomes relevant in this context only for the special input form `qfloat(A/B)` or when using a sub-expression `\xintexpr A/B\relax` embedded in the float expression with `A` or `B` having more digits than the prevailing float precision `P`.

## 8.3. Expansion matters

### 8.3.1. Full expansion of the first token

The whole business of [xint](#) is to build upon `\numexpr` and handle arbitrarily large numbers. Each basic operation is thus done via a macro: `\xintiiAdd`, `\xintiiSub`, `\xintiiMul`, `\xintiiDivision`. In order to handle more complex operations, it must be possible to nest these macros. An expandable macro can not execute a `\def` or an `\edef`. But the macro must expand its arguments to find the digits it is supposed to manipulate.  $\TeX$  provides a tool to do the job of (expandable !) repeated expansion of the first token found until hitting something non expandable, such as a digit, a `\def` token, a brace, a `\count` token, etc... is found. A space token also will stop the expansion (and be swallowed, contrarily to the non-expandable tokens).

By convention in this manual *f-expansion* (``full expansion'` or ``full first expansion'`) will be this  $\TeX$  process of expanding repeatedly the first token seen. For those familiar with  $\TeX$ 3

<sup>49</sup> The IEEE 754-1985 standard was for hardware implementations of binary floating-point arithmetic with a specific value for the precision (24 bits for single precision, 53 bits for double precision). The newer IEEE 754-2008 ([https://en.wikipedia.org/wiki/IEEE\\_floating\\_point](https://en.wikipedia.org/wiki/IEEE_floating_point)) normalizes five basic formats, three binaries and two decimals (16 and 34 decimal digits) and discusses extended formats with higher precision. These standards are only indirectly relevant to libraries like [xint](#) dealing with arbitrary precision. <sup>50</sup> The MPFR library <http://www.mpr.org/> implements this but it does not know fractions!

(which is not used by `xint`) this is what is called in its documentation full expansion, whereas expansion inside `\edef` would be described I think as ``exhaustive'' expansion and will be referred too in this manual as `x-expansion`.

Most of the package macros, and all those dealing with computations<sup>51</sup>, are expandable in the strong sense that they expand to their final result via this `f-expansion`. This will be signaled in their descriptions via a star in the margin.

★ These macros not only have this property of `f-expandability`, they all begin by first applying `f-expansion` to their arguments. Again from  $\TeX$ 's conventions this will be signaled by a margin annotation next to the description of the arguments.

### 8.3.2. Summary of important expandability aspects

1. the macros `f-expand` their arguments, this means that they expand the first token seen (for each argument), then expand, etc..., until something un-expandable such as a digit or a brace is hit against. This example

```
\def\x{98765}\def\y{43210} \xintiiAdd {\x}{\x\y}
```

is not a legal construct, as the `\y` will remain untouched by expansion and not get converted into the digits which are expected by the sub-routines of `\xintiiAdd`. It is a `\numexpr` which will expand it and an arithmetic overflow will arise as `9876543210` exceeds the  $\TeX$  bounds. The same would hold for `\xintAdd`.

To the contrary `\xinttheiexpr` and others have no issues with things such as `\xinttheiexpr \x+\x\y\relax`.

2. using `\if...\fi` constructs inside the package macro arguments requires suitably mastering  $\TeX$ niques (`\expandafter`'s and/or swapping techniques) to ensure that the `f-expansion` will indeed absorb the `\else` or closing `\fi`, else some error will arise in further processing. Therefore it is highly recommended to use the package provided conditionals such as `\xint-ifEq`, `\xintifGt`, `\xintifSgn`,... or, for  $\TeX$  users and when dealing with short integers the `etoolbox`<sup>52</sup> expandable conditionals (for small integers only) such as `\ifnumequal`, `\ifnumgreater`, .... Use of non-expandable things such as `\ifthenelse` is impossible inside the arguments of `xint` macros.

One can use naive `\if...\fi` things inside an `\xinttheexpr`-ession and cousins, as long as the test is expandable, for example

```
\xinttheiexpr\ifnum3>2 143\else 33\fi 0^2\relax→2044900=1430^2
```

3. after the definition `\def\x {12}`, one can not use `-x` as input to one of the package macros: the `f-expansion` will act only on the minus sign, hence do nothing. The only way is to use the `\xintOpp` macro (or `\xintiiOpp` which is integer only) which obtains the opposite of a given number.

Again, this is otherwise inside an `\xinttheexpr`-ession or `\xintthefloatexpr`-ession. There, the minus sign may prefix macros which will expand to numbers (or parentheses etc...)

4. With the definition

```
\def\AplusBC #1#2#3{\xintAdd {#1}{\xintMul {#2}{#3}}}
```

one obtains an expandable macro producing the expected result, not in two, but rather in three steps: a first expansion is consumed by the macro expanding to its definition. As the package macros expand their arguments until no more is possible (regarding what comes first), this `\AplusBC` may be used inside them: `\xintAdd {\AplusBC {1}{2}{3}}{4}` does work and returns `11/1[0]`.

If, for some reason, it is important to create a macro expanding in two steps to its final value, one may either do:

<sup>51</sup> except `\xintXTrunc`. <sup>52</sup> <https://ctan.org/pkg/etoolbox>

```
\def\AplusBC #1#2#3{\romannumeral-\`0\xintAdd {#1}{\xintMul {#2}{#3}}}
```

or use the lowercase form of `\xintAdd`:

```
\def\AplusBC #1#2#3{\romannumeral0\xintadd {#1}{\xintMul {#2}{#3}}}
```

and then `\AplusBC` will share the same properties as do the other `\xint` 'primitive' macros.

5. The `\romannumeral0` and `\romannumeral-\`0` things above look like an invitation to hacker's territory; if it is not important that the macro expands in two steps only, there is no reason to follow these guidelines. Just chain arbitrarily the package macros, and the new ones will be completely expandable and usable one within the other.

Since release 1.07 the `\xintNewExpr` macro automatizes the creation of such expandable macros:

```
\xintNewExpr\AplusBC[3]{#1+#2*#3}
```

creates the `\AplusBC` macro doing the above and expanding in two expansion steps.

6. In the expression parsers of `\xintexpr` such as `\xintexpr..\relax`, `\xintfloatexpr..\relax` the contents are expanded completely from left to right until the ending `\relax` is found and swallowed, and spaces and even (to some extent) catcodes do not matter.
7. For all variants, prefixing with `\xintthe` allows to print the result or use it in other contexts. Shortcuts `\xinttheexpr`, `\xintthefloatexpr`, `\xinttheiexpr`, ... are available.

## 8.4. Input formats for macros

Macros can have different types of arguments (we do not consider here the `\xintexpr`-parsers but only the macros of `\xintcore`/`\xint`/`\xintfrac`). In a macro description, a margin annotation signals what is the argument type.

num  
x

1. `\TeX` integers are handled inside a `\numexpr..\relax` hence may be count registers or variables. Beware that `-(1+1)` is not legal and raises an error, but `0-(1+1)` is. Also `2\cnta` with `\cnta` a `\count` isn't legal. Integers must be kept less than 2147483647 in absolute value, although the scaling operation `(a*b)/c` computes the intermediate product with twice as many bits.

The slash `/` does a `\rounded` division which is a fact of life of `\numexpr` which I have found very annoying in at least nine cases out of ten, not to say ninety-nine cases out of one hundred. Besides, it is at odds with `\TeX`'s `\divide` which does a truncated division (non-expandably).

But to follow-suit `/` also does rounded integer division in `\xintiexpr..\relax`, and the operator `//` does there the truncated division.

f

2. the strict format applies to macros handling big integers but only `f`-expanding their arguments. After this `f`-expansion the input should be a string of digits, optionally preceded by a unique minus sign. The first digit can be zero only if it is the only digit. A plus sign is not accepted. `-0` is not legal in the strict format. Macros of `\xint` with a double `ii` require this 'strict' format for the inputs.

Num  
f

3. the extended integer format applies when the macro parses its arguments via `\xintNum`. The input may then have arbitrarily many leading minus and plus signs, followed by leading zeroes, and further digits. With `\xintfrac` loaded, `\xintNum` is extended to accept fractions and its action is to truncate them to integers.

Frac  
f

4. the fraction input format applies to the arguments of `\xintfrac` macros handling genuine fractions. It allows two types of inputs: general and restricted. The restricted type is parsed faster, but... is restricted.

**general:** inputs of the shape `A.BeC/D.EeF`. Example:

```
\noindent\xintRaw{+--0367.8920280e17/-++278.289287e-15}\newline
\xintRaw{+--+1253.2782e+--+3/---0087.123e---5}\par
```

-3678920280/278289287[31]

-12532782/87123[7]

The input parser does not reduce fractions to smallest terms. Here are the rules of this general fraction format:

- everything is optional, absent numbers are treated as zero, here are some extreme cases:

```
\xintRaw{}, \xintRaw{.}, \xintRaw{./1.e}, \xintRaw{-.e}, \xintRaw{e/-1}
0/1[0], 0/1[0], 0/1[0], 0/1[0], 0/1[0]
```

- **AB** and **DE** may start with pluses and minuses, then leading zeroes, then digits.
- **C** and **F** will be given to `\numexpr` and can be anything recognized as such and not provoking arithmetic overflow (the lengths of **B** and **E** will also intervene to build the final exponent naturally which must obey the  $\TeX$  bound).
- the `/`, `.` (numerator and/or denominator) and `e` (numerator and/or denominator) are all optional components.
- each of **A**, **B**, **C**, **D**, **E** and **F** may arise from *f-expansion* of a macro.
- the whole thing may arise from *f-expansion*, however the `/`, `.`, and `e` should all come from this initial expansion. The `e` of scientific notation is mandatorily lowercased.

**restricted:** inputs either of the shape **A[N]** or **A/B[N]**, which represents the fraction **A/B** times  $10^N$ . The whole thing or each of **A**, **B**, **N** (but then not `/` or `[]`) may arise from *f-expansion*, **A** (after expansion) *must* have a unique optional minus sign and no leading zeroes, **B** (after expansion) *must* be a positive integer with no signs and no leading zeroes, **[N]** if present will be given to `\numexpr`. Any deviation from the rules above will result in errors.

Frac  
f

Notice that `*`, `+` and `-` contrarily to the `/` (which is treated simply as a kind of delimiter) are not acceptable within arguments of this type (see [subsection 8.6](#) for some exceptions to this.)

Generally speaking, there should be no spaces among the digits in the inputs (in arguments to the package macros). Although most would be harmless in most macros, there are some cases where spaces could break havoc.<sup>53</sup> So the best is to avoid them entirely.

This is entirely otherwise inside an `\xintexpr`-ession, where spaces are ignored (except when they occur inside arguments to some macros, thus escaping the `\xintexpr` parser). See the [section 2](#).

There are also some slightly more obscure expansion types: in particular, the `\xintApplyInline` and `\xintFor*` macros from `\xinttools` apply a special iterated *f-expansion*, which gobbles spaces, to the non-braced items (braced items are submitted to no expansion because the opening brace stops it) coming from their list argument; this is denoted by a special symbol in the margin. Some other macros such as `\xintSum` from `\xintfrac` first do an *f-expansion*, then treat each found (braced or not) item (skipping spaces between such items) via the general fraction input parsing, this is signaled as here in the margin where the signification of the `*` is thus a bit different from the previous case.

Frac  
f → \* f

<sup>53</sup> The `\xintNum` macro does not remove spaces between digits beyond the first non zero ones; however this should not really alter the subsequent functioning of the arithmetic macros, and besides, since `\xintcore` 1.2 there is an initial parsing of the entire number, during which spaces will be gobbled. However I have not done a complete review of the legacy code to be certain of all possibilities after 1.2 release. One thing to be aware of is that `\numexpr` stops on spaces between digits (although it provokes an expansion to see if an infix operator follows); the exponent for `\xintiPow` or the argument of the factorial `\xintiFac` are only subjected to such a `\numexpr` (there are a few other macros with such input types in `\xint`). If the input is given as, say `1 2\x` where `\x` is a macro, the macro `\x` will not be expanded by the `\numexpr`, and this will surely cause problems afterwards. Perhaps a later `\xint` will force `\numexpr` to expand beyond spaces, but I decided that was not really worth the effort. Another immediate cause of problems is an input of the type `\xintiiAdd {<space>\x }{\y }`, because the space will stop the initial expansion; this will most certainly cause an arithmetic overflow later when the `\x` will be expanded in a `\numexpr`. Thus in conclusion, damages due to spaces are unlikely if only explicit digits are involved in the inputs, or arguments are single macros with no preceding space.

*n*, resp. *o* A few macros from *xinttools* do not expand, or expand only once their argument. This is also signaled in the margin with notations à la  $\LaTeX$ 3.

## 8.5. Output formats of macros

We do not consider here the `\xintexpr`-parsers but only the macros from *xintcore*, *xint* and *xintfrac*. Macros of other components of the bundle may have their own output formats, for example for continuous fractions with *xintcfrac*. There are mainly three types of outputs:

- arithmetic macros from *xintcore*/*xint* deliver integers in the strict format as described in the previous section.
- arithmetic macros from *xintfrac* produce on output the strict fraction format  $A/B[N]$ , which stands for  $(A/B) \times 10^N$ , where  $A$  and  $B$  are integers,  $B$  is positive, and  $N$  is a ``short'' integer. The output is not reduced to smallest terms. The  $A$  and  $B$  may end with zeroes (i.e.  $N$  does not represent all powers of ten). The denominator  $B$  is always strictly positive. There is no  $+$  sign. The  $-$  is always first if present (i.e. the denominator on output is always positive.) The output will be expressed as such a fraction even if the inputs are both integers and the mathematical result is an integer. The  $B=1$  is not removed.<sup>54</sup>
- macros from *xintfrac* having *Float* in their names deliver a number in the scientific notation as described in the documentation of `\xintFloat`.

The exception is `\xintPFloat` which does some customizable pretty printing of the result.

## 8.6. Count registers and variables

Inside `\xintexpr...\relax` and its variants, a count register or count control sequence is automatically unpacked using `\number`, with tacit multiplication: `1.23\counta` is like `1.23*\number\counta`. There is a subtle difference between count *registers* and count *variables*. In `1.23*\counta` the unpacked `\counta` variable defines a complete operand thus `1.23*\counta 7` is a syntax error. But `1.23*\count0` just replaces `\count0` by `\number\count0` hence `1.23*\count0 7` is like `1.23*57` if `\count0` contains the integer value 5.

Regarding now the package macros, there is first the case of arguments having to be short integers: this means that they are fed to a `\numexpr...\relax`, hence submitted to a *complete expansion* which must deliver an integer, and count registers and even algebraic expressions with them like `\mycountA+\mycountB*17-\mycountC/12+\mycountD` are admissible arguments (the slash stands here for the rounded integer division done by `\numexpr`). This applies in particular to the number of digits to truncate or round with, to the indices of a series partial sum, ...

The macros allowing the extended format for long numbers or dealing with fractions will to *some extent* allow the direct use of count registers and even infix algebra inside their arguments: a count register `\mycountA` or `\count 255` is admissible as numerator or also as denominator, with no need to be prefixed by `\the` or `\number`. It is possible to have as argument an algebraic expression as would be acceptable by a `\numexpr...\relax`, under this condition: *each of the numerator and denominator is expressed with at most nine tokens*.<sup>55</sup> <sup>56</sup> Important: a slash for rounded division in a `\numexpr` should be written with braces `{/}` to not be confused with the *xintfrac* delimiter between numerator and denominator (braces will be removed internally and the slash will count for one token). Example: `\mycountA+\mycountB{/}17/1+\mycountA*\mycountB`, or `\count 0+\count 2{/}17/1+\count 0*\count 2`.

```
\cnta 10 \cntb 35 \xintRaw {\cnta+\cntb{/}17/1+\cnta*\cntb}->12/351[0]
```

For longer algebraic expressions using count registers, there are two possibilities:

<sup>54</sup> refer to the documentation of `\xintPRaw` for an alternative. <sup>55</sup> The 1.2k and earlier versions manual claimed up to 8 tokens, but low-level TeX error arose if the `\numexpr...\relax` occupied exactly 8 tokens *and* evaluated to zero. With 1.21 and later, up to 9 tokens are always safe and one may even drop the ending `\relax`. But well, all these explanations are somewhat silly because prefixing by `\the` or `\number` is always working with arbitrarily many tokens. <sup>56</sup> Attention! in the  $\LaTeX$  context a `\value{counternam}` will behave ok only if it is first in the input, if not it will not get expanded, and braces around the name will be removed and chaos will ensue inside a `\numexpr`. One should enclose the whole input in `\the\numexpr...\relax` in such cases.





1. let the numerator and the denominator be presented as `\the\numexpr...\relax`,
2. or as `\numexpr {...}\relax` (the braces are removed during processing; they are not legal for `\numexpr...\relax` syntax.)

```
\cnta 100 \cntb 10 \cntc 1
\xintPraw {\numexpr {\cnta*\cnta+\cntb*\cntb+\cntc*\cntc+
2*\cnta*\cntb+2*\cnta*\cntc+2*\cntb*\cntc}\relax/%
\numexpr {\cnta*\cnta+\cntb*\cntb+\cntc*\cntc}\relax }
12321/10101
```

## 8.7. Dimension registers and variables

`<dimen>` variables can be converted into (short) integers suitable for the `xint` macros by prefixing them with `\number`. This transforms a dimension into an explicit short integer which is its value in terms of the `sp` unit (1/65536pt). When `\number` is applied to a `<glue>` variable, the stretch and shrink components are lost.

For  $\TeX$  users: a length is a `<glue>` variable, prefixing a length macro defined by `\newlength` with `\number` will thus discard the `plus` and `minus` glue components and return the dimension component as described above, and usable in the `xint bundle` macros.

This conversion is done automatically inside an `\xintexpr`-essions, with tacit multiplication implied if prefixed by some (integral or decimal) number.

One may thus compute areas or volumes with no limitations, in units of `sp^2` respectively `sp^3`, do arithmetic with them, compare them, etc..., and possibly express some final result back in another unit, with the suitable conversion factor and a rounding to a given number of decimal places.

A [table of dimensions](#) illustrates that the internal values used by  $\TeX$  do not correspond always to the closest rounding. For example a millimeter exact value in terms of `sp` units is `72.27/10/2.54*65536=186467.981...` and  $\TeX$  uses internally `186467sp` ( $\TeX$  truncates to get an integral multiple of the `sp` unit; see at the end of this section the exact rules applied internally by  $\TeX$ ).

| Unit | definition      | Exact value in sp units        | $\TeX$ 's value in sp units | Relative error |
|------|-----------------|--------------------------------|-----------------------------|----------------|
| cm   | 0.01 m          | 236814336/127 = 1864679.811... | 1864679                     | -0.0000%       |
| mm   | 0.001 m         | 118407168/635 = 186467.981...  | 186467                      | -0.0005%       |
| in   | 2.54 cm         | 118407168/25 = 4736286.720...  | 4736286                     | -0.0000%       |
| pc   | 12 pt           | 786432 = 786432.000...         | 786432                      | 0%             |
| pt   | 1/72.27 in      | 65536 = 65536.000...           | 65536                       | 0%             |
| bp   | 1/72 in         | 1644544/25 = 65781.760...      | 65781                       | -0.0012%       |
| 3bp  | 1/24 in         | 4933632/25 = 197345.280...     | 197345                      | -0.0001%       |
| 12bp | 1/6 in          | 19734528/25 = 789381.120...    | 789381                      | -0.0000%       |
| 72bp | 1 in            | 118407168/25 = 4736286.720...  | 4736286                     | -0.0000%       |
| dd   | 1238/1157 pt    | 81133568/1157 = 70124.086...   | 70124                       | -0.0001%       |
| 11dd | 11*1238/1157 pt | 892469248/1157 = 771364.950... | 771364                      | -0.0001%       |
| 12dd | 12*1238/1157 pt | 973602816/1157 = 841489.037... | 841489                      | -0.0000%       |
| sp   | 1/65536 pt      | 1 = 1.000...                   | 1                           | 0%             |

### $\TeX$ dimensions

There is something quite amusing with the Didot point. According to the  $\TeX$ Book, `1157dd=1238pt`. The actual internal value of `1dd` in  $\TeX$  is `70124sp`. We can use `xintcfrc` to display the list of centered convergents of the fraction `70124/65536`:

```
\xintListWithSep{, }{\xintFtoCCv{70124/65536}}
1/1, 15/14, 61/57, 107/100, 1452/1357, 17531/16384, and we don't find 1238/1157 therein, but another approximant 1452/1357!
```



And indeed multiplying 70124/65536 by 1157, and respectively 1357, we find the approximations (wait for more, later):

```
``1157 dd' '=1237.998474121093...pt
``1357 dd' '=1451.999938964843...pt
```

and we seemingly discover that 1357 dd=1452 pt is *far more accurate* than the T<sub>E</sub>XBook formula 1157 d d=1238 pt ! The formula to compute N dd was

```
\xinttheexpr trunc(N\dimexpr ldd\relax/\dimexpr 1pt\relax,12)\relax}
```

What's the catch? The catch is that T<sub>E</sub>X does not compute 1157 dd like we just did:

```
1157 dd=\number\dimexpr 1157dd\relax/65536=1238.000000000000...pt
1357 dd=\number\dimexpr 1357dd\relax/65536=1452.001724243164...pt
```

We thus discover that T<sub>E</sub>X (or rather here, e-T<sub>E</sub>X, but one can check that this works the same in T<sub>E</sub>X82), uses 1238/1157 as a conversion factor (and necessarily intermediate computations simulate higher precision than a priori available with integers less than 2<sup>31</sup> or rather 2<sup>30</sup> for dimensions). Hence the 1452/1357 ratio is irrelevant, an artefact of the rounding (or rather, as we see, truncating) for one dd to be expressed as an integral number of sp's.

Let us now use [\xintexpr](#) to compute the value of the Didot point in millimeters, if the above rule is exactly verified:

```
\xinttheexpr trunc(1238/1157*25.4/72.27,12)\relax=0.376065027442...mm
```

This fits very well with the possible values of the Didot point as listed in the [Wikipedia Article](#). The value 0.376065 mm is said to be *the traditional value in European printers' offices*. So the 1157 dd=1238 pt rule refers to this Didot point, or more precisely to the *conversion factor* to be used between this Didot and T<sub>E</sub>X points.

The actual value in millimeters of exactly one Didot point as implemented in T<sub>E</sub>X is

```
\xinttheexpr trunc(\dimexpr ldd\relax/65536/72.27*25.4,12)\relax
=0.376064563929...mm
```

The difference of circa 5 Å is arguably tiny!

By the way the *European printers' offices* (dixit Wikipedia) *Didot* is thus exactly

```
\xinttheexpr reduce(.376065/(25.4/72.27))\relax=543564351/508000000 pt
```

and the centered convergents of this fraction are 1/1, 15/14, 61/57, 107/100, 1238/1157, 11249/10513, 23736/22183, 296081/276709, 615898/575601, 11382245/10637527, 22148592/20699453, 188570981/176233151, 543564351/508000000. We do recover the 1238/1157 therein!

Here is how T<sub>E</sub>X converts `abc.xyz...<unit>`. First the decimal is *rounded* to the nearest integral multiple of 1/65536, say X/65536. The <unit> is associated to a ratio N/D, which represents <unit>/pt. For the Didot point the ratio is indeed 1238/1157. T<sub>E</sub>X truncates the fraction XN/D to an integer M. The dimension is represented by M sp.

## 8.8. \ifcase, \ifnum, ... constructs

When using things such as `\ifcase \xintSgn{A}` one has to make sure to leave a space after the closing brace for T<sub>E</sub>X to stop its scanning for a number: once T<sub>E</sub>X has finished expanding `\xintSgn{A}` and has so far obtained either 1, 0, or -1, a space (or something 'unexpandable') must stop it looking for more digits. Using `\ifcase\xintSgnA` without the braces is very dangerous, because the blanks (including the end of line) following `A` will be skipped and not serve to stop the number which `\ifcase` is looking for.

```
\begin{enumerate}[nosep]\def\A{1}
\item \ifcase \xintSgn\A 0\or OK\else ERROR\fi
\item \ifcase \xintSgn\A\space 0\or OK\else ERROR\fi
\item \ifcase \xintSgn{\A} 0\or OK\else ERROR\fi
\end{enumerate}
```

1. ERROR

2. OK

3. OK

In order to use successfully `\if...\fi` constructions either as arguments to the [xint bundle](#) expandable macros, or when building up a completely expandable macro of one's own, one needs some  $\TeX$  technical expertise (see also [item 2](#) on page 131).

It is thus much to be recommended to use the expandable branching macros, provided by [xintfrac](#) such as `\xintifSgn`, `\xintifZero`, `\xintifOne`, `\xintifNotZero`, `\xintifTrueAelseB`, `\xintifCmp`, `\xintifGt`, `\xintifLt`, `\xintifEq`, `\xintifInt`... See their respective documentations. All these conditionals always have either two or three branches, and empty brace pairs `{}` for unused branches should not be forgotten.

If these tests are to be applied to standard  $\TeX$  short integers, it is more efficient to use (under  $\LaTeX$ ) the equivalent conditional tests from the [etoolbox](#)<sup>57</sup> package.

## 8.9. No variable declarations are needed

There is no notion of a *declaration of a variable*.

To do a computation and assign its result to some macro `\z`, the user will employ the `\def`, `\edef`, or `\newcommand` (in  $\LaTeX$ ) as usual, keeping in mind that two expansion steps are needed, thus `\edef` is initially the main tool:


```
\def\x{1729728} \def\y{352827927} \edef\z{\xintiiMul {\x}{\y}}
\meaning\z
```

macro:->610296344513856

As an alternative to `\edef` the package provides `\oodef` which expands exactly twice the replacement text, and `\fdef` which applies *f-expansion* to the replacement text during the definition.

```
\def\x{1729728}\def\y{352827927}
\oodef\w {\xintiiMul\x\y} \fdef\z{\xintiiMul {\x}{\y}}
\meaning\w, \meaning\z
```

macro:->610296344513856, macro:->610296344513856

 In practice `\oodef` is slower than `\edef`, except for computations ending in very big final replacement texts (thousands of digits). On the other hand `\fdef` appears to be slightly faster than `\edef` already in the case of expansions leading to only a few dozen digits.

[xintexpr](#) does provide an interface to declare and assign values to identifiers which can then be used in expressions: [subsection 2.9](#).

## 8.10. Possible syntax errors to avoid

Here is a list of imaginable input errors. Some will cause compilation errors, others are more annoying as they may pass through unsignaled.

- using `-` to prefix some macro: `-\xintiiSqr{35}/271`.<sup>58</sup>
- using one pair of braces too many `\xintIrr{{\xintiiPow {3}{13}}/243}` (the computation goes through with no error signaled, but the result is completely wrong).
- things like `\xintiiAdd { \x}{\y}` as the space will cause `\x` to be expanded later, most probably within a `\numexpr` thus provoking possibly an arithmetic overflow.
- using `[]` and decimal points at the same time `1.5/3.5[2]`, or with a sign in the denominator `3/-5[7]`. The scientific notation has no such restriction, the two inputs `1.5/-3.5e-2` and `-1.5e2/3.5` are equivalent: `\xintRaw{1.5/-3.5e-2}=-15/35[2]`, `\xintRaw{-1.5e2/3.5}=-15/35[2]`.

<sup>57</sup> <https://ctan.org/pkg/etoolbox> <sup>58</sup> to the contrary, this is allowed inside an `\xintexpr`-ession.

- generally speaking, using in a context expecting an integer (possibly restricted to the  $\text{\TeX}$  bound) a macro or expression which returns a fraction: `\xinttheexpr 4/2\relax` outputs `4/2`, not `2`. Use `\xintNum {\xinttheexpr 4/2\relax}` or `\xinttheiexpr 4/2\relax` (which rounds the result to the nearest integer, here, the result is already an integer) or `\xinttheiiexpr 4/2\relax`. Or, divide in your head `4` by `2` and insert the result directly in the  $\text{\TeX}$  source.

## 8.11. Error messages

In situations such as division by zero, the  $\text{\TeX}$  run will be interrupted with some error message. It conveys some short information on the cause of the problem,<sup>59</sup> then an optimistic statement about a possible recovery if the user (in interactive mode) simply hits the `<return>` key. In non-interactive (`nonstopmode`) the  $\text{\TeX}$  run goes on uninterrupted and the error data will be found in the compilation log. Often, `xint` will fall-back to using a zero value. This is still an experimental feature.<sup>60 61</sup>

The encouragements will be slightly better formatted if the run is with  $\text{\LaTeX}$  compared to Plain  $\varepsilon\text{-}\text{\TeX}$ : Plain by default does not set the `\newlinechar` which allows to issue linebreaks in messages at chosen locations. In the examples here, `xintsession` is used, and it loads `xint` in a way activating the nicer `\newlinechar` formatted messages, even though it runs (a priori, but not necessarily) under Plain  $\varepsilon\text{-}\text{\TeX}$ .

```
>>> 1/0;
Runaway argument?
! xint error: Division by zero: 1/0.
! Paragraph ended before \xint<...> is done, but will resume:
  hit <return> at the ? prompt to try fixing the error above
  which has been encountered before expansion was complete.
<to be read again>
      \par
...
1.602 \xintsession
      \endinput%^M
?
@_1      0
>>> (-1)^3.2;
Runaway argument?
! xint error: Fractional power 32/1[-1] of negative -1[0].
! Paragraph ended before \xint<...> is done, but will resume:
  hit <return> at the ? prompt to try fixing the error above
  which has been encountered before expansion was complete.
<to be read again>
      \par
...
1.602 \xintsession
      \endinput%^M
?
@_2      0
>>> cos 1);
Runaway argument?
! xint error: `cos1' unknown, say `Isome_var' or I use 0.
! Paragraph ended before \xint<...> is done, but will resume:
```

Changed  
at 1.4m!

<sup>59</sup> The wording of these messages has been last modified at 1.4m. <sup>60</sup> Customizable handlers, error traps, error flags are implemented in embryonic form but without user interface since 1.21 release. This is not ready yet. <sup>61</sup> The 1.4g new formatting implementation benefited from a May 2021 thread at the  $\text{\LaTeX}$ 3 site where expandable error messages were discussed, with in particular contributions of @blefloch and @Skillmon.

```

hit <return> at the ? prompt to try fixing the error above
which has been encountered before expansion was complete.
<to be read again>
        \par
...
1.602 \xintsession
        \endinput%^M
?
Runaway argument?
! xint error: Extra ) removed. Hit <return>, fingers crossed.
! Paragraph ended before \xint<...> is done, but will resume:
hit <return> at the ? prompt to try fixing the error above
which has been encountered before expansion was complete.
<to be read again>
        \par
...
1.602 \xintsession
        \endinput%^M
?
@_3      0
>>> 3=4;
Runaway argument?
! xint error: Expected an operator but got `='. Ignoring.
! Paragraph ended before \xint<...> is done, but will resume:
hit <return> at the ? prompt to try fixing the error above
which has been encountered before expansion was complete.
<to be read again>
        \par
...
1.602 \xintsession
        \endinput%^M
?
@_4      12
>>> &bye

```

In the last example, tacit multiplication was applied as `xintexpr` was looking for an operator, got some invalid input and then a number.

Some constructs in `xintexpr`-essions use delimited macros and there is thus possibility in case of an ill-formed expression to end up beyond the `\relax` end-marker. Such a situation can also occur from `\relax` being swallowed by a non-terminated `\numexpr`:

```
\xintexpr 3 + \numexpr 5+4\relax followed by some LaTeX code...
```

The correct input is

```
\xintexpr 3 + \numexpr 5+4\relax\relax
```

But people in their right mind will have done

```
\xintexpr 3 + 5 + 4\relax
```

A few will have done the computation in their heads.

In such cases low-level errors will arise and may lead to very cryptic messages; but nothing unusual or especially traumatizing for the daring experienced  $\TeX$ / $\LaTeX$  user, whose has seen zillions of un-helpful error messages already in her daily practice of  $\TeX$ / $\LaTeX$ .

## 8.12. Package namespace, catcodes

This section reviews (probably with some omissions) important miscellany regarding control sequence names and catcode matters and is basically in its entirety a

**T<sub>E</sub>X-hackers note:**

- The bundle packages force the `\space` and `\empty` control sequences into having their default meanings as in Plain T<sub>E</sub>X or M<sub>T</sub>E<sub>X</sub>2e formats.
- Private macros (or internally used `\count` registers, and one private `\toks`) have names starting with `\xint_` or `\XINT_`. Some, for legacy or technical reasons, have `\xint` or `\XINT` prefix with no underscore.
- All public macros have their names starting with `\xint` except for: the *xintkernel* provided `\odef`, `\ood`, `\ef`, `\fdef`. If macros with these names already exist *xinttools* will not overwrite them. Their meanings are also available under the names `\xintodef`, `\xintoodef`, etc...
- For the *xintfrac* macros to be able to parse their inputs, standard catcodes in the argument are assumed for the digits (of course), the plus and minus signs, the dot, the letter *e*, the forward slash, the square brackets. Spaces should be avoided although they may go unnoticed sometimes.
- For the *xintexpr* expressions there is more leeway: the digit tokens must have their standard catcodes, the letters must have their standard catcodes for variable and function names to be recognized, but other characters may mostly have unusual (but not extreme like catcode zero or one) catcodes. Active characters will be expanded and should usually be prefixed with `\string`. But, if activated via Babel this is not needed.

New with  
1.4n

A few syntax elements are implemented via delimited macros. So the comma, the equal sign and the closing parenthesis must have their normal catcodes for these syntax elements to work. `\string` won't do.

Spaces are gobbled. The *e* of scientific notation may be *E* on input, *xintfrac* macros on the other hand will not recognize the *E*.

- `\xintdefvar` and `\xintdeffunc` as they use automatically `\xintexprSafeCatcodes` and `\xintexprRestoreCatcodes` to temporarily set catcodes to safe values.
- `\xintexprSafeCatcodes` and `\xintexprRestoreCatcodes` can be employed at user level too.
- At loading time the catcode configuration may be arbitrary as long as it satisfies the following requirements:
  - `%` has its normal category code,
  - `\` has its normal category code,
  - Latin letters have their normal category code "letter",
  - Digits have their normal category code "other".

Nothing more is assumed, for example `{` and `}` may have unusual catcodes at package loading time. This will be admittedly unusual especially in M<sub>T</sub>E<sub>X</sub> as `\usepackage{xintexpr}` would then have had to be replaced by something such as `\usepackage<xintexpr>...`

- Loading the packages causes no insertion of space tokens.
- The previous two items also apply to usage of `\xintreloadxintlog` and of `\xintreloadxinttrig`.

## 8.13. Origins of the package

2013/03/28. Package *bigintcalc* by HEIKO OBERDIEK already provides expandable arithmetic operations on "big integers", i.e. integers beyond the T<sub>E</sub>X bound  $2^{31} - 1$ , so why another<sup>62</sup> one?

I got started on this in early March 2013, via a thread on the *c.t.tex* usenet group, where ULRICH DIEZ used the previously cited package together with a macro (`\ReverseOrder`) which I had contributed to another thread.<sup>63</sup> What I had learned in this other thread thanks to interaction with ULRICH DIEZ and GL on expandable manipulations of tokens motivated me to try my hands at addition and multiplication.

I wrote macros `\bigMul` and `\bigAdd` which I posted to the newsgroup; they appeared to work comparatively fast. These first versions did not use the  $\varepsilon$ -T<sub>E</sub>X `\numexpr` primitive, they worked one digit at a time, having previously stored carry-arithmetic in 1200 macros.

I noticed that the *bigintcalc* package used `\numexpr` if available, but (as far as I could tell) not to do computations many digits at a time. Using `\numexpr` for one digit at a time for `\bigAdd` and

<sup>62</sup> this section was written before the *xintfrac* package; the author is not aware of another package allowing expandable computations with arbitrarily big fractions. <sup>63</sup> the `\ReverseOrder` could be avoided in that circumstance, but it does play a crucial rôle here.

\bigMul slowed them a tiny bit but avoided cluttering TeX memory with the 1200 macros storing pre-computed digit arithmetic. I wondered if some speed could be gained by using \numexpr to do four digits at a time for elementary multiplications (as the maximal admissible number for \numexpr has ten digits).

2013/04/14. This initial [xint](#) was followed by [xintfrac](#) which handled exactly fractions and decimal numbers.

2013/05/25. Later came [xintexpr](#) and at the same time [xintfrac](#) got extended to handle floating point numbers.

2013/11/22. Later, [xinttools](#) was detached.

2014/10/28. Release 1.1 significantly extended the [xintexpr](#) parsers.

2015/10/10. Release 1.2 rewrote the core integer routines which had remained essentially unmodified, apart from a slight improvement of division early 2014.

This 1.2 release also got its impulse from a fast ``reversing'' macro, which I wrote after my interest got awakened again as a result of correspondence with Bruno Le Floch during September 2015: this new reverse uses a TeXnique which *requires* the tokens to be digits. I wrote a routine which works (expandably) in quasi-linear time, but a less fancy  $O(N^2)$  variant which I developed concurrently proved to be faster all the way up to perhaps 7000 digits, thus I dropped the quasi-linear one. The less fancy variant has the advantage that [xint](#) can handle numbers with more than 19900 digits (but not much more than 19950). This is with the current common values of the input save stack and maximal expansion depth: 5000 and 10000 respectively.

## 9. Macros of the **xintkernel** package

|    |                                                                     |     |     |                                                                 |     |
|----|---------------------------------------------------------------------|-----|-----|-----------------------------------------------------------------|-----|
| .1 | <code>\odef</code> , <code>\oodef</code> , <code>\fdef</code> ..... | 142 | .6  | <code>\xintFirstOne</code> .....                                | 143 |
| .2 | <code>\xintReverseOrder</code> .....                                | 142 | .7  | <code>\xintLastOne</code> .....                                 | 143 |
| .3 | <code>\xintLength</code> .....                                      | 142 | .8  | <code>\xintReplicate</code> , <code>\xintreplicate</code> ..... | 143 |
| .4 | <code>\xintFirstItem</code> .....                                   | 143 | .9  | <code>\xintGobble</code> , <code>\xintgobble</code> .....       | 144 |
| .5 | <code>\xintLastItem</code> .....                                    | 143 | .10 | (WIP) <code>\xintUniformDeviate</code> .....                    | 144 |

The **xintkernel** package contains mainly the common code base for handling the load-order of the bundle packages, the management of catcodes at loading time, definition of common constants and macro utilities which are used throughout the code etc ... it is automatically loaded by all packages of the bundle.

It provides a few macros possibly useful in other contexts.

[source](#)   [source](#)   [source](#)

### 9.1. `\odef`, `\oodef`, `\fdef`

`\oodef\controlsequence {<stuff>}` does

```
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\controlsequence
\expandafter\expandafter\expandafter{<stuff>}
```

This works only for a single `\controlsequence`, with no parameter text, even without parameters. An alternative would be:

```
\def\oodef #1#{\def\oodefparametertext{#1}%
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\oodefparametertext
\expandafter\expandafter\expandafter }
```

but it does not allow `\global` as prefix, and, besides, would have anyhow its use (almost) limited to parameter texts without macro parameter tokens (except if the expanded thing does not see them, or is designed to deal with them).

There is a similar macro `\odef` with only one expansion of the replacement text `<stuff>`, and `\fdef` which expands fully `<stuff>` using `\romannumeral-`0`.

They can be prefixed with `\global`. It appears than `\fdef` is generally a bit faster than `\edef`, `f` when expanding macros from the **xint bundle**, when the result has a few dozens of digits. `\oodef` needs thousands of digits it seems to become competitive.

**xintkernel** will not define these macros if the control sequence names already exist. It provides them always under the names `\xintodef`, `\xintoodef` and `\xintfdef` respectively.

[source](#)

### 9.2. `\xintReverseOrder`

**n ★** `\xintReverseOrder{<list>}` does not do any expansion of its argument and just reverses the order of the tokens in the `<list>`. Braces are removed once and the enclosed material, now unbraced, does not get reversed. Unprotected spaces (of any character code) are gobbled.

```
\xintReverseOrder{\xintDigitsOf\xintiiPow {2}{100}\to\Stuff}
gives: \Stuff\to1002\xintiiPow\xintDigitsOf
```

**xinttools** provides a variant `\xintRevWithBraces` which keeps brace pairs in the output, and `f-` expands its input first.

For inputs consisting only digit tokens, see `\xintReverseDigits` from **xint**.

[source](#)

### 9.3. `\xintLength`

**n ★** `\xintLength{<list>}` counts how many tokens (or braced items) there are (possibly none). It does no expansion of its argument, so to use it to count things in the replacement text of a macro `\x` one



should do `\expandafter\xintLength\expandafter{\x}`. Blanks between items are not counted. See also `\xintNthElt{0}` (from *xinttools*) which first *f-expands* its argument and then applies the same code.

```
\xintLength {\xintiiPow {2}{100}}=3
≠ \xintLen {\xintiiPow {2}{100}}=31
```

The maximal input size is limited by  $\text{\TeX}$  main memory (it seems to be about half of the  $\text{\TeX}$ Live `main_memory` setting from file `texmf.cnf`. Because  $\text{\TeX}$  main memory is also where the used format is stored, as well as all additional defined macros, it will for example be higher if compiling with *etex* (PDF $\text{\TeX}$  in dvi mode) than with *pdftex*. Testing expansion inside an `\edef` with *etex* the author obtained with  $\text{\TeX}$ Live 2025 a limit of 2492667 tokens with 1.4n.

*source*

## 9.4. `\xintFirstItem`

*n* ★ `\xintFirstItem{<list>}` returns the first item of its argument, one pair of braces removed. If the list has no items the output is empty.

It does no expansion. For this and the next similar ones, see *xintsource.pdf* for comments on limitations.

*source*

## 9.5. `\xintLastItem`

Added at 1.2i.

*n* ★ `\xintLastItem{<list>}` returns the last item of its argument, one pair of braces removed. If the list has no items the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s. See also `\xintNthElt{-1}` from *xinttools* which obtains the same result (but with another code) after having however *f-expanded* its argument first.

*source*

## 9.6. `\xintFirstOne`

*n* ★ `\xintFirstOne{<list>}` returns the first item as a braced item. i.e. if it was braced the braces are kept, else the braces are added. It looks like using `\xintFirstItem` within braces, but the difference is when the input was empty. Then the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s.

*source*

## 9.7. `\xintLastOne`

*n* ★ `\xintLastOne{<list>}` returns the last item as a braced item. i.e. if it was braced the braces are kept, else the braces are added. It looks like using `\xintLastItem` within braces, but the difference is when the input was empty. Then the output is empty.

It does no expansion, which should be obtained via suitable `\expandafter`'s.

*source*

*source*

## 9.8. `\xintReplicate`, `\xintreplicate`

*num* *x* *n* ★ `\romannumeral\xintreplicate{x}{<stuff>}` is simply copied over from  $\text{\TeX}$ 3's `\prg_replicate:nn` with some minor changes.<sup>64</sup>

And `\xintReplicate{x}` integrates the `\romannumeral` prefix.

It does not do any expansion of its second argument but inserts it in the upcoming token stream precisely *x* times. Using it with a negative *x* raises no error and does nothing.<sup>65</sup>

<sup>64</sup> I started with the code from Joseph WRIGHT available on an online site. <sup>65</sup> This behaviour may change in future.

## 9.9. `\xintGobble`, `\xintgobble`

num x ★ `\romannumeral\xintgobble{x}` is a Gobbling macro written in the spirit of  $\TeX$ 's `\prg_replicate:n` (which I cloned as `\xintreplicate`.) It gobbles `x` tokens upstream, with `x` allowed to be as large as 531440. Don't use it with `x<0`.

And `\xintGobble{x}` integrates the `\romannumeral`.

`\xintgobble` looks as if it must be related to `\xintTrim` from *xinttools*, but the latter uses different code (using directly `\xintgobble` is not possible because one must make sure not to gobble more than the number of available items; and counting available items first is an overhead which `\xintTrim` avoids.) It is rather `\xintKeep` with a negative first argument which hands over to `\xintgobble` (because in that case it is needed to count anyhow beforehand the number of items, hence `\xintgobble` can then be used safely.)

I wrote an `\xintcount` in the same spirit as `\xintreplicate` and `\xintgobble`. But it needs to be counting hundreds of tokens to be worth its salt compared to `\xintlength`.

## 9.10. (WIP) `\xintUniformDeviate`

num x ★ `\xintUniformDeviate{x}` is based upon the engine `\pdfuniformdeviate` (PDF $\TeX$ ) or `\uniformdeviate` (Xe $\TeX$ , Lua $\TeX$ ).<sup>66</sup>

The argument is expanded in `\numexpr` and the macro itself needs two expansion steps. It produces like the engine primitive an integer (digit tokens) with minimal value 0 and maximal one `x-1` if `x` is positive, or minimal value `x+1` and maximal value 0 if `x` is negative. For the discussion next, `x` is supposed positive as this avoids having to insert absolute values in formulas.

The underlying engine primitive accesses a random number generator (RNG) originally embedded into MetaPost and described in *The Art of Computer Programming*, Vol. 2. During discussions with Bruno Le Floch in May 2018, when he was adding to  $\TeX$ 3 interface for randomness, the author became aware of some limitations, some of them surprising, in the randomness of the numbers produced by this RNG. For example, the values produced by `\pdfuniformdeviate 201326592` and reduction modulo three are in the proportion 1:1:2, not 1:1:1.

Let's count how many 0's, 1's, and 2's we get from reducing modulo 3 the output of `\pdfuniformdeviate 201326592`:

```
\pdfsetrandomseed 87654321
\def\A{0}\def\B{0}\def\C{0}
\xintReplicate{504}{\ifcase\xinteval{\pdfuniformdeviate 201326592}/:3}
\edef\A{the\numexpr\A+1}\or
\edef\B{the\numexpr\B+1}\or
\edef\C{the\numexpr\C+1}\fi
}
```

We found \A{ 0's, \B{ 1's and \C{ 2's among 504 trials.

We found 124 0's, 147 1's and 233 2's among 504 trials.

In contrast, here is what happens if using `\xintUniformDeviate`:

```
\pdfsetrandomseed 87654321
\def\A{0}\def\B{0}\def\C{0}
\xintReplicate{504}{\ifcase\xinteval{\xintUniformDeviate{201326592}}/:3}
\edef\A{the\numexpr\A+1}\or
\edef\B{the\numexpr\B+1}\or
\edef\C{the\numexpr\C+1}\fi
}
```

We found \A{ 0's, \B{ 1's and \C{ 2's among 504 trials.

We found 161 0's, 174 1's and 169 2's among 504 trials.

**$\TeX$ -hackers note:** The RNG works with 28-bits integers. To output a supposedly uniform random integer in a given range `0..x-1`, it first produces a supposedly uniform integer in the range `0..228 - 1` (where  $2^{28} =$

<sup>66</sup> The `\uniformdeviate` primitive was added to Xe $\TeX$  for the  $\TeX$ Live 2019 release.

268435456) then rescales (with rounding) to the target range. Note that with  $x = 2^{29}$  this means in particular that all produced "random numbers" in the  $0..x-1$  range will be even...

But even with  $x = 2^{28}$  there are some more serious defects of the RNG: two seeds sharing the same low  $k$  bits generate sequences of 28-bits integers which are identical one-to-one modulo  $2^k$ ! In particular after setting the seed, there are only 2 distinct sequences for the parity bits for the integers generated by `\pdfuniformdeviate 268435456`.

Let's define the non-uniformity of `\pdfuniformdeviate x` to be the maximum, taken over all  $y$ 's from 0 to  $x-1$ , of  $|x \times \text{Prob}(\text{\pdfuniformdeviate } x = y) - 1|$ . For a general  $x$ , the engine primitive guarantees only a  $x/2^{28}$  relative non-uniformity for the outputs of `\pdfuniformdeviate x`.

`\xintUniformDeviate` improves this by a factor of  $2^{28}=268435456$ : the relative non-uniformity now is guaranteed to be bounded above by  $x/2^{56}$ . With such a small non-uniformity, modulo phenomena as mentioned earlier are not observable in reasonable computing time.

The implementation of `\xintUniformDeviate` consumes exactly 5 calls to the engine primitive at each execution; the improved  $x/2^{56}$  non-uniformity could be obtained with only 2 calls, but paranoia about the phenomenon of seeds with common bits has led me to accept the overhead of using the 7 high bits of 4 random 28-bits integers, rather than one single 28-bits integer, or two, or three.

Timings indicate that one `\xintUniformDeviate` has a time cost about 13 times the one for one call to the engine primitive (and not only 5, as the extra arithmetic expressions add overhead which is more costly than the primitive itself). Except if the code using the pseudo-random number is very short, this time penalty will prove in practice much less severe (and this is one important reason why we opted for obtaining 28bits via the 7 high bits of 4 successive pseudo random numbers from the engine primitive).

For example let's raise 100 times a random integer in the  $0..99999999$  range to the tenth power:<sup>67</sup>

```
\pdfsetrandomseed 12345678
\xintresettimer
\xintReplicate{100}{\edef\foo{\xintiiPow{\xintUniformDeviate{1000000000}}{10}}}%
\xinttheseconds s (if using \string\xintUniformDeviate)\newline
\pdfsetrandomseed 12345678
\xintresettimer
\xintReplicate{100}{\edef\foo{\xintiiPow{\pdfuniformdeviate 1000000000}{10}}}%
\xinttheseconds s (if using \string\pdfuniformdeviate)\par
```

0.01686s (if using `\xintUniformDeviate`)

0.01677s (if using `\pdfuniformdeviate`)

The macros `\xintRandomDigits` or `\xintiiRandRange`, and their variants, as well as the supporting macros for `random()` generate random decimal digits eight by eight as if using `\xintUniformDeviate{1000000000}`, but via a direct optimized call made possibly by the range being a power of 10.

<sup>67</sup> Timings done during dvi build on an Apple desktop with M4 Pro architecture.

## 10. Macros of the xintcore package

|     |                                |     |     |                                    |     |
|-----|--------------------------------|-----|-----|------------------------------------|-----|
| .1  | <code>\xintiNum</code> .....   | 146 | .15 | <code>\xintiiCmp</code> .....      | 148 |
| .2  | <code>\xintDouble</code> ..... | 147 | .16 | <code>\xintiiSub</code> .....      | 148 |
| .3  | <code>\xintHalf</code> .....   | 147 | .17 | <code>\xintiiMul</code> .....      | 148 |
| .4  | <code>\xintInc</code> .....    | 147 | .18 | <code>\xintiiSqr</code> .....      | 148 |
| .5  | <code>\xintDec</code> .....    | 147 | .19 | <code>\xintiiPow</code> .....      | 148 |
| .6  | <code>\xintDSL</code> .....    | 147 | .20 | <code>\xintiiFac</code> .....      | 148 |
| .7  | <code>\xintDSR</code> .....    | 147 | .21 | <code>\xintiiDivision</code> ..... | 149 |
| .8  | <code>\xintDSRr</code> .....   | 147 | .22 | <code>\xintiiQuo</code> .....      | 149 |
| .9  | <code>\xintFDg</code> .....    | 147 | .23 | <code>\xintiiRem</code> .....      | 149 |
| .10 | <code>\xintLDg</code> .....    | 147 | .24 | <code>\xintiiDivRound</code> ..... | 149 |
| .11 | <code>\xintiiSgn</code> .....  | 147 | .25 | <code>\xintiiDivTrunc</code> ..... | 149 |
| .12 | <code>\xintiiOpp</code> .....  | 148 | .26 | <code>\xintiiDivFloor</code> ..... | 150 |
| .13 | <code>\xintiiAbs</code> .....  | 148 | .27 | <code>\xintiiMod</code> .....      | 150 |
| .14 | <code>\xintiiAdd</code> .....  | 148 | .28 | <code>\xintNum</code> .....        | 150 |

Package xintcore is automatically loaded by xint.

xintcore provides for big integers the four basic arithmetic operations (addition, subtraction, multiplication, division), as well as powers and factorials.

In the descriptions of the macros `{N}` and `{M}` stand for (big) integers or macros *f-expanding* to such big integers in strict format as described in [subsection 8.4](#).

All macros require strict integer format on input and produce strict integer format on output, except:

- `\xintiNum` which converts to strict integer format an input in *extended* integer format, i.e. admitting multiple leading plus or minus signs, then possibly leading zeroes, then digits,
- and `\xintNum` which is an alias for the former, which gets redefined by xintfrac to accept more generally also decimal numbers or fractions as input and which truncates them to integers.

The *ii* in the names of the macros such as `\xintiiAdd` serves to stress that they accept only strict integers as input (this is signaled by the margin annotation *f*), or macros *f-expanding* to such strict format (big) integers and that they produce strict integers as output.

Other macros, such as `\xintDouble`, lack the *ii*, but this is only a legacy of the history of the package and they have the same requirements for input and format of output as the *ii*-macros.

The letter *x* (with margin annotation <sup>num</sup>*x*) stands for an argument which will be handled embedded in `\numexpr..\relax`. It will thus be completely expanded and must give an integer obeying the  $\TeX$  bounds. See also [subsection 8.6](#). This is the case for the argument of `\xintiiFac` or the exponent argument of `\xintiiPow`.

The *★*'s in the margin are there to remind of the complete expandability, even *f-expandability* of the macros, as discussed in [subsubsection 8.3.1](#).

[Table 3](#) summarizes the maximal allowed sizes for the four operations. The first column is the tested macro (it is expanded in an `\edef`; if deeper nested, the maximal admissible input sizes may actually prove lower than stated). The second column gives the maximal *N* such that the macro does not raise an error on both inputs having *N* digits (for division, the used test divisors had *N*/4, *N*/2 and *3N*/4 digits).

These maximal *N*'s depend on the values of  $\TeX$  parameters such as input stack size and expansion depth. The last column gives the  $\TeX$  parameter cited in the error message when trying with *N*+1 digits.

The table was last updated in July 2025, using 1.4m 2022/06/10 and TeXLive 2025 default settings which are: input stack size at 10000, expansion depth at 10000, parameter stack size at 20000.

*source*

### 10.1. `\xintiNum`

*f* ★ `\xintiNum{N}` removes chains of plus or minus signs, followed by zeroes.

|                              | Max length of inputs | Limiting factor       |
|------------------------------|----------------------|-----------------------|
| <code>\xintiiAdd</code>      | 26648                | expansion depth=10000 |
| <code>\xintiiSub</code>      | 26632                | expansion depth=10000 |
| <code>\xintiiMul</code>      | 13320                | expansion depth=10000 |
| <code>\xintiiDivision</code> | 26609                | expansion depth=10000 |

Table 3: Maximal sizes of inputs (using TeXLive 2025) for core arithmetic

```
\xintiNum{+-----000000000367941789479}
-367941789479
```

*source***10.2. \xintDouble***f* ★ `\xintDouble{N}` computes  $2N$ .*source***10.3. \xintHalf***f* ★ `\xintHalf{N}` computes  $N/2$  truncated towards zero.*source***10.4. \xintInc***f* ★ `\xintInc{N}` evaluates  $N+1$ .*source***10.5. \xintDec***f* ★ `\xintDec{N}` evaluates  $N-1$ .*source***10.6. \xintDSL***f* ★ `\xintDSL{N}` is decimal shift left, *i.e.* multiplication by ten.*source***10.7. \xintDSR***f* ★ `\xintDSR{N}` is truncated decimal shift right, *i.e.* it is the truncation of  $N/10$  towards zero.*source***10.8. \xintDSRr***f* ★ `\xintDSRr{N}` is rounded decimal shift right, *i.e.* it is the rounding of  $N/10$  away from zero. It is needed in xintcore for use by `\xintiiDivRound`.*source***10.9. \xintFDg***f* ★ `\xintFDg{N}` outputs the first digit (most significant) of the number.*source***10.10. \xintLDg***f* ★ `\xintLDg{N}` outputs the least significant digit. When the number is positive, this is the same as the remainder in the Euclidean division by ten.*source***10.11. \xintiiSgn***f* ★ `\xintiiSgn{N}` returns 1 if the number is positive, 0 if it is zero and -1 if it is negative.

*source***10.12. \xintiiOpp***f* ★ `\xintiiOpp{N}` outputs the opposite  $-N$  of the number  $N$ .

Important note: an input such as `-\foo` is not legal, generally speaking, as argument to the macros of the *xint bundle* (except, naturally in `\xintexpr`-essions). The reason is that the minus sign stops the *f*-expansion done during parsing of the inputs. One must use the syntax `\xintiiOpp{ $-\foo$ }` if one wants to pass `-\foo` as argument to other macros.

*source***10.13. \xintiiAbs***f* ★ `\xintiiAbs{N}` outputs the absolute value of the number.*source***10.14. \xintiiAdd***ff* ★ `\xintiiAdd{N}{M}` computes the sum of the two (big) integers.*source***10.15. \xintiiCmp***ff* ★ `\xintiiCmp{N}{M}` produces 1 if  $N > M$ , 0 if  $N = M$ , and -1 if  $N < M$ .

At 1.21 this macro was moved from package *xint* to *xintcore*.

*source***10.16. \xintiiSub***ff* ★ `\xintiiSub{N}{M}` computes the difference  $N - M$ .*source***10.17. \xintiiMul***ff* ★ `\xintiiMul{N}{M}` computes the product of two (big) integers.*source***10.18. \xintiiSqr***f* ★ `\xintiiSqr{N}` produces the square.*source***10.19. \xintiiPow**

*f* <sup>num</sup><sub>X</sub> ★ `\xintiiPow{N}{x}` computes  $N^x$ . For  $x=0$ , this is 1. For  $N=0$  and  $x < 0$ , or if  $|N| > 1$  and  $x < 0$ , an error is raised. There will also be an error if  $x$  exceeds the maximal  $\varepsilon$ -TeX number 2147483647, but the real limit for exponents comes from either the computation time or the settings of some TeX memory parameters.

Generally speaking the computation will end in an error if the output goes beyond what addition could accept on input, i.e. (with TeXLive 2025 default settings) has more than about 26600 digits.

For example the maximal power of 2 which `\xintiiPow` is able to compute (with TeXLive 2025 default memory parameters) is  $2^{88470}$  which has 26633 digits. I.e. `\edef\z{\xintiiPow{2}{88470}}` succeeds (if you are patient enough to wait) but `\edef\z{\xintiiPow{2}{88471}}` fails.

*source***10.20. \xintiiFac**

*num*<sub>X</sub> ★ `\xintiiFac{x}` computes the factorial.

The (theoretically) allowable range is  $0 \leq x \leq 10000$ .

However the maximal possible computation depends on the values of some memory parameters of the  $\varepsilon$ -TeX executable: with *xintcore* at 1.4m and using TeXLive 2025, the maximal (within an `\edef`) computable one is 7712! which has 26631 decimal digits. With the input stack size set at 10000, the limiting factor here is the expansion depth at 10000.

The `factorial()` function, or equivalently `!()` as post-fix operator is available in the three parsers:

Within `\xintfloateval`, the macro `\xintFloatFac` from package `xintfrac` is used.

7.886578673647905e374

3.316275092450633e5735

See its documentation for more.

## 10.21. \xintiiDivision

*source*

10.22. \xintii0uo

*source*

### 10.23. \xintiRem

*source*

### 10.24. \xintiiDivRound

```
\xintiiDivRound {100}{3}, \xintiiDivRound {101}{3}
```

33. 34

*source*

### 10.25. \xintiiDivTrunc

```
\xintiiQuo{1000}{57} (Euclidean), \xintiiDivTrunc{1000}{57} (truncated),
\xintiiDivRound{1000}{57} (rounded)\newline
\xintiiQuo{-1000}{57}, \xintiiDivTrunc{-1000}{57} (t), \xintiiDivRound{-1000}{57} (r)
\newline
\xintiiQuo{1000}{-57}, \xintiiDivTrunc{1000}{-57} (t), \xintiiDivRound{1000}{-57} (r)
\newline
\xintiiQuo{-1000}{-57}, \xintiiDivTrunc{-1000}{-57} (t), \xintiiDivRound{-1000}{-57} (r)
\par
```

17 (Euclidean), 17 (truncated), 18 (rounded)

-18, -17 (t), -18 (r)

-17, -17 (t), -18 (r)

18. 17 (t), 18 (r)



*source***10.26. `\xintiiDivFloor`**

*ff* ★ `\xintiiDivFloor{M}{N}` computes  $\text{floor}(M/N)$ . For positive divisor  $N > 0$  and arbitrary dividend  $M$  it is the same as the Euclidean quotient `\xintiiQuo`.

```
\xintiiQuo{1000}{57} (Euclidean), \xintiiDivFloor{1000}{57} (floored)\newline
\xintiiQuo{-1000}{57}, \xintiiDivFloor{-1000}{57}\newline
\xintiiQuo{1000}{-57}, \xintiiDivFloor{1000}{-57}\newline
\xintiiQuo{-1000}{-57}, \xintiiDivFloor{-1000}{-57}\par
```

17 (Euclidean), 17 (floored)

-18, -18

-17, -18

18, 17

*source***10.27. `\xintiiMod`**

*ff* ★ `\xintiiMod{M}{N}` computes  $M - N * \text{floor}(M/N)$ . For positive divisor  $N > 0$  and arbitrary dividend  $M$  it is the same as the Euclidean remainder `\xintiiRem`.

Formerly, this macro computed  $M - N * \text{trunc}(M/N)$ . The former meaning is retained as `\xintiiMod-Trunc`.

```
\xintiiRem {1000}{57} (Euclidean), \xintiiMod {1000}{57} (floored),
\xintiiModTrunc {1000}{57} (truncated)\newline
\xintiiRem {-1000}{57}, \xintiiMod {-1000}{57}, \xintiiModTrunc {-1000}{57}\newline
\xintiiRem {1000}{-57}, \xintiiMod {1000}{-57}, \xintiiModTrunc {1000}{-57}\newline
\xintiiRem {-1000}{-57}, \xintiiMod {-1000}{-57}, \xintiiModTrunc {-1000}{-57}\par
```

31 (Euclidean), 31 (floored), 31 (truncated)

26, 26, -31

31, -26, 31

26, -31, -31

*source***10.28. `\xintNum`**

*f* ★ `\xintNum` is originally an alias for `\xintiNum`. But with *xintfrac* loaded its meaning is *modified* to accept more general inputs. It then becomes an alias to `\xintTTrunc` which truncates the general input to an integer in strict format.

## 11. Macros of the **xint** package

|     |                                                                 |     |     |                                                            |     |
|-----|-----------------------------------------------------------------|-----|-----|------------------------------------------------------------|-----|
| .1  | <code>\xintLen</code> .....                                     | 152 | .30 | <code>\xintiifOdd</code> .....                             | 155 |
| .2  | <code>\xintReverseDigits</code> .....                           | 152 | .31 | <code>\xintiiSum</code> .....                              | 155 |
| .3  | <code>\xintDecSplit</code> .....                                | 152 | .32 | <code>\xintiiPrd</code> .....                              | 156 |
| .4  | <code>\xintDecSplitL</code> , <code>\xintDecSplitR</code> ..... | 153 | .33 | <code>\xintiiSquareRoot</code> .....                       | 156 |
| .5  | <code>\xintiiE</code> .....                                     | 153 | .34 | <code>\xintiiSqrt</code> , <code>\xintiiSqrtR</code> ..... | 156 |
| .6  | <code>\xintDSH</code> .....                                     | 153 | .35 | <code>\xintiiBinomial</code> .....                         | 156 |
| .7  | <code>\xintDSHr</code> , <code>\xintDSx</code> .....            | 153 | .36 | <code>\xintiiPFactorial</code> .....                       | 157 |
| .8  | <code>\xintiiEq</code> .....                                    | 153 | .37 | <code>\xintiiMax</code> .....                              | 158 |
| .9  | <code>\xintiiNotEq</code> .....                                 | 153 | .38 | <code>\xintiiMin</code> .....                              | 158 |
| .10 | <code>\xintiiGeq</code> .....                                   | 153 | .39 | <code>\xintiiMaxof</code> .....                            | 158 |
| .11 | <code>\xintiiGt</code> .....                                    | 153 | .40 | <code>\xintiiMinof</code> .....                            | 158 |
| .12 | <code>\xintiiLt</code> .....                                    | 154 | .41 | <code>\xintifTrueAelseB</code> .....                       | 158 |
| .13 | <code>\xintiiGtorEq</code> .....                                | 154 | .42 | <code>\xintifFalseAelseB</code> .....                      | 158 |
| .14 | <code>\xintiiLtorEq</code> .....                                | 154 | .43 | <code>\xintNOT</code> .....                                | 158 |
| .15 | <code>\xintiiIsZero</code> .....                                | 154 | .44 | <code>\xintAND</code> .....                                | 158 |
| .16 | <code>\xintiiIsNotZero</code> .....                             | 154 | .45 | <code>\xintOR</code> .....                                 | 159 |
| .17 | <code>\xintiiIsOne</code> .....                                 | 154 | .46 | <code>\xintXOR</code> .....                                | 159 |
| .18 | <code>\xintiiOdd</code> .....                                   | 154 | .47 | <code>\xintANDof</code> .....                              | 159 |
| .19 | <code>\xintiiEven</code> .....                                  | 154 | .48 | <code>\xintORof</code> .....                               | 159 |
| .20 | <code>\xintiiMON</code> .....                                   | 154 | .49 | <code>\xintXORof</code> .....                              | 159 |
| .21 | <code>\xintiiMMON</code> .....                                  | 154 | .50 | <code>\xintiiGCD</code> .....                              | 159 |
| .22 | <code>\xintiiifSgn</code> .....                                 | 154 | .51 | <code>\xintiiLCM</code> .....                              | 159 |
| .23 | <code>\xintiiifZero</code> .....                                | 154 | .52 | <code>\xintiiGCDof</code> .....                            | 159 |
| .24 | <code>\xintiiifNotZero</code> .....                             | 155 | .53 | <code>\xintiiLCMof</code> .....                            | 160 |
| .25 | <code>\xintiiifOne</code> .....                                 | 155 | .54 | <code>\xintLen</code> .....                                | 160 |
| .26 | <code>\xintiiifCmp</code> .....                                 | 155 | .55 | (WIP) <code>\xintRandomDigits</code> .....                 | 160 |
| .27 | <code>\xintiiifEq</code> .....                                  | 155 | .56 | (WIP) <code>\xintXRandomDigits</code> .....                | 160 |
| .28 | <code>\xintiiifGt</code> .....                                  | 155 | .57 | (WIP) <code>\xintiiRandRange</code> .....                  | 161 |
| .29 | <code>\xintiiifLt</code> .....                                  | 155 | .58 | (WIP) <code>\xintiiRandRangeAtoB</code> .....              | 161 |

This package loads automatically **xintcore** (and **xintkernel**) hence all macros described in [section 10](#) are still available.

This is 1.4o of 2025/09/06.

Version 1.0 was released 2013/03/28. Since 1.1 2014/10/28 the core arithmetic macros have been moved to a separate package **xintcore**, which is automatically loaded by **xint**. Only the `\xintiiSum`, `\xintiiPrd`, `\xintiiSquareRoot`, `\xintiiSqrt`, `\xintiiSqrtR`, `\xintiiPFactorial`, `\xintiiBinomial` genuinely add to the arithmetic macros from **xintcore**. (`\xintiiFac` which computes factorials is already in **xintcore**.)

With the exception of `\xintLen`, of the “Boolean logic macros” (see next paragraphs) all macros require inputs being integers in strict format, see [subsection 8.4](#).<sup>68</sup> The **ii** in the macro names is here as a reminder of that fact. The output is an integer in strict format, or a pair of two braced such integers for `\xintiiSquareRoot`, with the exception of `\xintiiE` which may produce strings of zero's if its first argument is zero.

Macros `\xintDecSplit` and `\xintReverseDigits` are non-arithmetic and have their own specific rules.

For all macros described here for which it makes sense, package **xintfrac** defines a similar one without **ii** in its name. This will handle more general inputs: decimal, scientific numbers, fractions. The **ii** macros provided here by **xint** can be nested inside macros of **xintfrac** but the opposite does not apply, because the output format of the **xintfrac** macros, even for representing integers,

<sup>68</sup> of course for conditionals such as `\xintiiifCmp` this constraint applies only to the first two arguments.

is not understood by the `ii` macros. The “Boolean macros” `\xintAND` etc... are exceptions though, they work fine if served as inputs some `xintfrac` output, despite doing only *f*-expansion. Prior to 1.2o, these macros did apply the `\xintNum` or the more general `xintfrac` general parsing, but this overhead was deemed superfluous as it serves only to handle hand-written input and is not needed if the input is obtained as a nested chain of `xintfrac` macros for example.

Prior to release 1.2o, `xint` defined additional macros which applied `\xintNum` to their input arguments. All these macros were deprecated at 1.2o and have been removed at 1.3.

At 1.3d macros `\xintiiGCD` and `\xintiiLCM` from package `xintgcd` are also available from loading `xint` only. They are support macros for the (multi-arguments) functions `gcd()` and `lcm()` in `\xintiexpr`.

See subsection 8.3.1 for the significance of the  $\overset{\text{Num}}{f}$ ,  $f$ ,  $\overset{\text{num}}{x}$  and  $\star$  margin annotations.

source

### 11.1. `\xintilen`

$\overset{\text{Num}}{f} \star$  `\xintilen{N}` returns the length of the number, after its parsing via `\xintNum`. The count does not include the sign.

```
\xintilen{-12345678901234567890123456789}
```

29

Prior to 1.2o, the package defined only `\xintLen`, which is extended by `xintfrac` to fractions or decimal numbers, hence acquires a bit more overhead then.

source

### 11.2. `\xintReverseDigits`

3.60004pt, 8.39996pt, 12.0pt

$f \star$  `\xintReverseDigits{N}` will reverse the order of the digits of the number. `\xintRev` is the former denomination and is kept as an alias. Leading zeroes resulting from the operation are not removed. Contrarily to `\xintReverseOrder` this macro *f*-expands its argument; it is only usable with digit tokens. It does not apply `\xintNum` to its argument (so this must be done explicitly if the argument is an integer produced from some `xintfrac` macros). It does accept a leading minus sign which will be left upfront in the output.

```
\oodef\x{\xintReverseDigits
  {9876543210987654321098765432109876543210}}\meaning\x\par
\noindent\oodef\x{\xintReverseDigits {\xintReverseDigits
  {9876543210987654321098765432109876543210}}}\meaning\x\par
```

macro:->01234567890123456789012345678901234567890123456789

macro:->98765432109876543210987654321098765432109876543210

source

### 11.3. `\xintDecSplit`

$\overset{\text{num}}{x} f \star$  `\xintDecSplit{x}{N}` cuts the *N* (a list of digits) into two pieces *L* and *R*: it outputs `{L}{R}` where the original *N* is the concatenation *LR*. These two pieces are decided according to *x*:

- for  $x > 0$ , *R* coincides with the *x* least significant digits. If *x* equals or exceeds the length of *N* the first piece *L* will thus be empty,
- for  $x = 0$ , *R* is empty, and *L* is all of *N*,
- for  $x < 0$ , the first piece *L* consists of the  $|x|$  most significant digits and the second piece *R* gets the remaining ones. If *x* equals or exceeds the length of *N* the second piece *R* will thus be empty.

This macro provides public interface to some functionality which is primarily of internal interest. It operates only (after *f*-expansion) on ‘‘strings’’ of digits tokens: leading zeroes are allowed but a leading sign (even a minus sign) will provoke an error.

Breaking change with 1.2i: formerly  $N < 0$  was replaced by its absolute value. Now, a sign (positive or negative) will create an error.

*source*

`\xintDecSplitL{x}{N}` returns the first piece (unbraced) from the `\xintDecSplit` output.  
`\xintDecSplitR{x}{N}` returns the second piece (unbraced) from the `\xintDecSplit` output.

*source*

`\xintiiE{N}{x}` serves to extend  $N$  with  $x$  zeroes. The parameter  $x$  must be non-negative. The same output would be obtained via `\xintDSH{-x}{N}`, except for  $N=0$ , as `\xintDSH{-x}{N}` multiplies  $N$  by  $10^x$  hence produces 0 if  $N=0$  whereas `\xintiiE{0}{x}` produces  $x+1$  zeros.

[illegible]

## 11.6. \xintDSH

$\sum_{x=f}^{\text{num}} f \star \backslash\text{xintDSH}\{x\}\{N\}$  is parametrized decimal shift. When  $x$  is negative, it is like iterating  $\backslash\text{xintDSL}$   $|x|$  times (i.e. multiplication by  $10^{-x}$ ). When  $x$  positive, it is like iterating  $\backslash\text{xintDSR}$   $x$  times (and is more efficient), and for a non-negative  $N$  this is thus the same as the quotient from the Euclidean division by  $10^N$ .

*source*

$\sum_x f \star$  `\xintDSHr{x}{N}` expects `x` to be zero or positive and it returns then a value `R` which is correlated to the value `Q` returned by `\xintDSH{x}{N}` in the following manner:

- if  $N$  is positive or zero,  $Q$  and  $R$  are the quotient and remainder in the Euclidean division by  $10^x$  (obtained in a more efficient manner than using `\xintiiDivision`),
- if  $N$  is negative let  $Q1$  and  $R1$  be the quotient and remainder in the Euclidean division by  $10^x$  of the absolute value of  $N$ . If  $Q1$  does not vanish, then  $Q=-Q1$  and  $R=R1$ . If  $Q1$  vanishes, then  $Q=0$  and  $R=-R1$ .
- for  $x=0$ ,  $Q=N$  and  $R=0$ .

So one has  $N = 10^x Q + R$  if  $Q$  turns out to be zero or positive, and  $N = 10^x Q - R$  if  $Q$  turns out to be negative, which is exactly the case when  $N$  is at most  $-10^x$ .

$\text{num}_x f \star \backslash\text{xintDSx}\{x\}\{N\}$  for  $x$  negative is exactly as  $\backslash\text{xintDSH}\{x\}\{N\}$ , i.e. multiplication by  $10^{-x}$ . For  $x$  zero or positive it returns the two numbers  $\{Q\}\{R\}$  described above, each one within braces. So  $Q$  is  $\backslash\text{xintDSH}\{x\}\{N\}$ , and  $R$  is  $\backslash\text{xintDSHr}\{x\}\{N\}$ , but computed simultaneously.

11.8. \xintiiEq

*ff*★ `\xintiiEq{N}{M}` returns 1 if  $N=M$ , 0 otherwise.

### 11.9. \xintiiNotEq

*ff*★ `\xintiiNotEq{N}{M}` returns 0 if  $N=M$ , 1 otherwise.

### 11.10. \xintiigeq

```
ff★ \xintiiGeq{N}{M} returns 1 if the absolute value of the first number is at least equal to the
absolute value of the second number. If  $|N| < |M|$  it returns 0.
```

Important: the macro compares *absolute values*.

11.11. \xintiiGt

*ff*★ `\xintiiGt{N}{M}` returns 1 if  $N > M$ , 0 otherwise.

*source***11.12. \xintiilt***ff* ★ `\xintiilt{N}{M}` returns 1 if  $N < M$ , 0 otherwise.*source***11.13. \xintiiGtorEq***ff* ★ `\xintiiGtorEq{N}{M}` returns 1 if  $N \geq M$ , 0 otherwise. Extended by *xintfrac* to fractions.*source***11.14. \xintiiltorEq***ff* ★ `\xintiiltorEq{N}{M}` returns 1 if  $N \leq M$ , 0 otherwise.*source***11.15. \xintiiIsZero***f* ★ `\xintiiIsZero{N}` returns 1 if  $N = 0$ , 0 otherwise.*source***11.16. \xintiiIsNotZero***f* ★ `\xintiiIsNotZero{N}` returns 1 if  $N \neq 0$ , 0 otherwise.*source***11.17. \xintiiIsOne***f* ★ `\xintiiIsOne{N}` returns 1 if  $N = 1$ , 0 otherwise.*source***11.18. \xintiiOdd***f* ★ `\xintiiOdd{N}` is 1 if the number is odd and 0 otherwise.*source***11.19. \xintiiEven***f* ★ `\xintiiEven{N}` is 1 if the number is even and 0 otherwise.*source***11.20. \xintiiMON***f* ★ `\xintiiMON{N}` computes  $(-1)^N$ .`\xintiiMON {-280914019374101929}``-1`*source***11.21. \xintiiMMON***f* ★ `\xintiiMMON{N}` computes  $(-1)^{N-1}$ .`\xintiiMMON {280914019374101929}``1`*source***11.22. \xintiiifSgn***source**fnnn* ★ `\xintiiifSgn{<N>}{<A>}{<B>}{<C>}` executes either the  $\langle A \rangle$ ,  $\langle B \rangle$  or  $\langle C \rangle$  code, depending on its first argument being respectively negative, zero, or positive.*source***11.23. \xintiiifZero***fnn* ★ `\xintiiifZero{<N>}{<IsZero>}{<IsNotZero>}` expandably checks if the first mandatory argument  $N$  (a number, possibly a fraction if *xintfrac* is loaded, or a macro expanding to one such) is zero or not. It then either executes the first or the second branch.

Beware that both branches must be present.

*source***11.24. `\xintiifNotZero`**

*fn* ★ `\xintiifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩}` expandably checks if the first mandatory argument **N** is not zero or is zero. It then either executes the first or the second branch.  
Beware that both branches must be present.

*source***11.25. `\xintiifOne`**

*fn* ★ `\xintiifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩}` expandably checks if the first mandatory argument **N** is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

*source***11.26. `\xintiifCmp`**

*ffnn* ★ `\xintiifCmp{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨A=B⟩}{⟨A>B⟩}` compares its first two arguments and chooses accordingly the correct branch.

*source***11.27. `\xintiifEq`**

*ffnn* ★ `\xintiifEq{⟨A⟩}{⟨B⟩}{⟨A=B⟩}{⟨not(A=B)⟩}` checks equality of its two first arguments and executes the corresponding branch.

*source***11.28. `\xintiifGt`**

*ffnn* ★ `\xintiifGt{⟨A⟩}{⟨B⟩}{⟨A>B⟩}{⟨not(A>B)⟩}` checks if  $A > B$  and executes the corresponding branch.

*source***11.29. `\xintiifLt`**

*ffnn* ★ `\xintiifLt{⟨A⟩}{⟨B⟩}{⟨A<B⟩}{⟨not(A<B)⟩}` checks if  $A < B$  and executes the corresponding branch.

*source***11.30. `\xintiifOdd`**

*fn* ★ `\xintiifOdd{⟨A⟩}{⟨A odd⟩}{⟨A even⟩}` checks if **A** is an odd integer and executes the corresponding branch.

*source***11.31. `\xintiiSum`**

*\*f* ★ `\xintiiSum{⟨braced things⟩}` after expanding its argument expects to find a sequence of tokens (or braced material). Each is *f-expanded*, and the sum of all these numbers is returned.

```
\xintiiSum{{123}{-98763450}{\xintiiFac{7}}{\xintiiMul{3347}{591}}}\newline
\xintiiSum{1234567890}\newline
\xintiiSum{1234}\newline
\xintiiSum{}
```

-96780210

45

10

0

A sum with only one term returns that number: `\xintiiSum {-1234}=-1234`. Attention that `\xintiiSum {-1234}` is not legal input and would make the  $\TeX$  run fail.

11.32. \xintiPrd

```
\xintiiPrd{{-9876}}{\xintiiFac{7}}{\xintiiMul{3347}{591}}}\newline
\xintiiPrd{123456789123456789}\newline
\xintiiPrd {1234}\newline
\xintiiPrdf}
```

1

$$2^{200}3^{100}7^{100}=\text{\texttt{\textbackslash printnumber}}$$

```
{\xintiiPrd {{\xintiiPow {2}{200}}{\xintiiPow {3}{100}}{\xintiiPow {7}{100}}}}$
```

With `xintexpr`, the syntax is the natural one:

```
$2^{200}3^{100}7^{100}=\backslash\mathrm{printnumber}\{\backslash\mathrm{xinttheiiexpr}\ 2^{200} * 3^{100} * 7^{100}\backslash\mathrm{relax}\}$
```

$$2^{200}3^{100}7^{100} = 2678727931661577575766279517007548402324740266374015348974459614815426412965499490000444007240765727130000165312076406545621180143571994015903343539244028212438966822248927862988084382716133376$$

### 11.33. \xintiiSquareRoot

```
\xintAssign\xintiiSquareRoot {17000000000000000000000000}\to\A\B
\xintiiSub{\xintiiSqr\A}\B=\A\string^2-\B
```

[illegible]

A rational approximation to  $\sqrt{N}$  is  $M - \frac{d}{2M}$  which is a majorant and the error is at most  $1/2M$  (if  $N$  is a perfect square  $k^2$  this gives  $k+1/(2k+2)$ , not  $k$ .)

Package `xintfrac` has `\xintFloatSqrt` for square roots of floating point numbers.

*source*

11.34. `\xintiiSqrt`, `\xintiiSqrtR`

*f* ★ `\xintiiSqrt{N}` computes the largest integer whose square is at most equal to `N`. `\xintiiSqrtR` produces the rounded, not truncated, square root.

```
\begin{itemize}[nosep]
\item \xintiiSqrt {300000000000000000000000000000000}
\item \xintiiSqrtR {300000000000000000000000000000000}
\item \xintiiSqrt {\xintiiE {3}{100}}
\end{itemize}
```

- 1732050807568877293
- 1732050807568877294
- 173205080756887729352744634150587236694280525381038

11.35. \xintiiBinomial

$\sum_x \sum_y \star$  `\xintiiBinomial{x}{y}` computes binomial coefficients.

If  $x < 0$  an out-of-range error is raised. Else, if  $y < 0$  or if  $x < y$  the macro evaluates to 0.



The allowable range is  $0 \leq x \leq 99999999$ . But this theoretical range includes binomial coefficients with more than the roughly 19950 digits that the arithmetics of *xint* can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

It turns out that  $\binom{65000}{32500}$  has 19565 digits and  $\binom{64000}{32000}$  has 19264 digits. The latter can be evaluated (this takes a long long time) but presumably not the former (I didn't try). Reasonable feasible evaluations are with binomial coefficients not exceeding about one thousand digits.

The *binomial* function is available in the *xintexpr* parsers.

```
\xinttheiexpr seq(binomial(100,i), i=47..53)\relax
84413487283064039501507937600, 93206558875049876949581681100, 98913082887808032681188722800,
100891344545564193334812497256, 98913082887808032681188722800, 93206558875049876949581681100,
84413487283064039501507937600
```

See *\xintFloatBinomial* from package *xintfrac* for the float variant, used in *\xintfloatexpr*.

In order to evaluate binomial coefficients  $\binom{x}{y}$  with  $x > 99999999$ , or even  $x \geq 2^{31}$ , but  $y$  is not too large, one may use an ad hoc function definition such as:

```
\xintdefunc mybigbinomial(x,y):=`*(x-y+1..[1]..x)//y!;%
%
% without [1], x would have been limited to < 2^31
\printnumber{\xinttheexpr mybigbinomial(98765432109876543210,10)\relax}
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710
27360575372715482389633359878460739973726786576925067784100587971261422326652270975592667517
4871960261
```

To get this functionality in macro form, one can do:

```
\xintNewIExpr\MyBigBinomial [2]{`*(#1-#2+1..[1]..#1)//#2!}
\printnumber{\MyBigBinomial {98765432109876543210}{10}}
24338098741940755592729533173058146177070669479669793038510211146784065843698581878582323710
27360575372715482389633359878460739973726786576925067784100587971261422326652270975592667517
4871960261
```

As we used *\xintNewIExpr*, this macro will only accept strict integers. Had we used *\xintNewExpr* the *\MyBigBinomial* would have accepted general fractions or decimal numbers, and computed the product at the numerator without truncating them to integers; but the factorial at the denominator would truncate its argument.

source

### 11.36. *\xintiiPFactorial*

$\frac{\text{num}}{X} \frac{\text{num}}{X} \star$  *\xintiiPFactorial*{a}{b} computes the partial factorial  $(a+1)(a+2)\dots b$ . For  $a=b$  the product is considered empty hence returns 1.

The allowed range is  $-100000000 \leq a, b \leq 99999999$ . The rule is to interpret the formula as the product of the  $j$ 's such that  $a < j \leq b$ , hence in particular if  $a \geq b$  the product is empty and the macro evaluates to 1.

Only for  $0 \leq a \leq b$  is the behaviour to be considered stable. For  $a > b$  or negative arguments, the definitive rules have not yet been fixed.

```
\xintiiPFactorial {100}{130}
69293021885203871012298422845822803287591970060789350400000000
```

This theoretical range allows computations whose result values would have more than the roughly 19950 digits that the arithmetics of *xint* can handle. In such cases, the computation will end up in a low-level TeX error after a long time.

The *pfactorial* function is available in the *xintexpr* parsers.

```
\xinttheiexpr pfactorial(100,130)\relax
69293021885203871012298422845822803287591970060789350400000000
```

See *\xintFloatPFactorial* from package *xintfrac* for the float variant, used in *\xintfloatexpr*.

In case values are needed with  $b > 99999999$ , or even  $b \geq 2^{31}$ , but  $b - a$  is not too large, one may use an ad hoc function definition such as:

```
\xintdefunc mybigpfac(a,b):=`*(a+1..[1]..b);%
%
% without [1], b would have been limited to < 2^31
```

```
\printnumber{\xinttheexpr mybigpfac(98765432100,98765432120)\relax}
78000855017567528067298107313023778438653002029049647467208196028116499434050587656870489322
99630604482236853566403912561449912587404607844104078121472675461815442734098676283450069933
322948600573016997034009566576640000
```

source

**11.37. \xintiiMax**

*ff* ★ `\xintiiMax{N}{M}` returns the largest of the two in the sense of the order structure on the relative integers (i.e. the right-most number if they are put on a line with positive numbers on the right):  
`\xintiiMax {-5}{-6}=-5.`

source

**11.38. \xintiiMin**

*ff* ★ `\xintiiMin{N}{M}` returns the smallest of the two in the sense of the order structure on the relative integers (i.e. the left-most number if they are put on a line with positive numbers on the right): `\xintiiMin {-5}{-6}=-6.`

source

**11.39. \xintiiMaxof**

*f* → \* *f* ★ `\xintiiMaxof{{a}{b}{c}...}` returns the maximum. The list argument may be a macro, it is *f*-expanded first.

source

**11.40. \xintiiMinof**

*f* → \* *f* ★ `\xintiiMinof{{a}{b}{c}...}` returns the minimum. The list argument may be a macro, it is *f*-expanded first.

source

**11.41. \xintifTrueAelseB**

*fnn* ★ `\xintifTrueAelseB{<f>}{<true branch>}{<false branch>}` is a synonym for `\xintiiifNotZero`.  
`\xintiiifnotzero` is lowercase companion macro.

Note 1: as it does only *f*-expansion on its argument it fails with inputs such as `--0`. But with *xintfrac* loaded, it does work fine if nested with other *xintfrac* macros, because the output format of such macros is fine as input to `\xintiiifNotZero`. This remark applies to all other “Boolean logic” macros next.

Note 2: prior to 1.20 this macro was using `\xintifNotZero` which applies `\xintNum` to its argument (or gets redefined by *xintfrac* to handle general decimal numbers or fractions). Hence it would have worked with input such as `--0`. But it was decided at 1.20 that the overhead was not worth it. The same remark applies to the other “Boolean logic” type macros next.

source

**11.42. \xintifFalseAelseB**

*fnn* ★ `\xintifFalseAelseB{<f>}{<false branch>}{<true branch>}` is a synonym for `\xintiiifZero`.  
`\xintiiifzero` is lowercase companion macro.

source

**11.43. \xintNOT**

*f* ★ `\xintNOT` is a synonym for `\xintiiIsZero`.  
`\xintiiiszero` serves as lowercase companion macro.

source

**11.44. \xintAND**

*ff* ★ `\xintAND{f}{g}` returns 1 if *f*!=0 and *g*!=0 and 0 otherwise.

*source***11.45. \xintOR***ff* ★ `\xintOR{f}{g}` returns 1 if `f!=0` or `g!=0` and 0 otherwise.*source***11.46. \xintXOR***ff* ★ `\xintXOR{f}{g}` returns 1 if exactly one of `f` or `g` is true (i.e. non-zero), else 0.*source***11.47. \xintANDof***f* → \**f* ★ `\xintANDof{{a}{b}{c}...}` returns 1 if all are true (i.e. non zero) and 0 otherwise. The list argument may be a macro, it (or rather its first token) is *f-expanded* first to deliver its items.*source***11.48. \xintORof***f* → \**f* ★ `\xintORof{{a}{b}{c}...}` returns 1 if at least one is true (i.e. does not vanish), else it produces 0. The list argument may be a macro, it is *f-expanded* first.*source***11.49. \xintXORof***f* → \**f* ★ `\xintXORof{{a}{b}{c}...}` returns 1 if an odd number of them are true (i.e. do not vanish), else it produces 0. The list argument may be a macro, it is *f-expanded* first.*source***11.50. \xintiiGCD***ff* ★ `\xintiiGCD{N}{M}` computes the greatest common divisor. It is positive, except when both `N` and `M` vanish, in which case the macro returns zero.`\xintiiGCD{10000}{1113}=1``\xintiiGCD{123456789012345}{9876543210321}=3`At 1.3d, this macro (which is used by the `gcd()` function in `\xintiiexpr`) was copied over to `xint`, thus removing a partial dependency of `xintexpr` on `xintgcd`.At 1.4 `xintgcd` requires `xint` and the latter is thus the one providing the macro.*source***11.51. \xintiiLCM***ff* ★ `\xintiiLCM{N}{M}` computes the least common multiple. It is positive, except if one of `N` or `M` vanish, in which case the macro returns zero.`\xintiiLCM{10000}{1113}=11130000``\xintiiLCM{123456789012345}{9876543210321}=406442103762636081733470915`At 1.3d, this macro (which is used by the `lcm()` function in `\xintiiexpr`) was copied over to `xint`, thus removing a partial dependency of `xintexpr` on `xintgcd`.At 1.4 `xintgcd` requires `xint` and the latter is thus the one providing the macro.*source***11.52. \xintiiGCDof***f* → \**f* ★ `\xintiiGCDof{{a}{b}{c}...}` computes the greatest common divisor of the integers `a`, `b`, .... It is a support macro for the `gcd()` function of the `\xintiiexpr` parser.It replaces the `\xintGCDof` which was formerly provided by `xintgcd` and is now available via `xintfrac` in a version handling also fractions.

source

### 11.53. `\xintiilCMof`

$f \rightarrow *f$  ★ `\xintiilCMof{{a}{b}{c}...}` computes the least common multiple of the integers `a`, `b`, .... It is a support macro for the `lcm()` function of the `\xintiexpr` parser.

It replaces the `\xintLCMof` which was formerly provided by `xintgcd` and is now available via `xintfrac` in a version handling also fractions.

source

### 11.54. `\xintLen`

Num  $f$  ★ `\xintLen` is originally an alias for `\xintiLen`. But with `xintfrac` loaded its meaning is modified to accept more general inputs.

source

### 11.55. (WIP) `\xintRandomDigits`

All randomness related macros are Work-In-Progress: implementation and user interface may change. They work only if the  $\text{\TeX}$  engine provides the `\uniformdeviate` or `\pdfuniformdeviate` primitive. See `\xintUniformDeviate` for additional information.

Num  $X$  ★ `\xintRandomDigits{N}` expands in two steps to `N` random decimal digits. The argument must be non-negative and is limited by  $\text{\TeX}$  memory parameters. On  $\text{\TeX}$ Live 2018 with input save stack size at 5000 the maximal allowed `N` is at most 19984 (tested within a `\write` to an auxiliary file, the macro context may cause a reduced maximum).

```
\pdfsetrandomseed 271828182
```

```
\xintRandomDigits{92}
```

```
60033782389146151207277993539344280578090871919638745398735577686436165769394958639376355806
```

**$\text{\TeX}$ -hackers note:** the digits are produced eight by eight by the same method which would result from `\xintUniformDeviate{100000000}` but with less overhead.

source

### 11.56. (WIP) `\xintXRandomDigits`

Num  $X$  ☆ `\xintXRandomDigits{N}` expands under exhaustive expansion (`\edef`, `\write`, `\csname` ...) to `N` random decimal digits. The argument must be non-negative. For example:

```
\newwrite\out
\immediate\openout\out=\jobname-out.txt
\immediate\write\out{\xintXRandomDigits{4500000}}
\immediate\closeout\out
```

creates a 4500001 bytes file (it ends with a line feed character). Trying with 5000000 raises this error:

```
Runaway text?
588875947168511582764514135070217555354479805240439407753451354223283\ETC.
! TeX capacity exceeded, sorry [main memory size=5000000].
<inserted text> 666515098
```

```
1.15 ...ate\write\out{\xintXRandomDigits{5000000}}
```

```
No pages of output.
Transcript written on temp.log.
```

This can be lifted by increasing the  $\text{\TeX}$  memory settings (installation dependent).

**$\text{\TeX}$ -hackers note:** the digits are produced eight by eight by the same method which would result from `\xintUniformDeviate{100000000}` but with less overhead.

*source***11.57. (WIP) \xintiiRandRange**

*f* ★ `\xintiiRandRange{A}` expands to a random (big) integer  $N$  such that  $0 \leq N < A$ . It is a supporting macro for `randrange()`. As with Python's function of the same name, it is an error if  $A \leq 0$ .

```
\pdfsetrandomseed 271828314
xx\newline
\xintiiRandRange{\xintNum{1e40}}\newline
\pdfsetrandomseed 271828314
\xinttheiexpr randrange(num(1e40))\relax\newline
% bare 1e40 not understood by \xintiepr
\pdfsetrandomseed 271828314
\xinttheexpr randrange(1e40)\relax
```

```
xx
1408107837990425263001878034077495278697
1408107837990425263001878034077495278697
1408107837990425263001878034077495278697
```

Of course, keeping in mind that the set of seeds is of cardinality  $2^{28}$ , randomness is a bit illusory here say with  $A=10^N$ ,  $N>8$ , if we proceed immediately after having set the seed. If we add some entropy in any way, then it is slightly more credible; but I think that for each seed the period is something like  $2^{27}(2^{55}-1)55$ ,<sup>69</sup> so we expect at most about  $2^{110}55$  'points in time', and this is already small compared to the  $10^{40}$  from example above. Thus already we are very far from being intrinsically able to generate all numbers with forty digits as random numbers, and this makes the previous section about usage of `\xintXRandomDigits` to generate millions of digits a bit comical...

**T<sub>E</sub>X-hackers note:** the digits are produced eight by eight by the same method which would result from `\xintUniformDeviate{100000000}` but with less overhead.

*source***11.58. (WIP) \xintiiRandRangeAtoB**

*ff* ★ `\xintiiRandRangeAtoB{A}{B}` expands to a random (big) integer  $N$  such that  $A \leq N < B$ . It is a supporting macro for `randrange()`. As with Python's function of the same name, it is an error if  $B \leq A$ .

```
\pdfsetrandomseed 271828314
12345678911111111111111111111111\newline
\xintiiRandRangeAtoB{12345678911111111111111111111111}{12345678922222222222222222222222}%
\newline
\pdfsetrandomseed 271828314
\def\test{%
\xinttheiexpr
    randrange(12345678911111111111111111111111,12345678922222222222222222222222)
\relax}%
\romannumeral\xintreplicate{10}{\test\newline}%
123456789222222222222222222222222222
```

```
12345678911111111111111111111111
12345678916037426188606389808
12345678916037426188606389808
12345678916060337223949101536
12345678912190033095886250034
12345678917323740152668511995
12345678915424847208552293485
12345678921595726610650510660
```

<sup>69</sup> Compare the result of exercise 3.2.2-30 in TAOCP, vol II.

## TOC

*TOC, xint bundle, xintkernel, xintcore, xint, xintfrac, xintbinhex, xintgcd, xintseries, xintcfrac*

12345678911673261982088192858  
12345678911339325803675947159  
12345678917791540296982027151  
12345678913602899909728811895  
1234567892222222222222222222

**T<sub>E</sub>X-hackers note:** the digits are produced eight by eight by the same method which would result from `\xint-UniformDeviate{100000000}` but with less overhead.

## 12. Macros of the **xintfrac** package

|     |                                  |     |      |                                          |     |
|-----|----------------------------------|-----|------|------------------------------------------|-----|
| .1  | <code>\xintTeXFromSci</code>     | 164 | .52  | <code>\xintifCmp</code>                  | 178 |
| .2  | <code>\xintTeXFrac</code>        | 165 | .53  | <code>\xintifEq</code>                   | 178 |
| .3  | <code>\xintTeXsignedFrac</code>  | 165 | .54  | <code>\xintifGt</code>                   | 178 |
| .4  | <code>\xintTeXOver</code>        | 165 | .55  | <code>\xintifLt</code>                   | 178 |
| .5  | <code>\xintTeXsignedOver</code>  | 166 | .56  | <code>\xintifInt</code>                  | 178 |
| .6  | <code>\xintLen</code>            | 166 | .57  | <code>\xintSgn</code>                    | 179 |
| .7  | <code>\xintNum</code>            | 166 | .58  | <code>\xintSignBit</code>                | 179 |
| .8  | <code>\xintRaw</code>            | 166 | .59  | <code>\xintOpp</code>                    | 179 |
| .9  | <code>\xintRawBraced</code>      | 166 | .60  | <code>\xintAbs</code>                    | 179 |
| .10 | <code>\xintNumerator</code>      | 167 | .61  | <code>\xintAdd</code>                    | 179 |
| .11 | <code>\xintDenominator</code>    | 167 | .62  | <code>\xintSub</code>                    | 179 |
| .12 | <code>\xintRawWithZeros</code>   | 167 | .63  | <code>\xintMul</code>                    | 179 |
| .13 | <code>\xintREZ</code>            | 167 | .64  | <code>\xintDiv</code>                    | 179 |
| .14 | <code>\xintIrr</code>            | 167 | .65  | <code>\xintDivFloor</code>               | 179 |
| .15 | <code>\xintPIrr</code>           | 168 | .66  | <code>\xintMod</code>                    | 179 |
| .16 | <code>\xintJrr</code>            | 168 | .67  | <code>\xintDivMod</code>                 | 180 |
| .17 | <code>\xintPRaw</code>           | 168 | .68  | <code>\xintDivTrunc</code>               | 180 |
| .18 | <code>\xintDecToStringREZ</code> | 168 | .69  | <code>\xintModTrunc</code>               | 180 |
| .19 | <code>\xintDecToString</code>    | 169 | .70  | <code>\xintDivRound</code>               | 180 |
| .20 | <code>\xintFracToSci</code>      | 170 | .71  | <code>\xintSqr</code>                    | 180 |
| .21 | <code>\xintFracToDecimal</code>  | 170 | .72  | <code>\xintPow</code>                    | 180 |
| .22 | <code>\xintTrunc</code>          | 171 | .73  | <code>\xintFac</code>                    | 181 |
| .23 | <code>\xintXTrunc</code>         | 172 | .74  | <code>\xintBinomial</code>               | 181 |
| .24 | <code>\xintTFrac</code>          | 174 | .75  | <code>\xintPFactorial</code>             | 181 |
| .25 | <code>\xintRound</code>          | 174 | .76  | <code>\xintMax</code>                    | 181 |
| .26 | <code>\xintFloor</code>          | 175 | .77  | <code>\xintMin</code>                    | 181 |
| .27 | <code>\xintCeil</code>           | 175 | .78  | <code>\xintMaxof</code>                  | 181 |
| .28 | <code>\xintiTrunc</code>         | 175 | .79  | <code>\xintMinof</code>                  | 181 |
| .29 | <code>\xintTTrunc</code>         | 176 | .80  | <code>\xintSum</code>                    | 182 |
| .30 | <code>\xintiRound</code>         | 176 | .81  | <code>\xintPrd</code>                    | 182 |
| .31 | <code>\xintiFloor</code>         | 176 | .82  | <code>\xintGCD</code>                    | 182 |
| .32 | <code>\xintiCeil</code>          | 176 | .83  | <code>\xintLCM</code>                    | 182 |
| .33 | <code>\xintE</code>              | 176 | .84  | <code>\xintGCDof</code>                  | 182 |
| .34 | <code>\xintCmp</code>            | 177 | .85  | <code>\xintLCMof</code>                  | 182 |
| .35 | <code>\xintEq</code>             | 177 | .86  | <code>\xintDigits, \xinttheDigits</code> | 183 |
| .36 | <code>\xintNotEq</code>          | 177 | .87  | <code>\xintSetDigits</code>              | 183 |
| .37 | <code>\xintGeq</code>            | 177 | .88  | <code>\xintFloat</code>                  | 183 |
| .38 | <code>\xintGt</code>             | 177 | .89  | <code>\xintFloatBraced</code>            | 185 |
| .39 | <code>\xintLt</code>             | 177 | .90  | <code>\xintFloatToDecimal</code>         | 185 |
| .40 | <code>\xintGtorEq</code>         | 177 | .91  | <code>\xintPFloat</code>                 | 186 |
| .41 | <code>\xintLtorEq</code>         | 177 | .92  | <code>\xintFloatAdd</code>               | 189 |
| .42 | <code>\xintIsZero</code>         | 177 | .93  | <code>\xintFloatSub</code>               | 190 |
| .43 | <code>\xintIsNotZero</code>      | 177 | .94  | <code>\xintFloatMul</code>               | 190 |
| .44 | <code>\xintIsOne</code>          | 177 | .95  | <code>\xintFloatDiv</code>               | 190 |
| .45 | <code>\xintOdd</code>            | 177 | .96  | <code>\xintFloatPow</code>               | 190 |
| .46 | <code>\xintEven</code>           | 177 | .97  | <code>\xintFloatPower</code>             | 190 |
| .47 | <code>\xintifSgn</code>          | 178 | .98  | <code>\xintFloatSqrt</code>              | 191 |
| .48 | <code>\xintifZero</code>         | 178 | .99  | <code>\xintFloatFac</code>               | 191 |
| .49 | <code>\xintifNotZero</code>      | 178 | .100 | <code>\xintFloatBinomial</code>          | 192 |
| .50 | <code>\xintifOne</code>          | 178 | .101 | <code>\xintFloatPFactorial</code>        | 192 |
| .51 | <code>\xintifOdd</code>          | 178 |      |                                          |     |



First version of this package was in release 1.03 (2013/04/14) of the *xint bundle*.

At release 1.3 (2018/02/28) the behaviour of `\xintAdd` (and of `\xintSub`) was modified: when adding  $a/b$  and  $c/d$  they will use always the least common multiple of the denominators. This helps limit the build-up of denominators, but the author still hesitates if the fraction should be reduced to smallest terms. The current method allows (for example when multiplying two polynomials) to keep a well-predictable denominator among various terms, even though some may be reducible.

`xintfrac` loads automatically `xintcore` and `xint` and inherits their macro definitions. Only these two are redefined: `\xintNum` and `\xintLen`. As explained in subsection 8.4 and subsection 8.5 the interchange format for the `xintfrac` macros, i.e.  $A/B[N]$ , is not understood by the `ii`-named macros of `xintcore/xint` which expect the so-called strict integer format. Hence, to use such an `ii`-macro with an output from an `xintfrac` macro, an extra `\xintNum` wrapper is required. But macros already defined by `xintfrac` cover most use cases hence this should be a rarely needed.

Frac  
f

In the macro descriptions, the variable `f` and the margin indicator stand for the `xintfrac` input format for integers, scientific numbers, and fractions as described in subsection 8.4.

num  
X

As in the `xint.sty` documentation, `x` stands for something which internally will be handled in a `\numexpr`. It may thus be an expression as understood by `\numexpr` but its evaluation and intermediate steps must obey the  $\text{T}_{\text{E}}\text{X}$  bound.

The output format for most macros is the  $A/B[N]$  format but naturally the float macros use the scientific notation on output. And some macros are special, for example `\xintTrunc` produces decimal numbers, `\xintIrr` produces an  $A/B$  with no  $[N]$ , `\xintiTrunc` and `\xintiRound` produce integers without trailing  $[N]$  either, etc. . .

At 1.4g, old legacy typesetting macros `\xintFrac`, `\xintSignedFrac`, `\xintFwOver` and `\xintSign`, `edFwOver` were renamed into `\xintTeXFrac`, `\xintTeXSignedFrac`, `\xintTeXOver`, `\xintTeXSignedOver`. The old names will raise errors and will be removed completely soon.

Changed  
at 1.4m!

[source](#)

## 12.1. `\xintTeXFromSci`

☆ Experimental. This typesetting math-mode-only macro expects an input which is already in, or will expand to, decimal or scientific notation. A trailing  $/B$  is accepted and will be handled differently according to whether it follows some scientific exponent  $eN$  part or not.

It was formerly `\xintTeXfromSci`. Old name deprecated at 1.4l. Also it used to be *f-expandable* but is now only *x-expandable*. Use `\expanded` if needed.

This macro can be used as a typesetting wrapper for `\xinteval` or `\xintfloateval` output: it expects its input (after expansion) to have been already “prettified” for example via the removal of trailing zeros, usage of fixed point notation if scientific exponent is small, etc. . . It simply transforms the `e<exponent>` part, if actually present, into `\cdot 10^{<exponent>}`. A fractional part  $/B$  if found in the expansion of the input must be last and will be transformed into `\cdot B^{-1}` if there was a scientific part, else the output will be using `\frac{A}{B}` (or the  $\text{T}_{\text{E}}\text{X}$  equivalent in place of `\frac`) with  $A$  the numerator.

**$\text{T}_{\text{E}}\text{X}$ -hackers note:**

- I am hesitating whether the `\frac{A}{B}` branch choice should require  $A$  to be an integer, or will also, as currently, be done with  $A$  being a number in decimal notation. Please advise.
- The package does:

```
\ifdefined\frac
\protected\def\xintTeXFromSci#1#2{\frac{#2}{#1}}%
\else
\protected\def\xintTeXFromSci#1#2{{#2\over#1}}%
\fi
```

Customize as desired. Notice the interversion of arguments.

Example:

```
\xintTeXFromSci{\xintfloateval{1.1^10000/5}}$,
\xintTeXFromSci{\xinteval{1.1^10000/5}}$\par
```

1.689980050240070 · 10<sup>413</sup>, 8.449900251200348 · 10<sup>413</sup> · 5<sup>-1</sup>

The above examples are in the case of a single numerical value. To handle more complex outputs from `\xinteval` or `\xintfloateval` one will need to proceed via a redefinition of `\xintfloatevalPrintOne` and/or `\xintexprPrintOne` like this:

```
\[def\xintfloatevalPrintOne[#1]#2{\xintTeXFromSci{\xintPFloat[#1]{#2}}}\xintfloateval[10]{2^100, 3^100, 13^100}\]
```

1.267650600 · 10<sup>30</sup>, 5.153775207 · 10<sup>47</sup>, 2.479335111 · 10<sup>111</sup>

```
\[def\xintexprPrintOne#1{\xintTeXFromSci{\xintFracToSci{#1}}}\xinteval{sqrt(2^101,60), 355/113, 6.02e23/1000}\]
```

1.59226291813144314115595358963043315049844681269444074447413 · 10<sup>15</sup>,  $\frac{355}{113}$ , 6.02 · 10<sup>23</sup> · 1000<sup>-1</sup>

This will however make then impossible, due to the added  $\TeX$  mark-up in the output, the nesting of `\xintfloateval`/`\xinteval` inside one another. The core `\xintexpr... \relax` syntax remains usable and is anyhow the recommended way for such nesting as it is more efficient.

Some similar effect is also possible without `\xintTeXFromSci`, simply by a customization of `\xintPFloat` like this:

```
\begingroup\def\xintPFloatE#1.{\cdot10^{#1}.}%\xintfloateval{1.1^10000/5}$, $\xinteval{1.1^10000/5}$\endgroup\newline
```

1.689980050240070 · 10<sup>413</sup>, 8.449900251200348 · 10<sup>413</sup>/5

This method is simpler-minded but will leave the trailing `/B`'s "as is", even if the numerator has no scientific exponent part. The presence of extra  $\TeX$  mark-up in the output has the consequences on nesting which were mentioned above.

*source*

## 12.2. `\xintTeXFrac`

$\frac{f}{f}$  ★ This is a typesetting  $\TeX$  only macro, math mode only as it expands to `\frac{A}{B}10^n` for an input which ends up parsed into raw format `A/B[n]`.

If the denominator `B` is 1, the output is `A\cdot 10^n`. If the exponent `n` is 0, the `[\cdot]10^n` part is omitted.

```
\xintTeXFrac {178.000/256000000}$, $\xintTeXFrac {178.000/1}$,\xintTeXFrac {3.5/5.7}$\newline
```

$\frac{178000}{256000000}10^{-3}$ ,  $178000 \cdot 10^{-3}$ ,  $\frac{35}{57}$

The input, if in a fraction form, is not simplified in any way, except for transforming numerator and denominator into integers, separating a power of ten part. Macros such as `\xintIrr`, `\xintPIrr`, `\xintREZ` can be inserted to wrap the input and help simplify it. The minus sign ends up in the numerator.

**Changed at 1.4m!** It is the new name since 1.4g of `\xintFrac`. The old name now raises a  $\TeX$  error.

*source*

## 12.3. `\xintTeXsignedFrac`

$\frac{f}{f}$  ★ This is as `\xintTeXFrac` except that a negative fraction has the sign ending up in front, not in the numerator.

```
\xintTeXsignedFrac {-355/113}=\xintTeXsignedFrac {-355/113}$\newline
```

$-\frac{355}{113} = -\frac{355}{113}$

**Changed at 1.4m!** It is the new name since 1.4g of `\xintSignedFrac`. The old name now raises a  $\TeX$  error.

*source*

## 12.4. `\xintTeXOver`

$\frac{f}{f}$  ★ This does the same as `\xintTeXFrac` except that the `\over` primitive is used for the fraction (in case the denominator is not one; and a pair of braces contains the `A\over B` part).

```
\xintTeXOver {178.000/25600000}$, \xintTeXOver {178.000/1}$,
\xintTeXOver {3.5/5.7}$\newline
```

Changed  
at 1.4m!

$$\frac{178000}{25600000} 10^{-3}, 178000 \cdot 10^{-3}, \frac{35}{57}$$

It is the new name since 1.4g of `\xintFwOver`. The old name now raises a  $\TeX$  error.

*source*

## 12.5. `\xintTeXsignedOver`

$\frac{f}{f}$  ★ This is as `\xintTeXOver` except that a negative fraction has the sign put in front, not in the numerator.

```
\xintTeXOver{-355/113}=\xintTeXsignedOver{-355/113}$\newline
```

$$\frac{-355}{113} = -\frac{355}{113}$$

Changed  
at 1.4m!

It is the new name since 1.4g of `\xintSignedFwOver`. The old name now raises a  $\TeX$  error.

*source*

## 12.6. `\xintLen`

$\frac{f}{f}$  ★ The `\xintLen` macro from *xint* is extended to accept a fraction on input: the length of `A/B[n]` is the length of `A` plus the length of `B` plus the absolute value of `n` and minus one (an integer input as `N` is internally represented in a form equivalent to `N/1[0]` so the minus one means that the extended `\xintLen` behaves the same as the original for integers).

```
\xintLen{201710/298219}=\xintLen{201710}+\xintLen{298219}-1\newline
\xintLen{1234/1}=\xintLen{1234}=\xintLen{1234[0]}=\xintiLen{1234}\newline
\xintLen{-1e3/5.425} (\xintRaw {-1e3/5.425})\par
```

11=6+6-1

4=4=4=4

10 (-1/5425[6])

The length is computed on the `A/B[n]` which would have been returned by `\xintRaw`, as illustrated by the last example above.

`\xintLen` is only for use with such (scientific) numbers or fractions. See also `\xintNthElt` from *xinttools*. See also `\xintLength` (which however does not expand its argument) from *xintkernel* for counting more general tokens (or rather braced items).

*source*

## 12.7. `\xintNum`

$\frac{f}{f}$  ★ The `\xintNum` from *xint* is transformed into a synonym to `\xintTTrunc`.

Attention that for example `\xintNum{1e100000}` expands to the needed 100001 digits...

The original `\xintNum` from *xintcore* which does not understand the fraction slash or the scientific notation is still available under the name `\xintiNum`.

*source*

## 12.8. `\xintRaw`

$\frac{f}{f}$  ★ This macro 'prints' the fraction `f` as it is received by the package after its parsing and expansion, in a form `A/B[N]` equivalent to the internal representation: the denominator `B` is always strictly positive and is printed even if it has value 1.

```
\xintRaw{\the\numexpr 571*987\relax.123e-10/\the\numexpr-201+59\relax e-7}
```

-563577123/142[-6]

No simplification is done, not even of common zeroes between numerator and denominator:

```
\xintRaw {178000/25600000}
```

178000/25600000[0]

*source*

## 12.9. `\xintRawBraced`

$\frac{f}{f}$  ★ This macro expands and parses its input `f` as all *xintfrac* macros and produces as output `{N}{A}{B}` (with  $\TeX$  braces) where `\xintRaw` would have returned `A/B[N]`.

## 12.10. \xintNumerator

```
\xintNumerator {178000/2560000[17]}\newline
\xintNumerator {312.289001/20198.27}\newline
\xintNumerator {178000e-3/256e5}\newline
\xintNumerator {178.000/2560000}
```

## 12.11. \xintDenominator

```
\xintDenominator {178000/25600000[17]}\newline
\xintDenominator {312.289001/20198.27}\newline
\xintDenominator {178000e-3/256e5}\newline
\xintDenominator {178.000/25600000}
```

## 12.12. \xintRawWithZeros

```
\xintRowWithZeros{178000/25600000[17]}\newline
\xintRowWithZeros{312.289001/20198.27}\newline
\xintRowWithZeros{178000e-3/256e5}\newline
\xintRowWithZeros{178.000/25600000}\newline
\xintRowWithZeros{\the\numexpr 571*987\relax.123e-10/\the\numexpr -201+59\relax e-7}
```

### 12.13. \xintREZ

```
\xintREZ {178000/25600000[17]}
178/256[15]
```

## 12.14. \xintIrr

```
\xintIrr {178.256/256.1780}, \xintIrr {178000/25600000[17]}
```

6856/9853, 695312500000000/1

The current implementation does not cleverly first factor powers of 2 and 5, and `\xintIrr {2/3 [100]}` will execute the Euclidean division of  $2 \cdot 10^{100}$  by 3, which is a bit stupid as it could have known that the 100 trailing zeros can not bring any divisibility by 3.

Starting with release 1.08, `\xintIrr` does not remove the trailing /1 when the output is an integer. This was deemed better for various (questionable?) reasons, anyway the output format is since always A/B with B>0, even in cases where it turns out that B=1. Use `\xintPRaw` on top of `\xintIrr` if it is needed to get rid of such a trailing /1.

[source](#)

## 12.15. `\xintPIrr`

Frac f ★ This puts the fraction into irreducible form, *keeping as is the decimal part [N]* from raw internal A/B[N] format. (P stands here for *Partial*)

```
\xintPIrr {178.256/256.1780}, \xintPIrr {178000/25600000[17]}
3428/49265[1], 89/12800[17]
```

Notice that the output always has the ending [N], which is exactly the opposite of `\xintIrr`'s behaviour. The interest of this macro is mainly in handling fractions which somehow acquired a big [N] (perhaps from input in scientific notation) and for which the reduced fraction would have a very large number of digits. This large number of digits can considerably slow-down computations done afterwards.

For example package `polexpr` uses `\xintPIrr` when differentiating a polynomial, or in setting up a Sturm chain for localization of the real roots of a polynomial. This is relevant to polynomials whose coefficients were input in decimal notation, as this automatically creates internally some [N]. Keeping and combining those [N]'s during computations significantly increases their speed.

[source](#)

## 12.16. `\xintJrr`

Frac f ★ This also puts the fraction into its unique irreducible form:

```
\xintJrr {178.256/256.178}
6856/9853
```

This is (supposedly, not tested for ages) faster than `\xintIrr` for fractions having some big common factor in the numerator and the denominator.

```
\xintJrr {\xintiiPow{\xintiiFac {15}}{3}/%
\xintiiPrd{\xintiiFac{10}}{\xintiiFac{30}}{\xintiiFac{5}}}}
1001/51705840
```

But to notice the difference one would need computations with much bigger numbers than in this example. As `\xintIrr`, `\xintJrr` does not remove the trailing /1 from a fraction reduced to an integer.

[source](#)

## 12.17. `\xintPRaw`

Frac f ★ `PRaw` stands for ``pretty raw''. It does like `\xintRaw` apart from removing the [N] part if N=0 and removing the B if B=1.

```
\xintPRaw {123e10/321e10}, \xintPRaw {123e9/321e10}, \xintPRaw {\xintIrr{861/123}}
123/321, 123/321[-1], 7
```

[source](#)

## 12.18. `\xintDecToStringREZ`

Frac f ★ `\xintDecToStringREZ` uses fixed point (decimal) notation for the output. The REZ means that it trims (REmoves) trailing Zeros. The name is a bit strange, because it is not limited to *decimal numbers* but accepts the same kind of inputs as most other `xintfrac` macros. The parsing of this input transforms it first into an internal format having a numerator A, a denominator B and a

power of ten exponent **N**, standing for the fraction **A/B** times 10 to the power **N**. Then the following recipe applies:

- the zero value is printed as 0 (no decimal point).
- trailing zeros of **A** and **B** are removed and **N** is adjusted,
- if the new **B** is not 1, it will appear in the output as /**B**,
- fixed point decimal notation is used for **AeN**:
  - if **N** is non-negative, the output is an integer with **N** trailing zeros (and no decimal mark)
  - if **N** is negative a decimal point is used, and if **AeN** is less than one in absolute value, output will start with 0. (i.e. a decimal mark).

The following should be noted:

1. the fraction **AeN/B** or even **A/B** is not pre-reduced into lowest terms,
2. the macro does not check if **B** contains only powers of 2 and 5, so 1/2 is printed as 1/2, not as 0.5.

The definitive behaviour remains to be decided regarding these last two points.

```
\xintDecToStringREZ{0}, \xintDecToStringREZ{1/2}, \xintDecToStringREZ{0.5000}\newline
\xintDecToStringREZ{1.23456789e5}, \xintDecToStringREZ {1.23456789e-3}\newline
\xintDecToStringREZ{12345e-1}, \xintDecToStringREZ {12345e-2},
\xintDecToStringREZ{12345e-3}\newline
\xintDecToStringREZ{12345e-4}, \xintDecToStringREZ {12345e-5},
\xintDecToStringREZ{12345e-6}\newline
\xintDecToStringREZ{1.234567890000e12}, \xintDecToStringREZ{1.23456000e-5/10}\newline
\xintDecToStringREZ{70/14} % is not reduced to lowest terms
```

```
0, 1/2, 0.5
123456.789, 0.00123456789
1234.5, 123.45, 12.345
1.2345, 0.12345, 0.012345
1234567890000, 0.00000123456
70/14
```

See `\xintFloatToDecimal` for a variant which first rounds the input to some given number of significant digits.

*source*

## 12.19. `\xintDecToString`

Frac f ★ `\xintDecToString` uses fixed point notation for the output. Its behaviour remains somewhat undecided in so far as whether it should identify inputs which correspond to decimal numbers (i.e. fractions with only powers of two and five in their denominator, once reduced to lowest terms).

As with `\xintDecToStringREZ`, the name is a bit strange as inputs are in no way limited to decimal numbers but are of the most general type accepted by the `xintfrac` macros.

It is the same macro as `\xintDecToStringREZ` except that it does not remove trailing zeros, in fact `\xintDecToStringREZ{f}` is defined as `\xintDecToString{\xintREZ{f}}`.

```
\xintDecToString{0}, \xintDecToString{1/2}, \xintDecToString{0.5000}\newline
\xintDecToString{1.23456789e5}, \xintDecToString {1.23456789e-3}\newline
\xintDecToString{12345e-1}, \xintDecToString {12345e-2}, \xintDecToString{12345e-3}%
\newline
\xintDecToString{12345e-4}, \xintDecToString {12345e-5}, \xintDecToString{12345e-6}%
\newline
\xintDecToString{1.234567890000e12}, \xintDecToString{1.23456000e-5/10}\newline
\xintDecToString{70/14}
```

```
0, 1/2, 0.5000
123456.789, 0.00123456789
```

1234.5, 123.45, 12.345  
 1.2345, 0.12345, 0.012345  
 1234567890000, 0.0000123456000/10  
 70/14

Since 1.4e, `\xintDecToString` is the default for `\xintiexprPrintOne`, which governs the `\xintieval` output format: in this use case there is never a `/B` fractional part and the output is always either an integer (if `\xintieval` was used without optional argument) or a decimal string

```
\def\xintiexprPrintOne{\xintDecToString}
```

Any replacement of `\xintDecToString` as the expansion of `\xintiexprPrintOne` should obey the following blueprint:

- ☆ • to be expandable, but not necessarily *f-expandable*,
- to accept on input `A` or `A[N]`.

*source*

## 12.20. `\xintFracToSci`

Frac  
f ★ `\xintFracToSci` was initially at 1.4 a private macro which served as default customization of `\xintexprPrintOne` and, despite being documented in the user manual, was not supposed to be used at user level (not being *f-expandable* it could not be nested within macros of `xintfrac`, and besides accepted a limited range of inputs).

It has been upgraded at 1.41 to behave like all other `xintfrac` macros.

Here is what it does:

- it first parses the input like any other `xintfrac` macro and convert it into the ``raw' A/B[N]` format,
- it then produces this output: `A/B` if `N=0` (and `/B` is omitted if not `1`), and for `N` not zero, the output numerator will be `AeN` written in scientific notation exactly like it would by `\xintPFloat` but without of course prior rounding to a given number of digits; the trailing zeros in the significand will be removed always (the `\xintPFloatMinTrimmed` configuration is ignored). Then this value in scientific notation (or in decimal fixed point notation if the scientific exponent is in the `\xintPFloatNoSciEmin` to `\xintPFloatNoSciEmax` range) will be attached to a trailing denominator `/B` (omitted if it is `/1`).

Please note:

- there is no reduction of the fraction `A/B` to lowest terms,
- trailing zeros in the integer denominator `B` are not moved and incorporated into the final scientific exponent,
- no attempt is made to check if `B` is a product of only `2`'s and `5`'s and thus could be integrated into some pure decimal notation for the numerator or at least its significand.

Changes of `\xintPFloat` at 1.4k have an impact here. In particular the zero value will give `0` whether the input was some `0`, `0e-5`, `0/3`, `0.00`, etc. . . , whereas at 1.4e it would have been `0.0` for cases triggering some `\xintPFloat` subroutine.

The general blueprint is still to be considered *unstable*.

The output routine of `\xinteval` is customizable via redefining `\xintexprPrintOne` whose current default is (equivalent to):

```
\def\xintexprPrintOne{\xintFracToSci}
```

*source*

## 12.21. `\xintFracToDecimal`

Frac  
f ★ `\xintFracToDecimal` is a variant of `\xintFracToSci` which differs from it in so far as it outputs a numerator using decimal notation, i.e. with as many zeros as are needed (and no more) and no scientific exponent. The denominator goes through ``as is'` except if it is `1`, then it is omitted.



In other terms its behaviour is currently intermediate between `\xintDecToString` and `\xintDecToStringREZ`, as it does not remove trailing zeros of the denominator. Consider its behaviour as *unstable*.

It can be used to customize `\xintexprPrintOne`:

```
\def\xintexprPrintOne{\xintFracToDecimal}
```

It was initially at 1.4k a private macro which served as an alternative to `\xintFracToSci` default customization of `\xintexprPrintOne` and, despite being documented in the user manual, was not supposed to be used at user level (not being *f-expandable* it could not be nested within macros of `xintfrac`, and besides accepted a limited range of inputs).

It has been upgraded at 1.41 to behave like all other `xintfrac` macros.

*source*

## 12.22. `\xintTrunc`

$\frac{\text{num}}{x} \frac{\text{Frac}}{f}$  ★ `\xintTrunc{x}{f}` returns the start of the decimal expansion of the fraction `f`, truncated to:

- if `x>0`, `x` digits after the decimal mark,
- if `x=0`, an integer,
- if `x<0`, an integer multiple of  $10^{-x}$  (in scientific notation).

The output is the sole digit token `0` if and only if the input was exactly zero; else it contains always either a decimal mark (even if `x=0`) or a scientific part and it conserves the sign of `f` (even if the truncated value represents the zero value).

Truncation is done towards zero.

```
\begin{multicols}{2}
  \noindent\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintTrunc{#1}{-11e12/7}\newline}%
  \xintTrunc{10}{1e-11}\newline \xintTrunc{10}{1/65536}\par
\end{multicols}
```

|                        |              |
|------------------------|--------------|
| -1571428571428.5714285 | -15714285e5  |
| -1571428571428.571428  | -1571428e6   |
| -1571428571428.57142   | -157142e7    |
| -1571428571428.5714    | -15714e8     |
| -1571428571428.571     | -1571e9      |
| -1571428571428.57      | -157e10      |
| -1571428571428.5       | -15e11       |
| -1571428571428.        | -1e12        |
| -157142857142e1        | -0e13        |
| -15714285714e2         | -0e14        |
| -1571428571e3          | 0.0000000000 |
| -157142857e4           | 0.0000152587 |

**Warning:** *it is not yet decided is the current behaviour is definitive.*

Currently `xintfrac` has no notion of a positive zero or a negative zero. Hence transitivity of `\xintTrunc` is broken for the case where the first truncation gives on output `0.00...0` or `-0.00...0`: a second truncation to less digits will then output `0`, whereas if it had been applied directly to the initial input it would have produced `0.00...0` or respectively `-0.00...0` (with less zeros after decimal mark).

If `xintfrac` distinguished zero, positive zero, and negative zero then it would be possible to maintain transitivity.

The problem would also be fixed, even without distinguishing a negative zero on input, if `\xintTrunc` always produced `0.00...0` (with no sign) when the mathematical result is zero, discarding the information on original input being positive, zero, or negative.

I have multiple times hesitated about what to do and must postpone again final decision.

*source*

## 12.23. `\xintXTrunc`

$\frac{\text{num}}{\text{x}}$   $\frac{\text{Frac}}{f}$  ☆

`\xintXTrunc{x}{f}` is similar to `\xintTrunc` with the following important differences:

- it is completely expandable but not *f-expandable*, as is indicated by the hollow star in the margin,
- hence it can not be used as argument to the other package macros, but as it *f-expands* its `{f}` argument, it accepts arguments expressed with other *xintfrac* macros,
- it requires `x>0`,
- contrarily to `\xintTrunc` the number of digits on output is not limited to about 19950 and may go well beyond 100000 (this is mainly useful for outputting a decimal expansion to a file),
- when the mathematical result is zero, it always prints it as `0.00...0` or `-0.00...0` with `x` zeros after the decimal mark.

**Warning:** transitivity is broken too (see discussion of `\xintTrunc`), due to the sign in the last item. Hence *the definitive policy is yet to be fixed*.

Transitivity is here in the sense of using a first `\edef` and then a second one, because it is not possible to nest `\xintXTrunc` directly as argument to itself. Besides, although the number of digits on output isn't limited, nevertheless `x` should be less than about 19970 when the number of digits of the input (assuming it is expressed as a decimal number) is even bigger: `\xintXTrunc{30000}{Z}` after `\edef Z{\xintXTrunc{60000}{1/66049}}` raises an error in contrast with a direct `\xintXTrunc{30000}{1/66049}`. But `\xintXTrunc{30000}{123.456789}` works, because here the number of digits originally present is smaller than what is asked for, thus the routine only has to add trailing zeros, and this has no limitation (apart from  $\text{\TeX}$  main memory).

`\xintXTrunc` will expand fully in an `\edef` or a `\write (\message, \wlog, ...)` or in an `\xint-expression`, or as list argument to `\xintFor*`.

Here is an example session where the user checks that the decimal expansion of  $1/66049 = 1/257^2$  has the maximal period length  $257 * 256 = 65792$  (this period length must be a divisor of  $\phi(66049)$  and to check it is the maximal one it is enough to show that neither 32896 nor 256 are periods.)

```
$ rlwrap etex -jobname worksheet-66049
This is pdfTeX, Version 3.14159265-2.6-1.40.17 (TeX Live 2016) (preloaded format=etex)
restricted \write18 enabled.
**xintfrac.sty
entering extended mode
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintfrac.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xint.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintcore.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xintkernel.sty)))
% we load xinttools for \xintKeep, etc... \xintXTrunc itself has no more

% any dependency on xinttools.sty since 1.2i

*\input xinttools.sty
(/usr/local/texlive/2016/texmf-dist/tex/generic/xint/xinttools.sty)
*\def\m#1;{\message{#1}}

*\m \the\numexpr 257*257\relax;
66049
*\m \the\numexpr 257*256\relax;
65792
```

```

*% Thus 1/66049 will have a period length dividing 65792.

*% Let us first check it is indeed periodical.

*\edef\Z{\xintXTrunc{66000}{1/66049}}

*% Let's display the first decimal digits.

*\m \xintXTrunc{208}{\Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423
*% let's now fetch the trailing digits

*\m \xintKeep{65792-66000}{\Z};% 208 trailing digits

0000151402746445820527184363124347075655952398976517434026253236233705279413768
5657617829187421459825281230601523111629244954503474693030931581098881133703765
38630410755651107511090251177156353616254598858423
*% yes they match! we now check that 65792/2 and 65792/257=256 aren't periods.

*\m \xintXTrunc{256}{\Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
554118911717058547442
*\m \xintXTrunc{256+256}{\Z};

0.00001514027464458205271843631243470756559523989765174340262532362337052794137
6856576178291874214598252812306015231116292449545034746930309315810988811337037
6538630410755651107511090251177156353616254598858423291798513225029902042423049
5541189117170585474420505987978621932201850141561567926842192917379521264515738
3154930430438008145467758785144362518736089872670290239064936637950612424109373
3440324607488379839210283274538600130206361943405653378552286938485064119063119
8049932625777831609865402958409665551333
*% now with 65792/2=32896. Problem: we can't do \xintXTrunc{32896+100}{\Z}

*% but only direct \xintXTrunc{32896+100}{1/66049}. Anyway we want to nest it

*% hence let's do it all with (slower) \xintKeep, \xintKeepUnbraced.

*\m \xintKeep {-100}{\xintKeepUnbraced{2+65792/2+100}{\Z}};

9999848597253554179472815636875652924344047601023482565973746763766294720586231
434238217081257854017
*% This confirms 32896 isn't a period length.

*% To conclude let's write the 66000 digits to the log.

*\wlog{\Z}

*% We want always more digits:

*\wlog{\xintXTrunc{150000}{1/66049}}

```

\*\bye

The acute observer will have noticed that there is something funny when one compares the first digits with those after the middle-period:

```
0000151402746445820527184363124347075655952398976517434026253236233705279413768...
9999848597253554179472815636875652924344047601023482565973746763766294720586231...
```

Mathematical exercise: can you explain why the two indeed add to 9999...9999?

You can try your hands at this simpler one:

```
1/49=\xintTrunc{42+5}{1/49}...\newline
\xintTrim{2}{\xintTrunc{21}{1/49}}\newline
\xintKeep{-21}{\xintTrunc{42}{1/49}}
```

```
1/49=0.02040816326530612244897959183673469387755102040...
020408163265306122448
979591836734693877551
```

This was again an example of the type  $1/N$  with  $N$  the square of a prime. One can also find counter-examples within this class:  $1/31^2$  and  $1/37^2$  have an odd period length (465 and respectively 111) hence they can not exhibit the symmetry.

Mathematical challenge: prove generally that if the period length of the decimal expansion of  $1/p^r$  (with  $p$  a prime distinct from 2 and 5 and  $r$  a positive exponent) is even, then the previously observed symmetry about the two halves of the period adding to a string of nine's applies.

[source](#)

## 12.24. \xintTFrac

Frac  
f ★

`\xintTFrac{f}` returns the fractional part,  $f = \text{trunc}(f) + \text{frac}(f)$ . Thus if  $f < 0$ , then  $-1 < \text{frac}(f) \leq 0$  and if  $f > 0$  one has  $0 \leq \text{frac}(f) < 1$ . The **T** stands for 'Trunc', and there should exist also similar macros associated respectively with 'Round', 'Floor', and 'Ceil', each type of rounding to an integer deserving arguably to be associated with a fractional 'modulo'. By sheer laziness, the package currently implements only the 'modulo' associated with 'Truncation'. Other types of modulo may be obtained more clumsily via a combination of the rounding with a subsequent subtraction from  $f$ .

Notice that the result is filtered through `\xintREZ`, and will thus be of the form  $A/B[N]$ , where neither  $A$  nor  $B$  has trailing zeros. But the output fraction is not reduced to smallest terms.

The function call in expressions (`\xintexpr`, `\xintfloatexpr`) is `frac`. Inside `\xintexpr..relax`,  $x$ , the function `frac` is mapped to `\xintTFrac`. Inside `\xintfloatexpr..relax`, `frac` first applies `\xintTFrac` to its argument (which may be an exact fraction with more digits than the floating point precision) and only in a second stage makes the conversion to a floating point number with the precision as set by `\xintDigits` (default is 16).

```
\xintTFrac {1235/97}, \xintTFrac {-1235/97}\newline
\xintTFrac {1235.973}, \xintTFrac {-1235.973}\newline
\xintTFrac {1.122435727e5}\par
```

```
71/97[0], -71/97[0]
973/1[-3], -973/1[-3]
5727/1[-4]
```

[source](#)

## 12.25. \xintRound

num  
x Frac  
f ★

`\xintRound{x}{f}` returns the start of the decimal expansion of the fraction  $f$ , rounded to:

- if  $x > 0$ ,  $x$  digits after the decimal mark,
- if  $x = 0$ , an integer,

- if  $x < 0$ , an integer multiple of  $10^{-x}$  (in scientific notation).

The output is the sole digit token 0 if and only if the input was exactly zero; else it contains always either a decimal mark (even if  $x=0$ ) or a scientific part and it conserves the sign of  $f$  (even if the rounded value represents the zero value).

```
\begin{multicols}{2}
\noindent\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintRound{#1}{-11e12/7}\newline}%
\xintRound{10}{1e-11}\newline \xintRound{10}{1/65536}\newline
\end{multicols}
```

|                        |              |
|------------------------|--------------|
| -1571428571428.5714286 | -1571429e6   |
| -1571428571428.571429  | -157143e7    |
| -1571428571428.57143   | -15714e8     |
| -1571428571428.5714    | -1571e9      |
| -1571428571428.571     | -157e10      |
| -1571428571428.57      | -16e11       |
| -1571428571428.6       | -2e12        |
| -1571428571429.        | -0e13        |
| -157142857143e1        | -0e14        |
| -15714285714e2         | 0.0000000000 |
| -1571428571e3          | 0.0000152588 |
| -157142857e4           |              |
| -15714286e5            |              |

Rounding is done with half-way numbers going towards infinity of the same sign.

*source*

## 12.26. `\xintFloor`

$\frac{\text{Frac}}{f}$  ★

`\xintFloor {f}` returns the largest relative integer  $N$  with  $N \leq f$ .

```
\xintFloor {-2.13}, \xintFloor {-2}, \xintFloor {2.13}
```

-3/1[0], -2/1[0], 2/1[0] Note the trailing [0], see `\xintiFloor` if it is not desired.

*source*

## 12.27. `\xintCeil`

$\frac{\text{Frac}}{f}$  ★

`\xintCeil {f}` returns the smallest relative integer  $N$  with  $N > f$ .

```
\xintCeil {-2.13}, \xintCeil {-2}, \xintCeil {2.13}
```

-2/1[0], -2/1[0], 3/1[0]

*source*

## 12.28. `\xintiTrunc`

$\frac{\text{num}}{x} \frac{\text{Frac}}{f}$  ★

`\xintiTrunc{x}{f}` returns the integer equal to  $10^x$  times what `\xintTrunc{x}{f}` would produce. Attention that leading zeros are automatically removed: the output is in strict integer format.

```
\begin{multicols}{2}
\noindent\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintiTrunc{#1}{-11e12/7}\newline}%
\xintiTrunc{10}{1e-11}\newline \xintiTrunc{10}{1/65536}\par
\end{multicols}
```

|                       |                 |
|-----------------------|-----------------|
| -15714285714285714285 | -15714285714285 |
| -1571428571428571428  | -1571428571428  |
| -157142857142857142   | -157142857142   |
| -15714285714285714    | -15714285714    |
| -1571428571428571     | -1571428571     |
| -157142857142857      | -157142857      |

|           |        |
|-----------|--------|
| -15714285 | -15    |
| -1571428  | -1     |
| -157142   | 0      |
| -15714    | 0      |
| -1571     | 0      |
| -157      | 152587 |

*source***12.29. \xintTTrunc**

$\frac{f}{f}$  ★ `\xintTTrunc{f}` truncates to an integer (truncation towards zero). This is the same as `\xintiTrunc{f}` and also the same as `\xintNum`.

*source***12.30. \xintiRound**

$\frac{\text{num}}{x} \frac{f}{f}$  ★ `\xintiRound{x}{f}` returns the integer equal to  $10^x$  times what `\xintRound{x}{f}` would return. The output has no leading zeroes, it is always in strict integer format.

```
\begin{multicols}{2}
  \noindent\xintFor* #1 in {\xintSeq[-1]{7}{-14}}:{\xintiRound{#1}{-11e12/7}\newline}%
  \xintiRound{10}{1e-11}\newline \xintiRound{10}{1/65536}\par
\end{multicols}
```

|                       |           |
|-----------------------|-----------|
| -15714285714285714286 | -15714286 |
| -1571428571428571429  | -1571429  |
| -157142857142857143   | -157143   |
| -15714285714285714    | -15714    |
| -1571428571428571     | -1571     |
| -157142857142857      | -157      |
| -15714285714286       | -16       |
| -1571428571429        | -2        |
| -157142857143         | 0         |
| -15714285714          | 0         |
| -1571428571           | 0         |
| -157142857            | 152588    |

*source***12.31. \xintiFloor**

$\frac{f}{f}$  ★ `\xintiFloor {f}` does the same as `\xintFloor` but without the trailing `/1[0]`.

```
\xintiFloor {-2.13}, \xintiFloor {-2}, \xintiFloor {2.13}
-3, -2, 2
```

*source***12.32. \xintiCeil**

$\frac{f}{f}$  ★ `\xintiCeil {f}` does the same as `\xintCeil` but its output is without the `/1[0]`.

```
\xintiCeil {-2.13}, \xintiCeil {-2}, \xintiCeil {2.13}
-2, -2, 3
```

*source***12.33. \xintE**

$\frac{f}{f} \frac{\text{num}}{x}$  ★ `\xintE {f}{x}` multiplies the fraction `f` by  $10^x$ . The second argument `x` must obey the  $\text{T}_{\text{E}}\text{X}$  bounds. Example:

```
\count 255 123456789 \xintE {10}{\count 255}
10/1[123456789] Don't feed this example to \xintNum!
```

[source](#)

## 12.34. `\xintCmp`

$\frac{F}{f} \frac{F}{f}$  ★ This compares two fractions  $F$  and  $G$  and produces  $-1$ ,  $0$ , or  $1$  according to  $F < G$ ,  $F = G$ ,  $F > G$ .  
For choosing branches according to the result of comparing  $f$  and  $g$ , see `\xintifCmp`.

[source](#)

## 12.35. `\xintEq`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintEq{f}{g}` returns  $1$  if  $f = g$ ,  $0$  otherwise.

[source](#)

## 12.36. `\xintNotEq`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintNotEq{f}{g}` returns  $0$  if  $f = g$ ,  $1$  otherwise.

[source](#)

## 12.37. `\xintGeq`

$\frac{F}{f} \frac{F}{f}$  ★ This compares the *absolute values* of two fractions. `\xintGeq{f}{g}` outputs  $1$  if  $|f| \geq |g|$  and  $0$  if not.  
Important: the macro compares *absolute values*.

[source](#)

## 12.38. `\xintGt`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintGt{f}{g}` returns  $1$  if  $f > g$ ,  $0$  otherwise.

[source](#)

## 12.39. `\xintLt`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintLt{f}{g}` returns  $1$  if  $f < g$ ,  $0$  otherwise.

[source](#)

## 12.40. `\xintGtorEq`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintGtorEq{f}{g}` returns  $1$  if  $f \geq g$ ,  $0$  otherwise. Extended by `xintfrac` to fractions.

[source](#)

## 12.41. `\xintLtorEq`

$\frac{F}{f} \frac{F}{f}$  ★ `\xintLtorEq{f}{g}` returns  $1$  if  $f \leq g$ ,  $0$  otherwise.

[source](#)

## 12.42. `\xintIsZero`

$f$  ★ `\xintIsZero{f}` returns  $1$  if  $f = 0$ ,  $0$  otherwise.

[source](#)

## 12.43. `\xintIsNotZero`

$f$  ★ `\xintIsNotZero{f}` returns  $1$  if  $f \neq 0$ ,  $0$  otherwise.

[source](#)

## 12.44. `\xintIsOne`

$f$  ★ `\xintIsOne{f}` returns  $1$  if  $f = 1$ ,  $0$  otherwise.

[source](#)

## 12.45. `\xintOdd`

$f$  ★ `\xintOdd{f}` returns  $1$  if the integer obtained by truncation is odd, and  $0$  otherwise.

[source](#)

## 12.46. `\xintEven`

$f$  ★ `\xintEven{f}` returns  $1$  if the integer obtained by truncation is even, and  $0$  otherwise.



*source***12.47. \xintifSgn***source*

$\frac{f}{f} nnn \star$  `\xintifSgn{⟨f⟩}{⟨A⟩}{⟨B⟩}{⟨C⟩}` executes either the `⟨A⟩`, `⟨B⟩` or `⟨C⟩` code, depending on its first argument being respectively negative, zero, or positive.

*source***12.48. \xintifZero** $\frac{f}{f}$ 

$\frac{f}{f} nn \star$  `\xintifZero{⟨f⟩}{⟨IsZero⟩}{⟨IsNotZero⟩}` expandably checks if the first mandatory argument `N` (a number, possibly a fraction if `xintfrac` is loaded, or a macro expanding to one such) is zero or not. It then either executes the first or the second branch.

Beware that both branches must be present.

*source***12.49. \xintifNotZero** $\frac{f}{f}$ 

$\frac{f}{f} nn \star$  `\xintifNotZero{⟨N⟩}{⟨IsNotZero⟩}{⟨IsZero⟩}` expandably checks if the first mandatory argument `f` is not zero or is zero. It then either executes the first or the second branch.

Beware that both branches must be present.

*source***12.50. \xintifOne** $\frac{f}{f}$ 

$\frac{f}{f} nn \star$  `\xintifOne{⟨N⟩}{⟨IsOne⟩}{⟨IsNotOne⟩}` expandably checks if the first mandatory argument `f` is one or not one. It then either executes the first or the second branch. Beware that both branches must be present.

*source***12.51. \xintifOdd** $\frac{f}{f}$ 

$\frac{f}{f} nn \star$  `\xintifOdd{⟨N⟩}{⟨odd⟩}{⟨not odd⟩}` expandably checks if the first mandatory argument `f`, after truncation to an integer, is odd or even. It then executes accordingly the first or the second branch. Beware that both branches must be present.

*source***12.52. \xintifCmp** $\frac{f}{f}$ 

$\frac{f}{f} f nnn \star$  `\xintifCmp{⟨f⟩}{⟨g⟩}{⟨if f<g⟩}{⟨if f=g⟩}{⟨if f>g⟩}` compares its first two arguments and chooses accordingly the correct branch.

*source***12.53. \xintifEq** $\frac{f}{f}$ 

$\frac{f}{f} f nn \star$  `\xintifEq{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks equality of its two first arguments and executes accordingly the `YES` or the `NO` branch.

*source***12.54. \xintifGt** $\frac{f}{f}$ 

$\frac{f}{f} f nn \star$  `\xintifGt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks if `f > g` and in that case executes the `YES` branch.

*source***12.55. \xintifLt** $\frac{f}{f}$ 

$\frac{f}{f} f nn \star$  `\xintifLt{⟨f⟩}{⟨g⟩}{⟨YES⟩}{⟨NO⟩}` checks if `f < g` and in that case executes the `YES` branch.

*source***12.56. \xintifInt** $\frac{f}{f}$ 

$\frac{f}{f} nn \star$  `\xintifInt{⟨f⟩}{YES branch}{NO branch}` expandably chooses the `YES` branch if `f` reveals itself after expansion and simplification to be an integer.

[source](#)**12.57. \xintSgn**

$\frac{\text{Frac}}{f}$  ★ The sign of a fraction.

[source](#)**12.58. \xintSignBit**

$\frac{\text{Frac}}{f}$  ★ Expands to 1 for negative input, to 0 else.  
Added at 1.41.

[source](#)**12.59. \xintOpp**

$\frac{\text{Frac}}{f}$  ★ The opposite of a fraction. Note that `\xintOpp {3}` produces -3/1[0] whereas `\xintiiOpp {3}` produces -3.

[source](#)**12.60. \xintAbs**

$\frac{\text{Frac}}{f}$  ★ The absolute value. Note that `\xintAbs {-2}=2/1[0]` where `\xintiiAbs {-2}` outputs =2.

[source](#)**12.61. \xintAdd**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the addition of two fractions.  
Since 1.3 always uses the least common multiple of the denominators.

[source](#)**12.62. \xintSub**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the difference of two fractions (`\xintSub{F}{G}` computes F-G).  
Since 1.3 always uses the least common multiple of the denominators.

[source](#)**12.63. \xintMul**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the product of two fractions.  
Output is not reduced to smallest terms.

[source](#)**12.64. \xintDiv**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the quotient of two fractions. (`\xintDiv{F}{G}` computes F/G).  
Output is not reduced to smallest terms.

[source](#)**12.65. \xintDivFloor**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the quotient of two arguments then apply floor function to get an integer (in strict format). This macro was defined at 1.1 (but was left not documented until 1.3a...) and changed at 1.2p, formerly it appended /1[0] to output.

```
\xintDivFloor{-170/3}{23/2}
```

```
-5
```

[source](#)**12.66. \xintMod**

$\frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$  ★ Computes the remainder associated to the floored division `\xintDivFloor`. Prior to 1.2p the meaning was the one of `\xintModTrunc`. Was left undocumented until 1.3a.

```
\xintMod{-170/3}{23/2}
```

```
5/6[0]
```

Modified at 1.3 to use a l.c.m. for the denominator of the result.

*source*

## 12.67. `\xintDivMod`

*Frac Frac*  
*f f* ★ Computes both the floored division and the remainder `\xintDivFloor`. New at 1.2p and documented at 1.3a.

```
\oodef\foo{\xintDivMod{-170/3}{23/2}}\meaning\foo
```

```
macro:->{-5}{5/6[0]}
```

*source*

## 12.68. `\xintDivTrunc`

*Frac Frac*  
*f f* ★ Computes the quotient of two arguments then truncates to an integer (in strict format).

```
\xintDivTrunc{-170/3}{23/2}
```

```
-4
```

*source*

## 12.69. `\xintModTrunc`

*Frac Frac*  
*f f* ★ Computes the remainder associated with the truncated division of two arguments. Prior to 1.2p it was named `\xintMod`, but the latter then got associated with floored division.

```
\xintModTrunc{-170/3}{23/2}
```

```
-64/6[0]
```

Modified at 1.3 to use a l.c.m. for the denominator of the result.

*source*

## 12.70. `\xintDivRound`

*Frac Frac*  
*f f* ★ Computes the quotient of the two arguments then rounds to an integer (in strict format).

```
\xintDivRound{-170/3}{23/2}
```

```
-5
```

*source*

## 12.71. `\xintSqr`

*Frac*  
*f* ★ Computes the square of one fraction.

*source*

## 12.72. `\xintPow`

*Frac Num*  
*f f* ★ `\xintPow{f}{x}`: computes  $f^x$  with  $f$  a fraction and the exponent  $x$  possibly also, but if only `xintfrac` is loaded it will be truncated to an integer.

At 1.4e the behaviour of the macro is enhanced if `xintexpr` is loaded, at it then becomes the support macro for powers  $a^b$ ,  $a**b$  (and the `pow()` function) in `\xinteval`: it now handles also non-integer exponents. Also, if the exponent is an integer, it checks a priori if an exact evaluation would produce more than about 10000 digits and then does in its place a floating point evaluation.

The check whether the exponent is integer is not on the mathematical value but on the format (for reasons of efficiency). So  $4/2$  will not be recognized as integer and it will thus trigger usage of the floating point evaluations; however  $2.0$  will be recognized as an integer, as of course  $2$ .

If the exponent is considered an integer it is then checked if it is less than 10000 (in absolute value) and if the output would contain less than 10000 digits (separately for numerator and denominator) and only then is the power computed exactly. Else it is computed as by `\xintFloatPower` (but the output uses raw `A[N]` format not scientific notation). Use `\xintiiPow` (on integers only, not fractions) for exact powers with larger exponents.

Also, a check is done whether the exponent is half-integer. Again this check is not on the value but on the format, so  $2.5$  is an half integer, as is  $25e-1$ , or  $2.50$  but  $5/2$  is not considered an half-integer (for reasons of internal efficiency). If the exponent is half-integer the power is computed by combining suitably `\xintFloatPower` with `\xintFloatSqrt` (but the output uses raw `A[N]` format not scientific notation).

If the exponent is neither an integer nor an half-integer, the power is computed using logarithm and exponential based approach (and uses raw `A[N]` output format). If `Digits` is at most 8

(which triggers `poormanlog` usage, for very fast logarithms but only with about 8 or 9 accurate fractional digits) this will start being inaccurate in the last digit already with fractional exponents  $x > 10$ . It is recommended to split then the exponent into an integer or half-integer part and a fractional part. Powers with integer or half-integer exponents, even very big, are always computed accurately, for any value of `Digits`.

Within an `\xintiexpr..relax` the infix operators `^` and `**` are mapped to `\xintiiPow` and powers are always computed exactly even if they would produce more than 10000 digits and melt your CPU; within an `\xintexpr`-ession `^` and `**` are mapped to `\xintPow` as described here.

*source*

### 12.73. `\xintFac`

*Num f* ★ This is a convenience variant of `\xintiiFac` which applies `\xintNum` to its argument. Notice however that the output will have a trailing `[0]` according to the `xintfrac` format for integers.

*source*

### 12.74. `\xintBinomial`

*Num Num f f* ★ This is a convenience variant of `\xintiiBinomial` which applies `\xintNum` to its arguments. Notice however that the output will have a trailing `[0]` according to the `xintfrac` format for integers.

*source*

### 12.75. `\xintPFactorial`

*Num Num f f* ★ This is a convenience variant of `\xintiiPFactorial` which applies `\xintNum` to its arguments. Notice however that the output will have a trailing `[0]` according to the `xintfrac` format for integers.

*source*

### 12.76. `\xintMax`

*Frac Frac f f* ★ The maximum of two fractions. Beware that `\xintMax {2}{3}` produces `3/1[0]`. The original, for use with integers only with no need of normalization, is available as `\xintiiMax: \xintiiMax {2}{3}`.  
*ff* ★ `=3.`  
`\xintMax {2.5}{7.2}`  
`72/1[-1]`

*source*

### 12.77. `\xintMin`

*Frac Frac f f* ★ The minimum of two fractions. Beware that `\xintMin {2}{3}` produces `2/1[0]`. The original, for use with integers only with no need of normalization, is available as `\xintiiMin: \xintiiMin {2}{3}`.  
*ff* ★ `=2.`  
`\xintMin {2.5}{7.2}`  
`25/1[-1]`

*source*

### 12.78. `\xintMaxof`

*f → \* f* ★ The maximum of any number of fractions, each within braces, and the whole thing within braces.  
`\xintMaxof {{1.23}{1.2299}{1.2301}}` and `\xintMaxof {{-1.23}{-1.2299}{-1.2301}}`  
`12301/1[-4]` and `-12299/1[-4]`

*source*

### 12.79. `\xintMinof`

*f → \* f* ★ The minimum of any number of fractions, each within braces, and the whole thing within braces.  
`\xintMinof {{1.23}{1.2299}{1.2301}}` and `\xintMinof {{-1.23}{-1.2299}{-1.2301}}`  
`12299/1[-4]` and `-12301/1[-4]`

*source*

## 12.80. \xintSum

$f \rightarrow * \overset{\text{Frac}}{f} \star$  This computes the sum of fractions. The output will now always be in the form  $A/B[n]$ . The original, for big integers only (in strict format), is available as `\xintiiSum`.

```
\xintSum {{1282/2196921}}{-281710/291927}{4028/28612}}
-5037928302100692/6116678670072468[0]
```

No simplification attempted.

*source*

## 12.81. \xintPrd

$f \rightarrow * \overset{\text{Frac}}{f} \star$  This computes the product of fractions. The output will now always be in the form  $A/B[n]$ . The original, for big integers only (in strict format), is available as `\xintiiPrd`.

```
\xintPrd {{1282/2196921}}{-281710/291927}{4028/28612}}
-1454721142160/18350036010217404[0]
```

No simplification attempted.

```
\xintIsOne {21921379213/21921379213}\neq
\xintIsOne {1.00000000000000000000000000000001}$
```

$1 \neq 0$

*source*

## 12.82. \xintGCD

$\overset{\text{Frac}}{f} \overset{\text{Frac}}{f} \star$  The greatest common divisor of its two arguments, which are possibly *fractions*. Prior to 1.4 a macro of the same name existed in `xintgcd`. But it truncated its two arguments to integers via `\xintNum`. See `\xintiiGCD` for the integer only variant.

*source*

## 12.83. \xintLCM

$\overset{\text{Frac}}{f} \overset{\text{Frac}}{f} \star$  The least common multiple of its two arguments, which are possibly *fractions*. Prior to 1.4 a macro of the same name existed in `xintgcd`. But it truncated its two arguments to integers via `\xintNum`. See `\xintiiLCM` for the integer only variant.

*source*

## 12.84. \xintGCDof

$f \rightarrow * \overset{\text{Frac}}{f} \star$  `\xintGCDof{{a}{b}{c}...}` computes the greatest common divisor of  $a, b, \dots$ . The arguments are allowed to be *fractions*: the macro produces the non-negative generator of the fractional ideal they generate. The list argument may be a macro as it is *f-expanded* first. If all arguments vanish, then also the output.

Prior to 1.4 a macro of the same name existed in `xintgcd`. But it truncated all its arguments to integers via `\xintNum` and then proceeded with integer only computations.

See `\xintiiGCDof` for the integer only variant.

*source*

## 12.85. \xintLCMof

$f \rightarrow * \overset{\text{Frac}}{f} \star$  `\xintLCMof{{a}{b}{c}...}` computes the least common multiple of  $a, b, \dots$ . The arguments are allowed to be *fractions*: the macro produces the non-negative generator of the intersection of the corresponding fractional ideals. The list argument may be a macro, it is *f-expanded* first. If one of the item vanishes, then also the output.

Prior to 1.4 a macro of the same name existed in `xintgcd`. But it truncated all its arguments to integers via `\xintNum`.

See `\xintiiLCMof` for the integer only variant.

[source](#)

## 12.86. `\xintDigits`, `\xinttheDigits`

The syntax `\xintDigits := N`; or (recommended) `\xintDigits := N\relax` assigns the value of `N` to the number of digits to be used by floating point operations (this uses internally a `\mathchardef` assignment, and `N` stands for (or expands to) a legal  $\TeX$  number). The default is `16`. The maximal value is `32767`.

Accepted syntax includes also `\xintDigits = N`; or `\xintDigits = N\relax`, i.e. the colon before the equality sign is optional.



`xintexpr` adds the variant `\xintDigits*` which executes `\xintreloadxinttrig` and `\xintreloadxintlog`.

A priori, you want `\xintDigits*:=N\relax`. Use `\xintDigits:=N\relax` only if not needing trigonometric or logarithm/exponential functions and wanting to avoid the overhead of reloading their libraries. Perhaps for a local temporary configuration.

Spaces do not matter as long as they do not occur in-between digits:

```
\xintDigits := 24\relax\xinttheDigits, %
\xintDigits:=36 \relax\xinttheDigits, %
\xintDigits:= 16 \relax and \xinttheDigits.
```

★ `24`, `36`, and `16`. As shown above `\xinttheDigits` expands to the stored value.

An ending active semi-colon `;` is not compatible: it can and will cause low-level  $\TeX$  errors. This is why the alternative syntax

```
\xintDigits:= N\relax
```

is recommended (with or without the semi-colon). This is hopefully the syntax now in use in most examples from the documentation.

Actually, any non-expanding token can be used in place of the `\relax`. This non-expanding ending token (for example a full stop) will get removed from the token stream.

```
\xintDigits = 24\def\xinttheDigits, % only for showing it works! don't do that!
\xintDigits := 36.\xinttheDigits, % one can use a dot in place of semi-colon
\xintDigits = 16\relax and \xinttheDigits.\par % with \relax, even better
```

`24`, `36`, and `16`.

[source](#)

## 12.87. `\xintSetDigits`

 $\frac{\text{num}}{x}$ 

To be used as `\xintSetDigits{expression}` where the expression will be fed to `\numexpr`. It is a shortcut for doing `\xintDigits := \numexpr expression \relax \relax`.

```
\xintSetDigits{1+2+3+4+5}The value is now \xinttheDigits.
\xintSetDigits{2*8}The value is now \xinttheDigits.\par
```

The value is now `15`. The value is now `16`.

The `xintexpr`-added variant `\xintSetDigits*` is the preferred usage as it does the extra work to update the math functions from `xinttrig` and `xintlog`.

[source](#)

## 12.88. `\xintFloat`

 $\frac{\text{num}}{x}$   $\frac{\text{Frac}}{f}$  ★

The macro `\xintFloat [P]{f}` has an optional argument `P` which replaces the current value of `\xinttheDigits`. The fraction `f` is then printed in scientific notation with a rounding to `P` digits.

That is, on output: the first digit is from `1` to `9`, it is possibly prefixed by a minus sign and is followed by a dot and `P-1` digits, then a lower case `e` and an exponent `N`. The trailing zeroes are not trimmed.

```
\def\xintFloatZero{0.0e0}
```

[illegible]

```
{\def\x{137893789173289739179317/13890138013801398}%  
\xintFor* #1 in {\xintSeq{4}{20}}  
\do{#1: \xintFloat[#1]{\x}\newline}}%  
\xintFloat{5/9999999999999999}\newline  
\xintFloat[32]{5/9999999999999999}\newline  
\xintFloat[48]{5/9999999999999999}\par
```

184



```

7: 9.927460e6
8: 9.9274600e6
9: 9.92745997e6
10: 9.927459975e6
11: 9.9274599746e6
12: 9.92745997457e6
13: 9.927459974572e6
14: 9.9274599745717e6
15: 9.92745997457166e6
16: 9.927459974571665e6
17: 9.9274599745716647e6
18: 9.92745997457166465e6
19: 9.927459974571664655e6
20: 9.9274599745716646545e6
5.0000000000000001e-16
5.000000000000000500000000000001e-16
5.000000000000000500000000000005000000000001e-16

```

*source***12.89. \xintFloatBraced**

$$\left[\frac{\text{num}}{x}\right]_f^{\text{Frac}} \star$$

The experimental macro `\xintFloatBraced[P]{f}` does like `\xintFloat` but its output consists of three TeX-braced groups

$\{\langle \text{sign bit} \rangle\}\{\langle \text{scientific exponent} \rangle\}\{\langle \text{full width mantissa with decimal point} \rangle\}$

It is provided for users knowing how to pick one or the other of these constituents from usage of auxiliary macros. Or one can use `\xintAssign`:

```

\xbegingroup
\xintAssign\xintFloatBraced[7]{-3.1234e-14}\to\A\B\C
\string\A\ has meaning \meaning\A\newline
\string\B\ has meaning \meaning\B\newline
\string\C\ has meaning \meaning\C\par
\endgroup

```

`\A` has meaning macro:->1

`\B` has meaning macro:->-14

`\C` has meaning macro:->3.123400

Some aspects are undecided:

- should the first item be rather -1, 0, or 1? or -, nothing, nothing?
- in case of zero value the output ignores `\xintFloatZero`, it uses a zero exponent and full width fractional mantissa `0.000000000000000` (here no `[P]` and `\xinttheDigits` has value 16), should it be otherwise?
- should the mantissa be without the decimal separator? should it incorporate the sign?
- in case the mantissa is without separator, should the exponent be biased to match it?

*source***12.90. \xintFloatToDecimal**

$$\left[\frac{\text{num}}{x}\right]_f^{\text{Frac}} \star$$

`\xintFloatToDecimal [P]{f}` does float rounding on input like `\xintFloat` then outputs the number using decimal notation, i.e. with as many zeros as are needed (and no more) and no scientific exponent.

In other terms it behaves (and is essentially defined) as:

```
\xintDecToStringREZ{\xintFloat[optional P]{<input>}}
```

Examples:

```

\xintFloatToDecimal{6.02e23}\newline
\xintFloatToDecimal{6.02000000000000e23}\newline

```

See [\xintDecToString](#).

## 12.91. \xintPFloat

This macro was initially added at [1.1](#) as a (very primitive) "prettifying printer" for floating point number, and was then somewhat influenced by Maple, for example the zero value was printed as "0.". Then at [1.4e](#) there was breaking change and the rules became somewhat similar to observed Python behaviour: mantissas trimmed of trailing zeros (whether or not scientific notation was used in the output) and integers printed with a trailing ".0", in particular the zero value was printed as "0.0".

- Integers (when scientific notation is dropped according to criteria mentioned next) without a ".0" suffix.
- Same for the zero value, now "0".
- Significands are trimmed of trailing zeros only if that removes at least 4 zeros. The rationale is that automatic removal of trailing zeros (which was influenced at 1.4e from practice with Python in interactive mode) proves annoying visually with aligned values in tables, as this creates voids, so we want to do this only when really the presence of trailing zeros is not some kind of numerical fluke.



In this documentation ``trailing zeros'`` refers not to how the input looked like, but to the corresponding mantissa of width `P` or `\xinttheDigits`.

1. The input is float-rounded to either **Digits** or the optional argument.
2. zero is printed as **0**.
3. **x.yz...eN** is printed in decimal fixed point if  $-4 \leq N \leq +5$  else it is printed in scientific notation.
4. Trailing zeros of the mantissa are trimmed if, and only if there are at least **4** of them.
5. In case of fixed point output format, and the value is an integer, the integer is printed with no decimal mark.
6. In case of scientific notation output format, and the mantissa has only one digit, no decimal mark is used.

- $1.2340000e-3 \rightarrow 0.001234$
- $1.2340000e-2 \rightarrow 0.01234$
- $1.2340000e-1 \rightarrow 0.1234$
- $1.2340000e0 \rightarrow 1.234$
- $1.2340000e1 \rightarrow 12.34$
- $1.2340000e2 \rightarrow 123.4$

- $1.2340000e3 \rightarrow 1234$
- $1.2340000e4 \rightarrow 12340$
- $1.2340000e5 \rightarrow 123400$
- $1.2340000e6 \rightarrow 1.234e6$
- $1.2340000e7 \rightarrow 1.234e7$
- $1e-7/7 \rightarrow 1.428571428571429e-8$
- $1e-6/7 \rightarrow 1.428571428571429e-7$
- $1e-5/7 \rightarrow 1.428571428571429e-6$
- $1e-4/7 \rightarrow 1.428571428571429e-5$
- $1e-3/7 \rightarrow 0.0001428571428571429$
- $1e-2/7 \rightarrow 0.001428571428571429$
- $1e-1/7 \rightarrow 0.01428571428571429$
- $1e0/7 \rightarrow 0.1428571428571429$
- $1e1/7 \rightarrow 1.428571428571429$
- $1e2/7 \rightarrow 14.28571428571429$
- $1e3/7 \rightarrow 142.8571428571429$
- $1e4/7 \rightarrow 1428.571428571429$
- $1e5/7 \rightarrow 14285.71428571429$
- $1e6/7 \rightarrow 142857.1428571429$
- $1e7/7 \rightarrow 1.428571428571429e6$

[source](#)

### 12.91.1. Customizing macros of `\xintPFloat`

A number of macros allow to customize the behaviour of `\xintPFloat`:

- `\xintPFloatE` allows to modify the separator of the scientific notation. Here is its default:  
<sup>70</sup>

```
\def\xintPFloatE{e}
```

- `\xintPFloatZero` says how to print the zero value. The default:

```
\def\xintPFloatZero{0}
```

- `\xintPFloatIntSuffix` is postfixed to integer values (when scientific notation is not used). Its default at **1.4k** is to add nothing. It replaces the formerly hard-coded ".0" from **1.4e** (prior to that trailing zeros from the full significand of **P** or `\xinttheDigits` digits were not trimmed).

```
\def\xintPFloatIntSuffix{}
```

- `\xintPFloatLengthOneSuffix` is postfixed to trimmed mantissas having only one digit, when scientific notation is used. Its default at **1.4k** is to add nothing. It replaces formerly hard-coded ".0".

```
\def\xintPFloatLengthOneSuffix{}
```

- `\xintPFloatNoSciEmax` is the maximal scientific exponent which will trigger use of decimal fixed point notation and `\xintPFloatNoSciEmin` is the minimal one. Their defaults at **1.4k** are the same as the formerly hard-coded behaviour from **1.4e**:

```
\def\xintPFloatNoSciEmax{5}
\def\xintPFloatNoSciEmin{-4}
```

For example (with the package default width of **16** digits for mantissas of floating point numbers):

```
\begingroup
\def\xintPFloatNoSciEmin{-20}
\xintPFloat{1e-19/7}\newline
\xintPFloat{1e-20/7}\par
\def\xintPFloatNoSciEmax{19}
\xintPFloat{1e20/7}\newline
\xintPFloat{1e21/7}\par
```

<sup>70</sup> For  $\text{\TeX}$ pers: it is allowed to define `\xintPFloatE` as a macro which grabs the exponent as an argument delimited by a dot, and produces *f-expand*ably an output also delimited by a dot (it will be removed via further internal processing).

[TOC](#), [xint bundle](#), [xintkernel](#), [xintcore](#), [xint](#), [xintfrac](#), [xintbinhex](#), [xintgcd](#), [xintseries](#), [xintcfrac](#)

[illegible]

- ```
\def\xintPFloatMinTrimmed{4}
```

This setting is ignored for the case of an integer value, if the criteria for using fixed point notation are met, and for the case of a one-digit mantissa in scientific notation.

```
\def\xintPFloatZero{0.0}%
\def\xintPFloatIntSuffix{.0}%
\def\xintPFloatLengthOneSuffix{.0}%
\def\xintPFloatNoSciEmax{15}%
\def\xintPFloatNoSciEmin{-4}%
\def\xintPFloatMinTrimmed{-1}%
```

$0. \rightarrow 0.0$   
 $1234. \rightarrow 1234.0$   
 $6e100 \rightarrow 6.0e100$   
 $1234567812345678.12345678 \rightarrow 1234567812345678.0$   
 $12345678123456781.2345678 \rightarrow 1.234567812345678e16$   
 $12345678.12340 \rightarrow 12345678.1234$   
 $12345678.123400 \rightarrow 12345678.1234$   
 $0.1234567812345678 \rightarrow 0.1234567812345678$   
 $0.00012345678123456785 \rightarrow 0.0001234567812345679$   
 $0.000012345678123456785 \rightarrow 1.234567812345679e-5$

Same, but playing with `xintsession` in its `&fp` mode:

```
>>> &fp
fp mode (16 digits)
>>> \\def\xintPFloatZero{0.0}
(executing \\def\xintPFloatZero {0.0} in background)
)
Runaway argument?
def\xintPFloatZero {0.0}\message {
}\xs_fetch_aa \endingut
! File ended while scanning use of \\.
<inserted text>
\par
<*> xintsession^^M

? S
OK, entering \scrollmode...
```

```
*\xintsession
```

```
You are back to the xintexpr interactive session!
(current mode: fp (Digits=16), with Digits=16)
```

```
">>> " means central computing is waiting for input
"... " means that multi-line input continues. Use `;' to terminate it.
```

```
Say '&bye' at any time to terminate the session and the TeX run.
```

```
>>> \def\xintPFloatZero{0.0}
```

```
(executing \def \xintPFloatZero {0.0} in background)
```

```
>>> \def\xintPFloatIntSuffix{.0}\def\xintPFloatLengthOneSuffix{.0}
```

```
(executing \def \xintPFloatIntSuffix {.0}\def \xintPFloatLengthOneSuffix {.0} i
n background)
```

```
>>> \def\xintPFloatNoSciEmax{15}\def\xintPFloatNoSciEmin{-4}
```

```
(executing \def \xintPFloatNoSciEmax {15}\def \xintPFloatNoSciEmin {-4} in back
ground)
```

```
>>> \def\xintPFloatMinTrimmed{-1}
```

```
(executing \def \xintPFloatMinTrimmed {-1} in background)
```

```
>>> 0., 1234., 6e100;
```

```
@_1      0.0, 1234.0, 6.0e100
```

```
>>> 1234567812345678.12345678;
```

```
@_2      1234567812345678.0
```

```
>>> 12345678123456781.2345678;
```

```
@_3      1.234567812345678e16
```

```
>>> 12345678.12340;
```

```
@_4      12345678.1234
```

```
>>> 12345678.123400;
```

```
@_5      12345678.1234
```

```
>>> 0.1234567812345678;
```

```
@_6      0.1234567812345678
```

```
>>> 0.00012345678123456785;
```

```
@_7      0.0001234567812345679
```

```
>>> 0.000012345678123456785;
```

```
@_8      1.234567812345679e-5
```

```
>>> &bye
```

This is with version **0.4alpha** (2021-11-01) of **xintsession**. Probably some ``magic'' shortcuts will be added in future to its interface for this kind of tasks, in place of the **\def**.

*source*

## 12.92. \xintFloatAdd

$\text{num. } \frac{\text{Frac}}{\text{x}} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$  **\xintFloatAdd** [P]{f}{g} first replaces **f** and **g** with their float approximations **f'** and **g'** to **P** significant places or to the precision from **\xintDigits**. It then produces the sum **f'+g'**, correctly rounded to nearest with the same number of significant places.

*source***12.93. \xintFloatSub**

$\frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$  `\xintFloatSub [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` significant places or to the precision from `\xintDigits`. It then produces the difference `f'-g'` correctly rounded to nearest `P`-float.

*source***12.94. \xintFloatMul**

$\frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$  `\xintFloatMul [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` (or `\xinttheDigits`) significant places. It then correctly rounds the product `f'*g'` to nearest `P`-float.

See [subsection 8.2](#) for more.

It is obviously much needed that the author improves its algorithms to avoid going through the exact  $2P$  or  $2P-1$  digits before throwing to the waste-bin half of those digits !

*source***12.95. \xintFloatDiv**

$\frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$  `\xintFloatDiv [P]{f}{g}` first replaces `f` and `g` with their float approximations `f'` and `g'` to `P` (or `\xinttheDigits`) significant places. It then correctly rounds the fraction `f'/g'` to nearest `P`-float.

See [subsection 8.2](#) for more.

Notice in the special situation with `f` and `g` integers that `\xintFloatDiv [P]{f}{g}` will not necessarily give the correct rounding of the exact fraction `f/g`. Indeed the macro arguments are each first individually rounded to `P` digits of precision. The correct syntax to get the correctly rounded integer fraction `f/g` is `\xintFloat[P]{f/g}`.

*source***12.96. \xintFloatPow**

$\frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{num}}{x} \star$  `\xintFloatPow [P]{f}{x}` uses either the optional argument `P` or in its absence the value of `\xinttheDigits`. It computes a floating approximation to `f^x`.

The exponent `x` will be handed over to a `\numexpr`, hence count registers are accepted on input for this `x`. And the absolute value `|x|` must obey the  $\mathbb{T}_X$  bound.

The argument `f` is first rounded to `P` significant places to give `f'`. The output `Z` is such that the exact `f'^x` differs from `Z` by an absolute error less than  $0.52 \text{ ulp}(Z)$ .

```
\xintFloatPow [8]{3.1415}{1234567890}
1.6122066e613749456
```

*source***12.97. \xintFloatPower**

$\frac{\text{num}}{x} \frac{\text{Frac}}{f} \frac{\text{Num}}{f} \star$  `\xintFloatPower[P]{f}{g}` computes a floating point value `f^g` where the exponent `g` is not constrained to be at most the  $\mathbb{T}_X$  bound `2147483647`. It may even be a fraction `A/B` but will be truncated to an integer. The exponent of the output however *must* at any rate obey the  $\mathbb{T}_X$  bound.

The argument `f` is first rounded to `P` significant places to give `f'`. The output `Z` is then such that the exact `f'^g` differs from `Z` by an absolute error less than  $0.52 \text{ ulp}(Z)$ .

For integer exponents this is the support macro which is used for the `^` (or `**`) infix operators in `\xintfloateval`, or also in `\xinteval` for very big integer exponents. It is also used in `\xintfloateval` and `\xinteval` for half-integer exponents, via a combination with the `\xintFloatSqrt` square-root extraction.

The macro itself was *NOT* modified at **1.4e**: when used directly it still starts by truncating the exponent to an integer... As for other user-level floating-point macros, its output is handled by `\xintFloat`, i.e. it uses scientific notation.

The integer exponent `g` may have more than `P` (or `Digits`) digits, it is handled exactly. And as said above its absolute value may exceed the `TeX` bound.

### 12.98. \xintFloatSqrt

More precisely since 1.2f the macro achieves so-called *correct rounding*: the produced value is the rounding to  $P$  significant places of the abstract exact value, *if the input has itself at most  $P$  digits* (and an arbitrary exponent).

`3.1622776601683793319988935444327185337195551393252168268575048527925944386392382213442481e0`

`1.000e1`

`1.111111106055555440541666143353469245878409860134351071458570675251471479496366736579136e4`

```
\xintFloatSqrt [80]{562500000000000000000000000750000000000000000000001}\newline
```

(we observe in passing illustrations that rounding to nearest is not transitive.)

12.99. \xintFloatFac

$1000! \approx 4.02387260077093773543702433923e2567$  The computation proceeds via doing explicitly the



product.<sup>71</sup>

The maximal allowed argument is 99999999, but already 100000! currently takes, for 16 digits of precision, a few seconds on my laptop (it returns 2.824229407960348e456573).

The `factorial` function is available in `\xintfloatexpr`:

```
\xintthefloatexpr factorial(1000)\relax % same as 1000!
4.023872600770938e2567
```

[source](#)

## 12.100. `\xintFloatBinomial`

[ $\frac{\text{num}}{x}$ ] Num Num f f ★ `\xintFloatBinomial[P]{x}{y}` computes binomial coefficients with either `\xinttheDigits` or `P` digits of precision.

When  $x < 0$  an out-of-range error is raised. Else if  $y < 0$  or if  $x < y$  the macro evaluates to `0.0e0`. The exact theoretical value differs from the calculated one `Y` by an absolute error strictly less than `0.6 ulp(Y)`.

```
${3000\choose 1500}\approx{\xintFloatBinomial [24]{3000}{1500}}
(3000 \choose 1500) \approx 1.79196793754756005073269e901
```

The associated function in `\xintfloatexpr` is `binomial()`:

```
\xintthefloatexpr binomial(3000,1500)\relax
1.791967937547560e901
```

The computation is based on the formula  $(x-y+1)\dots x/y!$  (here one arranges  $y \leq x-y$  naturally).

[source](#)

## 12.101. `\xintFloatPFactorial`

[ $\frac{\text{num}}{x}$ ] Num Num f f ★ `\xintFloatPFactorial[P]{x}{y}` computes the product  $(x+1)\dots y$ .

The arguments must be integers (they are expanded inside `\numexpr`) and the allowed range is  $-1000000000 \leq x, y \leq 99999999$ . If  $x \geq y$  the product is considered empty hence returns one (as a floating point value). See also `\xintiiPFactorial`.

The exact theoretical value differs from the calculated one `Y` by an absolute error strictly less than `0.6 ulp(Y)`.

The associated function in `\xintfloatexpr` is `pfactorial()`:

```
\xintthefloatexpr pfactorial(2500,5000)\relax
2.595989917947957e8914
```

<sup>71</sup> An approach based upon the Stirling formula could not be done at time of implementation because of lack of exponential and logarithm. This is now supported via package `xintlog`. So perhaps at some point in future Gamma function will be implemented.

## 13. Macros of the **xintbinhex** package

.1	<b>xintexpr</b> -essions .....	193	.9	<code>\xintOctToHex</code> .....	195
.2	<code>\xintHexToDec</code> .....	194	.10	<code>\xintOctToDec</code> .....	195
.3	<code>\xintHexToOct</code> .....	194	.11	<code>\xintOctToBin</code> .....	196
.4	<code>\xintHexToBin</code> .....	194	.12	<code>\xintCOctToBin</code> .....	196
.5	<code>\xintCHexToBin</code> .....	194	.13	<code>\xintBinToHex</code> .....	196
.6	<code>\xintDecToHex</code> .....	195	.14	<code>\xintBinToDec</code> .....	196
.7	<code>\xintDecToOct</code> .....	195	.15	<code>\xintBinToOct</code> .....	196
.8	<code>\xintDecToBin</code> .....	195	.16	Maximal sizes of inputs .....	197

This package provides expandable conversions of (big) integers between the hexadecimal, decimal, octal (since 1.4n) and binary bases.

First version of this package was in the 1.08 (2013/06/07) release of **xint**. Its routines remained unmodified until their complete rewrite at release 1.2m (2017/07/31). Macros became faster, but the inputs got limited to a few thousand digits, whereas the 1.08 versions could handle (slowly...) tens of thousands of digits. At 1.4n some internals were refactored to use the **\expanded** primitive (which was not available in 2017). The maximal sizes got increased, see subsection 13.16. More significant probably, the octal radix was added to the ones covered.

New with  
1.4n

For each provided conversion macro, its argument is first *f-expanded*. This expansion is supposed to give a sequence of digits, with perhaps a (unique) leading minus sign, which gets prepended to output (note that `\xintDecToHex{-0}` thus expands to `-0`).

Let's insist that inputs can not start (after expansion) with a `0b`, `0o`, `0x`, `#x`, `"`, `'`, or similar prefix notation: they must consist only of digits as fitting to the binary, octal, decimal, or hexadecimal radix. Situation is different if using **xintexpr**-essions, see subsection 13.1 next.

Low-level unrecoverable errors will occur if for example an octal input contains the decimal digit 8 (more instructive errors are raised if inside an **xintexpr**-ession).

Hexadecimal digits in input must be uppercased. Category codes for them may be indifferently *letter* or *other*. In output they are of category *letter* (and uppercased).

Leading zeroes in the input are allowed, and depending on the macro may show up or not in the output. Note in particular:

- Inputs with no leading zeros give outputs with no leading zeros.
- All rules have (deliberate) exceptions, check the docs of `\xintCHexToBin` and `\xintCOctToBin` which are variants of `\xintHexToBin` and `\xintOctToBin`.
- Outputs (if non vanishing) from `\xintDecToHex` or `\xintDecToOct` have no leading zeros whether or not the inputs had some.
- `\xintBinToHex` and `\xintBinToOct` always use the minimal number of hexadecimal resp. octal digits as needed to represent the original binary digits, inclusive of their leading zeroes. For example `\xintBinToHex{0000001}` outputs `01` and `\xintBinToOct{0000001}` outputs `001`.

### 13.1. **xintexpr**-essions

New with  
1.4n

Inside **xintexpr**-essions, hexadecimal can be input using either `"` or `0x` prefixes, octal using either `'` or `0o`, and binary using `0b`. Prior to 1.4n only `"` was implemented and it was needed to load **xintbinhex** additionally to **xintexpr**. This is now done automatically.

Hexadecimal letters must be uppercased. In both of `\xinteval` and `\xintfloateval` a ``fractional part'' after the full stop as separator is allowed for all three bases. Here is an example using the three non-decimal bases:

```
\xinteval{subsm({x, y, x=y}, x="FF.FF * '777.777 * 0b11111.11111;
y=(16^2-16^2)(8^3-8^3)(2^5-2^5);)}
4.1901280783689022064208984375e6, 17574670959615/4194304, 1
```

The **p** postfix notation from some programming languages, which stands for an extra power of two, is however not implemented so far.

New with  
1.4n

With `\xintiieval`, which handles only integers, there is an optional parameter `[h]`, `[o]`, or `[b]` for automatic conversion of the output (this works also with comma separated inputs and even nested bracketed inputs). Here is an example illustrating the new more condensed syntax:

```
\xintiieval[b]{0b1011100101001110 * 0b111100011}
```

```
1010111011001111000101010
```

Compare with how one would have had to input it prior to 1.4n:

```
\xintDecToBin{\xintiieval{\xintBinToDec{1011100101001110}*\xintBinToDec{111100011}}}
```

```
1010111011001111000101010
```

[source](#)

## 13.2. `\xintHexToDec`

*f* ★ Converts from hexadecimal to decimal.

```
\xintHexToDec{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603  
2936BF37DAC918814C63}
```

```
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217  
8525166427427466391932003
```

[source](#)

## 13.3. `\xintHexToOct`

*f* ★ Converts from hexadecimal to octal.

New with  
1.4n

```
\xintHexToOct{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603  
2936BF37DAC918814C63}
```

```
->432447137063224465134050767641240274424707602613704066107044272736255273013547064471364333  
4710661401451155374676654443040246143
```

[source](#)

## 13.4. `\xintHexToBin`

*f* ★ Converts from hexadecimal to binary. Up to three leading zeroes of the binary output are trimmed.

```
\xintHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C603  
2936BF37DAC918814C63}
```

```
->10001101010010011100101111000110011010010100100110101001011100000101000111110111110100001  
01010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001  
0111010111011110010101101010111011000001011101100111000110100100111001011110100011011011001  
110010001101100011000000011001010010011011010111110011011110110101100100100011000100000010  
100110001100011
```

[source](#)

## 13.5. `\xintCHexToBin`

*f* ★ Converts from hexadecimal to binary. Same as `\xintHexToBin`, but an input with **N** hexadecimal digits will give an output with exactly **4N** binary digits, leading zeroes are not trimmed.

```
\xintCHexToBin{11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C60  
32936BF37DAC918814C63}
```

```
->000100011010100100111001011111000110011010010100100110101001011100000101000111110111110100  
00101010000001011110010001010011100011111000001011000101111100010000011011000100011100010010  
0010111010111011110010101101010110110000010111011001110001101001001110010111101000110110111  
001110010001101100011000000011001010010011011010111110011011110110101100100100011000100000  
010100110001100011
```

This can be combined with `\xintBinToHex` for round-trips preserving leading zeroes for **4N** binary digits numbers, whereas using `\xintHexToBin` gives reproducing round-trips only for **4N** binary numbers not starting with `0000`.

```
\xintBinToHex{0001111}\par
```

0F

Chaining, we end up with 4N-3 digits, as three binary zeroes are trimmed:

`\xintHexToBin{\xintBinToHex{0001111}}\par`

01111

But the next will always reproduce the initial input zero-filled to length 4N:

`\xintCHexToBin{\xintBinToHex{0001111}}\par`

00001111

Another example (visible space characters manually inserted):

000000001111101001010001	$\xrightarrow{\text{\code{\xintBinToHex}}}$	00FA51	$\xrightarrow{\text{\code{\xintHexToBin}}}$	0000001111101001010001
000000001111101001010001	$\xrightarrow{\text{\code{\xintBinToHex}}}$	00FA51	$\xrightarrow{\text{\code{\xintCHexToBin}}}$	000000001111101001010001

[source](#)

### 13.6. \xintDecToHex

*f* ★ Converts from decimal to hexadecimal.

```
\xintDecToHex{2718281828459045235360287471352662497757247093699959574966967627724076630353
547594571382178525166427427466391932003}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918
814C63
```

[source](#)

### 13.7. \xintDecToOct

*f* ★  
New with  
1.4n

```
\xintDecToOct{2718281828459045235360287471352662497757247093699959574966967627724076630353
547594571382178525166427427466391932003}
->432447137063224465134050767641240274424707602613704066107044272736255273013547064471364333
4710661401451155374676654443040246143
```

[source](#)

### 13.8. \xintDecToBin

*f* ★ Converts from decimal to binary.

```
\xintDecToBin{2718281828459045235360287471352662497757247093699959574966967627724076630353
547594571382178525166427427466391932003}
->10001101010010011100101111000110011010010100100110101001011100000101000111110111110100001
01010000001011110010001010011100011111000001011000101111100010000011011000100011100010010001
011101011101110010101101010110110000010111011001110001101001001110010111101000110110111001
110010001101100011000000011001010010011011010111110011011110110101100100100011000100000010
100110001100011
```

[source](#)

### 13.9. \xintOctToHex

*f* ★  
New with  
1.4n

```
\xintOctToHex{4324471370632244651340507676412402744247076026137040661070442727362552730135
470644713643334710661401451155374676654443040246143}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918
814C63
```

[source](#)

### 13.10. \xintOctToDec

*f* ★  
New with  
1.4n

```
\xintOctToDec{4324471370632244651340507676412402744247076026137040661070442727362552730135
470644713643334710661401451155374676654443040246143}
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217
8525166427427466391932003
```

[source](#)**13.11. \xintOctToBin***f* ★  
New with  
1.4n

Converts from octal to binary. Up to two leading zeroes of the binary output are trimmed.

```
\xintOctToBin{1432447137063224465134050767641240274424707602613704066107044272736255273013
5470644713643334710661401451155374676654443040246143}
->110001101010010011100101111100011001101001010010011010100101110000010100011111011111010000
10101000000101111001000101001110001111100000101100010111110001000001101100010001110001001000
10111010111011110010101101010111011000001011101100111000110100100111001011110100011011011100
1110010001101100011000000011001010010011011010111111001101111011010110010010001100010000001
0100110001100011
```

[source](#)**13.12. \xintCOctToBin***f* ★  
New with  
1.4n

Converts from octal to binary.

Same as `\xintOctToBin`, except that an input with **N** octal digits will give an output with exactly **3N** binary digits, leading zeroes are not trimmed.

```
\xintCOctToBin{143244713706322446513405076764124027442470760261370406610704427273625527301
35470644713643334710661401451155374676654443040246143}
->001100011010100100111001011111000110011010010100100110101001011100000101000111110111110100
00101010000001011110010001010011100011111000001011000101111100010000011011000100011100010010
00101110101110111100101011010101110110000010111011001110001101001001110010111101000110110111
0011100100011011000110000000110010100100110110101111110011011110110101100100100011000100000
010100110001100011
```

[source](#)**13.13. \xintBinToHex***f* ★ 

Converts from binary to hexadecimal.

The input is first extended if need-be by leading zeros in order to have **4N** binary digits, then the output will have **N** hexadecimal digits (thus, if the input did not have a leading zero, the output will not either).

```
\xintBinToHex{1000110101001001110010111110001100110100101001001101010010111000001010001111
10111110100001010100000010111100100010100111000111110000010110001011111000100000110110001000
11100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010
00110110111001110010001101100011000000011001010010011011010111111001101111101101011001001000
11000100000010100110001100011}
->11A9397C66949A97051F7D0A817914E3E0B17C41B11C48BAEF2B5760BB38D272F46DCE46C6032936BF37DAC918
814C63
```

[source](#)**13.14. \xintBinToDec***f* ★ 

Converts from binary to decimal.

```
\xintBinToDec{1000110101001001110010111110001100110100101001001101010010111000001010001111
10111110100001010100000010111100100010100111000111110000010110001011111000100000110110001000
11100010010001011101011101111001010110101011101100000101110110011100011010010011100101111010
00110110111001110010001101100011000000011001010010011011010111111001101111101101011001001000
11000100000010100110001100011}
->271828182845904523536028747135266249775724709369995957496696762772407663035354759457138217
8525166427427466391932003
```

[source](#)**13.15. \xintBinToOct***f* ★ 

Converts from binary to octal.

The input is first extended if need-be by leading zeros in order to have **3N** binary digits, then the output will have **N** octal digits (thus, if the input did not have a leading zero, the output will not either).

New with  
1.4n

```
\xintBinToOct{1000110101001001110010111110001100110100101001001101010010111000001010001111
10111110100001010100000010111100100010100111000111110000010110001011111000100000110110001000
1110001001000101110101110111100101011010101110110000010111011001110001101001001110010111010
00110110111001110010001101100011000000011001010010011011010111111001101111101101011001001000
11000100000010100110001100011}
```

->432447137063224465134050767641240274424707602613704066107044272736255273013547064471364333  
4710661401451155374676654443040246143

### 13.16. Maximal sizes of inputs

**Table 4** recapitulates the maximal allowed sizes, as found out with the  $\text{\TeX}$  installation of the author. The tests are done putting the macro inside an `\edef` and compiling with the `etex` binary.

The value in the second column is the maximal **N** such that the macro does not raise an error on an input with **N** digits (if nested in another macro, the maximal input size may become lower than stated). The third column gives the corresponding maximal size of the output.

These maximal sizes depend on  $\text{\TeX}$  parameters such as input stack size, expansion depth, and parameter stack size. The fourth column gives the  $\text{\TeX}$  parameter cited in the error message when trying with **N+1** digits. When the limiting parameter is the main memory size, the upper limits depend on external factors such as how many macros are loaded in  $\text{\TeX}$  memory (for example they would be lower in a  $\text{\LaTeX}$  document or if `xintexpr` is loaded), so they are here given simply as indications. They are so large anyhow that basically in practice, this means no real limitation.

Regarding the conversions to and from decimal radix, they allow a more limited range, but are still able to handle inputs with ten thousand digits, one can thus consider in practice that their size limitations are also only of theoretical interest.

	Max length of input	-> length of output	Limiting factor
<code>\xintDecToHex</code>	16042	13323	expansion depth=10000
<code>\xintDecToOct</code>	15040	16654	expansion depth=10000
<code>\xintDecToBin</code>	16042	53291	expansion depth=10000
<code>\xintHexToDec</code>	11072	13333	expansion depth=10000
<code>\xintOctToDec</code>	14763	13333	expansion depth=10000
<code>\xintBinToDec</code>	44290	13333	expansion depth=10000
<code>\xintHexToOct</code>	553514	738019	main memory size=5000000
<code>\xintHexToBin</code>	553514	2214056	main memory size=5000000
<code>\xintOctToHex</code>	711660	533745	main memory size=5000000
<code>\xintOctToBin</code>	711660	2134980	main memory size=5000000
<code>\xintBinToHex</code>	1992650	498163	main memory size=5000000
<code>\xintBinToOct</code>	1868109	622703	main memory size=5000000

Table 4: Maximal sizes for `xintbinhex` 1.4n macros with TeXLive 2025

## 14. Macros of the *xintgcd* package

.1	<code>\xintBezout</code> .....	198	.4	<code>\xintTypesetEuclideanAlgorithm</code> .....	199
.2	<code>\xintEuclideanAlgorithm</code> .....	198	.5	<code>\xintTypesetBezoutAlgorithm</code> .....	199
.3	<code>\xintBezoutAlgorithm</code> .....	198			

This package was included in the original release 1.0 (2013/03/28) of the *xint bundle*.

At 1.3d macros `\xintiigcd` and `\xintiilcm` are copied over to *xint*, hence `gcd()` and `lcm()` functions in `\xintiexpr` were available simply from loading only *xintexpr*, and the *xintgcd* dependency got removed.

From 1.1 to 1.3f the package loaded only *xintcore*, not *xint* and neither *xinttools*. But at 1.4 it loads automatically both *xint* and *xinttools* (the latter being a requirement since 1.09h of the `\xintTypesetEuclideanAlgorithm` and `\xintTypesetBezoutAlgorithm` macros). The macros `\xintiigcd` and `\xintiilcm` got relocated into *xint*. The macros `\xintgcd`, `\xintlcm`, `\xintgcdof`, and `\xintlcmof` are removed: *xintfrac* provides under these names more powerful macros handling general fractions and not only integers.

source

### 14.1. `\xintBezout`

*Num Num f f* ★ `\xintBezout{N}{M}` returns three numbers *U*, *V*, *D* within braces where *D* is the (non-negative) GCD, and  $UN + VM = D$ .

```
\oodef\X{\xintBezout {10000}{1113}}\meaning\X\par
\xintAssign {\xintBezout {10000}{1113}}\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiexpr 10000*\U+1113*\V\relax\par
\noindent\oodef\X{\xintBezout {123456789012345}{9876543210321}}\meaning\X\par
\xintAssign \X\to\U\V\D
U: \meaning\U, V: \meaning\V, D: \meaning\D\par
AU+BV: \xinttheiexpr 123456789012345*\U+9876543210321*\V\relax
```

```
macro:->{-131}{1177}{1}
```

```
U: macro:->-131, V: macro:->1177, D: macro:->1
```

```
AU+BV: 1
```

```
macro:->{256654313730}{-3208178892607}{3}
```

```
U: macro:->256654313730, V: macro:->-3208178892607, D: macro:->3
```

```
AU+BV: 3
```

source

### 14.2. `\xintEuclideanAlgorithm`

*Num Num f f* ★ `\xintEuclideanAlgorithm{N}{M}` applies the Euclidean algorithm and keeps a copy of all quotients and remainders.

```
\edef\X{\xintEuclideanAlgorithm {10000}{1113}}\meaning\X
macro:->{5}{10000}{1}{1113}{8}{1096}{1}{17}{64}{8}{2}{1}{8}{0}
```

The first item is the number of steps, the second is *N*, the third is the GCD, the fourth is *M* then the first quotient and remainder, the second quotient and remainder, ... until the final quotient and last (zero) remainder.

source

### 14.3. `\xintBezoutAlgorithm`

*Num Num f f* ★ `\xintBezoutAlgorithm{N}{M}` applies the Euclidean algorithm and keeps a copy of all quotients and remainders. Furthermore it computes the entries of the successive products of the 2 by 2 matrices  $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$  formed from the quotients arising in the algorithm.

```
\edef\X{\xintBezoutAlgorithm {10000}{1113}}\printnumber{\meaning\X}
macro:->{5}{10000}{0}{1}{1}{1113}{1}{0}{8}{1096}{8}{1}{1}{17}{9}{1}{64}{8}{584}{65}{2}{1}{1177}{131}{8}{0}{10000}{1113}
```

The first item is the number of steps, the second is **N**, then **0**, **1**, the GCD, **M**, **1**, **0**, the first quotient, the first remainder, the top left entry of the first matrix, the bottom left entry, and then these four things at each step until the end.

*source*

#### 14.4. \xintTypesetEuclideanAlgorithm

Num Num  
f f

This macro is just an example of how to organize the data returned by \xintEuclideanAlgorithm. Copy the source code to a new macro and modify it to what is needed.

```
\xintTypesetEuclideanAlgorithm {123456789012345}{9876543210321}
```

$$123456789012345 = 12 \times 9876543210321 + 4938270488493$$

$$9876543210321 = 2 \times 4938270488493 + 2233335$$

$$4938270488493 = 2211164 \times 2233335 + 536553$$

$$2233335 = 4 \times 536553 + 87123$$

$$536553 = 6 \times 87123 + 13815$$

$$87123 = 6 \times 13815 + 4233$$

$$13815 = 3 \times 4233 + 1116$$

$$4233 = 3 \times 1116 + 885$$

$$1116 = 1 \times 885 + 231$$

$$885 = 3 \times 231 + 192$$

$$231 = 1 \times 192 + 39$$

$$192 = 4 \times 39 + 36$$

$$39 = 1 \times 36 + 3$$

$$36 = 12 \times 3 + 0$$

*source*

#### 14.5. \xintTypesetBezoutAlgorithm

Num Num  
f f

This macro is just an example of how to organize the data returned by \xintBezoutAlgorithm. Copy the source code to a new macro and modify it to what is needed.

```
\xintTypesetBezoutAlgorithm {10000}{1113}
```

$$10000 = 8 \times 1113 + 1096$$

$$8 = 8 \times 1 + 0$$

$$1 = 8 \times 0 + 1$$

$$1113 = 1 \times 1096 + 17$$

$$9 = 1 \times 8 + 1$$

$$1 = 1 \times 1 + 0$$

$$1096 = 64 \times 17 + 8$$

$$584 = 64 \times 9 + 8$$

$$65 = 64 \times 1 + 1$$

$$17 = 2 \times 8 + 1$$

$$1177 = 2 \times 584 + 9$$

$$131 = 2 \times 65 + 1$$

$$8 = 8 \times 1 + 0$$

$$10000 = 8 \times 1177 + 584$$

$$1113 = 8 \times 131 + 65$$

$$131 \times 10000 - 1177 \times 1113 = -1$$



## 15. Macros of the *xintseries* package

.1	<code>\xintSeries</code> .....	200	.7	<code>\xintFxFtPowerSeries</code> .....	209
.2	<code>\xintiSeries</code> .....	201	.8	<code>\xintFxFtPowerSeriesX</code> .....	209
.3	<code>\xintRationalSeries</code> .....	202	.9	<code>\xintFloatPowerSeries</code> .....	211
.4	<code>\xintRationalSeriesX</code> .....	205	.10	<code>\xintFloatPowerSeriesX</code> .....	211
.5	<code>\xintPowerSeries</code> .....	207	.11	Computing $\log 2$ and $\pi$ .....	211
.6	<code>\xintPowerSeriesX</code> .....	208			

This package was first released with version 1.03 (2013/04/14) of the *xint bundle*.

The  $\overset{\text{Frac}}{f}$  expansion type of various macro arguments is only a  $\overset{\text{Num}}{f}$  if only *xint* but not *xintfrac* is loaded. The macro `\xintiSeries` is special and expects summing big integers obeying the strict format, even if *xintfrac* is loaded.

The arguments serving as indices are of the  $\overset{\text{num}}{x}$  expansion type.

In some cases one or two of the macro arguments are only expanded at a later stage not immediately.

Since 1.3, `\xintAdd` and `\xintSub` use systematically the least common multiple of the denominators. Some of the comments in this chapter refer to the earlier situation where often the denominators were simply multiplied together. *They have yet to be updated to reflect the new situation brought by the 1.3 release.* Some of these comments may now be off-synced from the actual computation results and thus may be wrong.

source

### 15.1. `\xintSeries`

`\xintSeries{A}{B}{\coeff}` computes  $\sum_{n=A}^{n=B} \coeff{n}$ . The initial and final indices must obey the  $\overset{\text{num}}{n}$   $\text{umexpr}$  constraint of expanding to numbers at most  $2^{31}-1$ . The `\coeff` macro must be a one-parameter *f-expandable* macro, taking on input an explicit number *n* and producing some number or fraction  $\overset{\text{num}}{c}$   $\overset{\text{Frac}}{f}$   $\star$  `\coeff{n}`; it is expanded at the time it is needed.

```
\def\coeff #1{\xintiMON{#1}/#1.5} % (-1)^n/(n+1/2)
\edef\w {\xintSeries {0}{50}{\coeff}} % we want to re-use it
\edef\z {\xintJrr {\w}[0]} % the [0] for a microsecond gain.
% \xintJrr preferred to \xintIrr: a big common factor is suspected.
% But numbers much bigger would be needed to show the greater efficiency.
\l[\sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} = \xintTeXFrac\z \]
```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = \frac{173909338287370940432112792101626602278714}{110027467159390003025279917226039729050575}$$

The definition of `\coeff` as `\xintiMON{#1}/#1.5` is quite suboptimal. It allows *#1* to be a big integer, but anyhow only small integers are accepted as initial and final indices (they are of the  $\overset{\text{num}}{x}$  type). Second, when the *xintfrac* parser sees the *#1.5* it will remove the dot hence create a denominator with one digit more. For example  $1/3.5$  turns internally into  $10/35$  whereas it would be more efficient to have  $2/7$ . For info here is the non-reduced `\w`:

$$\frac{86954669143685470216056396050813301139357}{550137335796950015126399586130198645252875} 10^1$$

It would have been bigger still in releases earlier than 1.1: now, the *xintfrac* `\xintAdd` routine does not multiply blindly denominators anymore, it checks if one is a multiple of the other. However it does not practice systematic reduction to lowest terms.

A more efficient way to code `\coeff` is illustrated next.

```
\def\coeff #1{\the\umexpr\ifodd #1 -2\else2\fi\relax/\the\umexpr 2*#1+1\relax [0]}%
```

```
% The [0] in \coeff is a tiny optimization: in its presence the \xintfracname parser
% sees something which is already in internal format.
\def\w {\xintSeries {0}{50}{\coeff}}
\[\sum_{n=0}^{\infty} \frac{(-1)^n}{n+\frac{1}{2}} = \xintTeXFrac{w\}
```

$$\sum_{n=0}^{50} \frac{(-1)^n}{n + \frac{1}{2}} = \frac{173909338287370940432112792101626602278714}{110027467159390003025279917226039729050575}$$

The reduced form `\z` as displayed above only differs from this one by a factor of 1.

```
\def\coeffleibnitz #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}
\cnta 1
\loop
% in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
\xintTrunc {12}{\xintSeries {1}{\cnta}{\coeffleibnitz}}\dots
\endgraf
\ifnum\cnta < 30 \advance\cnta 1 \repeat
```

1. 1.000000000000...	11. 0.736544011544...	21. 0.716390450794...
2. 0.500000000000...	12. 0.653210678210...	22. 0.670935905339...
3. 0.833333333333...	13. 0.730133755133...	23. 0.714414166209...
4. 0.583333333333...	14. 0.658705183705...	24. 0.672747499542...
5. 0.783333333333...	15. 0.725371850371...	25. 0.712747499542...
6. 0.616666666666...	16. 0.662871850371...	26. 0.674285961081...
7. 0.759523809523...	17. 0.721695379783...	27. 0.711322998118...
8. 0.634523809523...	18. 0.666139824228...	28. 0.675608712404...
9. 0.745634920634...	19. 0.718771403175...	29. 0.710091471024...
10. 0.645634920634...	20. 0.668771403175...	30. 0.676758137691...

*source*

## 15.2. `\xintiSeries`

$\sum_{n=A}^B f$  ★

`\xintiSeries{A}{B}{\coeff}` computes  $\sum_{n=A}^{n=B} \text{\coeff{n}}$  where `\coeff{n}` must *f-expand* to a (possibly long) integer in the strict format.

```
\def\coeff #1{\xintiTrunc {40}{\xintiMON{#1}/#1.5}}%
% better:
\def\coeff #1{\xintiTrunc {40}
  {\the\numexpr 2*\xintiMON{#1}\relax/\the\numexpr 2*#1+1\relax [0]}}%
% better still:
\def\coeff #1{\xintiTrunc {40}
  {\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}}%
% (-1)^n/(n+1/2) times 10^40, truncated to an integer.
\[\sum_{n=0}^{\infty} \frac{(-1)^n}{n+\frac{1}{2}} \approx
\xintTrunc {40}{\xintiSeries {0}{50}{\coeff}[-40]}\dots]
```

$$\sum_{n=0}^{50} \frac{(-1)^n}{n + \frac{1}{2}} \approx 1.5805993064935250412367895069567264144810$$

We should have cut out at least the last two digits: truncating errors originating with the first coefficients of the sum will never go away, and each truncation introduces an uncertainty in the last digit, so as we have 40 terms, we should trash the last two digits, or at least round at 38 digits. It is interesting to compare with the computation where rounding rather than truncation is used, and with the decimal expansion of the exactly computed partial sum of the series:

```
\def\coeff #1{\xintiRound {40} % rounding at 40
```

```

\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}\%
% (-1)^n/(n+1/2) times 10^40, rounded to an integer.
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}} \approx
\ \xintTrunc {40}{\xintiSeries {0}{50}{\coeff}{-40}}\}
\def\exactcoeff #1%
\the\numexpr\ifodd #1 -2\else2\fi\relax/\the\numexpr 2*#1+1\relax [0]}\%
\[ \sum_{n=0}^{n=50} \frac{(-1)^n}{n+\frac{1}{2}}
= \xintTrunc {50}{\xintiSeries {0}{50}{\exactcoeff}}\dots\}

```

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} \approx 1.5805993064935250412367895069567264144804$$

$$\sum_{n=0}^{n=50} \frac{(-1)^n}{n + \frac{1}{2}} = 1.58059930649352504123678950695672641448068680288367 \dots$$

This shows indeed that our sum of truncated terms estimated wrongly the 39th and 40th digits of the exact result<sup>72</sup> and that the sum of rounded terms fared a bit better.

*source*

### 15.3. `\xintRationalSeries`

`\xintRationalSeries{A}{B}{f}{\ratio}` evaluates  $\sum_{n=A}^{n=B} F(n)$ , where  $F(n)$  is specified indirectly via the data of  $f=F(A)$  and the one-parameter macro `\ratio` which must be such that `\macro{n}` expands to  $F(n)/F(n-1)$ . The name indicates that `\xintRationalSeries` was designed to be useful in the cases where  $F(n)/F(n-1)$  is a rational function of  $n$  but it may be anything expanding to a fraction. The macro `\ratio` must be an expandable-only compatible macro and expand to its value after iterated full expansion of its first item.  $A$  and  $B$  are fed to a `\numexpr` hence may be count registers or arithmetic expressions built with such; they must obey the  $\TeX$  bound. The initial term  $f$  may be a macro `\f`, it will be expanded to its value representing  $F(A)$ .

```

\def\ratio #1{2/#1[0]}\% 2/n, to compute exp(2)
\cnta 0 % previously declared count
\begin{quote}
\loop \fdef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{2^n}{n!}=
\xintTrunc{12}\z\dots=
\xintTeXFrac\z=\xintTeXFrac{\xintIrr\z}\$\vtop to 5pt{}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}

```

$$\sum_{n=0}^0 \frac{2^n}{n!} = 1.000000000000 \dots = 1 = 1$$

$$\sum_{n=0}^1 \frac{2^n}{n!} = 3.000000000000 \dots = 3 = 3$$

$$\sum_{n=0}^2 \frac{2^n}{n!} = 5.000000000000 \dots = \frac{10}{2} = 5$$

$$\sum_{n=0}^3 \frac{2^n}{n!} = 6.333333333333 \dots = \frac{38}{6} = \frac{19}{3}$$

$$\sum_{n=0}^4 \frac{2^n}{n!} = 7.000000000000 \dots = \frac{168}{24} = 7$$

$$\sum_{n=0}^5 \frac{2^n}{n!} = 7.266666666666 \dots = \frac{872}{120} = \frac{109}{15}$$

$$\sum_{n=0}^6 \frac{2^n}{n!} = 7.355555555555 \dots = \frac{5296}{720} = \frac{331}{45}$$

$$\sum_{n=0}^7 \frac{2^n}{n!} = 7.380952380952 \dots = \frac{37200}{5040} = \frac{155}{21}$$

$$\sum_{n=0}^8 \frac{2^n}{n!} = 7.387301587301 \dots = \frac{297856}{40320} = \frac{2327}{315}$$

<sup>72</sup> as the series is alternating, we can roughly expect an error of  $\sqrt{40}$  and the last two digits are off by 4 units, which is not contradictory to our expectations.

$$\begin{aligned}
\sum_{n=0}^9 \frac{2^n}{n!} &= 7.388712522045 \dots = \frac{2681216}{362880} = \frac{20947}{2835} \\
\sum_{n=0}^{10} \frac{2^n}{n!} &= 7.388994708994 \dots = \frac{26813184}{3628800} = \frac{34913}{4725} \\
\sum_{n=0}^{11} \frac{2^n}{n!} &= 7.389046015712 \dots = \frac{294947072}{39916800} = \frac{164591}{22275} \\
\sum_{n=0}^{12} \frac{2^n}{n!} &= 7.389054566832 \dots = \frac{3539368960}{479001600} = \frac{691283}{93555} \\
\sum_{n=0}^{13} \frac{2^n}{n!} &= 7.389055882389 \dots = \frac{46011804672}{6227020800} = \frac{14977801}{2027025} \\
\sum_{n=0}^{14} \frac{2^n}{n!} &= 7.389056070325 \dots = \frac{644165281792}{87178291200} = \frac{314533829}{42567525} \\
\sum_{n=0}^{15} \frac{2^n}{n!} &= 7.389056095384 \dots = \frac{9662479259648}{1307674368000} = \frac{4718007451}{638512875} \\
\sum_{n=0}^{16} \frac{2^n}{n!} &= 7.389056098516 \dots = \frac{154599668219904}{20922789888000} = \frac{1572669151}{212837625} \\
\sum_{n=0}^{17} \frac{2^n}{n!} &= 7.389056098884 \dots = \frac{2628194359869440}{355687428096000} = \frac{16041225341}{2170943775} \\
\sum_{n=0}^{18} \frac{2^n}{n!} &= 7.389056098925 \dots = \frac{47307498477912064}{6402373705728000} = \frac{103122162907}{13956067125} \\
\sum_{n=0}^{19} \frac{2^n}{n!} &= 7.389056098930 \dots = \frac{898842471080853504}{121645100408832000} = \frac{4571749222213}{618718975875} \\
\sum_{n=0}^{20} \frac{2^n}{n!} &= 7.389056098930 \dots = \frac{17976849421618118656}{2432902008176640000} = \frac{68576238333199}{9280784638125}
\end{aligned}$$

```

\def\ratio #1{-1/#1[0]}% -1/n, comes from the series of exp(-1)
\cnta 0 % previously declared count
\begin{quote}
\loop
\fddef\z {\xintRationalSeries {0}{\cnta}{1}{\ratio }}%
\noindent$\sum_{n=0}^{\the\cnta} \frac{(-1)^n}{n!}=
\xintTrunc{20}\z\dots=\xintTeXFrac{\z}=\xintTeXFrac{\xintIrr\z}$%
\vtop to 5pt{}\par
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{quote}

```

$$\begin{aligned}
\sum_{n=0}^0 \frac{(-1)^n}{n!} &= 1.00000000000000000000 \dots = 1 = 1 \\
\sum_{n=0}^1 \frac{(-1)^n}{n!} &= 0 \dots = 0 = 0 \\
\sum_{n=0}^2 \frac{(-1)^n}{n!} &= 0.50000000000000000000 \dots = \frac{1}{2} = \frac{1}{2} \\
\sum_{n=0}^3 \frac{(-1)^n}{n!} &= 0.33333333333333333333 \dots = \frac{2}{6} = \frac{1}{3} \\
\sum_{n=0}^4 \frac{(-1)^n}{n!} &= 0.37500000000000000000 \dots = \frac{9}{24} = \frac{3}{8} \\
\sum_{n=0}^5 \frac{(-1)^n}{n!} &= 0.36666666666666666666 \dots = \frac{44}{120} = \frac{11}{30} \\
\sum_{n=0}^6 \frac{(-1)^n}{n!} &= 0.36805555555555555555 \dots = \frac{265}{720} = \frac{53}{144} \\
\sum_{n=0}^7 \frac{(-1)^n}{n!} &= 0.36785714285714285714 \dots = \frac{1854}{5040} = \frac{103}{280} \\
\sum_{n=0}^8 \frac{(-1)^n}{n!} &= 0.36788194444444444444 \dots = \frac{14833}{40320} = \frac{2119}{5760} \\
\sum_{n=0}^9 \frac{(-1)^n}{n!} &= 0.36787918871252204585 \dots = \frac{133496}{362880} = \frac{16687}{45360} \\
\sum_{n=0}^{10} \frac{(-1)^n}{n!} &= 0.36787946428571428571 \dots = \frac{1334961}{3628800} = \frac{16481}{44800} \\
\sum_{n=0}^{11} \frac{(-1)^n}{n!} &= 0.36787943923360590027 \dots = \frac{14684570}{39916800} = \frac{1468457}{3991680} \\
\sum_{n=0}^{12} \frac{(-1)^n}{n!} &= 0.36787944132128159905 \dots = \frac{176214841}{479001600} = \frac{16019531}{43545600} \\
\sum_{n=0}^{13} \frac{(-1)^n}{n!} &= 0.36787944116069116069 \dots = \frac{2290792932}{6227020800} = \frac{63633137}{172972800}
\end{aligned}$$



```
\xintTrunc{8}{\xintDiv{z}{w}\dots$ \vtop to 5pt}}\endgraf
\ifnum\cnta<20 \advance\cnta 1 \repeat
\end{multicols}
```

$$\begin{array}{ll} \sum_{n=1}^1 \frac{1^n}{n!} / \sum_{n=0}^1 \frac{1^n}{n!} = 0.50000000 \dots & \sum_{n=11}^{21} \frac{11^n}{n!} / \sum_{n=0}^{21} \frac{11^n}{n!} = 0.53907332 \dots \\ \sum_{n=2}^3 \frac{2^n}{n!} / \sum_{n=0}^3 \frac{2^n}{n!} = 0.52631578 \dots & \sum_{n=12}^{23} \frac{12^n}{n!} / \sum_{n=0}^{23} \frac{12^n}{n!} = 0.53772178 \dots \\ \sum_{n=3}^5 \frac{3^n}{n!} / \sum_{n=0}^5 \frac{3^n}{n!} = 0.53804347 \dots & \sum_{n=13}^{25} \frac{13^n}{n!} / \sum_{n=0}^{25} \frac{13^n}{n!} = 0.53644744 \dots \\ \sum_{n=4}^7 \frac{4^n}{n!} / \sum_{n=0}^7 \frac{4^n}{n!} = 0.54317053 \dots & \sum_{n=14}^{27} \frac{14^n}{n!} / \sum_{n=0}^{27} \frac{14^n}{n!} = 0.53525726 \dots \\ \sum_{n=5}^9 \frac{5^n}{n!} / \sum_{n=0}^9 \frac{5^n}{n!} = 0.54502576 \dots & \sum_{n=15}^{29} \frac{15^n}{n!} / \sum_{n=0}^{29} \frac{15^n}{n!} = 0.53415135 \dots \\ \sum_{n=6}^{11} \frac{6^n}{n!} / \sum_{n=0}^{11} \frac{6^n}{n!} = 0.54518217 \dots & \sum_{n=16}^{31} \frac{16^n}{n!} / \sum_{n=0}^{31} \frac{16^n}{n!} = 0.53312615 \dots \\ \sum_{n=7}^{13} \frac{7^n}{n!} / \sum_{n=0}^{13} \frac{7^n}{n!} = 0.54445274 \dots & \sum_{n=17}^{33} \frac{17^n}{n!} / \sum_{n=0}^{33} \frac{17^n}{n!} = 0.53217628 \dots \\ \sum_{n=8}^{15} \frac{8^n}{n!} / \sum_{n=0}^{15} \frac{8^n}{n!} = 0.54327992 \dots & \sum_{n=18}^{35} \frac{18^n}{n!} / \sum_{n=0}^{35} \frac{18^n}{n!} = 0.53129566 \dots \\ \sum_{n=9}^{17} \frac{9^n}{n!} / \sum_{n=0}^{17} \frac{9^n}{n!} = 0.54191055 \dots & \sum_{n=19}^{37} \frac{19^n}{n!} / \sum_{n=0}^{37} \frac{19^n}{n!} = 0.53047810 \dots \\ \sum_{n=10}^{19} \frac{10^n}{n!} / \sum_{n=0}^{19} \frac{10^n}{n!} = 0.54048295 \dots & \sum_{n=20}^{39} \frac{20^n}{n!} / \sum_{n=0}^{39} \frac{20^n}{n!} = 0.52971771 \dots \end{array}$$

source

## 15.4. \xintRationalSeriesX

$\sum \frac{f}{f} \frac{f}{f} \frac{f}{f} \star$  `\xintRationalSeriesX{A}{B}{\first}{\ratio}{\g}` is a parametrized version of `\xintRationalSeries` where `\first` is now a one-parameter macro such that `\first{\g}` gives the initial term and `\ratio` is a two-parameter macro such that `\ratio{n}{\g}` represents the ratio of one term to the previous one. The parameter `\g` is evaluated only once at the beginning of the computation, and can thus itself be the yet unevaluated result of a previous computation.

Let `\ratio` be such a two-parameter macro; note the subtle differences between

```
\xintRationalSeries {A}{B}{\first}{\ratio{\g}}
```

```
and \xintRationalSeriesX {A}{B}{\first}{\ratio}{\g}.
```

First the location of braces differ... then, in the former case `\first` is a *no-parameter* macro expanding to a fractional number, and in the latter, it is a *one-parameter* macro which will use `\g`. Furthermore the **X** variant will expand `\g` at the very beginning whereas the former non-**X** former variant will evaluate it each time it needs it (which is bad if this evaluation is time-costly, but good if `\g` is a big explicit fraction encapsulated in a macro).

The example will use the macro `\xintPowerSeries` which computes efficiently exact partial sums of power series, and is discussed in the next section.

```
\def\firstterm #1{1[0]}% first term of the exponential series
% although it is the constant 1, here it must be defined as a
% one-parameter macro. Next comes the ratio function for exp:
\def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the (-1)^{n-1}/n of the log(1+h) series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the log(1+h) series and
% let E(t) be the first 10 terms of the exp(t) series.
% The following computes E(L(a/10)) for a=1,...,12.
\begin{multicols}{3}\raggedcolumns
\cnta 0
\loop
\noindent\xintTrunc {18}{%
\xintRationalSeriesX {0}{9}{\firstterm}{\ratioexp}
{\xintPowerSeries{1}{10}{\coefflog}{\the\cnta[-1]}}}\dots
}\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
```

```
\end{multicols}
```

```
1.00000000000000000000...      1.499954310225476533...      1.907197560339468199...
1.099999999999083906...      1.599659266069210466...      1.845117565491393752...
1.199999998111624029...      1.698137473697423757...      1.593831932293536053...
1.299999835744121464...      1.791898112718884531...
1.399996091955359088...      1.870485649686617459...
```

These completely exact operations rapidly create numbers with many digits. Let us print in full the raw fractions created by the operation illustrated above:

```
E(L(1[-1]))=163591443693117889303431088806087634148250735791023497657261314014159107395739
0639913787199465741057336677116573252341295218688/1487194942665946638644674560000000000[-90]
(length of numerator: 127)
E(L(12[-2]))=16656583357757234467643895619026874191327320993157183247568125775059356018362
23193439604540053754226444871502834816644808336288211299845887246066795041160882231219805166
927273729660728412213074817261522841754729971712/1487194942665946638644674560000000000[-180]
(length of numerator: 217)
E(L(123[-3]))=1670119920600555026998663239069002278266215966968669508145191626887938734862
73269986546078658803979014003116903378025935148900448814698936627633558066738151958530603167
40612785673175692992742863679398303407413205084692383474722719804622771982161117197045873620
25769049115687215712723182386527055033735053312/1487194942665946638644674560000000000[-270] (length
of numerator: 307)
```

We see that the denominators here remain the same, as our input only had various powers of ten as denominators, and *xintfrac* efficiently assemble (some only, as we can see) powers of ten. Notice that 1 more digit in an input denominator seems to mean 90 more in the raw output. We can check that with some other test cases:

```
E(L(1/7))=48228203862750885848048032297655163193719083498752126622944863683478921463537652
0421966954177876452794933/421996783816227187824437252776031227863306380633210580813174165609
500569367213288120561612881920000000000[0] (length of numerator: 105; length of denominator:
105)
E(L(1/71))=6190039670785350346406550995159476540272948182884398462882888922997238323197498
85971940015218249059435720836832839237391067287499316324605873244670430502854291696282116287
58603878135499973539887212860467/61040668975799982582643863990035761895246771100033570420271
67109220333298498184289107451083577982695694446256675834390041749715017225626389830761170775
7919998778523559418340083473473151235522560000000000[0] (length of numerator: 203; length of
denominator: 203)
E(L(1/712))=300356435377840602055967040841188592538909311419930838799656013626071029784174
49681929088495804136203813242174405561415315426829241317287053037273453329055814153891517325
75694112320026364569495366534918031439051104610487529796192058205725999641657806615904929048
98946463533146662233869249/29993517810522090976696848959176310536177550755703969736435921535
22460410892328532539738041911202121412424715881734049254716640082470987340985151932504281494
24064596788874441470533147848207863549778847000617103264666638782677019019130113930837421531
810478062025966102914017525760000000000[0] (length of numerator: 288; length of denominator:
288)
```

Thus decimal numbers such as *0.123* (equivalently *123[-3]*) give less computing intensive tasks than fractions such as *1/712*: in the case of decimal numbers the (raw) denominators originate in the coefficients of the series themselves, powers of ten of the input within brackets being treated separately. And even then the numerators will grow with the size of the input in a sort of linear way, the coefficient being given by the order of series: here 10 from the log and 9 from the exp, so 90. One more digit in the input means 90 more digits in the numerator of the output: obviously we can not go on composing such partial sums of series and hope that *xint* will joyfully do all at the speed of light!



Hence, truncating the output (or better, rounding) is the only way to go if one needs a general calculus of special functions. This is why the package *xintseries* provides, besides `\xintSeries`, `\xintRationalSeries`, or `\xintPowerSeries` which compute exact sums, `\xintFxFtPowerSeries` for fixed-point computations and a (tentative naive) `\xintFloatPowerSeries`.

*source*

## 15.5. `\xintPowerSeries`

$\sum_{n=A}^B \frac{\text{num}}{X} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f} \star$  `\xintPowerSeries{A}{B}{\coeff}{f}` evaluates the sum  $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$ . The initial and final indices are given to a `\numexpr` expression. The `\coeff` macro (which, as argument to `\xintPowerSeries` is expanded only at the time `\coeff{n}` is needed) should be defined as a one-parameter expandable macro, its input will be an explicit number.

The `f` can be either a fraction directly input or a macro `\f` expanding to such a fraction. It is actually more efficient to encapsulate an explicit fraction `f` in such a macro, if it has big numerators and denominators ('big' means hundreds of digits) as it will then take less space in the processing until being (repeatedly) used.

This macro computes the exact result (one can use it also for polynomial evaluation), using a Horner scheme which helps avoiding a denominator build-up (this problem however, even if using a naive additive approach, is much less acute since release 1.1 and its new policy regarding `\xintAdd`).

```
\def\geom #1{1[0]} % the geometric series
\def\f {5/17[0]}
\[ \sum_{n=0}^{n=20} \Bigl(\frac{5}{17}\Bigr)^n
=\xintTeXFrac{\xintIrr{\xintPowerSeries {0}{20}{\geom}{\f}}}
=\xintTeXFrac{\xinttheexpr (17^21-5^21)/12/17^20\relax}\]
```

$$\sum_{n=0}^{n=20} \left(\frac{5}{17}\right)^n = \frac{5757661159377657976885341}{4064231406647572522401601} = \frac{69091933912531895722624092}{48770776879770870268819212}$$

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\f {1/2[0]}%
\[ \log 2 \approx \sum_{n=1}^{n=20} \frac{1}{n \cdot 2^n}
=\xintTeXFrac {\xintIrr {\xintPowerSeries {1}{20}{\coefflog}{\f}}}
\[ \log 2 \approx \sum_{n=1}^{n=50} \frac{1}{n \cdot 2^n}
=\xintTeXFrac {\xintIrr {\xintPowerSeries {1}{50}{\coefflog}{\f}}}\]
```

$$\log 2 \approx \sum_{n=1}^{20} \frac{1}{n \cdot 2^n} = \frac{42299423848079}{61025172848640}$$

$$\log 2 \approx \sum_{n=1}^{50} \frac{1}{n \cdot 2^n} = \frac{60463469751752265663579884559739219}{87230347965792839223946208178339840}$$

```
\setlength{\columnsep}{0pt}
\begin{multicols}{3}
\cnta 1 % previously declared count
\loop % in this loop we recompute from scratch each partial sum!
% we can afford that, as \xintPowerSeries is fast enough.
\noindent\hbox to 2em{\hfil\texttt{\the\cnta.} }%
\xintTrunc {12}
{\xintPowerSeries {1}{\cnta}{\coefflog}{\f}}\dots
\endgraf
\ifnum \cnta < 30 \advance\cnta 1 \repeat
\end{multicols}
```



[TOC](#), [xint bundle](#), [xintkernel](#), [xintcore](#), [xint](#), [xintfrac](#), [xintbinhex](#), [xintgcd](#), [xintseries](#), [xintcfrac](#)

1. 0.500000000000...	11. 0.693109245355...	21. 0.693147159757...
2. 0.625000000000...	12. 0.693129590407...	22. 0.693147170594...
3. 0.666666666666...	13. 0.693138980431...	23. 0.693147175777...
4. 0.682291666666...	14. 0.693143340085...	24. 0.693147178261...
5. 0.688541666666...	15. 0.693145374590...	25. 0.693147179453...
6. 0.691145833333...	16. 0.693146328265...	26. 0.693147180026...
7. 0.692261904761...	17. 0.693146777052...	27. 0.693147180302...
8. 0.692750186011...	18. 0.693146988980...	28. 0.693147180435...
9. 0.692967199900...	19. 0.693147089367...	29. 0.693147180499...
10. 0.693064856150...	20. 0.693147137051...	30. 0.693147180530...

```

\def\coeffarctg #1{1/\the\numexpr\ifodd #1 -2*#1-1\else2*#1+1\fi\relax }%
% the above gives (-1)^n/(2n+1). The sign being in the denominator,
%
%      **** no [0] should be added ****,
% else nothing is guaranteed to work (even if it could by sheer luck)
% Notice in passing this aspect of \numexpr:
%
%      **** \numexpr -1\relax is illegal !!! ****
\def\ f {1/25[0]}% 1/5^2
\[\mathrm{Arctg}(\frac{15}{25})\approx \frac{15}{25}\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)25^n}
= \xintTeXFrac{\xintIrr}{\xintDiv}{\xintPowerSeries}{0}{15}{\coeffarctg}{f}{5}{}

```

$$\text{Arctg}\left(\frac{1}{5}\right) \approx \frac{1}{5} \sum_{n=0}^{15} \frac{(-1)^n}{(2n+1)25^n} = \frac{165918726519122955895391793269168}{840539304153062403202056884765625}$$

*source*

$$\frac{\text{num}}{X} \frac{\text{num}}{X} \frac{\text{Frac}}{f} \frac{\text{Frac}}{f}$$

This is the same as `\xintPowerSeries` apart from the fact that the last parameter `f` is expanded once and for all before being then used repeatedly. If the `f` parameter is to be an explicit big fraction with many (dozens) digits, rather than using it directly it is slightly better to have some macro `\g` defined to expand to the explicit fraction and then use `\xintPowerSeries` with `\g`; but if `f` has not yet been evaluated and will be the output of a complicated expansion of some `\f`, and if, due to an expanding only context, doing `\edef\g{\f}` is no option, then `\xintPowerSeriesX` should be used with `\f` as last parameter.

```
% \def\ratioexp #1#2{\xintDiv {#1}{#2}}% x/n
% These are the  $(-1)^{n-1}/n$  of the  $\log(1+h)$  series:
\def\coefflog #1{\the\numexpr\ifodd #1 1\else-1\fi\relax/#1[0]}%
% Let L(h) be the first 10 terms of the  $\log(1+h)$  series and
% let E(t) be the first 10 terms of the  $\exp(t)$  series.
% The following computes  $L(E(a/10)-1)$  for  $a=1, \dots, 12$ .
\begin{multicols}{3}\raggedcolumns
\cnta 1
\loop
\noindent\xintTrunc {18}{%
  \xintPowerSeriesX {1}{10}{\coefflog}
  {\xintSub
    {\xintRationalSeries {0}{9}{1[0]}\ratioexp{\the\cnta[-1]}}
    {1}}}\dots
\endgraf
\ifnum\cnta < 12 \advance \cnta 1 \repeat
\end{multicols}
```

0.099999999998556159...	0.499511320760604148...	-1.597091692317639401...
0.199999995263443554...	0.593980619762352217...	-12.648937932093322763...
0.299999338075041781...	0.645144282733914916...	-66.259639046914679687...
0.399974460740121112...	0.398118280111436442...	-304.768437445462801227...

source

15.7. `\xintFxFtPowerSeries`

$$\frac{\frac{\frac{\text{num}}{f} \frac{\text{num}}{f}}{\frac{\text{num}}{f} \frac{\text{num}}{f}} \frac{\text{num}}{f} \frac{\text{num}}{f}} \star$$

`\xintFxFtPowerSeries{A}{B}{\coeff}{f}{D}` computes  $\sum_{n=A}^{n=B} \text{\coeff}\{n\} \cdot f^n$  with each term of the series truncated to *D* digits after the decimal point. As usual, *A* and *B* are completely expanded through their inclusion in a `\numexpr` expression. Regarding *D* it will be similarly be expanded each time it is used inside an `\xintTrunc`. The one-parameter macro `\coeff` is similarly expanded at the time it is used inside the computations. Idem for *f*. If *f* itself is some complicated macro it is thus better to use the variant `\xintFxFtPowerSeriesX` which expands it first and then uses the result of that expansion.

The current (1.04) implementation is: the first power  $f^A$  is computed exactly, then *truncated*. Then each successive power is obtained from the previous one by multiplication by the exact value of *f*, and truncated. And  $\text{\coeff}\{n\} \cdot f^n$  is obtained from that by multiplying by `\coeff{n}` (untruncated) and then truncating. Finally the sum is computed exactly. Apart from that `\xintFxFtPowerSeries` (where *FxFt* means 'fixed-point') is like `\xintPowerSeries`.

There should be a variant for things of the type  $\sum c_n \frac{f^n}{n!}$  to avoid having to compute the factorial from scratch at each coefficient, the same way `\xintFxFtPowerSeries` does not compute  $f^n$  from scratch at each *n*. Perhaps in the next package release.

$$e^{-\frac{1}{2}} \approx$$

1.00000000000000000000	0.60653056795634920635	0.60653065971263344622
0.50000000000000000000	0.60653066483754960317	0.60653065971263342289
0.62500000000000000000	0.60653065945526069224	0.60653065971263342361
0.60416666666666666667	0.60653065972437513778	0.60653065971263342359
0.60677083333333333333	0.60653065971214266299	0.60653065971263342359
0.60651041666666666667	0.60653065971265234943	0.60653065971263342359
0.60653211805555555555	0.60653065971263274611	

```
\def\coeffexp #1{1/\xintiiFac {#1}[0]}% 1/n!
\def\ f {-1/2[0]}% [0] for faster input parsing
\cnta 0 % previously declared \count register
\noindent\loop
$\xintFxFtPowerSeries {0}{\cnta}{\coeffexp}{\f}{20}$\
\ifnum\cnta<19 \advance\cnta 1 \repeat\par
\xintFxFtPowerSeries {0}{19}{\coeffexp}{\f}{25}= 0.6065306597126334236037992
```

It is no difficulty for `xintfrac` to compute exactly, with the help of `\xintPowerSeries`, the nineteenth partial sum, and to then give (the start of) its exact decimal expansion:

$$\begin{aligned} \text{\xintPowerSeries {0}{19}{\coeffexp}{\f}} &= \frac{38682746160036397317757}{63777066403145711616000} \\ &= 0.606530659712633423603799152126 \dots \end{aligned}$$

Thus, one should always estimate a priori how many ending digits are not reliable: if there are *N* terms and *N* has *k* digits, then digits up to but excluding the last *k* may usually be trusted. If we are optimistic and the series is alternating we may even replace *N* with  $\sqrt{N}$  to get the number *k* of digits possibly of dubious significance.

source

15.8. `\xintFxFtPowerSeriesX`

$$\frac{\frac{\frac{\text{num}}{f} \frac{\text{num}}{f}}{\frac{\text{num}}{f} \frac{\text{num}}{f}} \frac{\text{num}}{f} \frac{\text{num}}{f}} \star$$

`\xintFxFtPowerSeriesX{A}{B}{\coeff}{f}{D}` computes, exactly as `\xintFxFtPowerSeries`, the sum of  $\text{\coeff}\{n\} \cdot f^n$  from *n=A* to *n=B* with each term of the series being *truncated* to *D* digits after

the decimal point. The sole difference is that `\f` is first expanded and it is the result of this which is used in the computations.

Let us illustrate this on the numerical exploration of the identity

$$\log(1+x) = -\log(1/(1+x))$$

Let  $L(h)=\log(1+h)$ , and  $D(h)=L(h)+L(-h/(1+h))$ . Theoretically thus,  $D(h)=0$  but we shall evaluate  $L(h)$  and  $-h/(1+h)$  keeping only 10 terms of their respective series. We will assume  $h < 0.5$ . With only ten terms kept in the power series we do not have quite 3 digits precision as  $2^{10} = 1024$ . So it wouldn't make sense to evaluate things more precisely than, say circa 5 digits after the decimal points.

```
\cnta 0
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}% (-1)^{n-1}/n
\def\coeffalt #1{\the\numexpr\ifodd#1 -1\else1\fi\relax [0]}% (-1)^n
\begin{multicols}2
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\xintAdd {\xintFxFtPowerSeriesX {1}{10}{\coefflog}{\the\cnta [-2]}\{5}}
{\xintFxFtPowerSeriesX {1}{10}{\coefflog}
{\xintFxFtPowerSeriesX {1}{10}{\coeffalt}{\the\cnta [-2]}\{5}}
{5}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
\end{multicols}
```

D(0/100): 0/1[0]

D(7/100): 2/1[-5]

D(14/100): 2/1[-5]

D(21/100): 3/1[-5]

D(28/100): 4/1[-5]

D(35/100): 4/1[-5]

D(42/100): 9/1[-5]

D(49/100): 42/1[-5]

Let's say we evaluate functions on  $[-1/2, +1/2]$  with values more or less also in  $[-1/2, +1/2]$  and we want to keep 4 digits of precision. So, roughly we need at least 14 terms in series like the geometric or log series. Let's make this 15. Then it doesn't make sense to compute intermediate summands with more than 6 digits precision. So we compute with 6 digits precision but return only 4 digits (rounded) after the decimal point. This result with 4 post-decimal points precision is then used as input to the next evaluation.

```
\begin{multicols}2
\loop
\noindent \hbox to 2.5cm {\hss\texttt{D(\the\cnta/100): }}%
\dt{\xintRound{4}
{\xintAdd {\xintFxFtPowerSeriesX {1}{15}{\coefflog}{\the\cnta [-2]}\{6}}
{\xintFxFtPowerSeriesX {1}{15}{\coefflog}
{\xintRound {4}{\xintFxFtPowerSeriesX {1}{15}{\coeffalt}
{\the\cnta [-2]}\{6}}
{6}}}%
}}\endgraf
\ifnum\cnta < 49 \advance\cnta 7 \repeat
\end{multicols}
```

D(0/100): 0

D(7/100): 0.0000

D(14/100): 0.0000

D(21/100): -0.0001

D(28/100): -0.0001

D(35/100): -0.0001

D(42/100): -0.0000

D(49/100): -0.0001

Not bad... I have cheated a bit: the 'four-digits precise' numeric evaluations were left unrounded in the final addition. However the inner rounding to four digits worked fine and made the

next step faster than it would have been with longer inputs. The morale is that one should not use the raw results of `\xintFxpPowerSeriesX` with the `D` digits with which it was computed, as the last are to be considered garbage. Rather, one should keep from the output only some smaller number of digits. This will make further computations faster and not less precise. I guess there should be some macro to do this final truncating, or better, rounding, at a given number `D' < D` of digits. Maybe for the next release.

[source](#)

## 15.9. `\xintFloatPowerSeries`

$\left[ \begin{smallmatrix} \text{num} & \text{num} & \text{num} \\ \text{Frac} & \text{Frac} & \text{Frac} \\ f & f & f \end{smallmatrix} \right] \star$

`\xintFloatPowerSeries[P]{A}{B}{\coeff}{f}` computes  $\sum_{n=A}^{n=B} \text{coeff}\{n\} \cdot f^n$  with a floating point precision given by the optional parameter `P` or by the current setting of `\xintDigits`.

In the current, preliminary, version, no attempt has been made to try to guarantee to the final result the precision `P`. Rather, `P` is used for all intermediate floating point evaluations. So rounding errors will make some of the last printed digits invalid. The operations done are first the evaluation of `f^A` using `\xintFloatPow`, then each successive power is obtained from this first one by multiplication by `f` using `\xintFloatMul`, then again with `\xintFloatMul` this is multiplied with `\coeff{n}`, and the sum is done adding one term at a time with `\xintFloatAdd`. To sum up, this is just the naive transformation of `\xintFxpPowerSeries` from fixed point to floating point.

```
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}
-6.9314718e-1
```

[source](#)

## 15.10. `\xintFloatPowerSeriesX`

$\left[ \begin{smallmatrix} \text{num} & \text{num} & \text{num} \\ \text{Frac} & \text{Frac} & \text{Frac} \\ f & f & f \end{smallmatrix} \right] \star$

`\xintFloatPowerSeriesX[P]{A}{B}{\coeff}{f}` is like `\xintFloatPowerSeries` with the difference that `f` is expanded once and for all at the start of the computation, thus allowing efficient chaining of such series evaluations.

```
\def\coeffexp #1{1/\xintiiFac {#1}[0]}% 1/n! (exact, not float)
\def\coefflog #1{\the\numexpr\ifodd#1 1\else-1\fi\relax/#1[0]}%
\xintFloatPowerSeriesX [8]{0}{30}{\coeffexp}
{\xintFloatPowerSeries [8]{1}{30}{\coefflog}{-1/2[0]}}
5.00000001e-1
```

## 15.11. Computing $\log 2$ and $\pi$

In this final section, the use of `\xintFxpPowerSeries` (and `\xintPowerSeries`) will be illustrated on the (expandable... why make things simple when it is so easy to make them difficult!) computations of the first digits of the decimal expansion of the familiar constants  $\log 2$  and  $\pi$ .

Let us start with  $\log 2$ . We will get it from this formula (which is left as an exercise):

```
log(2)=-2 log(1-13/256)-5 log(1-1/9)
```

The number of terms to be kept in the log series, for a desired precision of  $10^{-D}$  was roughly estimated without much theoretical analysis. Computing exactly the partial sums with `\xintPowerSeries` and then printing the truncated values, from `D=0` up to `D=100` showed that it worked in terms of quality of the approximation. Because of possible strings of zeroes or nines in the exact decimal expansion (in the present case of  $\log 2$ , strings of zeroes around the fortieth and the sixtieth decimals), this does not mean though that all digits printed were always exact. In the end one always end up having to compute at some higher level of desired precision to validate the earlier result.

Then we tried with `\xintFxpPowerSeries`: this is worthwhile only for `D`'s at least 50, as the exact evaluations are faster (with these short-length `f`'s) for a lower number of digits. And as expected the degradation in the quality of approximation was in this range of the order of two or three digits. This meant roughly that the 3+1=4 ending digits were wrong. Again, we ended up having to compute with five more digits and compare with the earlier value to validate it. We

use truncation rather than rounding because our goal is not to obtain the correct rounded decimal expansion but the correct exact truncated one.

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\xa {13/256[0]}% we will compute log(1-13/256)
\def\xb {1/9[0]}% we will compute log(1-1/9)
\def\LogTwo #1{%
  % get log(2)=-2log(1-13/256)- 5log(1-1/9)
  \romannumeral0\expandafter\logtwo
  % number of terms for 1/9:
  \the\numexpr #1*150/143\expandafter.%
  % number of Terms for 13/256:
  \the\numexpr #1*100/129\expandafter.%
  % We print #1 digits, but we know the ending ones are garbage
  % Use \numexpr to allow a \count as #1
  \the\numexpr #1.%
}%
\def\logtwo #1.#2.#3.{%
  % #1=nb of terms for 1/9,
  % #2=nb of terms for 13/256,
  % #3=nb of digits for computations, also used for printing
  \xinttrunc {#3}% will terminate the \romannumeral0
  {\xintAdd
    {\xintMul {2}{\xintFxFtPowerSeries {1}{#2}{\coefflog}{\xa}{#3}}}
    {\xintMul {5}{\xintFxFtPowerSeries {1}{#1}{\coefflog}{\xb}{#3}}}%
  }%
}%
\noindent $\log 2 \approx \LogTwo {60}\dots$\endgraf
\noindent\phantom{$\log 2$}\$\approx{}\$\printnumber{\LogTwo {65}}\dots\endgraf
\noindent\phantom{$\log 2$}\$\approx{}\$\printnumber{\LogTwo {70}}\dots\endgraf
```

$\log 2 \approx 0.693147180559945309417232121458176568075500134360255254120484\dots$

$\approx 0.69314718055994530941723212145817656807550013436025525412068000711\dots$

$\approx 0.6931471805599453094172321214581765680755001343602552541206800094933723\dots$

Here is the code doing an exact evaluation of the partial sums. We have added a +1 to the number of digits for estimating the number of terms to keep from the log series: we experimented that this gets exactly the first  $D$  digits, for all values from  $D=0$  to  $D=100$ , except in one case ( $D=40$ ) where the last digit is wrong. For values of  $D$  higher than 100 it is more efficient to use the code using `\xintFxFtPowerSeries`.

```
\def\coefflog #1{1/#1[0]}% 1/n
\def\xa {13/256[0]}% we will compute log(1-13/256)
\def\xb {1/9[0]}% we will compute log(1-1/9)
\def\LogTwo #1{% get log(2)=-2log(1-13/256)- 5log(1-1/9)
  \romannumeral0\expandafter\logtwo
  \the\numexpr (#1+1)*150/143\expandafter.%
  \the\numexpr (#1+1)*100/129\expandafter.%
  \the\numexpr #1.%
}%
\def\logtwo #1.#2.#3.{% #3=nb of digits for truncating an EXACT partial sum
  \xinttrunc {#3}% will terminate the \romannumeral0
  {\xintAdd
    {\xintMul {2}{\xintPowerSeries {1}{#2}{\coefflog}{\xa}}}
    {\xintMul {5}{\xintPowerSeries {1}{#1}{\coefflog}{\xb}}}%
  }%
}%
```

```
}%
```

Let us turn now to  $\pi$ , computed with the Machin formula (but see also the approach via the [Brent-Salamin algorithm](#) with `\xintfloatexpr`) Again the numbers of terms to keep in the two `arctg` series were roughly estimated, and some experimentations showed that removing the last three digits was enough (at least for `D=0-100` range). And the algorithm does print the correct digits when used with `D=1000` (to be convinced of that one needs to run it for `D=1000` and again, say for `D=1010`.) A theoretical analysis could help confirm that this algorithm always gets better than  $10^{-D}$  precision, but again, strings of zeroes or nines encountered in the decimal expansion may falsify the ending digits, nines may be zeroes (and the last non-nine one should be increased) and zeroes may be nine (and the last non-zero one should be decreased).

```
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax
/\the\numexpr 2*#1+1\relax [0]}%
%\def\coeffarctg #1{\romannumeral0\xintmon{#1}/\the\numexpr 2*#1+1\relax}%
\def\xa {1/25[0]}%      1/5^2, the [0] for faster parsing
\def\xb {1/57121[0]}%  1/239^2, the [0] for faster parsing
\def\Machin #1{% #1 may be a count register
  \romannumeral0\expandafter\machin
  % number of terms for arctg(1/5):
  \the\numexpr (#1+3)*5/7\expandafter.%
  % number of terms for arctg(1/239):
  \the\numexpr (#1+3)*10/45\expandafter.%
  % do the computations with 3 additional digits:
  \the\numexpr #1+3\expandafter.%
  % allow #1 to be a count register:
  \the\numexpr #1.%
}%
\def\machin #1.#2.#3.#4.{%
  \xinttrunc {#4}% will terminate the \romannumeral0
  {\xintSub
    {\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
    {\xintMul{4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
  }%
}%
\begin{framed}
  \[ \pi = \Machin{60}\dots \]
\end{framed}
```

$$\pi = 3.141592653589793238462643383279502884197169399375105820974944\dots$$

Here is a variant `\MachinBis`, which evaluates the partial sums exactly using `\xintPowerSeries`, before their final truncation. No need for a ```+3''` then.

```
\def\MachinBis #1{% #1 may be a count register,
  % the final result will be truncated to #1 digits post decimal point
  \romannumeral0\expandafter\machinbis
  % number of terms for arctg(1/5):
  \the\numexpr #1*5/7\expandafter.%
  % number of terms for arctg(1/239):
  \the\numexpr #1*10/45\expandafter.%
  % allow #1 to be a count register:
  \the\numexpr #1.%
}%
```

```
\def\machinbis #1.#2.#3.{%
  \xinttrunc {#3}% will terminate the \romannumeral0
  {\xintSub
    {\xintMul {16/5}{\xintPowerSeries {0}{#1}{\coeffarctg}{\xa}}}
    {\xintMul{4/239}{\xintPowerSeries {0}{#2}{\coeffarctg}{\xb}}}%
  }%
}%
```

Let us use this variant for a loop showing the build-up of digits:

```
\begin{multicols}{2}
  \cnta 0 % previously declared \count register
  \loop \noindent
    \centeredline{\dt{MachinBis{\cnta}}}%
  \ifnum\cnta < 30
    \advance\cnta 1 \repeat
\end{multicols}
```

	3.141592653589793
3.	3.1415926535897932
3.1	3.14159265358979323
3.14	3.141592653589793238
3.141	3.1415926535897932384
3.1415	3.14159265358979323846
3.14159	3.141592653589793238462
3.141592	3.1415926535897932384626
3.1415926	3.14159265358979323846264
3.14159265	3.141592653589793238462643
3.141592653	3.1415926535897932384626433
3.1415926535	3.14159265358979323846264338
3.14159265358	3.141592653589793238462643383
3.141592653589	3.1415926535897932384626433832
3.1415926535897	3.14159265358979323846264338327
3.14159265358979	3.141592653589793238462643383279

You want more digits and have some time? Save the following to a file, it is the `\Machin` code. Compile with `etex` (or `pdftex` or `xetex` or `luatex`):

```
% Compile with e-TeX extensions enabled (etex, pdftex, ...)
\input xintfrac.sty
\input xintseries.sty
% pi = 16 Arctg(1/5) - 4 Arctg(1/239) (John Machin's formula)
\def\coeffarctg #1{\the\numexpr\ifodd#1 -1\else1\fi\relax
  /\the\numexpr 2*#1+1\relax [0]}%
\def\xa {1/25[0]}%
\def\xb {1/57121[0]}%
\def\Machin #1{%
  \romannumeral0\expandafter\machin
  \the\numexpr (#1+3)*5/7\expandafter.%
  \the\numexpr (#1+3)*10/45\expandafter.%
  \the\numexpr #1+3\expandafter.%
  \the\numexpr #1.%
}%
\def\machin #1.#2.#3.#4.{%
  \xinttrunc {#4}% will terminate the \romannumeral0
  {\xintSub
```



```

{\xintMul {16/5}{\xintFxFtPowerSeries {0}{#1}{\coeffarctg}{\xa}{#3}}}
{\xintMul{4/239}{\xintFxFtPowerSeries {0}{#2}{\coeffarctg}{\xb}{#3}}}%
}%
}%
\xintresettimer
\edef\Z {\Machin {1000}}
\edef\W {\xinttheseconds}
\immediate\writel28{1000 places of pi via Machin formula (took \W s):}
\immediate\writel28{\Z}
\bye

```

This will log the first 1000 digits of  $\pi$  after the decimal point. On my laptop (a 2012 model) this took about 5.05 seconds last time I tried.<sup>73 74</sup>

As mentioned in the introduction, the file `pi.tex` by D. ROEGEL shows that orders of magnitude faster computations are possible within  $\text{T}_{\text{E}}\text{X}$ , but recall our constraints of complete expandability and be merciful, please.

**Why truncating rather than rounding?** One of our main competitors on the market of scientific computing, a canadian product (not encumbered with expandability constraints, and having barely ever heard of  $\text{T}_{\text{E}}\text{X}$  ;-), prints numbers rounded in the last digit. Why didn't we follow suit in the macros `\xintFxFtPowerSeries` and `\xintFxFtPowerSeriesX`? To round at  $D$  digits, and excluding a rewrite or cloning of the division algorithm which anyhow would add to it some overhead in its final steps, `xintfrac` needs to truncate at  $D+1$ , then round. And rounding loses information! So, with more time spent, we obtain a worst result than the one truncated at  $D+1$  (one could imagine that additions and so on, done with only  $D$  digits, cost less; true, but this is a negligible effect per summand compared to the additional cost for this term of having been truncated at  $D+1$  then rounded). Rounding is the way to go when setting up algorithms to evaluate functions destined to be composed one after the other: exact algebraic operations with many summands and an `f` variable which is a fraction are costly and create an even bigger fraction; replacing `f` with a reasonable rounding, and rounding the result, is necessary to allow arbitrary chaining.

But, for the computation of a single constant, we are really interested in the exact decimal expansion, so we truncate and compute more terms until the earlier result gets validated. Finally if we do want the rounding we can always do it on a value computed with  $D+1$  truncation.

<sup>73</sup> With 1.09i and earlier `xint`, this used to be 42 seconds; starting with 1.09j, and prior to 1.2, it was 16 seconds (this was probably due to a more efficient division with denominators at most 9999). The 1.2 `xintcore` achieved a further gain at 5.6 seconds.

<sup>74</sup> With `\xintDigits :=1001\relax`, the non-optimized implementation with the `iter` of `xintexpr` fame using the `Brent-Salamin algorithm`, took, last time I tried (1.2i), about 7 seconds on my laptop (the last two digits were wrong, which is ok as they serve as guard digits), and for obtaining about 500 digits, it was about 1.7s. This is not bad, taking into account that the syntax is almost free rolling speech, contrarily to the code above for the Machin formula computation; we would like to use the quadratically convergent Brent-Salamin algorithm for more digits, but with such computations with numbers of one thousand digits we are beyond the border of the reasonable range for `xint`. Innocent people not knowing what it means to compute with  $\text{T}_{\text{E}}\text{X}$ , and with the extra constraint of expandability will wonder why this is at least thousands of times slower than with any other language (with a little Python program using the `Decimal` library, I timed the Brent-Salamin algorithm to 4.4ms for about 1000 digits and 1.14ms for 500 digits.) I will just say that for example digits are represented and manipulated via their ascii-code ! all computations must convert from ascii-code to cpu words; furthermore nothing can be stored away. And there is no memory storage with  $O(1)$  time access... if expandability is to be verified.



## 16. Macros of the **xintcfrac** package

.1	Package overview .....	216	.16	<code>\xintCtoCv</code> .....	226
.2	<code>\xintCFrac</code> .....	221	.17	<code>\xintGctoCv</code> .....	226
.3	<code>\xintGCFrac</code> .....	221	.18	<code>\xintFtoCv</code> .....	226
.4	<code>\xintGGCFrac</code> .....	221	.19	<code>\xintFtoCCv</code> .....	226
.5	<code>\xintGctoGCx</code> .....	222	.20	<code>\xintCntoF</code> .....	226
.6	<code>\xintFtoC</code> .....	222	.21	<code>\xintGcntoF</code> .....	227
.7	<code>\xintFtoCs</code> .....	222	.22	<code>\xintCntoCs</code> .....	227
.8	<code>\xintFtoCx</code> .....	222	.23	<code>\xintCntoGC</code> .....	227
.9	<code>\xintFtoGC</code> .....	223	.24	<code>\xintGcntoGC</code> .....	228
.10	<code>\xintFGtoC</code> .....	223	.25	<code>\xintCstoGC</code> .....	228
.11	<code>\xintFtoCC</code> .....	223	.26	<code>\xintiCstoF</code> , <code>\xintiGctoF</code> , <code>\xintiCstoCv</code> , <code>\xintiGctoCv</code> .....	228
.12	<code>\xintCstoF</code> .....	224	.27	<code>\xintGctoGC</code> .....	228
.13	<code>\xintCtoF</code> .....	224	.28	Euler's number <i>e</i> .....	229
.14	<code>\xintGctoF</code> .....	225			
.15	<code>\xintCstoCv</code> .....	225			

First version of this package was included in release 1.04 (2013/04/25) of the *xint bundle*. It was kept almost unchanged until 1.09m of 2014/02/26 which brought some new macros: `\xintFtoC`, `\xintCtoF`, `\xintCtoCv`, dealing with sequences of braced partial quotients rather than comma separated ones, `\xintFGtoC` which is to produce ``guaranteed'' coefficients of some real number known approximately, and `\xintGGCFrac` for displaying arbitrary material as a continued fraction; also, some changes to existing macros: `\xintFtoCs` and `\xintCntoCs` insert spaces after the commas, `\xintCstoF` and `\xintCstoCv` authorize spaces in the input also before the commas.

`\xintCstoF` and `\xintCstoCv` create a partial dependency on *xinttools* as they use its `\xintCSVtoList`. Starting at 1.4n the loading of *xinttools* is done automatically, formerly it was up to user to do it.

### 16.1. Package overview

The package computes partial quotients and convergents of a fraction, or conversely start from coefficients and obtain the corresponding fraction; three macros `\xintCFrac`, `\xintGCFrac` and `\xintGGCFrac` are for typesetting, the others can be nested (if applicable) or see their outputs further processed by other macros from the *xint bundle*, particularly the macros of *xinttools* dealing with sequences of braced items or comma separated lists.

A *simple* continued fraction has coefficients  $[c_0, c_1, \dots, c_N]$  (usually called partial quotients, but I dislike this entrenched terminology), where  $c_0$  is a positive or negative integer and the others are positive integers.

Typesetting is usually done via the *amsmath* macro `\cfrac`:

```
\[ c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{c_3 + \cfrac{1}{\ddots}}}}\]
```

$$c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \frac{1}{c_3 + \frac{1}{\ddots}}}}$$

Here is a concrete example:

```
\[ \xintTeXFrac {208341/66317}=\xintCFrac {208341/66317}\]
```

$$\frac{208341}{66317} = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{2}}}}}$$

But it is the macro `\xintCFrac` which did all the work of *computing* the continued fraction and using `\cfrac` from `amsmath` to typeset it.

A *generalized* continued fraction has the same structure but the numerators are not restricted to be 1, and numbers used in the continued fraction may be arbitrary, also fractions, irrationals, complex, indeterminates.<sup>75</sup> The *centered* continued fraction is an example:

```
\[ \xintTeXFrac {915286/188421}=\xintGCFrac {5+-1/7+1/39+-1/53+-1/13}
=\xintCFrac {915286/188421}\]
```

$$\frac{915286}{188421} = 5 - \frac{1}{7 + \frac{1}{39 - \frac{1}{53 - \frac{1}{13}}}} = 4 + \frac{1}{1 + \frac{1}{6 + \frac{1}{38 + \frac{1}{1 + \frac{1}{51 + \frac{1}{1 + \frac{1}{12}}}}}}}$$

The macro `\xintGCFrac`, contrarily to `\xintCFrac`, does not compute anything, it just typesets starting from a generalized continued fraction in inline format, which in this example was input literally. We also used `\xintCFrac` for comparison of the two types of continued fractions.

To let  $\TeX$  compute the centered continued fraction of  $f$  there is `\xintFtoCC`:

```
\[ \xintTeXFrac {915286/188421}\to\xintFtoCC {915286/188421}\]
```

$$\frac{915286}{188421} \rightarrow 5 + -1/7 + 1/39 + -1/53 + -1/13$$

The package macros are expandable and may be nested (naturally `\xintCFrac` and `\xintGCFrac` must be at the top level, as they deal with typesetting).

```
\[ \xintGCFrac {\xintFtoCC{915286/188421}}\]
```

$$5 - \frac{1}{7 + \frac{1}{39 - \frac{1}{53 - \frac{1}{13}}}}$$

The 'inline' format expected on input by `\xintGCFrac` is

$$a_0 + b_0/a_1 + b_1/a_2 + b_2/a_3 + \cdots + b_{n-2}/a_{n-1} + b_{n-1}/a_n$$

Fractions among the coefficients are allowed but they must be enclosed within braces. Signed integers may be left without braces (but the  $+$  signs are mandatory). No spaces are allowed around the plus and fraction symbols. The coefficients may themselves be macros, as long as these macros are *f-expandable*.

```
\[ \xintTeXFrac{\xintGctoF {1+-1/57+\xintPow {-3}{7}}/\xintiiQuo {132}{25}}
= \xintGCFrac {1+-1/57+\xintPow {-3}{7}}/\xintiiQuo {132}{25}\]
```

<sup>75</sup> `xintcfrac` may be used with indeterminates, for basic conversions from one inline format to another, but not for actual computations. See `\xintGGCFrac`.

$$\frac{1907}{1902} = 1 - \frac{1}{57 - \frac{2187}{5}}$$

To compute the actual fraction one has `\xintGCtoF`:

```
\[\xintTeXFrac{\xintGCtoF {1+-1/57+\xintPow {-3}{7}/\xintiiQuo {132}{25}}}\]
```

$$\frac{1907}{1902}$$

For non-numeric input there is `\xintGGCFrac`.

```
\[\xintGGCFrac {a_0+b_0/a_1+b_1/a_2+b_2/\ddots+\ddots/a_{n-1}+b_{n-1}/a_n}\]
```

$$a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \frac{b_2}{\ddots + \frac{b_{n-1}}{a_{n-1} + \frac{b_n}{a_n}}}}}$$

For regular continued fractions, there is a simpler comma separated format:

```
\[-7,6,19,1,33\to\xintTeXFrac{\xintCstoF{-7,6,19,1,33}}=\xintCFrac{\xintCstoF{-7,6,19,1,33}}\]
```

$$-7, 6, 19, 1, 33 \rightarrow \frac{-28077}{4108} = -7 + \frac{1}{6 + \frac{1}{19 + \frac{1}{1 + \frac{1}{33}}}}$$

The macro `\xintFtoCs` produces from a fraction `f` the comma separated list of its coefficients.

```
\[\xintTeXFrac{1084483/398959}=[\xintFtoCs{1084483/398959}]\]
```

$$\frac{1084483}{398959} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 2]$$

If one prefers other separators, one can use the two arguments macros `\xintFtoCx` whose first argument is the separator (which may consist of more than one token) which is to be used.

```
\[\xintTeXFrac{2721/1001}=\xintFtoCx {+1/{}}{2721/1001}\cdots\]
```

$$\frac{2721}{1001} = 2 + 1/(1 + 1/(2 + 1/(1 + 1/(1 + 1/(4 + 1/(1 + 1/(1 + 1/(6 + 1/(2) \cdots)))))))$$

This allows under Plain  $\text{\TeX}$  with `amstex` to obtain the same effect as with  $\text{\LaTeX}$ +`\amsmath`+`\xintCFrac`:

```
$$\xintTeXOver{2721/1001}=\xintFtoCx {+\cfrac1{\ }}{2721/1001}\endcfrac$$
```

As a shortcut to `\xintFtoCx` with separator `1+/,` there is `\xintFtoGC`:

```
2721/1001=\xintFtoGC {2721/1001}
```

`2721/1001=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/2` Let us compare in that case with the output of `\xintFtoCC`:

```
2721/1001=\xintFtoCC {2721/1001}
```

`2721/1001=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2` To obtain the coefficients as a sequence of braced numbers, there is `\xintFtoC` (this is a shortcut for `\xintFtoCx {}`). This list (sequence) may then be manipulated using the various macros of `xinttools` such as the non-expandable macro `\xintAssignArray` or the expandable `\xintApply` and `\xintListWithSep`.

Conversely to go from such a sequence of braced coefficients to the corresponding fraction there is `\xintCtoF`.

The `\printhnumber` (subsection 1.6) macro which we use in this document to print long numbers can also be useful on long continued fractions.

```
\printhnumber{\xintFtoCC {35037018906350720204351049/244241737886197404558180}}
```

$143 + \frac{1}{2} + \frac{1}{5} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{3} + \frac{1}{2} + \frac{1}{2} + \frac{1}{6} + \frac{1}{22} + \frac{1}{2} + \frac{1}{10} + \frac{1}{5} + \frac{1}{11} + \frac{1}{3} + \frac{1}{4} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{2} + \frac{1}{23} + \frac{1}{3} + \frac{1}{8} + \frac{1}{6} + \frac{1}{9}$  If we apply `\xintGCtoF` to this generalized continued fraction, we discover that the original fraction was reducible:

```
\xintGCtoF {143+1/2+...+1/9}=2897319801297630107/20197107104701740
```

When a generalized continued fraction is built with integers, and numerators are only 1's or -1's, the produced fraction is irreducible. And if we compute it again with the last sub-fraction omitted we get another irreducible fraction related to the bigger one by a Bézout identity. Doing this here we get:

```
\xintGCtoF {143+1/2+...+1/6}=328124887710626729/2287346221788023
```

and indeed:

$$\begin{vmatrix} 2897319801297630107 & 328124887710626729 \\ 20197107104701740 & 2287346221788023 \end{vmatrix} = 1$$

The various fractions obtained from the truncation of a continued fraction to its initial terms are called the convergents. The macros of `xintcfrac` such as `\xintFtoCv`, `\xintFtoCCv`, and others which compute such convergents, return them as a list of braced items, with no separator (as does `\xintFtoC` for the partial quotients). Here is an example:

```
\[ \xintTeXFrac{915286/188421} \to
\xintListWithSep{,}{\xintApply\xintTeXFrac{\xintFtoCv{915286/188421}}}\]
```

$$\frac{915286}{188421} \rightarrow 4, 5, \frac{34}{7}, \frac{1297}{267}, \frac{1331}{274}, \frac{69178}{14241}, \frac{70509}{14515}, \frac{915286}{188421}$$

```
\[ \xintTeXFrac{915286/188421} \to
\xintListWithSep{,}{\xintApply\xintTeXFrac{\xintFtoCCv{915286/188421}}}\]
```

$$\frac{915286}{188421} \rightarrow 5, \frac{34}{7}, \frac{1331}{274}, \frac{70509}{14515}, \frac{915286}{188421}$$

We thus see that the 'centered convergents' obtained with `\xintFtoCCv` are among the fuller list of convergents as returned by `\xintFtoCv`.

Here is a more complicated use of `\xintApply` and `\xintListWithSep`. We first define a macro which will be applied to each convergent:

```
\newcommand{\mymacro}[1]{\xintTeXFrac{#1}=\xintFtoCs{#1}}\vtop to 6pt{}
```

Next, we use the following code:

```
\xintTeXFrac{49171/18089}\to{ }$
\xintListWithSep {, }{\xintApply{\mymacro}{\xintFtoCv{49171/18089}}}
```

It produces:

$\frac{49171}{18089} \rightarrow 2 = [2], 3 = [3], \frac{8}{3} = [2, 1, 2], \frac{11}{4} = [2, 1, 3], \frac{19}{7} = [2, 1, 2, 2], \frac{87}{32} = [2, 1, 2, 1, 1, 4], \frac{106}{39} = [2, 1, 2, 1, 1, 5], \frac{193}{71} = [2, 1, 2, 1, 1, 4, 2], \frac{1264}{465} = [2, 1, 2, 1, 1, 4, 1, 1, 6], \frac{1457}{536} = [2, 1, 2, 1, 1, 4, 1, 1, 7], \frac{2721}{1001} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 2], \frac{23225}{8544} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8], \frac{49171}{18089} = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 2]$

The macro `\xintCtoF` allows to specify the coefficients as a function given by a one-parameter macro. The produced values do not have to be integers.

```
\def\cn #1{\xintiiPow {2}{#1}}% 2^n
\[ \xintTeXFrac{\xintCtoF {6}{\cn}}=\xintCFrac [1]{\xintCtoF {6}{\cn}}\]
```

$$\frac{3541373}{2449193} = 1 + \frac{1}{2 + \frac{1}{4 + \frac{1}{8 + \frac{1}{16 + \frac{1}{32 + \frac{1}{64}}}}}}$$

Notice the use of the optional argument `[1]` to `\xintCFrac`. Other possibilities are `[r]` and (default) `[c]`.

```
\def\cn #1{\xintPow {2}{-#1}}%
\[\xintTeXFrac{\xintCntoF {6}{\cn}}=\xintGCFrac [r]{\xintCntoGC {6}{\cn}}=
[\xintFtoCs {\xintCntoF {6}{\cn}}]\]
```

$$\frac{3159019}{2465449} = 1 + \frac{1}{\frac{1}{2} + \frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{8} + \frac{1}{\frac{1}{16} + \frac{1}{\frac{1}{32} + \frac{1}{\frac{1}{64}}}}}}} = [1, 3, 1, 1, 4, 14, 1, 1, 1, 1, 79, 2, 1, 1, 2]$$

We used `\xintCntoGC` as we wanted to display also the continued fraction and not only the fraction returned by `\xintCntoF`.

There are also `\xintGCntoF` and `\xintGCntoGC` which allow the same for generalized fractions. An initial portion of a generalized continued fraction for  $\pi$  is obtained like this

```
\def\an #1{\the\numexpr 2*#1+1\relax}%
\def\bn #1{\the\numexpr (#1+1)*(#1+1)\relax}%
\[\xintTeXFrac{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}}=
\cfrac{4}{\xintGCFrac{\xintGCntoGC {5}{\an}{\bn}}}=
\xintTrunc {10}{\xintDiv {4}{\xintGCntoF {5}{\an}{\bn}}}\dots\]
```

$$\frac{92736}{29520} = \frac{4}{1 + \frac{1}{3 + \frac{4}{5 + \frac{9}{7 + \frac{16}{9 + \frac{25}{11}}}}}} = 3.1414634146\dots$$

We see that the quality of approximation is not fantastic compared to the simple continued fraction of  $\pi$  with about as many terms:

```
\[\xintTeXFrac{\xintCstoF{3,7,15,1,292,1,1}}=
\xintGCFrac{3+1/7+1/15+1/1+1/292+1/1+1/1}=
\xintTrunc{10}{\xintCstoF{3,7,15,1,292,1,1}}\dots\]
```

$$\frac{208341}{66317} = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{1 + \frac{1}{1}}}}}} = 3.1415926534\dots$$

When studying the continued fraction of some real number, there is always some doubt about how many terms are valid, when computed starting from some approximation. If  $f \leq x \leq g$  and  $f, g$  both have the same first  $K$  partial quotients, then  $x$  also has the same first  $K$  quotients and convergents. The macro `\xintFGtoC` outputs as a sequence of braced items the common partial quotients of its two arguments. We can thus use it to produce a sure list of valid convergents of  $\pi$  for example, starting from some proven lower and upper bound:

```
$$\pi\to [\xintListWithSep{,}]
```

```

{\xintFGtoC {3.14159265358979323}{3.14159265358979324}}, \dots}$$
\noindent$\pi\to\xintListWithSep{\allowbreak\;}
{\xintApply{\xintTeXFrac}
{\xintCtoCv{\xintFGtoC {3.14159265358979323}{3.14159265358979324}}}}, \dots$

```

$$\pi \rightarrow [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, \dots]$$

$$\pi \rightarrow 3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103993}{33102}, \frac{104348}{33215}, \frac{208341}{66317}, \frac{312689}{99532}, \frac{833719}{265381}, \frac{1146408}{364913}, \frac{4272943}{1360120}, \frac{5419351}{1725033}, \frac{80143857}{25510582}, \frac{165707065}{52746197}, \frac{245850922}{78256779}, \frac{411557987}{131002976}, \dots$$

source

## 16.2. `\xintCFrac`

Frac  
f

`\xintCFrac{f}` is a math-mode only,  $\TeX$  with `amsmath` only, macro which first computes then displays with the help of `\cfrac` the simple continued fraction corresponding to the given fraction. It admits an optional argument which may be `[l]`, `[r]` or (the default) `[c]` to specify the location of the one's in the numerators of the sub-fractions. This macro is *f-expandable* in the sense that it prepares expandably the whole expression with the multiple `\cfrac`'s, but it is not completely expandable naturally as `\cfrac` isn't.

source

## 16.3. `\xintGCFrac`

*f* `\xintGCFrac{a+b/c+d/e+f/g+h/...+x/y}` uses similarly `\cfrac` to prepare the typesetting with the `amsmath` `\cfrac` ( $\TeX$ ) of a generalized continued fraction given in inline format (or as macro which will *f-expand* to it). It admits the same optional argument as `\xintCFrac`. Plain  $\TeX$  with `amstex` users, see `\xintGCToGCx`.

```

\[\xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFrac {6}}{\}]

```

$$1 + \frac{3375 \cdot 10^{-3}}{\frac{1}{7} - \frac{3}{5}}$$

This is mostly a typesetting macro, although it does trigger the expansion of the coefficients. See `\xintGCToF` if you are impatient to see this specific fraction computed.

It admits an optional argument within square brackets which may be either `[l]`, `[c]` or `[r]`. Default is `[c]` (numerators are centered).

Numerators and denominators are made arguments to the `\xintTeXFrac` macro. This allows them to be themselves fractions or anything *f-expandable* giving numbers or fractions, but also means however that they can not be arbitrary material, they can not contain color changing macros for example. One of the reasons is that `\xintGCFrac` tries to determine the signs of the numerators and chooses accordingly to use + or -.

source

## 16.4. `\xintGGCFrac`

*f* `\xintGGCFrac{a+b/c+d/e+f/g+h/...+x/y}` is a clone of `\xintGCFrac`, hence again  $\TeX$  specific with package `amsmath`. It does not assume the coefficients to be numbers as understood by `xintfrac`. The macro can be used for displaying arbitrary content as a continued fraction with `\cfrac`, using only plus signs though. Note though that it will first *f-expand* its argument, which may be thus be one of the `xintcfrac` macros producing a (general) continued fraction in inline format, see `\xintFtoCx` for an example. If this expansion is not wished, it is enough to start the argument with a space.

```

\[\xintGGCFrac {1+q/1+q^2/1+q^3/1+q^4/1+q^5/\ddots}\]

```

$$1 + \frac{q}{1 + \frac{q^2}{1 + \frac{q^3}{1 + \frac{q^4}{1 + \frac{q^5}{\ddots}}}}}$$

*source***16.5. `\xintGtoGCx`**

*nn f* ★ `\xintGtoGCx{sepa}{sepb}{a+b/c+d/e+f/...+x/y}` returns the list of the coefficients of the generalized continued fraction of *f*, each one within a pair of braces, and separated with the help of *sepa* and *sepb*. Thus

`\xintGtoGCx ;{1+2/3+4/5+6/7}` gives 1:2;3:4;5:6;7

The following can be used by Plain  $\text{\TeX}$ +*amstex* users to obtain an output similar as the ones produced by `\xintGCfrac` and `\xintGGCfrac`:

```

 $\xintGtoGCx {+\cfrac{\{\}\{a+b/\dots\}\endcfrac{\$}$ 
 $\xintGtoGCx {+\cfrac{\xintTeXOver{\}\{\xintTeXOver{\}\{a+b/\dots\}\endcfrac{\$}$ 

```

*source***16.6. `\xintFtoC`**

*Frac f* ★ `\xintFtoC{f}` computes the coefficients of the simple continued fraction of *f* and returns them as a list (sequence) of braced items.

```
\fdef\test{\xintFtoC{-5262046/89233}}\texttt{\meaning\test}
```

macro:->{-59}{33}{27}{100}

*source***16.7. `\xintFtoCs`**

*Frac f* ★ `\xintFtoCs{f}` returns the comma separated list of the coefficients of the simple continued fraction of *f*. Notice that starting with 1.09m a space follows each comma (mainly for usage in text mode, as in math mode spaces are produced in the typeset output by  $\text{\TeX}$  itself).

```
\[ \xintTeXsignedFrac{-5262046/89233} \to [\xintFtoCs{-5262046/89233}]\]
```

$$-\frac{5262046}{89233} \rightarrow [-59, 33, 27, 100]$$

*source***16.8. `\xintFtoCx`**

*n Frac f* ★ `\xintFtoCx{sep}{f}` returns the list of the coefficients of the simple continued fraction of *f* separated with the help of *sep*, which may be anything (and is kept unexpanded). For example, with Plain  $\text{\TeX}$  and *amstex*,

```
 $\xintFtoCx {+\cfrac{1\{\}\{-5262046/89233\}\endcfrac{\$}$ 
```

will display the continued fraction using `\cfrac`. Each coefficient is inside a brace pair `{ }`, allowing a macro to end the separator and fetch it as argument, for example, again with Plain  $\text{\TeX}$  and *amstex*:

```

\def\highlight #1{\ifnum #1>200 \textcolor{red}{\#1}\else #1\fi}
 $\xintFtoCx {+\cfrac{1\{\}\highlight{\{104348/33215\}\endcfrac{\$}$ 

```

Due to the different and extremely cumbersome syntax of `\cfrac` under  $\text{\TeX}$  it proves a bit tortuous to obtain there the same effect. Actually, it is partly for this purpose that 1.09m added `\xintGGCfrac`. We thus use `\xintFtoCx` with a suitable separator, and then the whole thing as argument to `\xintGGCfrac`:

```
\def\highlight #1{\ifnum #1>200 \fcolorbox{blue}{white}{\boldmath\color{red}$#1$}%
\else #1\fi}
\[\xintGGCFrac {\xintFtoCx {+1/\highlight}{208341/66317}}\]
```

$$3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{\boxed{292} + \frac{1}{2}}}}}$$

source

16.9. `\xintFtoGC`

Frac f ★ `\xintFtoGC{f}` does the same as `\xintFtoCx{+1/}{f}`. Its output may thus be used in the package macros expecting such an 'inline format'.

```
566827/208524=\xintFtoGC {566827/208524}
```

```
566827/208524=2+1/1+1/2+1/1+1/1+1/4+1/1+1/1+1/6+1/1+1/1+1/8+1/1+1/1+1/11
```

source

16.10. `\xintFGtoC`

Frac f Frac f ★ `\xintFGtoC{f}{g}` computes the common initial coefficients to two given fractions `f` and `g`. Notice that any real number `f<x<g` or `f>x>g` will then necessarily share with `f` and `g` these common initial coefficients for its regular continued fraction. The coefficients are output as a sequence of braced numbers. This list can then be manipulated via macros from *xinttools*, or other macros of *xintcfrac*.

```
\fdef\test{\xintFGtoC{-5262046/89233}{-5314647/90125}}\texttt{\meaning\test}
```

```
macro:->{-59}{33}{27}
```

```
\fdef\test{\xintFGtoC{3.141592653}{3.141592654}}\texttt{\meaning\test}
```

```
macro:->{3}{7}{15}{1}
```

```
\fdef\test{\xintFGtoC{3.1415926535897932384}{3.1415926535897932385}}\meaning\test
```

```
macro:->{3}{7}{15}{1}{292}{1}{1}{1}{2}{1}{3}{1}{14}{2}{1}{1}{2}{2}{2}
```

```
\xintRound {30}{\xintCstoF{\xintListWithSep{,}{\test}}}
```

```
3.141592653589793238386377506390
```

```
\xintRound {30}{\xintCtoF{\test}}
```

```
3.141592653589793238386377506390
```

```
\fdef\test{\xintFGtoC{1.41421356237309}{1.4142135623731}}\meaning\test
```

```
macro:->{1}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}{2}
```

source

16.11. `\xintFtoCC`

Frac f ★ `\xintFtoCC{f}` returns the 'centered' continued fraction of `f`, in 'inline format'.

```
566827/208524=\xintFtoCC {566827/208524}
```

```
566827/208524=3+-1/4+-1/2+1/5+-1/2+1/7+-1/2+1/9+-1/2+1/11
```

```
\[\xintTeXFrac{566827/208524} = \xintGGCFrac{\xintFtoCC{566827/208524}}\]
```



$$\frac{566827}{208524} = 3 - \frac{1}{4 - \frac{1}{2 + \frac{1}{5 - \frac{1}{2 + \frac{1}{7 - \frac{1}{2 + \frac{1}{9 - \frac{1}{2 + \frac{1}{11}}}}}}}}}$$

[source](#)

## 16.12. `\xintCstoF`

*f* ★ `\xintCstoF{a,b,c,d,...,z}` computes the fraction corresponding to the coefficients, which may be fractions or even macros expanding to such fractions. The final fraction may then be highly reducible.

Starting with release 1.09m spaces before commas are allowed and trimmed automatically (spaces after commas were already silently handled in earlier releases).

```
\[\xintGCFrac {-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}=
\xintTeXsignedFrac{\xintCstoF {-1,3,-5,7,-9,11,-13}}=\xintTeXsignedFrac{\xintGctoF
{-1+1/3+1/-5+1/7+1/-9+1/11+1/-13}}\]
```

$$-1 + \frac{1}{3 + \frac{1}{-5 + \frac{1}{7 + \frac{1}{-9 + \frac{1}{11 + \frac{1}{-13}}}}}} = -\frac{75887}{118187} = -\frac{75887}{118187}$$

```
\[\xintGCFrac{{1/2}+1/{1/3}+1/{1/4}+1/{1/5}}=
\xintTeXFrac{\xintCstoF {1/2,1/3,1/4,1/5}}\]
```

$$\frac{1}{2} + \frac{1}{\frac{1}{\frac{1}{\frac{1}{3} + \frac{1}{\frac{1}{4} + \frac{1}{\frac{1}{5}}}}}} = \frac{159}{66}$$

A generalized continued fraction may produce a reducible fraction (`\xintCstoF` tries its best not to accumulate in a silly way superfluous factors but will not do simplifications which would be obvious to a human, like simplification by 3 in the result above).

[source](#)

## 16.13. `\xintCtoF`

*f* ★ `\xintCtoF{{a}{b}{c}...{z}}` computes the fraction corresponding to the coefficients, which may be fractions or even macros.

```
\xintCtoF {\xintApply {\xintiiPow 3}{\xintSeq {1}{5}}}
14946960/4805083
```

```
\[ \xintTeXFrac{14946960/4805083}=\xintCFrac {14946960/4805083}\]
```

$$\frac{14946960}{4805083} = 3 + \frac{1}{9 + \frac{1}{27 + \frac{1}{81 + \frac{1}{243}}}}$$

In the example above the power of 3 was already pre-computed via the expansion done by `\xintApply`, but if we try with `\xintApply { \xintiiPow 3}` where the space will stop this expansion, we can check that `\xintCtoF` will itself provoke the needed coefficient expansion.

*source*

## 16.14. `\xintGCtoF`

*f* ★ `\xintGCtoF{a+b/c+d/e+f/g+.....+v/w+x/y}` computes the fraction defined by the inline generalized continued fraction. Coefficients may be fractions but must then be put within braces. They can be macros. The plus signs are mandatory.

```
\[ \xintGCFrac {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}} =
\xintTeXFrac{\xintGCtoF {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}}} =
\xintTeXFrac{\xintIrr{\xintGCtoF
{1+\xintPow{1.5}{3}/{1/7}+{-3/5}/\xintiiFac {6}}}} \]
```

$$1 + \frac{3375 \cdot 10^{-3}}{\frac{1}{7} - \frac{\frac{3}{5}}{720}} = \frac{88629000}{3579000} = \frac{29543}{1193}$$

```
\[ \xintGCFrac{{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}} =
\xintTeXFrac{\xintGCtoF {{1/2}+{2/3}/{4/5}+{1/2}/{1/5}+{3/2}/{5/3}}} \]
```

$$\frac{1}{2} + \frac{\frac{2}{3}}{\frac{4}{5} + \frac{\frac{1}{2}}{\frac{1}{5} + \frac{\frac{3}{2}}{\frac{5}{3}}}} = \frac{4270}{4140}$$

The macro tries its best not to accumulate superfluous factor in the denominators, but doesn't reduce the fraction to irreducible form before returning it and does not do simplifications which would be obvious to a human.

*source*

## 16.15. `\xintCstoCv`

*f* ★ `\xintCstoCv{a,b,c,d,...,z}` returns the sequence of the corresponding convergents, each one within braces.

It is allowed to use fractions as coefficients (the computed convergents have then no reason to be the real convergents of the final fraction). When the coefficients are integers, the convergents are irreducible fractions, but otherwise it is not necessarily the case.

```
\xintListWithSep:{\xintCstoCv{1,2,3,4,5,6}}
```

```
1/1:3/2:10/7:43/30:225/157:1393/972
```

```
\xintListWithSep:{\xintCstoCv{1,1/2,1/3,1/4,1/5,1/6}}
```

```
1/1:3/1:9/7:45/19:225/159:1575/729
```

```
\[ \xintListWithSep{\to}{\xintApply\xintTeXFrac{\xintCstoCv {\xintPow
{- .3}{-5},7.3/4.57,\xintCstoF{3/4,9,-1/3}}}} \]
```

$$\frac{-100000}{243} \rightarrow \frac{-72888949}{177390} \rightarrow \frac{-2700356878}{6567804}$$

source

**16.16. \xintCtoCv**

$f$  ★ `\xintCtoCv{a}{b}{c}...{z}` returns the sequence of the corresponding convergents, each one within braces.

```
\fdef\test{\xintCtoCv {1111111111}}\texttt{\meaning\test}
```

```
macro:->{1/1}{2/1}{3/2}{5/3}{8/5}{13/8}{21/13}{34/21}{55/34}{89/55}{144/89}
```

source

**16.17. \xintGtoCv**

$f$  ★ `\xintGtoCv{a+b/c+d/e+f/g+.....+v/w+x/y}` returns the list of the corresponding convergents. The coefficients may be fractions, but must then be inside braces. Or they may be macros, too.

The convergents will in the general case be reducible. To put them into irreducible form, one needs one more step, for example it can be done with `\xintApply\xintIrr`.

```
\[\xintListWithSep{,}{\xintApply\xintTeXFrac
{\xintGtoCv{3+{-2}/{7/2}+{3/4}/12+{-56/3}}}\]}
\[\xintListWithSep{,}{\xintApply\xintTeXFrac{\xintApply\xintIrr
{\xintGtoCv{3+{-2}/{7/2}+{3/4}/12+{-56/3}}}\]}]
```

$$3, \frac{17}{7}, \frac{834}{342}, \frac{1306}{542}$$

$$3, \frac{17}{7}, \frac{139}{57}, \frac{653}{271}$$

source

**16.18. \xintFtoCv**

$\frac{f}{f}$  ★ `\xintFtoCv{f}` returns the list of the (braced) convergents of  $f$ , with no separator. To be treated with `\xintAssignArray` or `\xintListWithSep`.

```
\[\xintListWithSep{\to}{\xintApply\xintTeXFrac{\xintFtoCv{5211/3748}}}\]
```

$$1 \rightarrow \frac{3}{2} \rightarrow \frac{4}{3} \rightarrow \frac{7}{5} \rightarrow \frac{25}{18} \rightarrow \frac{32}{23} \rightarrow \frac{57}{41} \rightarrow \frac{317}{228} \rightarrow \frac{374}{269} \rightarrow \frac{691}{497} \rightarrow \frac{5211}{3748}$$

source

**16.19. \xintFtoCCv**

$\frac{f}{f}$  ★ `\xintFtoCCv{f}` returns the list of the (braced) centered convergents of  $f$ , with no separator. To be treated with `\xintAssignArray` or `\xintListWithSep`.

```
\[\xintListWithSep{\to}{\xintApply\xintTeXFrac{\xintFtoCCv{5211/3748}}}\]
```

$$1 \rightarrow \frac{4}{3} \rightarrow \frac{7}{5} \rightarrow \frac{32}{23} \rightarrow \frac{57}{41} \rightarrow \frac{374}{269} \rightarrow \frac{691}{497} \rightarrow \frac{5211}{3748}$$

source

**16.20. \xintCtoF**

$\frac{\text{num}}{x}$   $f$  ★ `\xintCtoF{N}{\macro}` computes the fraction  $f$  having coefficients  $c(j)=\macro{j}$  for  $j=0,1,\dots,N$ . The  $N$  parameter is given to a `\numexpr`. The values of the coefficients, as returned by `\macro` do not have to be positive, nor integers, and it is thus not necessarily the case that the original  $c(j)$  are the true coefficients of the final  $f$ .

```
\def\macro #1{\the\numexpr 1+#1*#1\relax} \xintCtoF {5}{\macro}
```

```
72625/49902[0]
```

This example shows that the fraction is output with a trailing number in square brackets (representing a power of ten), this is for consistency with what do most macros of *xintfrac*, and does not have to be always this annoying `[0]` as the coefficients may for example be numbers in scientific notation. To avoid these trailing square brackets, for example if the coefficients are

known to be integers, there is always the possibility to filter the output via `\xintPraw`, or `\xint-Irr` (the latter is overkill in the case of integer coefficients, as the fraction is guaranteed to be irreducible then).

*source*

## 16.21. `\xintGCntoF`

$\overset{\text{num}}{x}$   $f f \star$  `\xintGCntoF{N}{\macroA}{\macroB}` returns the fraction  $f$  corresponding to the inline generalized continued fraction  $a_0+b_0/a_1+b_1/a_2+\dots+b(N-1)/a_N$ , with  $a(j)=\macroA{j}$  and  $b(j)=\macroB{j}$ . The  $N$  parameter is given to a `\numexpr`.

```
\def\coeffA #1{\the\numexpr #1+4-3*((#1+2)/3)\relax}%
\def\coeffB #1{\the\numexpr \ifodd #1 -\fi 1\relax}% (-1)^n
\[\xintGCfrac{\xintGCntoGC {6}{\coeffA}{\coeffB}} =
\xintTeXfrac{\xintGCntoF {6}{\coeffA}{\coeffB}}\]
```

$$1 + \frac{1}{2 - \frac{1}{3 + \frac{1}{1 - \frac{1}{2 + \frac{1}{3 - \frac{1}{1}}}}}} = \frac{39}{25}$$

There is also `\xintGCntoGC` to get the 'inline format' continued fraction.

*source*

## 16.22. `\xintCntoCs`

$\overset{\text{num}}{x}$   $f \star$  `\xintCntoCs{N}{\macro}` produces the comma separated list of the corresponding coefficients, from  $n=0$  to  $n=N$ . The  $N$  is given to a `\numexpr`.

```
\xintCntoCs {5}{\macro}
```

1, 2, 5, 10, 17, 26

```
\[ \xintTeXfrac{\xintCntoF{5}{\macro}}=\xintCfrac{\xintCntoF {5}{\macro}}\]
```

$$\frac{72625}{49902} = 1 + \frac{1}{2 + \frac{1}{5 + \frac{1}{10 + \frac{1}{17 + \frac{1}{26}}}}}$$

*source*

## 16.23. `\xintCntoGC`

$\overset{\text{num}}{x}$   $f \star$  `\xintCntoGC{N}{\macro}` evaluates the  $c(j)=\macro{j}$  from  $j=0$  to  $j=N$  and returns a continued fraction written in inline format:  $\{c(0)\}+1/\{c(1)\}+1/\dots+1/\{c(N)\}$ . The parameter  $N$  is given to a `\numexpr`. The coefficients, after expansion, are, as shown, being enclosed in an added pair of braces, they may thus be fractions.

```
\def\macro #1{\the\numexpr\ifodd#1 -1-#1\else1+#1\fi\relax/\the\numexpr 1+#1*#1\relax}
\fdef\x{\xintCntoGC {5}{\macro}}\meaning\x
\[\xintGCfrac{\xintCntoGC {5}{\macro}}\]
```

`macro:->\{1/\the \numexpr 1+0*0\relax \}+1/{-2/\the \numexpr 1+1*1\relax \}+1/{3/\the \numexpr 1+2*2\relax \}+1/{-4/\the \numexpr 1+3*3\relax \}+1/{5/\the \numexpr 1+4*4\relax \}+1/{-6/\the`

`\numexpr 1+5*5\relax }`

$$1 + \frac{1}{\frac{-2}{2} + \frac{1}{\frac{3}{5} + \frac{1}{\frac{-4}{10} + \frac{1}{\frac{5}{17} + \frac{-6}{26}}}}}$$

[source](#)

## 16.24. `\xintGCntoGC`

[num](#) [X](#) [f](#) [f](#) ★ `\xintGCntoGC{N}{\macroA}{\macroB}` evaluates the coefficients and then returns the corresponding `{a0}+{b0}/{a1}+{b1}/{a2}+...+{b(N-1)}/{aN}` inline generalized fraction. `N` is given to a `\numexpr`. The coefficients are enclosed into pairs of braces, and may thus be fractions, the fraction slash will not be confused in further processing by the continued fraction slashes.

```
\def\an #1{\the\numexpr #1*#1*#1+1\relax}%
\def\bn #1{\the\numexpr \ifodd#1 -\fi 1*(#1+1)\relax}%
$\xintGCntoGC {5}{\an}{\bn}=\xintGCfrac {5}{\an}{\bn} =
\displaystyle\xintTeXfrac {\xintGCntoF {5}{\an}{\bn}}{\par
```

$$1 + 1/2 + -2/9 + 3/28 + -4/65 + 5/126 = 1 + \frac{1}{2 - \frac{3}{9 + \frac{4}{28 - \frac{5}{65 + \frac{1}{126}}}}} = \frac{5797655}{3712466}$$

[source](#)

## 16.25. `\xintCstoGC`

[f](#) ★ `\xintCstoGC{a,b,...,z}` transforms a comma separated list (or something expanding to such a list) into an 'inline format' continued fraction `{a}+1/{b}+1/...+1/{z}`. The coefficients are just copied and put within braces, without expansion. The output can then be used in `\xintGCfrac` for example.

```
\[\xintGCfrac {\xintCstoGC {-1,1/2,-1/3,1/4,-1/5}}=\xintTeXsignedFrac{\xintCstoF {-1,1/2,-1/3,1/4,-1/5}}\]
```

$$-1 + \frac{1}{\frac{1}{2} + \frac{1}{\frac{-1}{3} + \frac{1}{\frac{1}{4} + \frac{-1}{5}}}} = -\frac{145}{83}$$

[source](#)

[source](#)

[source](#)

[source](#)

## 16.26. `\xintiCstoF`, `\xintiGctoF`, `\xintiCstoCv`, `\xintiGctoCv`

[f](#) ★ Essentially the same as the corresponding macros without the 'i', but for integer-only input. Infinitesimally faster, mainly for internal use by the package.

[source](#)

## 16.27. `\xintGctoGC`

[f](#) ★ `\xintGctoGC{a+b/c+d/e+f/g+.....+v/w+x/y}` expands (with the usual meaning) each one of the co-

efficients and returns an inline continued fraction of the same type, each expanded coefficient being enclosed within braces.

```
\fdef\x {\xintGCtoGC {1+\xintPow{1.5}{3}/{1/7}+{-3/5}/%
\xintiFac {6}+\xintCstoF {2,-7,-5}/16}} \meaning\x
```

```
macro:->{1}+{3375/1[-3]}/{1/7}+{-3/5}/{720}+{67/36}/{16}
```

To be honest I have forgotten for which purpose I wrote this macro in the first place.

## 16.28. Euler's number e

Let us explore the convergents of Euler's number e. The volume of computation is kept minimal by the following steps:

- a comma separated list of the first 36 coefficients is produced by `\xintCntoCs`,
- this is then given to `\xintiCstoCv` which produces the list of the convergents (there is also `\xintCstoCv`, but our coefficients being integers we used the infinitesimally faster `\xintiCstoCv`),
- then the whole list was converted into a sequence of one-line paragraphs, each convergent becomes the argument to a macro printing it together with its decimal expansion with 30 digits after the decimal point.
- A count register `\cnta` was used to give a line count serving as a visual aid: we could also have done that in an expandable way, but well, let's relax from time to time...

```
\def\cn #1{\the\numexpr\ifcase \numexpr #1+3-3*((#1+2)/3)\relax
1\or1\or2*(#1/3)\fi\relax }
% produces the pattern 1,1,2,1,1,4,1,1,6,1,1,8,... which are the
% coefficients of the simple continued fraction of e-1.
\cnta 0
\def\mymacro #1{\advance\cnta by 1
\noindent
\hbox to 3em {\hfil\small\dt{\the\cnta.} }%
$\xintTrunc {30}{\xintAdd {1[0]}{#1}}\dots=
\xintTeXFrac{\xintAdd {1[0]}{#1}}{#1}}$}%
\xintListWithSep{\vtop to 6pt}{\vbox to 12pt}{\par}
{\xintApply\mymacro{\xintiCstoCv{\xintCntoCs {35}}{\cn}}}}
```

1. 2.000000000000000000000000000000... = 2
2. 3.000000000000000000000000000000... = 3
3. 2.666666666666666666666666666666... =  $\frac{8}{3}$
4. 2.750000000000000000000000000000... =  $\frac{11}{4}$
5. 2.714285714285714285714285714285... =  $\frac{19}{7}$
6. 2.718750000000000000000000000000... =  $\frac{87}{32}$
7. 2.717948717948717948717948717948... =  $\frac{106}{39}$
8. 2.718309859154929577464788732394... =  $\frac{193}{71}$
9. 2.718279569892473118279569892473... =  $\frac{1264}{465}$
10. 2.718283582089552238805970149253... =  $\frac{1457}{536}$
11. 2.718281718281718281718281718281... =  $\frac{2721}{1001}$

12.  $2.718281835205992509363295880149\dots = \frac{23225}{8544}$
13.  $2.718281822943949711891042430591\dots = \frac{25946}{9545}$
14.  $2.718281828735695726684725523798\dots = \frac{49171}{18089}$
15.  $2.718281828445401318035025074172\dots = \frac{517656}{190435}$
16.  $2.718281828470583721777828930962\dots = \frac{566827}{208524}$
17.  $2.718281828458563411277850606202\dots = \frac{1084483}{398959}$
18.  $2.718281828459065114074529546648\dots = \frac{13580623}{4996032}$
19.  $2.718281828459028013207065591026\dots = \frac{14665106}{5394991}$
20.  $2.718281828459045851404621084949\dots = \frac{28245729}{10391023}$
21.  $2.718281828459045213521983758221\dots = \frac{410105312}{150869313}$
22.  $2.718281828459045254624795027092\dots = \frac{438351041}{161260336}$
23.  $2.718281828459045234757560631479\dots = \frac{848456353}{312129649}$
24.  $2.718281828459045235379013372772\dots = \frac{14013652689}{5155334720}$
25.  $2.71828182845904523534353532787\dots = \frac{14862109042}{5467464369}$
26.  $2.718281828459045235360753230188\dots = \frac{28875761731}{10622799089}$
27.  $2.718281828459045235360274593941\dots = \frac{534625820200}{196677847971}$
28.  $2.718281828459045235360299120911\dots = \frac{563501581931}{207300647060}$
29.  $2.718281828459045235360287179900\dots = \frac{1098127402131}{403978495031}$
30.  $2.718281828459045235360287478611\dots = \frac{22526049624551}{8286870547680}$
31.  $2.718281828459045235360287464726\dots = \frac{23624177026682}{8690849042711}$
32.  $2.718281828459045235360287471503\dots = \frac{46150226651233}{16977719590391}$
33.  $2.718281828459045235360287471349\dots = \frac{1038929163353808}{382200680031313}$
34.  $2.718281828459045235360287471355\dots = \frac{1085079390005041}{399178399621704}$
35.  $2.718281828459045235360287471352\dots = \frac{2124008553358849}{781379079653017}$
36.  $2.718281828459045235360287471352\dots = \frac{52061284670617417}{19152276311294112}$

One can with no problem compute much bigger convergents. Let's get the 200th convergent. It turns out to have the same first 268 digits after the decimal point as  $e-1$ . Higher convergents get more and more digits in proportion to their index: the 500th convergent already gets 799 digits correct! To allow speedy compilation of the source of this document when the need arises, I limit here to the 200th convergent.

```
\fdef\z {\xintCnToF {199}{\cn}}%
\begingroup\parindent 0pt \leftskip 2.5cm
\indent\llap {Numerator = }\printnumber{\xintNumerator\z}\par
\indent\llap {Denominator = }\printnumber{\xintDenominator\z}\par
\indent\llap {Expansion = }\printnumber{\xintTrunc{268}\z}\dots\par\endgroup
```

```
Numerator = 568964038871896267597523892315807875293889017667917446057232024547192296961118 2
23017524386017499531081773136701241708609749634329382906
```

```
Denominator = 331123817669737619306256360816356753365468823729314438156205615463246659728581 2
```

[TOC](#)

[TOC](#), [xint bundle](#), [xintkernel](#), [xintcore](#), [xint](#), [xintfrac](#), [xintbinhex](#), [xintgcd](#), [xintseries](#), [xintcfrac](#)

```
86546133769206314891601955061457059255337661142645217223
Expansion = 1.7182818284590452353602874713526624977572470936999595749669676277240766303535
475945713821785251664274274663919320030599218174135966290435729003342952605956
307381323286279434907632338298807531952510190115738341879307021540891499348841
675092447614606680822648001684774118...
```

One can also use a centered continued fraction: we get more digits but there are also more computations as the numerators may be either 1 or -1.



## Part III.

# The *xintexpr* and allied packages source code

17	An introduction and a brief timeline . . . . .	233
18	Package <i>xintkernel</i> implementation . . . . .	236
19	Package <i>xinttools</i> implementation . . . . .	259
20	Package <i>xintcore</i> implementation . . . . .	302
21	Package <i>xint</i> implementation . . . . .	359
22	Package <i>xintbinhex</i> implementation . . . . .	400
23	Package <i>xintgcd</i> implementation . . . . .	420
24	Package <i>xintfrac</i> implementation . . . . .	430
25	Package <i>xintseries</i> implementation . . . . .	525
26	Package <i>xintcfraction</i> implementation . . . . .	534
27	Package <i>xintexpr</i> implementation . . . . .	557
28	Package <i>xinttrig</i> implementation . . . . .	689
29	Package <i>xintlog</i> implementation . . . . .	712
30	Cumulative line and macro count . . . . .	750

## 17. An introduction and a brief timeline

This is 1.4o of 2025/09/06.

The `xintchanges.md` file, included in the CTAN upload, contains the complete list of changes relevant to user level since the initial release of the package:

`texdoc xintchanges.md`

It exists in HTML format at the package web page:

<https://jfbu.github.io/xint/CHANGES.html>

At 1.4m I added hyperlinks to the macro code. Each instance of a macro in the code is linked with target the location of its definition (via `\def` or `\let` or variants). This has been done via a heist on `doc` (v2 version) automated indexing which has been transformed here into automated hyperlinking. Furthermore the codeline where the macro is defined will link to its description in the user manual part of `xint.pdf`. In (optional build) `xintsource.pdf` the link is more modestly targeting the sectioning heading referencing the macro name, if available. In `xint.pdf` the macros documented in the user manual are marked with a *source* link on top of them, targeting their source code.

The sad truth however is that my code is poorly documented. The comments

- are often too scarce,
- have occasional excessive verbosity,
- and are generally inadequate or irrelevant.

The macro comments have had a distinct tendency to

record the changes across releases or even those occurring during pre-release development phase, rather than explaining the interface, or perhaps an algorithm. As I am aware of that, I have a mechanism of “private comments” which are removed by the `dtx` build script. But then I sometimes use it en masse as it would be too much work to clean-up the existing comments, and as a result the code is not commented at all anymore... A typical example is with `\xintiSquareRoot` which is amply documented in the private sources but only 10% of it could be of any value to any other reader than myself and it would be simply the description of what #1, #2, ... stand for. As a result I converted at some point everything into private comments. Extracting the useful parts describing the macro parameters and checking they are actually still valid would be very time-consuming. The real problem here is that the actual underlying algorithms are rarely if ever described.

- Release 1.4n of 2025/09/05 is mainly a maintenance release, after a few years in a dormant state. The long expected overhaul of floating point is again postponed.
  - `xintbinhex` handles the octal base, and can manage (much) larger inputs,
  - `0x`, `0o`, and `0b` prefixes, and `'` added to the `\xinteval` syntax,
  - `[h]`, `[o]`, `[b]` optional parameter of `\xintiieval`,
  - Babel active characters are auto-tamed in `\xinteval` (hotfix at 1.4o 2025/09/06 as the feature was only true with `\xintiieval`),
  - Compatibility with `OpTeX`,
  - Compatibility with `ConTeXt` (only `mkx1`).

The extensive hyperlinking added in 2022 to the docs is better shown to user because `xint.pdf` now (again) contains both the user manual and the commented source code. The packaging was trimmed.

- Release 1.4m of 2022/06/09 is mainly a documentation upgrade, which added hyperlinks inside the commented macro code, as well as from the user manual to the source code (in `xint-all.pdf`, which is an optional build). It adds compatibility with `miniltx`. It also inaugurates usage of the engine string comparison primitive.
- Release 1.4i of 2021/06/11: extension of the «simultaneous assignments» concept (backwards compatible).
- Release 1.4g of 2021/05/25: powers are now parsed in a right associative way. Removal of the single-character operators `&`, `|`, and `=` (deprecated at 1.1). Reformatted expandable error messages.

- Release 1.4e of 2021/05/05: logarithms and exponentials up to 62 digits, trigonometry still mainly done at high level but with guard digits so all digits up to the last one included can be trusted for faithful rounding and high probability of correct rounding.
- Release 1.4 of 2020/01/31: [xintexpr](#) overhaul to use `\expanded` based expansion control. Many new features, in particular support for input and output of nested structures. Breaking changes, main ones being the (provisory) drop of `x*[a, b,...]`, `x+[a, b,...]` et al. syntax and the requirement of `\expanded` primitive (currently required only by [xintexpr](#)).
- Release 1.3e of 2019/04/05: packages [xinttrig](#), [xintlog](#); `\xintdefefunc` ``non-protected'' variant of `\xintdeffunc` (at 1.4 the two got merged and `\xintdefefunc` became a deprecated alias for `\xintdeffunc`). Indices removed from [xintsource.pdf](#).
- Release 1.3d of 2019/01/06: fix of 1.2p bug for division with a zero dividend and a one-digit divisor, `\xinteval` et al. wrappers, `gcd()` and `lcm()` work with fractions.
- Release 1.3c of 2018/06/17: documentation better hyperlinked, indices added to [xintsource.pdf](#). Colon in `:=` now optional for `\xintdefvar` and `\xintdeffunc`.
- Release 1.3b of 2018/05/18: randomness related additions (still WIP).
- Release 1.3a of 2018/03/07: efficiency fix of the mechanism for recursive functions.
- Release 1.3 of 2018/03/01: addition and subtraction use systematically least common multiple of denominators. Extensive under-the-hood refactoring of `\xintNewExpr` and `\xintdeffunc` which now allow recursive definitions. Removal of 1.2o deprecated macros.
- Release 1.2q of 2018/02/06: fix of 1.2l subtraction bug in special situation; tacit multiplication extended to cases such as `10!20!30!`.
- Release 1.2p of 2017/12/05: maps `//` and `/:` to the floored, not truncated, division. Simultaneous assignments possible with `\xintdefvar`. Efficiency improvements in [xinttools](#).
- Release 1.2o of 2017/08/29: massive deprecations of those macros from [xintcore](#) and [xint](#) which filtered their arguments via `\xintNum`.
- Release 1.2n of 2017/08/06: improvements of [xintbinhex](#).
- Release 1.2m of 2017/07/31: rewrite of [xintbinhex](#) in the style of the 1.2 techniques.
- Release 1.2l of 2017/07/26: under the hood efficiency improvements in the style of the 1.2 techniques; subtraction refactored. Compatibility of most [xintfrac](#) macros with arguments using non-delimited `\the\numexpr` or `\the\mathcode` etc...
- Release 1.2i of 2016/12/13: under the hood efficiency improvements in the style of the 1.2 techniques.
- Release 1.2 of 2015/10/10: complete refactoring of the core arithmetic macros and faster `\xintexpr` parser.
- Release 1.1 of 2014/10/28: extensive changes in [xintexpr](#). Addition and subtraction do not multiply denominators blindly but sometimes produce smaller ones. Also with that release, packages [xintkernel](#) and [xintcore](#) got extracted from [xinttools](#) and [xint](#).
- Release 1.09g of 2013/11/22: the [xinttools](#) package is extracted from [xint](#); addition of `\xint-loop` and `\xintiloop`.
- Release 1.09c of 2013/10/09: `\xintFor`, `\xintNewNumExpr` (ancestor of `\xintNewExpr`/`\xintdeffunc` mechanism).

## TOC

*TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog*

- Release 1.09a of 2013/09/24: support for functions by `xintexpr`.
- Release 1.08 of 2013/06/07: the `xintbinhex` package.
- Release 1.07 of 2013/05/25: support for floating point numbers added to `xintfrac` and first release of the `xintexpr` package (provided `\xintexpr` and `\xintfloatexpr`).
- Release 1.04 of 2013/04/25: the `xintcfrac` package.
- Release 1.03 of 2013/04/14: the `xintfrac` and `xintseries` packages.
- Release 1.0 of 2013/03/28: initial release of the `xint` and `xintgcd` packages.

## 18. Package [xintkernel](#) implementation

.1	Catcodes, $\varepsilon$ -TeX and reload detection . . .	236	.15	<code>\xintstrcmp</code> . . . . .	248
.1.1	<code>\XINTrestorecatcodes</code> , <code>\XINTsetcatcodes</code> , <code>\XINTrestorecatcodesendinginput</code> . . . . .	237	.16	<code>\xintresettimer</code> , <code>\xintelapsedtime</code> , <code>\xinttheseconds</code> . . . . .	248
.2	Package identification . . . . .	239	.17	<code>\xintReverseOrder</code> . . . . .	249
.3	Constants . . . . .	240	.18	<code>\xintLength</code> . . . . .	250
.4	Token management utilities . . . . .	241	.19	<code>\xintLastItem</code> . . . . .	250
.5	“gob til” macros and UD style fork . . . . .	242	.20	<code>\xintFirstItem</code> . . . . .	251
.6	<code>\xint_afterfi</code> . . . . .	243	.21	<code>\xintLastOne</code> . . . . .	251
.7	<code>\xint_bye</code> , <code>\xint_Bye</code> . . . . .	243	.22	<code>\xintFirstOne</code> . . . . .	252
.8	<code>\xintdothis</code> , <code>\xintorthat</code> . . . . .	243	.23	<code>\xintLengthUpTo</code> . . . . .	252
.9	<code>\xint_zapspace</code> . . . . .	243	.24	<code>\xintreplicate</code> , <code>\xintReplicate</code> . . . . .	253
.10	<code>\odef</code> , <code>\oodef</code> , <code>\fdef</code> . . . . .	244	.25	<code>\xintgobble</code> , <code>\xintGobble</code> . . . . .	254
.11	<code>\xintMessage</code> , <code>\ifxintverbose</code> . . . . .	244	.26	Random number generation . . . . .	256
.12	<code>\ifxintglobaldefs</code> , <code>\XINT_global</code> . . . . .	244	.26.1	<code>\xint_texuniformdeviate</code> . . . . .	257
.13	(WIP) Expandable error message . . . . .	245	.26.2	<code>\xint_texuniformdeviate_dgts</code> . . . . .	257
.14	<code>\xint_noxpd</code> (for <code>conTeXt-mkx1</code> compatibility) 247		.26.3	<code>\xintUniformDeviate</code> . . . . .	258

This package provides the common minimal code base for loading management and catcode control and also a few programming utilities. With 1.2 a few more helper macros and all `\chardef`'s have been moved here. The package is loaded by both `xintcore.sty` and `xinttools.sty` hence by all other packages.

Modified at 1.1 (2014/10/28). Separated package.

Modified at 1.2i (2016/12/13). `\xintreplicate`, `\xintgobble`, `\xintLengthUpTo` and `\xintLastItem`, and faster `\xintLength`.

Modified at 1.3b (2018/05/18). `\xintUniformDeviate`.

Modified at 1.4 (2020/01/31). `\xintReplicate`, `\xintGobble`, `\xintLastOne`, `\xintFirstOne`.

Modified at 1.4l (2022/05/29). Fix the 1.4 added bug that `\XINTrestorecatcodes` forgot to restore the catcode of `^^A` which is set to 3 by `\XINTsetcatcodes`.

Modified at 1.4m (2022/06/10). Fix incompatibility under  $\varepsilon$ -TeX with `minilTeX`, if latter was loaded before `xintexpr`. The fix happens here because it relates to matters of `\ProvidesPackage`.

### 18.1. Catcodes, $\varepsilon$ -TeX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified. The method for catcodes was also initially directly inspired by these packages.

1.4l replaces Info level user messages issued in case of problems such as `\numexpr` not being available with Warning level messages (in the LaTeX terminology). Should arguably be Error level in that case.

`xintkernel.sty` was the only `xint` package emitting such an Info, now Warning in case of being loaded twice (via `\input` in non-LaTeX). This was probably a left-over from initial development stage of the loading architecture for debugging. Starting with 1.4l, it will abort input silently in such case.

Also at 1.4l I refactored a bit the loading code in the `xint*sty` files for no real reason other than losing time.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2   \catcode13=5      % ^^M
3   \endlinechar=13 %
4   \catcode123=1     % {
5   \catcode125=2     % }
```

```

6 \catcode44=12 % ,
7 \catcode46=12 % .
8 \catcode58=12 % :
9 \catcode94=7 % ^
10 \def\space{ }\newlinechar10
11 \let\z\relax
12 \expandafter\ifx\csname numexpr\endcsname\relax
13 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
14 \immediate\write128{^^JPackage xintkernel Warning:^^J%
15 \space\space\space\space
16 \numexpr not available, aborting input.^^J}%
17 \else
18 \PackageWarningNoLine{xintkernel}{\numexpr not available, aborting input}%
19 \fi
20 \def\z{\endgroup\endinput}%
21 \else
22 \expandafter\ifx\csname XINTsetupcatcodes\endcsname\relax
23 \else
24 \def\z{\endgroup\endinput}%
25 \fi
26 \fi
27 \ifx\z\relax\else\expandafter\z\fi%
```

#### 18.1.1. `\XINTrestorecatcodes`, `\XINTsetcatcodes`, `\XINTrestorecatcodesendinput`

**Modified at 1.4e (2021/05/05).** Renamed `\XINT{set,restore}catcodes` to be without underscores, to facilitate the reloading process for `xintlog.sty` and `xinttrig.sty` in uncontrolled contexts.

**Modified at 1.4l (2022/05/29).** Fix the 1.4 bug of omission of `\catcode1` restore.

Reordered all catcodes assignments for easier maintenance and dropped most disparate indications of which packages make use of which settings.

The `\XINTrestorecatcodes` is somewhat misnamed as it is more a template to be used in an `\edef` to help define actual catcode restoring macros.

However `\edef` needs usually `{` and `}` so there is a potential difficulty with telling people to do `\edef\myrestore{\XINTrestorecatcodes}`, and I almost added at 1.4l some `\XINTsettorestore:#1->\edef#1{\XINTrestorecatcodes}` but well, this is not public interface anyhow. The reloading method of `xintlog.sty` and `xinttrig.sty` does protect itself though against such unreal usage possibility with non standard `{` or `}`.

Removed at 1.4l the `\XINT_setcatcodes` and `\XINT_restorecatcodes` not used anywhere now. Used by old version of `xintsession.tex`, but not anymore since a while.

**Modified at 1.4n (2025/09/05).** Compatibility with `OpTeX`. It has a cactode 11 letter, but `\abc_` or `\abc_d` will be interpreted un expectedly (but not `\abc_de`). So we must make sure this is deactivated during the whole duration of loading the `xint` modules (perhaps it is mainly `xinttools` which uses `\XINT_x`, `\XINT_y`, which is problematic).

We need catcode letter `_` for tokenization of `\PrepareCatcodes` to let it define `\XINTrestorecatcodes` conveniently for its usage in an `\edef` to query at that time the `_` status in `OpTeX`.

```

28 \catcode95=11 % _
29 \def\PrepareCatcodes
30 {%
31 \endgroup
32 \def\XINTrestorecatcodes
33 {% prepared for use in \edef
34 \catcode0=\the\catcode0 % ^^@
```

```

35      \catcode1=\the\catcode1      % ^^A
36      \catcode13=\the\catcode13    % ^^M
37      \catcode32=\the\catcode32    % <space>
38      \catcode33=\the\catcode33    % !
39      \catcode34=\the\catcode34    % "
40      \catcode35=\the\catcode35    % #
41      \catcode36=\the\catcode36    % $
42      \catcode38=\the\catcode38    % &
43      \catcode39=\the\catcode39    % '
44      \catcode40=\the\catcode40    % (
45      \catcode41=\the\catcode41    % )
46      \catcode42=\the\catcode42    % *
47      \catcode43=\the\catcode43    % +
48      \catcode44=\the\catcode44    % ,
49      \catcode45=\the\catcode45    % -
50      \catcode46=\the\catcode46    % .
51      \catcode47=\the\catcode47    % /
52      \catcode58=\the\catcode58    % :
53      \catcode59=\the\catcode59    % ;
54      \catcode60=\the\catcode60    % <
55      \catcode61=\the\catcode61    % =
56      \catcode62=\the\catcode62    % >
57      \catcode63=\the\catcode63    % ?
58      \catcode64=\the\catcode64    % @
59      \catcode91=\the\catcode91    % [
60      \catcode93=\the\catcode93    % ]
61      \catcode94=\the\catcode94    % ^
62      \catcode95=\the\catcode95    % _
63      \catcode96=\the\catcode96    % `
64      \catcode123=\the\catcode123  % {
65      \catcode124=\the\catcode124  % |
66      \catcode125=\the\catcode125  % }
67      \catcode126=\the\catcode126  % ~
68      \endlinechar=\the\endlinechar\relax
69      \ifdefined\_ifmathsb\_ifmathsb\noexpand\_mathsbon\_fi\fi %
70  }%

```

The `\noexpand` here before `\endinput` is required. This feels to me a bit surprising, but is a fact, and the source of this must be in the `\edef` implementation but I have not checked it out at this time.

**Modified at 1.4n (2025/09/05).** Compatibility with OpTeX.

```

71      \edef\XINTrestorecatcodesendinput
72      {%
73          \XINTrestorecatcodes\noexpand\endinput %
74      }%
75      \def\XINTsetcatcodes
76      {% standard settings with a few xint*sty specific ones
77          \catcode0=12      % for \romannumeral`&&@
78          \catcode1=3       % for safe separator &&A
79          \catcode13=5      % ^^M
80          \catcode32=10     % <space>
81          \catcode33=12     % ! but used as LETTER inside xintexpr.sty
82          \catcode34=12     % "

```

```

83      \catcode35=6      % #
84      \catcode36=3      % $
85      \catcode38=7      % & SUPERSCRIPT for && as replacement of ^^
86      \catcode39=12     % '
87      \catcode40=12     % (
88      \catcode41=12     % )
89      \catcode42=12     % *
90      \catcode43=12     % +
91      \catcode44=12     % ,
92      \catcode45=12     % -
93      \catcode46=12     % .
94      \catcode47=12     % /
95      \catcode58=11     % : LETTER
96      \catcode59=12     % ;
97      \catcode60=12     % <
98      \catcode61=12     % =
99      \catcode62=12     % >
100     \catcode63=11     % ? LETTER
101     \catcode64=11     % @ LETTER
102     \catcode91=12     % [
103     \catcode93=12     % ]
104     \catcode94=11     % ^ LETTER
105     \catcode95=11     % _ LETTER
106     \catcode96=12     % `
107     \catcode123=1      % {
108     \catcode124=12     % |
109     \catcode125=2      % }
110     \catcode126=3      % ~ MATH SHIFT
111     \endlinechar=13 %
112     \ifdefined\mathsboff\mathsboff\fi % Compatibility with OpTeX
113 }%
114 \XINTsetcatcodes
115 }%
116 \PrepareCatcodes
    Other modules could possibly be loaded under a different catcode regime. (or with a different
    status of _ under OpTeX).
117 \def\XINTsetupcatcodes {% for use by other modules
118     \edef\XINTrestorecatcodesendinput
119     {%
120         \XINTrestorecatcodes\noexpand\endinput %
121     }%
122     \XINTsetcatcodes
123 }%
```

## 18.2. Package identification

Inspired from HEIKO OBERDIEK's packages.

**Modified at 1.09b (2013/10/03).** Re-usability in the other modules. Also I assume now that if `\ProvidesPackage` exists it then does define `\ver@<pkgname>.sty`, code of H0 for some reason escaping me (compatibility with LaTeX 2.09 or other things ??) seems to set extra precautions. [nine years later I understood my mistake, see below].

**Modified at 1.09c (2013/10/09).** Usage of  $\varepsilon$ -TeX `\ifdefined`.



**Modified at 1.4m (2022/06/10).** Nine years too late, I understand that the HO “extra precautions” were there for some respectable reasons including `etex+miniltx` and surely other things I can not imagine. So let's now make sure `\ver@xintkernel.sty` and friends get defined on load, even if `\ProvidesPackage` exists! However I remain careless in using `\ifdefined` which could be fooled if some previous macro file ended up testing for `\ProvidesPackage` in a way letting it to `\relax`. I do not test for that. If I fixed that carelessness here I would have to fix it in other places where I use similarly `\ifdefined\RequirePackage` or `\ifdefined\PackageWarning` or whatever.

```

124 \ifdefined\ProvidesPackage
125   \def\XINT_providespackage\ProvidesPackage#1[#2]{%
126     \ProvidesPackage{#1}[#2]%
127     \expandafter\ifx\csname ver@#1.sty\endcsname\relax
128       \expandafter\xdef\csname ver@#1.sty\endcsname{#2}%
129     \fi
130   }%
131 \else
132   \def\XINT_providespackage\ProvidesPackage#1[#2]{%
133     \immediate\write-1{Package: #1 #2}%
134     \expandafter\xdef\csname ver@#1.sty\endcsname{#2}%
135   }%
136 \fi
137 \XINT_providespackage
138 \ProvidesPackage {xintkernel}%
139 [2025/09/06 v1.4o Paraphernalia for the xint packages (JFB)]%

```

### 18.3. Constants

```

140 \chardef\xint_c_      0
141 \chardef\xint_c_i     1
142 \chardef\xint_c_ii    2
143 \chardef\xint_c_iii   3
144 \chardef\xint_c_iv    4
145 \chardef\xint_c_v     5
146 \chardef\xint_c_vi    6
147 \chardef\xint_c_vii   7
148 \chardef\xint_c_viii  8
149 \chardef\xint_c_ix    9
150 \chardef\xint_c_x    10
151 \chardef\xint_c_xii   12
152 \chardef\xint_c_xiv   14
153 \chardef\xint_c_xvi   16
154 \chardef\xint_c_xvii  17
155 \chardef\xint_c_xviii 18
156 \chardef\xint_c_xx    20
157 \chardef\xint_c_xxii  22
158 \chardef\xint_c_ii^v   32
159 \chardef\xint_c_ii^vi  64
160 \chardef\xint_c_ii^vii 128
161 \mathchardef\xint_c_ii^viii 256
162 \mathchardef\xint_c_ii^ix 512
163 \mathchardef\xint_c_ii^xii 4096

```

Some of these usages of `\newcount` were in `xintcore` or in `xint` or in `xintbinhex` possibly conditionally on whether (pdf)`\uniformdeviate` is available. At 1.4n, let's not bother with outdated

restrictions of  $\TeX$ .

```

164 \ifdefined\m@ne\let\xint_c_mone\m@ne
165 \else\csname newcount\endcsname\xint_c_mone \xint_c_mone -1 %
166 \fi
167 \mathchardef\xint_c_x^iv 10000
168 \newcount\xint_c_x^v \xint_c_x^v 100000
169 \newcount\xint_c_x^viii \xint_c_x^viii 100000000
170 \newcount\xint_c_x^ix \xint_c_x^ix 1000000000
171 \newcount\xint_c_x^viii_mone \xint_c_x^viii_mone 99999999
172 \newcount\xint_c_nine_x^viii \xint_c_nine_x^viii 900000000
173 \newcount\xint_c_xi_e_viii_mone \xint_c_xi_e_viii_mone 1099999999
174 \newcount\xint_c_xii_e_viii \xint_c_xii_e_viii 1200000000

```

**Modified at 1.4n (2025/09/05).** For some reason this next one used to be defined by `\newcount` but `\mathchardef` is ok.

```

175 \mathchardef\xint_c_ii^xiv 16384 % "4000, 2**14
176 \newcount\xint_c_ii^xv \xint_c_ii^xv 32768 % 2**15
177 \newcount\xint_c_ii^xvi \xint_c_ii^xvi 65536 % 2**16
178 \newcount\xint_c_ii^xxi \xint_c_ii^xxi 2097152 % "200000, 2**21

```

## 18.4. Token management utilities

**Added at 1.2 (2015/10/10).** Check if `\empty` and `\space` have their standard meanings and raise a warning if not.

**Modified at 1.4 (2020/01/31).** Warn user if needed, and force then `\empty` and `\space` to have their standard meanings. This will be triggered even if the sole difference is that they are `\long`.

**Modified at 1.4n (2025/09/05).** The Warning used to be emitted only to the log file, make it go to console output as well. But why do I spend time on such silly things.

```

179 \def\XINT_tmpa { }%
180 \ifx\XINT_tmpa\space\else
181 \immediate\write128{Package xintkernel Warning:}%
182 \immediate\write128{The \string\space\XINT_tmpa macro does not have its
183 meaning as in Plain or LaTeX, but is:}%
184 \immediate\write128{\XINT_tmpa\XINT_tmpa\XINT_tmpa\XINT_tmpa\meaning\space.}%
185 \let\space\XINT_tmpa
186 \immediate\write128{Forcing it to be the usual one. Fingers crossed.}%
187 \fi
188 \def\XINT_tmpa {}%
189 \ifx\XINT_tmpa\empty\else
190 \immediate\write128{Package xintkernel Warning:}%
191 \immediate\write128{The \string\empty\space macro does not have its
192 meaning as in Plain or LaTeX, but is:}%
193 \immediate\write128{\space\space\space\space\meaning\empty.}%
194 \let\empty\XINT_tmpa
195 \immediate\write128{Forcing it to be the usual one. Fingers crossed.}%
196 \fi
197 \let\XINT_tmpa\relax
198 \let\xint_gobble_\empty
199 \long\def\xint_gobble_i #1{%
200 \long\def\xint_gobble_ii #1#2{%
201 \long\def\xint_gobble_iii #1#2#3{%
202 \long\def\xint_gobble_iv #1#2#3#4{%

```

```

203 \long\def\xint_gobble_v    #1#2#3#4#5{%
204 \long\def\xint_gobble_vi   #1#2#3#4#5#6{%
205 \long\def\xint_gobble_vii  #1#2#3#4#5#6#7{%
206 \long\def\xint_gobble_viii #1#2#3#4#5#6#7#8{%

```

**Modified at 1.3b (2018/05/18).** Moved here `\xint_gobandstop...` macros because this is handy for `\xintRandomDigits`.

For legacy reasons most top level macros use `\romannumeral0` trigger. This is stopped by a space token. Later in the history of the package `\romannumeral`&&@` was used with `&` of catcode 7. Also there are a few instances of `\romannumeral` ended by `\z@`. But `\romannumeral0` remains the publicly documented one, with CamelCase macros using it as prefix to lowercased macros.

```

207 \let\xint_gob_andstop_\space
208 \long\def\xint_gob_andstop_i    #1{ }%
209 \long\def\xint_gob_andstop_ii   #1#2{ }%
210 \long\def\xint_gob_andstop_iii  #1#2#3{ }%
211 \long\def\xint_gob_andstop_iv   #1#2#3#4{ }%
212 \long\def\xint_gob_andstop_v    #1#2#3#4#5{ }%
213 \long\def\xint_gob_andstop_vi   #1#2#3#4#5#6{ }%
214 \long\def\xint_gob_andstop_vii  #1#2#3#4#5#6#7{ }%
215 \long\def\xint_gob_andstop_viii #1#2#3#4#5#6#7#8{ }%
216 \let\xint_stop_aftergobble\xint_gob_andstop_i
217 \long\def\xint_firstofone    #1{#1}%
218 \long\def\xint_firstoftwo    #1#2{#1}%
219 \long\def\xint_secondoftwo    #1#2{#2}%
220 \long\def\xint_stop_atfirstofone #1{ #1}%
221 \long\def\xint_stop_atfirstoftwo #1#2{ #1}%
222 \long\def\xint_stop_atsecondoftwo #1#2{ #2}%
223 \long\def\xint_exchangetwo_keepbraces #1#2{{#2}{#1}}%

```

Moved here from `xint` at 1.4n.

```

224 \long\def\xint_firstofthree    #1#2#3{#1}%
225 \long\def\xint_secondofthree   #1#2#3{#2}%
226 \long\def\xint_thirdofthree    #1#2#3{#3}%
227 \long\def\xint_stop_atfirstofthree #1#2#3{ #1}%
228 \long\def\xint_stop_atsecondofthree #1#2#3{ #2}%
229 \long\def\xint_stop_atthirdofthree #1#2#3{ #3}%

```

## 18.5. “gob til” macros and UD style fork

```

230 \long\def\xint_gob_til_R #1\R {}%
231 \long\def\xint_gob_til_W #1\W {}%
232 \long\def\xint_gob_til_Z #1\Z {}%
233 \long\def\xint_gob_til_zero #10{}%
234 \long\def\xint_gob_til_one  #11{}%
235 \long\def\xint_gob_til_zeros_iii #1000{}%
236 \long\def\xint_gob_til_zeros_iv #10000{}%
237 \long\def\xint_gob_til_eightzeroes #100000000{}%
238 \long\def\xint_gob_til_dot #1.{}%
239 \long\def\xint_gob_til_G #1G{}%
240 \long\def\xint_gob_til_minus #1-{}%
241 \long\def\xint_UDzerominusfork #10-#2#3\krof {#2}%
242 \long\def\xint_UDzerofork #10#2#3\krof {#2}%
243 \long\def\xint_UDsignfork #1-#2#3\krof {#2}%
244 \long\def\xint_UDwfork #1\W#2#3\krof {#2}%

```

```

245 \long\def\xint_UDXINTWfork      #1\XINT_W#2#3\krof {#2}%
246 \long\def\xint_UDzerosfork      #100#2#3\krof {#2}%
247 \long\def\xint_UDonezerofork    #110#2#3\krof {#2}%
248 \long\def\xint_UDsignsfork      #1--#2#3\krof {#2}%
249 \let\xint:\char
250 \long\def\xint_gob_til_xint:#1\xint:{}%
251 \long\def\xint_gob_til_#1^#1{}%
252 \def\xint_bracedstopper{\xint:}%
253 \long\def\xint_gob_til_exclam #1!{}% This ! has catcode 12
254 \long\def\xint_gob_til_sc #1;{}%

```

## 18.6. \xint\_afterfi

```

255 \long\def\xint_afterfi #1#2\fi {\fi #1}%

```

## 18.7. \xint\_bye, \xint\_Bye

Modified at 1.09 (2013/09/23). [\xint\\_bye](#)

Modified at 1.2i (2016/12/13). [\xint\\_Bye](#) for [\xintDSRr](#) and [\xintRound](#). Also [\xint\\_stop\\_after](#) [bye](#).

```

256 \long\def\xint_bye #1\xint_bye {}%
257 \long\def\xint_Bye #1\xint_bye {}%
258 \long\def\xint_stop_afterbye #1\xint_bye { }%

```

## 18.8. \xintdothis, \xintorthat

Modified at 1.1 (2014/10/28).

Modified at 1.2 (2015/10/10). Names without underscores.

To be used this way:

```

\xif..\xint_dothis{...}\fi
\xif..\xint_dothis{...}\fi
\xif..\xint_dothis{...}\fi
...more such...
\xint_orthat{...}

```

Ancient testing indicated it is more efficient to list first the more improbable clauses.

```

259 \long\def\xint_dothis #1#2\xint_orthat #3{\fi #1}% 1.1
260 \let\xint_orthat \xint_firstofone
261 \long\def\xintdothis #1#2\xintorthat #3{\fi #1}%
262 \let\xintorthat \xint_firstofone

```

## 18.9. \xint\_zapspace

Modified at 1.1 (2014/10/28). This little (quite fragile in the normal sense i.e. non robust in the normal sense of programming lingua) utility zaps leading, intermediate, trailing, spaces in completely expanding context ([\edef](#), [\csname...](#)[\endcsname](#)).

Usage: [\xint\\_zapspace](#) [foo](#)<space>[\xint\\_gobble\\_i](#)

Explanation: if there are leading spaces, then the first [#1](#) will be empty, and the first [#2](#) being undelimited will be stripped from all the remaining leading spaces, if there was more than one to start with. Of course brace-stripping may occur. And this iterates: each time a [#2](#) is removed, either we then have spaces and next [#1](#) will be empty, or we have no spaces and [#1](#) will end at the first space. Ultimately [#2](#) will be [\xint\\_gobble\\_i](#).

The `\zap@spaces` of LaTeX2e handles unexpectedly things such as

```
\zap@spaces 1 {22} 3 4 \@empty
```

(spaces are not all removed). This does not happen with `\xint_zapspaces`.

But for example `\foo{aa} {bb} {cc}` where `\foo` is a macro with three non-delimited arguments breaks expansion, as expansion of `\foo` will happen with `\xint_zapspaces` still around, and even if it wasn't it would have stripped the braces around `{bb}`, certainly breaking other things.

Despite such obvious shortcomings it is enough for our purposes. It is currently used by `xintexpr` at various locations e.g. cleaning up optional argument of `\xintiexpr` and `\xintfloatexpr`; maybe in future internal usage will drop this in favour of a more robust utility.

Modified at 1.2e (2015/11/22). `\xint_zapspaces_o`.

Modified at 1.2i (2016/12/13). Made `\long`.

ATTENTION THAT `xinttools` HAS AN `\xintzapspaces` WHICH SHOULD NOT GET CONFUSED WITH THIS ONE.

```
263 \long\def\xint_zapspaces #1 #2{#1#2\xint_zapspaces }% 1.1
264 \long\def\xint_zapspaces_o #1{\expandafter\xint_zapspaces#1 \xint_gobble_i}%
```

## 18.10. `\odef`, `\oodef`, `\fdef`

May be prefixed with `\global`. No parameter text.

```
265 \def\xintodef #1{\expandafter\def\expandafter#1\expandafter }%
266 \def\xintoodef #1{\expandafter\expandafter\expandafter\def
267     \expandafter\expandafter\expandafter#1%
268     \expandafter\expandafter\expandafter }%
269 \def\xintfdef #1#2%
270     {\expandafter\def\expandafter#1\expandafter{\romannumeral`&&@#2}}%
271 \ifdefined\odef\else\let\odef\xintodef\fi
272 \ifdefined\oodef\else\let\oodef\xintoodef\fi
273 \ifdefined\fdef\else\let\fdef\xintfdef\fi
```

## 18.11. `\xintMessage`, `\ifxintverbose`

Modified at 1.2c (2015/11/16). For use by `\xintdefvar` and `\xintdefunc` of `xintexpr`.

Modified at 1.2e (2015/11/22). Uses `\write128` rather than `\write16` for compatibility with future extended range of output streams, in LuaTeX in particular.

Modified at 1.3e (2019/04/05). Set the `\newlinechar`.

```
274 \edef\XINT_fourspaces{\space\space\space\space}%
275 \def\xintMessage #1#2#3{%
276     \edef\XINT_newlinechar{\the\newlinechar}%
277     \newlinechar10
278     \immediate\write128{Package #1 #2: (on line \the\inputlineno)}%
279     \immediate\write128{\XINT_fourspaces#3}%
280     \newlinechar\XINT_newlinechar\space
281 }%
282 \newif\ifxintverbose
```

## 18.12. `\ifxintglobaldefs`, `\XINT_global`

Modified at 1.3c (2018/06/17).

```
283 \newif\ifxintglobaldefs
284 \def\XINT_global{\ifxintglobaldefs\global\fi}%
```

## 18.13. (WIP) Expandable error message

**Modified at 1.21 (2017/07/26).** But really belongs to next major release beyond 1.3. Basically copied over from l3kernel code. Using `\ ! /` control sequence, which must be left undefined. `\xintError:` would be 6 letters more.

**Modified at 1.4 (2020/01/31).** Finally rather than `\ ! /` I use `\xint/`.

**Modified at 1.4g (2021/05/25).** Rewrote to use not an undefined control sequence but trigger "Use of `\xint/` doesn't match its definition." message.

**Modified at 1.4g (2021/05/25).** Things evolve fast and I switch to a third method which will exploit "Paragraph ended before `\foo` was complete" style error. See

<https://github.com/latex3/latex3/issues/931#issuecomment-845367201>

However I can not fully exploit this because xint may be used with Plain etex which does not set `\newlinechar`. I can only use a poorman version with no usage of `^^J`. Also `xintsession` could use the `^^J`, maybe I will integrate it there.

I. Explanations on 2021/05/19 and 2021/05/20 before final change

First I tried out things with undefined control sequence such as

```
\
  an error was reported by xint ...
```

whose output produces a nice symmetrical display with no `\`, and with `...` both on left and right but this reduces drastically the available space for the actual error context. No go. But see 2021/05/20 update below!

Having replaced `\xint/` by `"\xint "`, I next opted provisorily for `"\Hit RET at ?"` control sequence, despite it being quite longer. And then I thought about using `"\ xint error"`, possibly with an included `^^J` in the name, or in the context.

I experimented with `^^J` in the context. But the context size is much constrained, and when `\errorcontextlines` is at its default value of 5 for etex, not -1 as done by LaTeX, having the info shifted to the right makes it actually more visible. (however I have now updated `xintsession` to 0.2b which sets `\errorcontextlines` to 0)

So I was finally back here to square one, apart from having replaced `"\xint/"` by the more longish `"\ xint error"`, hesitating with `"\xintinterrupt"`...

Then I had the idea to replace the undefined control sequence method by a method with a macro `\foo` defined as `\def\foo.{}{}` but used as `\foo<space>` for example. This gives something like this (the first line will be otherwise if engine is run with `-file-line-error`):

```
! Use of \xint/ doesn't match its definition.
```

```
<argument> \xint/
```

```
Ooops, looks like we are missing a ] (hit RET)
```

`\xint/<space>` (where the space is the unexpected token, the definition expecting rather a full stop) makes for 7 characters to compare to `\ xint error` which had 12, so I gained back 5.

Back to `^^J`: I had overlooked that TeX in the first part of the error message will display `\macro` fully, so inserting `^^J` in its name allows arbitrarily long expandable error messages... as pointed out by BLF in latex3/issues#931 as I read on the morning of 2021/05/20. This is very nice but requires to predefine control sequences for each message, and also the actual arguments #1, #2, ... values can appear only in the context.

And the situation with `^^J` is somewhat complicated:

`xintsession` sets the `\newlinechar` to 10, but this is not the case with bare usage of `xintexpr` with etex. And this matters. To discuss `^^J` we have to separate two locations:

- it appears in the control sequence name,
- or in the context (which itself has two parts)

1) When in the context, what happens with `^^J` is independent of the setting of `\newlinechar`, and with TeXLive pdflatex the `^^J` will induce a linebreak, but with xelatex it must be used with option `-8bit`.

2) When in the control sequence name the behaviour in log/terminal of `^^J` is influenced by the setting of `\newlinechar`. Although with pdflatex it will always induce a linebreak, the actual

count of characters where TeX will forcefully break is influenced by whether `^^J` is or not `\newlinechar`. And with xelatex if it is `\newlinechar`, it does not depend then if -8bit or not, but if not `\newlinechar` then it does and TeX forceful breaks also change as for pdflatex.

So, the control sequence name trick can be used to obtain arbitrarily long messages, but the `\newlinechar` must be set.

And in the context, we can try to insert some `^^J` but this would need with xetex the -8bit option, and anyhow the context size is limited, and there is apparently no trick to get it larger.

So, in view of all the above I decided not to use `^^J` (rather `&&J` here) at all, whether here in the control sequence or the context or inserted in `\XINT_signalcondition` in the context!

I also have a problem with usage from `bnumexpr` or `polexpr` for example, they would need their own to avoid perhaps displaying `\xint/` or analogous.

II. Finally I modified again the method (completely, and no more need for funny catcode 7 space as delimiter) as this allows a longer context message, starting at start of line, and which obeys `^^J` if `\newlinechar` is set to it. It also allows to incorporate non-limited generic explanations as a postfix, with linebreaks if `\newlinechar` is known.

But as `xintexpr` can be used with Plain+etex which does not set the `\newlinechar`, I can't use `^^J` out of the box. I can in `xintsession`. What I decided finally is to make a conditional definition here.

In both cases I include the "hit RET" (how rather "hit <return>") in the control sequence name serving to both provide extra information and trigger the error from being defined short and finding a `\par`.

The maximal size was increased from 48 characters (method with `\xint/` being badly delimited), to now 55 characters (using `"! xint error:<^^J or space>"` as prefix to the message). Longer messages are truncated at 56 characters with an appended `"\ETC."`.

As it is late on this 2021/05/20, and in order to not have to change all usages, I keep `\XINT_signalcondition` (in `xintcore`) as a one argument macro for time being, so will not include a more specific module name.

The `\par` token has a special role here, and can't be (I)nserted without damage, but who would want to insert it in an expandable computation anyhow... and I don't need it in my custom error messages for sure.

On 2021/05/21 I add a test about `\newlinechar` at time of package loading, and make two distinct definitions: one using `^^J` in the control sequence, the other not using it.

The `-file-line-error` toggle makes it impossible to control if the line-break on first line will match next lines. In the `^^J` branch I insert `"| "` (no, finally `" "` with two spaces) at start of continuation lines. Also I preferred to ensure a good-looking first line break for the case it starts with a `"! Paragraph ended ..."` because a priori error messages will be read if `-file-line-error` was emitted only a fortiori (this toggle suggests some IDE launched TeX and probably `-interaction=nonstopmode`).

I will perhaps make another definition in `xintsession` (it currently loads `xintexpr` prior to having set the `\newlinechar`, so the no `^^J` definition will be used, if nothing else is modified there).

With some hesitation I do not insert a `^^J` after `"! xint error:"`, as Emacs/AucTeX will display only the first line prominently and then the rest (which is in `file:line:error` mode) in one block under `"--- TeX said ---"`. I use the `^^J` only in the generic helper message embedded in the control sequence. The cases with or without `\newlinechar` being 10 diverge a bit, as in the latter case I had to ensure acceptable linebreaks at 79 chars, and I did that first and then had spent enough time on the matter not to add more to backport the latest `^^J` style message.

**Modified at 1.4m (2022/06/10).** Shorten the error message. I am always too verbose initially.

**Modified at 1.4n (2025/09/05).** Let not use explicit `\par` token as delimiter, but implicit one from empty line. This is for compatibility with OpTeX. Indeed, I observed that with OpTeX, `{\foo o\par}` with `\foo` being short reports an extra `}` rather than saying ```Paragraph ended before \foo was complete.''`.



```

285 \ifnum\newlinechar=10
286 \expandafter\def\csname
287 xint<...> is done, but will resume:&&J \space
288 hit <return> at the ? prompt to try fixing the error above&&J \space
289 which has been encountered before expansion\endcsname
290 #1\xint:{}%
291 \def\XINT_expandableerror#1{%
292 \def\XINT_expandableerror##1{%
293   \expandafter
294   \XINT_expandableerrorcontinue
295   #1! xint error: ##1%
296
297 }}\expandafter\XINT_expandableerror\csname
298 xint<...> is done, but will resume:&&J \space
299 hit <return> at the ? prompt to try fixing the error above&&J \space
300 which has been encountered before expansion\endcsname
301 \else
302 \expandafter\def\csname
303 xint<...> is done, but will resume: hit <return> at \space
304 the ? prompt to try fixing the error encountered before expansion\endcsname
305 #1\xint:{}%
306 \def\XINT_expandableerror#1{%
307 \def\XINT_expandableerror##1{%
308   \expandafter
309   \XINT_expandableerrorcontinue
310   #1! xint error: ##1%
311
312 }}\expandafter\XINT_expandableerror\csname
313 xint<...> is done, but will resume: hit <return> at \space
314 the ? prompt to try fixing the error encountered before expansion\endcsname
315 \fi
316 \def\XINT_expandableerrorcontinue#1%
317
318 {#1}%

```

## 18.14. \xint\_noxpd (for contex-mkx1 compatibility)

Added at 1.4n (2025/09/05).

The  $\text{\LaTeX}$  code (in `l3names.dtx`) uses `ConTeXt`'s `\normalunexpanded` and `\normalexpanded` which appear to be available both in Mark II and Mark IV. But I will not do any testing with earlier `context`, so here I shall assume the `\expanded` behaves as expected and I can use `\notexpanded` for `\unexpanded`. This is abstracted into an alias `\xint_noxpd`.

I don't know if there is mode of running `context` where the `\errhelp` tokens will be shown (trying `c_\ontext --mkII` was not conclusive; by the way `\errmessage` in `e-TeX` adds a full stop which is missing with `context`).

About the `\let` primitive I am not sure it behaves fully as in other engines.

Note that this uses `\xintMessage` in case of old `ConTeXt` so we had to have it defined first.

```

319 \let\xint_noxpd\unexpanded
320 \ifdefined\contextversion
321   \let\xint_noxpd\notexpanded
322   \ifdefined\notexpanded
323     \else

```



```

324 \xintMessage{xintkernel}{Error}{This ConTeXt appears to be too old.}
325 \errhelp{xint only supports the ConTeXt-LMTX as of 2025/09/05.}%
326 \errmessage{The \noexpand\notexpanded primitive does not exist.}%
327 \fi
328 \fi

```

## 18.15. `\xintstrcmp`

Added at 1.4m (2022/06/10) [on 2022/06/05]. For the LuaTeX engine the code is copied over from [l3names.dtx](#). I also looked at Heiko Oberdiek's [pdfdoccmds.sty](#) and [pdfdoccmds.lua](#), but I removed `\luaescapestring` and used `token.scan_string()` as seen in [l3names.dtx](#) (and I did try to inform myself about this in the LuaTeX manual, with limited success). I am not sure about the syntax below with the `local`'s. Should I use `\directlua0`? Testing was minimal. Memo: even with [pdfTeX](#)'s `\pdfstrcmp`, braces around the arguments are mandatory.

```

329 \ifdefined\strcmp\let\xintstrcmp\strcmp
330 \else\ifdefined\pdfstrcmp\let\xintstrcmp\pdfstrcmp
331 \else\ifdefined\directlua\directlua{%
332 xintkernel = xintkernel or {}
333 local minus_tok = token.new(string.byte'-', 12)
334 local zero_tok = token.new(string.byte'0', 12)
335 local one_tok = token.new(string.byte'1', 12)
336 function xintkernel.strptime()
337   local A = token.scan_string()
338   local B = token.scan_string()
339   if A < B then
340     tex.write(minus_tok, one_tok)
341   else
342     tex.write(A == B and zero_tok or one_tok)
343   end
344 end
345 }\def\xintstrcmp{%
346   \directlua{xintkernel.strptime()}%
347 }%
348 \else
349 \xintMessage{xintkernel}{Error}{Could not set-up \string\xintstrcmp.}%
350 \errhelp{What kind of format are you using? Perhaps write the author? Bye now}%
351 \errmessage{Sorry, could not find or define string comparison primitive}\fi\fi\fi

```

## 18.16. `\xintresettimer`, `\xintelapsedtime`, `\xinttheseconds`

Added at 1.4n (2025/09/05). If `\resettimer` is defined, the code assumes `\elapsedtime` is, too.

I completely forgot at release time to document these engine-agnostic utilities in the manual. They were originally in the [xint.dtx](#) preamble, to allow building the documentation also with the Unicode engines. They are used only once there, for the example with [\xintUniformDeviate](#). Also in the preamble is the needed aliasing of `\pdfsetrandomseed` into `\setrandomseed` for the Unicode engines, but I completely forgot to transfer this too here and provide `\xintsetrandomseed`, which actually was more important because the package provides already utilities related with randomness.

Perhaps I should for LuaTeX copy the [l3kernel](#) code (here the code was picked from some [tex.sx](#) answer and was in my files for years) but I am not Lua-proficient enough to do this confidently... probably their `// 1` and `math.tointeger()` is more efficient than the `math.floor()` and there must be a reason why they use `gettimeofday()`, while `os.clock()` is only used to set the `epoch`.

I have been using for years the underlying syntax trick here for `\xinttheseconds`, and I usually never bother actually removing the catcode 12 `pt`, but well let's do it here.

I hesitate whether with Lua<sub>T</sub><sub>E</sub><sub>X</sub> I should avoid the scaled seconds intermediate in favor of the raw `os.clock()`-`basetime` data but I would have to waste time seeing what happens with `tex.write()` then. And it would not help really in comparing with PDF<sub>T</sub><sub>E</sub><sub>X</sub> or Xe<sub>T</sub><sub>E</sub><sub>X</sub>.

Modified at 1.4o (2025/09/06).

`xinttimer` replaces in the Lua code `xintelapsedtimer`, which was silly name used when I adopted in a haste at 1.4n while improving the user manual on `\xintUniformDeviate`.

```

352 \ifdefined\resettimer
353   \let\xintresettimer\resettimer   \let\xintelapsedtime\elapsedtime
354 \else
355 \ifdefined\pdfresettimer
356   \let\xintresettimer\pdfresettimer\let\xintelapsedtime\pdfelapsedtime
357 \else
358 \ifdefined\directlua
359   \directlua{xinttimer_basetime=0}%
360   \def\xintresettimer{\directlua{xinttimer_basetime = os.clock()}}%
361   \def\xintelapsedtime{\numexpr
362     \directlua{tex.print(math.floor((os.clock()-xinttimer_basetime)*65536+0.5))}%
363   \relax}%
364 \fi\fi\fi
365 \def\xintstript{\expandafter\XINT_stript\the}%
366 \expanded{\xint_noexpd{\def\XINT_stript#1}\detokenize{pt}}{#1}%
367 \def\xinttheseconds{\xintstript\dimexpr\xintelapsedtime sp\relax}%

```

## 18.17. `\xintReverseOrder`

Modified at 1.0 (2013/03/28). Does not expand its argument. The whole of xint codebase now contains only two calls to `\XINT_rord_main` (in `xintgcd`).

Attention: removes brace pairs (and swallows spaces).

For digit tokens a faster reverse macro is provided by (1.2) `\xintReverseDigits` in `xint`.

For comma separated items, 1.2g has `\xintCSVReverse` in `xinttools`.

```

368 \def\xintReverseOrder {\romannumeral0\xintreverseorder }%
369 \long\def\xintreverseorder #1%
370 {%
371   \XINT_rord_main { }#1%
372   \xint:
373     \xint_bye\xint_bye\xint_bye\xint_bye
374     \xint_bye\xint_bye\xint_bye\xint_bye
375   \xint:
376 }%
377 \long\def\XINT_rord_main #1#2#3#4#5#6#7#8#9%
378 {%
379   \xint_bye #9\XINT_rord_cleanup\xint_bye
380   \XINT_rord_main {#9#8#7#6#5#4#3#2#1}%
381 }%
382 \def\XINT_rord_cleanup #1{%
383 \long\def\XINT_rord_cleanup\xint_bye\XINT_rord_main ##1##2\xint:
384 {%
385   \expandafter#1\xint_gob_til_xint: ##1%
386 }}\XINT_rord_cleanup { }%

```

## 18.18. `\xintLength`

**Modified at 1.0 (2013/03/28).** Does not expand its argument. See `\xintNthElt{0}` from [xinttools](#) which f-expands its argument.

**Modified at 1.2g (2016/03/19).** Added [\xintCSVLength](#) to [xinttools](#).

**Modified at 1.2i (2016/12/13).** Rewrote this venerable macro. New code about 40% faster across all lengths. Syntax with `\romannumeral0` adds some slight (negligible) overhead; it is done to fit some general principles of structure of the xint package macros but maybe at some point I should drop it. And in fact it is often called directly via the `\numexpr` access point. (bad coding...)

Remark: the argument may contain `\par` tokens. But generally speaking most other macros of the [xint](#) bundle are not declared `\long`. As `\xintLength` produces a numeric quantity it is conceivable that it could serve in the input of some of the [xint](#) macros. For example something such as `\xinteval{\xintLength{\par\par}^3}` or `\xintiiMul{\xintLength{\par\par\par}}{17}`. They both fail. I have known this issue for many years. It is only needed to make long those macros which grab the argument and f-expand it. After expansion, of course no `\par` tokens is admissible as numerical input. Still this is quite some work due to size of the codebase. Waiting for a real-life bug report... (TeX3 people have fixed that on their side by making all declarations `\long` per default, and this may also perhaps increase slightly the efficiency, not checked).

```

387 \def\xintLength {\romannumeral0\xintlength }%
388 \def\xintlength #1{%
389 \long\def\xintlength ##1%
390 {%
391   \expandafter#1\the\numexpr\XINT_length_loop
392   ##1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
393   \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
394   \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
395   \relax
396 }}\xintlength{ }%
397 \long\def\XINT_length_loop #1#2#3#4#5#6#7#8#9%
398 {%
399   \xint_gob_til_xint: #9\XINT_length_finish_a\xint:
400   \xint_c_ix+\XINT_length_loop
401 }%
402 \def\XINT_length_finish_a\xint:\xint_c_ix+\XINT_length_loop
403   #1#2#3#4#5#6#7#8#9%
404 {%
405   #9\xint_bye
406 }%

```

## 18.19. `\xintLastItem`

**Modified at 1.2i (2016/12/13).** One level of braces removed in output. Output empty if input empty. Attention! This means that an empty input or an input ending with a empty brace pair both give same output.

The `\xint:` token must not be among items. `\xintFirstItem` added at 1.4 for usage in `xintexpr`. It must contain neither `\xint:` nor `\xint_bye` in its first item.

```

407 \def\xintLastItem {\romannumeral0\xintlastitem }%
408 \long\def\xintlastitem #1%
409 {%
410   \XINT_last_loop {}.#1%
411   {\xint:\XINT_last_loop_enda}{\xint:\XINT_last_loop_endb}%

```

```

412     {\xint:\XINT_last_loop_endc}{\xint:\XINT_last_loop_endd}%
413     {\xint:\XINT_last_loop_ende}{\xint:\XINT_last_loop_endf}%
414     {\xint:\XINT_last_loop_endg}{\xint:\XINT_last_loop_endh}\xint_bye
415 }%
416 \long\def\XINT_last_loop #1.#2#3#4#5#6#7#8#9%
417 {%
418     \xint_gob_til_xint: #9%
419     {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
420     \XINT_last_loop {#9}.%
421 }%
422 \long\def\XINT_last_loop_enda #1#2\xint_bye{ #1}%
423 \long\def\XINT_last_loop_endb #1#2#3\xint_bye{ #2}%
424 \long\def\XINT_last_loop_endc #1#2#3#4\xint_bye{ #3}%
425 \long\def\XINT_last_loop_endd #1#2#3#4#5\xint_bye{ #4}%
426 \long\def\XINT_last_loop_ende #1#2#3#4#5#6\xint_bye{ #5}%
427 \long\def\XINT_last_loop_endf #1#2#3#4#5#6#7\xint_bye{ #6}%
428 \long\def\XINT_last_loop_endg #1#2#3#4#5#6#7#8\xint_bye{ #7}%
429 \long\def\XINT_last_loop_endh #1#2#3#4#5#6#7#8#9\xint_bye{ #8}%

```

## 18.20. \xintFirstItem

1.4. There must be neither `\xint:` nor `\xint_bye` in its first item.

```

430 \def\xintFirstItem      {\romannumeral0\xintfirstitem }%
431 \long\def\xintfirstitem #1{\XINT_firstitem #1{\xint:\XINT_firstitem_end}\xint_bye}%
432 \long\def\XINT_firstitem #1#2\xint_bye{\xint_gob_til_xint: #1\xint:\space #1}%
433 \def\XINT_firstitem_end\xint:{ }%

```

## 18.21. \xintLastOne

As `xintexpr` 1.4 uses `{c1}{c2}...{cN}` storage when gathering comma separated values we need to not handle identically an empty list and a list with an empty item (as the above allows hierarchical structures). But `\xintLastItem` removed one level of brace pair so it is inadequate for the `last()` function.

By the way it is logical to interpret «item» as meaning `{cj}` inclusive of the braces; but legacy `xint` user manual was not written in this spirit. And thus `\xintLastItem` did brace stripping, thus we need another name for maintaining backwards compatibility (although the cardinality of users is small).

The `\xint:` token must not be found (visible) among the item contents.

```

434 \def\xintLastOne {\romannumeral0\xintlstone }%
435 \long\def\xintlstone #1%
436 {%
437     \XINT_lstone_loop {#1}%
438     {\xint:\XINT_lstone_loop_enda}{\xint:\XINT_lstone_loop_endb}%
439     {\xint:\XINT_lstone_loop_endc}{\xint:\XINT_lstone_loop_endd}%
440     {\xint:\XINT_lstone_loop_ende}{\xint:\XINT_lstone_loop_endf}%
441     {\xint:\XINT_lstone_loop_endg}{\xint:\XINT_lstone_loop_endh}\xint_bye
442 }%
443 \long\def\XINT_lstone_loop #1.#2#3#4#5#6#7#8#9%
444 {%
445     \xint_gob_til_xint: #9%
446     {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
447     \XINT_lstone_loop {#9}.%

```

```

448 }%
449 \long\def\XINT_lastone_loop_enda #1#2\xint_bye{{#1}}%
450 \long\def\XINT_lastone_loop_endb #1#2#3\xint_bye{{#2}}%
451 \long\def\XINT_lastone_loop_endc #1#2#3#4\xint_bye{{#3}}%
452 \long\def\XINT_lastone_loop_endd #1#2#3#4#5\xint_bye{{#4}}%
453 \long\def\XINT_lastone_loop_ende #1#2#3#4#5#6\xint_bye{{#5}}%
454 \long\def\XINT_lastone_loop_endf #1#2#3#4#5#6#7\xint_bye{{#6}}%
455 \long\def\XINT_lastone_loop_endg #1#2#3#4#5#6#7#8\xint_bye{{#7}}%
456 \long\def\XINT_lastone_loop_endh #1#2#3#4#5#6#7#8#9\xint_bye{ #8}%

```

## 18.22. \xintFirstOne

For xintexpr 1.4 too. Jan 3, 2020.

This is an experimental macro, don't use it. If input is nil (empty set) it expands to nil, if not it fetches first item and braces it. Fetching will have stripped one brace pair if item was braced to start with, which is the case in non-symbolic xintexpr data objects.

I have not given much thought to this (make it shorter, allow all tokens, (we could first test if empty via combination with `\detokenize`), etc...) as I need to get xint 1.4 out soon. So in particular attention that the macro assumes the `\xint:` token is absent from first item of input.

```

457 \def\xintFirstOne {\romannumeral0\xintfirstone }%
458 \long\def\xintfirstone #1{\XINT_firstone #1{\xint:\XINT_firstone_empty}\xint:}%
459 \long\def\XINT_firstone #1#2\xint:{\xint_gob_til_xint: #1\xint:{#1}}%
460 \def\XINT_firstone_empty\xint:#1{ }%

```

## 18.23. \xintLengthUpTo

**Modified at 1.2i (2016/12/13).** For use by `\xintKeep` and `\xintTrim` ([xinttools](#)). The argument N **\*\*must be non-negative\*\***.

`\xintLengthUpTo{N}{List}` produces `-0` if `length(List)>N`, else it returns `N-length(List)`. Hence subtracting it from N always computes `min(N,length(List))`.

**Modified at 1.2j (2016/12/22).** Changed ending and interface to core loop.

```

461 \def\xintLengthUpTo {\romannumeral0\xintlenthupto}%
462 \long\def\xintlenthupto #1#2%
463 {%
464   \expandafter\XINT_lenthupto_loop
465   \the\numexpr#1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
466     \xint_c_vii\xint_c_v\xint_c_v\xint_c_iv
467     \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
468 }%
469 \def\XINT_lenthupto_loop_a #1%
470 {%
471   \xint_UDsignfork
472   #1\XINT_lenthupto_gt
473   -\XINT_lenthupto_loop
474   \krof #1%
475 }%
476 \long\def\XINT_lenthupto_gt #1\xint_bye.{-0}%
477 \long\def\XINT_lenthupto_loop #1.#2#3#4#5#6#7#8#9%
478 {%
479   \xint_gob_til_xint: #9\XINT_lenthupto_finish_a\xint:%
480   \expandafter\XINT_lenthupto_loop_a\the\numexpr #1-\xint_c_viii.%
481 }%

```

## 18.24. \xintreplicate, \xintReplicate

```

494 \def\xintReplicate{\romannumeral\xintreplicate}%
495 \def\xintreplicate#1%
496   {\expandafter\XINT_replicate\the\numexpr#1\endcsname}%
497 \def\XINT_replicate #1{\xint_UDsignfork
498   #1\XINT_rep_neg
499   -\XINT_rep
500   \krof #1}%
501 \long\def\XINT_rep_neg #1\endcsname #2{\xint_c_}%
502 \def\XINT_rep #1{\csname XINT_rep_f#1\XINT_rep_a}%
503 \def\XINT_rep_a #1{\csname XINT_rep_#1\XINT_rep_a}%
504 \def\XINT_rep_\XINT_rep_a{\endcsname}%
505 \long\expandafter\def\csname XINT_rep_0\endcsname #1%
506   {\endcsname{#1#1#1#1#1#1#1#1#1#1}}%
507 \long\expandafter\def\csname XINT_rep_1\endcsname #1%
508   {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1}%
509 \long\expandafter\def\csname XINT_rep_2\endcsname #1%
510   {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1}%
511 \long\expandafter\def\csname XINT_rep_3\endcsname #1%
512   {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1}%
513 \long\expandafter\def\csname XINT_rep_4\endcsname #1%
514   {\endcsname{#1#1#1#1#1#1#1#1#1#1}#1#1#1#1}%
515 \long\expandafter\def\csname XINT_rep_5\endcsname #1%

```

### 18.25. \xintgobble, \xintGobble

Usage: `\romannumeral\xintgobble{N}...`

254



```

561 \long\expandafter\def\csname XINT_g14\endcsname#1#2#3#4{\endcsname}%
562 \long\expandafter\def\csname XINT_g15\endcsname#1#2#3#4#5{\endcsname}%
563 \long\expandafter\def\csname XINT_g16\endcsname#1#2#3#4#5#6{\endcsname}%
564 \long\expandafter\def\csname XINT_g17\endcsname#1#2#3#4#5#6#7{\endcsname}%
565 \long\expandafter\def\csname XINT_g18\endcsname#1#2#3#4#5#6#7#8{\endcsname}%
566 \expandafter\let\csname XINT_g20\endcsname\endcsname
567 \long\expandafter\def\csname XINT_g21\endcsname #1#2#3#4#5#6#7#8#9%
568 {\endcsname}%
569 \long\expandafter\edef\csname XINT_g22\endcsname #1#2#3#4#5#6#7#8#9%
570 {\expandafter\noexpand\csname XINT_g21\endcsname}%
571 \long\expandafter\edef\csname XINT_g23\endcsname #1#2#3#4#5#6#7#8#9%
572 {\expandafter\noexpand\csname XINT_g22\endcsname}%
573 \long\expandafter\edef\csname XINT_g24\endcsname #1#2#3#4#5#6#7#8#9%
574 {\expandafter\noexpand\csname XINT_g23\endcsname}%
575 \long\expandafter\edef\csname XINT_g25\endcsname #1#2#3#4#5#6#7#8#9%
576 {\expandafter\noexpand\csname XINT_g24\endcsname}%
577 \long\expandafter\edef\csname XINT_g26\endcsname #1#2#3#4#5#6#7#8#9%
578 {\expandafter\noexpand\csname XINT_g25\endcsname}%
579 \long\expandafter\edef\csname XINT_g27\endcsname #1#2#3#4#5#6#7#8#9%
580 {\expandafter\noexpand\csname XINT_g26\endcsname}%
581 \long\expandafter\edef\csname XINT_g28\endcsname #1#2#3#4#5#6#7#8#9%
582 {\expandafter\noexpand\csname XINT_g27\endcsname}%
583 \expandafter\let\csname XINT_g30\endcsname\endcsname
584 \long\expandafter\edef\csname XINT_g31\endcsname #1#2#3#4#5#6#7#8#9%
585 {\expandafter\noexpand\csname XINT_g28\endcsname}%
586 \long\expandafter\edef\csname XINT_g32\endcsname #1#2#3#4#5#6#7#8#9%
587 {\noexpand\csname XINT_g31\expandafter\noexpand\csname XINT_g28\endcsname}%
588 \long\expandafter\edef\csname XINT_g33\endcsname #1#2#3#4#5#6#7#8#9%
589 {\noexpand\csname XINT_g32\expandafter\noexpand\csname XINT_g28\endcsname}%
590 \long\expandafter\edef\csname XINT_g34\endcsname #1#2#3#4#5#6#7#8#9%
591 {\noexpand\csname XINT_g33\expandafter\noexpand\csname XINT_g28\endcsname}%
592 \long\expandafter\edef\csname XINT_g35\endcsname #1#2#3#4#5#6#7#8#9%
593 {\noexpand\csname XINT_g34\expandafter\noexpand\csname XINT_g28\endcsname}%
594 \long\expandafter\edef\csname XINT_g36\endcsname #1#2#3#4#5#6#7#8#9%
595 {\noexpand\csname XINT_g35\expandafter\noexpand\csname XINT_g28\endcsname}%
596 \long\expandafter\edef\csname XINT_g37\endcsname #1#2#3#4#5#6#7#8#9%
597 {\noexpand\csname XINT_g36\expandafter\noexpand\csname XINT_g28\endcsname}%
598 \long\expandafter\edef\csname XINT_g38\endcsname #1#2#3#4#5#6#7#8#9%
599 {\noexpand\csname XINT_g37\expandafter\noexpand\csname XINT_g28\endcsname}%
600 \expandafter\let\csname XINT_g40\endcsname\endcsname
601 \expandafter\edef\csname XINT_g41\endcsname
602 {\noexpand\csname XINT_g38\expandafter\noexpand\csname XINT_g31\endcsname}%
603 \expandafter\edef\csname XINT_g42\endcsname
604 {\noexpand\csname XINT_g41\expandafter\noexpand\csname XINT_g41\endcsname}%
605 \expandafter\edef\csname XINT_g43\endcsname
606 {\noexpand\csname XINT_g42\expandafter\noexpand\csname XINT_g41\endcsname}%
607 \expandafter\edef\csname XINT_g44\endcsname
608 {\noexpand\csname XINT_g43\expandafter\noexpand\csname XINT_g41\endcsname}%
609 \expandafter\edef\csname XINT_g45\endcsname
610 {\noexpand\csname XINT_g44\expandafter\noexpand\csname XINT_g41\endcsname}%
611 \expandafter\edef\csname XINT_g46\endcsname
612 {\noexpand\csname XINT_g45\expandafter\noexpand\csname XINT_g41\endcsname}%

```



```

613 \expandafter\edef\csname XINT_g47\endcsname
614 {\noexpand\csname XINT_g46\expandafter\noexpand\csname XINT_g41\endcsname}%
615 \expandafter\edef\csname XINT_g48\endcsname
616 {\noexpand\csname XINT_g47\expandafter\noexpand\csname XINT_g41\endcsname}%
617 \expandafter\let\csname XINT_g50\endcsname\endcsname
618 \expandafter\edef\csname XINT_g51\endcsname
619 {\noexpand\csname XINT_g48\expandafter\noexpand\csname XINT_g41\endcsname}%
620 \expandafter\edef\csname XINT_g52\endcsname
621 {\noexpand\csname XINT_g51\expandafter\noexpand\csname XINT_g51\endcsname}%
622 \expandafter\edef\csname XINT_g53\endcsname
623 {\noexpand\csname XINT_g52\expandafter\noexpand\csname XINT_g51\endcsname}%
624 \expandafter\edef\csname XINT_g54\endcsname
625 {\noexpand\csname XINT_g53\expandafter\noexpand\csname XINT_g51\endcsname}%
626 \expandafter\edef\csname XINT_g55\endcsname
627 {\noexpand\csname XINT_g54\expandafter\noexpand\csname XINT_g51\endcsname}%
628 \expandafter\edef\csname XINT_g56\endcsname
629 {\noexpand\csname XINT_g55\expandafter\noexpand\csname XINT_g51\endcsname}%
630 \expandafter\edef\csname XINT_g57\endcsname
631 {\noexpand\csname XINT_g56\expandafter\noexpand\csname XINT_g51\endcsname}%
632 \expandafter\edef\csname XINT_g58\endcsname
633 {\noexpand\csname XINT_g57\expandafter\noexpand\csname XINT_g51\endcsname}%
634 \expandafter\let\csname XINT_g60\endcsname\endcsname
635 \expandafter\edef\csname XINT_g61\endcsname
636 {\noexpand\csname XINT_g58\expandafter\noexpand\csname XINT_g51\endcsname}%
637 \expandafter\edef\csname XINT_g62\endcsname
638 {\noexpand\csname XINT_g61\expandafter\noexpand\csname XINT_g61\endcsname}%
639 \expandafter\edef\csname XINT_g63\endcsname
640 {\noexpand\csname XINT_g62\expandafter\noexpand\csname XINT_g61\endcsname}%
641 \expandafter\edef\csname XINT_g64\endcsname
642 {\noexpand\csname XINT_g63\expandafter\noexpand\csname XINT_g61\endcsname}%
643 \expandafter\edef\csname XINT_g65\endcsname
644 {\noexpand\csname XINT_g64\expandafter\noexpand\csname XINT_g61\endcsname}%
645 \expandafter\edef\csname XINT_g66\endcsname
646 {\noexpand\csname XINT_g65\expandafter\noexpand\csname XINT_g61\endcsname}%
647 \expandafter\edef\csname XINT_g67\endcsname
648 {\noexpand\csname XINT_g66\expandafter\noexpand\csname XINT_g61\endcsname}%
649 \expandafter\edef\csname XINT_g68\endcsname
650 {\noexpand\csname XINT_g67\expandafter\noexpand\csname XINT_g61\endcsname}%

```

## 18.26. Random number generation

Added at 1.3b (2018/05/18).

We provide a more random version of the (PDF) $\TeX$  `\pdfuniformdeviate`. I discusses the worries with the engine primitive with Bruno Le Floch in May 2018. Regarding `\pdfuniformdeviate x`:

1. with  $x=2^{29}$  or  $x=2^{30}$  the engine primitive produces only even numbers,
2. with  $x=3 \cdot 2^{26}$  the integers produced by the RNG when taken modulo three obey the proportion 1:1:2, not 1:1:1,
3. with  $x=3 \cdot 2^{14}$  there is analogous although weaker non-uniformity of the random integers when taken modulo 3,

- generally speaking pure powers of two should generate uniform random integers, but when the range is divisible by large powers of two, the non-uniformity may be amplified in surprising ways by modulo operations.

Moreover, two seeds sharing the same low  $k$  bits generate sequences of 28-bits integers which are one-to-one identical modulo  $2^k$ !

In order to mitigate the issues commented upon in the user manual, `\xintUniformDeviate` currently only uses the seven high bits from the underlying random stream, using multiple calls to `\pdfunif_ormdeviate 128`. From the Birthday Effect, after about  $2^{11}$  seeds one will likely pick a new one sharing its 22 low bits with an earlier one.

- but as the final random integer is obtained by additional operations involving the range  $x$  (currently a modulo operation), for odd ranges it is more difficult for bit correlations to be seen,
- anyway as they are only  $2^{28}$  seeds in total, after only  $2^{14}$  seeds it is likely to encounter one already explored, and then random integers are identical, however complicated the RNG's raw output is malaxed, and whatever the target range  $x$ . And  $2^{14}$  is only eight times as large as  $2^{11}$ .

It would be nice if the engine provided some user interface for letting its RNG execute a given number of iterations without the overhead of replicated executions of `\pdfuniformdeviate`. This could help gain entropy and would reduce correlations across series from distinct seeds.

**Modified at 1.4n (2025/09/05).** Print a warning immediately if no uniformdeviate is available.

#### 18.26.1. `\xint_texuniformdeviate`

```
651 \ifdefined\pdfuniformdeviate \let\xint_texuniformdeviate\pdfuniformdeviate\fi
652 \ifdefined\uniformdeviate \let\xint_texuniformdeviate\uniformdeviate \fi
653 \ifx\xint_texuniformdeviate\relax\let\xint_texuniformdeviate\xint_undefined\fi
```

#### 18.26.2. `\xint_texuniformdeviate_dgts`

**Added at 1.4n (2025/09/05).** Needed for compatibility with ConT<sub>E</sub>Xt.

One finds a macro `\randomnumber (/usr/local/texlive/2025/texmf-dist/tex/context/base/mkx1/supp-ran.mkx1)`, which takes two arguments. Fortunately it seems that throughout the `xint` code base, we use `\xint_texuniformdeviate` always with a chardef or count argument with one sole exception where it is followed with digits terminated by `\xint:`. In order not to change anything to current code and to support ConT<sub>E</sub>Xt-LMTX we make here a suitable definition compatible with these use cases.

Warning: I have no idea and will not check now if the RNG has the same issues for the less significant decimal digits as in PDF<sub>T</sub>eX, thus I don't know if the overhead in the definition below of `\xintUniformDeviate` has any rationale in the LMTX context.

MEMO: In the code below, I have not checked yet and do not remember (at time of preparing 1.4n) if it is possible that the `#1` fetched by `\xint_texuniformdeviate_dgts` has some chance to be zero (as I don't remember the details of the construction). Attention that `\randomnumber{0}{-1}` may output `-1`.

**Modified at 1.4o (2025/09/06).** Remove the test whether some uniformdeviate already exists with ConT<sub>E</sub>Xt and use unconditionally `\randomnumber` without checking if it exists.

```
654 \ifdefined\contextversion
655 \def\xint_texuniformdeviate#1{\randomnumber{0}{#1-1}}%
656 \def\xint_texuniformdeviate_dgts#1\xint:{\randomnumber{0}{#1-1}\xint:}%
657 \else
658 \let\xint_texuniformdeviate_dgts\xint_texuniformdeviate
659 \fi
```

### 18.26.3. \xintUniformDeviate

For negative `#1` the output (as per the doc I wrote in the user manual some years ago and that I trust) ranges from `#1+1` to zero inclusive. This is same behaviour as original primitive and the input must be within the  $\text{T}_{\text{E}}\text{X}$  bounds. Note that `\the\numexpr-0\relax` gives 0.

We make 5 calls to the primitive and the user doc says that when I tested the cost was about 13 times the one of the primitive.

```

660 \ifdefined\xint_texuniformdeviate
661     \expandafter\xint_firstoftwo
662 \else\expandafter\xint_secondoftwo
663 \fi
664 {%
665     \def\xintUniformDeviate#1%
666         {\the\numexpr\expandafter\XINT_uniformdeviate_sgnfork\the\numexpr#1\xint:}%
667     \def\XINT_uniformdeviate_sgnfork#1%
668         {%
669             \if-#1\XINT_uniformdeviate_neg\fi \XINT_uniformdeviate{ }#1%
670         }%
671     \def\XINT_uniformdeviate_neg\fi\XINT_uniformdeviate#1-%
672         {%
673             \fi-\numexpr\XINT_uniformdeviate\relax
674         }%
675     \def\XINT_uniformdeviate#1#2\xint:
676         {%
677             \expandafter\XINT_uniformdeviate_a\the\numexpr%
678                 -\xint_texuniformdeviate\xint_c_ii^vii%
679                 -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
680                 -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
681                 -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
682                 +\xint_texuniformdeviate_dgts#2\xint:/#2)*#2\xint:+#2\fi\relax#1%
683         }%
684     \def\XINT_uniformdeviate_a #1\xint:
685         {%
686             \expandafter\XINT_uniformdeviate_b\the\numexpr#1-(#1%
687         }%
688     \def\XINT_uniformdeviate_b#1#2\xint:{#1#2\if-#1}%
689 }%
690 {%
691     \xintMessage{xintkernel}{Warning}%
692     {No \string\uniformdeviate like primitive identified, macros producing^^}%
693     \XINT_fourspace "random numbers" will raise (expandable) errors.}%
694     \def\xintUniformDeviate#1%
695         {%
696             \the\numexpr
697             \XINT_expandableerror{(xintkernel) No uniformdeviate primitive!}%
698             0\relax
699         }%
700 }%
701 \XINTrestorecatcodesendinginput%
```

## 19. Package [xinttools](#) implementation

.1	Catcodes, $\varepsilon$ -T <sub>E</sub> X and reload detection . . .	259	
.2	Package identification . . . . .	260	
.3	<code>\xintgodef</code> , <code>\xintgoodef</code> , <code>\xintgfdef</code> . . .	260	
.4	<code>\xintRevWithBraces</code> . . . . .	260	
.5	<code>\xintZapFirstSpaces</code> . . . . .	261	
.6	<code>\xintZapLastSpaces</code> . . . . .	262	
.7	<code>\xintZapSpaces</code> . . . . .	262	
.8	<code>\xintZapSpacesB</code> . . . . .	263	
.9	<code>\xintCSVtoList</code> , <code>\xintCSVtoListNon-</code> <code>Stripped</code> . . . . .	263	
.10	<code>\xintListWithSep</code> . . . . .	265	
.11	<code>\xintNthElt</code> . . . . .	266	
.12	<code>\xintNthOnePy</code> . . . . .	267	
.13	<code>\xintKeep</code> . . . . .	268	
.14	<code>\xintKeepUnbraced</code> . . . . .	269	
.15	<code>\xintTrim</code> . . . . .	270	
.16	<code>\xintTrimUnbraced</code> . . . . .	272	
.17	<code>\xintApply</code> . . . . .	273	
.18	<code>\xintApply:x</code> (WIP, commented-out) . . .	273	
.19	<code>\xintApplyUnbraced</code> . . . . .	274	
.20	<code>\xintApplyUnbraced:x</code> (WIP, commented- out) . . . . .	275	
.21	<code>\xintZip</code> (WIP, not public) . . . . .	276	
.22	<code>\xintSeq</code> . . . . .	278	
.23	<code>\xintloop</code> , <code>\xintbreakloop</code> , <code>\xintbreak-</code> <code>loopanddo</code> , <code>\xintloopskiptonext</code> . . .	281	
.24	<code>\xintiloop</code> , <code>\xintiloopindex</code> , <code>\xint-</code> <code>bracediloopindex</code> , <code>\xintouteriloopindex</code> ,		<code>\xintbracedouteriloopindex</code> , <code>\xintbreak-</code> <code>iloop</code> , <code>\xintbreakiloopanddo</code> , <code>\xintiloop-</code> <code>skiptonext</code> , <code>\xintiloopskipandredo</code> . . .
.25	<code>\XINT_xflet</code> . . . . .	282	
.26	<code>\xintApplyInline</code> . . . . .	282	
.27	<code>\xintFor</code> , <code>\xintFor*</code> , <code>\xintBreakFor</code> , <code>\xintBreakForAndDo</code> . . . . .	283	
.28	<code>\XINT_forever</code> , <code>\xintintegers</code> , <code>\xintdi-</code> <code>mensions</code> , <code>\xintrationals</code> . . . . .	285	
.29	<code>\xintForpair</code> , <code>\xintForthree</code> , <code>\xintFor-</code> <code>four</code> . . . . .	287	
.30	<code>\xintAssign</code> , <code>\xintAssignArray</code> , <code>\xint-</code> <code>DigitsOf</code> . . . . .	289	
.31	CSV (non user documented) variants of Length, Keep, Trim, NthElt, Reverse . . .	292	
.31.1	<code>\xintLength:f:csv</code> . . . . .	292	
.31.2	<code>\xintLengthUpTo:f:csv</code> . . . . .	293	
.31.3	<code>\xintKeep:f:csv</code> . . . . .	294	
.31.4	<code>\xintTrim:f:csv</code> . . . . .	296	
.31.5	<code>\xintNthEltPy:f:csv</code> . . . . .	298	
.31.6	<code>\xintReverse:f:csv</code> . . . . .	299	
.31.7	<code>\xintFirstItem:f:csv</code> . . . . .	299	
.31.8	<code>\xintLastItem:f:csv</code> . . . . .	299	
.31.9	<code>\xintKeep:x:csv</code> . . . . .	300	
.31.10	Public names for the undocumented csv macros: <code>\xintCSVLength</code> , <code>\xintCSVKeep</code> , <code>\xintCSVKeepx</code> , <code>\xintCSVTrim</code> , <code>\xintCSVNthEltPy</code> , <code>\xintCSVRe-</code> <code>verse</code> , <code>\xintCSVFirstItem</code> , <code>\xintCSVLastItem</code>	301	

Added at 1.09g (2013/11/22). Splits off [xinttools](#) from [xint](#).

Modified at 1.1 (2014/10/28). [xinttools](#) ceases being loaded automatically by [xint](#).

### 19.1. Catcodes, $\varepsilon$ -T<sub>E</sub>X and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \beginingroup\catcode61\catcode48\catcode32=10\relax%
2   \catcode13=5      % ^^M
3   \endlinechar=13 %
4   \catcode123=1     % {
5   \catcode125=2     % }
6   \catcode64=11     % @
7   \catcode44=12     % ,
8   \catcode46=12     % .
9   \catcode58=12     % :
10  \catcode94=7      % ^
11  \def\empty{}\def\space{ }\newlinechar10
12  \def\z{\endgroup}%
13  \expandafter\let\expandafter\x\csname ver@xinttools.sty\endcsname
14  \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname

```

```

15 \expandafter\ifx\csname numexpr\endcsname\relax
16 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17 \immediate\write128{^^JPackage xinttools Warning:^^J%
18 \space\space\space\space
19 \numexpr not available, aborting input.^^J}%
20 \else
21 \PackageWarningNoLine{xinttools}{\numexpr not available, aborting input}%
22 \fi
23 \def\z{\endgroup\endinput}%
24 \else
25 \ifx\x\relax % plain-TeX, first loading of xinttools.sty
26 \ifx\w\relax % but xintkernel.sty not yet loaded.
27 \def\z{\endgroup\input xintkernel.sty\relax}%
28 \fi
29 \else
30 \ifx\x\empty % LaTeX, first loading,
31 % variable is initialized, but \ProvidesPackage not yet seen
32 \ifx\w\relax % xintkernel.sty not yet loaded.
33 \def\z{\endgroup\RequirePackage{xintkernel}}%
34 \fi
35 \else
36 \def\z{\endgroup\endinput}% xinttools already loaded.
37 \fi
38 \fi
39 \fi
40 \z%
41 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 19.2. Package identification

```

42 \XINT_providespackage
43 \ProvidesPackage{xinttools}%
44 [2025/09/06 v1.4o Expandable and non-expandable utilities (JFB)]%

```

`\XINT_toks` is used in macros such as `\xintFor`. It is not used elsewhere in the xint bundle.

```

45 \newtoks\XINT_toks
46 \xint_firstofone{\let\XINT_sptoken= } %<- space here!

```

## 19.3. `\xintgodef`, `\xintgoodef`, `\xintgfdef`

Added at 1.09i (2013/12/18). For use in `\xintAssign`.

```

47 \def\xintgodef {\global\xintodef}%
48 \def\xintgoodef {\global\xintoodef}%
49 \def\xintgfdef {\global\xintfdef}%

```

## 19.4. `\xintRevWithBraces`

Added at 1.06 (2013/05/07). Makes the expansion of its argument and then reverses the resulting tokens or braced tokens, adding a pair of braces to each (thus, maintaining it when it was already there.) The reason for `\xint:`, here and in other locations, is in case `#1` expands to nothing, the `\romannumeral-`0` must be stopped.

```

50 \def\xintRevWithBraces {\romannumeral0\xintrevwithbraces}%

```

```

51 \def\xintRevWithBracesNoExpand {\romannumeral0\xintrevwithbracesnoexpand}%
52 \long\def\xintrevwithbraces #1%
53 {%
54   \expandafter\XINT_revwbr_loop\expandafter{\expandafter}%
55   \romannumeral`&&@#1\xint:\xint:\xint:\xint:%
56   \xint:\xint:\xint:\xint:\xint_bye
57}%
58 \long\def\xintrevwithbracesnoexpand #1%
59 {%
60   \XINT_revwbr_loop {}%
61   #1\xint:\xint:\xint:\xint:%
62   \xint:\xint:\xint:\xint:\xint_bye
63}%
64 \long\def\XINT_revwbr_loop #1#2#3#4#5#6#7#8#9%
65 {%
66   \xint_gob_til_xint: #9\XINT_revwbr_finish_a\xint:%
67   \XINT_revwbr_loop {{#9}{#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}%
68}%
69 \long\def\XINT_revwbr_finish_a\xint:\XINT_revwbr_loop #1#2\xint_bye
70 {%
71   \XINT_revwbr_finish_b #2\R\R\R\R\R\R\R\Z #1%
72}%
73 \def\XINT_revwbr_finish_b #1#2#3#4#5#6#7#8\Z
74 {%
75   \xint_gob_til_R
76   #1\XINT_revwbr_finish_c \xint_gobble_viii
77   #2\XINT_revwbr_finish_c \xint_gobble_vii
78   #3\XINT_revwbr_finish_c \xint_gobble_vi
79   #4\XINT_revwbr_finish_c \xint_gobble_v
80   #5\XINT_revwbr_finish_c \xint_gobble_iv
81   #6\XINT_revwbr_finish_c \xint_gobble_iii
82   #7\XINT_revwbr_finish_c \xint_gobble_ii
83   \R\XINT_revwbr_finish_c \xint_gobble_i\Z
84}%

```

1.1c revisited this old code and improved upon the earlier endings.

```

85 \def\XINT_revwbr_finish_c#1{%
86 \def\XINT_revwbr_finish_c##1##2\Z{\expandafter#1##1}%
87 }\XINT_revwbr_finish_c{ }%

```

## 19.5. \xintZapFirstSpaces

Added at 1.09f (2013/11/04) [on 2013/11/01].

Modified at 1.1 (2014/10/28). To correct the bug in case of an empty argument, or argument containing only spaces, which had been forgotten in first version. New version is simpler than the initial one. This macro does NOT expand its argument.

```

88 \def\xintZapFirstSpaces {\romannumeral0\xintzapfirstspaces}%
89 \def\xintzapfirstspaces#1{\long
90 \def\xintzapfirstspaces ##1{\XINT_zapbsp_a #1##1\xint:#1#1\xint:}%
91 }\xintzapfirstspaces{ }%

```

If the original #1 started with a space, the grabbed #1 is empty. Thus `_again?` will see `#1=\xint_bye`, and hand over control to `_again` which will loop back into `\XINT_zapbsp_a`, with one

initial space less. If the original #1 did not start with a space, or was empty, then the #1 below will be a <sptoken>, then an extract of the original #1, not empty and not starting with a space, which contains what was up to the first <sp><sp> present in original #1, or, if none pre-existed, <sptoken> and all of #1 (possibly empty) plus an ending `\xint:`. The added initial space will stop later the `\romannumeral0`. No brace stripping is possible. Control is handed over to `\XINT_zapbsp_b` which strips out the ending `\xint:<sp><sp>\xint:`

```

92 \def\xINT_zapbsp_a#1{\long\def\xINT_zapbsp_a ##1#1#1{%
93   \XINT_zapbsp_again?##1\xint_bye\xINT_zapbsp_b ##1#1#1}%
94 }\XINT_zapbsp_a{ }%
95 \long\def\xINT_zapbsp_again? #1{\xint_bye #1\xINT_zapbsp_again }%
96 \xint_firstofone{\def\xINT_zapbsp_again\xINT_zapbsp_b}{\XINT_zapbsp_a }%
97 \long\def\xINT_zapbsp_b #1\xint:#2\xint:{#1}%

```

## 19.6. `\xintZapLastSpaces`

Added at 1.09f (2013/11/04) [on 2013/11/01].

```

98 \def\xintZapLastSpaces {\romannumeral0\xintzaplastspaces }%
99 \def\xintzaplastspaces#1{\long
100 \def\xintzaplastspaces ##1{\XINT_zapbsp_a {} \empty##1#1#1\xint_bye\xint:}%
101 }\xintzaplastspaces{ }%

```

The `\empty` from `\xintzaplastspaces` is to prevent brace removal in the #2 below. The `\expandafter` chain removes it.

```

102 \xint_firstofone {\long\def\xINT_zapbsp_a #1#2 } %<- second space here
103   {\expandafter\xINT_zapbsp_b\expandafter{#2}{#1}}%

```

Notice again an `\empty` added here. This is in preparation for possibly looping back to `\XINT_zapbsp_a`. If the initial #1 had no <sp><sp>, the stuff however will not loop, because #3 will already be <some spaces>`\xint_bye`. Notice that this macro fetches all way to the ending `\xint:`. This looks not very efficient, but how often do we have to strip ending spaces from something which also has inner stretches of `_multiple_` space tokens ?;-).

```

104 \long\def\xINT_zapbsp_b #1#2#3\xint:%
105   {\XINT_zapbsp_end? #3\xINT_zapbsp_e {#2#1} \empty #3\xint:}%

```

When we have been over all possible <sp><sp> things, we reach the ending space tokens, and #3 will be a bunch of spaces (possibly none) followed by `\xint_bye`. So the #1 in `_end?` will be `\xint_bye`. In all other cases #1 can not be `\xint_bye` (assuming naturally this token does not arise in original input), hence control falls back to `\XINT_zapbsp_e` which will loop back to `\XINT_zapbsp_a`.

```

106 \long\def\xINT_zapbsp_end? #1{\xint_bye #1\xINT_zapbsp_end }%

```

We are done. The #1 here has accumulated all the previous material, and is stripped of its ending spaces, if any.

```

107 \long\def\xINT_zapbsp_end\xINT_zapbsp_e #1#2\xint:{ #1}%

```

We haven't yet reached the end, so we need to re-inject two space tokens after what we have gotten so far. Then we loop.

```

108 \def\xINT_zapbsp_e#1{%
109 \long\def\xINT_zapbsp_e ##1{\XINT_zapbsp_a {##1#1#1}}%
110 }\XINT_zapbsp_e{ }%

```

## 19.7. `\xintZapSpaces`

Added at 1.09f (2013/11/04) [on 2013/11/01].



**Modified at 1.1 (2014/10/28).** It had the same bug as `\xintZapFirstSpaces`. We in effect do first `\xintZapFirstSpaces`, then `\xintZapLastSpaces`.

```

111 \def\xintZapSpaces {\romannumeral0\xintzapspace }%
112 \def\xintzapspace#1{%
113 \long\def\xintzapspace ##1% like \xintZapFirstSpaces.
114     {\XINT_zapsp_a #1##1\xint:#1#1\xint:}%
115 }\xintzapspace{ }%
116 \def\XINT_zapsp_a#1{%
117 \long\def\XINT_zapsp_a ##1#1#1%
118     {\XINT_zapsp_again?##1\xint_bye\XINT_zapsp_b##1#1#1}%
119 }\XINT_zapsp_a{ }%
120 \long\def\XINT_zapsp_again? #1{\xint_bye #1\XINT_zapsp_again }%
121 \xint_firstofone{\def\XINT_zapsp_again\XINT_zapsp_b} {\XINT_zapsp_a }%
122 \xint_firstofone{\def\XINT_zapsp_b} {\XINT_zapsp_c }%
123 \def\XINT_zapsp_c#1{%
124 \long\def\XINT_zapsp_c ##1\xint:##2\xint:%
125     {\XINT_zapsp_a{ }\empty ##1#1#1\xint_bye\xint:}%
126 }\XINT_zapsp_c{ }%

```

## 19.8. `\xintZapSpacesB`

**Added at 1.09f (2013/11/04) [on 2013/11/01].** Strips up to one pair of braces (but then does not strip spaces inside).

```

127 \def\xintZapSpacesB {\romannumeral0\xintzapspaceb }%
128 \long\def\xintzapspaceb #1{\XINT_zapspb_one? #1\xint:\xint:%
129     \xint_bye\xintzapspace {#1}}%
130 \long\def\XINT_zapspb_one? #1#2%
131     {\xint_gob_til_xint: #1\XINT_zapspb_onlyspace\xint:%
132     \xint_gob_til_xint: #2\XINT_zapspb_bracedorone\xint:%
133     \xint_bye {#1}}%
134 \def\XINT_zapspb_onlyspace\xint:%
135     \xint_gob_til_xint:\xint:XINT_zapspb_bracedorone\xint:%
136     \xint_bye #1\xint_bye\xintzapspace #2{ }%
137 \long\def\XINT_zapspb_bracedorone\xint:%
138     \xint_bye #1\xint:\xint_bye\xintzapspace #2{ #1}%

```

## 19.9. `\xintCSVtoList`, `\xintCSVtoListNonStripped`

**Added at 1.06 (2013/05/07).** `\xintCSVtoList` transforms `a,b,...,z` into `{a}{b}...{z}`. The comma separated list may be a macro which is first f-expanded. Here, use of `\Z` (and `\R`) perfectly safe.

**Modified at 1.09f (2013/11/04).** Automatically filters items with `\xintZapSpacesB` to strip away all spaces around commas, and spaces at the start and end of the list. The original is kept as `\xintCSVtoListNonStripped`, and is faster. But ... it doesn't strip spaces.

ATTENTION: if the input is empty the output contains one item (empty, of course). This means an `\xintFor` loop always executes at least once the iteration, contrarily to `\xintFor*`.

```

139 \def\xintCSVtoList {\romannumeral0\xintcsvtolist }%
140 \long\def\xintcsvtolist #1{\expandafter\xintApply
141     \expandafter\xintzapspaceb
142     \expandafter{\romannumeral0\xintcsvtolistnonstripped{#1}}}%
143 \def\xintCSVtoListNoExpand {\romannumeral0\xintcsvtolistnoexpand }%

```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

144 \long\def\xintcsvtolistnoexpand #1{\expandafter\xintApply
145     \expandafter\xintzapspacesb
146     \expandafter{\romannumeral0\xintcsvtolistnonstrippednoexpand{#1}}}%
147 \def\xintCSVtoListNonStripped {\romannumeral0\xintcsvtolistnonstripped}%
148 \def\xintCSVtoListNonStrippedNoExpand
149     {\romannumeral0\xintcsvtolistnonstrippednoexpand}%
150 \long\def\xintcsvtolistnonstripped #1%
151 {%
152     \expandafter\XINT_csvtol_loop_a\expandafter
153     {\expandafter}\romannumeral`&&@#1%
154     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye
155     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
156 }%
157 \long\def\xintcsvtolistnonstrippednoexpand #1%
158 {%
159     \XINT_csvtol_loop_a
160     {#1,\xint_bye,\xint_bye,\xint_bye,\xint_bye
161     ,\xint_bye,\xint_bye,\xint_bye,\xint_bye,\Z
162 }%
163 \long\def\XINT_csvtol_loop_a #1#2,#3,#4,#5,#6,#7,#8,#9,%
164 {%
165     \xint_bye #9\XINT_csvtol_finish_a\xint_bye
166     \XINT_csvtol_loop_b {#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
167 }%
168 \long\def\XINT_csvtol_loop_b #1#2{\XINT_csvtol_loop_a {#1#2}}%
169 \long\def\XINT_csvtol_finish_a\xint_bye\XINT_csvtol_loop_b #1#2#3\Z
170 {%
171     \XINT_csvtol_finish_b #3\R,\R,\R,\R,\R,\R,\R,\Z #2{#1}%
172 }%

```

1.1c revisits this old code and improves upon the earlier endings. But as the `_d..` macros have already nine parameters, I needed the `\expandafter` and `\xint_gob_til_Z` in `finish_b` (compare `\XINT_`, `T_keep_endb`, or also `\XINT_RQ_end_b`).

```

173 \def\XINT_csvtol_finish_b #1,#2,#3,#4,#5,#6,#7,#8\Z
174 {%
175     \xint_gob_til_R
176     #1\expandafter\XINT_csvtol_finish_dviii\xint_gob_til_Z
177     #2\expandafter\XINT_csvtol_finish_dvii \xint_gob_til_Z
178     #3\expandafter\XINT_csvtol_finish_dvi \xint_gob_til_Z
179     #4\expandafter\XINT_csvtol_finish_dv \xint_gob_til_Z
180     #5\expandafter\XINT_csvtol_finish_div \xint_gob_til_Z
181     #6\expandafter\XINT_csvtol_finish_diii \xint_gob_til_Z
182     #7\expandafter\XINT_csvtol_finish_dii \xint_gob_til_Z
183     \R\XINT_csvtol_finish_di \Z
184 }%
185 \long\def\XINT_csvtol_finish_dviii #1#2#3#4#5#6#7#8#9{ #9}%
186 \long\def\XINT_csvtol_finish_dvii #1#2#3#4#5#6#7#8#9{ #9{#1}}%
187 \long\def\XINT_csvtol_finish_dvi #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}}%
188 \long\def\XINT_csvtol_finish_dv #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}}%
189 \long\def\XINT_csvtol_finish_div #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}}%
190 \long\def\XINT_csvtol_finish_diii #1#2#3#4#5#6#7#8#9{ #9{#1}{#2}{#3}{#4}{#5}}%
191 \long\def\XINT_csvtol_finish_dii #1#2#3#4#5#6#7#8#9{
192     { #9{#1}{#2}{#3}{#4}{#5}{#6}}%

```

```

193 \long\def\XINT_csvtol_finish_di\Z #1#2#3#4#5#6#7#8#9%
194 { #9{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%

```

## 19.10. \xintListWithSep

**Added at 1.04 (2013/04/25).** `\xintListWithSep {\sep}{{a}{b}...{z}}` returns a `\sep b \sep ... \sep z`. It f-expands its second argument. The 'sep' may be `\par`'s: the macro `\xintlistwithsep` etc... are all declared long. 'sep' does not have to be a single token. It is not expanded. The "list" argument may be empty.

`\xintListWithSepNoExpand` does not f-expand its second argument.

**Modified at 1.2p (2017/12/05).** This venerable macro from 1.04 remained unchanged for a long time and was finally refactored at 1.2p for increased speed. Tests done with a list of identical `{\x}` items and a sep of `\z` demonstrated a speed increase of about:

- 3x for 30 items,
- 4.5x for 100 items,
- 7.5x--8x for 1000 items.

```

195 \def\xintListWithSep      {\romannumeral0\xintlistwithsep}%
196 \def\xintListWithSepNoExpand {\romannumeral0\xintlistwithsepnoexpand}%
197 \long\def\xintlistwithsep #1#2%
198   {\expandafter\XINT_lws\expandafter {\romannumeral`&&@#2}{#1}}%
199 \long\def\xintlistwithsepnoexpand #1#2%
200 {%
201   \XINT_lws_loop_a {#1}#2{\xint_bye\XINT_lws_e_vi}%
202   {\xint_bye\XINT_lws_e_v}{\xint_bye\XINT_lws_e_iv}%
203   {\xint_bye\XINT_lws_e_iii}{\xint_bye\XINT_lws_e_ii}%
204   {\xint_bye\XINT_lws_e_i}{\xint_bye\XINT_lws_e}%
205   {\xint_bye\expandafter\space}\xint_bye
206 }%
207 \long\def\XINT_lws #1#2%
208 {%
209   \XINT_lws_loop_a {#2}#1{\xint_bye\XINT_lws_e_vi}%
210   {\xint_bye\XINT_lws_e_v}{\xint_bye\XINT_lws_e_iv}%
211   {\xint_bye\XINT_lws_e_iii}{\xint_bye\XINT_lws_e_ii}%
212   {\xint_bye\XINT_lws_e_i}{\xint_bye\XINT_lws_e}%
213   {\xint_bye\expandafter\space}\xint_bye
214 }%
215 \long\def\XINT_lws_loop_a #1#2#3#4#5#6#7#8#9%
216 {%
217   \xint_bye #9\xint_bye
218   \XINT_lws_loop_b {#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}%
219 }%
220 \long\def\XINT_lws_loop_b #1#2#3#4#5#6#7#8#9%
221 {%
222   \XINT_lws_loop_a {#1}{#2#1#3#1#4#1#5#1#6#1#7#1#8#1#9}%
223 }%
224 \long\def\XINT_lws_e_vi\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7#8#9\xint_bye
225   { #2#1#3#1#4#1#5#1#6#1#7#1#8}%
226 \long\def\XINT_lws_e_v\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7#8\xint_bye
227   { #2#1#3#1#4#1#5#1#6#1#7}%
228 \long\def\XINT_lws_e_iv\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6#7\xint_bye
229   { #2#1#3#1#4#1#5#1#6}%
230 \long\def\XINT_lws_e_iii\xint_bye\XINT_lws_loop_b #1#2#3#4#5#6\xint_bye

```

```

231 { #2#1#3#1#4#1#5}%
232 \long\def\XINT_lws_e_ii\xint_bye\XINT_lws_loop_b #1#2#3#4#5\xint_bye
233 { #2#1#3#1#4}%
234 \long\def\XINT_lws_e_i\xint_bye\XINT_lws_loop_b #1#2#3#4\xint_bye
235 { #2#1#3}%
236 \long\def\XINT_lws_e\xint_bye\XINT_lws_loop_b #1#2#3\xint_bye
237 { #2}%

```

### 19.11. \xintNthElt

Added at 1.06 (2013/05/07).

Modified at 1.2j (2016/12/22). Last refactored in 1.2j.

`\xintNthElt {i}{List}` returns the  $i$ th item from List (one pair of braces removed). The list is first f-expanded. The `\xintNthEltNoExpand` does no expansion of its second argument. Both variants expand  $i$  inside `\numexpr`.

With  $i = 0$ , the number of items is returned using `\xintLength` but with the List argument f-expanded first.

Negative values return the  $|i|$ th element from the end.

When  $i$  is out of range, an empty value is returned.

```

238 \def\xintNthElt { \romannumeral0\xintnthelt }%
239 \def\xintNthEltNoExpand { \romannumeral0\xintntheltnoexpand }%
240 \long\def\xintnthelt #1#2{\expandafter\XINT_nthelt_a\the\numexpr #1\expandafter.%
241 \expandafter{\romannumeral`&&@#2}}%
242 \def\xintntheltnoexpand #1{\expandafter\XINT_nthelt_a\the\numexpr #1.}%
243 \def\XINT_nthelt_a #1%
244 {%
245 \xint_UDzerominusfork
246 #1-\XINT_nthelt_zero
247 0#1\XINT_nthelt_neg
248 0-{\XINT_nthelt_pos #1}%
249 \krof
250 }%
251 \def\XINT_nthelt_zero #1.{\xintlength }%
252 \long\def\XINT_nthelt_neg #1.#2%
253 {%
254 \expandafter\XINT_nthelt_neg_a\the\numexpr\xint_c_i+\XINT_length_loop
255 #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
256 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
257 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
258 -#1.#2\xint_bye
259 }%
260 \def\XINT_nthelt_neg_a #1%
261 {%
262 \xint_UDzerominusfork
263 #1-\xint_stop_afterbye
264 0#1\xint_stop_afterbye
265 0-{}%
266 \krof
267 \expandafter\XINT_nthelt_neg_b
268 \romannumeral\expandafter\XINT_gobble\the\numexpr-\xint_c_i+#1%
269 }%
270 \long\def\XINT_nthelt_neg_b #1#2\xint_bye{ #1}%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
271 \long\def\XINT_nthelt_pos #1.#2%
272 {%
273   \expandafter\XINT_nthelt_pos_done
274   \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_x.%
275   #2\xint:\xint:\xint:\xint:\xint:%
276   \xint:\xint:\xint:\xint:\xint:%
277   \xint_bye
278 }%
279 \def\XINT_nthelt_pos_done #1{%
280 \long\def\XINT_nthelt_pos_done ##1##2\xint_bye{%
281   \xint_gob_til_xint:##1\expandafter#1\xint_gobble_ii\xint:#1##1}%
282 }\XINT_nthelt_pos_done{ }%
```

### 19.12. \xintNthOnePy

Added at 1.4 (2020/01/31). See relevant code comments in [xintexpr](#).

```
283 \def\xintNthOnePy          {\romannumeral0\xintnthonepy }%
284 \def\xintNthOnePyNoExpand {\romannumeral0\xintnthonepynoexpand }%
285 \long\def\xintnthonepy #1#2{\expandafter\XINT_nthonepy_a\the\numexpr #1\expandafter.%
286   \expandafter{\romannumeral`&&@#2}}%
287 \def\xintnthonepynoexpand #1{\expandafter\XINT_nthonepy_a\the\numexpr #1.}%
288 \def\XINT_nthonepy_a #1%
289 {%
290   \xint_UDsignfork
291   #1\XINT_nthonepy_neg
292   -{\XINT_nthonepy_nonneg #1}%
293   \krof
294 }%
295 \long\def\XINT_nthonepy_neg #1.#2%
296 {%
297   \expandafter\XINT_nthonepy_neg_a\the\numexpr\xint_c_i+\XINT_length_loop
298   #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
299   \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
300   \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
301   -#1.#2\xint_bye
302 }%
303 \def\XINT_nthonepy_neg_a #1%
304 {%
305   \xint_UDzerominusfork
306   #1-\xint_stop_afterbye
307   0#1\xint_stop_afterbye
308   0-{}%
309   \krof
310   \expandafter\XINT_nthonepy_neg_b
311   \romannumeral\expandafter\XINT_gobble\the\numexpr-\xint_c_i+#1%
312 }%
313 \long\def\XINT_nthonepy_neg_b #1#2\xint_bye{{#1}}%
314 \long\def\XINT_nthonepy_nonneg #1.#2%
315 {%
316   \expandafter\XINT_nthonepy_nonneg_done
317   \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_ix.%
318   #2\xint:\xint:\xint:\xint:\xint:%
```

```

319 \xint:\xint:\xint:\xint:\xint:%
320 \xint_bye
321 }%
322 \def\xINT_nthonepy_nonneg_done #1{%
323 \long\def\xINT_nthonepy_nonneg_done ##1##2\xint_bye{%
324 \xint_gob_til_xint:##1\expandafter#1\xint_gobble_ii\xint:{##1}}%
325 }\xINT_nthonepy_nonneg_done{ }%

```

### 19.13. \xintKeep

**Added at 1.09m (2014/02/26).** `\xintKeep{i}{L}` f-expands its second argument L. It then grabs the first i items from L and discards the rest.

ATTENTION: \*\*each such kept item is returned inside a brace pair\*\* Use `\xintKeepUnbraced` to avoid that.

For i equal or larger to the number N of items in (expanded) L, the full L is returned (with braced items). For i=0, the macro returns an empty output. For i<0, the macro discards the first N-|i| items. No brace pairs added to the remaining items. For i is less or equal to -N, the full L is returned (with no braces added.)

`\xintKeepNoExpand` does not expand the L argument.

**Modified at 1.2i (2016/12/13).** Prior to 1.2i the code proceeded along a loop with no pre-computation of the length of L, for the i>0 case. The faster 1.2i version takes advantage of novel `\xintLengthUpTo` from `xintkernel.sty`.

```

326 \def\xintKeep          {\romannumeral0\xintkeep }%
327 \def\xintKeepNoExpand {\romannumeral0\xintkeepnoexpand }%
328 \long\def\xintkeep #1#2{\expandafter\xINT_keep_a\the\numexpr #1\expandafter.%
329 \expandafter{\romannumeral`&&@#2}}%
330 \def\xintkeepnoexpand #1{\expandafter\xINT_keep_a\the\numexpr #1.}%
331 \def\xINT_keep_a #1%
332 {%
333 \xint_UDzerominusfork
334 #1-\xINT_keep_keepnone
335 0#1\xINT_keep_neg
336 0-{\xINT_keep_pos #1}%
337 \krof
338 }%
339 \long\def\xINT_keep_keepnone .#1{ }%
340 \long\def\xINT_keep_neg #1.#2%
341 {%
342 \expandafter\xINT_keep_neg_a\the\numexpr
343 #1-\numexpr\xINT_length_loop
344 #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
345 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
346 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.#2%
347 }%
348 \def\xINT_keep_neg_a #1%
349 {%
350 \xint_UDsignfork
351 #1{\expandafter\space\romannumeral\xINT_gobble}%
352 -\xINT_keep_keepall
353 \krof
354 }%
355 \def\xINT_keep_keepall #1.{ }%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
356 \long\def\XINT_keep_pos #1.#2%
357 {%
358     \expandafter\XINT_keep_loop
359     \the\numexpr#1-\XINT_lengthupto_loop
360     #1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:
361     \xint_c_vii\xint_c_vi\xint_c_v\xint_c_iv
362     \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
363     -\xint_c_viii.{ }#2\xint_bye%
364 }%
365 \def\XINT_keep_loop #1#2.%
366 {%
367     \xint_gob_til_minus#1\XINT_keep_loop_end-%
368     \expandafter\XINT_keep_loop
369     \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keep_loop_pickeight
370 }%
371 \long\def\XINT_keep_loop_pickeight
372     #1#2#3#4#5#6#7#8#9{#1{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}}%
373 \def\XINT_keep_loop_end-\expandafter\XINT_keep_loop
374     \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keep_loop_pickeight
375     {\csname XINT_keep_end#1\endcsname}%
376 \long\expandafter\def\csname XINT_keep_end1\endcsname
377     #1#2#3#4#5#6#7#8#9\xint_bye { #1{#2}{#3}{#4}{#5}{#6}{#7}{#8}}%
378 \long\expandafter\def\csname XINT_keep_end2\endcsname
379     #1#2#3#4#5#6#7#8\xint_bye { #1{#2}{#3}{#4}{#5}{#6}{#7}}%
380 \long\expandafter\def\csname XINT_keep_end3\endcsname
381     #1#2#3#4#5#6#7\xint_bye { #1{#2}{#3}{#4}{#5}{#6}}%
382 \long\expandafter\def\csname XINT_keep_end4\endcsname
383     #1#2#3#4#5#6\xint_bye { #1{#2}{#3}{#4}{#5}}%
384 \long\expandafter\def\csname XINT_keep_end5\endcsname
385     #1#2#3#4#5\xint_bye { #1{#2}{#3}{#4}}%
386 \long\expandafter\def\csname XINT_keep_end6\endcsname
387     #1#2#3#4\xint_bye { #1{#2}{#3}}%
388 \long\expandafter\def\csname XINT_keep_end7\endcsname
389     #1#2#3\xint_bye { #1{#2}}%
390 \long\expandafter\def\csname XINT_keep_end8\endcsname
391     #1#2\xint_bye { #1}%
```

### 19.14. \xintKeepUnbraced

**Added at 1.2a (2015/10/19).** Same as `\xintKeep` but will *not* add (or maintain) brace pairs around the kept items when  $\text{length}(L) > i > 0$ .

The name may cause a mis-understanding: for  $i < 0$ , (i.e. keeping only trailing items), there is no brace removal at all happening.

**Modified at 1.2i (2016/12/13).** As `\xintKeep`.

```
392 \def\xintKeepUnbraced        {\romannumeral0\xintkeepunbraced }%
393 \def\xintKeepUnbracedNoExpand {\romannumeral0\xintkeepunbracednoexpand }%
394 \long\def\xintkeepunbraced #1#2%
395     {\expandafter\XINT_keepunbr_a\the\numexpr #1\expandafter.%
396         \expandafter{\romannumeral`&&#2}}%
397 \def\xintkeepunbracednoexpand #1%
398     {\expandafter\XINT_keepunbr_a\the\numexpr #1.}%
399 \def\XINT_keepunbr_a #1%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
400 {%
401     \xint_UDzerominusfork
402     #1-\XINT_keep_keepnone
403     0#1\XINT_keep_neg
404     0-{\XINT_keepunbr_pos #1}%
405     \krof
406 }%
407 \long\def\XINT_keepunbr_pos #1.#2%
408 {%
409     \expandafter\XINT_keepunbr_loop
410     \the\numexpr#1-\XINT_lengthupto_loop
411     #1.#2\xint:\xint:\xint:\xint:\xint:\xint:\xint:
412     \xint_c_vii\xint_c_vi\xint_c_v\xint_c_iv
413     \xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye.%
414     -\xint_c_viii.{ }#2\xint_bye%
415 }%
416 \def\XINT_keepunbr_loop #1#2.%
417 {%
418     \xint_gob_til_minus#1\XINT_keepunbr_loop_end-%
419     \expandafter\XINT_keepunbr_loop
420     \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keepunbr_loop_pickeight
421 }%
422 \long\def\XINT_keepunbr_loop_pickeight
423     #1#2#3#4#5#6#7#8#9{{#1#2#3#4#5#6#7#8#9}}%
424 \def\XINT_keepunbr_loop_end-\expandafter\XINT_keepunbr_loop
425     \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keepunbr_loop_pickeight
426     {\csname XINT_keepunbr_end#1\endcsname}%
427 \long\expandafter\def\csname XINT_keepunbr_end1\endcsname
428     #1#2#3#4#5#6#7#8#9\xint_bye { #1#2#3#4#5#6#7#8}%
429 \long\expandafter\def\csname XINT_keepunbr_end2\endcsname
430     #1#2#3#4#5#6#7#8\xint_bye { #1#2#3#4#5#6#7}%
431 \long\expandafter\def\csname XINT_keepunbr_end3\endcsname
432     #1#2#3#4#5#6#7\xint_bye { #1#2#3#4#5#6}%
433 \long\expandafter\def\csname XINT_keepunbr_end4\endcsname
434     #1#2#3#4#5#6\xint_bye { #1#2#3#4#5}%
435 \long\expandafter\def\csname XINT_keepunbr_end5\endcsname
436     #1#2#3#4#5\xint_bye { #1#2#3#4}%
437 \long\expandafter\def\csname XINT_keepunbr_end6\endcsname
438     #1#2#3#4\xint_bye { #1#2#3}%
439 \long\expandafter\def\csname XINT_keepunbr_end7\endcsname
440     #1#2#3\xint_bye { #1#2}%
441 \long\expandafter\def\csname XINT_keepunbr_end8\endcsname
442     #1#2\xint_bye { #1}%
```

### 19.15. \xintTrim

**Added at 1.09m (2014/02/26).** `\xintTrim{i}{L}` f-expands its second argument L. It then removes the first i items from L and keeps the rest. For i equal or larger to the number N of items in (expanded) L, the macro returns an empty output. For i=0, the original (expanded) L is returned. For i<0, the macro proceeds from the tail. It thus removes the last |i| items, i.e. it keeps the first N-|i| items. For |i|>= N, the empty list is returned.  
`\xintTrimNoExpand` does not expand the L argument.



**Modified at 1.2i (2016/12/13).** Speed improvements for  $i < 0$  branch (which hands over to `\xintKeep`).  
Speed improvements with 1.2j for  $i > 0$  branch which gobbles items nine by nine despite not knowing in advance if it will go too far.

```

443 \def\xintTrim          {\romannumeral0\xinttrim }%
444 \def\xintTrimNoExpand {\romannumeral0\xinttrimnoexpand }%
445 \long\def\xinttrim #1#2{\expandafter\XINT_trim_a\the\numexpr #1\expandafter.%
446                      \expandafter{\romannumeral`&&@#2}}%
447 \def\xinttrimnoexpand #1{\expandafter\XINT_trim_a\the\numexpr #1.}%
448 \def\XINT_trim_a #1%
449 {%
450   \xint_UDzerominusfork
451     #1-\XINT_trim_trimnone
452     0#1\XINT_trim_neg
453     0-{\XINT_trim_pos #1}%
454   \krof
455 }%
456 \long\def\XINT_trim_trimnone .#1{ #1}%
457 \long\def\XINT_trim_neg #1.#2%
458 {%
459   \expandafter\XINT_trim_neg_a\the\numexpr
460   #1-\numexpr\XINT_length_loop
461   #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
462   \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
463   \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
464   .{ }#2\xint_bye
465 }%
466 \def\XINT_trim_neg_a #1%
467 {%
468   \xint_UDsignfork
469   #1{\expandafter\XINT_keep_loop\the\numexpr-\xint_c_viii+}%
470   -\XINT_trim_trimall
471   \krof
472 }%
473 \def\XINT_trim_trimall#1{%
474 \def\XINT_trim_trimall {\expandafter#1\xint_bye}%
475 }\XINT_trim_trimall{ }%

```

This branch doesn't pre-evaluate the length of the list argument. Redone again for 1.2j, manages to trim nine by nine. Some non optimal looking aspect of the code is for allowing sharing with `\xintNthElt`.

```

476 \long\def\XINT_trim_pos #1.#2%
477 {%
478   \expandafter\XINT_trim_pos_done\expandafter\space
479   \romannumeral0\expandafter\XINT_trim_loop\the\numexpr#1-\xint_c_ix.%
480   #2\xint:\xint:\xint:\xint:\xint:
481   \xint:\xint:\xint:\xint:\xint:
482   \xint_bye
483 }%
484 \def\XINT_trim_loop #1#2.%
485 {%
486   \xint_gob_til_minus#1\XINT_trim_finish-%
487   \expandafter\XINT_trim_loop\the\numexpr#1#2\XINT_trim_loop_trimnine
488 }%

```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

489 \long\def\XINT_trim_loop_trimnine #1#2#3#4#5#6#7#8#9%
490 {%
491     \xint_gob_til_xint: #9\XINT_trim_toofew\xint:-\xint_c_ix.%
492 }%
493 \def\XINT_trim_toofew\xint:{*\xint_c_}%
494 \def\XINT_trim_finish#1{%
495 \def\XINT_trim_finish-
496     \expandafter\XINT_trim_loop\the\numexpr-##1\XINT_trim_loop_trimnine
497 {%
498     \expandafter\expandafter\expandafter#1%
499     \csname xint_gobble_\romannumeral\numexpr\xint_c_ix-##1\endcsname
500 }}\XINT_trim_finish{ }%
501 \long\def\XINT_trim_pos_done #1\xint:#2\xint_bye {#1}%

```

## 19.16. \xintTrimUnbraced

Added at 1.2a (2015/10/19).

Modified at 1.2i (2016/12/13). As \xintTrim.

```

502 \def\xintTrimUnbraced {\romannumeral0\xinttrimunbraced }%
503 \def\xintTrimUnbracedNoExpand {\romannumeral0\xinttrimunbracednoexpand }%
504 \long\def\xinttrimunbraced #1#2%
505     {\expandafter\XINT_trimunbr_a\the\numexpr #1\expandafter.%
506     \expandafter{\romannumeral`&&@#2}}%
507 \def\xinttrimunbracednoexpand #1%
508     {\expandafter\XINT_trimunbr_a\the\numexpr #1.}%
509 \def\XINT_trimunbr_a #1%
510 {%
511     \xint_UDzerominusfork
512     #1-\XINT_trim_trimnone
513     0#1\XINT_trimunbr_neg
514     0-{\XINT_trim_pos #1}%
515     \krof
516 }%
517 \long\def\XINT_trimunbr_neg #1.#2%
518 {%
519     \expandafter\XINT_trimunbr_neg_a\the\numexpr
520     #1-\numexpr\XINT_length_loop
521     #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
522     \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
523     \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
524     .{ }#2\xint_bye
525 }%
526 \def\XINT_trimunbr_neg_a #1%
527 {%
528     \xint_UDsignfork
529     #1{\expandafter\XINT_keeputnbr_loop\the\numexpr-\xint_c_viii+}%
530     -\XINT_trim_trimall
531     \krof
532 }%

```

## 19.17. `\xintApply`

Added at 1.04 (2013/04/25). `\xintApply` `{\macro}{a}{b}...{z}` returns `{\macro{a}}...{\macro{z}}` where each instance of `\macro` is f-expanded. The list itself is first f-expanded and may thus be a macro.

```

533 \def\xintApply          {\romannumeral0\xintapply }%
534 \def\xintApplyNoExpand {\romannumeral0\xintapplynoexpand }%
535 \long\def\xintapply #1#2%
536 {%
537     \expandafter\XINT_apply\expandafter {\romannumeral`&&@#2}%
538     {#1}%
539 }%
540 \long\def\XINT_apply #1#2{\XINT_apply_loop_a {}{#2}#1\xint_bye }%
541 \long\def\xintapplynoexpand #1#2{\XINT_apply_loop_a {}{#1}#2\xint_bye }%
542 \long\def\XINT_apply_loop_a #1#2#3%
543 {%
544     \xint_bye #3\XINT_apply_end\xint_bye
545     \expandafter
546     \XINT_apply_loop_b
547     \expandafter {\romannumeral`&&@#2{#3}}{#1}{#2}%
548 }%
549 \long\def\XINT_apply_loop_b #1#2{\XINT_apply_loop_a {#2{#1}}}%
550 \long\def\XINT_apply_end\xint_bye\expandafter\XINT_apply_loop_b
551 \expandafter #1#2#3{ #2}%

```

## 19.18. `\xintApply:x` (WIP, commented-out)

Added at 1.4 (2020/01/31) [on 2020/01/27]. For usage in the NumPy-like slicing routines. Well, actually, in the end I stuck with old-fashioned (quadratic cost) `\xintApply` for 1.4 2020/01/31 release. See comments there.

(Comments mainly from 2020/01/27, but on 2020/02/24 I comment out the code and add an alternative)

To expand in `\expanded` context, and does not need to do any expansion of its second argument.

This uses techniques I had developed for 1.2i/1.2j `Keep`, `Trim`, `Length`, `LastItem` like macros, and I should revamp venerable `\xintApply` probably too. But the latter f-expandability (if it does not have `\expanded` at disposal) complicates significantly matters as it has to store material and release at very end.

Here it is simpler and I am doing it quickly as I really want to release 1.4. The `\xint:` token should not be located in looped over items. I could use something more exotic like the null char with catcode 3...

```

\long\def\xintApply:x #1#2%
{%
    \XINT_apply:x_loop {#1}#2%
    {\xint:\XINT_apply:x_loop_enda}{\xint:\XINT_apply:x_loop_endb}%
    {\xint:\XINT_apply:x_loop_endc}{\xint:\XINT_apply:x_loop_endd}%
    {\xint:\XINT_apply:x_loop_ende}{\xint:\XINT_apply:x_loop_endf}%
    {\xint:\XINT_apply:x_loop_endg}{\xint:\XINT_apply:x_loop_endh}\xint_bye
}%
\long\def\XINT_apply:x_loop #1#2#3#4#5#6#7#8#9%
{%
    \xint_gob_til_xint: #9\xint:
    {#1{#2}}{#1{#3}}{#1{#4}}{#1{#5}}{#1{#6}}{#1{#7}}{#1{#8}}{#1{#9}}%

```

```

\XINT_apply:x_loop {#1}%
}%
\long\def\XINT_apply:x_loop_endh\xint: #1\xint_bye{%
\long\def\XINT_apply:x_loop_endg\xint: #1#2\xint_bye{{#1}}%
\long\def\XINT_apply:x_loop_endf\xint: #1#2#3\xint_bye{{#1}{#2}}%
\long\def\XINT_apply:x_loop_ende\xint: #1#2#3#4\xint_bye{{#1}{#2}{#3}}%
\long\def\XINT_apply:x_loop_endd\xint: #1#2#3#4#5\xint_bye{{#1}{#2}{#3}{#4}}%
\long\def\XINT_apply:x_loop_endc\xint: #1#2#3#4#5#6\xint_bye{{#1}{#2}{#3}{#4}{#5}}%
\long\def\XINT_apply:x_loop_endb\xint: #1#2#3#4#5#6#7\xint_bye{{#1}{#2}{#3}{#4}{#5}{#6}}%
\long\def\XINT_apply:x_loop_enda\xint: #1#2#3#4#5#6#7#8\xint_bye{{#1}{#2}{#3}{#4}{#5}{#6}{#7}}%

```

For small number of items gain with respect to `\xintApply` is little if any (might even be a loss).

Picking one by one is possibly better for small number of items. Like this for example, the natural simple minded thing:

```

\long\def\xintApply:x #1#2%
{%
\XINT_apply:x_loop {#1}#2\xint_bye\xint_bye
}%
\long\def\XINT_apply:x_loop #1#2%
{%
\xint_bye #2\xint_bye {#1}{#2}}%
\XINT_apply:x_loop {#1}%
}%

```

Some variant on 2020/02/24

```

\long\def\xint_Bye#1\xint_Bye{%
\long\def\xintApply:x #1#2%
{%
\XINT_apply:x_loop {#1}#2%
{\xint_bye}{\xint_bye}{\xint_bye}{\xint_bye}%
{\xint_bye}{\xint_bye}{\xint_bye}{\xint_bye}\xint_bye
}%
\long\def\XINT_apply:x_loop #1#2#3#4#5#6#7#8#9%
{%
\xint_Bye #2\xint_bye {#1}{#2}}%
\xint_Bye #3\xint_bye {#1}{#3}}%
\xint_Bye #4\xint_bye {#1}{#4}}%
\xint_Bye #5\xint_bye {#1}{#5}}%
\xint_Bye #6\xint_bye {#1}{#6}}%
\xint_Bye #7\xint_bye {#1}{#7}}%
\xint_Bye #8\xint_bye {#1}{#8}}%
\xint_Bye #9\xint_bye {#1}{#9}}%
\XINT_apply:x_loop {#1}%
}%

```

## 19.19. `\xintApplyUnbraced`

Added at 1.06b (2013/05/14). `\xintApplyUnbraced` `{\macro}{a}{b}...{z}` returns `\macro{a}...\macro{z}` where each instance of `\macro` is f-expanded using `\romannumeral-`0`. The second argument may be a macro as it is itself also f-expanded. No braces are added: this allows for example a non-expandable `\def` in `\macro`, without having to do `\gdef`.

```

552 \def\xintApplyUnbraced {\romannumeral0\xintapplyunbraced}%
553 \def\xintApplyUnbracedNoExpand {\romannumeral0\xintapplyunbracednoexpand}%
554 \long\def\xintapplyunbraced #1#2%

```

```

555 {%
556   \expandafter\XINT_applyunbr\expandafter {\romannumeral`&&@#2}%
557   {#1}%
558 }%
559 \long\def\XINT_applyunbr #1#2{\XINT_applyunbr_loop_a {}{#2}#1\xint_bye }%
560 \long\def\xintapplyunbracednoexpand #1#2%
561   {\XINT_applyunbr_loop_a {}{#1}#2\xint_bye }%
562 \long\def\XINT_applyunbr_loop_a #1#2#3%
563 {%
564   \xint_bye #3\XINT_applyunbr_end\xint_bye
565   \expandafter\XINT_applyunbr_loop_b
566   \expandafter {\romannumeral`&&@#2{#3}}{#1}{#2}%
567 }%
568 \long\def\XINT_applyunbr_loop_b #1#2{\XINT_applyunbr_loop_a {#2}#1}%
569 \long\def\XINT_applyunbr_end\xint_bye\expandafter\XINT_applyunbr_loop_b
570   \expandafter #1#2#3{ #2}%

```

## 19.20. \xintApplyUnbraced:x (WIP, commented-out)

**Added at 1.4 (2020/01/31) [on 2020/01/27].** For usage in the NumPy-like slicing routines.

The items should not contain `\xint:` and the applied macro should not contain `\empty`.

Finally, `xintexpr.sty` 1.4 code did not use this macro but the f-expandable one `\xintApplyUnbraced`.

**Modified at 1.4b (2020/02/25).** For 1.4b I prefer to keep the `\xintApplyUnbraced:x` code commented out, and classify it as WIP.

```

\long\def\xintApplyUnbraced:x #1#2%
{%
  \XINT_applyunbraced:x_loop {#1}#2%
  {\xint:\XINT_applyunbraced:x_loop_enda}{\xint:\XINT_applyunbraced:x_loop_endb}%
  {\xint:\XINT_applyunbraced:x_loop_endc}{\xint:\XINT_applyunbraced:x_loop_endd}%
  {\xint:\XINT_applyunbraced:x_loop_ende}{\xint:\XINT_applyunbraced:x_loop_endf}%
  {\xint:\XINT_applyunbraced:x_loop_endg}{\xint:\XINT_applyunbraced:x_loop_endh}\xint_bye
}%
\long\def\XINT_applyunbraced:x_loop #1#2#3#4#5#6#7#8#9%
{%
  \xint_gob_til_xint: #9\xint:
    #1{#2}%
    \empty#1{#3}%
    \empty#1{#4}%
    \empty#1{#5}%
    \empty#1{#6}%
    \empty#1{#7}%
    \empty#1{#8}%
    \empty#1{#9}%
  \XINT_applyunbraced:x_loop {#1}%
}%
\long\def\XINT_applyunbraced:x_loop_endh\xint: #1\xint_bye{%
\long\def\XINT_applyunbraced:x_loop_endg\xint: #1\empty#2\xint_bye{#1}%
\long\def\XINT_applyunbraced:x_loop_endf\xint: #1\empty
    #2\empty#3\xint_bye{#1#2}%
\long\def\XINT_applyunbraced:x_loop_ende\xint: #1\empty
    #2\empty
    #3\empty#4\xint_bye{#1#2#3}%

```

```

\long\def\XINT_applyunbraced:x_loop_endd\xint: #1\empty
                                         #2\empty
                                         #3\empty
                                         #4\empty#5\xint_bye{#1#2#3#4}%

\long\def\XINT_applyunbraced:x_loop_endc\xint: #1\empty
                                         #2\empty
                                         #3\empty
                                         #4\empty
                                         #5\empty#6\xint_bye{#1#2#3#4#5}%

\long\def\XINT_applyunbraced:x_loop_endb\xint: #1\empty
                                         #2\empty
                                         #3\empty
                                         #4\empty
                                         #5\empty
                                         #6\empty#7\xint_bye{#1#2#3#4#5#6}%

\long\def\XINT_applyunbraced:x_loop_enda\xint: #1\empty
                                         #2\empty
                                         #3\empty
                                         #4\empty
                                         #5\empty
                                         #6\empty
                                         #7\empty#8\xint_bye{#1#2#3#4#5#6#7}%

```

## 19.21. \xintZip (WIP, not public)

**Added at 1.4b (2020/02/25) [on 2020/02/25].** Support for zip(). Requires `\expanded`.

The implementation here thus considers the argument is already completely expanded and is a sequence of nut-ples. I will come back at later date for more generic macros.

Consider even the name of the function zip() as WIP.

As per what this does, it imitates the zip() function. See *xint-manual.pdf*.

I use lame terminators. Will think again later on this. I have to be careful with the used terminators, in particular with the NE context in mind.

Generally speaking I will think another day about efficiency else I will never start this.

OK, done. More compact than I initially thought. Various things should be commented upon here.

Well, actually not so compact in the end as I basically had to double the whole thing simply to avoid the overhead of having to grab the final result delimited by some `\xint_bye\xint_bye\xint_bye\xint_bye\empty` terminator. Now actually rather `\xint_bye\xint_bye\xint_bye\xint_bye\xint:`

```

571 \def\xintZip #1{\expanded\XINT_zip_A#1\xint_bye\xint_bye}%
572 \def\XINT_zip_A#1%
573 {%
574   \xint_bye#1{\expandafter}\xint_bye
575   \expanded{\xint_noexpd{\XINT_ziptwo_A
576     #1\xint_bye\xint_bye\xint_bye\xint_bye\xint:}\expandafter}%
577   \expanded\XINT_zip_a
578 }%
579 \def\XINT_zip_a#1%
580 {%
581   \xint_bye#1\XINT_zip_terminator\xint_bye
582   \expanded{\xint_noexpd{\XINT_ziptwo_a
583     #1\xint_bye\xint_bye\xint_bye\xint_bye\xint:}\expandafter}%
584   \expanded\XINT_zip_a
585 }%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
586 \def\XINT_zip_terminator\xint_bye#1\xint_bye{{}}\empty\empty\empty\empty\xint:}%
587 \def\XINT_ziptwo_a #1#2#3#4#5\xint:#6#7#8#9%
588 {%
589     \bgroup
590     \xint_bye #1\XINT_ziptwo_e \xint_bye
591     \xint_bye #6\XINT_ziptwo_e \xint_bye {{#1}#6}%
592     \xint_bye #2\XINT_ziptwo_e \xint_bye
593     \xint_bye #7\XINT_ziptwo_e \xint_bye {{#2}#7}%
594     \xint_bye #3\XINT_ziptwo_e \xint_bye
595     \xint_bye #8\XINT_ziptwo_e \xint_bye {{#3}#8}%
596     \xint_bye #4\XINT_ziptwo_e \xint_bye
597     \xint_bye #9\XINT_ziptwo_e \xint_bye {{#4}#9}%
```

Attention here that #6 can very well deliver no tokens at all. But the `\ifx` will then do the expected thing. Only mentioning!

By the way, the `\xint_bye` method means TeX needs to look into tokens but skipping braced groups. A conditional based method lets TeX look only at the start but then it has to find `\else` or `\fi` so here also it must look at tokens, and actually goes into braced groups. But (written 2020/02/26) I never did serious testing comparing the two, and in xint I have usually preferred `\xint_bye/\xint_gob_til_foo` types of methods (they proved superior than `\ifnum` to check for 0000 in numerical core context for example, at the early days when xint used blocks of 4 digits, not 8), or usage of `\if/\ifx` only on single tokens, combined with some `\xint_dothis/\xint_orthat` syntax.

```
598     \ifx \empty#6\expandafter\XINT_zipone_a\fi
599     \XINT_ziptwo_b #5\xint:
600 }%
601 \def\XINT_zipone_a\XINT_ziptwo_b{\XINT_zipone_b}%
602 \def\XINT_ziptwo_b #1#2#3#4#5\xint:#6#7#8#9%
603 {%
604     \xint_bye #1\XINT_ziptwo_e \xint_bye
605     \xint_bye #6\XINT_ziptwo_e \xint_bye {{#1}#6}%
606     \xint_bye #2\XINT_ziptwo_e \xint_bye
607     \xint_bye #7\XINT_ziptwo_e \xint_bye {{#2}#7}%
608     \xint_bye #3\XINT_ziptwo_e \xint_bye
609     \xint_bye #8\XINT_ziptwo_e \xint_bye {{#3}#8}%
610     \xint_bye #4\XINT_ziptwo_e \xint_bye
611     \xint_bye #9\XINT_ziptwo_e \xint_bye {{#4}#9}%
612     \XINT_ziptwo_b #5\xint:
613 }%
614 \def\XINT_ziptwo_e #1\XINT_ziptwo_b #2\xint:#3\xint:
615     {\iffalse{\fi}\xint_bye\xint_bye\xint_bye\xint_bye\xint:}%
616 \def\XINT_zipone_b #1#2#3#4%
617 {%
618     \xint_bye #1\XINT_zipone_e \xint_bye {{#1}}%
619     \xint_bye #2\XINT_zipone_e \xint_bye {{#2}}%
620     \xint_bye #3\XINT_zipone_e \xint_bye {{#3}}%
621     \xint_bye #4\XINT_zipone_e \xint_bye {{#4}}%
622     \XINT_zipone_b
623 }%
624 \def\XINT_zipone_e #1\XINT_zipone_b #2\xint:
625     {\iffalse{\fi}\xint_bye\xint_bye\xint_bye\xint_bye\empty}%
626 \def\XINT_ziptwo_A #1#2#3#4#5\xint:#6#7#8#9%
627 {%
628     \bgroup
```

```

629 \xint_bye #1\XINT_ziptwo_end \xint_bye
630 \xint_bye #6\XINT_ziptwo_end \xint_bye {{#1}#6}%
631 \xint_bye #2\XINT_ziptwo_end \xint_bye
632 \xint_bye #7\XINT_ziptwo_end \xint_bye {{#2}#7}%
633 \xint_bye #3\XINT_ziptwo_end \xint_bye
634 \xint_bye #8\XINT_ziptwo_end \xint_bye {{#3}#8}%
635 \xint_bye #4\XINT_ziptwo_end \xint_bye
636 \xint_bye #9\XINT_ziptwo_end \xint_bye {{#4}#9}%
637 \ifx \empty#6\expandafter\XINT_zipone_A\fi
638 \XINT_ziptwo_B #5\xint:
639 }%
640 \def\XINT_zipone_A\XINT_ziptwo_B{\XINT_zipone_B}%
641 \def\XINT_ziptwo_B #1#2#3#4#5\xint:#6#7#8#9%
642 {%
643 \xint_bye #1\XINT_ziptwo_end \xint_bye
644 \xint_bye #6\XINT_ziptwo_end \xint_bye {{#1}#6}%
645 \xint_bye #2\XINT_ziptwo_end \xint_bye
646 \xint_bye #7\XINT_ziptwo_end \xint_bye {{#2}#7}%
647 \xint_bye #3\XINT_ziptwo_end \xint_bye
648 \xint_bye #8\XINT_ziptwo_end \xint_bye {{#3}#8}%
649 \xint_bye #4\XINT_ziptwo_end \xint_bye
650 \xint_bye #9\XINT_ziptwo_end \xint_bye {{#4}#9}%
651 \XINT_ziptwo_B #5\xint:
652 }%
653 \def\XINT_ziptwo_end #1\XINT_ziptwo_B #2\xint:#3\xint:{\iffalse{\fi}}%
654 \def\XINT_zipone_B #1#2#3#4%
655 {%
656 \xint_bye #1\XINT_zipone_end \xint_bye {{#1}}%
657 \xint_bye #2\XINT_zipone_end \xint_bye {{#2}}%
658 \xint_bye #3\XINT_zipone_end \xint_bye {{#3}}%
659 \xint_bye #4\XINT_zipone_end \xint_bye {{#4}}%
660 \XINT_zipone_B
661 }%
662 \def\XINT_zipone_end #1\XINT_zipone_B #2\xint:#3\xint:{\iffalse{\fi}}%

```

## 19.22. \xintSeq

**Added at 1.09c (2013/10/09).** Without the optional argument puts stress on the input stack, should not be used to generated thousands of terms then.

**Modified at 1.4j (2021/07/13).** This venerable macro had a brace removal bug in case it produced a single number: `\xintSeq{10}{10}` expanded to `10` not `{10}`. When I looked at the code the bug looked almost deliberate to me, but reading the documentation (which I have not modified), the behaviour is really unexpected. And the variant with step parameter `\xintSeq[1]{10}{10}` did produce `{10}`, so yes, definitely it was a bug!

I take this occasion to do some style (and perhaps efficiency) refactoring in the coding. I feel there is room for improvement, no time this time. And I don't touch the variant with step parameter.

Memo: `xintexpr` has some variants, a priori on ultra quick look they do not look like having similar bug as this one had.

```

663 \def\xintSeq {\romannumeral0\xintseq}%
664 \def\xintseq #1{\XINT_seq_chkopt #1\xint_bye}%
665 \def\XINT_seq_chkopt #1%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
666 {%
667     \ifx [#1\expandafter\XINT_seq_opt
668         \else\expandafter\XINT_seq_noopt
669     \fi #1%
670 }%
671 \def\XINT_seq_noopt #1\xint_bye #2%
672 {%
673     \expandafter\XINT_seq
674     \the\numexpr#1\expandafter.\the\numexpr #2.%
675 }%
676 \def\XINT_seq #1.#2.%
677 {%
678     \ifnum #1=#2 \xint_dothis\XINT_seq_e\fi
679     \ifnum #2>#1 \xint_dothis\XINT_seq_pa\fi
680         \xint_orthat\XINT_seq_na
681     #2.{#1}{#2}%
682 }%
683 \def\XINT_seq_e#1.#2{%
684 \def\XINT_seq_pa {\expandafter\XINT_seq_p\the\numexpr-\xint_c_i+}%
685 \def\XINT_seq_na {\expandafter\XINT_seq_n\the\numexpr\xint_c_i+}%
686 \def\XINT_seq_p #1.#2%
687 {%
688     \ifnum #1>#2
689         \expandafter\XINT_seq_p\the
690     \else
691         \expandafter\XINT_seq_e
692     \fi
693     \numexpr #1-\xint_c_i.{#2}{#1}%
694 }%
695 \def\XINT_seq_n #1.#2%
696 {%
697     \ifnum #1<#2
698         \expandafter\XINT_seq_n\the
699     \else
700         \expandafter\XINT_seq_e
701     \fi
702     \numexpr #1+\xint_c_i.{#2}{#1}%
703 }%
```

Note at time of the 1.4j bug fix : I definitely should improve this branch and diminish the number of `\expandafter`'s but no time this time.

```
704 \def\XINT_seq_opt [\xint_bye #1]#2#3%
705 {%
706     \expandafter\XINT_seqo\expandafter
707     {\the\numexpr #2\expandafter}\expandafter
708     {\the\numexpr #3\expandafter}\expandafter
709     {\the\numexpr #1}%
710 }%
711 \def\XINT_seqo #1#2%
712 {%
713     \ifcase\ifnum #1=#2 0\else\ifnum #2>#1 1\else -1\fi\fi\space
714         \expandafter\XINT_seqo_a
715     \or
```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

716     \expandafter\XINT_seqo_pa
717 \else
718     \expandafter\XINT_seqo_na
719 \fi
720     {#1}{#2}%
721 }%
722 \def\XINT_seqo_a #1#2#3{ {#1}}%
723 \def\XINT_seqo_o #1#2#3#4{ #4}%
724 \def\XINT_seqo_pa #1#2#3%
725 {%
726     \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
727         \expandafter\XINT_seqo_o
728     \or
729         \expandafter\XINT_seqo_pb
730     \else
731         \xint_afterfi{\expandafter\space\xint_gobble_iv}%
732     \fi
733     {#1}{#2}{#3}{#1}%
734 }%
735 \def\XINT_seqo_pb #1#2#3%
736 {%
737     \expandafter\XINT_seqo_pc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
738 }%
739 \def\XINT_seqo_pc #1#2%
740 {%
741     \ifnum #1>#2
742         \expandafter\XINT_seqo_o
743     \else
744         \expandafter\XINT_seqo_pd
745     \fi
746     {#1}{#2}%
747 }%
748 \def\XINT_seqo_pd #1#2#3#4{\XINT_seqo_pb {#1}{#2}{#3}{#4{#1}}}%
749 \def\XINT_seqo_na #1#2#3%
750 {%
751     \ifcase\ifnum #3=\xint_c_ 0\else\ifnum #3>\xint_c_ 1\else -1\fi\fi\space
752         \expandafter\XINT_seqo_o
753     \or
754         \xint_afterfi{\expandafter\space\xint_gobble_iv}%
755     \else
756         \expandafter\XINT_seqo_nb
757     \fi
758     {#1}{#2}{#3}{#1}%
759 }%
760 \def\XINT_seqo_nb #1#2#3%
761 {%
762     \expandafter\XINT_seqo_nc\expandafter{\the\numexpr #1+#3}{#2}{#3}%
763 }%
764 \def\XINT_seqo_nc #1#2%
765 {%
766     \ifnum #1<#2
767         \expandafter\XINT_seqo_o

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
768 \else
769 \expandafter\XINT_seqo_nd
770 \fi
771 {#1}{#2}%
772 }%
773 \def\XINT_seqo_nd #1#2#3#4{\XINT_seqo_nb {#1}{#2}{#3}{#4{#1}}}%
```

### 19.23. \xintloop, \xintbreakloop, \xintbreakloopanddo, \xintloopskiptonext

Added at 1.09g (2013/11/22) [on 2013/11/22].

Modified at 1.09h (2013/11/28). Made \long.

```
774 \long\def\xintloop #1#2\repeat {#1#2\xintloop_again\fi\xint_gobble_i {#1#2}}%
775 \long\def\xintloop_again\fi\xint_gobble_i #1{\fi
776 #1\xintloop_again\fi\xint_gobble_i {#1}}%
777 \long\def\xintbreakloop #1\xintloop_again\fi\xint_gobble_i #2{%
778 \long\def\xintbreakloopanddo #1#2\xintloop_again\fi\xint_gobble_i #3{#1}%
779 \long\def\xintloopskiptonext #1\xintloop_again\fi\xint_gobble_i #2{%
780 #2\xintloop_again\fi\xint_gobble_i {#2}}%
```

### 19.24. \xintiloop, \xintiloopindex, \xintbracediloopindex, \xintouteriloopindex, \xintbracedouteriloopindex, \xintbreakiloop, \xintbreakiloopanddo, \xintiloopskiptonext, \xintiloopskipandredo

Added at 1.09g (2013/11/22) [on 2013/11/22].

Modified at 1.09h (2013/11/28). Made \long.

Added at 1.3b (2018/05/18) [on 2018/04/24]. “braced” variants.

```
781 \def\xintiloop [#1+#2]{%
782 \expandafter\xintiloop_a\the\numexpr #1\expandafter.\the\numexpr #2.}%
783 \long\def\xintiloop_a #1.#2.#3#4\repeat{%
784 #3#4\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3#4}}%
785 \def\xintiloop_again\fi\xint_gobble_iii #1#2{%
786 \fi\expandafter\xintiloop_again_b\the\numexpr#1+#2.#2.}%
787 \long\def\xintiloop_again_b #1.#2.#3{%
788 #3\xintiloop_again\fi\xint_gobble_iii {#1}{#2}{#3}}%
789 \long\def\xintbreakiloop #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{%
790 \long\def\xintbreakiloopanddo
791 #1.#2\xintiloop_again\fi\xint_gobble_iii #3#4#5{#1}%
792 \long\def\xintiloopindex #1\xintiloop_again\fi\xint_gobble_iii #2%
793 {#2#1\xintiloop_again\fi\xint_gobble_iii {#2}}%
794 \long\def\xintbracediloopindex #1\xintiloop_again\fi\xint_gobble_iii #2%
795 {{#2}#1\xintiloop_again\fi\xint_gobble_iii {#2}}%
796 \long\def\xintouteriloopindex #1\xintiloop_again
797 #2\xintiloop_again\fi\xint_gobble_iii #3%
798 {#3#1\xintiloop_again #2\xintiloop_again\fi\xint_gobble_iii {#3}}%
799 \long\def\xintbracedouteriloopindex #1\xintiloop_again
800 #2\xintiloop_again\fi\xint_gobble_iii #3%
801 {{#3}#1\xintiloop_again #2\xintiloop_again\fi\xint_gobble_iii {#3}}%
802 \long\def\xintiloopskiptonext #1\xintiloop_again\fi\xint_gobble_iii #2#3{%
803 \expandafter\xintiloop_again_b \the\numexpr#2+#3.#3.}%
804 \long\def\xintiloopskipandredo #1\xintiloop_again\fi\xint_gobble_iii #2#3#4{%
805 #4\xintiloop_again\fi\xint_gobble_iii {#2}{#3}{#4}}%
```

## 19.25. \XINT\_xflet

**Added at 1.09e (2013/10/29) [on 2013/10/29].** We f-expand unbraced tokens and swallow arising space tokens until the dust settles.

```

806 \def\XINT_xflet #1%
807 {%
808   \def\XINT_xflet_macro {#1}\XINT_xflet_zapsp
809 }%
810 \def\XINT_xflet_zapsp
811 {%
812   \expandafter\futurelet\expandafter\XINT_token
813   \expandafter\XINT_xflet_sp?\romannumeral`&&@%
814 }%
815 \def\XINT_xflet_sp?
816 {%
817   \ifx\XINT_token\XINT_sptoken
818     \expandafter\XINT_xflet_zapsp
819   \else\expandafter\XINT_xflet_zapspB
820   \fi
821 }%
822 \def\XINT_xflet_zapspB
823 {%
824   \expandafter\futurelet\expandafter\XINT_tokenB
825   \expandafter\XINT_xflet_spB?\romannumeral`&&@%
826 }%
827 \def\XINT_xflet_spB?
828 {%
829   \ifx\XINT_tokenB\XINT_sptoken
830     \expandafter\XINT_xflet_zapspB
831   \else\expandafter\XINT_xflet_eq?
832   \fi
833 }%
834 \def\XINT_xflet_eq?
835 {%
836   \ifx\XINT_token\XINT_tokenB
837     \expandafter\XINT_xflet_macro
838   \else\expandafter\XINT_xflet_zapsp
839   \fi
840 }%
```

## 19.26. \xintApplyInline

**Added at 1.09a (2013/09/24).** `\xintApplyInline\macro{a}{b}...{z}` has the same effect as executing `\macro{a}` and then applying again `\xintApplyInline` to the shortened list `{b}...{z}` until nothing is left. This is a non-expandable command which will result in quicker code than using `\xintApplyUnbraced`. It f-expands its second (list) argument first, which may thus be encapsulated in a macro.

**Modified at 1.09c (2013/10/09).** Rewritten. Nota bene: uses catcode 3 Z as privated list terminator.

```

841 \catcode`Z 3
842 \long\def\xintApplyInline #1#2%
843 {%
```

```

844 \long\expandafter\def\expandafter\XINT_inline_macro
845 \expandafter ##\expandafter 1\expandafter {#1{##1}}%
846 \XINT_xflet\XINT_inline_b #2Z% this Z has catcode 3
847 }%
848 \def\XINT_inline_b
849 {%
850 \ifx\XINT_token Z\expandafter\xint_gobble_i
851 \else\expandafter\XINT_inline_d\fi
852 }%
853 \long\def\XINT_inline_d #1%
854 {%
855 \long\def\XINT_item{#1}\XINT_xflet\XINT_inline_e
856 }%
857 \def\XINT_inline_e
858 {%
859 \ifx\XINT_token Z\expandafter\XINT_inline_w
860 \else\expandafter\XINT_inline_f\fi
861 }%
862 \def\XINT_inline_f
863 {%
864 \expandafter\XINT_inline_g\expandafter{\XINT_inline_macro {##1}}%
865 }%
866 \long\def\XINT_inline_g #1%
867 {%
868 \expandafter\XINT_inline_macro\XINT_item
869 \long\def\XINT_inline_macro ##1{#1}\XINT_inline_d
870 }%
871 \def\XINT_inline_w #1%
872 {%
873 \expandafter\XINT_inline_macro\XINT_item
874 }%

```

## 19.27. \xintFor, \xintFor\*, \xintBreakFor, \xintBreakForAndDo

**Added at 1.09c (2013/10/09) [on 2013/10/09].** A new kind of loop which uses macro parameters #1, #2, #3, #4 rather than macros; while not expandable it survives executing code closing groups, like what happens in an alignment with the & character. When inserted in a macro for later use, the # character must be doubled.

The non-star variant works on a csv list, which it expands once, the star variant works on a token list, which it (repeatedly) f-expands.

**Modified at 1.09e (2013/10/29).** Adds `\XINT_forever` with `\xintintegers`, `\xintdimensions`, `\xinrationals` and `\xintBreakFor`, `\xintBreakForAndDo`, `\xintifForFirst`, `\xintifForLast`. On this occasion `\xint_firstoftwo` and `\xint_secondoftwo` are made long.

**Modified at 1.09f (2013/11/04).** Rewrites large parts of `\xintFor` code in order to filter the comma separated list via `\xintCSVtoList` which gets rid of spaces. The #1 in `\XINT_for_forever?` has an initial space token which serves two purposes: preventing brace stripping, and stopping the expansion made by `\xintcsvtolist`. If the `\XINT_forever` branch is taken, the added space will not be a problem there.

Now allows all macro parameters from #1 to #9 in `\xintFor`, `\xintFor*`, and `\XINT_forever`.

**Modified at 1.2i (2016/12/13).** Slightly more robust `\xintifForFirst/Last` in case of nesting.

```

875 \def\XINT_tmpa #1#2{\ifnum #2<#1 \xint_afterfi {{#####2}}\fi}%
876 \def\XINT_tmpb #1#2{\ifnum #1<#2 \xint_afterfi {{#####2}}\fi}%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfraction](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

877 \def\XINT_tmpc #1%
878 {%
879   \expandafter\edef \csname XINT_for_left#1\endcsname
880     {\xintApplyUnbraced {\XINT_tmpa #1}{123456789}}%
881   \expandafter\edef \csname XINT_for_right#1\endcsname
882     {\xintApplyUnbraced {\XINT_tmpb #1}{123456789}}%
883 }%
884 \xintApplyInline \XINT_tmpc {123456789}%
885 \long\def\xintBreakFor #1Z{%
886 \long\def\xintBreakForAndDo #1#2Z{#1}%
887 \def\xintFor {\let\xintifForFirst\xint_firstoftwo
888   \let\xintifForLast\xint_secondoftwo
889   \futurelet\XINT_token\XINT_for_ifstar }%
890 \def\XINT_for_ifstar {\ifx\XINT_token*\expandafter\XINT_forx
891   \else\expandafter\XINT_for \fi }%
892 \catcode\U 3 % with numexpr
893 \catcode\V 3 % with xintfrac.sty (xint.sty not enough)
894 \catcode\D 3 % with dimexpr
895 \def\XINT_flet_zapsp
896 {%
897   \futurelet\XINT_token\XINT_flet_sp?
898 }%
899 \def\XINT_flet_sp?
900 {%
901   \ifx\XINT_token\XINT_sptoken
902     \xint_afterfi{\expandafter\XINT_flet_zapsp\romannumeral0}%
903   \else\expandafter\XINT_flet_macro
904   \fi
905 }%
906 \long\def\XINT_for #1#2in#3#4#5%
907 {%
908   \expandafter\XINT_toks\expandafter
909     {\expandafter\XINT_for_d\the\numexpr #2\relax {#5}}%
910   \def\XINT_flet_macro {\expandafter\XINT_for_forever?\space}%
911   \expandafter\XINT_flet_zapsp #3Z%
912 }%
913 \def\XINT_for_forever? #1Z%
914 {%
915   \ifx\XINT_token U\XINT_to_forever\fi
916   \ifx\XINT_token V\XINT_to_forever\fi
917   \ifx\XINT_token D\XINT_to_forever\fi
918   \expandafter\the\expandafter\XINT_toks\romannumeral0\xintcsvtolist {#1}Z%
919 }%
920 \def\XINT_to_forever\fi #1\xintcsvtolist #2{\fi \XINT_forever #2}%
921 \long\def\XINT_forx *#1#2in#3#4#5%
922 {%
923   \expandafter\XINT_toks\expandafter
924     {\expandafter\XINT_forx_d\the\numexpr #2\relax {#5}}%
925   \XINT_xflet\XINT_forx_forever? #3Z%
926 }%
927 \def\XINT_forx_forever?
928 {%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
929 \ifx\XINT_token U\XINT_to_forever\fi
930 \ifx\XINT_token V\XINT_to_forever\fi
931 \ifx\XINT_token D\XINT_to_forever\fi
932 \XINT_forx_empty?
933 }%
934 \def\XINT_to_forever\fi #1\XINT_forx_empty? {\fi \XINT_forever }%
935 \catcode`U 11
936 \catcode`D 11
937 \catcode`V 11
938 \def\XINT_forx_empty?
939 {%
940 \ifx\XINT_token Z\expandafter\xintBreakFor\fi
941 \the\XINT_toks
942 }%
943 \long\def\XINT_for_d #1#2#3%
944 {%
945 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
946 \XINT_toks {{#3}}%
947 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
948 \the\XINT_toks \csname XINT_for_right#1\endcsname }%
949 \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo
950 \let\xintifForLast\xint_secondoftwo\XINT_for_d #1{#2}}%
951 \futurelet\XINT_token\XINT_for_last?
952 }%
953 \long\def\XINT_forx_d #1#2#3%
954 {%
955 \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
956 \XINT_toks {{#3}}%
957 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
958 \the\XINT_toks \csname XINT_for_right#1\endcsname }%
959 \XINT_toks {\XINT_x\let\xintifForFirst\xint_secondoftwo
960 \let\xintifForLast\xint_secondoftwo\XINT_forx_d #1{#2}}%
961 \XINT_xflet\XINT_for_last?
962 }%
963 \def\XINT_for_last?
964 {%
965 \ifx\XINT_token Z\expandafter\XINT_for_last?yes\fi
966 \the\XINT_toks
967 }%
968 \def\XINT_for_last?yes
969 {%
970 \let\xintifForLast\xint_firstoftwo
971 \xintBreakForAndDo{\XINT_x\xint_gobble_i Z}%
972 }%
```

### 19.28. \XINT\_forever, \xintintegers, \xintdimensions, \xintrationals

**Added at 1.09e (2013/10/29).** But this used inadvertently `\xintiadd/\xintimul` which have the unnecessary `\xintnum` overhead.

**Modified at 1.09f (2013/11/04).** Use `\xintiadd/\xintiimul` which do not have this overhead. Also 1.09f uses `\xintZapSpacesB` for the `\xintrationals` case to get rid of leading and ending spaces in the #4 and #5 delimited parameters of `\XINT_forever_opt_a` (for `\xintintegers` and

`\xintdimensions` this is not necessary, due to the use of `\numexpr` resp. `\dimexpr` in `\XINT_?_`  
`expr_Ua`, resp. `\XINT_?expr_Da`).

```

973 \catcode`U 3
974 \catcode`D 3
975 \catcode`V 3
976 \let\xintegers U%
977 \let\xintintegers U%
978 \let\xintdimensions D%
979 \let\xintrationals V%
980 \def\XINT_forever #1%
981 {%
982 \expandafter\XINT_forever_a
983 \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi a\expandafter\endcsname
984 \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi i\expandafter\endcsname
985 \csname XINT_?expr_\ifx#1UU\else\ifx#1DD\else V\fi\fi \endcsname
986 }%
987 \catcode`U 11
988 \catcode`D 11
989 \catcode`V 11
990 \def\XINT_?expr_Ua #1#2%
991 {\expandafter{\expandafter\numexpr\the\numexpr #1\expandafter\relax
992 \expandafter\relax\expandafter}%
993 \expandafter{\the\numexpr #2}}%
994 \def\XINT_?expr_Da #1#2%
995 {\expandafter{\expandafter\dimexpr\the\dimexpr #1\expandafter\relax
996 \expandafter s\expandafter p\expandafter\relax\expandafter}%
997 \expandafter{\the\dimexpr #2}}%
998 \catcode`Z 11
999 \def\XINT_?expr_Va #1#2%
1000 {%
1001 \expandafter\XINT_?expr_Vb\expandafter
1002 {\romannumeral\&&\xintrawithzeros{\xintZapSpacesB{#2}}}%
1003 {\romannumeral\&&\xintrawithzeros{\xintZapSpacesB{#1}}}%
1004 }%
1005 \catcode`Z 3
1006 \def\XINT_?expr_Vb #1#2{\expandafter\XINT_?expr_Vc #2.#1}%
1007 \def\XINT_?expr_Vc #1/#2.#3/#4.%
1008 {%
1009 \xintifEq {#2}{#4}%
1010 {\XINT_?expr_Vf {#3}{#1}{#2}}%
1011 {\expandafter\XINT_?expr_Vd\expandafter
1012 {\romannumeral0\xintiimul {#2}{#4}}%
1013 {\romannumeral0\xintiimul {#1}{#4}}%
1014 {\romannumeral0\xintiimul {#2}{#3}}%
1015 }%
1016 }%
1017 \def\XINT_?expr_Vd #1#2#3{\expandafter\XINT_?expr_Ve\expandafter {#2}{#3}{#1}}%
1018 \def\XINT_?expr_Ve #1#2{\expandafter\XINT_?expr_Vf\expandafter {#2}{#1}}%
1019 \def\XINT_?expr_Vf #1#2#3{{#2/#3}{#0}{#1}{#2}{#3}}%
1020 \def\XINT_?expr_Ui {\numexpr 1\relax}{1}%
1021 \def\XINT_?expr_Di {\dimexpr 0pt\relax}{65536}%
1022 \def\XINT_?expr_Vi {{1/1}{0111}}%

```

```

1023 \def\XINT_?expr_U #1#2%
1024   {\expandafter{\expandafter\numexpr\the\numexpr #1+#2\relax\relax}{#2}}%
1025 \def\XINT_?expr_D #1#2%
1026   {\expandafter{\expandafter\dimexpr\the\numexpr #1+#2\relax sp\relax}{#2}}%
1027 \def\XINT_?expr_V #1#2{\XINT_?expr_Vx #2}%
1028 \def\XINT_?expr_Vx #1#2%
1029 {%
1030   \expandafter\XINT_?expr_Vy\expandafter
1031     {\romannumeral0\xintiiadd {#1}{#2}}{#2}%
1032 }%
1033 \def\XINT_?expr_Vy #1#2#3#4%
1034 {%
1035   \expandafter{\romannumeral0\xintiiadd {#3}{#1}/#4}{#1}{#2}{#3}{#4}%
1036 }%
1037 \def\XINT_forever_a #1#2#3#4%
1038 {%
1039   \ifx #4[\expandafter\XINT_forever_opt_a
1040     \else\expandafter\XINT_forever_b
1041   \fi #1#2#3#4%
1042 }%
1043 \def\XINT_forever_b #1#2#3Z{\expandafter\XINT_forever_c\the\XINT_toks #2#3}%
1044 \long\def\XINT_forever_c #1#2#3#4#5%
1045   {\expandafter\XINT_forever_d\expandafter #2#4#5{#3}Z}%
1046 \def\XINT_forever_opt_a #1#2#3[#4+#5]#6Z%
1047 {%
1048   \expandafter\expandafter\expandafter
1049     \XINT_forever_opt_c\expandafter\the\expandafter\XINT_toks
1050     \romannumeral`&&@#1{#4}{#5}#3%
1051 }%
1052 \long\def\XINT_forever_opt_c #1#2#3#4#5#6{\XINT_forever_d #2{#4}{#5}#6{#3}Z}%
1053 \long\def\XINT_forever_d #1#2#3#4#5%
1054 {%
1055   \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#5}%
1056   \XINT_toks {{#2}}%
1057   \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
1058     \the\XINT_toks \csname XINT_for_right#1\endcsname }%
1059   \XINT_x
1060   \let\xintifForFirst\xint_secondoftwo
1061   \let\xintifForLast\xint_secondoftwo
1062   \expandafter\XINT_forever_d\expandafter #1\romannumeral`&&@#4{#2}{#3}#4{#5}%
1063 }%

```

## 19.29. \xintForpair, \xintForthree, \xintForfour

Added at 1.09c (2013/10/09).

Modified at 1.09f (2013/11/04). [\xintForpair](#) delegate to [\xintCSVtoList](#) and its [\xintZapSpacesB](#) the handling of spaces. Does not share code with [\xintFor](#) anymore.

[\xintForpair](#) extended to accept #1#2, #2#3 etc... up to #8#9, [\xintForthree](#), #1#2#3 up to #7#8#9, [\xintForfour](#) id.

Modified at 1.2i (2016/12/13). Slightly more robust [\xintifForFirst](#)/[\xintifForLast](#) in case of nesting.

Modified at 1.4n (2025/09/05). Allow one final extraneous comma (formerly, such input caused a



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

crash). This goes via testing an extra token #6, which should be an opening parenthesis. The potential brace removal changes behavior for illegal inputs. Successive commas in the input list are still not allowed.

```
1064 \catcode`j 3
1065 \long\def\xintForpair #1#2#3in#4#5#6%
1066 {%
1067   \let\xintifForFirst\xint_firstoftwo
1068   \let\xintifForLast\xint_secondoftwo
1069   \XINT_toks {\XINT_forpair_d #2{#6}}%
1070   \expandafter\the\expandafter\XINT_toks #4jZ%
1071 }%
1072 \long\def\XINT_forpair_d #1#2#3(#4)#5#6%
1073 {%
1074   \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
1075   \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
1076   \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
1077     \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_i\endcsname}%
1078   \if1\ifx #5j1\else\ifx#6j1\else0\fi\fi\expandafter\XINT_for_last?yes\fi
1079   \XINT_x
1080   \let\xintifForFirst\xint_secondoftwo
1081   \let\xintifForLast\xint_secondoftwo
1082   \XINT_forpair_d #1{#2}#6%
1083 }%
1084 \long\def\xintForthree #1#2#3in#4#5#6%
1085 {%
1086   \let\xintifForFirst\xint_firstoftwo
1087   \let\xintifForLast\xint_secondoftwo
1088   \XINT_toks {\XINT_forthree_d #2{#6}}%
1089   \expandafter\the\expandafter\XINT_toks #4jZ%
1090 }%
1091 \long\def\XINT_forthree_d #1#2#3(#4)#5#6%
1092 {%
1093   \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
1094   \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
1095   \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
1096     \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_ii\endcsname}%
1097   \if1\ifx #5j1\else\ifx#6j1\else0\fi\fi\expandafter\XINT_for_last?yes\fi
1098   \XINT_x
1099   \let\xintifForFirst\xint_secondoftwo
1100   \let\xintifForLast\xint_secondoftwo
1101   \XINT_forthree_d #1{#2}#6%
1102 }%
1103 \long\def\xintForfour #1#2#3in#4#5#6%
1104 {%
1105   \let\xintifForFirst\xint_firstoftwo
1106   \let\xintifForLast\xint_secondoftwo
1107   \XINT_toks {\XINT_forfour_d #2{#6}}%
1108   \expandafter\the\expandafter\XINT_toks #4jZ%
1109 }%
1110 \long\def\XINT_forfour_d #1#2#3(#4)#5#6%
1111 {%
1112   \long\def\XINT_y ##1##2##3##4##5##6##7##8##9{#2}%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1113 \XINT_toks \expandafter{\romannumeral0\xintcsvtolist{ #4}}%
1114 \long\edef\XINT_x {\noexpand\XINT_y \csname XINT_for_left#1\endcsname
1115 \the\XINT_toks \csname XINT_for_right\the\numexpr#1+\xint_c_iii\endcsname}%
1116 \if1\ifx #5j1\else\ifx#6j1\else0\fi\fi\expandafter\XINT_for_last?yes\fi
1117 \XINT_x
1118 \let\xintifForFirst\xint_seconddoftwo
1119 \let\xintifForLast\xint_seconddoftwo
1120 \XINT_forfour_d #1{#2}#6%
1121 }%
1122 \catcode`Z 11
1123 \catcode`j 11
```

### 19.30. \xintAssign, \xintAssignArray, \xintDigitsOf

[\xintAssign](#) {a}{b}..{z}\to A\B...Z resp. [\xintAssignArray](#) {a}{b}..{z}\to U.  
[\xintDigitsOf](#)=[\xintAssignArray](#).

**Modified at 1.1c (2015/09/12).** Belatedly corrects some "features" of [\xintAssign](#) which didn't like the case of a space right before the "[\to](#)", or the case with the first token not an opening brace and the subsequent material containing brace groups. The new code handles gracefully these situations.

```
1124 \def\xintAssign{\def\XINT_flet_macro {\XINT_assign_fork}\XINT_flet_zapsp }%
1125 \def\XINT_assign_fork
1126 {%
1127 \let\XINT_assign_def\def
1128 \ifx\XINT_token[\expandafter\XINT_assign_opt
1129 \else\expandafter\XINT_assign_a
1130 \fi
1131 }%
1132 \def\XINT_assign_opt [#1]%
1133 {%
1134 \ifcsname #1def\endcsname
1135 \expandafter\let\expandafter\XINT_assign_def \csname #1def\endcsname
1136 \else
1137 \expandafter\let\expandafter\XINT_assign_def \csname xint#1def\endcsname
1138 \fi
1139 \XINT_assign_a
1140 }%
1141 \long\def\XINT_assign_a #1\to
1142 {%
1143 \def\XINT_flet_macro{\XINT_assign_b}%
1144 \expandafter\XINT_flet_zapsp\romannumeral`&&@#1\xint:\to
1145 }%
1146 \long\def\XINT_assign_b
1147 {%
1148 \ifx\XINT_token\bgroup
1149 \expandafter\XINT_assign_c
1150 \else\expandafter\XINT_assign_f
1151 \fi
1152 }%
1153 \long\def\XINT_assign_f #1\xint:\to #2%
1154 {%
1155 \XINT_assign_def #2{#1}%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1156 }%
1157 \long\def\XINT_assign_c #1%
1158 {%
1159   \def\XINT_assign_tmp {#1}%
1160   \ifx\XINT_assign_tmp\xint_bracedstopper
1161     \expandafter\XINT_assign_e
1162   \else
1163     \expandafter\XINT_assign_d
1164   \fi
1165 }%
1166 \long\def\XINT_assign_d #1\to #2%
1167 {%
1168   \expandafter\XINT_assign_def\expandafter #2\expandafter{\XINT_assign_tmp}%
1169   \XINT_assign_c #1\to
1170 }%
1171 \def\XINT_assign_e #1\to {}%
1172 \def\xintRelaxArray #1%
1173 {%
1174   \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax}%
1175   \escapechar -1
1176   \expandafter\def\expandafter\xint_arrayname\expandafter {\string #1}%
1177   \XINT_restoreescapechar
1178   \xintilooop [\csname\xint_arrayname 0\endcsname+-1]
1179   \global
1180   \expandafter\let\csname\xint_arrayname\xintilooopindex\endcsname\relax
1181   \ifnum \xintilooopindex > \xint_c_
1182   \repeat
1183   \global\expandafter\let\csname\xint_arrayname 00\endcsname\relax
1184   \global\let #1\relax
1185 }%
1186 \def\xintAssignArray{\def\XINT_flet_macro {\XINT_assignarray_fork}%
1187   \XINT_flet_zapsp }%
1188 \def\XINT_assignarray_fork
1189 {%
1190   \let\XINT_assignarray_def\def
1191   \ifx\XINT_token[\expandafter\XINT_assignarray_opt
1192     \else\expandafter\XINT_assignarray
1193   \fi
1194 }%
1195 \def\XINT_assignarray_opt [#1]%
1196 {%
1197   \ifcsname #1def\endcsname
1198     \expandafter\let\expandafter\XINT_assignarray_def \csname #1def\endcsname
1199   \else
1200     \expandafter\let\expandafter\XINT_assignarray_def
1201       \csname xint#1def\endcsname
1202   \fi
1203   \XINT_assignarray
1204 }%
1205 \long\def\XINT_assignarray #1\to #2%
1206 {%
1207   \edef\XINT_restoreescapechar {\escapechar\the\escapechar\relax }%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1208 \escapechar -1
1209 \expandafter\def\expandafter\xint_arrayname\expandafter {\string #2}%
1210 \XINT_restoreescapechar
1211 \def\xint_itemcount {0}%
1212 \expandafter\XINT_assignarray_loop \romannumeral`&&@#1\xint:
1213 \csname\xint_arrayname 00\expandafter\endcsname
1214 \csname\xint_arrayname 0\expandafter\endcsname
1215 \expandafter {\xint_arrayname}#2%
1216 }%
1217 \long\def\XINT_assignarray_loop #1%
1218 {%
1219 \def\XINT_assign_tmp {#1}%
1220 \ifx\XINT_assign_tmp\xint_bracedstopper
1221 \expandafter\def\csname\xint_arrayname 0\expandafter\endcsname
1222 \expandafter{\the\numexpr\xint_itemcount}%
1223 \expandafter\expandafter\expandafter\XINT_assignarray_end
1224 \else
1225 \expandafter\def\expandafter\xint_itemcount\expandafter
1226 {\the\numexpr\xint_itemcount+\xint_c_i}%
1227 \expandafter\XINT_assignarray_def
1228 \csname\xint_arrayname\xint_itemcount\expandafter\endcsname
1229 \expandafter{\XINT_assign_tmp }%
1230 \expandafter\XINT_assignarray_loop
1231 \fi
1232 }%
1233 \def\XINT_assignarray_end #1#2#3#4%
1234 {%
1235 \def #4##1%
1236 {%
1237 \romannumeral0\expandafter #1\expandafter{\the\numexpr ##1}%
1238 }%
1239 \def #1##1%
1240 {%
1241 \ifnum ##1<\xint_c_
1242 \xint_afterfi{\XINT_expandableerror{Array index is negative: ##1.} }%
1243 \else
1244 \xint_afterfi {%
1245 \ifnum ##1>#2
1246 \xint_afterfi
1247 {\XINT_expandableerror{Array index is beyond range: ##1 > #2.} }%
1248 \else\xint_afterfi
1249 {\expandafter\expandafter\expandafter\space\csname #3##1\endcsname}%
1250 \fi}%
1251 \fi
1252 }%
1253 }%
1254 \let\xintDigitsOf\xintAssignArray

```

## 19.31. CSV (non user documented) variants of Length, Keep, Trim, NthElt, Reverse

Modified at 1.2j (2016/12/22). These routines are for use by `\xintListSel:x:csv` and `\xintListSel,f:csv` from `xintexpr`, and also for the `reversed` and `len` functions. Refactored for 1.2j release, following 1.2i updates to `\xintKeep`, `\xintTrim`, ...

These macros will remain undocumented in the user manual:

-- they exist primarily for internal use by the `xintexpr` parsers, hence don't have to be general purpose; for example, they a priori need to handle only catcode 12 tokens (not true in `\xintNewExpr`, though) hence they are not really worried about controlling brace stripping (nevertheless 1.2j has paid some secondary attention to it, see below.) They are not worried about normalizing leading spaces either, because none will be encountered when the macros are used as auxiliaries to the expression parsers.

-- crucial design elements may change in future:

1. whether the handled lists must have or not have a final comma. Currently, the model is the one of comma separated lists with `**no**` final comma. But this means that there can not be a distinction of principle between a truly empty list and a list which contains one item which turns out to be empty. More importantly it makes the coding more complicated as it is needed to distinguish the empty list from the single-item list, both lacking commas.

For the internal use of `xintexpr`, it would be ok to require all list items to be terminated by a comma, and this would bring quite some simplifications here, but as initially I started with non-terminated lists, I have left it this way in the 1.2j refactoring.

2. the way to represent the empty list. I was tempted for matter of optimization and synchronization with `xintexpr` context to require the empty list to be always represented by a space token and to not let the macros admit a completely empty input. But there were complications so for the time being 1.2j does accept truly empty output (it is not distinguished from an input equal to a space token) and produces empty output for empty list. This means that the status of the «nil» object for the `xintexpr` parsers is not completely clarified (currently it is represented by a space token).

The original Python slicing code in `xintexpr 1.1` used `\xintCSVtoList` and `\xintListWithSep{,}` to convert back and forth to token lists and apply `\xintKeep/\xintTrim`. Release 1.2g switched to devoted f-expandable macros added to `xinttools`. Release 1.2j refactored all these macros as a follow-up to 1.2i improvements to `\xintKeep/\xintTrim`. They were made `\long` on this occasion and auxiliary `\xintLengthUpTo:f:csv` was added.

Leading spaces in items are currently maintained as is by the 1.2j macros, even by `\xintNthEltPy:f:csv`, with the exception of the first item, as the list is f-expanded. Perhaps `\xintNthEltPy:f:csv` should remove a leading space if present in the picked item; anyway, there are no spaces for the lists handled internally by the Python slicer of `xintexpr`, except the «nil» object currently represented by exactly one space.

Kept items (with no leading spaces; but first item special as it will have lost a leading space due to f-expansion) will lose a brace pair under `\xintKeep:f:csv` if the first argument was positive and strictly less than the length of the list. This differs of course from `\xintKeep` (which always braces items it outputs when used with positive first argument) and also from `\xintKeepUnbraced` in the case when the whole list is kept. Actually the case of singleton list is special, and brace removal will happen then.

This behaviour was otherwise for releases earlier than 1.2j and may change again.

Directly usable names are provided, but these macros (and the behaviour as described above) are to be considered *unstable* for the time being.

### 19.31.1. `\xintLength:f:csv`

Added at 1.2g (2016/03/19).

Modified at 1.2j (2016/12/22). Contrarily to `\xintLength` from [xintkernel](#) this one expands its argument.

```

1255 \def\xintLength:f:csv {\romannumeral0\xintlength:f:csv}%
1256 \def\xintlength:f:csv #1%
1257 {\long\def\xintlength:f:csv ##1{%
1258   \expandafter#1\the\numexpr\expandafter\XINT_length:f:csv_a
1259   \romannumeral`&&@##1\xint:,\xint:,\xint:,\xint:,%
1260   \xint:,\xint:,\xint:,\xint:,\xint:,%
1261   \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
1262   \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
1263   \relax
1264 }}\xintlength:f:csv { }%
    Must first check if empty list.
1265 \long\def\XINT_length:f:csv_a #1%
1266 {%
1267   \xint_gob_til_xint: #1\xint_c_\xint_bye\xint:%
1268   \XINT_length:f:csv_loop #1%
1269 }%
1270 \long\def\XINT_length:f:csv_loop #1,#2,#3,#4,#5,#6,#7,#8,#9,%
1271 {%
1272   \xint_gob_til_xint: #9\XINT_length:f:csv_finish\xint:%
1273   \xint_c_ix+\XINT_length:f:csv_loop
1274 }%
1275 \def\XINT_length:f:csv_finish\xint:\xint_c_ix+\XINT_length:f:csv_loop
1276   #1,#2,#3,#4,#5,#6,#7,#8,#9,{#9\xint_bye}%

```

### 19.31.2. `\xintLengthUpTo:f:csv`

Added at 1.2j (2016/12/22). `\added{1.2j}\xintLengthUpTo:f:csv{N}{comma-list}`. No ending comma. Returns -0 if length>N, else returns difference N-length. \*\*N must be non-negative!!\*\* Attention to the dot after `\xint_bye` for the loop interface.

```

1277 \def\xintLengthUpTo:f:csv {\romannumeral0\xintlengthupto:f:csv}%
1278 \long\def\xintlengthupto:f:csv #1#2%
1279 {%
1280   \expandafter\XINT_lengthupto:f:csv_a
1281   \the\numexpr#1\expandafter.%
1282   \romannumeral`&&@#2\xint:,\xint:,\xint:,\xint:,%
1283   \xint:,\xint:,\xint:,\xint:,%
1284   \xint_c_viii,\xint_c_vii,\xint_c_vi,\xint_c_v,%
1285   \xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye.%
1286 }%
    Must first recognize if empty list. If this is the case, return N.
1287 \long\def\XINT_lengthupto:f:csv_a #1.#2%
1288 {%
1289   \xint_gob_til_xint: #2\XINT_lengthupto:f:csv_empty\xint:%
1290   \XINT_lengthupto:f:csv_loop_b #1.#2%
1291 }%
1292 \def\XINT_lengthupto:f:csv_empty\xint:%
1293   \XINT_lengthupto:f:csv_loop_b #1.#2\xint_bye.{ #1}%
1294 \def\XINT_lengthupto:f:csv_loop_a #1%
1295 {%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1296 \xint_UDsignfork
1297 #1\XINT_lengthupto:f:csv_gt
1298 -\XINT_lengthupto:f:csv_loop_b
1299 \krof #1%
1300 }%
1301 \long\def\XINT_lengthupto:f:csv_gt #1\xint_bye.{-0}%
1302 \long\def\XINT_lengthupto:f:csv_loop_b #1.#2,#3,#4,#5,#6,#7,#8,#9,%
1303 {%
1304 \xint_gob_til_xint: #9\XINT_lengthupto:f:csv_finish_a\xint:%
1305 \expandafter\XINT_lengthupto:f:csv_loop_a\the\numexpr #1-\xint_c_viii.%
1306 }%
1307 \def\XINT_lengthupto:f:csv_finish_a\xint:
1308 \expandafter\XINT_lengthupto:f:csv_loop_a
1309 \the\numexpr #1-\xint_c_viii.#2,#3,#4,#5,#6,#7,#8,#9,%
1310 {%
1311 \expandafter\XINT_lengthupto:f:csv_finish_b\the\numexpr #1-#9\xint_bye
1312 }%
1313 \def\XINT_lengthupto:f:csv_finish_b #1#2.%
1314 {%
1315 \xint_UDsignfork
1316 #1{-0}%
1317 -{ #1#2}%
1318 \krof
1319 }%

```

### 19.31.3. \xintKeep:f:csv

Added at 1.2g (2016/03/19) [on 2016/03/17].

Modified at 1.2j (2016/12/22). Redone with use of `\xintLengthUpTo:f:csv`. Same code skeleton as `\xintKeep` but handling comma separated but non terminated lists has complications. The `\xintKeep` in case of a negative #1 uses `\xintgobble`, we don't have that for comma delimited items, hence we do a special loop here (this style of loop is surely competitive with `xintgobble` for a few dozens items and even more). The loop knows before starting that it will not go too far.

```

1320 \def\xintKeep:f:csv {\romannumeral0\xintkeep:f:csv }%
1321 \long\def\xintkeep:f:csv #1#2%
1322 {%
1323 \expandafter\xint_stop_aftergobble
1324 \romannumeral0\expandafter\XINT_keep:f:csv_a
1325 \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
1326 }%
1327 \def\XINT_keep:f:csv_a #1%
1328 {%
1329 \xint_UDzerominusfork
1330 #1-\XINT_keep:f:csv_keepnone
1331 0#1\XINT_keep:f:csv_neg
1332 0-{\XINT_keep:f:csv_pos #1}%
1333 \krof
1334 }%
1335 \long\def\XINT_keep:f:csv_keepnone .#1{,}%
1336 \long\def\XINT_keep:f:csv_neg #1.#2%
1337 {%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1338 \expandafter\XINT_keep:f:csv_neg_done\expandafter,%
1339 \romannumeral0%
1340 \expandafter\XINT_keep:f:csv_neg_a\the\numexpr
1341 #1-\numexpr\XINT_length:f:csv_a
1342 #2\xint:,\xint:,\xint:,\xint:,%
1343 \xint:,\xint:,\xint:,\xint:,\xint:,%
1344 \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
1345 \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
1346 .#2\xint_bye
1347 }%
1348 \def\XINT_keep:f:csv_neg_a #1%
1349 {%
1350 \xint_UDsignfork
1351 #1{\expandafter\XINT_keep:f:csv_trimloop\the\numexpr-\xint_c_ix+}%
1352 -\XINT_keep:f:csv_keepall
1353 \krof
1354 }%
1355 \def\XINT_keep:f:csv_keepall #1.{ }%
1356 \long\def\XINT_keep:f:csv_neg_done #1\xint_bye{#1}%
1357 \def\XINT_keep:f:csv_trimloop #1#2.%
1358 {%
1359 \xint_gob_til_minus#1\XINT_keep:f:csv_trimloop_finish-%
1360 \expandafter\XINT_keep:f:csv_trimloop
1361 \the\numexpr#1#2-\xint_c_ix\expandafter.\XINT_keep:f:csv_trimloop_trimnine
1362 }%
1363 \long\def\XINT_keep:f:csv_trimloop_trimnine #1,#2,#3,#4,#5,#6,#7,#8,#9,{}%
1364 \def\XINT_keep:f:csv_trimloop_finish-%
1365 \expandafter\XINT_keep:f:csv_trimloop
1366 \the\numexpr-#1-\xint_c_ix\expandafter.\XINT_keep:f:csv_trimloop_trimnine
1367 {\csname XINT_trim:f:csv_finish#1\endcsname}%
1368 \long\def\XINT_keep:f:csv_pos #1.#2%
1369 {%
1370 \expandafter\XINT_keep:f:csv_pos_fork
1371 \romannumeral0\XINT_lengthupto:f:csv_a
1372 #1.#2\xint:,\xint:,\xint:,\xint:,%
1373 \xint:,\xint:,\xint:,\xint:,%
1374 \xint_c_viii,\xint_c_vii,\xint_c_vi,\xint_c_v,%
1375 \xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye.%
1376 .#1.{ }#2\xint_bye%
1377 }%
1378 \def\XINT_keep:f:csv_pos_fork #1#2.%
1379 {%
1380 \xint_UDsignfork
1381 #1{\expandafter\XINT_keep:f:csv_loop\the\numexpr-\xint_c_viii+}%
1382 -\XINT_keep:f:csv_pos_keepall
1383 \krof
1384 }%
1385 \long\def\XINT_keep:f:csv_pos_keepall #1.#2#3\xint_bye{,#3}%
1386 \def\XINT_keep:f:csv_loop #1#2.%
1387 {%
1388 \xint_gob_til_minus#1\XINT_keep:f:csv_loop_end-%
1389 \expandafter\XINT_keep:f:csv_loop

```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1390 \the\numexpr#1#2-\xint_c_viii\expandafter.\XINT_keep:f:csv_loop_pickeight
1391 }%
1392 \long\def\XINT_keep:f:csv_loop_pickeight
1393 #1#2,#3,#4,#5,#6,#7,#8,#9,{\#1,#2,#3,#4,#5,#6,#7,#8,#9}%
1394 \def\XINT_keep:f:csv_loop_end-\expandafter\XINT_keep:f:csv_loop
1395 \the\numexpr-#1-\xint_c_viii\expandafter.\XINT_keep:f:csv_loop_pickeight
1396 {\csname XINT_keep:f:csv_end#1\endcsname}%
1397 \long\expandafter\def\csname XINT_keep:f:csv_end1\endcsname
1398 #1#2,#3,#4,#5,#6,#7,#8,#9\xint_bye {\#1,#2,#3,#4,#5,#6,#7,#8}%
1399 \long\expandafter\def\csname XINT_keep:f:csv_end2\endcsname
1400 #1#2,#3,#4,#5,#6,#7,#8\xint_bye {\#1,#2,#3,#4,#5,#6,#7}%
1401 \long\expandafter\def\csname XINT_keep:f:csv_end3\endcsname
1402 #1#2,#3,#4,#5,#6,#7\xint_bye {\#1,#2,#3,#4,#5,#6}%
1403 \long\expandafter\def\csname XINT_keep:f:csv_end4\endcsname
1404 #1#2,#3,#4,#5,#6\xint_bye {\#1,#2,#3,#4,#5}%
1405 \long\expandafter\def\csname XINT_keep:f:csv_end5\endcsname
1406 #1#2,#3,#4,#5\xint_bye {\#1,#2,#3,#4}%
1407 \long\expandafter\def\csname XINT_keep:f:csv_end6\endcsname
1408 #1#2,#3,#4\xint_bye {\#1,#2,#3}%
1409 \long\expandafter\def\csname XINT_keep:f:csv_end7\endcsname
1410 #1#2,#3\xint_bye {\#1,#2}%
1411 \long\expandafter\def\csname XINT_keep:f:csv_end8\endcsname
1412 #1#2\xint_bye {\#1}%
```

### 19.31.4. \xintTrim:f:csv

Added at 1.2g (2016/03/19) [on 2016/03/17].

Modified at 1.2j (2016/12/22). Redone on the basis of new \xintTrim.

```
1413 \def\xintTrim:f:csv {\romannumeral0\xinttrim:f:csv }%
1414 \long\def\xinttrim:f:csv #1#2%
1415 {%
1416 \expandafter\xint_stop_aftergobble
1417 \romannumeral0\expandafter\XINT_trim:f:csv_a
1418 \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
1419 }%
1420 \def\XINT_trim:f:csv_a #1%
1421 {%
1422 \xint_UDzerominusfork
1423 #1-\XINT_trim:f:csv_trimnone
1424 0#1\XINT_trim:f:csv_neg
1425 0-{\XINT_trim:f:csv_pos #1}%
1426 \krof
1427 }%
1428 \long\def\XINT_trim:f:csv_trimnone .#1{,#1}%
1429 \long\def\XINT_trim:f:csv_neg #1.#2%
1430 {%
1431 \expandafter\XINT_trim:f:csv_neg_a\the\numexpr
1432 #1-\numexpr\XINT_length:f:csv_a
1433 #2\xint:,\xint:,\xint:,\xint:,%
1434 \xint:,\xint:,\xint:,\xint:,\xint:,%
1435 \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
1436 \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1437 .{ }#2\xint_bye
1438 }%
1439 \def\xINT_trim:f:csv_neg_a #1%
1440 {%
1441     \xint_UDsignfork
1442     #1{\expandafter\xINT_keep:f:csv_loop\the\numexpr-\xint_c_viii+}%
1443     -\XINT_trim:f:csv_trimall
1444     \krof
1445 }%
1446 \def\xINT_trim:f:csv_trimall {\expandafter,\xint_bye}%
1447 \long\def\xINT_trim:f:csv_pos #1.#2%
1448 {%
1449     \expandafter\xINT_trim:f:csv_pos_done\expandafter,%
1450     \romannumeral0%
1451     \expandafter\xINT_trim:f:csv_loop\the\numexpr#1-\xint_c_ix.%
1452     #2\xint:,\xint:,\xint:,\xint:,\xint:,%
1453     \xint:,\xint:,\xint:,\xint:,\xint:\xint_bye
1454 }%
1455 \def\xINT_trim:f:csv_loop #1#2.%
1456 {%
1457     \xint_gob_til_minus#1\xINT_trim:f:csv_finish-%
1458     \expandafter\xINT_trim:f:csv_loop\the\numexpr#1#2\xINT_trim:f:csv_loop_trimnine
1459 }%
1460 \long\def\xINT_trim:f:csv_loop_trimnine #1,#2,#3,#4,#5,#6,#7,#8,#9,%
1461 {%
1462     \xint_gob_til_xint: #9\xINT_trim:f:csv_toofew\xint:-\xint_c_ix.%
1463 }%
1464 \def\xINT_trim:f:csv_toofew\xint:{*\xint_c_}%
1465 \def\xINT_trim:f:csv_finish-%
1466     \expandafter\xINT_trim:f:csv_loop\the\numexpr-#1\xINT_trim:f:csv_loop_trimnine
1467 {%
1468     \csname XINT_trim:f:csv_finish#1\endcsname
1469 }%
1470 \long\expandafter\def\csname XINT_trim:f:csv_finish1\endcsname
1471     #1,#2,#3,#4,#5,#6,#7,#8,{ }%
1472 \long\expandafter\def\csname XINT_trim:f:csv_finish2\endcsname
1473     #1,#2,#3,#4,#5,#6,#7,{ }%
1474 \long\expandafter\def\csname XINT_trim:f:csv_finish3\endcsname
1475     #1,#2,#3,#4,#5,#6,{ }%
1476 \long\expandafter\def\csname XINT_trim:f:csv_finish4\endcsname
1477     #1,#2,#3,#4,#5,{ }%
1478 \long\expandafter\def\csname XINT_trim:f:csv_finish5\endcsname
1479     #1,#2,#3,#4,{ }%
1480 \long\expandafter\def\csname XINT_trim:f:csv_finish6\endcsname
1481     #1,#2,#3,{ }%
1482 \long\expandafter\def\csname XINT_trim:f:csv_finish7\endcsname
1483     #1,#2,{ }%
1484 \long\expandafter\def\csname XINT_trim:f:csv_finish8\endcsname
1485     #1,{ }%
1486 \expandafter\let\csname XINT_trim:f:csv_finish9\endcsname\space
1487 \long\def\xINT_trim:f:csv_pos_done #1\xint:#2\xint_bye{#1}%

```

**19.31.5. \xintNthEltpy:f:csv**

Counts like Python starting at zero. Last refactored with 1.2j. Attention, makes currently no effort at removing leading spaces in the picked item.

```

1488 \def\xintNthEltpy:f:csv {\romannumeral0\xintntheltpy:f:csv }%
1489 \long\def\xintntheltpy:f:csv #1#2%
1490 {%
1491   \expandafter\xINT_nthelt:f:csv_a
1492   \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
1493 }%
1494 \def\xINT_nthelt:f:csv_a #1%
1495 {%
1496   \xint_UDsignfork
1497     #1\xINT_nthelt:f:csv_neg
1498     -\xINT_nthelt:f:csv_pos
1499   \krof #1%
1500 }%
1501 \long\def\xINT_nthelt:f:csv_neg -#1.#2%
1502 {%
1503   \expandafter\xINT_nthelt:f:csv_neg_fork
1504   \the\numexpr\xINT_length:f:csv_a
1505   #2\xint:,\xint:,\xint:,\xint:,%
1506   \xint:,\xint:,\xint:,\xint:,\xint:,%
1507   \xint_c_ix,\xint_c_viii,\xint_c_vii,\xint_c_vi,%
1508   \xint_c_v,\xint_c_iv,\xint_c_iii,\xint_c_ii,\xint_c_i,\xint_bye
1509   -#1.#2,\xint_bye
1510 }%
1511 \def\xINT_nthelt:f:csv_neg_fork #1%
1512 {%
1513   \if#1-\expandafter\xint_stop_afterbye\fi
1514   \expandafter\xINT_nthelt:f:csv_neg_done
1515   \romannumeral0%
1516   \expandafter\xINT_keep:f:csv_trimloop\the\numexpr-\xint_c_ix+#1%
1517 }%
1518 \long\def\xINT_nthelt:f:csv_neg_done#1,#2\xint_bye{ #1}%
1519 \long\def\xINT_nthelt:f:csv_pos #1.#2%
1520 {%
1521   \expandafter\xINT_nthelt:f:csv_pos_done
1522   \romannumeral0%
1523   \expandafter\xINT_trim:f:csv_loop\the\numexpr#1-\xint_c_ix.%
1524   #2\xint:,\xint:,\xint:,\xint:,\xint:,%
1525   \xint:,\xint:,\xint:,\xint:,\xint:,\xint_bye
1526 }%
1527 \def\xINT_nthelt:f:csv_pos_done #1{%
1528 \long\def\xINT_nthelt:f:csv_pos_done ##1,##2\xint_bye{%
1529   \xint_gob_til_xint:##1\xINT_nthelt:f:csv_pos_cleanup\xint:#1##1}%
1530 }\xINT_nthelt:f:csv_pos_done{ }%

```

This strange thing is in case the picked item was the last one, hence there was an ending `\xint:` (we could not put a comma earlier for matters of not confusing empty list with a singleton list), and we do this here to activate brace-stripping of item as all other items may be brace-stripped if picked. This is done for coherence. Of course, in the context of the `xintexpr.sty` parsers, there are no braces in list items...

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1531 \xint_firstofone{\long\def\XINT_nthelt:f:csv_pos_cleanup\xint:} %
1532 #1\xint:{ #1}%
```

### 19.31.6. \xintReverse:f:csv

**Added at 1.2g (2016/03/19) [on 2016/03/17].** Contrarily to [\xintReverseOrder](#) from [xintkernel.sty](#), this one expands its argument. Handles empty list too. .

**Modified at 1.2j (2016/12/22).** Made [\long](#).

```
1533 \def\xintReverse:f:csv {\romannumeral0\xintreverse:f:csv}%
1534 \long\def\xintreverse:f:csv #1%
1535 {%
1536   \expandafter\XINT_reverse:f:csv_loop
1537   \expandafter{\expandafter}\romannumeral`&&@#1,%
1538   \xint:,%
1539   \xint_bye,\xint_bye,\xint_bye,\xint_bye,%
1540   \xint_bye,\xint_bye,\xint_bye,\xint_bye,%
1541   \xint:
1542}%
1543 \long\def\XINT_reverse:f:csv_loop #1#2,#3,#4,#5,#6,#7,#8,#9,%
1544 {%
1545   \xint_bye #9\XINT_reverse:f:csv_cleanup\xint_bye
1546   \XINT_reverse:f:csv_loop {,#9,#8,#7,#6,#5,#4,#3,#2#1}%
1547}%
1548 \long\def\XINT_reverse:f:csv_cleanup\xint_bye\XINT_reverse:f:csv_loop #1#2\xint:
1549 {%
1550   \XINT_reverse:f:csv_finish #1%
1551}%
1552 \long\def\XINT_reverse:f:csv_finish #1\xint:,{ }%
```

### 19.31.7. \xintFirstItem:f:csv

**Added at 1.2k (2017/01/06).** For use by `first()` in [\xintexpr](#)-essions, and some amount of compatibility with [\xintNewExpr](#).

```
1553 \def\xintFirstItem:f:csv {\romannumeral0\xintfirstitem:f:csv}%
1554 \long\def\xintfirstitem:f:csv #1%
1555 {%
1556   \expandafter\XINT_first:f:csv_a\romannumeral`&&@#1,\xint_bye
1557}%
1558 \long\def\XINT_first:f:csv_a #1,#2\xint_bye{ #1}%
```

### 19.31.8. \xintLastItem:f:csv

**Added at 1.2k (2017/01/06).** Based on and sharing code with [xintkernel](#)'s [\xintLastItem](#) from 1.2i. Output empty if input empty. `f`-expands its argument (hence first item, if not protected.) For use by `last()` in [\xintexpr](#)-essions with to some extent [\xintNewExpr](#) compatibility.

```
1559 \def\xintLastItem:f:csv {\romannumeral0\xintlastitem:f:csv}%
1560 \long\def\xintlastitem:f:csv #1%
1561 {%
1562   \expandafter\XINT_last:f:csv_loop\expandafter{\expandafter}\expandafter.%
1563   \romannumeral`&&@#1,%
1564   \xint:\XINT_last_loop_enda,\xint:\XINT_last_loop_endb,%
1565   \xint:\XINT_last_loop_endc,\xint:\XINT_last_loop_endd,%
```

```

1566 \xint:\XINT_last_loop_ende,\xint:\XINT_last_loop_endf,%
1567 \xint:\XINT_last_loop_endg,\xint:\XINT_last_loop_endh,\xint_bye
1568 }%
1569 \long\def\XINT_last:f:csv_loop #1.#2,#3,#4,#5,#6,#7,#8,#9,%
1570 {%
1571 \xint_gob_til_xint: #9%
1572 {#8}{#7}{#6}{#5}{#4}{#3}{#2}{#1}\xint:
1573 \XINT_last:f:csv_loop {#9}%.%
1574 }%

```

### 19.31.9. \xintKeep:x:csv

Added at 1.2j (2016/12/22). To [xintexpr](#). Moved here at 1.4. Not part of publicly supported macros, may be removed at any time.

```

1575 \def\xintKeep:x:csv #1#2%
1576 {%
1577 \expandafter\xint_gobble_i
1578 \romannumeral0\expandafter\XINT_keep:x:csv_pos
1579 \the\numexpr #1\expandafter.\expandafter{\romannumeral`&&@#2}%
1580 }%
1581 \def\XINT_keep:x:csv_pos #1.#2%
1582 {%
1583 \expandafter\XINT_keep:x:csv_loop\the\numexpr#1-\xint_c_viii.%
1584 #2\xint_Bye,\xint_Bye,\xint_Bye,\xint_Bye,%
1585 \xint_Bye,\xint_Bye,\xint_Bye,\xint_Bye,\xint_bye
1586 }%
1587 \def\XINT_keep:x:csv_loop #1%
1588 {%
1589 \xint_gob_til_minus#1\XINT_keep:x:csv_finish-%
1590 \XINT_keep:x:csv_loop_pickeight #1%
1591 }%
1592 \def\XINT_keep:x:csv_loop_pickeight #1.#2,#3,#4,#5,#6,#7,#8,#9,%
1593 {%
1594 ,#2,#3,#4,#5,#6,#7,#8,#9%
1595 \expandafter\XINT_keep:x:csv_loop\the\numexpr#1-\xint_c_viii.%
1596 }%
1597 \def\XINT_keep:x:csv_finish-\XINT_keep:x:csv_loop_pickeight -#1.%
1598 {%
1599 \csname XINT_keep:x:csv_finish#1\endcsname
1600 }%
1601 \expandafter\def\csname XINT_keep:x:csv_finish1\endcsname
1602 #1,#2,#3,#4,#5,#6,#7,{,#1,#2,#3,#4,#5,#6,#7\xint_Bye}%
1603 \expandafter\def\csname XINT_keep:x:csv_finish2\endcsname
1604 #1,#2,#3,#4,#5,#6,{,#1,#2,#3,#4,#5,#6\xint_Bye}%
1605 \expandafter\def\csname XINT_keep:x:csv_finish3\endcsname
1606 #1,#2,#3,#4,#5,{,#1,#2,#3,#4,#5\xint_Bye}%
1607 \expandafter\def\csname XINT_keep:x:csv_finish4\endcsname
1608 #1,#2,#3,#4,{,#1,#2,#3,#4\xint_Bye}%
1609 \expandafter\def\csname XINT_keep:x:csv_finish5\endcsname
1610 #1,#2,#3,{,#1,#2,#3\xint_Bye}%
1611 \expandafter\def\csname XINT_keep:x:csv_finish6\endcsname
1612 #1,#2,{,#1,#2\xint_Bye}%

```

```

1613 \expandafter\def\csname XINT_keep:x:csv_finish7\endcsname
1614   #1,{, #1\xint_Bye}%
1615 \expandafter\let\csname XINT_keep:x:csv_finish8\endcsname\xint_Bye

```

#### 19.31.10. Public names for the undocumented csv macros: `\xintCSVLength`, `\xintCSVKeep`, `\xintCSVKeepx`, `\xintCSVTrim`, `\xintCSVNthEltPy`, `\xintCSVReverse`, `\xintCSVFirstItem`, `\xintCSVLastItem`

Completely unstable macros: currently they expand the list argument and want no final comma. But for matters of `xintexpr.sty` I could as well decide to require a final comma, and then I could simplify implementation but of course this would break the macros if used with current functionalities.

```

1616 \let\xintCSVLength \xintLength:f:csv
1617 \let\xintCSVKeep \xintKeep:f:csv
1618 \let\xintCSVKeepx \xintKeep:x:csv
1619 \let\xintCSVTrim \xintTrim:f:csv
1620 \let\xintCSVNthEltPy \xintNthEltPy:f:csv
1621 \let\xintCSVReverse \xintReverse:f:csv
1622 \let\xintCSVFirstItem\xintFirstItem:f:csv
1623 \let\xintCSVLastItem \xintLastItem:f:csv
1624 \let\XINT_tmpa\relax \let\XINT_tmpb\relax \let\XINT_tmpc\relax
1625 \XINTrestorecatcodesendinginput%

```

## 20. Package [xintcore](#) implementation

.1	Catcodes, $\varepsilon$ -T <sub>E</sub> X and reload detection . . .	302	.25	\XINT_sepbyviii_Z . . . . .	315
.2	Package identification . . . . .	303	.26	\XINT_sepbyviii_andcount . . . . .	316
.3	(WIP!) Error conditions and exceptions . . .	303	.27	\XINT_rsepyviii . . . . .	316
	Routines handling integers as lists of token digits	306	.28	\XINT_sepandrev . . . . .	317
.4	\XINT_cuz_small . . . . .	306	.29	\XINT_sepandrev_andcount . . . . .	317
.5	\xintNum, \xintiNum . . . . .	306	.30	\XINT_rev_nounsep . . . . .	318
.6	\xintiiSgn . . . . .	307	.31	\XINT_unrevbyviii . . . . .	318
.7	\xintiiOpp . . . . .	308		Core arithmetic . . . . .	319
.8	\xintiiAbs . . . . .	308	.32	\xintiiAdd . . . . .	319
.9	\xintFDg . . . . .	308	.33	\xintiiCmp . . . . .	322
.10	\xintLDg . . . . .	309	.34	\xintiiSub . . . . .	324
.11	\xintDouble . . . . .	309	.35	\xintiiMul . . . . .	330
.12	\xintHalf . . . . .	310	.36	\xintiiDivision . . . . .	333
.13	\xintInc . . . . .	310		Derived arithmetic . . . . .	348
.14	\xintDec . . . . .	311	.37	\xintiiQuo, \xintiiRem . . . . .	348
.15	\xintDSL . . . . .	311	.38	\xintiiDivRound . . . . .	348
.16	\xintDSR . . . . .	312	.39	\xintiiDivTrunc . . . . .	349
.17	\xintDSRr . . . . .	312	.40	\xintiiModTrunc . . . . .	349
	Blocks of eight digits . . . . .	313	.41	\xintiiDivMod . . . . .	350
.18	\XINT_cuz . . . . .	313	.42	\xintiiDivFloor . . . . .	351
.19	\XINT_cuz_byviii . . . . .	313	.43	\xintiiMod . . . . .	351
.20	\XINT_unsep_loop . . . . .	313	.44	\xintiiSqr . . . . .	351
.21	\XINT_unsep_cuzsmall . . . . .	314	.45	\xintiiPow . . . . .	352
.22	\XINT_div_unsepQ . . . . .	314	.46	\xintiiFac . . . . .	355
.23	\XINT_div_unsepR . . . . .	315	.47	\XINT_useiimessage . . . . .	358
.24	\XINT_zeroes_forviii . . . . .	315			

Got split off from [xint](#) with release 1.1.

The core arithmetic routines have been entirely rewritten for release 1.2. The 1.2i and 1.2i brought again some improvements.

The commenting continues (2025/09/06) to be very sparse: actually it got worse than ever with release 1.2. I will possibly add comments at a later date, but for the time being the new routines are not commented at all.

1.3 removes all macros which were deprecated at 1.2o.

### 20.1. Catcodes, $\varepsilon$ -T<sub>E</sub>X and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^
11 \def\empty{}\def\space{ }\newlinechar10

```

```

12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintcore.sty\endcsname
14 \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname
15 \expandafter\ifx\csname numexpr\endcsname\relax
16   \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17     \immediate\write128{^^JPackage xintcore Warning:^^J%
18                           \space\space\space\space
19                           \numexpr not available, aborting input.^^J}%
20   \else
21     \PackageWarningNoLine{xintcore}{\numexpr not available, aborting input}%
22   \fi
23 \def\z{\endgroup\endinput}%
24 \else
25   \ifx\x\relax % plain-TeX, first loading of xintcore.sty
26     \ifx\w\relax % but xintkernel.sty not yet loaded.
27       \def\z{\endgroup\input xintkernel.sty\relax}%
28     \fi
29   \else
30     \ifx\x\empty % LaTeX, first loading,
31     % variable is initialized, but \ProvidesPackage not yet seen
32     \ifx\w\relax % xintkernel.sty not yet loaded.
33       \def\z{\endgroup\RequirePackage{xintkernel}}%
34     \fi
35   \else
36     \def\z{\endgroup\endinput}% xintkernel already loaded.
37   \fi
38 \fi
39 \fi
40 \z%
41 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 20.2. Package identification

```

42 \XINT_providespackage
43 \ProvidesPackage{xintcore}%
44 [2025/09/06 v1.4o Expandable arithmetic on big integers (JFB)]%

```

## 20.3. (WIP!) Error conditions and exceptions

As per the Mike Cowlishaw/IBM's General Decimal Arithmetic Specification

<http://speleotrove.com/decimal/decarith.html>

and the Python3 implementation in its Decimal module.

Clamped, ConversionSyntax, DivisionByZero, DivisionImpossible, DivisionUndefined, Inexact, InsufficientStorage, InvalidContext, InvalidOperation, Overflow, Inexact, Rounded, Subnormal, Underflow.

X3.274 rajoute LostDigits

Python rajoute FloatOperation (et n'inclut pas InsufficientStorage)

quote de decarith.pdf: The Clamped, Inexact, Rounded, and Subnormal conditions can coincide with each other or with other conditions. In these cases then any trap enabled for another condition takes precedence over (is handled before) all of these, any Subnormal trap takes precedence over Inexact, any Inexact trap takes precedence over Rounded, and any Rounded trap takes precedence over Clamped.

WORK IN PROGRESS ! (1.21, 2017/07/26)



I follow the Python terminology: a trapped signal means it raises an exception which for us means an expandable error message with some possible user interaction. In this WIP state, the interaction is commented out. A non-trapped signal or condition would activate a (presumably silent) handler.

Here, no signal-raising condition is "ignored" and all are "trapped" which means that error handlers are never activated, thus left in garbage state in the code.

Various conditions can raise the same signal.

Only signals, not conditions, raise Flags.

If a signal is ignored it does not raise a Flag, but it activates the signal handler (by default now no signal is ignored.)

If a signal is not ignored it raises a Flag and then if it is not trapped it activates the handler of the `_condition_`.

If trapped (which is default now) an «exception» is raised, which means an expandable error message (I copied over the LaTeX3 code for expandable error messages, basically) interrupts the TeX run. In future, user input could be solicited, but currently this is commented out.

For now macros to reset flags are done but without public interface nor documentation.

Only four conditions are currently possibly encountered:

- InvalidOperation
- DivisionByZero
- DivisionUndefined (which signals InvalidOperation)
- Underflow

I did it quickly, anyhow this will become more palpable when some of the Decimal Specification is actually implemented. The plan is to first do the X3.274 norm, then more complete implementation will follow... perhaps...

```

45 \csname XINT_Clamped_istrapped\endcsname
46 \csname XINT_ConversionSyntax_istrapped\endcsname
47 \csname XINT_DivisionByZero_istrapped\endcsname
48 \csname XINT_DivisionImpossible_istrapped\endcsname
49 \csname XINT_DivisionUndefined_istrapped\endcsname
50 \csname XINT_InvalidOperation_istrapped\endcsname
51 \csname XINT_Overflow_istrapped\endcsname
52 \csname XINT_Underflow_istrapped\endcsname
53 \catcode`- 11
54 \def\XINT_ConversionSyntax-signal {{InvalidOperation}}%
55 \let\XINT_DivisionImpossible-signal\XINT_ConversionSyntax-signal
56 \let\XINT_DivisionUndefined-signal \XINT_ConversionSyntax-signal
57 \let\XINT_InvalidContext-signal \XINT_ConversionSyntax-signal
58 \catcode`- 12
59 \def\XINT_signalcondition #1{\expandafter\XINT_signalcondition_a
60   \romannumeral0\ifcsname XINT_#1-signal\endcsname
61     \xint_dothis{\csname XINT_#1-signal\endcsname}%
62     \fi\xint_orthat{{#1}}{{#1}}%
63 \def\XINT_signalcondition_a #1#2#3#4#5{% copied over from Python Decimal module
64   #1=signal, #2=condition, #3=explanation for user, #4=context for error handlers, #5=used.
65   \xint_dothis{\csname XINT_#1.handler\endcsname {#4}}%
66   \fi
67   \expandafter\xint_gobble_i\csname XINT_#1Flag_ON\endcsname
68   \unless\ifcsname XINT_#1_istrapped\endcsname
69     \xint_dothis{\csname XINT_#2.handler\endcsname {#4}}%
70   \fi
71   \xint_orthat{%

```

```
72      % the flag raised is named after the signal #1, but we show condition
73      % #2
```

On 2021/05/19, 1.4g, I re-examined `\XINT_expandableerror` experimenting at first with an added `^^J` to shift to next line the actual message.

Previously I was calling it thrice (condition #2, user context #3, next tokens #5) here but it seems more reasonable to use it only once. As total size is so limited, I decided to only display #3 (information for user) and drop the #2 (condition, first argument of `\XINT_signalcondition`) and the display of the #5 (next tokens, fourth argument of `\XINT_signalcondition`).

Besides, why was I doing here `\xint_stop_atfirstofone{#5}`, which adds limitations to usage? Now inserting #5 directly so callers will have to insert a `\romannumeral0` stopping space token if needed. I thus have to update all usages across (mainly, I think) `xintfrac`. Done, but using here `\xint_firstofone{#5}`. This looks silly, but allows some hypothetical future usage by user of `\xintUse{stuff}` usage where `\xintUse` would be `\xint_firstofthree`.

The problem is that this would have to be explained to user in the error context but space there is so extremely limited...

After having reviewed existing usage of `\XINT_signalcondition`, I noticed there was free space in most cases and added here " (hit RET)" after #3.

I experimented with `^^J` here too (its effect in the "context" is independent of the `\newlinechar` setting, but it depends on the engine: works with TeXLive pdftex, requires -8bit with xetex)

However, due to `\errorcontextlines` being 5 by default in etex (but `xintsession 0.2b` sets it to 0), I finally decided to not insert a `^^J` (&&J) at all to separate the " (hit RET)" hint.

On 2021/05/20 evening I found another completely different method for `\XINT_expandableerror`, which has some advantages. In particular it allows me to not use here "#3 (hit RET)" but simply "#3" as such information can be integrated in a non size limited generic message.

The maximal size of #3 here was increased from 48 characters (method with `\xint/` being badly delimited), to now 55 characters, longer messages being truncated at 56 characters with an appended `"\ETC."`.

```
74      \XINT_expandableerror{#3}%
75      % not for X3.274
76      % \XINT_expandableerror{<RET>, or I\xintUse{...}<RET>, or I\xintCTRLC<RET>}%
77      \xint_firstofone{#5}%
78  }%
79 }%
80 %% \def\xintUse{\xint_firstofthree} % defined in xint.sty
81 \def\XINT_ifFlagRaised #1{%
82   \ifcsname XINT_#1Flag_ON\endcsname
83     \expandafter\xint_firstoftwo
84   \else
85     \expandafter\xint_secondoftwo
86   \fi}%
87 \def\XINT_resetFlag #1%
88   {\expandafter\let\csname XINT_#1Flag_ON\endcsname\XINT_undefined}%
89 \def\XINT_resetFlags {% WIP
90   \XINT_resetFlag{InvalidOperation}% also from DivisionUndefined
91   \XINT_resetFlag{DivisionByZero}%
92   \XINT_resetFlag{Underflow}% (\xintiiPow with negative exponent)
93   \XINT_resetFlag{Overflow}% not encountered so far in xint code 1.21
94   % .. others ..
95 }%
96 \def\XINT_RaiseFlag #1{\expandafter\xint_gobble_i\csname XINT_#1Flag_ON\endcsname}%
97   NOT IMPLEMENTED! WORK IN PROGRESS! (ALL SIGNALS TRAPPED, NO HANDLERS USED)
98 \catcode`. 11
```

## Routines handling integers as lists of token digits

These routines or their sub-routines are mainly for internal usage.

`\XINT_cuz_small` removes leading zeroes from the first eight digits. Expands following `\romannumeral0`. At least one digit is produced.

## 20.5. \xintNum, \xintiNum

Very old routine got completely rewritten at 1.21.

`\xintiNum{\the\numexpr 1}` was ok, but a bit luckily so.

```

116 \def\xintiNum {\romannumeral0\xintinum }%
117 \def\xintinum #1%
118 {%
119     \expandafter\XINT_num_cleanup\the\numexpr\expandafter\XINT_num_loop
120     \romannumeral`&&@#1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
121 }%
122 \def\xintNum {\romannumeral0\xintnum }%
123 \let\xintnum\xintinum

```

Attention `\xintnum` (hence `\xintNum`) gets redefined by `xintfrac`. Click on names to see the redefinition there.

```

124 \def\xINT_num #1%
125 {%
126   \expandafter\xINT_num_cleanup\the\numexpr\xINT_num_loop
127   #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
128 }%
129 \def\xINT_num_loop #1#2#3#4#5#6#7#8#9%
130 {%
131   \xint_gob_til_xint: #9\xINT_num_end\xint:
132   #1#2#3#4#5#6#7#8#9%
133   \ifnum \numexpr #1#2#3#4#5#6#7#8#9+\xint_c_ = \xint_c_

```

means that so far only signs encountered, (if syntax is legal) then possibly zeroes or a terminated or not terminated `\numexpr` evaluating to zero In that latter case a correct zero will be produced in the end.

```

134   \expandafter\xINT_num_loop
135   \else
136   non terminated \numexpr (with nine tokens total) are safe as after \fi, there is then \xint:
137   \expandafter\relax
138   \fi
139 }%
140 \def\xINT_num_end\xint:#1\xint:{#1+\xint_c_\xint:}% empty input ok
141 \def\xINT_num_cleanup #1\xint:#2\Z { #1}%

```

## 20.6. `\xintiisgn`

1.21 made `\xintiisgn` robust against non terminated input.

1.2o deprecates here `\xintSgn` (it requires `xintfrac.sty`).

```

141 \def\xintiisgn {\romannumeral0\xintiisgn }%
142 \def\xintiisgn #1%
143 {%
144   \expandafter\xINT_sgn \romannumeral`&&@#1\xint:
145 }%
146 \def\xINT_sgn #1#2\xint:
147 {%
148   \xint_UDzerominusfork
149   #1-{ 0}%
150   0#1{-1}%
151   0-{ 1}%
152   \krof
153 }%
154 \def\xINT_Sgn #1#2\xint:
155 {%
156   \xint_UDzerominusfork
157   #1-{0}%
158   0#1{-1}%
159   0-{1}%
160   \krof
161 }%
162 \def\xINT_cntSgn #1#2\xint:
163 {%

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
164 \xint_UDzerominusfork
165 #1-\xint_c_
166 0#1\xint_c_mone
167 0-\xint_c_i
168 \krof
169 }%
```

### 20.7. \xintiiOpp

Attention, `\xintiiOpp` non robust against non terminated inputs. Reason is I don't want to have to grab a delimiter at the end, as everything happens "upfront".

```
170 \def\xintiiOpp {\romannumeral0\xintiiopp }%
171 \def\xintiiopp #1%
172 {%
173 \expandafter\xINT_opp \romannumeral`&&@#1%
174 }%
175 \def\xINT_opp #1{\romannumeral0\xINT_opp #1}%
176 \def\xINT_opp #1%
177 {%
178 \xint_UDzerominusfork
179 #1-{ 0}% zero
180 0#1{ }% negative
181 0-{ -#1}% positive
182 \krof
183 }%
```

### 20.8. \xintiiAbs

Attention `\xintiiAbs` non robust against non terminated input.

```
184 \def\xintiiAbs {\romannumeral0\xintiiabs }%
185 \def\xintiiabs #1%
186 {%
187 \expandafter\xINT_abs \romannumeral`&&@#1%
188 }%
189 \def\xINT_abs #1%
190 {%
191 \xint_UDsignfork
192 #1{ }%
193 -{ #1}%
194 \krof
195 }%
196 \def\xINT_Abs #1%
197 {%
198 \xint_UDsignfork
199 #1{}%
200 -{#1}%
201 \krof
202 }%
```

### 20.9. \xintFDg

FIRST DIGIT.

```

203 \def\xintFDg {\romannumeral0\xintfdg }%
204 \def\xintfdg #1{\expandafter\XINT_fdg \romannumeral`&&@#1\xint:\Z}%
205 \def\XINT_FDg #1%
206   {\romannumeral0\expandafter\XINT_fdg\romannumeral`&&@\xintnum{#1}\xint:\Z }%
207 \def\XINT_fdg #1#2#3\Z
208 {%
209   \xint_UDzerominusfork
210     #1-{ 0}%    zero
211     0#1{ #2}%   negative
212     0-{ #1}%    positive
213   \krof
214 }%

```

LAST DIGIT.

Attention \xintLDg non robust against non terminated input.

## 20.11. \xintDouble

309

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
241     #1\XINT_dbl_neg
242     -\XINT_dbl
243     \krof #1%
244 }%
245 \def\XINT_dbl_neg-{\expandafter-\romannumeral0\XINT_dbl}%
246 \def\XINT_dbl #1{%
247 \def\XINT_dbl ##1##2##3##4##5##6##7##8%
248     {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8\XINT_dbl_a}%
249 }\XINT_dbl{ }%
250 \def\XINT_dbl_a #1#2#3#4#5#6#7#8%
251     {\expandafter\XINT_dbl_e\the\numexpr 1#1#2#3#4#5#6#7#8\XINT_dbl_a}%
252 \def\XINT_dbl_e#1{* \xint_c_ii\if#13+\xint_c_i\fi\relax}%
```

## 20.12. \xintHalf

Attention \xintHalf non robust against non terminated input.

```
253 \def\xintHalf {\romannumeral0\xinthalf}%
254 \def\xinthalf #1{\expandafter\XINT_half_fork\romannumeral`&&@#1%
255     \xint_bye\xint_Bye345678\xint_bye
256     * \xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax}%
257 \def\XINT_half_fork #1%
258 {%
259     \xint_UDsignfork
260     #1\XINT_half_neg
261     -\XINT_half
262     \krof #1%
263 }%
264 \def\XINT_half_neg-{\xintiioopp\XINT_half}%
265 \def\XINT_half #1{%
266 \def\XINT_half ##1##2##3##4##5##6##7##8%
267     {\expandafter#1\the\numexpr(##1##2##3##4##5##6##7##8\XINT_half_a}%
268 }\XINT_half{ }%
269 \def\XINT_half_a#1{\xint_Bye#1\xint_bye\XINT_half_b#1}%
270 \def\XINT_half_b #1#2#3#4#5#6#7#8%
271     {\expandafter\XINT_half_e\the\numexpr(1#1#2#3#4#5#6#7#8\XINT_half_a}%
272 \def\XINT_half_e#1{* \xint_c_v+#1-\xint_c_v)\relax}%
```

## 20.13. \xintInc

1.2i much delayed complete rewrite in 1.2 style.

As we take 9 by 9 with the input save stack at 5000 this allows a bit less than 9 times 2500 = 22500 digits on input.

Attention \xintInc non robust against non terminated input.

```
273 \def\xintInc {\romannumeral0\xintinc}%
274 \def\xintinc #1{\expandafter\XINT_inc_fork\romannumeral`&&@#1%
275     \xint_bye23456789\xint_bye+\xint_c_i\relax}%
276 \def\XINT_inc_fork #1%
277 {%
278     \xint_UDsignfork
279     #1\XINT_inc_neg
280     -\XINT_inc
281     \krof #1%
```

```

282 }%
283 \def\XINT_inc_neg-#1\xint_bye#2\relax
284   {\xintiopp\XINT_dec #1\XINT_dec_bye234567890\xint_bye}%
285 \def\XINT_inc #1{%
286 \def\XINT_inc ##1##2##3##4##5##6##7##8##9%
287   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_inc_a}%
288 }\XINT_inc{ }%
289 \def\XINT_inc_a #1#2#3#4#5#6#7#8#9%
290   {\expandafter\XINT_inc_e\the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_inc_a}%
291 \def\XINT_inc_e#1{\if#12+\xint_c_i\fi\relax}%

```

## 20.14. \xintDec

1.2i much delayed complete rewrite in the 1.2 style. Things are a bit more complicated than `\xintInc` because 2999999999 is too big for TeX.

Attention `\xintDec` non robust against non terminated input.

```

292 \def\xintDec {\romannumeral0\xintdec}%
293 \def\xintdec #1{\expandafter\XINT_dec_fork\romannumeral`&&@#1%
294   \XINT_dec_bye234567890\xint_bye}%
295 \def\XINT_dec_fork #1%
296 {%
297   \xint_UDsignfork
298   #1\XINT_dec_neg
299   -\XINT_dec
300   \krof #1%
301 }%
302 \def\XINT_dec_neg-#1\XINT_dec_bye#2\xint_bye
303   {\expandafter-%
304   \romannumeral0\XINT_inc #1\xint_bye23456789\xint_bye+\xint_c_i\relax}%
305 \def\XINT_dec #1{%
306 \def\XINT_dec ##1##2##3##4##5##6##7##8##9%
307   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_dec_a}%
308 }\XINT_dec{ }%
309 \def\XINT_dec_a #1#2#3#4#5#6#7#8#9%
310   {\expandafter\XINT_dec_e\the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_dec_a}%
311 \def\XINT_dec_bye #1\XINT_dec_a#2#3\xint_bye
312   {\if#20-\xint_c_ii\relax+\else-\fi\xint_c_i\relax}%
313 \def\XINT_dec_e#1{\unless\if#11\xint_dothis{-\xint_c_i#1}\fi\xint_orthat\relax}%

```

## 20.15. \xintDSL

DECIMAL SHIFT LEFT (=MULTIPLICATION PAR 10). Rewritten for 1.2i. This was very old code... I never came back to it, but I should have rewritten it long time ago.

Attention `\xintDSL` non robust against non terminated input.

```

314 \def\xintDSL {\romannumeral0\xintdsl }%
315 \def\xintdsl #1{\expandafter\XINT_dsl\romannumeral`&&@#10}%
316 \def\XINT_dsl#1{%
317 \def\XINT_dsl ##1{\xint_gob_til_zero ##1\xint_dsl_zero 0#1##1}%
318 }\XINT_dsl{ }%
319 \def\xint_dsl_zero 0 0{ }%

```



## 20.16. \xintDSR

Decimal shift right, truncates towards zero. Rewritten for 1.2i. Limited to 22483 digits on input.

Attention `\xintDSR` non robust against non terminated input.

```

320 \def\xintDSR{\romannumeral0\xintdsr}%
321 \def\xintdsr #1{\expandafter\XINT_dsr_fork\romannumeral`&&@#1%
322   \xint_bye\xint_Bye3456789\xint_bye+\xint_c_v)/\xint_c_x-\xint_c_i\relax}%
323 \def\XINT_dsr_fork #1%
324 {%
325   \xint_UDsignfork
326   #1\XINT_dsr_neg
327   -\XINT_dsr
328   \krof #1%
329 }%
330 \def\XINT_dsr_neg-{\xintiopp\XINT_dsr}%
331 \def\XINT_dsr #1{%
332 \def\XINT_dsr ##1##2##3##4##5##6##7##8##9%
333   {\expandafter#1\the\numexpr(##1##2##3##4##5##6##7##8##9\XINT_dsr_a}%
334 }\XINT_dsr{ }%
335 \def\XINT_dsr_a#1{\xint_Bye#1\xint_bye\XINT_dsr_b#1}%
336 \def\XINT_dsr_b #1#2#3#4#5#6#7#8#9%
337   {\expandafter\XINT_dsr_e\the\numexpr(1#1#2#3#4#5#6#7#8#9\XINT_dsr_a}%
338 \def\XINT_dsr_e #1{\relax}%

```

## 20.17. \xintDSRr

New with 1.2i. Decimal shift right, rounds away from zero; done in the 1.2 spirit (with much delay, sorry). Used by `\xintRound`, `\xintDivRound`.

This is about the first time I am happy that the division in `\numexpr` rounds!

Attention `\xintDSRr` non robust against non terminated input.

```

339 \def\xintDSRr{\romannumeral0\xintdsrr}%
340 \def\xintdsrr #1{\expandafter\XINT_dsrr_fork\romannumeral`&&@#1%
341   \xint_bye\xint_Bye3456789\xint_bye+\xint_c_x\relax}%
342 \def\XINT_dsrr_fork #1%
343 {%
344   \xint_UDsignfork
345   #1\XINT_dsrr_neg
346   -\XINT_dsrr
347   \krof #1%
348 }%
349 \def\XINT_dsrr_neg-{\xintiopp\XINT_dsrr}%
350 \def\XINT_dsrr #1{%
351 \def\XINT_dsrr ##1##2##3##4##5##6##7##8##9%
352   {\expandafter#1\the\numexpr##1##2##3##4##5##6##7##8##9\XINT_dsrr_a}%
353 }\XINT_dsrr{ }%
354 \def\XINT_dsrr_a#1{\xint_Bye#1\xint_bye\XINT_dsrr_b#1}%
355 \def\XINT_dsrr_b #1#2#3#4#5#6#7#8#9%
356   {\expandafter\XINT_dsrr_e\the\numexpr1#1#2#3#4#5#6#7#8#9\XINT_dsrr_a}%
357 \let\XINT_dsrr_e\XINT_inc_e

```

## Blocks of eight digits

The lingua of release 1.2.

### 20.18. \XINT\_cuz

This (launched by `\romannumeral0`) iterately removes all leading zeroes from a sequence of 8N digits ended by `\R`.

Rewritten for 1.21, now uses `\numexpr` governed expansion and `\ifnum` test rather than delimited gobbling macros.

Note 2015/11/28: with only four digits the `gob_til_fourzeroes` had proved in some old testing faster than `\ifnum` test. But with eight digits, the execution times are much closer, as I tested back then.

```

358 \def\XINT_cuz #1{%
359 \def\XINT_cuz {\expandafter#1\the\numexpr\XINT_cuz_loop}%
360 }\XINT_cuz{ }%
361 \def\XINT_cuz_loop #1#2#3#4#5#6#7#8#9%
362 {%
363     #1#2#3#4#5#6#7#8%
364     \xint_gob_til_R #9\XINT_cuz_hitend\R
365     \ifnum #1#2#3#4#5#6#7#8>\xint_c_
366         \expandafter\XINT_cuz_cleantoend
367     \else\expandafter\XINT_cuz_loop
368     \fi #9%
369 }%
370 \def\XINT_cuz_hitend\R #1\R{\relax}%
371 \def\XINT_cuz_cleantoend #1\R{\relax #1}%

```

### 20.19. \XINT\_cuz\_byviii

This removes eight by eight leading zeroes from a sequence of 8N digits ended by `\R`. Thus, we still have 8N digits on output. Expansion started by `\romannumeral0`

```

372 \def\XINT_cuz_byviii #1#2#3#4#5#6#7#8#9%
373 {%
374     \xint_gob_til_R #9\XINT_cuz_byviii_e \R
375     \xint_gob_til_eightzeroes #1#2#3#4#5#6#7#8\XINT_cuz_byviii_z 00000000%
376     \XINT_cuz_byviii_done #1#2#3#4#5#6#7#8#9%
377 }%
378 \def\XINT_cuz_byviii_z 00000000\XINT_cuz_byviii_done 00000000{\XINT_cuz_byviii}%
379 \def\XINT_cuz_byviii_done #1\R { #1}%
380 \def\XINT_cuz_byviii_e\R #1\XINT_cuz_byviii_done #2\R{ #2}%

```

### 20.20. \XINT\_unsep\_loop

This is used as

```

\the\numexpr0\XINT_unsep_loop (blocks of 1<8digits>!)
\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax

```

It removes the 1's and !'s, and outputs the 8N digits with a 0 token as a prefix which will have to be cleaned out by caller.

Actually it does not matter whether the blocks contain really 8 digits, all that matters is that they have 1 as first digit (and at most 9 digits after that to obey the TeX-`\numexpr` bound).

Done at 1.21 for usage by other macros. The similar code in earlier releases was strangely in  $O(N^2)$  style, apparently to avoid some memory constraints. But these memory constraints related

to `\numexpr` chaining seems to be in many places in xint code base. The 1.2l version is written in the 1.2i style of `\xintInc` etc... and is compatible with some 1! block without digits among the treated blocks, they will disappear.

```

381 \def\xINT_unsep_loop #1!#2!#3!#4!#5!#6!#7!#8!#9!%
382 {%
383   \expandafter\xINT_unsep_clean
384   \the\numexpr #1\expandafter\xINT_unsep_clean
385   \the\numexpr #2\expandafter\xINT_unsep_clean
386   \the\numexpr #3\expandafter\xINT_unsep_clean
387   \the\numexpr #4\expandafter\xINT_unsep_clean
388   \the\numexpr #5\expandafter\xINT_unsep_clean
389   \the\numexpr #6\expandafter\xINT_unsep_clean
390   \the\numexpr #7\expandafter\xINT_unsep_clean
391   \the\numexpr #8\expandafter\xINT_unsep_clean
392   \the\numexpr #9\xINT_unsep_loop
393 }%
394 \def\xINT_unsep_clean 1{\relax}%

```

## 20.21. \XINT\_unsep\_cuzsmall

This is used as

```

\romannumeral0\xINT_unsep_cuzsmall (blocks of 1<8d>!)
\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax

```

It removes the 1's and !'s, and removes the leading zeroes \*of the first block\*.

Redone for 1.2l: the 1.2 variant was strangely in  $O(N^2)$  style.

```

395 \def\xINT_unsep_cuzsmall
396 {%
397   \expandafter\xINT_unsep_cuzsmall_x\the\numexpr0\xINT_unsep_loop
398 }%
399 \def\xINT_unsep_cuzsmall_x #1{%
400 \def\xINT_unsep_cuzsmall_x 0##1##2##3##4##5##6##7##8%
401 {%
402   \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8\relax
403 }}\xINT_unsep_cuzsmall_x{ }%

```

## 20.22. \XINT\_div\_unsepQ

This is used by division to remove separators from the produced quotient. The quotient is produced in the correct order. The routine will also remove leading zeroes. An extra initial block of 8 zeroes is possible and thus if present must be removed. Then the next eight digits must be cleaned of leading zeroes. Attention that there might be a single block of 8 zeroes. Expansion launched by `\romannumeral0`.

Rewritten for 1.2l in 1.2i style.

```

404 \def\xINT_div_unsepQ_delim {\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\Z}%
405 \def\xINT_div_unsepQ
406 {%
407   \expandafter\xINT_div_unsepQ_x\the\numexpr0\xINT_unsep_loop
408 }%
409 \def\xINT_div_unsepQ_x #1{%
410 \def\xINT_div_unsepQ_x 0##1##2##3##4##5##6##7##8##9%
411 {%
412   \xint_gob_til_Z ##9\xINT_div_unsepQ_one\Z

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
413 \xint_gob_til_eightzeroes ##1##2##3##4##5##6##7##8\XINT_div_unsepQ_y 00000000%
414 \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8\relax ##9%
415 }}\XINT_div_unsepQ_x{ }%
416 \def\XINT_div_unsepQ_y #1{%
417 \def\XINT_div_unsepQ_y ##1\relax ##2##3##4##5##6##7##8##9%
418 {%
419 \expandafter#1\the\numexpr ##2##3##4##5##6##7##8##9\relax
420 }}\XINT_div_unsepQ_y{ }%
421 \def\XINT_div_unsepQ_one#1\expandafter{\expandafter}%
```

### 20.23. \XINT\_div\_unsepR

This is used by division to remove separators from the produced remainder. The remainder is here in correct order. It must be cleaned of leading zeroes, possibly all the way.

Also rewritten for 1.21, the 1.2 version was  $O(N^2)$  style.

Terminator `\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\R`

We have a need for something like `\R` because it is not guaranteed the thing is not actually zero.

```
422 \def\XINT_div_unsepR
423 {%
424 \expandafter\XINT_div_unsepR_x\the\numexpr0\XINT_unsep_loop
425 }%
426 \def\XINT_div_unsepR_x#1{%
427 \def\XINT_div_unsepR_x 0{\expandafter#1\the\numexpr\XINT_cuz_loop}%
428 }\XINT_div_unsepR_x{ }%
```

### 20.24. \XINT\_zeroes\_forviii

`\romannumeral0\XINT_zeroes_forviii #1\R\R\R\R\R\R\R\R{10}0000001\W`

produces a string of  $k$  0's such that  $k + \text{length}(\#1)$  is smallest bigger multiple of eight.

```
429 \def\XINT_zeroes_forviii #1##2##3##4##5##6##7##8%
430 {%
431 \xint_gob_til_R #8\XINT_zeroes_forviii_end\R\XINT_zeroes_forviii
432 }%
433 \def\XINT_zeroes_forviii_end#1{%
434 \def\XINT_zeroes_forviii_end\R\XINT_zeroes_forviii ##1##2##3##4##5##6##7##8##9\W
435 {%
436 \expandafter#1\xint_gob_til_one ##2##3##4##5##6##7##8%
437 }}\XINT_zeroes_forviii_end{ }%
```

### 20.25. \XINT\_sepbyviii\_Z

This is used as

`\the\numexpr\XINT_sepbyviii_Z <8Ndigits>\XINT_sepbyviii_Z_end 2345678\relax`  
It produces `1<8d>!...1<8d>!1;!`

Prior to 1.21 it used `\Z` as terminator (hence the name). At 1.21 a switch to `;` was done. This was at a time I thought perhaps I would use an internal format maintaining such 8 digits blocks, and this had to be compatible with the `\csname...\endcsname` encapsulation in `\xintexpr` parsers. That rationale is obsolete since 1.4 usage of `\expanded` in `xintexpr`, but if an internal format is one day used it would be nice to be able to externalize it easily, so catcode 12 tokens are the most convenient.

As the expansion is done via successive `\numexpr`, it is convenient to use something such as `;` which terminates it and stays there. The `!` itself would do. But it proved convenient to have unique ending pattern. I could use two `!` perhaps.

Modified at 1.4n (2025/09/05). Unfortunately LuaMetaTeX has given to ; and : a meaning as operators inside its `\numexpr`. Let's hope ! does not acquire meaning too there. Anyway here I only need to use `\relax;! in \XINT_sepbyviii_Z_end`. First step in a tedious check of the entire codebase...

```
438 \def\XINT_sepbyviii_Z #1#2#3#4#5#6#7#8%
439 {%
440   1#1#2#3#4#5#6#7#8\expandafter!\the\numexpr\XINT_sepbyviii_Z
441 }%
442 \def\XINT_sepbyviii_Z_end #1\relax {\relax;!}%
```

## 20.26. \XINT\_sepbyviii\_andcount

This is used as

```
\the\numexpr\XINT_sepbyviii_andcount <8Ndigits>%
\XINT_sepbyviii_end 2345678\relax
\xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
\xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
```

It will produce

```
1<8d>!1<8d>!...1<8d>!1\xint:<count of blocks>\xint:
```

Used by `\XINT_div_prepare_g` for `\XINT_div_prepare_h`, and also by `\xintiiCmp`.

```
443 \def\XINT_sepbyviii_andcount
444 {%
445   \expandafter\XINT_sepbyviii_andcount_a\the\numexpr\XINT_sepbyviii
446 }%
447 \def\XINT_sepbyviii #1#2#3#4#5#6#7#8%
448 {%
449   1#1#2#3#4#5#6#7#8\expandafter!\the\numexpr\XINT_sepbyviii
450 }%
451 \def\XINT_sepbyviii_end #1\relax {\relax\XINT_sepbyviii_andcount_end!}%
452 \def\XINT_sepbyviii_andcount_a {\XINT_sepbyviii_andcount_b \xint_c_\xint:}%
453 \def\XINT_sepbyviii_andcount_b #1\xint:#2!#3!#4!#5!#6!#7!#8!#9!%
454 {%
455   #2\expandafter!\the\numexpr#3\expandafter!\the\numexpr#4\expandafter
456   !\the\numexpr#5\expandafter!\the\numexpr#6\expandafter!\the\numexpr
457   #7\expandafter!\the\numexpr#8\expandafter!\the\numexpr#9\expandafter!\the\numexpr
458   \expandafter\XINT_sepbyviii_andcount_b\the\numexpr #1+\xint_c_viii\xint:%
459 }%
460 \def\XINT_sepbyviii_andcount_end #1\XINT_sepbyviii_andcount_b\the\numexpr
461   #2+\xint_c_viii\xint:#3#4\W {\expandafter\xint:\the\numexpr #2+#3\xint:}%
```

## 20.27. \XINT\_rsepbyviii

This is used as

```
\the\numexpr1\XINT_rsepbyviii <8Ndigits>%
\XINT_rsepbyviii_end_A 2345678%
\XINT_rsepbyviii_end_B 2345678\relax UV%
```

and will produce

```
1<8digits>!1<8digits>\xint:1<8digits>!...
```

where the original digits are organized by eight, and the order inside successive pairs of blocks separated by `\xint:` has been reversed. Output ends either in `1<8d>!1<8d>\xint:1U\xint:` (even) or `1<8d>!1<8d>\xint:1V!1<8d>\xint:` (odd)

The U an V should be `\numexpr1` stoppers (or will expand and be ended by !). This macro is currently (1.2..1.2l) exclusively used in combination with `\XINT_sepandrev_andcount` or `\XINT_sepandrev`.

```

462 \def\XINT_rsepbyviii #1#2#3#4#5#6#7#8%
463 {%
464     \XINT_rsepbyviii_b {#1#2#3#4#5#6#7#8}%
465 }%
466 \def\XINT_rsepbyviii_b #1#2#3#4#5#6#7#8#9%
467 {%
468     #2#3#4#5#6#7#8#9\expandafter!\the\numexpr
469     1#1\expandafter\xint:\the\numexpr 1\XINT_rsepbyviii
470 }%
471 \def\XINT_rsepbyviii_end_B #1\relax #2#3{#2\xint:}%
472 \def\XINT_rsepbyviii_end_A #1#2\expandafter #3\relax #4#5{#5!1#2\xint:}%

```

## 20.28. \XINT\_sepandrev

This is used typically as

```

\romannumeral0\XINT_sepandrev <8Ndigits>%
    \XINT_rsepbyviii_end_A 2345678%
    \XINT_rsepbyviii_end_B 2345678\relax UV%
    \R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\W

```

and will produce

```
1<8digits>!1<8digits>!1<8digits>!...
```

where the blocks have been globally reversed. The UV here are only place holders (must be `\numexpr1` stoppers) to share same syntax as `\XINT_sepandrev_andcount`, they are gobbled (#2 in `\XINT_sepandrev_done`).

```

473 \def\XINT_sepandrev
474 {%
475     \expandafter\XINT_sepandrev_a\the\numexpr 1\XINT_rsepbyviii
476 }%
477 \def\XINT_sepandrev_a {\XINT_sepandrev_b {}}%
478 \def\XINT_sepandrev_b #1#2\xint:#3\xint:#4\xint:#5\xint:#6\xint:#7\xint:#8\xint:#9\xint:%
479 {%
480     \xint_gob_til_R #9\XINT_sepandrev_end\R
481     \XINT_sepandrev_b {#9!#8!#7!#6!#5!#4!#3!#2!#1}%
482 }%
483 \def\XINT_sepandrev_end\R\XINT_sepandrev_b #1#2\W {\XINT_sepandrev_done #1}%
484 \def\XINT_sepandrev_done #1#2!{ }%

```

## 20.29. \XINT\_sepandrev\_andcount

This is used typically as

```

\romannumeral0\XINT_sepandrev_andcount <8Ndigits>%
    \XINT_rsepbyviii_end_A 2345678%
    \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
    \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_
vi
    \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W

```

and will produce

```
<length>.1<8digits>!1<8digits>!1<8digits>!...
```

where the blocks have been globally reversed and `<length>` is the number of blocks.

```

485 \def\XINT_sepandrev_andcount

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
486 {%
487   \expandafter\XINT_sepandrev_andcount_a\the\numexpr 1\XINT_rsepybyviii
488 }%
489 \def\XINT_sepandrev_andcount_a {\XINT_sepandrev_andcount_b 0!{}}%
490 \def\XINT_sepandrev_andcount_b #1!#2#3\xint:#4\xint:#5\xint:#6\xint:#7\xint:#8\xint:#9\xint:%
491 {%
492   \xint_gob_til_R #9\XINT_sepandrev_andcount_end\R
493   \expandafter\XINT_sepandrev_andcount_b \the\numexpr #1+\xint_c_i!%
494   {#9!#8!#7!#6!#5!#4!#3!#2}%
495 }%
496 \def\XINT_sepandrev_andcount_end\R
497   \expandafter\XINT_sepandrev_andcount_b\the\numexpr #1+\xint_c_i!#2#3#4\W
498 {\expandafter\XINT_sepandrev_andcount_done\the\numexpr #3+\xint_c_xiv*#1!#2}%
499 \def\XINT_sepandrev_andcount_done#1{%
500 \def\XINT_sepandrev_andcount_done##1!##21##3!{\expandafter#1\the\numexpr##1-##3\xint:}%
501 }\XINT_sepandrev_andcount_done{ }%
```

### 20.30. \XINT\_rev\_nounsep

This is used as

`\romannumeral0\XINT_rev_nounsep {<blocks 1<8d>!>\R!\R!\R!\R!\R!\R!\R!\R!\W`

It reverses the blocks, keeping the 1's and ! separators. Used multiple times in the division algorithm. The inserted {} here is not optional.

```
502 \def\XINT_rev_nounsep #1#2!#3!#4!#5!#6!#7!#8!#9!%
503 {%
504   \xint_gob_til_R #9\XINT_rev_nounsep_end\R
505   \XINT_rev_nounsep {#9!#8!#7!#6!#5!#4!#3!#2!#1}%
506 }%
507 \def\XINT_rev_nounsep_end\R\XINT_rev_nounsep #1#2\W {\XINT_rev_nounsep_done #1}%
508 \def\XINT_rev_nounsep_done #11{ 1}%
```

### 20.31. \XINT\_unrevbyviii

Used as `\romannumeral0\XINT_unrevbyviii 1<8d>!...1<8d>!` terminated by

`1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W`

The `\romannumeral` in `unrevbyviii_a` is for special effects (expand some token which was put as `1<token>!` at the end of the original blocks). This mechanism is used by 1.2 subtraction (still true for 1.2l).

```
509 \def\XINT_unrevbyviii #11#2!1#3!1#4!1#5!1#6!1#7!1#8!1#9!%
510 {%
511   \xint_gob_til_R #9\XINT_unrevbyviii_a\R
512   \XINT_unrevbyviii {#9#8#7#6#5#4#3#2#1}%
513 }%
514 \def\XINT_unrevbyviii_a#1{%
515 \def\XINT_unrevbyviii_a\R\XINT_unrevbyviii ##1##2\W
516   {\expandafter#1\romannumeral`&&@\xint_gob_til_sc ##1}%
517 }\XINT_unrevbyviii_a{ }%
```

Can work with shorter ending pattern: `1;!1\R!1\R!1\R!1\R!1\R!1\R!\W` but the longer one of `unrevbyviii` is ok here too. Used currently (1.2) only by addition, now (1.2c) with long ending pattern. Does the final clean up of leading zeroes contrarily to general `\XINT_unrevbyviii`.

```
518 \def\XINT_smallunrevbyviii 1#1!1#2!1#3!1#4!1#5!1#6!1#7!1#8!#9\W%
519 {%
```



```

520 \expandafter\XINT_cuz_small\xint_gob_til_sc #8#7#6#5#4#3#2#1%
521 }%

```

## Core arithmetic

The four operations have been rewritten entirely for release 1.2. The new routines works with separated blocks of eight digits. They all measure first the lengths of the arguments, even addition and subtraction (this was not the case with [xintcore](#) 1.1 or earlier.)

The technique of chaining `\the\numexpr` induces a limitation on the maximal size depending on the size of the input save stack and the maximum expansion depth.

Side remark: I tested that `\the\numexpr` was more efficient than `\number`. But it reduced the allowable numbers for addition from 19976 digits to 19968 digits.

2025 update: both with 1.2 and 1.4m, using TeXLive 2025 (input stack size and expansion depth both at 10000) the maximal input size for addition is observed to be at 26648 (it was at 19968 with input stack size at 5000 still in 2021 with 1.4d) and the one for multiplication (or rather squaring) is 13320, i.e. about half which sounds logical. In both cases, the expansion depth is the limiting factor. The macro expansion is tested within an `\edef`.

During the ten years after 1.2 release it seems we indicated here a wrong value for the maximal input size for multiplication, perhaps a confusion between input and output size was made at that time. Sorry (but nobody reads this anyhow).

### 20.32. `\xintiiAdd`

1.21: `\xintiiAdd` made robust against non terminated input.

```

522 \def\xintiiAdd {\romannumeral0\xintiiadd }%
523 \def\xintiiadd #1{\expandafter\XINT_iiadd\romannumeral`&&@#1\xint:}%
524 \def\XINT_iiadd #1#2\xint:#3%
525 {%
526 \expandafter\XINT_add_nfork\expandafter#1\romannumeral`&&@#3\xint:#2\xint:
527 }%
528 \def\XINT_add_fork #1#2\xint:#3\xint:{\XINT_add_nfork #1#3\xint:#2\xint:}%
529 \def\XINT_add_nfork #1#2%
530 {%
531 \xint_UDzerofork
532 #1\XINT_add_firstiszero
533 #2\XINT_add_secondiszero
534 0{}%
535 \krof
536 \xint_UDsignsfork
537 #1#2\XINT_add_minusminus
538 #1-\XINT_add_minusplus
539 #2-\XINT_add_plusminus
540 --\XINT_add_plusplus
541 \krof #1#2%
542 }%
543 \def\XINT_add_firstiszero #1\krof 0#2#3\xint:#4\xint:{ #2#3}%
544 \def\XINT_add_secondiszero #1\krof #20#3\xint:#4\xint:{ #2#4}%
545 \def\XINT_add_minusminus #1#2%
546 {\expandafter-\romannumeral0\XINT_add_pp_a {}}}%
547 \def\XINT_add_minusplus #1#2{\XINT_sub_mm_a {}}#2}%
548 \def\XINT_add_plusminus #1#2%
549 {\expandafter\XINT_opp\romannumeral0\XINT_sub_mm_a #1{}}}%
550 \def\XINT_add_pp_a #1#2#3\xint:

```



## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

551 {%
552   \expandafter\XINT_add_pp_b
553     \romannumeral0\expandafter\XINT_sepandrev_andcount
554     \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\{10}0000001\W
555     #2#3\XINT_rsepbyviii_end_A 2345678%
556     \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
557       \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
558       \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
559   \X #1%
560 }%
561 \let\XINT_add_plusplus \XINT_add_pp_a
562 \def\XINT_add_pp_b #1\xint:#2\X #3\xint:
563 {%
564   \expandafter\XINT_add_checklengths
565   \the\numexpr #1\expandafter\xint:%
566   \romannumeral0\expandafter\XINT_sepandrev_andcount
567   \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\{10}0000001\W
568   #3\XINT_rsepbyviii_end_A 2345678%
569   \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
570     \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
571     \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
572     1;!1;!1;!1;!1\W #21;!1;!1;!1;!1\W
573     1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\W
574 }%

```

I keep #1.#2. to check if at most 6 + 6 base 10<sup>8</sup> digits which can be treated faster for final reverse. But is this overhead at all useful ?

```

575 \def\XINT_add_checklengths #1\xint:#2\xint:%
576 {%
577   \ifnum #2>#1
578     \expandafter\XINT_add_exchange
579   \else
580     \expandafter\XINT_add_A
581   \fi
582   #1\xint:#2\xint:%
583 }%
584 \def\XINT_add_exchange #1\xint:#2\xint:#3\W #4\W
585 {%
586   \XINT_add_A #2\xint:#1\xint:#4\W #3\W
587 }%
588 \def\XINT_add_A #1\xint:#2\xint:%
589 {%
590   \ifnum #1>\xint_c_vi
591     \expandafter\XINT_add_aa
592   \else \expandafter\XINT_add_aa_small
593   \fi
594 }%
595 \def\XINT_add_aa {\expandafter\XINT_add_out\the\numexpr\XINT_add_a \xint_c_ii}%
596 \def\XINT_add_out{\expandafter\XINT_cuz_small\romannumeral0\XINT_unrevbyviii {}}%
597 \def\XINT_add_aa_small
598   {\expandafter\XINT_smallunrevbyviii\the\numexpr\XINT_add_a \xint_c_ii}%

```

2 as first token of #1 stands for "no carry", 3 will mean a carry (we are adding 1<8digits> to 1<8digits>.) Version 1.2c has terminators of the shape 1;! , replacing the \Z! used in 1.2.

Call: `\the\numexpr\XINT_add_a 2#11;!!1;!!1;!!\W #21;!!1;!!1;!!\W` where #1 and #2 are blocks of `1<8d>!`, and #1 is at most as long as #2. This last requirement is a bit annoying (if one wants to do recursive algorithms but not have to check lengths), and I will probably remove it at some point.

Output: blocks of `1<8d>!` representing the addition, (least significant first), and a final `1;!`. In recursive algorithm this `1;!` terminator can thus conveniently be reused as part of input terminator (up to the length problem).

```

599 \def\XINT_add_a #1!#2!#3!#4!#5\W
600         #6!#7!#8!#9!%
601 {%
602     \XINT_add_b
603         #1!#6!#2!#7!#3!#8!#4!#9!%
604         #5\W
605 }%
606 \def\XINT_add_b #11#2#3!#4!%
607 {%
608     \xint_gob_til_sc #2\XINT_add_bi ;%
609     \expandafter\XINT_add_c\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
610 }%
611 \def\XINT_add_bi;\expandafter\XINT_add_c
612     \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6!#7!#8!#9!\W
613 {%
614     \XINT_add_k #1#3!#5!#7!#9!%
615 }%
616 \def\XINT_add_c #1#2\xint:%
617 {%
618     1#2\expandafter!\the\numexpr\XINT_add_d #1%
619 }%
620 \def\XINT_add_d #11#2#3!#4!%
621 {%
622     \xint_gob_til_sc #2\XINT_add_di ;%
623     \expandafter\XINT_add_e\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
624 }%
625 \def\XINT_add_di;\expandafter\XINT_add_e
626     \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6!#7!#8\W
627 {%
628     \XINT_add_k #1#3!#5!#7!%
629 }%
630 \def\XINT_add_e #1#2\xint:%
631 {%
632     1#2\expandafter!\the\numexpr\XINT_add_f #1%
633 }%
634 \def\XINT_add_f #11#2#3!#4!%
635 {%
636     \xint_gob_til_sc #2\XINT_add_fi ;%
637     \expandafter\XINT_add_g\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
638 }%
639 \def\XINT_add_fi;\expandafter\XINT_add_g
640     \the\numexpr#1+#2+#3-\xint_c_ii\xint:#4!#5!#6\W
641 {%
642     \XINT_add_k #1#3!#5!%
643 }%
```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
644 \def\XINT_add_g #1#2\xint:%
645 {%
646     1#2\expandafter!\the\numexpr\XINT_add_h #1%
647 }%
648 \def\XINT_add_h #1#2#3!#4!%
649 {%
650     \xint_gob_til_sc #2\XINT_add_hi ;%
651     \expandafter\XINT_add_i\the\numexpr#1+1#2#3+#4-\xint_c_ii\xint:%
652 }%
653 \def\XINT_add_hi;%
654     \expandafter\XINT_add_i\the\numexpr#1+#2+#3-\xint_c_ii\xint:#4\W
655 {%
656     \XINT_add_k #1#3!%
657 }%
658 \def\XINT_add_i #1#2\xint:%
659 {%
660     1#2\expandafter!\the\numexpr\XINT_add_a #1%
661 }%
662 \def\XINT_add_k #1{\if #12\expandafter\XINT_add_ke\else\expandafter\XINT_add_l \fi}%
663 \def\XINT_add_ke #1#2\W {\XINT_add_kf #11;!}%
664 \def\XINT_add_kf 1{\relax }%
665 \def\XINT_add_l 1#1#2{\xint_gob_til_sc #1\XINT_add_lf ;\XINT_add_m 1#1#2}%
666 \def\XINT_add_lf #1\W {1\relax 00000001!1;!}%
667 \def\XINT_add_m #1!\{\expandafter\XINT_add_n\the\numexpr\xint_c_i+#1\xint:%
668 \def\XINT_add_n #1#2\xint:{1#2\expandafter!\the\numexpr\XINT_add_o #1}%
    Here 2 stands for "carry", and 1 for "no carry" (we have been adding 1 to 1<8digits>.)
669 \def\XINT_add_o #1{\if #12\expandafter\XINT_add_l\else\expandafter\XINT_add_ke \fi}%

```

### 20.33. \xintiicmp

Modified at 1.4m (2022/06/10). Now uses the `\xintstrcmp` engine primitive.

```
670 \def\xintiicmp {\romannumeral0\xintiicmp }%
671 \def\xintiicmp #1{\expandafter\XINT_iicmp\romannumeral`&&@#1\xint:%
672 \def\XINT_iicmp #1#2\xint:#3%
673 {%
674     \expandafter\XINT_cmp_nfork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
675 }%
676 \def\XINT_cmp_nfork #1#2%
677 {%
678     \xint_UDzerofork
679     #1\XINT_cmp_firstiszero
680     #2\XINT_cmp_secondiszero
681     0{}%
682     \krof
683     \xint_UDsignsfork
684     #1#2\XINT_cmp_minusminus
685     #1-\XINT_cmp_minusplus
686     #2-\XINT_cmp_plusminus
687     --\XINT_cmp_plusplus
688     \krof #1#2%
689 }%
690 \def\XINT_cmp_firstiszero #1\krof 0#2#3\xint:#4\xint:

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

691 {%
692   \xint_UDzerominusfork
693     #2-{ 0}%
694     0#2{ 1}%
695     0-{ -1}%
696   \krof
697 }%
698 \def\xINT_cmp_secondiszero #1\krof #20#3\xint:#4\xint:
699 {%
700   \xint_UDzerominusfork
701     #2-{ 0}%
702     0#2{ -1}%
703     0-{ 1}%
704   \krof
705 }%
706 \def\xINT_cmp_plusminus #1\xint:#2\xint:{ 1}%
707 \def\xINT_cmp_minusplus #1\xint:#2\xint:{ -1}%
708 \def\xINT_cmp_minusminus
709   --{\expandafter\xINT_opp\romannumeral0\xINT_cmp_plusplus {}}}%

The \romannumeral0 trigger induces some complications here to terminate nicely without grabbing
too many tokens in the stream or deteriorating expansion quality of the non-equal-length branches.
\expanded simplifies things.

710 \def\xINT_cmp_plusplus #1#2#3\xint:#4\xint:{\expanded{ %
711   \ifcase\expandafter\xINT_cntSgn\the\numexpr\xintLength{#1#4}-\xintLength{#2#3}\xint:
712     \xintstrcmp{#1#4}{#2#3}\or1\else-1\fi
713   }}%
714 }%
```

Prior to 1.4m the «strcmp» primitive was not used by *xintcore*. Here is the old implementation:

```

\def\xINT_cmp_plusplus #1#2#3\xint:
{%
  \expandafter\xINT_cmp_pp
  \the\numexpr\expandafter\xINT_sepbyviii_andcount
  \romannumeral0\xINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
  #2#3\xINT_sepbyviii_end 2345678\relax
  \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
  \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
  #1%
}%
\def\xINT_cmp_pp #1\xint:#2\xint:#3\xint:
{%
  \expandafter\xINT_cmp_checklengths
  \the\numexpr #2\expandafter\xint:%
  \the\numexpr\expandafter\xINT_sepbyviii_andcount
  \romannumeral0\xINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
  #3\xINT_sepbyviii_end 2345678\relax
  \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
  \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
  #1;!1;!1;!1;!1;\W
}%
\def\xINT_cmp_checklengths #1\xint:#2\xint:#3\xint:
{%
  \ifnum #1=#3
```

```

        \expandafter\xint_firstoftwo
    \else
        \expandafter\xint_secondoftwo
    \fi
    \XINT_cmp_a {\XINT_cmp_distinctlengths {#1}{#3}}#2;!1;!1;!1;!1\W
}%
\def\XINT_cmp_distinctlengths #1#2#3\W #4\W
{%
    \ifnum #1>#2
        \expandafter\xint_firstoftwo
    \else
        \expandafter\xint_secondoftwo
    \fi
    { -1}{ 1}%
}%
\def\XINT_cmp_a 1#1!1#2!1#3!1#4!#5\W 1#6!1#7!1#8!1#9!%
{%
    \xint_gob_til_sc #1\XINT_cmp_equal ;%
    \ifnum #1>#6 \XINT_cmp_gt\fi
    \ifnum #1<#6 \XINT_cmp_lt\fi
    \xint_gob_til_sc #2\XINT_cmp_equal ;%
    \ifnum #2>#7 \XINT_cmp_gt\fi
    \ifnum #2<#7 \XINT_cmp_lt\fi
    \xint_gob_til_sc #3\XINT_cmp_equal ;%
    \ifnum #3>#8 \XINT_cmp_gt\fi
    \ifnum #3<#8 \XINT_cmp_lt\fi
    \xint_gob_til_sc #4\XINT_cmp_equal ;%
    \ifnum #4>#9 \XINT_cmp_gt\fi
    \ifnum #4<#9 \XINT_cmp_lt\fi
    \XINT_cmp_a #5\W
}%
\def\XINT_cmp_lt#1{\def\XINT_cmp_lt\fi ##1\W ##2\W {\fi#1-1}}\XINT_cmp_lt{ }%
\def\XINT_cmp_gt#1{\def\XINT_cmp_gt\fi ##1\W ##2\W {\fi#11}}\XINT_cmp_gt{ }%
\def\XINT_cmp_equal #1\W #2\W { 0}%

```

## 20.34. \xintiisub

Entirely rewritten for 1.2.

Refactored at 1.2l. I was initially aiming at clinching some internal format of the type 1<8digits>!...1<8digits>! for chaining the arithmetic operations (as a preliminary step to deciding upon some internal format for *xintfrac* macros), thus I wanted to uniformize delimiters in particular and have some core macros inputting and outputting such formats. But the way division is implemented makes it currently very hard to obtain a satisfactory solution. For subtraction I got there almost, but there was added overhead and, as the core sub-routine still assumed the shorter number will be positioned first, one would need to record the length also in the basic internal format, or add the overhead to not make assumption on which one is shorter. I thus but back-tracked my steps but in passing I improved the efficiency (probably) in the worst case branch.

Sadly this 1.2l refactoring left an extra ! in macro *\XINT\_sub\_1\_Ida*. This bug shows only in rare circumstances which escaped out test suite :( Fixed at 1.2q.

The other reason for backtracking was in relation with the decimal numbers. Having a core format in base 10<sup>8</sup> but ultimately the radix is actually 10 leads to complications. I could use radix 10<sup>8</sup> for *\xintiexpr* only, but then I need to make it compatible with sub-*\xintiexpr* in *\xintexpr*, etc... there are many issues of this type.

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

I considered also an approach like in the 1.21 `\xintiiCmp`, but decided to stick with the method here for now.

```
715 \def\xintiiSub    {\romannumeral0\xintiisub }%
716 \def\xintiisub #1{\expandafter\XINT_iisub\romannumeral`&&@#1\xint:}%
717 \def\XINT_iisub #1#2\xint:#3%
718 {%
719   \expandafter\XINT_sub_nfork\expandafter
720     #1\romannumeral`&&@#3\xint:#2\xint:
721 }%
722 \def\XINT_sub_nfork #1#2%
723 {%
724   \xint_UDzerofork
725     #1\XINT_sub_firstiszero
726     #2\XINT_sub_secondiszero
727     0{}}%
728   \krof
729   \xint_UDsignsfork
730     #1#2\XINT_sub_minusminus
731     #1-\XINT_sub_minusplus
732     #2-\XINT_sub_plusminus
733     --\XINT_sub_plusplus
734   \krof #1#2%
735 }%
736 \def\XINT_sub_firstiszero #1\krof 0#2#3\xint:#4\xint:{\XINT_opp #2#3}%
737 \def\XINT_sub_secondiszero #1\krof #20#3\xint:#4\xint:{ #2#4}%
738 \def\XINT_sub_plusminus #1#2{\XINT_add_pp_a #1{}}%
739 \def\XINT_sub_plusplus #1#2%
740   {\expandafter\XINT_opp\romannumeral0\XINT_sub_mm_a #1#2}%
741 \def\XINT_sub_minusplus #1#2%
742   {\expandafter-\romannumeral0\XINT_add_pp_a {}}#2}%
743 \def\XINT_sub_minusminus #1#2{\XINT_sub_mm_a {}}}%
744 \def\XINT_sub_mm_a #1#2#3\xint:
745 {%
746   \expandafter\XINT_sub_mm_b
747     \romannumeral0\expandafter\XINT_sepandrev_andcount
748     \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
749     #2#3\XINT_rsepbyviii_end_A 2345678%
750     \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
751     \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
752     \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
753   \X #1%
754 }%
755 \def\XINT_sub_mm_b #1\xint:#2\X #3\xint:
756 {%
757   \expandafter\XINT_sub_checklengths
758   \the\numexpr #1\expandafter\xint:%
759   \romannumeral0\expandafter\XINT_sepandrev_andcount
760   \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
761   #3\XINT_rsepbyviii_end_A 2345678%
762   \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
763   \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
764   \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

765      1;!1;!1;!1;!1;\W
766      #21;!1;!1;!1;!1;\W
767      1;!1\R!1\R!1\R!1\R!%
768      1\R!1\R!1\R!1\R!\W
769 }%
770 \def\XINT_sub_checklengths #1\xint:#2\xint:%
771 {%
772     \ifnum #2>#1
773         \expandafter\XINT_sub_exchange
774     \else
775         \expandafter\XINT_sub_aa
776     \fi
777 }%
778 \def\XINT_sub_exchange #1\W #2\W
779 {%
780     \expandafter\XINT_opp\romannumeral0\XINT_sub_aa #2\W #1\W
781 }%
782 \def\XINT_sub_aa
783 {%
784     \expandafter\XINT_sub_out\the\numexpr\XINT_sub_a\xint_c_i
785 }%

```

The post-processing (clean-up of zeros, or rescue of situation with A-B where actually B turns out bigger than A) will be done by a macro which depends on circumstances and will be initially last token before the reversion done by `\XINT_unrevbyviii`.

```

786 \def\XINT_sub_out {\XINT_unrevbyviii{}}%

```

1 as first token of #1 stands for "no carry", 0 will mean a carry.

Call: `\the\numexpr`  
`\XINT_sub_a 1#11;!1;!1;!1;!1;\W`  
`#21;!1;!1;!1;!1;\W`

where #1 and #2 are blocks of  $1 < 8d < 1$ , #1 (=B) \*must\* be at most as long as #2 (=A), (in radix  $10^8$ ) and the routine wants to compute  $\#2 - \#1 = A - B$

1.21 uses 1;! delimiters to match those of addition (and multiplication). But in the end I reverted the code branch which made it possible to chain such operations keeping internal format in 8 digits blocks throughout.

`\numexpr` governed expansion stops with various possibilities:

- Type Ia: #1 shorter than #2, no final carry
- Type Ib: #1 shorter than #2, a final carry but next block of #2 > 1
- Type Ica: #1 shorter than #2, a final carry, next block of #2 is final and = 1
- Type Icb: as Ica except that 00000001 block from #2 was not final
- Type Id: #1 shorter than #2, a final carry, next block of #2 = 0
- Type IIa: #1 same length as #2, turns out it was  $\leq$  #2.
- Type IIb: #1 same length as #2, but turned out > #2.

Various type of post actions are then needed:

- Ia: clean up of zeros in most significant block of 8 digits
- Ib: as Ia
- Ic: there may be significant blocks of 8 zeros to clean up from result. Only case Ica may have arbitrarily many of them, case Icb has only one such block.
- Id: blocks of 99999999 may propagate and there might a be final zero block created which has to be cleaned up.
- IIa: arbitrarily many zeros might have to be removed.
- IIb: We wanted  $\#2 - \#1 = -(\#1 - \#2)$ , but we got  $10^{\{8N\}} + \#2 - \#1 = 10^{\{8N\}} - (\#1 - \#2)$ . We need to do the correction then we are as in IIa situation, except that final result can not be zero.

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

The 1.2l method for this correction is (presumably, testing takes lots of time, which I do not have) more efficient than in 1.2 release.

```
787 \def\XINT_sub_a #1!#2!#3!#4!#5\W #6!#7!#8!#9!%
788 {%
789     \XINT_sub_b
790     #1!#6!#2!#7!#3!#8!#4!#9!%
791     #5\W
792 }%
```

As 1.2l code uses `1<8digits>!` blocks one has to be careful with the carry digit 1 or 0: A `#11#2#3` pattern would result into an empty `#1` if the carry digit which is upfront is 1, rather than setting `#1=1`.

```
793 \def\XINT_sub_b #1#2#3#4!#5!%
794 {%
795     \xint_gob_til_sc #3\XINT_sub_bi ;%
796     \expandafter\XINT_sub_c\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:%
797 }%
798 \def\XINT_sub_c 1#1#2\xint:%
799 {%
800     1#2\expandafter!\the\numexpr\XINT_sub_d #1%
801 }%
802 \def\XINT_sub_d #1#2#3#4!#5!%
803 {%
804     \xint_gob_til_sc #3\XINT_sub_di ;%
805     \expandafter\XINT_sub_e\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:%
806 }%
807 \def\XINT_sub_e 1#1#2\xint:%
808 {%
809     1#2\expandafter!\the\numexpr\XINT_sub_f #1%
810 }%
811 \def\XINT_sub_f #1#2#3#4!#5!%
812 {%
813     \xint_gob_til_sc #3\XINT_sub_fi ;%
814     \expandafter\XINT_sub_g\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:%
815 }%
816 \def\XINT_sub_g 1#1#2\xint:%
817 {%
818     1#2\expandafter!\the\numexpr\XINT_sub_h #1%
819 }%
820 \def\XINT_sub_h #1#2#3#4!#5!%
821 {%
822     \xint_gob_til_sc #3\XINT_sub_hi ;%
823     \expandafter\XINT_sub_i\the\numexpr#1+1#5-#3#4-\xint_c_i\xint:%
824 }%
825 \def\XINT_sub_i 1#1#2\xint:%
826 {%
827     1#2\expandafter!\the\numexpr\XINT_sub_a #1%
828 }%
829 \def\XINT_sub_bi;%
830     \expandafter\XINT_sub_c\the\numexpr#1+1#2-#3\xint:
831     #4!#5!#6!#7!#8!#9!\W
832 {%
833     \XINT_sub_k #1#2!#5!#7!#9!%
```



## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

834 }%
835 \def\XINT_sub_di;%
836   \expandafter\XINT_sub_e\the\numexpr#1+1#2-#3\xint:
837   #4!#5!#6!#7!#8\W
838 {%
839   \XINT_sub_k #1#2!#5!#7!%
840 }%
841 \def\XINT_sub_fi;%
842   \expandafter\XINT_sub_g\the\numexpr#1+1#2-#3\xint:
843   #4!#5!#6\W
844 {%
845   \XINT_sub_k #1#2!#5!%
846 }%
847 \def\XINT_sub_hi;%
848   \expandafter\XINT_sub_i\the\numexpr#1+1#2-#3\xint:
849   #4\W
850 {%
851   \XINT_sub_k #1#2!%
852 }%

```

B terminated. Have we reached the end of A (necessarily at least as long as B) ? (we are computing A-B, digits of B come first).

If not, then we are certain that even if there is carry it will not propagate beyond the end of A. But it may propagate far transforming chains of 00000000 into 99999999, and if it does go to the final block which possibly is just  $1<00000001>!$ , we will have those eight zeros to clean up.

If A and B have the same length (in base  $10^8$ ) then arbitrarily many zeros might have to be cleaned up, and if  $A < B$ , the whole result will have to be complemented first.

```

853 \def\XINT_sub_k #1#2#3%
854 {%
855   \xint_gob_til_sc #3\XINT_sub_p;\XINT_sub_l #1#2#3%
856 }%
857 \def\XINT_sub_l #1%
858   {\xint_UDzerofork #1\XINT_sub_l_carry 0\XINT_sub_l_Ia\krof}%
859 \def\XINT_sub_l_Ia 1#1;!#2\W{1\relax#1;!1\XINT_sub_fix_none!}%
860 \def\XINT_sub_l_carry 1#1!{\ifcase #1
861   \expandafter \XINT_sub_l_Id
862   \or \expandafter \XINT_sub_l_Ic
863   \else\expandafter \XINT_sub_l_Ib\fi 1#1!}%

```

No `\cs{relax}` here at 1.4n before the ; for LuaMetaTeX's `\cs{numexpr}`. The #1 will bring own delimiter. AM I CERTAIN OF THAT?

```

864 \def\XINT_sub_l_Ib #1;#2\W {-\xint_c_i+#1;!1\XINT_sub_fix_none!}%
865 \def\XINT_sub_l_Ic 1#1!1#2#3!#4;#5\W
866 {%
867   \xint_gob_til_sc #2\XINT_sub_l_Ica;%
868   1\relax 00000000!1#2#3!#4;!1\XINT_sub_fix_none!%
869 }%

```

We need to add some extra delimiters at the end for post-action by `\XINT_num`, so we first grab the material up to `\W`

**Modified at 1.4n (2025/09/05).** A `\relax` added for LuaMetaTeX compatibility.

```

870 \def\XINT_sub_l_Ica#1\W
871 {%
872   1\relax;!1\XINT_sub_fix_cuz!%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

873      1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
874      \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
875 }%
876 \def\xINT_sub_1_Id 1#1!%
877     {199999999\expandafter!\the\numexpr \XINT_sub_1_Id_a}%
878 \def\xINT_sub_1_Id_a 1#1!{\ifcase #1
879     \expandafter \XINT_sub_1_Id
880     \or \expandafter \XINT_sub_1_Id_b
881     \else\expandafter \XINT_sub_1_Id_b\fi 1#1!}%
882 \def\xINT_sub_1_Id_b 1#1!1#2#3!#4!#5\W
883 {%
884     \xint_gob_til_sc #2\XINT_sub_1_Id_a;%
885     1\relax 00000000!1#2#3!#4!#5\XINT_sub_fix_none!%
886 }%

```

**Modified at 1.4n (2025/09/05).** LuaMetaTeX

```

887 \def\xINT_sub_1_Ida#1\XINT_sub_fix_none{1\relax;!1\XINT_sub_fix_none}%

```

This is the case where both operands have same  $10^8$ -base length.

We were handling A-B but perhaps B>A. The situation with A=B is also annoying because we then have to clean up all zeros but don't know where to stop (if A>B the first non-zero 8 digits block would tell use when).

Here again we need to grab #3\W to position the actually used terminating delimiters.

**Modified at 1.4n (2025/09/05).** `\relax` for LuaMetaTeX `\numexpr`

```

888 \def\xINT_sub_p;\XINT_sub_1 #1#2\W #3\W
889 {%
890     \xint_UDzerofork
891     #1{1\relax;!1\XINT_sub_fix_neg!%
892     1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
893     \xint_bye2345678\xint_bye1099999988\relax}% A - B, B > A
894     0{1\relax;!1\XINT_sub_fix_cuz!%
895     1;!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W}%
896     \krof
897     \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
898 }%

```

Routines for post-processing after reversal, and removal of separators. It is a matter of cleaning up zeros, and possibly in the bad case to take a complement before that.

```

899 \def\xINT_sub_fix_none;\XINT_cuz_small}%
900 \def\xINT_sub_fix_cuz ;{\expandafter\xINT_num_cleanup\the\numexpr\xINT_num_loop}%

```

Case with A and B same number of digits in base  $10^8$  and B>A.

1.2l subtle chaining on the model of the 1.2i rewrite of `\xintInc` and similar routines. After taking complement, leading zeroes need to be cleaned up as in B<=A branch.

```

901 \def\xINT_sub_fix_neg;%
902 {%
903     \expandafter-\romannumeral0\expandafter
904     \XINT_sub_comp_finish\the\numexpr\xINT_sub_comp_loop
905 }%
906 \def\xINT_sub_comp_finish 0{\XINT_sub_fix_cuz;}%
907 \def\xINT_sub_comp_loop #1#2#3#4#5#6#7#8%
908 {%
909     \expandafter\xINT_sub_comp_clean
910     \the\numexpr \xint_c_xi_e_viii_mone-#1#2#3#4#5#6#7#8\xINT_sub_comp_loop
911 }%

```

#1 = 0 signifie une retenue, #1 = 1 pas de retenue, ce qui ne peut arriver que tant qu'il n'y a que des zéros du côté non significatif. Lorsqu'on est revenu au début on a forcément une retenue.

```
912 \def\XINT_sub_comp_clean 1#1{+#1\relax}%
```

## 20.35. \xintiiMul

Completely rewritten for 1.2.

1.21: \xintiiMul made robust against non terminated input.

```
913 \def\xintiiMul {\romannumeral0\xintiimul }%
914 \def\xintiimul #1%
915 {%
916   \expandafter\XINT_iimul\romannumeral`&&@#1\xint:
917 }%
918 \def\XINT_iimul #1#2\xint:#3%
919 {%
920   \expandafter\XINT_mul_nfork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
921 }%
```

1.2 I have changed the fork, and it complicates matters elsewhere.

ATTENTION for example that 1.4 \xintiiPrd uses \XINT\_mul\_nfork now.

```
922 \def\XINT_mul_fork #1#2\xint:#3\xint:{\XINT_mul_nfork #1#3\xint:#2\xint:}%
923 \def\XINT_mul_nfork #1#2%
924 {%
925   \xint_UDzerofork
926   #1\XINT_mul_zero
927   #2\XINT_mul_zero
928   0{}}%
929   \krof
930   \xint_UDsignsfork
931   #1#2\XINT_mul_minusminus
932   #1-\XINT_mul_minusplus
933   #2-\XINT_mul_plusminus
934   --\XINT_mul_plusplus
935   \krof #1#2%
936 }%
937 \def\XINT_mul_zero #1\krof #2#3\xint:#4\xint:{ 0}%
938 \def\XINT_mul_minusminus #1#2{\XINT_mul_plusplus {}}}%
939 \def\XINT_mul_minusplus #1#2%
940   {\expandafter-\romannumeral0\XINT_mul_plusplus {}}#2}%
941 \def\XINT_mul_plusminus #1#2%
942   {\expandafter-\romannumeral0\XINT_mul_plusplus #1{}}}%
943 \def\XINT_mul_plusplus #1#2#3\xint:
944 {%
945   \expandafter\XINT_mul_pre_b
946   \romannumeral0\expandafter\XINT_sepandrev_andcount
947   \romannumeral0\XINT_zeroes_forviii #2#3\R\R\R\R\R\R\R\R{10}0000001\W
948   #2#3\XINT_rsepybviii_end_A 2345678%
949   \XINT_rsepybviii_end_B 2345678\relax\xint_c_ii\xint_c_i
950   \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
951   \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
952   \W #1%
953 }%
954 \def\XINT_mul_pre_b #1\xint:#2\W #3\xint:
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

955 {%
956   \expandafter\XINT_mul_checklengths
957   \the\numexpr #1\expandafter\xint:%
958   \romannumeral0\expandafter\XINT_sepandrev_andcount
959   \romannumeral0\XINT_zeroes_forviii #3\R\R\R\R\R\R\R\R{10}0000001\W
960   #3\XINT_rsepbyviii_end_A 2345678%
961   \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
962   \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
963   \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
964   1;!\W #21;!\%
965   1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\W
966 }%

```

Cooking recipe, 2015/10/05.

```

967 \def\XINT_mul_checklengths #1\xint:#2\xint:%
968 {%
969   \ifnum #2=\xint_c_i\expandafter\XINT_mul_smallbyfirst\fi
970   \ifnum #1=\xint_c_i\expandafter\XINT_mul_smallbysecond\fi
971   \ifnum #2<#1
972     \ifnum \numexpr (#2-\xint_c_i)*(#1-#2)<383
973       \XINT_mul_exchange
974     \fi
975   \else
976     \ifnum \numexpr (#1-\xint_c_i)*(#2-#1)>383
977       \XINT_mul_exchange
978     \fi
979   \fi
980   \XINT_mul_start
981 }%
982 \def\XINT_mul_smallbyfirst #1\XINT_mul_start 1#2!1;!\W
983 {%
984   \ifnum#2=\xint_c_i\expandafter\XINT_mul_oneisone\fi
985   \ifnum#2<\xint_c_xxii\expandafter\XINT_mul_verysmall\fi
986   \expandafter\XINT_mul_out\the\numexpr\XINT_smallmul 1#2!%
987 }%
988 \def\XINT_mul_smallbysecond #1\XINT_mul_start #2\W 1#3!1;!\%
989 {%
990   \ifnum#3=\xint_c_i\expandafter\XINT_mul_oneisone\fi
991   \ifnum#3<\xint_c_xxii\expandafter\XINT_mul_verysmall\fi
992   \expandafter\XINT_mul_out\the\numexpr\XINT_smallmul 1#3!#2%
993 }%
994 \def\XINT_mul_oneisone #1!{\XINT_mul_out }%
995 \def\XINT_mul_verysmall\expandafter\XINT_mul_out
996   \the\numexpr\XINT_smallmul 1#1!%
997   {\expandafter\XINT_mul_out\the\numexpr\XINT_verysmallmul 0\xint:#1!}%
998 \def\XINT_mul_exchange #1\XINT_mul_start #2\W #31;!\%
999   {\fi\fi\XINT_mul_start #31;!\W #2}%
1000 \def\XINT_mul_start
1001   {\expandafter\XINT_mul_out\the\numexpr\XINT_mul_loop 100000000!1;!\W}%
1002 \def\XINT_mul_out
1003   {\expandafter\XINT_cuz_small\romannumeral0\XINT_unrevbyviii {}}%

```

Call:

```

\the\numexpr \XINT_mul_loop 100000000!1;!\W #11;!\W #21;!
```

where #1 and #2 are (globally reversed) blocks  $1<8d>!$ . Its is generally more efficient if #1 is the shorter one, but a better recipe is implemented in `\XINT_mul_checklengths`. One may call `\XINT_mul_loop` directly (but multiplication by zero will produce many 100000000! blocks on output).

Ends after having produced:  $1<8d>!\dots 1<8d>!1;!$ . The last 8-digits block is significant one. It can not be 100000000! except if the loop was called with a zero operand.

Thus `\XINT_mul_loop` can be conveniently called directly in recursive routines, as the output terminator can serve as input terminator, we can arrange to not have to grab the whole thing again.

```
1004 \def\XINT_mul_loop #1\W #2\W 1#3!%
1005 {%
1006   \xint_gob_til_sc #3\XINT_mul_e ;%
1007   \expandafter\XINT_mul_a\the\numexpr \XINT_smallmul 1#3!#2\W
1008   #1\W #2\W
1009 }%
```

Each of #1 and #2 brings its 1; for `\XINT_add_a`.

```
1010 \def\XINT_mul_a #1\W #2\W
1011 {%
1012   \expandafter\XINT_mul_b\the\numexpr
1013   \XINT_add_a \xint_c_ii #21;!1;!1;!1\W #11;!1;!1;!1\W\W
1014 }%
1015 \def\XINT_mul_b 1#1!{1#1\expandafter!\the\numexpr\XINT_mul_loop }%
1016 \def\XINT_mul_e;#1\W 1#2\W #3\W {1\relax #2}%

```

1.2 small and mini multiplication in base  $10^8$  with carry. Used by the main multiplication routines. But division, float factorial, etc.. have their own variants as they need output with specific constraints.

The `minimulwc` has  $1<8\text{digits carry}>.<4\text{ high digits}>.<4\text{ low digits}>.<8\text{digits}>$ .

It produces a block  $1<8d>!$  and then jump back into `\XINT_smallmul_a` with the new 8digits carry as argument. The `\XINT_smallmul_a` fetches a new  $1<8d>!$  block to multiply, and calls back `\XINT_minimulwc` having stored the multiplicand for re-use later. When the loop terminates, the final carry is checked for being nul, and in all cases the output is terminated by a 1;!

Multiplication by zero will produce blocks of zeros.

```
1017 \def\XINT_minimulwc_a 1#1\xint:#2\xint:#3!#4#5#6#7#8\xint:%
1018 {%
1019   \expandafter\XINT_minimulwc_b
1020   \the\numexpr \xint_c_x^ix+#1+#3*#8\xint:
1021               #3*#4#5#6#7+#2*#8\xint:
1022               #2*#4#5#6#7\xint:%
1023 }%
1024 \def\XINT_minimulwc_b 1#1#2#3#4#5#6\xint:#7\xint:%
1025 {%
1026   \expandafter\XINT_minimulwc_c
1027   \the\numexpr \xint_c_x^ix+#1#2#3#4#5+#7\xint:#6\xint:%
1028 }%
1029 \def\XINT_minimulwc_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
1030 {%
1031   1#6#7\expandafter!%
1032   \the\numexpr\expandafter\XINT_smallmul_a
1033   \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8\xint:%
1034 }%
1035 \def\XINT_smallmul 1#1#2#3#4#5!{\XINT_smallmul_a 100000000\xint:#1#2#3#4\xint:#5!}%
1036 \def\XINT_smallmul_a #1\xint:#2\xint:#3!1#4!%
1037 {%

```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1038 \xint_gob_til_sc #4\XINT_smallmul_e;%
1039 \XINT_minimulwc_a #1\xint:#2\xint:#3!#4\xint:#2\xint:#3!%
1040 }%
1041 \def\XINT_smallmul_e;\XINT_minimulwc_a 1#1\xint:#2;#3!%
1042 {\xint_gob_til_eightzeroes #1\XINT_smallmul_f 000000001\relax #1!1;!}%
1043 \def\XINT_smallmul_f 000000001\relax 00000000!1{1\relax}%
1044 \def\XINT_verysmallmul #1\xint:#2!1#3!%
1045 {%
1046 \xint_gob_til_sc #3\XINT_verysmallmul_e;%
1047 \expandafter\XINT_verysmallmul_a
1048 \the\numexpr #2*#3+#1\xint:#2!%
1049 }%
1050 \def\XINT_verysmallmul_e;\expandafter\XINT_verysmallmul_a\the\numexpr
1051 #1+#2#3\xint:#4!%
1052 {\xint_gob_til_zero #2\XINT_verysmallmul_f 0\xint_c_x^viii+#2#3!1;!}%
1053 \def\XINT_verysmallmul_f #1!1{1\relax}%
1054 \def\XINT_verysmallmul_a #1#2\xint:%
1055 {%
1056 \unless\ifnum #1#2<\xint_c_x^ix
1057 \expandafter\XINT_verysmallmul_bi\else
1058 \expandafter\XINT_verysmallmul_bj\fi
1059 \the\numexpr \xint_c_x^ix+#1#2\xint:%
1060 }%
1061 \def\XINT_verysmallmul_bj{\expandafter\XINT_verysmallmul_cj }%
1062 \def\XINT_verysmallmul_cj 1#1#2\xint:%
1063 {1#2\expandafter!\the\numexpr\XINT_verysmallmul #1\xint:}%
1064 \def\XINT_verysmallmul_bi\the\numexpr\xint_c_x^ix+#1#2#3\xint:%
1065 {1#3\expandafter!\the\numexpr\XINT_verysmallmul #1#2\xint:}%

```

Used by division and by squaring, not by multiplication itself.

This routine does not loop, it only does one mini multiplication with input format <4 high digits>.<4 low digits>!<8 digits>!, and on output 1<8d>!1<8d>!, with least significant block first.

```

1066 \def\XINT_minimul_a #1\xint:#2!#3#4#5#6#7!%
1067 {%
1068 \expandafter\XINT_minimul_b
1069 \the\numexpr \xint_c_x^viii+#2*#7\xint:#2*#3#4#5#6+#1*#7\xint:#1*#3#4#5#6\xint:%
1070 }%
1071 \def\XINT_minimul_b 1#1#2#3#4#5\xint:#6\xint:%
1072 {%
1073 \expandafter\XINT_minimul_c
1074 \the\numexpr \xint_c_x^ix+#1#2#3#4+#6\xint:#5\xint:%
1075 }%
1076 \def\XINT_minimul_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
1077 {%
1078 1#6#7\expandafter!\the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8!%
1079 }%

```

## 20.36. \xintiiDivision

Completely rewritten for 1.2.

WARNING: some comments below try to describe the flow of tokens but they date back to xint 1.09j and I updated them on the fly while doing the 1.2 version. As the routine now works in base  $10^8$ , not  $10^4$  and "drops" the quotient digits, rather than store them upfront as the earlier code, I may

well have not correctly converted all such comments. At the last minute some previously #1 became stuff like #1#2#3#4, then of course the old comments describing what the macro parameters stand for are necessarily wrong.

Side remark: the way tokens are grouped was not essentially modified in 1.2, although the situation has changed. It was fine-tuned in xint 1.0/1.1 but the context has changed, and perhaps I should revisit this. As a corollary to the fact that quotient digits are now left behind thanks to the chains of `\numexpr`, some macros which in 1.0/1.1 fetched up to 9 parameters now need handle less such parameters. Thus, some rationale for the way the code was structured has disappeared.

1.21: `\xintiDivision` et al. made robust against non terminated input.

#1 = A, #2 = B. On calcule le quotient et le reste dans la division euclidienne de A par B:  $A=BQ+R$ ,  $0 \leq R < |B|$ .

```
1080 \def\xintiDivision {\romannumeral0\xintiDivision}%
1081 \def\xintiDivision #1{\expandafter\XINT_iidivision \romannumeral`&&#1\xint:}%
1082 \def\XINT_iidivision #1#2\xint:#3{\expandafter\XINT_iidivision_a\expandafter #1%
1083 \romannumeral`&&#3\xint:#2\xint:}%
```

On regarde les signes de A et de B.

```
1084 \def\XINT_iidivision_a #1#2% #1 de A, #2 de B.
1085 {%
1086 \if0#2\xint_dothis{\XINT_iidivision_divbyzero #1#2}\fi
1087 \if0#1\xint_dothis\XINT_iidivision_aiszero\fi
1088 \if-#2\xint_dothis{\expandafter\XINT_iidivision_bneg
1089 \romannumeral0\XINT_iidivision_bpos #1}\fi
1090 \xint_orthat{\XINT_iidivision_bpos #1#2}%
1091}%
1092 \def\XINT_iidivision_divbyzero#1#2#3\xint:#4\xint:
1093 {\if0#1\xint_dothis{\XINT_signalcondition{DivisionUndefined}}\fi
1094 \xint_orthat{\XINT_signalcondition{DivisionByZero}}%
1095 {Division by zero: #1#4/#2#3.}\{\{0\}\{0\}}}%
1096 \def\XINT_iidivision_aiszero #1\xint:#2\xint:{\{0\}\{0\}}%
1097 \def\XINT_iidivision_bneg #1% q->-q, r unchanged
1098 {\expandafter{\romannumeral0\XINT_opp #1}}%
1099 \def\XINT_iidivision_bpos #1%
1100 {%
1101 \xint_UDsignfork
1102 #1\XINT_iidivision_aneg
1103 -{\XINT_iidivision_apos #1}%
1104 \krof
1105}%
```

Donc attention malgré son nom `\XINT_div_prepare` va jusqu'au bout. C'est donc en fait l'entrée principale (pour  $B > 0$ ,  $A > 0$ ) mais elle va regarder si B est  $< 10^8$  et s'il vaut alors 1 ou 2, et si  $A < 10^8$ . Dans tous les cas le résultat est produit sous la forme  $\{Q\}\{R\}$ , avec Q et R sous leur forme final. On doit ensuite ajuster si le B ou le A initial était négatif. Je n'ai pas fait beaucoup d'efforts pour être un minimum efficace si A ou B n'est pas positif.

```
1106 \def\XINT_iidivision_apos #1#2\xint:#3\xint:{\XINT_div_prepare {#2}\{#1#3}}%
1107 \def\XINT_iidivision_aneg #1\xint:#2\xint:
1108 {\expandafter
1109 \XINT_iidivision_aneg_b\romannumeral0\XINT_div_prepare {#1}\{#2}\{#1}}%
1110 \def\XINT_iidivision_aneg_b #1#2{\if0\XINT_Sgn #2\xint:
1111 \expandafter\XINT_iidivision_aneg_rzero
1112 \else
1113 \expandafter\XINT_iidivision_aneg_rpos
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1114             \fi {#1}{#2}}%
1115 \def\XINT_iidivision_aneg_rzero #1#2#3{{-#1}{0}}% necessarily q was >0
1116 \def\XINT_iidivision_aneg_rpos #1%
1117 {%
1118     \expandafter\XINT_iidivision_aneg_end\expandafter
1119         {\expandafter-\romannumeral0\xintinc {#1}}% q-> -(1+q)
1120 }%
1121 \def\XINT_iidivision_aneg_end #1#2#3%
1122 {%
1123     \expandafter\xint_exchangetwo_keepbraces
1124     \expandafter{\romannumeral0\XINT_sub_mm_a {}{}#3\xint:#2\xint:}{#1}% r-> b-r
1125 }%

```

Le diviseur B va être étendu par des zéros pour que sa longueur soit multiple de huit. Les zéros seront mis du côté non significatif.

```

1126 \def\XINT_div_prepare #1%
1127 {%
1128     \XINT_div_prepare_a #1\R\R\R\R\R\R\R {10}0000001\W !{#1}%
1129 }%
1130 \def\XINT_div_prepare_a #1#2#3#4#5#6#7#8#9%
1131 {%
1132     \xint_gob_til_R #9\XINT_div_prepare_small\R
1133     \XINT_div_prepare_b #9%
1134 }%

```

B a au plus huit chiffres. On se débarrasse des trucs superflus. Si  $B > 0$  n'est ni 1 ni 2, le point d'entrée est `\XINT_div_small_a {B}{A}` (avec un A positif).

```

1135 \def\XINT_div_prepare_small\R #1!#2%
1136 {%
1137     \ifcase #2
1138     \or\expandafter\XINT_div_BisOne
1139     \or\expandafter\XINT_div_BisTwo
1140     \else\expandafter\XINT_div_small_a
1141     \fi {#2}%
1142 }%
1143 \def\XINT_div_BisOne #1#2{{#2}{0}}%
1144 \def\XINT_div_BisTwo #1#2%
1145 {%
1146     \expandafter\expandafter\expandafter\XINT_div_BisTwo_a
1147     \ifodd\xintLDg{#2} \expandafter1\else \expandafter0\fi {#2}%
1148 }%
1149 \def\XINT_div_BisTwo_a #1#2%
1150 {%
1151     \expandafter{\romannumeral0\XINT_half
1152         #2\xint_bye\xint_Bye345678\xint_bye
1153         *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax}{#1}%
1154 }%

```

B a au plus huit chiffres et est au moins 3. On va l'utiliser directement, sans d'abord le multiplier par une puissance de 10 pour qu'il ait 8 chiffres.

```

1155 \def\XINT_div_small_a #1#2%
1156 {%
1157     \expandafter\XINT_div_small_b
1158     \the\numexpr #1/\xint_c_ii\expandafter

```



## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
1159 \xint:\the\numexpr \xint_c_x^viii+#1\expandafter!%
1160 \romannumeral0%
1161 \XINT_div_small_ba #2\R\R\R\R\R\R\R\R{10}0000001\W
1162 #2\XINT_sepbyviii_Z_end 2345678\relax
1163 }%
```

Le #2 poursuivra l'expansion par `\XINT_div_dosmallsmall` ou par `\XINT_smalldivx_a` suivi de `\XINT_sdiv_out`.

```
1164 \def\XINT_div_small_b #1!#2{#2#1!}%
```

On ajoute des zéros avant A, puis on le prépare sous la forme de blocs 1<8d>! Au passage on repère le cas d'un  $A < 10^8$ .

```
1165 \def\XINT_div_small_ba #1#2#3#4#5#6#7#8#9%
1166 {%
1167 \xint_gob_til_R #9\XINT_div_smallsmall\R
1168 \expandafter\XINT_div_dosmalldiv
1169 \the\numexpr\expandafter\XINT_sepbyviii_Z
1170 \romannumeral0\XINT_zeroes_forviii
1171 #1#2#3#4#5#6#7#8#9%
1172 }%
```

Si  $A < 10^8$ , on va poursuivre par `\XINT_div_dosmallsmall round(B/2).10^8+B!`{A}. On fait la division directe par `\numexpr`. Le résultat est produit sous la forme {Q}{R}.

```
1173 \def\XINT_div_smallsmall\R
1174 \expandafter\XINT_div_dosmalldiv
1175 \the\numexpr\expandafter\XINT_sepbyviii_Z
1176 \romannumeral0\XINT_zeroes_forviii #1\R #2\relax
1177 {{\XINT_div_dosmallsmall}}{#1}}%
1178 \def\XINT_div_dosmallsmall #1\xint:1#2!#3%
1179 {%
1180 \expandafter\XINT_div_smallsmallend
1181 \the\numexpr (#3+#1)/#2-\xint_c_i\xint:#2\xint:#3\xint:%
1182 }%
```

```
1183 \def\XINT_div_smallsmallend #1\xint:#2\xint:#3\xint:{\expandafter
1184 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #3-#1*#2}}%
```

Si  $A \geq 10^8$ , il est maintenant sous la forme 1<8d>!\...1<8d>!1;! avec plus significatifs en premier. Donc on poursuit par

`\expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a x.1B!1<8d>!\...1<8d>!1;! avec  $x = \text{round}(B/2)$ ,  $1B = 10^8 + B$ .`

```
1185 \def\XINT_div_dosmalldiv
1186 {{\expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a}}%
```

Ici B est au moins  $10^8$ , on détermine combien de zéros lui adjoindre pour qu'il soit de longueur 8N.

```
1187 \def\XINT_div_prepare_b
1188 {\expandafter\XINT_div_prepare_c\romannumeral0\XINT_zeroes_forviii }%
1189 \def\XINT_div_prepare_c #1!%
1190 {%
1191 \XINT_div_prepare_d #1.00000000!{#1}%
1192 }%
1193 \def\XINT_div_prepare_d #1#2#3#4#5#6#7#8#9%
1194 {%
1195 \expandafter\XINT_div_prepare_e\xint_gob_til_dot #1#2#3#4#5#6#7#8#9!%
1196 }%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1197 \def\XINT_div_prepare_e #1!#2!#3#4%
1198 {%
1199     \XINT_div_prepare_f #4#3\X {#1}{#3}%
1200 }%
```

attention qu'on calcule ici  $x'=x+1$  ( $x$  = huit premiers chiffres du diviseur) et que si  $x=99999999$ ,  $x'$  aura donc 9 chiffres, pas compatible avec `div_mini` (avant 1.2,  $x$  avait 4 chiffres, et on faisait la division avec  $x'$  dans un `\numexpr`). Bon, facile à dire après avoir laissé passer ce bug dans 1.2. C'est le problème lorsqu'au lieu de tout refaire à partir de zéro on recycle d'anciennes routines qui avaient un contexte différent.

```
1201 \def\XINT_div_prepare_f #1#2#3#4#5#6#7#8#9\X
1202 {%
1203     \expandafter\XINT_div_prepare_g
1204     \the\numexpr #1#2#3#4#5#6#7#8+\xint_c_i\expandafter
1205     \xint:\the\numexpr (#1#2#3#4#5#6#7#8+\xint_c_i)/\xint_c_ii\expandafter
1206     \xint:\the\numexpr #1#2#3#4#5#6#7#8\expandafter
1207     \xint:\romannumeral0\XINT_sepdandrev_andcount
1208     #1#2#3#4#5#6#7#8#9\XINT_rsepybviii_end_A 2345678%
1209     \XINT_rsepybviii_end_B 2345678\relax\xint_c_ii\xint_c_i
1210     \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
1211     \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
1212     \X
1213 }%
1214 \def\XINT_div_prepare_g #1\xint:#2\xint:#3\xint:#4\xint:#5\X #6#7#8%
1215 {%
1216     \expandafter\XINT_div_prepare_h
1217     \the\numexpr\expandafter\XINT_sepybviii_andcount
1218     \romannumeral0\XINT_zeroes_forviii #8#7\R\R\R\R\R\R\R\{10}0000001\W
1219     #8#7\XINT_sepybviii_end 2345678\relax
1220     \xint_c_vii!\xint_c_vi!\xint_c_v!\xint_c_iv!%
1221     \xint_c_iii!\xint_c_ii!\xint_c_i!\xint_c_\W
1222     {#1}{#2}{#3}{#4}{#5}{#6}%
1223 }%
1224 \def\XINT_div_prepare_h #11\xint:#2\xint:#3#4#5#6%#7#8%
1225 {%
1226     \XINT_div_start_a {#2}{#6}{#1}{#3}{#4}{#5}%{#7}{#8}%
1227 }%
```

L, K, A,  $x'$ , y, x, B, «c». Attention que K est diminué de 1 plus loin. Comme xint 1.2 a déjà repéré  $K=1$ , on a ici au minimum  $K=2$ . Attention B est à l'envers, A est à l'endroit et les deux avec séparateurs. Attention que ce n'est pas ici qu'on boucle mais en `\XINT_div_I_a`.

```
1228 \def\XINT_div_start_a #1#2%
1229 {%
1230     \ifnum #1 < #2
1231         \expandafter\XINT_div_zeroQ
1232     \else
1233         \expandafter\XINT_div_start_b
1234     \fi
1235     {#1}{#2}%
1236 }%
1237 \def\XINT_div_zeroQ #1#2#3#4#5#6#7%
1238 {%
1239     \expandafter\XINT_div_zeroQ_end
1240     \romannumeral0\XINT_unsep_cuzsmall
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1241      #3\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\xint:
1242 }%
1243 \def\xINT_div_zeroQ_end #1\xint:#2%
1244   {\expandafter{\expandafter0\expandafter}\XINT_div_cleanR #1#2\xint:}%
      L, K, A, x', y, x, B, «c»->K.A.x{LK{x'y}x}B«c»
1245 \def\xINT_div_start_b #1#2#3#4#5#6%
1246 {%
1247   \expandafter\xINT_div_finish\the\numexpr
1248   \XINT_div_start_c {#2}\xint:#3\xint:{#6}{#{1}{#2}{#4}{#5}}{#6}}%
1249 }%
1250 \def\xINT_div_finish
1251 {%
1252   \expandafter\xINT_div_finish_a \romannumeral`&&\xINT_div_unsepQ
1253 }%
1254 \def\xINT_div_finish_a #1Z #2\xint:{\XINT_div_finish_b #2\xint:{#1}}%
      Ici ce sont routines de fin. Le reste déjà nettoyé. R.Q«c».
1255 \def\xINT_div_finish_b #1%
1256 {%
1257   \if0#1%
1258     \expandafter\xINT_div_finish_bRzero
1259   \else
1260     \expandafter\xINT_div_finish_bRpos
1261   \fi
1262   #1%
1263 }%
1264 \def\xINT_div_finish_bRzero 0\xint:#1#2{#{1}{0}}%
1265 \def\xINT_div_finish_bRpos #1\xint:#2#3%
1266 {%
1267   \expandafter\xint_exchangetwo_keepbraces\xINT_div_cleanR #1#3\xint:{#2}%
1268 }%
1269 \def\xINT_div_cleanR #1000000000\xint:{#{1}}%
      Kalpha.A.x{LK{x'y}x}, B, «c», au début #2=alpha est vide. On fait une boucle pour prendre K unités
      de A (on a au moins L égal à K) et les mettre dans alpha.
1270 \def\xINT_div_start_c #1%
1271 {%
1272   \ifnum #1>\xint_c_vi
1273     \expandafter\xINT_div_start_ca
1274   \else
1275     \expandafter\xINT_div_start_cb
1276   \fi {#1}%
1277 }%
1278 \def\xINT_div_start_ca #1#2\xint:#3!#4!#5!#6!#7!#8!#9!%
1279 {%
1280   \expandafter\xINT_div_start_c\expandafter
1281   {\the\numexpr #1-\xint_c_vii}{#2#3!#4!#5!#6!#7!#8!#9!\xint:%
1282 }%
1283 \def\xINT_div_start_cb #1%
1284   {\csname XINT_div_start_c\romannumeral\numexpr#1\endcsname}%
1285 \def\xINT_div_start_c_i #1\xint:#2!%
1286   {\XINT_div_start_c_ #1#2!\xint:}%
1287 \def\xINT_div_start_c_ii #1\xint:#2!#3!%

```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1288     {\XINT_div_start_c_ #1#2!#3!\xint:}%
1289 \def\XINT_div_start_c_iii #1\xint:#2!#3!#4!%
1290     {\XINT_div_start_c_ #1#2!#3!#4!\xint:}%
1291 \def\XINT_div_start_c_iv #1\xint:#2!#3!#4!#5!%
1292     {\XINT_div_start_c_ #1#2!#3!#4!#5!\xint:}%
1293 \def\XINT_div_start_c_v #1\xint:#2!#3!#4!#5!#6!%
1294     {\XINT_div_start_c_ #1#2!#3!#4!#5!#6!\xint:}%
1295 \def\XINT_div_start_c_vi #1\xint:#2!#3!#4!#5!#6!#7!%
1296     {\XINT_div_start_c_ #1#2!#3!#4!#5!#6!#7!\xint:}%

#1=a, #2=alpha (de longueur K, à l'endroit).#3=reste de A.#4=x, #5={LK{x'y}x},#6=B,«c» -> a, x,
alpha, B, {00000000}, L, K, {x'y},x, alpha'=reste de A, B«c».

1297 \def\XINT_div_start_c_ 1#1!#2\xint:#3\xint:#4#5#6%
1298 {%
1299     \XINT_div_I_a {#1}{#4}{1#1!#2}{#6}{00000000}#5{#3}{#6}%
1300 }%

Ceci est le point de retour de la boucle principale. a, x, alpha, B, q0, L, K, {x'y}, x, alpha',
B«c»

1301 \def\XINT_div_I_a #1#2%
1302 {%
1303     \expandafter\XINT_div_I_b\the\numexpr #1/#2\xint:{#1}{#2}%
1304 }%
1305 \def\XINT_div_I_b #1%
1306 {%
1307     \xint_gob_til_zero #1\XINT_div_I_czero 0\XINT_div_I_c #1%
1308 }%

On intercepte petit quotient nul: #1=a, x, alpha, B, #5=q0, L, K, {x'y}, x, alpha', B«c» -> on
lâche un q puis {alpha} L, K, {x'y}, x, alpha', B«c».

1309 \def\XINT_div_I_czero 0\XINT_div_I_c 0\xint:#1#2#3#4#5{1#5\XINT_div_I_g {#3}}%
1310 \def\XINT_div_I_c #1\xint:#2#3%
1311 {%
1312     \expandafter\XINT_div_I_da\the\numexpr #2-#1*#3\xint:#1\xint:{#2}{#3}%
1313 }%

r.q.alpha, B, q0, L, K, {x'y}, x, alpha', B«c»

1314 \def\XINT_div_I_da #1\xint:%
1315 {%
1316     \ifnum #1>\xint_c_ix
1317         \expandafter\XINT_div_I_dP
1318     \else
1319         \ifnum #1<\xint_c_
1320             \expandafter\expandafter\expandafter\XINT_div_I_dN
1321         \else
1322             \expandafter\expandafter\expandafter\XINT_div_I_db
1323         \fi
1324     \fi
1325 }%

attention très mauvaises notations avec _b et _db.

1326 \def\XINT_div_I_dN #1\xint:%
1327 {%
1328     \expandafter\XINT_div_I_b\the\numexpr #1-\xint_c_i\xint:%
1329 }%
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1330 \def\XINT_div_I_db #1\xint:#2#3#4#5%
1331 {%
1332     \expandafter\XINT_div_I_dc\expandafter #1%
1333     \romannumeral0\expandafter\XINT_div_sub\expandafter
1334         {\romannumeral0\XINT_rev_nounsep }#4\R!\R!\R!\R!\R!\R!\R!\R!\W}%
1335     {\the\numexpr\XINT_div_verysmallmul #1!#51;!}%
1336     \Z {#4}{#5}%
1337 }%

    La soustraction spéciale renvoie simplement - si le chiffre q est trop grand. On invoque dans ce
    cas I_dP.

1338 \def\XINT_div_I_dc #1#2%
1339 {%
1340     \if-#2\expandafter\XINT_div_I_dd\else\expandafter\XINT_div_I_de\fi
1341     #1#2%
1342 }%
1343 \def\XINT_div_I_dd #1-\Z
1344 {%
1345     \if #11\expandafter\XINT_div_I_dz\fi
1346     \expandafter\XINT_div_I_dP\the\numexpr #1-\xint_c_i\xint: XX%
1347 }%
1348 \def\XINT_div_I_dz #1XX#2#3#4%
1349 {%
1350     1#4\XINT_div_I_g {#2}%
1351 }%
1352 \def\XINT_div_I_de #1#2\Z #3#4#5{1#5+#1\XINT_div_I_g {#2}}%
    q.alpha, B, q0, L, K, {x'y},x, alpha'B«c» (q=0 has been intercepted) -> 1nouveauq.nouvel alpha,
    L, K, {x'y}, x, alpha',B«c»
1353 \def\XINT_div_I_dP #1\xint:#2#3#4#5#6%
1354 {%
1355     1#6+#1\expandafter\XINT_div_I_g\expandafter
1356     {\romannumeral0\expandafter\XINT_div_sub\expandafter
1357         {\romannumeral0\XINT_rev_nounsep }#4\R!\R!\R!\R!\R!\R!\R!\R!\W}%
1358     {\the\numexpr\XINT_div_verysmallmul #1!#51;!}%
1359 }%
1360 }%

    1#1=nouveau q. nouvel alpha, L, K, {x'y},x,alpha', BQ«c»

    #1=q,#2=nouvel alpha,#3=L, #4=K, #5={x'y}, #6=x, #7= alpha',#8=B, «c» -> on laisse q puis
    {x'y}alpha.alpha'.{{x'y}xKL}B«c»
1361 \def\XINT_div_I_g #1#2#3#4#5#6#7%
1362 {%
1363     \expandafter !\the\numexpr
1364     \ifnum#2=#3
1365         \expandafter\XINT_div_exittofinish
1366     \else
1367         \expandafter\XINT_div_I_h
1368     \fi
1369     {#4}#1\xint:#6\xint:{{#4}{#5}{#3}{#2}}{#7}%
1370 }%

    {x'y}alpha.alpha'.{{x'y}xKL}B«c» -> Attention retour à l'envoyeur ici par terminaison des \the\,
    numexpr. On doit reprendre le Q déjà sorti, qui n'a plus de séparateurs, ni de leading 1. Ensuite
    R sans leading zeros.«c»

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

1371 \def\XINT_div_exittofinish #1#2\xint:#3\xint:#4#5%
1372 {%
1373     1\expandafter\expandafter\expandafter!\expandafter\XINT_div_unsepQ_delim
1374     \romannumeral0\XINT_div_unsepR #2#3%
1375     \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax\R\xint:
1376 }%
    ATTENTION DESCRIPTION OBSOLÈTE. #1={x'y}alpha.#2!#3=reste de A. #4={{x'y},x,K,L},#5=B,«c» de-
    vient {x'y},alpha sur K+4 chiffres.B, {{x'y},x,K,L}, #6= nouvel alpha',B,«c»
1377 \def\XINT_div_I_h #1\xint:#2!#3\xint:#4#5%
1378 {%
1379     \XINT_div_II_b #1#2!\xint:{#5}{#4}{#3}{#5}%
1380 }%
    {x'y}alpha.B, {{x'y},x,K,L}, nouveau alpha',B,«c»
1381 \def\XINT_div_II_b #1#2!#3!%
1382 {%
1383     \xint_gob_til_eightzeroes #2\XINT_div_II_skipc 00000000%
1384     \XINT_div_II_c #1{1#2}{#3}%
1385 }%
    x'y{100000000}{1<8>}reste de alpha.#6=B,#7={{x'y},x,K,L}, alpha',B, «c» -> {x'y}x,K,L (à dimin-
    uer de 4), {alpha sur K}B{q1=00000000}{alpha'}B,«c»
1386 \def\XINT_div_II_skipc 00000000\XINT_div_II_c #1#2#3#4#5\xint:#6#7%
1387 {%
1388     \XINT_div_II_k #7{#4!#5}{#6}{00000000}%
1389 }%
    x'ya->1qx'yalpha.B, {{x'y},x,K,L}, nouveau alpha',B, «c». En fait, attention, ici #3 et #4 sont
    les 16 premiers chiffres du numérateur,sous la forme blocs 1<8chiffres>.
1390 \def\XINT_div_II_c #1#2#3#4%
1391 {%
1392     \expandafter\XINT_div_II_d\the\numexpr\XINT_div_xmini
1393     #1\xint:#2!#3!#4!{#1}{#2}#3!#4!%
1394 }%
1395 \def\XINT_div_xmini #1%
1396 {%
1397     \xint_gob_til_one #1\XINT_div_xmini_a 1\XINT_div_mini #1%
1398 }%
1399 \def\XINT_div_xmini_a 1\XINT_div_mini 1#1%
1400 {%
1401     \xint_gob_til_zero #1\XINT_div_xmini_b 0\XINT_div_mini 1#1%
1402 }%
1403 \def\XINT_div_xmini_b 0\XINT_div_mini 10#1#2#3#4#5#6#7%
1404 {%
1405     \xint_gob_til_zero #7\XINT_div_xmini_c 0\XINT_div_mini 10#1#2#3#4#5#6#7%
1406 }%
    x'=10^8 and we return #1=1<8digits>.
1407 \def\XINT_div_xmini_c 0\XINT_div_mini 100000000\xint:50000000!#1!#2!{#1!}%
    1 suivi de q1 sur huit chiffres! #2=x', #3=y, #4=alpha.#5=B, {{x'y},x,K,L}, alpha', B, «c» -->
    nouvel alpha.x',y,B,q1,{{x'y},x,K,L}, alpha', B, «c»
1408 \def\XINT_div_II_d 1#1#2#3#4#5!#6#7#8\xint:#9%
1409 {%
1410     \expandafter\XINT_div_II_e

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1411 \romannumeral0\expandafter\XINT_div_sub\expandafter
1412 { \romannumeral0\XINT_rev_nounsep {}#8\R!\R!\R!\R!\R!\R!\R!\R!\W}%
1413 { \the\numexpr\XINT_div_smallmul_a 100000000\mint:#1#2#3#4\mint:#5!#91;!}%
1414 \mint:{#6}{#7}{#9}{#1#2#3#4#5}%
1415 }%

alpha.x',y,B,q1, {{x'y},x,K,L}, alpha', B, «c». Attention la soustraction spéciale doit main-
tenir les blocs 1<8>!

1416 \def\XINT_div_II_e 1#1!%
1417 {%
1418 \mint_gob_til_eightzeroes #1\XINT_div_II_skipf 00000000%
1419 \XINT_div_II_f 1#1!%
1420 }%

100000000! alpha sur K chiffres.#2=x',#3=y,#4=B,#5=q1, #6={{x'y},x,K,L}, #7=alpha',B«c» ->
{x'y}x,K,L (à diminuer de 1), {alpha sur K}B{q1}{alpha'}B«c»

1421 \def\XINT_div_II_skipf 00000000\XINT_div_II_f 100000000!#1\mint:#2#3#4#5#6%
1422 {%
1423 \XINT_div_II_k #6{#1}{#4}{#5}%
1424 }%

1<a1>!1<a2>!, alpha (sur K+1 blocs de 8). x', y, B, q1, {{x'y},x,K,L}, alpha', B,«c».
Here also we are dividing with x' which could be 10^8 in the exceptional case x=99999999. Must
intercept it before sending to \XINT_div_mini.

1425 \def\XINT_div_II_f #1!#2!#3\mint:%
1426 {%
1427 \XINT_div_II_fa {#1!#2!}{#1!#2!#3}%
1428 }%
1429 \def\XINT_div_II_fa #1#2#3#4%
1430 {%
1431 \expandafter\XINT_div_II_g \the\numexpr\XINT_div_xmini #3\mint:#4!#1#2}%
1432 }%

#1=q, #2=alpha (K+4), #3=B, #4=q1, {{x'y},x,K,L}, alpha', BQ«c» -> 1 puis nouveau q sur 8
chiffres. nouvel alpha sur K blocs, B, {{x'y},x,K,L}, alpha',B«c»

1433 \def\XINT_div_II_g 1#1#2#3#4#5!#6#7#8%
1434 {%
1435 \expandafter \XINT_div_II_h
1436 \the\numexpr 1#1#2#3#4#5+#8\expandafter\expandafter\expandafter
1437 \mint:\expandafter\expandafter\expandafter
1438 {\expandafter\mint_gob_til_exclam
1439 \romannumeral0\expandafter\XINT_div_sub\expandafter
1440 { \romannumeral0\XINT_rev_nounsep {}#6\R!\R!\R!\R!\R!\R!\R!\R!\W}%
1441 { \the\numexpr\XINT_div_smallmul_a 100000000\mint:#1#2#3#4\mint:#5!#71;!}%
1442 {#7}%
1443 }%

1 puis nouveau q sur 8 chiffres, #2=nouvel alpha sur K blocs, #3=B, #4={{x'y},x,K,L} avec L à
ajuster, alpha', BQ«c» -> {x'y}x,K,L à diminuer de 1, {alpha}B{q}, alpha', BQ«c»

1444 \def\XINT_div_II_h 1#1\mint:#2#3#4%
1445 {%
1446 \XINT_div_II_k #4{#2}{#3}{#1}%
1447 }%

{x'y}x,K,L à diminuer de 1, alpha, B{q}alpha',B«c» ->nouveau L.K,x',y,x,alpha.B,q,alpha',B,«c»
->{LK{x'y}x},x,a,alpha.B,q,alpha',B,«c»

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1448 \def\XINT_div_II_k #1#2#3#4#5%
1449 {%
1450   \expandafter\XINT_div_II_l \the\numexpr #4-\xint_c_i\xint:{#3}#1{#2}#5\xint:%
1451 }%
1452 \def\XINT_div_II_l #1\xint:#2#3#4#51#6!%
1453 {%
1454   \XINT_div_II_m {{#1}{#2}{{#3}{#4}{{#5}{{#5}{{#6}1#6!%
1455 }%

      {LK{x'y}x},x,a,alpha.B{q}alpha'B -> a, x, alpha, B, q, L, K, {x'y}, x, alpha', B<c>
1456 \def\XINT_div_II_m #1#2#3#4\xint:#5#6%
1457 {%
1458   \XINT_div_I_a {#3}{#2}{#4}{#5}{#6}#1%
1459 }%

      This multiplication is exactly like \XINT_smallmul -- apart from not inserting an ending 1;!--,
      but keeps ever a vanishing ending carry.
1460 \def\XINT_div_minimulwc_a 1#1\xint:#2\xint:#3!#4#5#6#7#8\xint:%
1461 {%
1462   \expandafter\XINT_div_minimulwc_b
1463   \the\numexpr \xint_c_x^ix+#1+#3*#8\xint:#3*#4#5#6#7+#2*#8\xint:#2*#4#5#6#7\xint:%
1464 }%
1465 \def\XINT_div_minimulwc_b 1#1#2#3#4#5#6\xint:#7\xint:%
1466 {%
1467   \expandafter\XINT_div_minimulwc_c
1468   \the\numexpr \xint_c_x^ix+#1#2#3#4#5+#7\xint:#6\xint:%
1469 }%
1470 \def\XINT_div_minimulwc_c 1#1#2#3#4#5#6\xint:#7\xint:#8\xint:%
1471 {%
1472   1#6#7\expandafter!%
1473   \the\numexpr\expandafter\XINT_div_smallmul_a
1474   \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#8\xint:%
1475 }%
1476 \def\XINT_div_smallmul_a #1\xint:#2\xint:#3!1#4!%
1477 {%
1478   \xint_gob_til_sc #4\XINT_div_smallmul_e;%
1479   \XINT_div_minimulwc_a #1\xint:#2\xint:#3!#4\xint:#2\xint:#3!%
1480 }%
1481 \def\XINT_div_smallmul_e;\XINT_div_minimulwc_a 1#1\xint:#2;#3!{1\relax #1!}%

      Special very small multiplication for division. We only need to cater for multiplicands from 1
      to 9. The ending is different from standard verysmallmul, a zero carry is not suppressed. And no
      final 1;!-- is added. If multiplicand is just 1 let's not forget to add the zero carry 100000000! at
      the end.
1482 \def\XINT_div_verysmallmul #1%
1483   {\xint_gob_til_one #1\XINT_div_verysmallisone 1\XINT_div_verysmallmul_a 0\xint:#1}%
1484 \def\XINT_div_verysmallisone 1\XINT_div_verysmallmul_a 0\xint:1!1#11;!--%
1485   {1\relax #1100000000!}%
1486 \def\XINT_div_verysmallmul_a #1\xint:#2!1#3!%
1487 {%
1488   \xint_gob_til_sc #3\XINT_div_verysmallmul_e;%
1489   \expandafter\XINT_div_verysmallmul_b
1490   \the\numexpr \xint_c_x^ix+#2*#3+#1\xint:#2!%
1491 }%

```



## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1492 \def\XINT_div_verysmallmul_b 1#1#2\xint:%
1493   {1#2\expandafter!\the\numexpr\XINT_div_verysmallmul_a #1\xint:}%
1494 \def\XINT_div_verysmallmul_e;#1;+#2#3!{1\relax 0000000#2!}%
Special subtraction for division purposes. If the subtracted thing turns out to be bigger, then
just return a -. If not, then we must reverse the result, keeping the separators.
1495 \def\XINT_div_sub #1#2%
1496 {%
1497   \expandafter\XINT_div_sub_clean
1498   \the\numexpr\expandafter\XINT_div_sub_a\expandafter
1499   1#2;!!;!!;!!\W #1;!!;!!;!!\W
1500 }%
1501 \def\XINT_div_sub_clean #1-#2#3\W
1502 {%
1503   \if1#2\expandafter\XINT_rev_nounsep\else\expandafter\XINT_div_sub_neg\fi
1504   {}#1\R!\R!\R!\R!\R!\R!\R!\R!\W
1505 }%
1506 \def\XINT_div_sub_neg #1\W { -}%
1507 \def\XINT_div_sub_a #1!#2!#3!#4!#5\W #6!#7!#8!#9!%
1508 {%
1509   \XINT_div_sub_b #1!#6!#2!#7!#3!#8!#4!#9!#5\W
1510 }%
1511 \def\XINT_div_sub_b #1#2#3!#4!%
1512 {%
1513   \xint_gob_til_sc #4\XINT_div_sub_bi ;%
1514   \expandafter\XINT_div_sub_c\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
1515 }%
1516 \def\XINT_div_sub_c 1#1#2\xint:%
1517 {%
1518   1#2\expandafter!\the\numexpr\XINT_div_sub_d #1%
1519 }%
1520 \def\XINT_div_sub_d #1#2#3!#4!%
1521 {%
1522   \xint_gob_til_sc #4\XINT_div_sub_di ;%
1523   \expandafter\XINT_div_sub_e\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
1524 }%
1525 \def\XINT_div_sub_e 1#1#2\xint:%
1526 {%
1527   1#2\expandafter!\the\numexpr\XINT_div_sub_f #1%
1528 }%
1529 \def\XINT_div_sub_f #1#2#3!#4!%
1530 {%
1531   \xint_gob_til_sc #4\XINT_div_sub_fi ;%
1532   \expandafter\XINT_div_sub_g\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
1533 }%
1534 \def\XINT_div_sub_g 1#1#2\xint:%
1535 {%
1536   1#2\expandafter!\the\numexpr\XINT_div_sub_h #1%
1537 }%
1538 \def\XINT_div_sub_h #1#2#3!#4!%
1539 {%
1540   \xint_gob_til_sc #4\XINT_div_sub_hi ;%
1541   \expandafter\XINT_div_sub_i\the\numexpr#1-#3+1#4-\xint_c_i\xint:%
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

1542 }%
1543 \def\XINT_div_sub_i 1#1#2\xint:%
1544 {%
1545     1#2\expandafter!\the\numexpr\XINT_div_sub_a #1%
1546 }%
1547 \def\XINT_div_sub_bi;%
1548     \expandafter\XINT_div_sub_c\the\numexpr#1-#2+#3\xint:#4!#5!#6!#7!#8!#9!;! \W
1549 {%
1550     \XINT_div_sub_l #1#2!#5!#7!#9!%
1551 }%
1552 \def\XINT_div_sub_di;%
1553     \expandafter\XINT_div_sub_e\the\numexpr#1-#2+#3\xint:#4!#5!#6!#7!#8\W
1554 {%
1555     \XINT_div_sub_l #1#2!#5!#7!%
1556 }%
1557 \def\XINT_div_sub_fi;%
1558     \expandafter\XINT_div_sub_g\the\numexpr#1-#2+#3\xint:#4!#5!#6\W
1559 {%
1560     \XINT_div_sub_l #1#2!#5!%
1561 }%
1562 \def\XINT_div_sub_hi;%
1563     \expandafter\XINT_div_sub_i\the\numexpr#1-#2+#3\xint:#4\W
1564 {%
1565     \XINT_div_sub_l #1#2!%
1566 }%
1567 \def\XINT_div_sub_l #1%
1568 {%
1569     \xint_UDzerofork
1570     #1{-2\relax}%
1571     0\XINT_div_sub_r
1572     \krof
1573 }%
1574 \def\XINT_div_sub_r #1!%
1575 {%
1576     -\ifnum 0#1=\xint_c_ 1\else2\fi\relax
1577 }%

```

Ici  $B < 10^8$  (et est  $> 2$ ). On exécute

`\expandafter\XINT_sdiv_out\the\numexpr\XINT_smalldivx_a x.1B!1<8d>!\dots1<8d>!1;!`

avec  $x = \text{round}(B/2)$ ,  $1B = 10^8 + B$ , et A déjà en blocs  $1<8d>!$  (non renversés). Le `\the\numexpr\XINT_smalldivx_a` va produire  $Q \setminus Z \setminus R \setminus W$  avec un  $R < 10^8$ , et un Q sous forme de blocs  $1<8d>!$  terminé par 1! et nécessitant le nettoyage du premier bloc. Dans cette branche le B n'a pas été multiplié par une puissance de 10, il peut avoir moins de huit chiffres.

```

1578 \def\XINT_sdiv_out #1;#2!%
1579     {\expandafter
1580     {\romannumeral0\XINT_unsep_cuzsmall
1581     #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax}%
1582     {#2}}%

```

La toute première étape fait la première division pour être sûr par la suite d'avoir un premier bloc pour A qui sera  $< B$ .

```

1583 \def\XINT_smalldivx_a #1\xint:1#2!1#3!%
1584 {%
1585     \expandafter\XINT_smalldivx_b

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1586 \the\numexpr (#3+#1)/#2-\xint_c_i!#1\xint:#2!#3!%
1587 }%
1588 \def\xINT_smalldivx_b #1#2!%
1589 {%
1590 \if0#1\else
1591 \xint_c_x^viii+#1#2\xint_afterfi{\expandafter!\the\numexpr}\fi
1592 \xINT_smalldiv_c #1#2!%
1593 }%
1594 \def\xINT_smalldiv_c #1!#2\xint:#3!#4!%
1595 {%
1596 \expandafter\xINT_smalldiv_d\the\numexpr #4-#1*#3!#2\xint:#3!%
1597 }%
```

On va boucler ici: #1 est un reste, #2 est x.B (avec B sans le 1 mais sur huit chiffres). #3#4 est le premier bloc qui reste de A. Si on a terminé avec A, alors #1 est le reste final. Le quotient lui est terminé par un 1! ce 1! disparaîtra dans le nettoyage par `\XINT_unsep_cuzsmall`.

```
1598 \def\xINT_smalldiv_d #1!#2!1#3#4!%
1599 {%
1600 \xint_gob_til_sc #3\xINT_smalldiv_end ;%
1601 \xINT_smalldiv_e #1!#2!1#3#4!%
1602 }%
1603 \def\xINT_smalldiv_end;\xINT_smalldiv_e #1!#2!1;!\{1!;!#1!}%
1604 Il est crucial que le reste #1 est < #3. J'ai documenté cette routine dans le fichier où j'ai
1605 préparé 1.2, il faudra transférer ici. Il n'est pas nécessaire pour cette routine que le diviseur
1606 B ait au moins 8 chiffres. Mais il doit être < 10^8.
1607 \def\xINT_smalldiv_e #1!#2\xint:#3!%
1608 {%
1609 \expandafter\xINT_smalldiv_f\the\numexpr
1610 \xint_c_xi_e_viii_mone+#1*\xint_c_x^viii/#3!#2\xint:#3!#1!%
1611 }%
1612 \def\xINT_smalldiv_f 1#1#2#3#4#5#6!#7\xint:#8!%
1613 {%
1614 \xint_gob_til_zero #1\xINT_smalldiv_fz 0%
1615 \expandafter\xINT_smalldiv_g
1616 \the\numexpr\xINT_minimul_a #2#3#4#5\xint:#6!#8!#2#3#4#5#6!#7\xint:#8!%
1617 }%
1618 \def\xINT_smalldiv_fz 0%
1619 \expandafter\xINT_smalldiv_g\the\numexpr\xINT_minimul_a
1620 9999\xint:9999!#1!99999999!#2!0!1#3!%
1621 {%
1622 \xINT_smalldiv_i \xint:#3!\xint_c_!#2!%
1623 }%
1624 \def\xINT_smalldiv_g 1#1!1#2!#3!#4!#5!#6!%
1625 {%
1626 \expandafter\xINT_smalldiv_h\the\numexpr 1#6-#1\xint:#2!#5!#3!#4!%
1627 }%
1628 \def\xINT_smalldiv_h 1#1#2\xint:#3!#4!%
1629 {%
1630 \expandafter\xINT_smalldiv_i\the\numexpr #4-#3+#1-\xint_c_i\xint:#2!%
1631 }%
1632 \def\xINT_smalldiv_i #1\xint:#2!#3!#4\xint:#5!%
1633 {%
1634 \expandafter\xINT_smalldiv_j\the\numexpr (#1#2+#4)/#5-\xint_c_i!#3!#1#2!#4\xint:#5!%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1632 }%
1633 \def\XINT_smallldiv_j #1!#2!%
1634 {%
1635     \xint_c_x^viii+#1+#2\expandafter!\the\numexpr\XINT_smallldiv_k
1636     #1!%
1637 }%
```

On boucle vers \XINT\_smallldiv\_d.

```
1638 \def\XINT_smallldiv_k #1!#2!#3\xint:#4!%
1639 {%
1640     \expandafter\XINT_smallldiv_d\the\numexpr #2-#1*#4!#3\xint:#4!%
1641 }%
```

Cette routine fait la division euclidienne d'un nombre de seize chiffres par #1 = C = diviseur sur huit chiffres  $\geq 10^7$ , avec #2 = sa moitié utilisée dans \numexpr pour contrebalancer l'arrondi (ARRRRRRGGGGGHHH) fait par /. Le nombre divisé  $XY = X \cdot 10^8 + Y$  se présente sous la forme 1<8chiffres>1<8chiffres>! avec plus significatif en premier.

Seul le quotient est calculé, pas le reste. En effet la routine de division principale va utiliser ce quotient pour déterminer le "grand" reste, et le petit reste ici ne nous serait d'à peu près aucune utilité.

ATTENTION UNIQUEMENT UTILISÉ POUR DES SITUATIONS OÙ IL EST GARANTI QUE  $X < C$  ! (et C au moins  $10^7$ ) le quotient euclidien de  $X \cdot 10^8 + Y$  par C sera donc  $< 10^8$ . Il sera renvoyé sous la forme 1<8chiffres>.

```
1642 \def\XINT_div_mini #1\xint:#2!1#3!%
1643 {%
1644     \expandafter\XINT_div_mini_a\the\numexpr
1645     \xint_c_xi_e_viii_mone+#3*\xint_c_x^viii/#1!#1\xint:#2!#3!%
1646 }%
```

Note (2015/10/08). Attention à la différence dans l'ordre des arguments avec ce que je vois en dans \XINT\_smallldiv\_f. Je ne me souviens plus du tout s'il y a une raison quelconque.

```
1647 \def\XINT_div_mini_a 1#1#2#3#4#5#6!#7\xint:#8!%
1648 {%
1649     \xint_gob_til_zero #1\XINT_div_mini_w 0%
1650     \expandafter\XINT_div_mini_b
1651     \the\numexpr\XINT_minimul_a #2#3#4#5\xint:#6!#7!#2#3#4#5#6!#7\xint:#8!%
1652 }%
1653 \def\XINT_div_mini_w 0%
1654     \expandafter\XINT_div_mini_b\the\numexpr\XINT_minimul_a
1655     9999\xint:9999!#1!99999999!#2\xint:#3!00000000!#4!%
1656 {%
1657     \xint_c_x^viii_mone+(#4+#3)/#2!%
1658 }%
1659 \def\XINT_div_mini_b 1#1!1#2!#3!#4!#5!#6!%
1660 {%
1661     \expandafter\XINT_div_mini_c
1662     \the\numexpr 1#6-#1\xint:#2!#5!#3!#4!%
1663 }%
1664 \def\XINT_div_mini_c 1#1#2\xint:#3!#4!%
1665 {%
1666     \expandafter\XINT_div_mini_d
1667     \the\numexpr #4-#3+#1-\xint_c_i\xint:#2!%
1668 }%
1669 \def\XINT_div_mini_d #1\xint:#2!#3!#4\xint:#5!%
```

```

1670 {%
1671     \xint_c_x^viii_mone+#3+(#1#2+#5)/#4!%
1672 }%

```

## Derived arithmetic

### 20.37. \xintiiQuo, \xintiiRem

```

1673 \def\xintiiQuo {\romannumeral0\xintiiquo }%
1674 \def\xintiiRem {\romannumeral0\xintiirem }%
1675 \def\xintiiquo
1676     {\expandafter\xint_stop_atfirstoftwo\romannumeral0\xintiidivision }%
1677 \def\xintiirem
1678     {\expandafter\xint_stop_atsecondoftwo\romannumeral0\xintiidivision }%

```

### 20.38. \xintiiDivRound

1.1, transferred from first release of bnumexpr. Rewritten for 1.2. Ending rewritten for 1.2i.  
(new \xintDSRr).

1.21: \xintiiDivRound made robust against non terminated input.

```

1679 \def\xintiiDivRound {\romannumeral0\xintiidivround }%
1680 \def\xintiidivround #1{\expandafter\XINT_iidivround\romannumeral`&&@#1\xint:}%
1681 \def\XINT_iidivround #1#2\xint:#3%
1682     {\expandafter\XINT_iidivround_a\expandafter #1\romannumeral`&&@#3\xint:#2\xint:}%
1683 \def\XINT_iidivround_a #1#2% #1 de A, #2 de B.
1684 {%
1685     \if0#2\xint_dothis{\XINT_iidivround_divbyzero#1#2}\fi
1686     \if0#1\xint_dothis\XINT_iidivround_aiszero\fi
1687     \if-#2\xint_dothis{\XINT_iidivround_bneg #1}\fi
1688     \xint_orthat{\XINT_iidivround_bpos #1#2}%
1689 }%
1690 \def\XINT_iidivround_divbyzero #1#2#3\xint:#4\xint:
1691     {\XINT_signalcondition{DivisionByZero}{Division by zero: #1#4/#2#3.}}{0}%
1692 \def\XINT_iidivround_aiszero #1\xint:#2\xint:{0}%
1693 \def\XINT_iidivround_bpos #1%
1694 {%
1695     \xint_UDsignfork
1696         #1{\xintiiopt\XINT_iidivround_pos {}}%
1697         -{\XINT_iidivround_pos #1}%
1698     \krof
1699 }%
1700 \def\XINT_iidivround_bneg #1%
1701 {%
1702     \xint_UDsignfork
1703         #1{\XINT_iidivround_pos {}}%
1704         -{\xintiiopt\XINT_iidivround_pos #1}%
1705     \krof
1706 }%
1707 \def\XINT_iidivround_pos #1#2\xint:#3\xint:
1708 {%
1709     \expandafter\expandafter\expandafter\XINT_dsrr
1710     \expandafter\xint_firstoftwo
1711     \romannumeral0\XINT_div_prepare {#2}{#1#30}%

```

```

1712 \xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax
1713 }%

```

## 20.39. \xintiDivTrunc

1.21: \xintiDivTrunc made robust against non terminated input.

```

1714 \def\xintiDivTrunc {\romannumeral0\xintiDivTrunc }%
1715 \def\xintiDivTrunc #1{\expandafter\XINT_iDivTrunc\romannumeral`&&@#1\xint:}%
1716 \def\XINT_iDivTrunc #1#2\xint:#3{\expandafter\XINT_iDivTrunc_a\expandafter #1%
1717 \romannumeral`&&@#3\xint:#2\xint:}%
1718 \def\XINT_iDivTrunc_a #1#2% #1 de A, #2 de B.
1719 {%
1720 \if0#2\xint_dothis{\XINT_iDivTrunc_divbyzero#1#2}\fi
1721 \if0#1\xint_dothis\XINT_iDivTrunc_aiszero\fi
1722 \if-#2\xint_dothis{\XINT_iDivTrunc_bneg #1}\fi
1723 \xint_orthat{\XINT_iDivTrunc_bpos #1#2}%
1724 }%

```

Attention to not move DivRound code beyond that point.

```

1725 \let\XINT_iDivTrunc_divbyzero\XINT_iDivRound_divbyzero
1726 \let\XINT_iDivTrunc_aiszero \XINT_iDivRound_aiszero
1727 \def\XINT_iDivTrunc_bpos #1%
1728 {%
1729 \xint_UDsignfork
1730 #1{\xintiOpp\XINT_iDivTrunc_pos {}}%
1731 -{\XINT_iDivTrunc_pos #1}%
1732 \krof
1733 }%
1734 \def\XINT_iDivTrunc_bneg #1%
1735 {%
1736 \xint_UDsignfork
1737 #1{\XINT_iDivTrunc_pos {}}%
1738 -{\xintiOpp\XINT_iDivTrunc_pos #1}%
1739 \krof
1740 }%
1741 \def\XINT_iDivTrunc_pos #1#2\xint:#3\xint:
1742 {\expandafter\xint_stop_atfirstoftwo
1743 \romannumeral0\XINT_div_prepare {#2}{#1#3}}%

```

## 20.40. \xintiModTrunc

Renamed from \xintiMod to \xintiModTrunc at 1.2p.

```

1744 \def\xintiModTrunc {\romannumeral0\xintiModTrunc }%
1745 \def\xintiModTrunc #1{\expandafter\XINT_iModTrunc\romannumeral`&&@#1\xint:}%
1746 \def\XINT_iModTrunc #1#2\xint:#3{\expandafter\XINT_iModTrunc_a\expandafter #1%
1747 \romannumeral`&&@#3\xint:#2\xint:}%
1748 \def\XINT_iModTrunc_a #1#2% #1 de A, #2 de B.
1749 {%
1750 \if0#2\xint_dothis{\XINT_iModTrunc_divbyzero#1#2}\fi
1751 \if0#1\xint_dothis\XINT_iModTrunc_aiszero\fi
1752 \if-#2\xint_dothis{\XINT_iModTrunc_bneg #1}\fi
1753 \xint_orthat{\XINT_iModTrunc_bpos #1#2}%
1754 }%

```

Attention to not move DivRound code beyond that point. A bit of abuse here for divbyzero defaulted-to value, which happily works in both.

```

1755 \let\XINT_iimodtrunc_divbyzero\XINT_iidivround_divbyzero
1756 \let\XINT_iimodtrunc_aiszero \XINT_iidivround_aiszero
1757 \def\XINT_iimodtrunc_bpos #1%
1758 {%
1759     \xint_UDsignfork
1760         #1{\xintiiopp\XINT_iimodtrunc_pos {}}%
1761         -{\XINT_iimodtrunc_pos #1}%
1762     \krof
1763 }%
1764 \def\XINT_iimodtrunc_bneg #1%
1765 {%
1766     \xint_UDsignfork
1767         #1{\xintiiopp\XINT_iimodtrunc_pos {}}%
1768         -{\XINT_iimodtrunc_pos #1}%
1769     \krof
1770 }%
1771 \def\XINT_iimodtrunc_pos #1#2\xint:#3\xint:
1772     {\expandafter\xint_stop_atsecondoftwo\romannumeral0\XINT_div_prepare
1773     {#2}{#1#3}}%

```

## 20.41. \xintiiDivMod

Modified at 1.2p (2017/12/05). It is associated with floored division (like Python divmod function), and with the // operator in `\xintiiexpr`.

```

1774 \def\xintiiDivMod {\romannumeral0\xintiiidivmod}%
1775 \def\xintiiidivmod #1{\expandafter\XINT_iidivmod\romannumeral`&&@#1\xint:}%
1776 \def\XINT_iidivmod #1#2\xint:#3{\expandafter\XINT_iidivmod_a\expandafter #1%
1777     \romannumeral`&&@#3\xint:#2\xint:}%
1778 \def\XINT_iidivmod_a #1#2% #1 de A, #2 de B.
1779 {%
1780     \if0#2\xint_dothis{\XINT_iidivmod_divbyzero#1#2}\fi
1781     \if0#1\xint_dothis\XINT_iidivmod_aiszero\fi
1782     \if-#2\xint_dothis{\XINT_iidivmod_bneg #1}\fi
1783     \xint_orthat{\XINT_iidivmod_bpos #1#2}%
1784 }%
1785 \def\XINT_iidivmod_divbyzero #1#2\xint:#3\xint:
1786 {%
1787     \XINT_signalcondition{DivisionByZero}{Division by zero: #1#3/#2.}%}%
1788     {{0}{0}}% à revoir...
1789 }%
1790 \def\XINT_iidivmod_aiszero #1\xint:#2\xint:{{0}{0}}%
1791 \def\XINT_iidivmod_bneg #1%
1792 {%
1793     \expandafter\XINT_iidivmod_bneg_finish
1794     \romannumeral0\xint_UDsignfork
1795         #1{\XINT_iidivmod_bpos {}}%
1796         -{\XINT_iidivmod_bpos {-#1}}%
1797     \krof
1798 }%
1799 \def\XINT_iidivmod_bneg_finish#1#2%

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1800 {%
1801     \expandafter\xint_exchangetwo_keepbraces\expandafter
1802     {\romannumeral0\xintiopp#2}{#1}%
1803 }%
1804 \def\xINT_iidivmod_bpos #1#2\xint:#3\xint:{\xintiidivision{#1#3}{#2}}%
```

### 20.42. \xintiiDivFloor

1.2p. For `bnumexpr` actually, because `\xintiiexpr` could use `\xintDivFloor` which also outputs an integer in strict format.

```
1805 \def\xintiiDivFloor {\romannumeral0\xintiidivfloor}%
1806 \def\xintiidivfloor {\expandafter\xint_stop_atfirstoftwo
1807     \romannumeral0\xintiidivmod}%

```

### 20.43. \xintiiMod

Associated with floored division at 1.2p. Formerly was associated with truncated division.

```
1808 \def\xintiiMod {\romannumeral0\xintiimod}%
1809 \def\xintiimod {\expandafter\xint_stop_atsecondoftwo
1810     \romannumeral0\xintiidivmod}%

```

### 20.44. \xintiiSqr

1.2l: `\xintiiSqr` made robust against non terminated input.

```
1811 \def\xintiiSqr {\romannumeral0\xintiisqr }%
1812 \def\xintiisqr #1%
1813 {%
1814     \expandafter\xINT_sqr\romannumeral0\xintiiaabs{#1}\xint:
1815 }%
1816 \def\xINT_sqr #1\xint:
1817 {%
1818     \expandafter\xINT_sqr_a
1819     \romannumeral0\expandafter\xINT_sepandrev_andcount
1820     \romannumeral0\xINT_zeroes_forviii #1\R\R\R\R\R\R\R\R{10}0000001\W
1821     #1\xINT_rsepbyviii_end_A 2345678%
1822     \xINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
1823     \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
1824     \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_\W
1825     \xint:
1826 }%

```

1.2c `\XINT_mul_loop` can now be called directly even with small arguments, thus the following check is not anymore a necessity.

```
1827 \def\xINT_sqr_a #1\xint:
1828 {%
1829     \ifnum #1=\xint_c_i \expandafter\xINT_sqr_small
1830     \else\expandafter\xINT_sqr_start\fi
1831 }%
1832 \def\xINT_sqr_small 1#1#2#3#4#5!\xint:
1833 {%
1834     \ifnum #1#2#3#4#5<46341 \expandafter\xINT_sqr_verysmall\fi
1835     \expandafter\xINT_sqr_small_out
1836     \the\numexpr\xINT_minimul_a #1#2#3#4\xint:#5!#1#2#3#4#5!%

```



[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

An ending 1;! is produced on output for `\XINT_mul_loop` and gets incorporated to the delimiter needed by the `\XINT_unrevbyiii` done by `\XINT_mul_out`.

### 20.45. \xintiPow

1.21: `\xintiiPow` made robust against non terminated input.

```

1854 \def\xintiIPow {\romannumeral0\xintiipow }%
1855 \def\xintiipow #1#2%
1856 {%
1857     \expandafter\xint_pow\the\numexpr #2\expandafter
1858     .\romannumeral`&&@#1\xint:
1859 }%
1860 \def\xint_pow #1.#2%#3\xint:
1861 {%
1862     \xint_UDzerominusfork
1863     #2-\XINT_pow_AisZero
1864     0#2\XINT_pow_Aneg
1865     0-{ \XINT_pow_Apos #2}%
1866     \krof {#1}%
1867 }%
1868 \def\XINT_pow_AisZero #1#2\xint:
1869 {%
1870     \ifcase\XINT_cntSgn #1\xint:
1871         \xint_afterfi { 1}%
1872     \or
1873         \xint_afterfi { 0}%
1874     \else

```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

1875         \xint_afterfi
1876         {\XINT_signalcondition{DivisionByZero}{0 raised to power #1.}{0}}%
1877     \fi
1878 }%
1879 \def\XINT_pow_Aneg #1%
1880 {%
1881     \ifodd #1
1882         \expandafter\XINT_opp\romannumeral0%
1883     \fi
1884     \XINT_pow_Apos {}{#1}%
1885 }%
1886 \def\XINT_pow_Apos #1#2{\XINT_pow_Apos_a {}{#2}#1}%
1887 \def\XINT_pow_Apos_a #1#2#3%
1888 {%
1889     \xint_gob_til_xint: #3\XINT_pow_Apos_short\xint:
1890     \XINT_pow_AatleastTwo {}{#1}#2#3%
1891 }%
1892 \def\XINT_pow_Apos_short\xint:\XINT_pow_AatleastTwo #1#2\xint:
1893 {%
1894     \ifcase #2
1895         \xintError:thiscannothappen
1896     \or \expandafter\XINT_pow_AisOne
1897     \else\expandafter\XINT_pow_AatleastTwo
1898     \fi {}{#1}#2\xint:
1899 }%
1900 \def\XINT_pow_AisOne #1\xint:{ 1}%
1901 \def\XINT_pow_AatleastTwo #1%
1902 {%
1903     \ifcase\XINT_cntSgn #1\xint:
1904         \expandafter\XINT_pow_BisZero
1905     \or
1906         \expandafter\XINT_pow_I_in
1907     \else
1908         \expandafter\XINT_pow_BisNegative
1909     \fi
1910     {}{#1}%
1911 }%
1912 \def\XINT_pow_BisNegative #1\xint:{\XINT_signalcondition{Underflow}%
1913     {Inverse power is not an integer.}{0}}%
1914 \def\XINT_pow_BisZero #1\xint:{ 1}%
1915     B = #1 > 0, A = #2 > 1. Earlier code checked if size of B did not exceed a given limit (for example
1916     131000).
1917 \def\XINT_pow_I_in #1#2\xint:
1918 {%
1919     \expandafter\XINT_pow_I_loop
1920     \the\numexpr #1\expandafter\xint:%
1921     \romannumeral0\expandafter\XINT_sepandrev
1922     \romannumeral0\XINT_zeroes_forviii #2\R\R\R\R\R\R\R\{10}0000001\W
1923     #2\XINT_rsepbyviii_end_A 2345678%
1924     \XINT_rsepbyviii_end_B 2345678\relax XX%
1925     \R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\R\xint:\W
1926     1;! \W

```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1925     1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
1926 }%
1927 \def\XINT_pow_I_loop #1\xint:%
1928 {%
1929     \ifnum #1 = \xint_c_i\expandafter\XINT_pow_I_exit\fi
1930     \ifodd #1
1931         \expandafter\XINT_pow_II_in
1932     \else
1933         \expandafter\XINT_pow_I_squareit
1934     \fi #1\xint:%
1935 }%
1936 \def\XINT_pow_I_exit \ifodd #1\fi #2\xint:#3\W {\XINT_mul_out #3}%
```

The 1.2c `\XINT_mul_loop` can be called directly even with small arguments, hence the "butcheckifsmall" is not a necessity as it was earlier with 1.2. On  $2^{30}$ , it does bring roughly a 40% time gain though, and 30% gain for  $2^{60}$ . The overhead on big computations should be negligible.

```
1937 \def\XINT_pow_I_squareit #1\xint:#2\W%
1938 {%
1939     \expandafter\XINT_pow_I_loop
1940     \the\numexpr #1/\xint_c_ii\expandafter\xint:%
1941     \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W
1942 }%
1943 \def\XINT_pow_mulbutcheckifsmall #1!1#2%
1944 {%
1945     \xint_gob_til_sc #2\XINT_pow_mul_small;%
1946     \XINT_mul_loop 100000000!1;! \W #1!1#2%
1947 }%
1948 \def\XINT_pow_mul_small;\XINT_mul_loop
1949 100000000!1;! \W 1#1!1;! \W
1950 {%
1951     \XINT_smallmul 1#1!%
1952 }%
1953 \def\XINT_pow_II_in #1\xint:#2\W
1954 {%
1955     \expandafter\XINT_pow_II_loop
1956     \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter\xint:%
1957     \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W #2\W
1958 }%
1959 \def\XINT_pow_II_loop #1\xint:%
1960 {%
1961     \ifnum #1 = \xint_c_i\expandafter\XINT_pow_II_exit\fi
1962     \ifodd #1
1963         \expandafter\XINT_pow_II_odda
1964     \else
1965         \expandafter\XINT_pow_II_even
1966     \fi #1\xint:%
1967 }%
1968 \def\XINT_pow_II_exit\ifodd #1\fi #2\xint:#3\W #4\W
1969 {%
1970     \expandafter\XINT_mul_out
1971     \the\numexpr\XINT_pow_mulbutcheckifsmall #4\W #3%
1972 }%
1973 \def\XINT_pow_II_even #1\xint:#2\W
```

## TOC

TOC, *xintkernel*, *xinttools*, [xintcore](#), *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

1974 {%
1975   \expandafter\XINT_pow_II_loop
1976   \the\numexpr #1/\xint_c_ii\expandafter\xint:%
1977   \the\numexpr\XINT_pow_mulbutcheckifsmall #2\W #2\W
1978 }%
1979 \def\XINT_pow_II_odda #1\xint:#2\W #3\W
1980 {%
1981   \expandafter\XINT_pow_II_oddb
1982   \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter\xint:%
1983   \the\numexpr\XINT_pow_mulbutcheckifsmall #3\W #2\W #2\W
1984 }%
1985 \def\XINT_pow_II_oddb #1\xint:#2\W #3\W
1986 {%
1987   \expandafter\XINT_pow_II_loop
1988   \the\numexpr #1\expandafter\xint:%
1989   \the\numexpr\XINT_pow_mulbutcheckifsmall #3\W #3\W #2\W
1990 }%

```

## 20.46. \xintiiFac

Moved here from xint.sty with release 1.2 (to be usable by `\bnumexpr`).

Partially rewritten with release 1.2 to benefit from the inner format of the 1.2 multiplication.

With current default settings of the etex memory and a.t.t.o.w (11/2015) the maximal possible computation is 5971! (which has 19956 digits).

Note (end november 2015): I also tried out a quickly written recursive (binary split) implementation

```

\catcode`_ 11
\catcode`^ 11
\long\def\xint_firstofthree #1#2#3{#1}%
\long\def\xint_secondofthree #1#2#3{#2}%
\long\def\xint_thirdofthree #1#2#3{#3}%
% quickly written factorial using binary split recursive method
\def\tFac {\romannumeral-`0\tfac }%
\def\tfac #1{\expandafter\XINT_mul_out
  \romannumeral-`0\ufac {1}{#1}1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W}%
\def\ufac #1#2{\ifcase\numexpr#2-#1\relax
  \expandafter\xint_firstofthree
  \or
  \expandafter\xint_secondofthree
  \else
  \expandafter\xint_thirdofthree
  \fi
  {\the\numexpr\xint_c_x^viii+#1!1;!}%
  {\the\numexpr\xint_c_x^viii+#1*#2!1;!}%
  {\expandafter\vfac\the\numexpr (#1+#2)/\xint_c_ii.#1.#2.}%
}%
\def\vfac #1.#2.#3.%
{%
  \expandafter
  \wfac\expandafter
  {\romannumeral-`0\expandafter
    \ufac\expandafter{\the\numexpr #1+\xint_c_i}{#3}}%
  {\ufac {#2}{#1}}%
}

```

```
}%
\def\wfac #1#2{\expandafter\zfac\romannumeral-\`0#2\W #1}%
\def\zfac {\the\numexpr\XINT_mul_loop 100000000!1;!\W }% core multiplication...
\catcode\_ 8
\catcode\_ ^ 7
```

and I was quite surprised that it was only about 1.6x--2x slower in the range N=200 to 2000 than the `\xintiiFac` here which attempts to be smarter...

Note (2017, 1.21): I found out some code comment of mine that the code here should be more in the style of `\xintiiBinomial`, but I left matters untouched.

```
1991 \def\xintiiFac {\romannumeral0\xintiifac }%
1992 \def\xintiifac #1{\expandafter\XINT_fac_fork\the\numexpr#1}%
1993 \def\XINT_fac_fork #1#2.%
1994 {%
1995   \xint_UDzerominusfork
1996   #1-\XINT_fac_zero
1997   0#1\XINT_fac_neg
1998   0-\XINT_fac_checksize
1999   \krof #1#2.%
2000 }%
2001 \def\XINT_fac_zero #1.{ 1}%
2002 \def\XINT_fac_neg #1.{\XINT_signalcondition{InvalidOperation}{Factorial of
2003   negative argument: #1.}}{ 0}}%
2004 \def\XINT_fac_checksize #1.%
2005 {%
2006   \ifnum #1>\xint_c_x^iv \xint_dothis{\XINT_fac_toobig #1.}\fi
2007   \ifnum #1>465 \xint_dothis{\XINT_fac_bigloop_a #1.}\fi
2008   \ifnum #1>101 \xint_dothis{\XINT_fac_medloop_a #1.\XINT_mul_out}\fi
2009   \xint_orthat{\XINT_fac_smallloop_a #1.\XINT_mul_out}%
2010   1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!1\R!\W
2011 }%
2012 \def\XINT_fac_toobig
2013 #1.#2\W{\XINT_signalcondition{InvalidOperation}{Factorial
2014   argument is too large: #1 > 10^4.}}{ 0}}%
2015 \def\XINT_fac_bigloop_a #1.%
2016 {%
2017   \expandafter\XINT_fac_bigloop_b \the\numexpr
2018   #1+\xint_c_i-\xint_c_ii*((#1-464)/\xint_c_ii).#1.%
2019 }%
2020 \def\XINT_fac_bigloop_b #1.#2.%
2021 {%
2022   \expandafter\XINT_fac_medloop_a
2023   \the\numexpr #1-\xint_c_i.{\XINT_fac_bigloop_loop #1.#2.}%
2024 }%
2025 \def\XINT_fac_bigloop_loop #1.#2.%
2026 {%
2027   \ifnum #1>#2 \expandafter\XINT_fac_bigloop_exit\fi
2028   \expandafter\XINT_fac_bigloop_loop
2029   \the\numexpr #1+\xint_c_ii\expandafter.%
2030   \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_bigloop_mul #1!%
2031 }%
2032 \def\XINT_fac_bigloop_exit #1!{\XINT_mul_out}%
2033 \def\XINT_fac_bigloop_mul #1!%
```

## TOC

TOC, xintkernel, xinttools, [xintcore](#), xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

2034 {%
2035     \expandafter\XINT_smallmul
2036     \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
2037 }%
2038 \def\XINT_fac_medloop_a #1.%
2039 {%
2040     \expandafter\XINT_fac_medloop_b
2041     \the\numexpr #1+\xint_c_i-\xint_c_iii*((#1-100)/\xint_c_iii).#1.%
2042 }%
2043 \def\XINT_fac_medloop_b #1.#2.%
2044 {%
2045     \expandafter\XINT_fac_smallloop_a
2046     \the\numexpr #1-\xint_c_i.{\XINT_fac_medloop_loop #1.#2.}%
2047 }%
2048 \def\XINT_fac_medloop_loop #1.#2.%
2049 {%
2050     \ifnum #1>#2 \expandafter\XINT_fac_loop_exit\fi
2051     \expandafter\XINT_fac_medloop_loop
2052     \the\numexpr #1+\xint_c_iii\expandafter.%
2053     \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_medloop_mul #1!%
2054 }%
2055 \def\XINT_fac_medloop_mul #1!%
2056 {%
2057     \expandafter\XINT_smallmul
2058     \the\numexpr
2059     \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
2060 }%
2061 \def\XINT_fac_smallloop_a #1.%
2062 {%
2063     \csname
2064         XINT_fac_smallloop_\the\numexpr #1-\xint_c_iv*(#1/\xint_c_iv)\relax
2065     \endcsname #1.%
2066 }%
2067 \expandafter\def\csname XINT_fac_smallloop_1\endcsname #1.%
2068 {%
2069     \XINT_fac_smallloop_loop 2.#1.100000001!1;!%
2070 }%
2071 \expandafter\def\csname XINT_fac_smallloop_-2\endcsname #1.%
2072 {%
2073     \XINT_fac_smallloop_loop 3.#1.100000002!1;!%
2074 }%
2075 \expandafter\def\csname XINT_fac_smallloop_-1\endcsname #1.%
2076 {%
2077     \XINT_fac_smallloop_loop 4.#1.100000006!1;!%
2078 }%
2079 \expandafter\def\csname XINT_fac_smallloop_0\endcsname #1.%
2080 {%
2081     \XINT_fac_smallloop_loop 5.#1.1000000024!1;!%
2082 }%
2083 \def\XINT_fac_smallloop_loop #1.#2.%
2084 {%
2085     \ifnum #1>#2 \expandafter\XINT_fac_loop_exit\fi

```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfraction](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
2086 \expandafter\XINT_fac_smallloop_loop
2087 \the\numexpr #1+\xint_c_iv\expandafter.%
2088 \the\numexpr #2\expandafter.\the\numexpr\XINT_fac_smallloop_mul #1!%
2089 }%
2090 \def\XINT_fac_smallloop_mul #1!%
2091 {%
2092 \expandafter\XINT_smallmul
2093 \the\numexpr
2094 \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
2095 }%
2096 \def\XINT_fac_loop_exit #1!#2;!#3{#3#2;!}%
```

## 20.47. \XINT\_useiimessage

### 1.2o

```
2097 \def\XINT_useiimessage #1% used in LaTeX only
2098 {%
2099 \XINT_ifFlagRaised {#1}%
2100 {\@backslashchar#1
2101 (load xintfrac or use \@backslashchar xintii\xint_gobble_iv#1!)\MessageBreak}%
2102 }%
2103 }%
2104 \XINTrestorecatcodesendinginput%
```

## 21. Package *xint* implementation

.1	Package identification . . . . .	360	.34	\xintiifOne . . . . .	372
.2	\xintLen, \xintiLen . . . . .	360	.35	\xintiifOdd . . . . .	372
.3	\xintiLogTen . . . . .	361	.36	\xintifTrueAelseB, \xintifFalseAelseB	372
.4	\xintReverseDigits . . . . .	361	.37	\xintIsTrue, \xintIsFalse . . . . .	372
.5	\xintiiE . . . . .	362	.38	\xintNOT . . . . .	373
.6	\xintDecSplit . . . . .	362	.39	\xintAND, \xintOR, \xintXOR . . . . .	373
.7	\xintDecSplitL . . . . .	364	.40	\xintANDof . . . . .	373
.8	\xintDecSplitR . . . . .	364	.41	\xintORof . . . . .	373
.9	\xintDSHr . . . . .	364	.42	\xintXORof . . . . .	374
.10	\xintDSH . . . . .	365	.43	\xintiiMax . . . . .	374
.11	\xintDSx . . . . .	365	.44	\xintiiMin . . . . .	375
.12	\xintiiEq . . . . .	367	.45	\xintiiMaxof . . . . .	376
.13	\xintiiNotEq . . . . .	367	.46	\xintiiMinof . . . . .	377
.14	\xintiiGeq . . . . .	367	.47	\xintiiSum . . . . .	377
.15	\xintiiGt . . . . .	368	.48	\xintiiPrd . . . . .	378
.16	\xintiiLt . . . . .	368	.49	\xintiiSquareRoot . . . . .	378
.17	\xintiiGtorEq . . . . .	368	.50	\xintiiSqrt, \xintiiSqrtR . . . . .	384
.18	\xintiiLtorEq . . . . .	368	.51	\xintiiBinomial . . . . .	384
.19	\xintiiIsZero . . . . .	368	.52	\xintiiPFactorial . . . . .	390
.20	\xintiiIsNotZero . . . . .	368	.53	\xintBool, \xintToggle . . . . .	393
.21	\xintiiIsOne . . . . .	368	.54	\xintiiGCD . . . . .	393
.22	\xintiiOdd . . . . .	369	.55	\xintiiGCDof . . . . .	394
.23	\xintiiEven . . . . .	369	.56	\xintiiLCM . . . . .	394
.24	\xintiiMON . . . . .	369	.57	\xintiiLCMof . . . . .	395
.25	\xintiiMMON . . . . .	369	.58	(WIP) \xintRandomDigits . . . . .	395
.26	\xintSgnFork . . . . .	370	.59	(WIP) \XINT_eightrandomdigits, \xintEightRandomDigits . . . . .	396
.27	\xintiifSgn . . . . .	370	.60	(WIP) \xintRandBit . . . . .	396
.28	\xintiifCmp . . . . .	370	.61	(WIP) \xintXRandomDigits . . . . .	396
.29	\xintiifEq . . . . .	371	.62	(WIP) \xintiiRandRangeAtoB . . . . .	397
.30	\xintiifGt . . . . .	371	.63	(WIP) \xintiiRandRange . . . . .	397
.31	\xintiifLt . . . . .	371	.64	(WIP) Adjustments for engines without uniformdeviate primitive . . . . .	398
.32	\xintiifZero . . . . .	371			
.33	\xintiifNotZero . . . . .	372			

With release 1.1 the core arithmetic routines `\xintiiAdd`, `\xintiiSub`, `\xintiiMul`, `\xintiiQuo`, `\xintiiPow` were separated to be the main component of the then new *xintcore*.

1.3b adds randomness related macros.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2   \catcode13=5      % ^^M
3   \endlinechar=13 %
4   \catcode123=1     % {
5   \catcode125=2     % }
6   \catcode64=11     % @
7   \catcode44=12     % ,
8   \catcode46=12     % .
9   \catcode58=12     % :
10  \catcode94=7       % ^
11  \def\empty{} \def\space{ }\newlinechar10
12  \def\z{\endgroup}%
13  \expandafter\let\expandafter\x\csname ver@xint.sty\endcsname

```



```

14 \expandafter\let\expandafter\w\csname ver@xintcore.sty\endcsname
15 \expandafter\ifx\csname numexpr\endcsname\relax
16 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17 \immediate\write128{^^JPackage xint Warning:^^J}%
18 \space\space\space\space
19 \numexpr not available, aborting input.^^J}%
20 \else
21 \PackageWarningNoLine{xint}{\numexpr not available, aborting input}%
22 \fi
23 \def\z{\endgroup\endinput}%
24 \else
25 \ifx\x\relax % plain-TeX, first loading of xintcore.sty
26 \ifx\w\relax % but xintkernel.sty not yet loaded.
27 \def\z{\endgroup\input xintcore.sty\relax}%
28 \fi
29 \else
30 \ifx\x\empty % LaTeX, first loading,
31 % variable is initialized, but \ProvidesPackage not yet seen
32 \ifx\w\relax % xintcore.sty not yet loaded.
33 \def\z{\endgroup\RequirePackage{xintcore}}%
34 \fi
35 \else
36 \def\z{\endgroup\endinput}% xint already loaded.
37 \fi
38 \fi
39 \fi
40 \z%
41 \XINTsetupcatcodes% defined in xintkernel.sty (loaded by xintcore.sty)

```

## 21.1. Package identification

```

42 \XINT_providespackage
43 \ProvidesPackage{xint}%
44 [2025/09/06 v1.4o Expandable operations on big integers (JFB)]%

```

## 21.2. \xintLen, \xintiLen

`\xintLen` gets extended to fractions by `xintfrac.sty`:  $A/B$  is given length  $\text{len}(A)+\text{len}(B)-1$  (somewhat arbitrary). It applies `\xintNum` to its argument. A minus sign is accepted and ignored.

For parallelism with `\xintiNum/\xintNum`, 1.2o defines `\xintiLen`.

`\xintLen` gets redefined by `xintfrac`.

```

45 \def\xintiLen {\romannumeral0\xintilen }%
46 \def\xintilen #1{\def\xintilen ##1%
47 {%
48 \expandafter#1\the\numexpr
49 \expandafter\XINT_len_fork\romannumeral0\xintinum{##1}%
50 \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
51 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
52 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye\relax
53 }}\xintilen{ }%
54 \def\xintLen {\romannumeral0\xintlen }%
55 \let\xintlen\xintilen
56 \def\XINT_len_fork #1%

```

### 21.3. \xintiiLogTen

```

60 \def\xintiilogTen {\the\numexpr\xintiilogten}%
61 \def\xintiilogten #1%
62 {%
63     \expandafter\XINT_iilogten\romannumeral`&&@#1%
64     \xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
65     \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
66     \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
67     \relax
68 }%
69 \def\XINT_iilogten #1{\if#10-"7FFF8000\fi -1+%
70     \expandafter\XINT_length_loop\xint_UDsianfork#1{\}-#1\krof}%

```

This macro is currently not used elsewhere in xint code.

361

```

95      1#9#8#7#6#5#4#3#2#1\expandafter!\the\numexpr\XINT_microrevsep
96 }%
97 \def\XINT_microrevsep_end #1\W #2\expandafter #3\Z{\relax#2!}%
98 \def\XINT_revdigits_b #11#2!1#3!1#4!1#5!1#6!1#7!1#8!1#9!%
99 {%
100     \xint_gob_til_R #9\XINT_revdigits_end\R
101         \XINT_revdigits_b {#9#8#7#6#5#4#3#2#1}%
102 }%
103 \def\XINT_revdigits_end#1{%
104 \def\XINT_revdigits_end\R\XINT_revdigits_b ##1##2\W
105     {\expandafter#1\xint_gob_til_Z ##1}%
106 }\XINT_revdigits_end{ }%
107 \let\xintRev\xintReverseDigits

```

## 21.5. \xintiiE

Originally was used in [\xintiexpr](#). Transferred from [xintfrac](#) for 1.1. Code rewritten for 1.2i. [\xintiiE{x}{e}](#) extends x with e zeroes if e is positive and simply outputs x if e is zero or negative. Attention, le comportement pour  $e < 0$  ne doit pas être modifié car [\xintMod](#) et autres macros en dépendent.

```

108 \def\xintiiE {\romannumeral0\xintiiE }%
109 \def\xintiiE #1#2%
110     {\expandafter\XINT_iiE_fork\the\numexpr #2\expandafter.\romannumeral`&&@#1;}%
111 \def\XINT_iiE_fork #1%
112 {%
113     \xint_UDsignfork
114     #1\XINT_iiE_neg
115     -\XINT_iiE_a
116     \krof #1%
117 }%

```

le #2 a le bon pattern terminé par ; #1=0 est OK pour [\XINT\\_rep](#).

```

118 \def\XINT_iiE_a #1.%
119     {\expandafter\XINT_dsx_append\romannumeral\XINT_rep #1\endcsname 0.}%
120 \def\XINT_iiE_neg #1.#2;{ #2}%

```

## 21.6. \xintDecSplit

### DECIMAL SPLIT

The macro [\xintDecSplit {x}{A}](#) cuts A which is composed of digits (leading zeroes ok, but no sign) (\*) into two (each possibly empty) pieces L and R. The concatenation LR always reproduces A.

The position of the cut is specified by the first argument x. If x is zero or positive the cut location is x slots to the left of the right end of the number. If x becomes equal to or larger than the length of the number then L becomes empty. If x is negative the location of the cut is |x| slots to the right of the left end of the number.

(\*) versions earlier than 1.2i first replaced A with its absolute value. This is not the case anymore. This macro should NOT be used for A with a leading sign (+ or -).

Entirely rewritten for 1.2i (2016/12/11).

Attention: [\xintDecSplit](#) not robust against non terminated second argument.

```

121 \def\xintDecSplit {\romannumeral0\xintdecsplit }%
122 \def\xintdecsplit #1#2%
123 {%
124     \expandafter\XINT_split_finish

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

125 \romannumeral0\expandafter\XINT_split_xfork
126 \the\numexpr #1\expandafter.\romannumeral`&&@#2%
127 \xint_bye2345678\xint_bye..%
128 }%
129 \def\XINT_split_finish #1.#2.{{#1}{#2}}%
130 \def\XINT_split_xfork #1%
131 {%
132 \xint_UDzerominusfork
133 #1-\XINT_split_zerosplit
134 0#1\XINT_split_fromleft
135 0-\XINT_split_fromright #1}%
136 \kroft
137 }%
138 \def\XINT_split_zerosplit .#1\xint_bye#2\xint_bye..{ #1..}%
139 \def\XINT_split_fromleft
140 {\expandafter\XINT_split_fromleft_a\the\numexpr\xint_c_viii-}%
141 \def\XINT_split_fromleft_a #1%
142 {%
143 \xint_UDsignfork
144 #1\XINT_split_fromleft_b
145 -{\XINT_split_fromleft_end_a #1}%
146 \kroft
147 }%
148 \def\XINT_split_fromleft_b #1.#2#3#4#5#6#7#8#9%
149 {%
150 \expandafter\XINT_split_fromleft_clean
151 \the\numexpr1#2#3#4#5#6#7#8#9\expandafter
152 \XINT_split_fromleft_a\the\numexpr\xint_c_viii-#1.%
153 }%
154 \def\XINT_split_fromleft_end_a #1.%
155 {%
156 \expandafter\XINT_split_fromleft_clean
157 \the\numexpr1\csname XINT_split_fromleft_end#1\endcsname
158 }%
159 \def\XINT_split_fromleft_clean 1{ }%
160 \expandafter\def\csname XINT_split_fromleft_end7\endcsname #1%
161 {#1\XINT_split_fromleft_end_b}%
162 \expandafter\def\csname XINT_split_fromleft_end6\endcsname #1#2%
163 {#1#2\XINT_split_fromleft_end_b}%
164 \expandafter\def\csname XINT_split_fromleft_end5\endcsname #1#2#3%
165 {#1#2#3\XINT_split_fromleft_end_b}%
166 \expandafter\def\csname XINT_split_fromleft_end4\endcsname #1#2#3#4%
167 {#1#2#3#4\XINT_split_fromleft_end_b}%
168 \expandafter\def\csname XINT_split_fromleft_end3\endcsname #1#2#3#4#5%
169 {#1#2#3#4#5\XINT_split_fromleft_end_b}%
170 \expandafter\def\csname XINT_split_fromleft_end2\endcsname #1#2#3#4#5#6%
171 {#1#2#3#4#5#6\XINT_split_fromleft_end_b}%
172 \expandafter\def\csname XINT_split_fromleft_end1\endcsname #1#2#3#4#5#6#7%
173 {#1#2#3#4#5#6#7\XINT_split_fromleft_end_b}%
174 \expandafter\def\csname XINT_split_fromleft_end0\endcsname #1#2#3#4#5#6#7#8%
175 {#1#2#3#4#5#6#7#8\XINT_split_fromleft_end_b}%
176 \def\XINT_split_fromleft_end_b #1\xint_bye#2\xint_bye.{.#1}% puis .

```

```

177 \def\XINT_split_fromright #1.#2\xint_bye
178 {%
179     \expandafter\XINT_split_fromright_a
180     \the\numexpr#1-\numexpr\XINT_length_loop
181     #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
182     \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
183     \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
184     .#2\xint_bye
185 }%
186 \def\XINT_split_fromright_a #1%
187 {%
188     \xint_UDsignfork
189     #1\XINT_split_fromleft
190     -\XINT_split_fromright_Lempty
191     \krof
192 }%
193 \def\XINT_split_fromright_Lempty #1.#2\xint_bye#3..{.#2.}%

```

## 21.7. \xintDecSplitL

```

194 \def\xintDecSplitL {\romannumeral0\xintdecsplitl }%
195 \def\xintdecsplitl #1#2%
196 {%
197     \expandafter\XINT_splitl_finish
198     \romannumeral0\expandafter\XINT_split_xfork
199     \the\numexpr #1\expandafter.\romannumeral`&&@#2%
200     \xint_bye2345678\xint_bye..%
201 }%
202 \def\XINT_splitl_finish #1.#2.{ #1}%

```

## 21.8. \xintDecSplitR

```

203 \def\xintDecSplitR {\romannumeral0\xintdecsplitr }%
204 \def\xintdecsplitr #1#2%
205 {%
206     \expandafter\XINT_splitr_finish
207     \romannumeral0\expandafter\XINT_split_xfork
208     \the\numexpr #1\expandafter.\romannumeral`&&@#2%
209     \xint_bye2345678\xint_bye..%
210 }%
211 \def\XINT_splitr_finish #1.#2.{ #2}%

```

## 21.9. \xintDSHr

DECIMAL SHIFTS `\xintDSH {x}{A}`

si  $x \leq 0$ , fait  $A \rightarrow A \cdot 10^{|x|}$ . si  $x > 0$ , et  $A \geq 0$ , fait  $A \rightarrow \text{quo}(A, 10^x)$

si  $x > 0$ , et  $A < 0$ , fait  $A \rightarrow -\text{quo}(-A, 10^x)$

(donc pour  $x > 0$  c'est comme DSR itéré  $x$  fois)

`\xintDSHr` donne le 'reste' (si  $x \leq 0$  donne zéro).

Badly named macros.

Rewritten for 1.2i, this was old code and `\xintDSx` has changed interface.

```

212 \def\xintDSHr {\romannumeral0\xintdshr }%
213 \def\xintdshr #1#2%

```

```

214 {%
215     \expandafter\XINT_dshr_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%
216 }%
217 \def\XINT_dshr_fork #1%
218 {%
219     \xint_UDzerominusfork
220     0#1\XINT_dshr_xzeroorneg
221     #1-\XINT_dshr_xzeroorneg
222     0-\XINT_dshr_xpositive
223     \krof #1%
224 }%
225 \def\XINT_dshr_xzeroorneg #1;{ 0}%
226 \def\XINT_dshr_xpositive
227 {%
228     \expandafter\xint_stop_atsecondoftwo\romannumeral0\XINT_dsx_xisPos
229 }%

```

## 21.10. \xintDSH

```

230 \def\xintDSH {\romannumeral0\xintdsh }%
231 \def\xintdsh #1#2%
232 {%
233     \expandafter\XINT_dsh_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%
234 }%
235 \def\XINT_dsh_fork #1%
236 {%
237     \xint_UDzerominusfork
238     #1-\XINT_dsh_xiszero
239     0#1\XINT_dsx_xisNeg_checkA
240     0-{\XINT_dsh_xisPos #1}%
241     \krof
242 }%
243 \def\XINT_dsh_xiszero #1.#2;{ #2}%
244 \def\XINT_dsh_xisPos
245 {%
246     \expandafter\xint_stop_atfirstoftwo\romannumeral0\XINT_dsx_xisPos
247 }%

```

## 21.11. \xintDSx

--> Attention le cas  $x=0$  est traité dans la même catégorie que  $x > 0$  <--

si  $x < 0$ , fait  $A \rightarrow A \cdot 10^{|x|}$

si  $x \geq 0$ , et  $A \geq 0$ , fait  $A \rightarrow \{\text{quo}(A, 10^x)\}\{\text{rem}(A, 10^x)\}$

si  $x \geq 0$ , et  $A < 0$ , d'abord on calcule  $\{\text{quo}(-A, 10^x)\}\{\text{rem}(-A, 10^x)\}$

puis, si le premier n'est pas nul on lui donne le signe -

si le premier est nul on donne le signe - au second.

On peut donc toujours reconstituer l'original  $A$  par  $10^x Q \pm R$  où il faut prendre le signe plus si  $Q$  est positif ou nul et le signe moins si  $Q$  est strictement négatif.

Rewritten for 1.2i, this was old code.

```

248 \def\xintDSx {\romannumeral0\xintdsx }%
249 \def\xintdsx #1#2%
250 {%
251     \expandafter\XINT_dsx_fork\the\numexpr#1\expandafter.\romannumeral`&&@#2;%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```
252 }%
253 \def\XINT_dsx_fork #1%
254 {%
255     \xint_UDzerominusfork
256     #1-\XINT_dsx_xisZero
257     0#1\XINT_dsx_xisNeg_checkA
258     0-\XINT_dsx_xisPos #1}%
259 \krof
260 }%
261 \def\XINT_dsx_xisZero #1.#2;{{#2}{0}}%
262 \def\XINT_dsx_xisNeg_checkA #1.#2%
263 {%
264     \xint_gob_til_zero #2\XINT_dsx_xisNeg_Azero 0%
265     \expandafter\XINT_dsx_append\romannumeral\XINT_rep #1\endcsname 0.#2%
266 }%
267 \def\XINT_dsx_xisNeg_Azero #1;{ 0}%
268 \def\XINT_dsx_addzeros #1%
269     {\expandafter\XINT_dsx_append\romannumeral\XINT_rep#1\endcsname0.}%
270 \def\XINT_dsx_addzerosnofuss #1%
271     {\expandafter\XINT_dsx_append\romannumeral\xintreplicate{#1}0.}%
272 \def\XINT_dsx_append #1.#2;{ #2#1}%
273 \def\XINT_dsx_xisPos #1.#2%
274 {%
275     \xint_UDzerominusfork
276     #2-\XINT_dsx_AisZero
277     0#2\XINT_dsx_AisNeg
278     0-\XINT_dsx_AisPos
279     \krof #1.#2%
280 }%
281 \def\XINT_dsx_AisZero #1;{{0}{0}}%
282 \def\XINT_dsx_AisNeg #1.-#2;%
283 {%
284     \expandafter\XINT_dsx_AisNeg_checkiffirstempty
285     \romannumeral0\XINT_split_xfork #1.#2\xint_bye2345678\xint_bye..%
286 }%
287 \def\XINT_dsx_AisNeg_checkiffirstempty #1%
288 {%
289     \xint_gob_til_dot #1\XINT_dsx_AisNeg_finish_zero.%
290     \XINT_dsx_AisNeg_finish_notzero #1%
291 }%
292 \def\XINT_dsx_AisNeg_finish_zero.\XINT_dsx_AisNeg_finish_notzero.#1.%
293 {%
294     \expandafter\XINT_dsx_end
295     \expandafter {\romannumeral0\XINT_num {-#1}}{0}%
296 }%
297 \def\XINT_dsx_AisNeg_finish_notzero #1.#2.%
298 {%
299     \expandafter\XINT_dsx_end
300     \expandafter {\romannumeral0\XINT_num {#2}}{-#1}%
301 }%
302 \def\XINT_dsx_AisPos #1.#2;%
303 {%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
304 \expandafter\XINT_dsx_AisPos_finish
305 \romannumeral0\XINT_split_xfork #1.#2\xint_bye2345678\xint_bye..%
306 }%
307 \def\XINT_dsx_AisPos_finish #1.#2.%
308 {%
309 \expandafter\XINT_dsx_end
310 \expandafter {\romannumeral0\XINT_num {#2}}%
311 {\romannumeral0\XINT_num {#1}}%
312 }%
313 \def\XINT_dsx_end #1#2{\expandafter{#2}{#1}}%
```

### 21.12. \xintiiEq

no \xintiieq.

```
314 \def\xintiiEq #1#2{\romannumeral0\xintiiefeq{#1}{#2}{1}{0}}%
```

### 21.13. \xintiiNotEq

Pour xintexpr. Pas de version en lowercase.

```
315 \def\xintiiNotEq #1#2{\romannumeral0\xintiiefeq {#1}{#2}{0}{1}}%
```

### 21.14. \xintiiGeq

PLUS GRAND OU ÉGAL attention compare les \*\*valeurs absolues\*\*

1.2l made \xintiiGeq robust against non terminated items.

1.2l rewrote \xintiiCmp, but forgot to handle \xintiiGeq too. Done at 1.2m.

This macro should have been called \xintGEq for example.

```
316 \def\xintiiGeq {\romannumeral0\xintiigeq }%
317 \def\xintiigeq #1{\expandafter\XINT_iigeq\romannumeral`&&@#1\xint:}%
318 \def\XINT_iigeq #1#2\xint:#3%
319 {%
320 \expandafter\XINT_geq_fork\expandafter #1\romannumeral`&&@#3\xint:#2\xint:
321 }%
322 \def\XINT_geq #1#2\xint:#3%
323 {%
324 \expandafter\XINT_geq_fork\expandafter #1\romannumeral0\xintnum{#3}\xint:#2\xint:
325 }%
326 \def\XINT_geq_fork #1#2%
327 {%
328 \xint_UDzerofork
329 #1\XINT_geq_firstiszero
330 #2\XINT_geq_secondiszero
331 0{}}%
332 \krof
333 \xint_UDsignsfork
334 #1#2\XINT_geq_minusminus
335 #1-\XINT_geq_minusplus
336 #2-\XINT_geq_plusminus
337 --\XINT_geq_plusplus
338 \krof #1#2%
339 }%
340 \def\XINT_geq_firstiszero #1\krof 0#2#3\xint:#4\xint:
```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
341             {\xint_UDzerofork #2{ 1}{ 0}{ 0}\krof }%
342 \def\xINT_geq_secondiszero #1\krof #20#3\xint:#4\xint:{ 1}%
343 \def\xINT_geq_plusminus #1-{\XINT_geq_plusplus #1{}}%
344 \def\xINT_geq_minusplus -#1{\XINT_geq_plusplus {}#1}%
345 \def\xINT_geq_minusminus --{\XINT_geq_plusplus {}{}}%
346 \def\xINT_geq_plusplus
347   {\expandafter\xINT_geq_finish\romannumeral0\xINT_cmp_plusplus}%
348 \def\xINT_geq_finish #1{\if-#1\expandafter\xINT_geq_no
349   \else\expandafter\xINT_geq_yes\fi}%
350 \def\xINT_geq_no 1{ 0}%
351 \def\xINT_geq_yes { 1}%
```

### 21.15. \xintiiGt

```
352 \def\xintiiGt #1#2{\romannumeral0\xintiiifgt{#1}{#2}{1}{0}}%
```

### 21.16. \xintiiLt

```
353 \def\xintiiLt #1#2{\romannumeral0\xintiiiflt{#1}{#2}{1}{0}}%
```

### 21.17. \xintiiGtorEq

```
354 \def\xintiiGtorEq #1#2{\romannumeral0\xintiiiflt {#1}{#2}{0}{1}}%
```

### 21.18. \xintiiLtorEq

```
355 \def\xintiiLtorEq #1#2{\romannumeral0\xintiiifgt {#1}{#2}{0}{1}}%
```

### 21.19. \xintiiIsZero

1.09a. restyled in 1.09i. 1.1 adds `\xintiiIsZero`, etc... for optimization in `\xintexpr`

```
356 \def\xintiiIsZero {\romannumeral0\xintiiiszero }%
357 \def\xintiiiszero #1{\if0\xintiiSgn{#1}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
```

### 21.20. \xintiiIsNotZero

1.09a. restyled in 1.09i. 1.1 adds `\xintiiIsZero`, etc... for optimization in `\xintexpr`

```
358 \def\xintiiIsNotZero {\romannumeral0\xintiiisnotzero }%
359 \def\xintiiisnotzero
360   #1{\if0\xintiiSgn{#1}\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
```

### 21.21. \xintiiIsOne

Added in 1.03. 1.09a defines `\xintIsOne`. 1.1a adds `\xintiiIsOne`.

`\XINT_isOne` rewritten for 1.2g. Works with expanded strict integers, positive or negative.

```
361 \def\xintiiIsOne {\romannumeral0\xintiiisone }%
362 \def\xintiiisone #1{\expandafter\XINT_isone\romannumeral`&&#1XY}%
363 \def\XINT_isone #1#2#3Y%
364 {%
365   \unless\if#2X\xint_dothis{ 0}\fi
366   \unless\if#11\xint_dothis{ 0}\fi
367   \xint_orthat{ 1}%
368 }%
```

```

369 \def\XINT_isOne #1{\XINT_is_One#1XY}%
370 \def\XINT_is_One #1#2#3Y%
371 {%
372     \unless\if#2X\xint_dothis0\fi
373     \unless\if#11\xint_dothis0\fi
374     \xint_orthat1%
375 }%

```

## 21.22. \xintiiOdd

`\xintOdd` is needed for the `xintexpr`-essions `even()` and `odd()` functions (and also by `\xintNewExpr`).

```

376 \def\xintiiOdd {\romannumeral0\xintiiodd }%
377 \def\xintiiodd #1%
378 {%
379     \ifodd\xintLDg{#1} %<- intentional space
380     \xint_afterfi{ 1}%
381     \else
382     \xint_afterfi{ 0}%
383     \fi
384 }%

```

## 21.23. \xintiiEven

```

385 \def\xintiiEven {\romannumeral0\xintiieven }%
386 \def\xintiieven #1%
387 {%
388     \ifodd\xintLDg{#1} %<- intentional space
389     \xint_afterfi{ 0}%
390     \else
391     \xint_afterfi{ 1}%
392     \fi
393 }%

```

## 21.24. \xintiiMON

MINUS ONE TO THE POWER N

```

394 \def\xintiiMON {\romannumeral0\xintiimon }%
395 \def\xintiimon #1%
396 {%
397     \ifodd\xintLDg {#1} %<- intentional space
398     \xint_afterfi{ -1}%
399     \else
400     \xint_afterfi{ 1}%
401     \fi
402 }%

```

## 21.25. \xintiiMMON

MINUS ONE TO THE POWER N-1

```

403 \def\xintiiMMON {\romannumeral0\xintiimmon }%
404 \def\xintiimmon #1%
405 {%

```

```

406 \ifodd\xintLDg {#1} %<- intentional space
407 \xint_afterfi{ 1}%
408 \else
409 \xint_afterfi{ -1}%
410 \fi
411 }%

```

## 21.26. \xintSgnFork

Expandable three-way fork added in 1.07. The argument #1 must expand to non-self-ending -1,0 or 1. 1.09i with `_thenstop` (now `_stop_at...`).

```

412 \def\xintSgnFork {\romannumeral0\xintsgnfork }%
413 \def\xintsgnfork #1%
414 {%
415 \ifcase #1 \expandafter\xint_stop_atsecondofthree
416 \or\expandafter\xint_stop_atthirdofthree
417 \else\expandafter\xint_stop_atfirstofthree
418 \fi
419 }%

```

## 21.27. \xintiiifSgn

Expandable three-way fork added in 1.09a. Branches expandably depending on whether <0, =0, >0. Choice of branch guaranteed in two steps.

1.09i has `\xint_firstofthreeafterstop` (now `\xint_stop_atfirstofthree`) etc for faster expansion.

1.1 adds `\xintiiifSgn` for optimization in `xintexpr`-essions. Should I move them to `xintcore`? (for `bnumexpr`)

```

420 \def\xintiiifSgn {\romannumeral0\xintiiifsgn }%
421 \def\xintiiifsgn #1%
422 {%
423 \ifcase \xintiiSgn{#1}
424 \expandafter\xint_stop_atsecondofthree
425 \or\expandafter\xint_stop_atthirdofthree
426 \else\expandafter\xint_stop_atfirstofthree
427 \fi
428 }%

```

## 21.28. \xintiiifCmp

1.09e `\xintifCmp {n}{m}{if n<m}{if n=m}{if n>m}`. 1.1a adds `ii` variant

```

429 \def\xintiiifCmp {\romannumeral0\xintiiifcmp }%
430 \def\xintiiifcmp #1#2%
431 {%
432 \ifcase\xintiiCmp {#1}{#2}
433 \expandafter\xint_stop_atsecondofthree
434 \or\expandafter\xint_stop_atthirdofthree
435 \else\expandafter\xint_stop_atfirstofthree
436 \fi
437 }%

```

### 21.29. `\xintiiifEq`

1.09a `\xintifEq` {n}{m}{YES if n=m}{NO if n<>m}. 1.1a adds ii variant

```

438 \def\xintiiifEq {\romannumeral0\xintiiifeq }%
439 \def\xintiiifeq #1#2%
440 {%
441     \if0\xintiiCmp{#1}{#2}%
442         \expandafter\xint_stop_atfirstoftwo
443     \else\expandafter\xint_stop_atsecondoftwo
444     \fi
445 }%
```

### 21.30. `\xintiiifGt`

1.09a `\xintifGt` {n}{m}{YES if n>m}{NO if n<=m}. 1.1a adds ii variant

```

446 \def\xintiiifGt {\romannumeral0\xintiiifgt }%
447 \def\xintiiifgt #1#2%
448 {%
449     \if1\xintiiCmp{#1}{#2}%
450         \expandafter\xint_stop_atfirstoftwo
451     \else\expandafter\xint_stop_atsecondoftwo
452     \fi
453 }%
```

### 21.31. `\xintiiifLt`

1.09a `\xintifLt` {n}{m}{YES if n<m}{NO if n>=m}. Restyled in 1.09i. 1.1a adds ii variant

```

454 \def\xintiiifLt {\romannumeral0\xintiiiflt }%
455 \def\xintiiiflt #1#2%
456 {%
457     \ifnum\xintiiCmp{#1}{#2}<\xint_c_
458         \expandafter\xint_stop_atfirstoftwo
459     \else \expandafter\xint_stop_atsecondoftwo
460     \fi
461 }%
```

### 21.32. `\xintiiifZero`

Expandable two-way fork added in 1.09a. Branches expandably depending on whether the argument is zero (branch A) or not (branch B). 1.09i restyling. By the way it appears (not thoroughly tested, though) that `\if` tests are faster than `\ifnum` tests. 1.1 adds ii versions.

```

462 \def\xintiiifZero {\romannumeral0\xintiiifzero }%
463 \def\xintiiifzero #1%
464 {%
465     \if0\xintiiSgn{#1}%
466         \expandafter\xint_stop_atfirstoftwo
467     \else
468         \expandafter\xint_stop_atsecondoftwo
469     \fi
470 }%
```

### 21.33. `\xintiiifNotZero`

```

471 \def\xintiiifNotZero {\romannumeral0\xintiiifnotzero }%
472 \def\xintiiifnotzero #1%
473 {%
474   \if0\xintiiSgn{#1}%
475     \expandafter\xint_stop_atsecondoftwo
476   \else
477     \expandafter\xint_stop_atfirstoftwo
478   \fi
479 }%

```

### 21.34. `\xintiiifOne`

added in 1.09i. 1.1a adds `\xintiiifOne`.

```

480 \def\xintiiifOne {\romannumeral0\xintiiifone }%
481 \def\xintiiifone #1%
482 {%
483   \if1\xintiiIsOne{#1}%
484     \expandafter\xint_stop_atfirstoftwo
485   \else
486     \expandafter\xint_stop_atsecondoftwo
487   \fi
488 }%

```

### 21.35. `\xintiiifOdd`

1.09e. Restyled in 1.09i. 1.1a adds `\xintiiifOdd`.

```

489 \def\xintiiifOdd {\romannumeral0\xintiiifodd }%
490 \def\xintiiifodd #1%
491 {%
492   \if\xintiiOdd{#1}1%
493     \expandafter\xint_stop_atfirstoftwo
494   \else
495     \expandafter\xint_stop_atsecondoftwo
496   \fi
497 }%

```

### 21.36. `\xintifTrueAelseB`, `\xintifFalseAelseB`

1.09i, with name changes at 1.2i.

1.2o uses `\xintiiifNotZero`, see comments to `\xintAND` etc... This will work fine with arguments being nested `xintfrac.sty` macros, without the overhead of `\xintNum` or `\xintRaw` parsing.

```

498 \def\xintifTrueAelseB {\romannumeral0\xintiiifnotzero}%
499 \def\xintifFalseAelseB{\romannumeral0\xintiiifzero}%

```

### 21.37. `\xintIsTrue`, `\xintIsFalse`

1.09c. Suppressed at 1.2o. They seem not to have been documented, fortunately.

```

500 %\let\xintIsTrue \xintIsNotZero
501 %\let\xintIsFalse\xintIsZero

```

### 21.38. `\xintNOT`

1.09c with name change at 1.2o. Besides, the macro is now defined as ii-type.

```
502 \def\xintNOT{\romannumeral0\xintiiszero}%
```

### 21.39. `\xintAND`, `\xintOR`, `\xintXOR`

Added with 1.09a. But they used `\xintSgn`, etc... rather than `\xintiiSgn`. This brings `\xintNum` overhead which is not really desired, and which is not needed for use by `xintexpr.sty`. At 1.2o I modify them to use only ii macros. This is enough for sign or zeroness even for `xintfrac` format, as manipulated inside the `\xintexpr`. Big hesitation whether there should be however `\xintiiAND` outputting 1 or 0 versus an `\xintAND` outputting 1[0] versus 0[0] for example.

```
503 \def\xintAND {\romannumeral0\xintand }%
504 \def\xintand #1#2{\if0\xintiiSgn{#1}\expandafter\xint_firstoftwo
505             \else\expandafter\xint_secondoftwo\fi
506             { 0}{\xintiiisnotzero{#2}}}%
507 \def\xintOR {\romannumeral0\xintor }%
508 \def\xintor #1#2{\if0\xintiiSgn{#1}\expandafter\xint_firstoftwo
509             \else\expandafter\xint_secondoftwo\fi
510             {\xintiiisnotzero{#2}}{ 1}}%
511 \def\xintXOR {\romannumeral0\xintxor }%
512 \def\xintxor #1#2{\if\xintiiIsZero{#1}\xintiiIsZero{#2}%
513             \xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi }%
```

### 21.40. `\xintANDof`

New with 1.09a. `\xintANDof` works also with an empty list. Empty items however are not accepted.

1.2l made `\xintANDof` robust against non terminated items.

1.2o's `\xintifTrueAelseB` is now an ii macro, actually.

1.4. This macro as well as `ORof` and `XORof` were formally not used by `xintexpr`, which uses comma separated items, but at 1.4 `xintexpr` uses braced items. And the macros here got slightly refactored and `\XINT_ANDof` added for usage by `xintexpr` and the `NewExpr` hook. For some random reason I decided to use `^` as delimiter this has to do that other macros in `xintfrac` in same family (such as `\xintGCDof`, `\xintSum`) also use `\xint:` internally and although not strictly needed having two separate ones clarifies.

```
514 \def\xintANDof {\romannumeral0\xintandof }%
515 \def\xintandof #1{\expandafter\XINT_andof\romannumeral`&&@#1^}%
516 \def\XINT_ANDof {\romannumeral0\XINT_andof}%
517 \def\XINT_andof #1%
518 {%
519     \xint_gob_til_^ #1\XINT_andof_yes ^%
520     \xintiiifNotZero{#1}\XINT_andof\XINT_andof_no
521 }%
522 \def\XINT_andof_no #1^{ 0}%
523 \def\XINT_andof_yes ^#1\XINT_andof_no{ 1}%
```

### 21.41. `\xintORof`

New with 1.09a. Works also with an empty list. Empty items however are not accepted.

1.2l made `\xintORof` robust against non terminated items.

Refactored at 1.4.

```
524 \def\xintORof {\romannumeral0\xintorof }%
```

```

525 \def\xintorof #1{\expandafter\XINT_orof\romannumeral`&&@#1^}%
526 \def\XINT_ORof {\romannumeral0\XINT_orof}%
527 \def\XINT_orof #1%
528 {%
529     \xint_gob_til_ ^ #1\XINT_orof_no ^%
530     \xintiiifNotZero{#1}\XINT_orof_yes\XINT_orof
531 }%
532 \def\XINT_orof_yes#1^{ 1}%
533 \def\XINT_orof_no ^#1\XINT_orof{ 0}%

```

## 21.42. \xintXORof

New with 1.09a. Works with an empty list, too. Empty items however are not accepted. `\XINT_xorof` `f_c` more efficient in 1.09i.

1.21 made `\xintXORof` robust against non terminated items.

Refactored at 1.4 to use `\numexpr` (or an `\ifnum`). I have not tested if more efficient or not or if one can do better without `\the`. `\XINT_XORof` for *xintexpr* matters.

```

534 \def\xintXORof {\romannumeral0\xintxorof}%
535 \def\xintxorof #1{\expandafter\XINT_xorof\romannumeral`&&@#1^}%
536 \def\XINT_XORof {\romannumeral0\XINT_xorof}%
537 \def\XINT_xorof {\if1\the\numexpr\XINT_xorof_a}%
538 \def\XINT_xorof_a #1%
539 {%
540     \xint_gob_til_ ^ #1\XINT_xorof_e ^%
541     \xintiiifNotZero{#1}{-}{ }\XINT_xorof_a
542 }%
543 \def\XINT_xorof_e ^#1\XINT_xorof_a
544 {1\relax\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%

```

## 21.43. \xintiiMax

At 1.2m, a long-standing bug was fixed: `\xintiiMax` had the overhead of applying `\xintNum` to its arguments due to use of a sub-macro of `\xintGeq` code to which this overhead was added at some point.

And on this occasion I reduced even more number of times input is grabbed.

```

545 \def\xintiiMax {\romannumeral0\xintiimax}%
546 \def\xintiimax #1%
547 {%
548     \expandafter\xint_iimax \romannumeral`&&@#1\xint:
549 }%
550 \def\xint_iimax #1\xint:#2%
551 {%
552     \expandafter\XINT_max_fork\romannumeral`&&@#2\xint:#1\xint:
553 }%

```

`#3#4` vient du *\*premier\**, `#1#2` vient du *\*second\**. I have renamed the sub-macros at 1.2m because the terminology was quite counter-intuitive; there was no bug, but still.

```

554 \def\XINT_max_fork #1#2\xint:#3#4\xint:
555 {%
556     \xint_UDsignsfork
557     #1#3\XINT_max_minusminus % A < 0, B < 0
558     #1-\XINT_max_plusminus % B < 0, A >= 0
559     #3-\XINT_max_minusplus % A < 0, B >= 0
560     --{\xint_UDzerosfork

```

```

561          #1#3\XINT_max_zerozero % A = B = 0
562          #10\XINT_max_pluszero % B = 0, A > 0
563          #30\XINT_max_zeroplus % A = 0, B > 0
564          00\XINT_max_plusplus % A, B > 0
565      \krof }%
566  \krof
567  #3#1#2\xint:#4\xint:
568    \expandafter\xint_stop_atfirstoftwo
569  \else
570    \expandafter\xint_stop_atsecondoftwo
571  \fi
572  {#3#4}{#1#2}%
573 }%

```

Refactored at 1.2m for avoiding grabbing arguments. Position of inputs shared with *iiCmp* and *iiGeq* code.

```

574 \def\XINT_max_zerozero #1\fi{\xint_stop_atfirstoftwo }%
575 \def\XINT_max_zeroplus #1\fi{\xint_stop_atsecondoftwo }%
576 \def\XINT_max_pluszero #1\fi{\xint_stop_atfirstoftwo }%
577 \def\XINT_max_minusplus #1\fi{\xint_stop_atsecondoftwo }%
578 \def\XINT_max_plusminus #1\fi{\xint_stop_atfirstoftwo }%
579 \def\XINT_max_plusplus
580 {%
581   \if1\romannumeral0\XINT_geq_plusplus
582 }%

```

Premier des testés  $|A|=-A$ , second est  $|B|=-B$ . On veut le  $\max(A,B)$ , c'est donc A si  $|A|<|B|$  (ou  $|A|=|B|$ , mais peu importe alors). Donc on peut faire cela avec `\unless`. Simple.

```

583 \def\XINT_max_minusminus --%
584 {%
585   \unless\if1\romannumeral0\XINT_geq_plusplus{}\}%
586 }%

```

## 21.44. \xintiiMin

New with 1.09a. I add `\xintiiMin` in 1.1 and `\xintMin` is an *xintfrac* macro.

At 1.2m, a long-standing bug was fixed: `\xintiiMin` had the overhead of applying `\xintNum` to its arguments due to use of a sub-macro of `\xintGeq` code to which this overhead was added at some point.

And on this occasion I reduced even more number of times input is grabbed.

```

587 \def\xintiiMin {\romannumeral0\xintiimin }%
588 \def\xintiimin #1%
589 {%
590   \expandafter\xint_iimin \romannumeral`&&@#1\xint:
591 }%
592 \def\xint_iimin #1\xint:#2%
593 {%
594   \expandafter\XINT_min_fork\romannumeral`&&@#2\xint:#1\xint:
595 }%
596 \def\XINT_min_fork #1#2\xint:#3#4\xint:
597 {%
598   \xint_UDsignsfork
599     #1#3\XINT_min_minusminus % A < 0, B < 0
600     #1-\XINT_min_plusminus % B < 0, A >= 0

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

601      #3-\XINT_min_minusplus % A < 0, B >= 0
602      --{\xint_UDzerosfork
603          #1#3\XINT_min_zerozero % A = B = 0
604          #10\XINT_min_pluszero % B = 0, A > 0
605          #30\XINT_min_zeroplus % A = 0, B > 0
606          00\XINT_min_plusplus % A, B > 0
607      \krof }%
608  \krof
609  #3#1#2\xint:#4\xint:
610    \expandafter\xint_stop_atsecondoftwo
611  \else
612    \expandafter\xint_stop_atfirstoftwo
613  \fi
614  {#3#4}{#1#2}%
615 }%
616 \def\XINT_min_zerozero #1\fi{\xint_stop_atfirstoftwo }%
617 \def\XINT_min_zeroplus #1\fi{\xint_stop_atfirstoftwo }%
618 \def\XINT_min_pluszero #1\fi{\xint_stop_atsecondoftwo }%
619 \def\XINT_min_minusplus #1\fi{\xint_stop_atfirstoftwo }%
620 \def\XINT_min_plusminus #1\fi{\xint_stop_atsecondoftwo }%
621 \def\XINT_min_plusplus
622 {%
623   \if1\romannumeral0\XINT_geq_plusplus
624 }%
625 \def\XINT_min_minusminus --%
626 {%
627   \unless\if1\romannumeral0\XINT_geq_plusplus{}}%
628 }%
```

### 21.45. \xintiMaxof

New with 1.09a. 1.2 has NO MORE `\xintMaxof`, requires `\xintfracname`. 1.2a adds `\xintiMaxof`, as `\xintiMaxof.csv` is not public.

NOT compatible with empty list.

1.21 made `\xintiMaxof` robust against non terminated items.

1.4 refactors code to allow empty argument. For usage by `\xintiexpr`. Slight deterioration, will come back.

```

629 \def\xintiMaxof {\romannumeral0\xintiimaxof }%
630 \def\xintiimaxof #1{\expandafter\XINT_iimaxof\romannumeral`&&@#1^}%
631 \def\XINT_iimaxof{\romannumeral0\XINT_iimaxof}%
632 \def\XINT_iimaxof#1%
633 {%
634   \xint_gob_til_ ^ #1\XINT_iimaxof_empty ^%
635   \expandafter\XINT_iimaxof_loop\romannumeral`&&@#1\xint:
636 }%
637 \def\XINT_iimaxof_empty ^#1\xint:{ 0}%
638 \def\XINT_iimaxof_loop #1\xint:#2%
639 {%
640   \xint_gob_til_ ^ #2\XINT_iimaxof_e ^%
641   \expandafter\XINT_iimaxof_loop\romannumeral0\xintiimax{#1}{#2}\xint:
642 }%
643 \def\XINT_iimaxof_e ^#1\xintiimax #2#3\xint:{ #2}%

```

## 21.46. \xintiMinof

1.09a. 1.2a adds `\xintiMinof` which was lacking.

1.4 refactoring for `\xintiexpr` matters.

```

644 \def\xintiMinof {\romannumeral0\xintiiminof }%
645 \def\xintiiminof #1{\expandafter\XINT_iiminof\romannumeral`&&@#1^}%
646 \def\XINT_iMinof{\romannumeral0\XINT_iiminof}%
647 \def\XINT_iiminof#1%
648 {%
649   \xint_gob_til_ ^ #1\XINT_iiminof_empty ^%
650   \expandafter\XINT_iiminof_loop\romannumeral`&&@#1\xint:
651 }%
652 \def\XINT_iiminof_empty ^#1\xint:{ 0}%
653 \def\XINT_iiminof_loop #1\xint:#2%
654 {%
655   \xint_gob_til_ ^ #2\XINT_iiminof_e ^%
656   \expandafter\XINT_iiminof_loop\romannumeral0\xintiimin{#1}{#2}\xint:
657 }%
658 \def\XINT_iiminof_e ^#1\xintiimin #2#3\xint:{ #2}%

```

## 21.47. \xintiSum

`\xintiSum {{a}{b}...{z}}` Refactored at 1.4 for matters initially related to `xintexpr` delimiter choice.

```

659 \def\xintiSum {\romannumeral0\xintiisum }%
660 \def\xintiisum #1{\expandafter\XINT_iisum\romannumeral`&&@#1^}%
661 \def\XINT_iisum{\romannumeral0\XINT_iisum}%
662 \def\XINT_iisum #1%
663 {%
664   \expandafter\XINT_iisum_a\romannumeral`&&@#1\xint:
665 }%
666 \def\XINT_iisum_a #1%
667 {%
668   \xint_gob_til_ ^ #1\XINT_iisum_empty ^%
669   \XINT_iisum_loop #1%
670 }%
671 \def\XINT_iisum_empty ^#1\xint:{ 0}%

```

bad coding as it depends on internal conventions of `\XINT_add_nfork`

```

672 \def\XINT_iisum_loop #1#2\xint:#3%
673 {%
674   \expandafter\XINT_iisum_loop_a
675   \expandafter#1\romannumeral`&&@#3\xint:#2\xint:\xint:
676 }%
677 \def\XINT_iisum_loop_a #1#2%
678 {%
679   \xint_gob_til_ ^ #2\XINT_iisum_loop_end ^%
680   \expandafter\XINT_iisum_loop\romannumeral0\XINT_add_nfork #1#2%
681 }%

```

see previous comment!

```

682 \def\XINT_iisum_loop_end ^#1\XINT_add_nfork #2#3\xint:#4\xint:\xint:{ #2#4}%

```

## 21.48. \xintiiPrd

`\xintiiPrd` {{a}...{z}}

Macros renamed and refactored (slightly more macros here to supposedly bring micro-gain) at 1.4 to match changes in `xintfrac` of delimiter, in sync with some usage in `xintexpr`.

Contrarily to the `xintfrac` version `\xintPrd`, this one aborts as soon as it hits a zero value.

```
683 \def\xintiiPrd {\romannumeral0\xintiiprd }%
684 \def\xintiiprd #1{\expandafter\XINT_iiprd\romannumeral`&&@#1^}%
685 \def\XINT_iiprd{\romannumeral0\XINT_iiprd}%

The above romannumeral caused f-expansion of the list argument. We f-expand below the first item
and each successive items because we do not use \xintiiMul but jump directly into \XINT_mul_nfork.

686 \def\XINT_iiprd #1%
687 {%
688   \expandafter\XINT_iiprd_a\romannumeral`&&@#1\xint:
689 }%
690 \def\XINT_iiprd_a #1%
691 {%
692   \xint_gob_til_ ^ #1\XINT_iiprd_empty ^%
693   \xint_gob_til_zero #1\XINT_iiprd_zero 0%
694   \XINT_iiprd_loop #1%
695 }%
696 \def\XINT_iiprd_empty ^#1\xint:{ 1}%
697 \def\XINT_iiprd_zero 0#1^{ 0}%

bad coding as it depends on internal conventions of \XINT_mul_nfork

698 \def\XINT_iiprd_loop #1#2\xint:#3%
699 {%
700   \expandafter\XINT_iiprd_loop_a
701   \expandafter#1\romannumeral`&&@#3\xint:#2\xint:\xint:
702 }%
703 \def\XINT_iiprd_loop_a #1#2%
704 {%
705   \xint_gob_til_ ^ #2\XINT_iiprd_loop_end ^%
706   \xint_gob_til_zero #2\XINT_iiprd_zero 0%
707   \expandafter\XINT_iiprd_loop\romannumeral0\XINT_mul_nfork #1#2%
708 }%

see previous comment!

709 \def\XINT_iiprd_loop_end ^#1\XINT_mul_nfork #2#3\xint:#4\xint:\xint:{ #2#4}%
```

## 21.49. \xintiiSquareRoot

First done with 1.08.

1.1 added `\xintiiSquareRoot`.

1.1a added `\xintiiSqrtR`.

1.2f (2016/03/01-02-03) has rewritten the implementation, the underlying mathematics remaining about the same. The routine is much faster for inputs having up to 16 digits (because it does it all with `\numexpr` directly now), and also much faster for very long inputs (because it now fetches only the needed new digits after the first 16 (or 17) ones, via the geometric sequence 16, then 32, then 64, etc...; earlier version did the computations with all remaining digits after a suitable starting point with correct 4 or 5 leading digits). Note however that the fetching of tokens is via intrinsically  $O(N^2)$  macros, hence inevitably inputs with thousands of digits start being treated less well.

Actually there is some room for improvements, one could prepare better input X for the upcoming treatment of fetching its digits by 16, then 32, then 64, etc...

Incidentally, as `\xintiSqrt` uses subtraction and subtraction was broken from 1.2 to 1.2c, then for another reason from 1.2c to 1.2f, it could get wrong in certain (relatively rare) cases. There was also a bug that made it unneedlessly slow for odd number of digits on input.

1.2f also modifies `\xintFloatSqrt` in `xintfrac.sty` which now has more code in common with here and benefits from the same speed improvements.

1.2k belatedly corrects the output to `{1}{1}` and not 11 when input is zero. As braces are used in all other cases they should have been used here too.

Also, 1.2k adds an `\xintiSqrtR` macro, for coherence as `\xintiSqrt` is defined (and mentioned in user manual.)

```

710 \def\xintiSquareRoot {\romannumeral0\xintiisquareroot }%
711 \def\xintiisquareroot #1{\expandafter\XINT_sqrt_checkin\romannumeral`&&@#1\xint:}%
712 \def\XINT_sqrt_checkin #1%
713 {%
714   \xint_UDzerominusfork
715   #1-\XINT_sqrt_iszero
716   0#1\XINT_sqrt_isneg
717   0-\XINT_sqrt
718   \krof #1%
719 }%
720 \def\XINT_sqrt_iszero #1\xint:{{1}{1}}%
721 \def\XINT_sqrt_isneg #1\xint:
722   {\XINT_signalcondition{InvalidOperation}%
723     {Square root of negative: #1.}{{0}{0}}}%
724 \def\XINT_sqrt #1\xint:
725 {%
726   \expandafter\XINT_sqrt_start\romannumeral0\xintlenght {#1}.#1.%
727 }%
728 \def\XINT_sqrt_start #1.%
729 {%
730   \ifnum #1<\xint_c_x\xint_dothis\XINT_sqrt_small_a\fi
731   \xint_orthat\XINT_sqrt_big_a #1.%
732 }%
733 \def\XINT_sqrt_small_a #1.{\XINT_sqrt_a #1.\XINT_sqrt_small_d }%
734 \def\XINT_sqrt_big_a #1.{\XINT_sqrt_a #1.\XINT_sqrt_big_d }%
735 \def\XINT_sqrt_a #1.%
736 {%
737   \ifodd #1
738     \expandafter\XINT_sqrt_b0
739   \else
740     \expandafter\XINT_sqrt_bE
741   \fi
742   #1.%
743 }%
744 \def\XINT_sqrt_bE #1.#2#3#4%
745 {%
746   \XINT_sqrt_c {#3#4}#2{#1}#3#4%
747 }%
748 \def\XINT_sqrt_b0 #1.#2#3%
749 {%
750   \XINT_sqrt_c #3#2{#1}#3%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

751 }%
752 \def\XINT_sqrt_c #1#2%
753 {%
754     \expandafter #2%
755     \the\numexpr \ifnum #1>\xint_c_ii
756         \ifnum #1>\xint_c_vi
757             \ifnum #1>12 \ifnum #1>20 \ifnum #1>30
758             \ifnum #1>42 \ifnum #1>56 \ifnum #1>72
759             \ifnum #1>90
760             10\else 9\fi \else 8\fi \else 7\fi \else 6\fi \else 5\fi
761             \else 4\fi \else 3\fi \else 2\fi \else 1\fi .%
762 }%
763 \def\XINT_sqrt_small_d #1.#2%
764 {%
765     \expandafter\XINT_sqrt_small_e
766     \the\numexpr #1\ifcase \numexpr #2/\xint_c_ii-\xint_c_i\relax
767         \or 0\or 00\or 000\or 0000\fi .%
768 }%
769 \def\XINT_sqrt_small_e #1.#2.%
770 {%
771     \expandafter\XINT_sqrt_small_ea\the\numexpr #1*#1-#2.#1.%
772 }%
773 \def\XINT_sqrt_small_ea #1%
774 {%
775     \if0#1\xint_dothis\XINT_sqrt_small_ez\fi
776     \if-#1\xint_dothis\XINT_sqrt_small_eb\fi
777     \xint_orthat\XINT_sqrt_small_f #1%
778 }%
779 \def\XINT_sqrt_small_ez 0.#1.{\expandafter{\the\numexpr#1+\xint_c_i
780     \expandafter}\expandafter{\the\numexpr #1*\xint_c_ii+\xint_c_i}}%
781 \def\XINT_sqrt_small_eb -#1.#2.%
782 {%
783     \expandafter\XINT_sqrt_small_ec \the\numexpr
784     (#1-\xint_c_i+#2)/(\xint_c_ii*#2).#1.#2.%
785 }%
786 \def\XINT_sqrt_small_ec #1.#2.#3.%
787 {%
788     \expandafter\XINT_sqrt_small_f \the\numexpr
789     -#2+\xint_c_ii*#3*#1+#1*#1\expandafter.\the\numexpr #3+#1.%
790 }%
791 \def\XINT_sqrt_small_f #1.#2.%
792 {%
793     \expandafter\XINT_sqrt_small_g
794     \the\numexpr (#1+#2)/(\xint_c_ii*#2)-\xint_c_i.#1.#2.%
795 }%
796 \def\XINT_sqrt_small_g #1#2.%
797 {%
798     \if 0#1%
799         \expandafter\XINT_sqrt_small_end
800     \else
801         \expandafter\XINT_sqrt_small_h
802     \fi

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

803     #1#2.%
804 }%
805 \def\XINT_sqrt_small_h #1.#2.#3.%
806 {%
807     \expandafter\XINT_sqrt_small_f
808     \the\numexpr #2-\xint_c_ii*#1*#3+#1*#1\expandafter.%
809     \the\numexpr #3-#1.%
810 }%
811 \def\XINT_sqrt_small_end #1.#2.#3.{\#3}{\#2}}%
812 \def\XINT_sqrt_big_d #1.#2%
813 {%
814     \ifodd #2 \xint_dothis{\expandafter\XINT_sqrt_big_e0}\fi
815     \xint_orthat{\expandafter\XINT_sqrt_big_eE}%
816     \the\numexpr (#2-\xint_c_i)/\xint_c_ii.#1;%
817 }%
818 \def\XINT_sqrt_big_eE #1;#2#3#4#5#6#7#8#9%
819 {%
820     \XINT_sqrt_big_eE_a #1;{\#2#3#4#5#6#7#8#9}%
821 }%
822 \def\XINT_sqrt_big_eE_a #1.#2;#3%
823 {%
824     \expandafter\XINT_sqrt_bigormed_f
825     \romannumeral0\XINT_sqrt_small_e #2000.#3.#1;%
826 }%
827 \def\XINT_sqrt_big_e0 #1;#2#3#4#5#6#7#8#9%
828 {%
829     \XINT_sqrt_big_e0_a #1;{\#2#3#4#5#6#7#8#9}%
830 }%
831 \def\XINT_sqrt_big_e0_a #1.#2;#3#4%
832 {%
833     \expandafter\XINT_sqrt_bigormed_f
834     \romannumeral0\XINT_sqrt_small_e #20000.#3#4.#1;%
835 }%
836 \def\XINT_sqrt_bigormed_f #1#2#3;%
837 {%
838     \ifnum#3<\xint_c_ix
839         \xint_dothis {\csname XINT_sqrt_med_f\romannumeral#3\endcsname}%
840     \fi
841     \xint_orthat\XINT_sqrt_big_f #1.#2.#3;%
842 }%
843 \def\XINT_sqrt_med_fv {\XINT_sqrt_med_fa .}%
844 \def\XINT_sqrt_med_fvi {\XINT_sqrt_med_fa 0.}%
845 \def\XINT_sqrt_med_fvii {\XINT_sqrt_med_fa 00.}%
846 \def\XINT_sqrt_med_fviii{\XINT_sqrt_med_fa 000.}%
847 \def\XINT_sqrt_med_fa #1.#2.#3.#4;%
848 {%
849     \expandafter\XINT_sqrt_med_fb
850     \the\numexpr (#30#1-5#1)/(\xint_c_ii*#2).#1.#2.#3.%
851 }%
852 \def\XINT_sqrt_med_fb #1.#2.#3.#4.#5.%
853 {%
854     \expandafter\XINT_sqrt_small_ea

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

855 \the\numexpr (#40#2-\xint_c_ii*#3*#1)*10#2+(#1*#1-#5)\expandafter.%
856 \the\numexpr #30#2-#1.%
857 }%
858 \def\xINT_sqrt_big_f #1;#2#3#4#5#6#7#8#9%
859 {%
860 \xINT_sqrt_big_fa #1;{#2#3#4#5#6#7#8#9}%
861 }%
862 \def\xINT_sqrt_big_fa #1.#2.#3;#4%
863 {%
864 \expandafter\xINT_sqrt_big_ga
865 \the\numexpr #3-\xint_c_viii\expandafter.%
866 \romannumeral0\xINT_sqrt_med_fa 000.#1.#2.;#4.%
867 }%

868 \def\xINT_sqrt_big_ga #1.#2#3%
869 {%
870 \ifnum #1>\xint_c_viii
871 \expandafter\xINT_sqrt_big_gb\else
872 \expandafter\xINT_sqrt_big_ka
873 \fi #1.#3.#2.%
874 }%
875 \def\xINT_sqrt_big_gb #1.#2.#3.%
876 {%
877 \expandafter\xINT_sqrt_big_gc
878 \the\numexpr (\xint_c_ii*#2-\xint_c_i)*\xint_c_x^viii/(\xint_c_iv*#3).%
879 #3.#2.#1;%
880 }%
881 \def\xINT_sqrt_big_gc #1.#2.#3.%
882 {%
883 \expandafter\xINT_sqrt_big_gd
884 \romannumeral0\xintiadd
885 {\xintiiSub {#3000000000}{\xintDouble{\xintiiMul{#2}{#1}}00000000}%
886 {\xintiiSqr {#1}}.%
887 \romannumeral0\xintiisub{#2000000000}{#1}.%
888 }%
889 \def\xINT_sqrt_big_gd #1.#2.%
890 {%
891 \expandafter\xINT_sqrt_big_ge #2.#1.%
892 }%
893 \def\xINT_sqrt_big_ge #1;#2#3#4#5#6#7#8#9%
894 {\xINT_sqrt_big_gf #1.#2#3#4#5#6#7#8#9;%}
895 \def\xINT_sqrt_big_gf #1;#2#3#4#5#6#7#8#9%
896 {\xINT_sqrt_big_gg #1#2#3#4#5#6#7#8#9.}%
897 \def\xINT_sqrt_big_gg #1.#2.#3.#4.%
898 {%
899 \expandafter\xINT_sqrt_big_gloop
900 \expandafter\xint_c_xvi\expandafter.%
901 \the\numexpr #3-\xint_c_viii\expandafter.%
902 \romannumeral0\xintiisub {#2}{\xintiNum{#4}}.#1.%
903 }%
904 \def\xINT_sqrt_big_gloop #1.#2.%
905 {%
906 \unless\ifnum #1<#2 \xint_dothis\xINT_sqrt_big_ka \fi

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
907 \xint_orthat{\XINT_sqrt_big_gi #1.}#2.%
908 }%
909 \def\XINT_sqrt_big_gi #1.%
910 {%
911 \expandafter\XINT_sqrt_big_gj\romannumeral\xintreplicate{#1}0.#1.%
912 }%
913 \def\XINT_sqrt_big_gj #1.#2.#3.#4.#5.%
914 {%
915 \expandafter\XINT_sqrt_big_gk
916 \romannumeral0\xintiidivision {#4#1}%
917 {\XINT_dbl #5\xint_bye2345678\xint_bye*\xint_c_ii\relax}.%
918 #1.#5.#2.#3.%
919 }%
920 \def\XINT_sqrt_big_gk #1#2.#3.#4.%
921 {%
922 \expandafter\XINT_sqrt_big_gl
923 \romannumeral0\xintiadd {#2#3}{\xintiiSqr{#1}}.%
924 \romannumeral0\xintiisub {#4#3}{#1}.%
925 }%
926 \def\XINT_sqrt_big_gl #1.#2.%
927 {%
928 \expandafter\XINT_sqrt_big_gm #2.#1.%
929 }%
930 \def\XINT_sqrt_big_gm #1.#2.#3.#4.#5.%
931 {%
932 \expandafter\XINT_sqrt_big_gn
933 \romannumeral0\XINT_split_fromleft\xint_c_ii*#3.#5\xint_bye2345678\xint_bye..%
934 #1.#2.#3.#4.%
935 }%
936 \def\XINT_sqrt_big_gn #1.#2.#3.#4.#5.#6.%
937 {%
938 \expandafter\XINT_sqrt_big_gloop
939 \the\numexpr \xint_c_ii*#5\expandafter.%
940 \the\numexpr #6-#5\expandafter.%
941 \romannumeral0\xintiisub{#4}{\xintiNum{#1}}.#3.#2.%
942 }%
943 \def\XINT_sqrt_big_ka #1.#2.#3.#4.%
944 {%
945 \expandafter\XINT_sqrt_big_kb
946 \romannumeral0\XINT_dsx_addzeros {#1}#3;.%
947 \romannumeral0\xintiisub
948 {\XINT_dsx_addzerosnofuss {\xint_c_ii*#1}#2;}%
949 {\xintiNum{#4}}.%
950 }%
951 \def\XINT_sqrt_big_kb #1.#2.%
952 {%
953 \expandafter\XINT_sqrt_big_kc #2.#1.%
954 }%
955 \def\XINT_sqrt_big_kc #1%
956 {%
957 \if0#1\xint_dothis\XINT_sqrt_big_kz\fi
958 \xint_orthat\XINT_sqrt_big_kloop #1%
```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

959 }%
960 \def\XINT_sqrt_big_kz 0.#1.%
961 {%
962     \expandafter\XINT_sqrt_big_kend
963     \romannumeral0%
964     \xintinc{\XINT_dbl#1\xint_bye2345678\xint_bye*\xint_c_ii\relax}.#1.%
965 }%
966 \def\XINT_sqrt_big_kend #1.#2.%
967 {%
968     \expandafter{\romannumeral0\xintinc{#2}}{#1}%
969 }%
970 \def\XINT_sqrt_big_kloop #1.#2.%
971 {%
972     \expandafter\XINT_sqrt_big_ke
973     \romannumeral0\xintiidevision{#1}%
974     {\romannumeral0\XINT_dbl #2\xint_bye2345678\xint_bye*\xint_c_ii\relax}{#2}%
975 }%
976 \def\XINT_sqrt_big_ke #1%
977 {%
978     \if0\XINT_Sgn #1\xint:
979         \expandafter \XINT_sqrt_big_end
980     \else \expandafter \XINT_sqrt_big_kf
981     \fi {#1}%
982 }%
983 \def\XINT_sqrt_big_kf #1#2#3%
984 {%
985     \expandafter\XINT_sqrt_big_kg
986     \romannumeral0\xintiisub {#3}{#1}.%
987     \romannumeral0\xintiiadd {#2}{\xintiiSqr {#1}}.%
988 }%
989 \def\XINT_sqrt_big_kg #1.#2.%
990 {%
991     \expandafter\XINT_sqrt_big_kloop #2.#1.%
992 }%
993 \def\XINT_sqrt_big_end #1#2#3{{#3}{#2}}%

```

### 21.50. \xintiiSqr, \xintiiSqrR

```

994 \def\xintiiSqr {\romannumeral0\xintiisqr }%
995 \def\xintiisqr {\expandafter\XINT_sqrt_post\romannumeral0\xintiisquareroot }%
996 \def\XINT_sqrt_post #1#2{\XINT_dec #1\XINT_dec_bye234567890\xint_bye}%
997 \def\xintiiSqrR {\romannumeral0\xintiisqrtr }%
998 \def\xintiisqrtr {\expandafter\XINT_sqrtr_post\romannumeral0\xintiisquareroot }%

```

$N = (\#1)^2 - \#2$  avec  $\#1$  le plus petit possible et  $\#2 > 0$  (hence  $\#2 < 2 * \#1$ ).  $(\#1 - .5)^2 = \#1^2 - \#1 + .25 = N + \#2 - \#1 + .25$ . Si  $0 < \#2 < \#1$ ,  $<= N - 0.75 < N$ , donc rounded-> $\#1$  si  $\#2 \geq \#1$ ,  $(\#1 - .5)^2 \geq N + .25 > N$ , donc rounded-> $\#1 - 1$ .

```

999 \def\XINT_sqrtr_post #1#2%
1000     {\xintiiiflt {#2}{#1}{ #1}{\XINT_dec #1\XINT_dec_bye234567890\xint_bye}}%

```

### 21.51. \xintiiBinomial

2015/11/28-29 for 1.2f.

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

2016/11/19 for 1.2h: I truly can't understand why I hard-coded last year an error-message for arguments outside of the range for binomial formula. Naturally there should be no error but a rather a 0 return value for  $\text{binomial}(x,y)$ , if  $y < 0$  or  $x < y$  !

I really lack some kind of infinity or NaN value.

```
1001 \def\xintiibinomial {\romannumeral0\xintiibinomial }%
1002 \def\xintiibinomial #1#2%
1003 {%
1004   \expandafter\XINT_binom_pre\the\numexpr #1\expandafter.\the\numexpr #2.%
1005 }%
1006 \def\XINT_binom_pre #1.#2.%
1007 {%
1008   \expandafter\XINT_binom_fork \the\numexpr#1-#2.#2.#1.%
1009 }%
```

k.x-k.x. I hesitated to restrict maximal allowed value of x to 10000. Finally I don't. But due to using small multiplication and small division, x must have at most eight digits. If  $x \geq 2^{31}$  an arithmetic overflow error will have happened already.

```
1010 \def\XINT_binom_fork #1#2.#3#4.#5#6.%
1011 {%
1012   \if-#5\xint_dothis{\XINT_signalcondition{InvalidOperation}%
1013     {Binomial with negative first argument: #5#6.}}{ 0}}\fi
1014   \if-#1\xint_dothis{ 0}\fi
1015   \if-#3\xint_dothis{ 0}\fi
1016   \if0#1\xint_dothis{ 1}\fi
1017   \if0#3\xint_dothis{ 1}\fi
1018   \ifnum #5#6>\xint_c_x^viii_mone\xint_dothis
1019     {\XINT_signalcondition{InvalidOperation}%
1020       {Binomial with too large argument: #5#6 >= 10^8.}}{ 0}}\fi
1021   \ifnum #1#2>#3#4 \xint_dothis{\XINT_binom_a #1#2.#3#4.}\fi
1022   \xint_orthat{\XINT_binom_a #3#4.#1#2.}%
1023 }%
```

x-k.k. avec  $0 < k < x$ ,  $k \leq x - k$ . Les divisions produiront en extra après le quotient un terminateur 1!\Z!0!. On va procéder par petite multiplication suivie par petite division. Donc ici on met le 1!\Z!0! pour amorcer.

Le `\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax` est le terminateur pour le `\XINT_unsep_cuzsmall` final.

```
1024 \def\XINT_binom_a #1.#2.%
1025 {%
1026   \expandafter\XINT_binom_b\the\numexpr \xint_c_i+#1.1.#2.100000001!1!;!0!%
1027 }%
```

$y = x - k + 1$ .  $j = 1$ .  $k$ . On va évaluer par  $y/1 * (y+1)/2 * (y+2)/3$  etc... On essaie de regrouper de manière à utiliser au mieux `\numexpr`. On peut aller jusqu'à  $x = 10000$  car  $9999 * 10000 < 10^8$ .  $463 * 464 * 465 = 99896880$ ,  $98 * 99 * 100 * 101 = 97990200$ . On va vérifier à chaque étape si on dépasse un seuil. Le style de l'implémentation diffère de celui que j'avais utilisé pour `\xintiifac`. On pourrait tout-à-fait avoir une `verybigloop`, mais bon. Je rajoute aussi un `verysmall`. Le traitement est un peu différent pour elle afin d'aller jusqu'à  $x = 29$  (et pas seulement 26 si je suivais le modèle des autres, mais je veux pouvoir faire  $\text{binomial}(29,1)$ ,  $\text{binomial}(29,2)$ , ... en `vsmall`).

```
1028 \def\XINT_binom_b #1.%
1029 {%
1030   \ifnum #1>9999 \xint_dothis\XINT_binom_vbigloop \fi
1031   \ifnum #1>463 \xint_dothis\XINT_binom_bigloop \fi
1032   \ifnum #1>98 \xint_dothis\XINT_binom_medloop \fi
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1033 \ifnum #1>29 \xint_dothis\XINT_binom_smallloop \fi
1034 \xint_orthat\XINT_binom_vsmallloop #1.%
1035 }%
```

y.j.k. Au départ on avait x-k+1.1.k. Ensuite on a des blocs 1<8d>! donnant le résultat intermédiaire, dans l'ordre, et à la fin on a 1!1;!0!. Dans smallloop on peut prendre 4 par 4.

```
1036 \def\XINT_binom_smallloop #1.#2.#3.%
1037 {%
1038 \ifcase\numexpr #3-#2\relax
1039 \expandafter\XINT_binom_end_
1040 \or \expandafter\XINT_binom_end_i
1041 \or \expandafter\XINT_binom_end_ii
1042 \or \expandafter\XINT_binom_end_iii
1043 \else\expandafter\XINT_binom_smallloop_a
1044 \fi #1.#2.#3.%
1045 }%
```

Ça m'ennuie un peu de reprendre les #1, #2, #3 ici. On a besoin de `\numexpr` pour `\XINT_binom_div`, mais de `\romannumeral0` pour le unsep après `\XINT_binom_mul`.

```
1046 \def\XINT_binom_smallloop_a #1.#2.#3.%
1047 {%
1048 \expandafter\XINT_binom_smallloop_b
1049 \the\numexpr #1+\xint_c_iv\expandafter.%
1050 \the\numexpr #2+\xint_c_iv\expandafter.%
1051 \the\numexpr #3\expandafter.%
1052 \the\numexpr\expandafter\XINT_binom_div
1053 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
1054 !\romannumeral0\expandafter\XINT_binom_mul
1055 \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
1056 }%
1057 \def\XINT_binom_smallloop_b #1.%
1058 {%
1059 \ifnum #1>98 \expandafter\XINT_binom_medloop \else
1060 \expandafter\XINT_binom_smallloop \fi #1.%
1061 }%
```

Ici on prend trois par trois.

```
1062 \def\XINT_binom_medloop #1.#2.#3.%
1063 {%
1064 \ifcase\numexpr #3-#2\relax
1065 \expandafter\XINT_binom_end_
1066 \or \expandafter\XINT_binom_end_i
1067 \or \expandafter\XINT_binom_end_ii
1068 \else\expandafter\XINT_binom_medloop_a
1069 \fi #1.#2.#3.%
1070 }%
1071 \def\XINT_binom_medloop_a #1.#2.#3.%
1072 {%
1073 \expandafter\XINT_binom_medloop_b
1074 \the\numexpr #1+\xint_c_iii\expandafter.%
1075 \the\numexpr #2+\xint_c_iii\expandafter.%
1076 \the\numexpr #3\expandafter.%
1077 \the\numexpr\expandafter\XINT_binom_div
1078 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1079      !\romannumeral0\expandafter\XINT_binom_mul
1080      \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
1081 }%
1082 \def\XINT_binom_medloop_b #1.%
1083 {%
1084     \ifnum #1>463 \expandafter\XINT_binom_bigloop   \else
1085                 \expandafter\XINT_binom_medloop     \fi #1.%
1086 }%
    Ici on prend deux par deux.
1087 \def\XINT_binom_bigloop #1.#2.#3.%
1088 {%
1089     \ifcase\numexpr #3-#2\relax
1090         \expandafter\XINT_binom_end_
1091     \or \expandafter\XINT_binom_end_i
1092     \else\expandafter\XINT_binom_bigloop_a
1093     \fi #1.#2.#3.%
1094 }%
1095 \def\XINT_binom_bigloop_a #1.#2.#3.%
1096 {%
1097     \expandafter\XINT_binom_bigloop_b
1098     \the\numexpr #1+\xint_c_ii\expandafter.%
1099     \the\numexpr #2+\xint_c_ii\expandafter.%
1100     \the\numexpr #3\expandafter.%
1101     \the\numexpr\expandafter\XINT_binom_div
1102         \the\numexpr #2*(#2+\xint_c_i)\expandafter
1103     !\romannumeral0\expandafter\XINT_binom_mul
1104     \the\numexpr #1*(#1+\xint_c_i)!%
1105 }%
1106 \def\XINT_binom_bigloop_b #1.%
1107 {%
1108     \ifnum #1>9999 \expandafter\XINT_binom_vbigloop \else
1109                 \expandafter\XINT_binom_bigloop   \fi #1.%
1110 }%
    Et finalement un par un.
1111 \def\XINT_binom_vbigloop #1.#2.#3.%
1112 {%
1113     \ifnum #3=#2
1114         \expandafter\XINT_binom_end_
1115     \else\expandafter\XINT_binom_vbigloop_a
1116     \fi #1.#2.#3.%
1117 }%
1118 \def\XINT_binom_vbigloop_a #1.#2.#3.%
1119 {%
1120     \expandafter\XINT_binom_vbigloop
1121     \the\numexpr #1+\xint_c_i\expandafter.%
1122     \the\numexpr #2+\xint_c_i\expandafter.%
1123     \the\numexpr #3\expandafter.%
1124     \the\numexpr\expandafter\XINT_binom_div\the\numexpr #2\expandafter
1125     !\romannumeral0\XINT_binom_mul #1!%
1126 }%

```

y.j.k. La partie very small. y est au plus 26 (non 29 mais retesté dans `\XINT_binom_vsmallloop_a`), et tous les binomial(29,n) sont  $<10^8$ . On peut donc faire  $y(y+1)(y+2)(y+3)$  et aussi il y a

```

1127 \def\XINT_binom_vsmallloop #1.#2.#3.%
1128 {%
1129     \ifcase\numexpr #3-#2\relax
1130         \expandafter\XINT_binom_vsmallend_
1131     \or \expandafter\XINT_binom_vsmallend_i
1132     \or \expandafter\XINT_binom_vsmallend_ii
1133     \or \expandafter\XINT_binom_vsmallend_iii
1134     \else\expandafter\XINT_binom_vsmallloop_a
1135     \fi #1.#2.#3.%
1136 }%
1137 \def\XINT_binom_vsmallloop_a #1.%
1138 {%
1139     \ifnum #1>26 \expandafter\XINT_binom_smallloop_a \else
1140         \expandafter\XINT_binom_vsmallloop_b \fi #1.%
1141 }%
1142 \def\XINT_binom_vsmallloop_b #1.#2.#3.%
1143 {%
1144     \expandafter\XINT_binom_vsmallloop
1145     \the\numexpr #1+\xint_c_iv\expandafter.%
1146     \the\numexpr #2+\xint_c_iv\expandafter.%
1147     \the\numexpr #3\expandafter.%
1148     \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
1149     \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
1150     !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
1151 }%
1152 \def\XINT_binom_mul #1!#2!;!0!%
1153 {%
1154     \expandafter\XINT_rev_nounsep\expandafter{\expandafter}%
1155     \the\numexpr\expandafter\XINT_smallmul
1156     \the\numexpr\xint_c_x^viii+#1\expandafter
1157     !\romannumeral0\XINT_rev_nounsep {}1;!#2%
1158     \R!\R!\R!\R!\R!\R!\R!\R!\R!\W
1159     \R!\R!\R!\R!\R!\R!\R!\R!\R!\W
1160     1;!%
1161 }%
1162 \def\XINT_binom_div #1!1;!%
1163 {%
1164     \expandafter\XINT_smalldivx_a
1165     \the\numexpr #1/\xint_c_ii\expandafter\xint:
1166     \the\numexpr \xint_c_x^viii+#1!%
1167 }%

```

Vaguement envisagé d'éviter le  $10^8$  mais bon.

```

1168 \def\XINT_binom_vsmallmuldiv #1!#2!1#3!{\xint_c_x^viii+#2*#3/#1!}%

```

On a des terminaisons communes aux trois situations small, med, big, et on est sûr de p  
les multiplications dans \numexpr, car on vient ici \*après\* avoir comparé à 9999 ou 4

```

1169 \def\XINT_binom_end_iii #1.#2.#3.%
1170 {%
1171     \expandafter\XINT_binom_finish
1172     \the\numexpr\expandafter\XINT_binom_div

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1173 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
1174 !\romannumeral0\expandafter\XINT_binom_mul
1175 \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
1176 }%
1177 \def\XINT_binom_end_ii #1.#2.#3.%
1178 {%
1179 \expandafter\XINT_binom_finish
1180 \the\numexpr\expandafter\XINT_binom_div
1181 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter
1182 !\romannumeral0\expandafter\XINT_binom_mul
1183 \the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
1184 }%
1185 \def\XINT_binom_end_i #1.#2.#3.%
1186 {%
1187 \expandafter\XINT_binom_finish
1188 \the\numexpr\expandafter\XINT_binom_div
1189 \the\numexpr #2*(#2+\xint_c_i)\expandafter
1190 !\romannumeral0\expandafter\XINT_binom_mul
1191 \the\numexpr #1*(#1+\xint_c_i)!%
1192 }%
1193 \def\XINT_binom_end_ #1.#2.#3.%
1194 {%
1195 \expandafter\XINT_binom_finish
1196 \the\numexpr\expandafter\XINT_binom_div\the\numexpr #2\expandafter
1197 !\romannumeral0\XINT_binom_mul #1!%
1198 }%
1199 \def\XINT_binom_finish #1;!0!%
1200 {\XINT_unsep_cuzsmall #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_c_i\relax}%
    Duplication de code seulement pour la boucle avec très petits coeffs, mais en plus on fait au
    maximum des possibilités. (on pourrait tester plus le résultat déjà obtenu).
1201 \def\XINT_binom_vsmallend_iii #1.%
1202 {%
1203 \ifnum #1>26 \expandafter\XINT_binom_end_iii \else
1204 \expandafter\XINT_binom_vsmallend_iiib \fi #1.%
1205 }%
1206 \def\XINT_binom_vsmallend_iiib #1.#2.#3.%
1207 {%
1208 \expandafter\XINT_binom_vsmallfinish
1209 \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
1210 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)*(#2+\xint_c_iii)\expandafter
1211 !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
1212 }%
1213 \def\XINT_binom_vsmallend_ii #1.%
1214 {%
1215 \ifnum #1>27 \expandafter\XINT_binom_end_ii \else
1216 \expandafter\XINT_binom_vsmallend_iib \fi #1.%
1217 }%
1218 \def\XINT_binom_vsmallend_iib #1.#2.#3.%
1219 {%
1220 \expandafter\XINT_binom_vsmallfinish
1221 \the\numexpr \expandafter\XINT_binom_vsmallmuldiv
1222 \the\numexpr #2*(#2+\xint_c_i)*(#2+\xint_c_ii)\expandafter

```

```

1223      !\the\numexpr #1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
1224 }%
1225 \def\xINT_binom_vsmallend_i #1.%
1226 {%
1227     \ifnum #1>28 \expandafter\xINT_binom_end_i \else
1228         \expandafter\xINT_binom_vsmallend_ib \fi #1.%
1229 }%
1230 \def\xINT_binom_vsmallend_ib #1.#2.#3.%
1231 {%
1232     \expandafter\xINT_binom_vsmallfinish
1233     \the\numexpr \expandafter\xINT_binom_vsmallmuldiv
1234     \the\numexpr #2*(#2+\xint_c_i)\expandafter
1235     !\the\numexpr #1*(#1+\xint_c_i)!%
1236 }%
1237 \def\xINT_binom_vsmallend_ #1.%
1238 {%
1239     \ifnum #1>29 \expandafter\xINT_binom_end_ \else
1240         \expandafter\xINT_binom_vsmallend_b \fi #1.%
1241 }%
1242 \def\xINT_binom_vsmallend_b #1.#2.#3.%
1243 {%
1244     \expandafter\xINT_binom_vsmallfinish
1245     \the\numexpr\xINT_binom_vsmallmuldiv #2!#1!%
1246 }%
1247 \def\xINT_binom_vsmallfinish#1{%
1248 \def\xINT_binom_vsmallfinish1##1!1!;!0!{\expandafter#1\the\numexpr##1\relax}%
1249 }\xINT_binom_vsmallfinish{ }%

```

## 21.52. \xintiipFactorial

2015/11/29 for 1.2f. Partial factorial pfac(a,b)=(a+1)...b, only for non-negative integers with  $a \leq b < 10^8$ .

1.2h (2016/11/20) removes the non-negativity condition. It was a bit unfortunate that the code raised `\xintError:OutOfRangePFac` if  $0 \leq a \leq b < 10^8$  was violated. The rule now applied is to interpret pfac(a,b) as the product for  $a < j \leq b$  (not as a ratio of Gamma function), hence if  $a \geq b$ , return 1 because of an empty product. If  $a < b$ : if  $a < 0$ , return 0 for  $b \geq 0$  and  $(-1)^{(b-a)}$  times  $|b| \dots (|a|-1)$  for  $b < 0$ . But only for the range  $0 \leq a \leq b < 10^8$  is the macro result to be considered as stable.

```

1250 \def\xintiipFactorial {\romannumeral0\xintiipfactorial }%
1251 \def\xintiipfactorial #1#2%
1252 {%
1253     \expandafter\xINT_pfac_fork\the\numexpr#1\expandafter.\the\numexpr #2.%
1254 }%
1255 \def\xintPFactorial{\romannumeral0\xintpfactorial}%
1256 \let\xintpfactorial\xintiipfactorial

```

Code is a simplified version of the one for `\xintiiBinomial`, with no attempt at implementing a "very small" branch.

```

1257 \def\xINT_pfac_fork #1#2.#3#4.%
1258 {%
1259     \unless\ifnum #1#2<#3#4 \xint_dothis\xINT_pfac_one\fi
1260     \if-#3\xint_dothis\xINT_pfac_neg\fi
1261     \if-#1\xint_dothis\xINT_pfac_zero\fi
1262     \ifnum #3#4>\xint_c_x^viii_mone\xint_dothis\xINT_pfac_outofrange\fi

```

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

391



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

1315 \ifcase\numexpr #2-#1\relax
1316 \expandafter\XINT_pfac_end_
1317 \or \expandafter\XINT_pfac_end_i
1318 \or \expandafter\XINT_pfac_end_ii
1319 \else\expandafter\XINT_pfac_medloop_a
1320 \fi #1.#2.%
1321 }%
1322 \def\XINT_pfac_medloop_a #1.#2.%
1323 {%
1324 \expandafter\XINT_pfac_medloop_b
1325 \the\numexpr #1+\xint_c_iii\expandafter.%
1326 \the\numexpr #2\expandafter.%
1327 \the\numexpr\expandafter\XINT_smallmul
1328 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
1329 }%
1330 \def\XINT_pfac_medloop_b #1.%
1331 {%
1332 \ifnum #1>463 \expandafter\XINT_pfac_bigloop \else
1333 \expandafter\XINT_pfac_medloop \fi #1.%
1334 }%
1335 \def\XINT_pfac_bigloop #1.#2.%
1336 {%
1337 \ifcase\numexpr #2-#1\relax
1338 \expandafter\XINT_pfac_end_
1339 \or \expandafter\XINT_pfac_end_i
1340 \else\expandafter\XINT_pfac_bigloop_a
1341 \fi #1.#2.%
1342 }%
1343 \def\XINT_pfac_bigloop_a #1.#2.%
1344 {%
1345 \expandafter\XINT_pfac_bigloop_b
1346 \the\numexpr #1+\xint_c_ii\expandafter.%
1347 \the\numexpr #2\expandafter.%
1348 \the\numexpr\expandafter
1349 \XINT_smallmul\the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
1350 }%
1351 \def\XINT_pfac_bigloop_b #1.%
1352 {%
1353 \ifnum #1>9999 \expandafter\XINT_pfac_vbigloop \else
1354 \expandafter\XINT_pfac_bigloop \fi #1.%
1355 }%
1356 \def\XINT_pfac_vbigloop #1.#2.%
1357 {%
1358 \ifnum #2=#1
1359 \expandafter\XINT_pfac_end_
1360 \else\expandafter\XINT_pfac_vbigloop_a
1361 \fi #1.#2.%
1362 }%
1363 \def\XINT_pfac_vbigloop_a #1.#2.%
1364 {%
1365 \expandafter\XINT_pfac_vbigloop
1366 \the\numexpr #1+\xint_c_i\expandafter.%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1367 \the\numexpr #2\expandafter.%
1368 \the\numexpr\expandafter\XINT_smallmul\the\numexpr\xint_c_x^viii+#1!%
1369 }%
1370 \def\XINT_pfac_end_iii #1.#2.%
1371 {%
1372 \expandafter\XINT_mul_out
1373 \the\numexpr\expandafter\XINT_smallmul
1374 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
1375 }%
1376 \def\XINT_pfac_end_ii #1.#2.%
1377 {%
1378 \expandafter\XINT_mul_out
1379 \the\numexpr\expandafter\XINT_smallmul
1380 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
1381 }%
1382 \def\XINT_pfac_end_i #1.#2.%
1383 {%
1384 \expandafter\XINT_mul_out
1385 \the\numexpr\expandafter\XINT_smallmul
1386 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
1387 }%
1388 \def\XINT_pfac_end_ #1.#2.%
1389 {%
1390 \expandafter\XINT_mul_out
1391 \the\numexpr\expandafter\XINT_smallmul\the\numexpr \xint_c_x^viii+#1!%
1392 }%
```

### 21.53. \xintBool, \xintToggle

1.09c

```
1393 \def\xintBool #1{\romannumeral`&&@%
1394 \csname if#1\endcsname\expandafter1\else\expandafter0\fi }%
1395 \def\xintToggle #1{\romannumeral`&&@\iftoggle{#1}{1}{0}}%
```

### 21.54. \xintiiGCD

**1.3d:** [\xintiiGCD](#) code from [xintgcd](#) is copied here to support [gcd\(\)](#) function in [\xintiiexpr](#).

**1.4:** removed from [xintgcd](#) the original caode as now [xintgcd](#) loads [xint](#).

**Modified at 1.4d (2021/03/29).** Damn'ed! Since **1.3d** (2019/01/06) the code was broken if one of the arguments vanished due to a typo in macro names: "AisZero" at one location and "Aiszero" at next, and same for B...

How could this not be detected by my tests !?!

This caused [\xintiiGCD](#)of hence the [gcd\(\)](#) function in [\xintiiexpr](#) to break as soon as one argument was zero.

```
1396 \def\xintiiGCD {\romannumeral0\xintiigcd }%
1397 \def\xintiigcd #1{\expandafter\XINT_iigcd\romannumeral0\xintiiabs#1\xint:}%
1398 \def\XINT_iigcd #1#2\xint:#3%
1399 {%
1400 \expandafter\XINT_gcd_fork\expandafter#1%
1401 \romannumeral0\xintiiabs#3\xint:#1#2\xint:
1402 }%
1403 \def\XINT_gcd_fork #1#2%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```
1404 {%
1405     \xint_UDzerofork
1406     #1\xINT_gcd_Aiszero
1407     #2\xINT_gcd_Biszero
1408     0\xINT_gcd_loop
1409     \krof
1410     #2%
1411 }%
1412 \def\xINT_gcd_Aiszero #1\xint:#2\xint:{ #1}%
1413 \def\xINT_gcd_Biszero #1\xint:#2\xint:{ #2}%
1414 \def\xINT_gcd_loop #1\xint:#2\xint:
1415 {%
1416     \expandafter\expandafter\expandafter\xINT_gcd_CheckRem
1417     \expandafter\xint_seconddoftwo
1418     \romannumeral0\xINT_div_prepare {#1}{#2}\xint:#1\xint:
1419 }%
1420 \def\xINT_gcd_CheckRem #1%
1421 {%
1422     \xint_gob_til_zero #1\xINT_gcd_end0\xINT_gcd_loop #1%
1423 }%
1424 \def\xINT_gcd_end0\xINT_gcd_loop #1\xint:#2\xint:{ #2}%
```

### 21.55. \xintiigCDof

New with 1.09a (was located in xintgcd.sty).

1.21 adds protection against items being non-terminated `\the\numexpr`.

1.4 renames the macro into `\xintiigCDof` and moves it here. Terminator modified to `^` for direct call by `\xintiexpr` function.

1.4d fixes breakage inherited since 1.3d from `\xintiigCD`, in case any argument vanished.

Currently does not support empty list of arguments.

```
1425 \def\xintiigCDof    {\romannumeral0\xintiigcdof }%
1426 \def\xintiigcdof    #1{\expandafter\xINT_iigcdof_a\romannumeral`&&@#1^}%
1427 \def\xINT_iigCDof    {\romannumeral0\xINT_iigcdof_a}%
1428 \def\xINT_iigcdof_a  #1{\expandafter\xINT_iigcdof_b\romannumeral`&&@#1!}%
1429 \def\xINT_iigcdof_b  #1!#2{\expandafter\xINT_iigcdof_c\romannumeral`&&@#2!{#1!}%
1430 \def\xINT_iigcdof_c  #1{\xint_gob_til_ ^ #1\xINT_iigcdof_e ^\xINT_iigcdof_d #1}%
1431 \def\xINT_iigcdof_d  #1!{\expandafter\xINT_iigcdof_b\romannumeral0\xintiigcd {#1}}%
1432 \def\xINT_iigcdof_e  #1!#2!{ #2}%
```

### 21.56. \xintiilCM

Copied over `\xintiilCM` code from `xintgcd` at 1.3d in order to support `lcm()` function in `\xintiexpr`.

At 1.4 original code removed from `xintgcd` as the latter now requires `xint`.

```
1433 \def\xintiilCM {\romannumeral0\xintiilcm}%
1434 \def\xintiilcm #1{\expandafter\xINT_iilcm\romannumeral0\xintiilabs#1\xint:}%
1435 \def\xINT_iilcm #1#2\xint:#3%
1436 {%
1437     \expandafter\xINT_lcm_fork\expandafter#1%
1438     \romannumeral0\xintiilabs#3\xint:#1#2\xint:
1439 }%
1440 \def\xINT_lcm_fork #1#2%
1441 {%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1442 \xint_UDzerofork
1443 #1\XINT_lcm_iszero
1444 #2\XINT_lcm_iszero
1445 0\XINT_lcm_notzero
1446 \krof
1447 #2%
1448 }%
1449 \def\XINT_lcm_iszero #1\xint:#2\xint:{ 0}%
1450 \def\XINT_lcm_notzero #1\xint:#2\xint:
1451 {%
1452 \expandafter\XINT_lcm_end\romannumeral0%
1453 \expandafter\expandafter\expandafter\XINT_gcd_CheckRem
1454 \expandafter\xint_secondoftwo
1455 \romannumeral0\XINT_div_prepare {#1}{#2}\xint:#1\xint:
1456 \xint:#1\xint:#2\xint:
1457 }%
1458 \def\XINT_lcm_end #1\xint:#2\xint:#3\xint:{\xintiimul {#2}{\xintiiQuo{#3}{#1}}}%

```

### 21.57. \xintiilCMof

See comments of `\xintiigCDof`.

```
1459 \def\xintiilCMof {\romannumeral0\xintiilcmof }%
1460 \def\xintiilcmof #1{\expandafter\XINT_iilcmof_a\romannumeral`&&@#1^}%
1461 \def\XINT_iilCMof {\romannumeral0\XINT_iilcmof_a}%
1462 \def\XINT_iilcmof_a #1{\expandafter\XINT_iilcmof_b\romannumeral`&&@#1!}%
1463 \def\XINT_iilcmof_b #1!#2{\expandafter\XINT_iilcmof_c\romannumeral`&&@#2!{#1}!}%
1464 \def\XINT_iilcmof_c #1{\xint_gob_til^ #1\XINT_iilcmof_e ^\XINT_iilcmof_d #1}%
1465 \def\XINT_iilcmof_d #1!{\expandafter\XINT_iilcmof_b\romannumeral0\xintiilcm {#1}}%
1466 \def\XINT_iilcmof_e #1!#2!{ #2}%

```

### 21.58. (WIP) \xintRandomDigits

1.3b. See user manual. Whether this will be part of *xintkernel*, *xintcore*, or *xint* is yet to be decided.

```
1467 \def\xintRandomDigits{\romannumeral0\xintrandomdigits}%
1468 \def\xintrandomdigits#1%
1469 {%
1470 \csname xint_gob_andstop_\expandafter\XINT_randomdigits\the\numexpr#1\xint:
1471 }%
1472 \def\XINT_randomdigits#1\xint:
1473 {%
1474 \expandafter\XINT_randomdigits_a
1475 \the\numexpr(#1+\xint_c_iii)/\xint_c_viii\xint:#1\xint:
1476 }%
1477 \def\XINT_randomdigits_a#1\xint:#2\xint:
1478 {%
1479 \romannumeral\numexpr\xint_c_viii*#1-#2\csname XINT_%
1480 \romannumeral\XINT_replicate #1\endcsname \csname
1481 XINT_rdg\endcsname
1482 }%
1483 \def\XINT_rdg
1484 {%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, xint, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1485 \expandafter\XINT_rdg_aux\the\numexpr%
1486 \xint_c_nine_x^viii%
1487 -\xint_texuniformdeviate\xint_c_ii^vii%
1488 -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
1489 -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
1490 -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
1491 +\xint_texuniformdeviate\xint_c_x^viii%
1492 \relax%
1493 }%
1494 \def\XINT_rdg_aux#1\XINT_rdg\endcsname}%
1495 \let\XINT_XINT_rdg\endcsname
```

### 21.59. (WIP) \XINT\_eightrandomdigits, \xintEightRandomDigits

1.3b. 1.4 adds some public alias...

```
1496 \def\XINT_eightrandomdigits
1497 {%
1498 \expandafter\xint_gobble_i\the\numexpr%
1499 \xint_c_nine_x^viii%
1500 -\xint_texuniformdeviate\xint_c_ii^vii%
1501 -\xint_c_ii^vii*\xint_texuniformdeviate\xint_c_ii^vii%
1502 -\xint_c_ii^xiv*\xint_texuniformdeviate\xint_c_ii^vii%
1503 -\xint_c_ii^xxi*\xint_texuniformdeviate\xint_c_ii^vii%
1504 +\xint_texuniformdeviate\xint_c_x^viii%
1505 \relax%
1506 }%
1507 \let\xintEightRandomDigits\XINT_eightrandomdigits
1508 \def\xintRandBit{\xint_texuniformdeviate\xint_c_ii}%

```

### 21.60. (WIP) \xintRandBit

1.4 And let's add also \xintRandBit while we are at it.

```
1509 \def\xintRandBit{\xint_texuniformdeviate\xint_c_ii}%

```

### 21.61. (WIP) \xintXRandomDigits

1.3b.

```
1510 \def\xintXRandomDigits#1%
1511 {%
1512 \csname xint_gobble\expandafter\XINT_xrandomdigits\the\numexpr#1\xint:
1513 }%
1514 \def\XINT_xrandomdigits#1\xint:
1515 {%
1516 \expandafter\XINT_xrandomdigits_a
1517 \the\numexpr(#1+\xint_c_iii)/\xint_c_viii\xint:#1\xint:
1518 }%
1519 \def\XINT_xrandomdigits_a#1\xint:#2\xint:
1520 {%
1521 \romannumeral\numexpr\xint_c_viii*#1-#2\expandafter\endcsname
1522 \romannumeral`&&@\romannumeral
1523 \XINT_replicate #1\endcsname\XINT_eightrandomdigits
1524 }%

```

## 21.62. (WIP) \xintiiRandRangeAtoB

1.3b. Support for randrange() function.

We do it f-expandably for matters of \xintNewExpr etc... The \xintexpr will add \xintNum wrapper to possible fractional input. But \xintiiexpr will call as is.

TODO: ? implement third argument (STEP) TODO: \xintNum wrapper (which truncates) not so good in floatexpr. Use round?

It is an error if b<=a, as in Python.

```

1525 \def\xintiiRandRangeAtoB{\romannumeral`&&\xintiiRandRangeAtoB}%
1526 \def\xintiiRandRangeAtoB#1%
1527 {%
1528   \expandafter\XINT_randrangeAtoB_a\romannumeral`&&#1\xint:
1529 }%
1530 \def\XINT_randrangeAtoB_a#1\xint:#2%
1531 {%
1532   \xintiiadd{\expandafter\XINT_randrange
1533             \romannumeral0\xintiisub{#2}{#1}\xint:}%
1534   {#1}%
1535 }%
```

## 21.63. (WIP) \xintiiRandRange

1.3b. Support for randrange().

```

1536 \def\xintiiRandRange{\romannumeral`&&\xintiiRandRange}%
1537 \def\xintiiRandRange#1%
1538 {%
1539   \expandafter\XINT_randrange\romannumeral`&&#1\xint:
1540 }%
1541 \def\XINT_randrange #1%
1542 {%
1543   \xint_UDzerominusfork
1544   #1-\XINT_randrange_err:empty
1545   0#1\XINT_randrange_err:empty
1546   0-\XINT_randrange_a
1547   \krof #1%
1548 }%
1549 \def\XINT_randrange_err:empty#1\xint:
1550 {%
1551   \XINT_expandableerror{Empty range for randrange.} 0%
1552 }%
1553 \def\XINT_randrange_a #1\xint:
1554 {%
1555   \expandafter\XINT_randrange_b\romannumeral0\xintlength{#1}.#1\xint:
1556 }%
1557 \def\XINT_randrange_b #1.%
1558 {%
1559   \ifnum#1<\xint_c_x\xint_dothis{\the\numexpr\XINT_uniformdeviate{}}\fi
1560   \xint_orthat{\XINT_randrange_c #1.}%
1561 }%
1562 \def\XINT_randrange_c #1.#2#3#4#5#6#7#8#9%
1563 {%
1564   \expandafter\XINT_randrange_d
```

```

1565 \the\numexpr\expandafter\XINT_uniformdeviate\expandafter
1566 {\expandafter}\the\numexpr\xint_c_i+#2#3#4#5#6#7#8#9\xint:\xint:
1567 #2#3#4#5#6#7#8#9\xint:#1\xint:
1568 }%

```

This raises following annex question: immediately after setting the seed is it possible for `\xintUniformDeviate{N}` where  $N > 0$  has exactly eight digits to return either 0 or  $N-1$ ? It could be that this is never the case, then there is a bias in `randrange()`. Of course there are anyhow only  $2^{28}$  seeds so `randrange(10^X)` is by necessity biased when executed immediately after setting the seed, if  $X$  is at least 9.

```

1569 \def\XINT_randrange_d #1\xint:#2\xint:
1570 {%
1571 \ifnum#1=\xint_c_\xint_dothis\XINT_randrange_Z\fi
1572 \ifnum#1=#2 \xint_dothis\XINT_randrange_A\fi
1573 \xint_orthat\XINT_randrange_e #1\xint:
1574 }%
1575 \def\XINT_randrange_e #1\xint:#2\xint:#3\xint:
1576 {%
1577 \the\numexpr#1\expandafter\relax
1578 \romannumeral0\xintrandomdigits{#2-\xint_c_viii}%
1579 }%

```

This is quite unlikely to get executed but if it does it must pay attention to leading zeros, hence the `\xintinum`. We don't have to be overly obstinate about removing overheads...

```

1580 \def\XINT_randrange_Z 0\xint:#1\xint:#2\xint:
1581 {%
1582 \xintinum{\xintRandomDigits{#1-\xint_c_viii}}%
1583 }%

```

Here too, overhead is not such a problem. The idea is that we got by extraordinary same first 8 digits as upper range bound so we pick at random the remaining needed digits in one go and compare with the upper bound. If too big, we start again with another random 8 leading digits in given range. No need to aim at any kind of efficiency for the check and loop back.

```

1584 \def\XINT_randrange_A #1\xint:#2\xint:#3\xint:
1585 {%
1586 \expandafter\XINT_randrange_B
1587 \romannumeral0\xintrandomdigits{#2-\xint_c_viii}\xint:
1588 #3\xint:#2.#1\xint:
1589 }%
1590 \def\XINT_randrange_B #1\xint:#2\xint:#3.#4\xint:
1591 {%
1592 \xintiiflt{#1}{#2}{\XINT_randrange_E}{\XINT_randrange_again}%
1593 #4#1\xint:#3.#4#2\xint:
1594 }%
1595 \def\XINT_randrange_E #1\xint:#2\xint:{ #1}%
1596 \def\XINT_randrange_again #1\xint:{\XINT_randrange_c}%

```

## 21.64. (WIP) Adjustments for engines without `uniformdeviate` primitive

Added at 1.3b (2018/05/18).

```

1597 \ifdefined\xint_texuniformdeviate
1598 \else
1599 \def\xintrandomdigits#1%

```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfraction](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1600  {%
1601      \XINT_expandableerror
1602      {No uniformdeviate at engine level.} 0%
1603  }%
1604  \let\xintXRandomDigits\xintRandomDigits
1605  \def\xint_randrange#1\xint:
1606  {%
1607      \XINT_expandableerror
1608      {No uniformdeviate at engine level.} 0%
1609  }%
1610  \fi
1611  \XINTrestorecatcodesendinginput%
```



## 22. Package *xintbinhex* implementation

.1	Catcodes, $\varepsilon$ -T <sub>E</sub> X and reload detection . .	400	.9	<code>\xintOctToDec</code> . . . . .	412
.2	Package identification . . . . .	401	.10	<code>\xintBinToDec</code> . . . . .	414
.3	Storage macros . . . . .	401	.11	<code>\xintHexToOct</code> . . . . .	414
.4	Helper macros . . . . .	403	.12	<code>\xintHexToBin</code> . . . . .	415
.4.1	<code>\XINT_zeroes_foriv</code> . . . . .	403	.13	<code>\xintCHexToBin</code> . . . . .	416
.4.2	<code>\XINT_zeroes_foriii</code> . . . . .	403	.14	<code>\xintOctToHex</code> . . . . .	416
.5	<code>\xintDecToHex</code> . . . . .	403	.15	<code>\xintBinToHex</code> . . . . .	416
.6	<code>\xintDecToOct</code> . . . . .	406	.16	<code>\xintOctToBin</code> . . . . .	417
.7	<code>\xintDecToBin</code> . . . . .	409	.17	<code>\xintCOctToBin</code> . . . . .	418
.8	<code>\xintHexToDec</code> . . . . .	410	.18	<code>\xintBinToOct</code> . . . . .	418

The commenting is currently (2025/09/06) still very sparse.

1.2m (2017/07/31) rewrote entirely the original macros coming with 1.08 (2013/06/07).

At 1.2n the dependencies on *xintcore* were removed, so now the package loads only *xintkernel*.

Also at 1.2n (2017/08/06), `\csname` governed expansion was used at some places rather than `\numex`, `pr`. This increased the maximal input sizes for `\xintDecToHex`, `\xintDecToBin`, and `\xintBinToHex`.

1.4n (2025/09/05) adds:

- Conversion to and from octal base.
- Usage of `\expanded` in place of the `\csname`-based expansion from 1.2n. This has increased the maximal input sizes. The increase is spectacular for the conversions between binary, octal and hexadecimal, the limit on size being solely dependent on T<sub>E</sub>X main memory size. This change however caused seemingly (but barely tested) a slight (1%) decrease in speed for `\xintDecToHex` and `\xintHexToDec`.

Note that the whole architecture was last re-thought in 2017 at a time `\expanded` did not exist. The most subtle coding, which is the one regarding conversion to and from decimal radix is of such a nature that I am not motivated at this stage to modify it other than minimally (see initial code comments for `\xintDecToHex`). So the `\expanded` was used therein but only for some subroutines. For the conversions between binary, octal, and hexadecimal, the whole expansion basically is governed by a single `\expanded`. But for legacy reasons and stylistic coherence across the codebase, the code keeps using `\romannumeral0` trigger at the start of the macros with a systematic naming scheme. I am not much motivated to engage into systematic changes across about 20000 code lines in the *xint* bundle!

*xintexpr* loads *xintbinhex* automatically as of 1.4n. Formerly it supported only " prefix for hexadecimal, now it supports also ' for octal, as well as `0x`, `0o` and `0b`.

### 22.1. Catcodes, $\varepsilon$ -T<sub>E</sub>X and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^
11 \def\empty{}\def\space{ }\newlinechar10

```

```

12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintbinhex.sty\endcsname
14 \expandafter\let\expandafter\w\csname ver@xintkernel.sty\endcsname
15 \expandafter\ifx\csname numexpr\endcsname\relax
16   \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17     \immediate\write128{^^JPackage xintbinhex Warning:^^J}%
18     \space\space\space\space
19     \numexpr not available, aborting input.^^J}%
20   \else
21     \PackageWarningNoLine{xintbinhex}{\numexpr not available, aborting input}%
22   \fi
23 \def\z{\endgroup\endinput}%
24 \else
25   \ifx\x\relax % plain-TeX, first loading of xintbinhex.sty
26     \ifx\w\relax % but xintkernel.sty not yet loaded.
27       \def\z{\endgroup\input xintkernel.sty\relax}%
28     \fi
29   \else
30     \ifx\x\empty % LaTeX, first loading,
31     % variable is initialized, but \ProvidesPackage not yet seen
32     \ifx\w\relax % xintkernel.sty not yet loaded.
33       \def\z{\endgroup\RequirePackage{xintkernel}}%
34     \fi
35   \else
36     \def\z{\endgroup\endinput}% xintbinhex already loaded.
37   \fi
38 \fi
39 \fi
40 \z%
41 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 22.2. Package identification

```

42 \XINT_providespackage
43 \ProvidesPackage{xintbinhex}%
44 [2025/09/06 v1.4o Expandable radix conversions (2, 8, 10, and 16) (JFB)]%

```

## 22.3. Storage macros

**Modified at 1.2n (2017/08/06).** Switch to `\csname`-governed expansion at various places.

**Modified at 1.4n (2025/09/05).** Move two `\newcount`'s to *xintkernel*.

**Modified at 1.4n (2025/09/05).** This release uses `\expanded` in place of `\csname` governed expansion. This means that the mysterious `\endcsname`'s have vanished from the storage macros. And support for octal was added, hence a few more storage macros were added.

```

45 \def\XINT_tmpa #1{\ifx\relax#1\else
46   \expandafter\edef\csname XINT_csdth_#1\endcsname
47   {\ifcase #1 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or
48     8\or 9\or A\or B\or C\or D\or E\or F\fi}%
49   \expandafter\XINT_tmpa\fi }%
50 \XINT_tmpa {0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}\relax
51 \def\XINT_tmpa #1{\ifx\relax#1\else
52   \expandafter\edef\csname XINT_csdth_#1\endcsname

```

```

53  {\ifcase #1
54    0000\or 0001\or 0010\or 0011\or 0100\or 0101\or 0110\or 0111\or
55    1000\or 1001\or 1010\or 1011\or 1100\or 1101\or 1110\or 1111\fi}%
56  \expandafter\XINT_tmpa\fi }%
57  \XINT_tmpa {0}{1}{2}{3}{4}{5}{6}{7}{8}{9}{10}{11}{12}{13}{14}{15}\relax
58  \let\XINT_tmpa\relax
59  \expandafter\def\csname XINT_csbth_0000\endcsname {0}%
60  \expandafter\def\csname XINT_csbth_0001\endcsname {1}%
61  \expandafter\def\csname XINT_csbth_0010\endcsname {2}%
62  \expandafter\def\csname XINT_csbth_0011\endcsname {3}%
63  \expandafter\def\csname XINT_csbth_0100\endcsname {4}%
64  \expandafter\def\csname XINT_csbth_0101\endcsname {5}%
65  \expandafter\def\csname XINT_csbth_0110\endcsname {6}%
66  \expandafter\def\csname XINT_csbth_0111\endcsname {7}%
67  \expandafter\def\csname XINT_csbth_1000\endcsname {8}%
68  \expandafter\def\csname XINT_csbth_1001\endcsname {9}%
69  \expandafter\def\csname XINT_csbth_1010\endcsname {A}%
70  \expandafter\def\csname XINT_csbth_1011\endcsname {B}%
71  \expandafter\def\csname XINT_csbth_1100\endcsname {C}%
72  \expandafter\def\csname XINT_csbth_1101\endcsname {D}%
73  \expandafter\def\csname XINT_csbth_1110\endcsname {E}%
74  \expandafter\def\csname XINT_csbth_1111\endcsname {F}%
75  \expandafter\def\csname XINT_csbto_000\endcsname {0}%
76  \expandafter\def\csname XINT_csbto_001\endcsname {1}%
77  \expandafter\def\csname XINT_csbto_010\endcsname {2}%
78  \expandafter\def\csname XINT_csbto_011\endcsname {3}%
79  \expandafter\def\csname XINT_csbto_100\endcsname {4}%
80  \expandafter\def\csname XINT_csbto_101\endcsname {5}%
81  \expandafter\def\csname XINT_csbto_110\endcsname {6}%
82  \expandafter\def\csname XINT_csbto_111\endcsname {7}%
83  \expandafter\def\csname XINT_cshtb_0\endcsname {0000}%
84  \expandafter\def\csname XINT_cshtb_1\endcsname {0001}%
85  \expandafter\def\csname XINT_cshtb_2\endcsname {0010}%
86  \expandafter\def\csname XINT_cshtb_3\endcsname {0011}%
87  \expandafter\def\csname XINT_cshtb_4\endcsname {0100}%
88  \expandafter\def\csname XINT_cshtb_5\endcsname {0101}%
89  \expandafter\def\csname XINT_cshtb_6\endcsname {0110}%
90  \expandafter\def\csname XINT_cshtb_7\endcsname {0111}%
91  \expandafter\def\csname XINT_cshtb_8\endcsname {1000}%
92  \expandafter\def\csname XINT_cshtb_9\endcsname {1001}%
93  \def\XINT_cshtb_A {1010}%
94  \def\XINT_cshtb_B {1011}%
95  \def\XINT_cshtb_C {1100}%
96  \def\XINT_cshtb_D {1101}%
97  \def\XINT_cshtb_E {1110}%
98  \def\XINT_cshtb_F {1111}%
99  \expandafter\def\csname XINT_csotb_0\endcsname {000}%
100 \expandafter\def\csname XINT_csotb_1\endcsname {001}%
101 \expandafter\def\csname XINT_csotb_2\endcsname {010}%
102 \expandafter\def\csname XINT_csotb_3\endcsname {011}%
103 \expandafter\def\csname XINT_csotb_4\endcsname {100}%
104 \expandafter\def\csname XINT_csotb_5\endcsname {101}%

```

```

105 \expandafter\def\csname XINT_csotb_6\endcsname {110}%
106 \expandafter\def\csname XINT_csotb_7\endcsname {111}%

```

## 22.4. Helper macros

We need at 1.4n to ensure lengths either multiple of 5, 4 or 3. For multiples of three we simply imitate what we had in 2017 for multiples of four. For multiples of five we will do it rather ``in place'' appealing to [\xintLength](#) and [\xintReplicate](#) for convenience. I suspected grabbing five by five (as we can't do ten by ten) would prove less efficient, but I am not motivated enough at this stage to devote the time needed for implementing and comparing.

### 22.4.1. [\XINT\\_zeroes\\_foriv](#)

```

\romannumeral0\XINT_zeroes_foriv #1\R{0\R}{00\R}{000\R}%
\R{0\R}{00\R}{000\R}\R\W

```

expands to <empty> or 0 or 00 or 000 as needed to prepend to #1 to extend it to length 4N.

```

107 \def\XINT_zeroes_foriv #1#2#3#4#5#6#7#8%
108 {%
109   \xint_gob_til_R #8\XINT_zeroes_foriv_end\R\XINT_zeroes_foriv
110 }%
111 \def\XINT_zeroes_foriv_end\R\XINT_zeroes_foriv #1#2\W
112   {\XINT_zeroes_foriv_done #1}%
113 \def\XINT_zeroes_foriv_done #1\R{ #1}%

```

### 22.4.2. [\XINT\\_zeroes\\_foriii](#)

Added at 1.4n (2025/09/05).

```

\romannumeral0\XINT_zeroes_foriii #1\R{0\R}{00\R}%
\R{0\R}{00\R}%
\R{0\R}{00\R}\R\W

```

expands to <empty> or 0 or 00 as needed to prepend to #1 to extend it to length 3N.

```

114 \def\XINT_zeroes_foriii #1#2#3#4#5#6#7#8#9%
115 {%
116   \xint_gob_til_R #9\XINT_zeroes_foriii_end\R\XINT_zeroes_foriii
117 }%
118 \def\XINT_zeroes_foriii_end\R\XINT_zeroes_foriii #1#2\W
119   {\XINT_zeroes_foriii_done #1}%
120 \def\XINT_zeroes_foriii_done #1\R{ #1}%

```

## 22.5. [\xintDecToHex](#)

Now that illicit tools such as [unravel](#) exist, we can not hide anymore too much the crux of the matter. Let's simply say that the decimal input is tacitly converted into a sequence of successive elementary steps ``multiply by 10000 and add a base 10000 digit''. This is made to act on a sequence of radix  $16^4$  digits (note that  $10000 < 16^4$ ). Those radix  $16^4$  digits are of course manipulated via [\numexpr](#) as  $10^5$ -based digits (note that  $16^4 < 10^5$  and that not only  $16^4 \cdot 1000 < 10^9 < 2^{31}$  but also  $10^9 + 16^4 \cdot 1000 < 2^{31}$  which helps in ensuring certain operations expand to a fixed number of digit tokens). When the sequence of elementary steps is complete, the base- $16^4$  digits only need to be converted to hexadecimal notation and purged of separators. Of course doing the whole thing expandably requires some skill. This was done when I was not yet an old man, and obviously was still very clever. Enjoy.

Modified at 1.2m (2017/07/31). Rewritten from scratch using the [xintcore](#) 1.2 style. Now guards against non terminated inputs.

**Modified at 1.2n (2017/08/06).** Coding improvements, `\csname`-governed expansion, increased maximal size.

**Modified at 1.2o (2017/08/29).** Again coding improvements (efficiency gain about 6%).

**Modified at 1.4n (2025/09/05).** Replacement of `\csname`-based expansion by usage of `\expanded`. This has increased the maximal input size from 12035 to 16042 digits (depends on nesting level; evaluated with TL2025 default TeX memory parameters, see user manual). I noticed a slight performance decrease of the order of 1% but did not test extensively.

Perhaps `\expanded` could be used to a deeper refactoring but... I preferred to make minimal changes, not having kept in mind all subtle details of the 2017 code.

```

121 \def\xintDecToHex {\romannumeral0\xintdectohex }%
122 \def\xintdectohex #1%
123 {%
124   \expandafter\XINT_dth_checkin\romannumeral`&&@#1\xint:
125 }%
126 \def\XINT_dth_checkin #1%
127 {%
128   \xint_UDsignfork
129     #1\XINT_dth_neg
130     -{\XINT_dth_main #1}%
131   \krof
132 }%
133 \def\XINT_dth_neg {\expandafter-\romannumeral0\XINT_dth_main}%
134 \def\XINT_dth_main #1\xint:
135 {%
136   \expandafter\XINT_dth_finish
137   \romannumeral`&&@\expandafter\XINT_dthb_start
138   \romannumeral0\XINT_zeroes_foriv
139   #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
140   #1\xint_bye\XINT_dth_tohex
141 }%
142 \def\XINT_dthb_start #1#2#3#4#5%
143 {%
144   \xint_bye#5\XINT_dthb_small\xint_bye\XINT_dthb_start_a #1#2#3#4#5%
145 }%
146 \def\XINT_dthb_small\xint_bye\XINT_dthb_start_a #1\xint_bye#2{#2#1!}%
147 \def\XINT_dthb_start_a #1#2#3#4#5#6#7#8#9%
148 {%
149   \expandafter\XINT_dthb_again
150   \the\numexpr\expandafter\XINT_dthb_update
151   \the\numexpr#1#2#3#4%
152   \xint_bye#9\XINT_dthb_lastpass\xint_bye
153   #5#6#7#8!\XINT_dthb_exclam\relax\XINT_dthb_nextfour #9%
154 }%

```

The 1.2n inserted exclamations marks, which when bumping back from `\XINT_dthb_again` gave rise to a `\numexpr`-loop which gathered the ! delimited arguments and inserted `\expandafter\XINT_dthb_update\the\numexpr` dynamically. The 1.2o trick is to insert it here immediately. Then at `\XINT_dthb_again` the `\numexpr` will trigger an already prepared chain.

The crux of the thing is handling of #3 at `\XINT_dthb_update_a`.

```

155 \def\XINT_dthb_exclam {!\XINT_dthb_exclam\relax
156   \expandafter\XINT_dthb_update\the\numexpr}%
157 \def\XINT_dthb_update #1!%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
158 {%
159   \expandafter\XINT_dthb_update_a
160   \the\numexpr (#1+\xint_c_ii^xv)/\xint_c_ii^xvi-\xint_c_i\xint:
161   #1\xint:%
162 }%
```

The four zeros here will get appended to prior digits. They enact the multiplication by 10000.

```
163 \def\XINT_dthb_update_a #1\xint:#2\xint:#3%
164 {%
165   0000+#1\expandafter#3\the\numexpr#2-#1*\xint_c_ii^xvi
166 }%
167 \def\XINT_dthb_nextfour #1#2#3#4#5%
168 {%
169   \xint_bye#5\XINT_dthb_lastpass\xint_bye
170   #1#2#3#4!\XINT_dthb_exclam\relax\XINT_dthb_nextfour#5%
171 }%
172 \def\XINT_dthb_lastpass\xint_bye #1!#2\xint_bye#3{#1!#3!}%
173 \def\XINT_dthb_again #1!#2#3%
174 {%
175   \ifx#3\relax
176     \expandafter\xint_firstoftwo
177   \else
178     \expandafter\xint_secondoftwo
179   \fi
180   {\expandafter\XINT_dthb_again
181    \the\numexpr
182    \ifnum #1>\xint_c_
183      \xint_afterfi{\expandafter\XINT_dthb_update\the\numexpr#1}%
184    \fi}%
185   {\ifnum #1>\xint_c_ \xint_dothis{#2#1!}\fi\xint_orthat{!#2!}}%
186 }%
```

1.4n replaces here `\csname`'s method by simpler `\expanded` control. This is the part of the algorithm which rewrites base 16<sup>4</sup> (kept in decimal) digits into four hexadecimal digits.

```
187 \def\XINT_dth_tohex
188 {%
189   \expandafter\XINT_dth_tohex_a\expanded\XINT_tofourhex
190 }%
191 \def\XINT_dth_tohex_a{!\XINT_dth_tohex!}%
192 \def\XINT_tofourhex #1!%
193 {%
194   {\iffalse}\fi\expandafter\XINT_tofourhex_a
195   \the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\xint:
196   #1\xint:
197 }%
198 \def\XINT_tofourhex_a #1\xint:#2\xint:
199 {%
200   \expandafter\XINT_tofourhex_c
201   \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
202   #1\xint:
203   #2-\xint_c_ii^viii*#1!%
204 }%
205 \def\XINT_tofourhex_c #1\xint:#2\xint:
206 {%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
207 \csname XINT_csdth_#1\endcsname
208 \csname XINT_csdth_\the\numexpr #2-\xint_c_xvi*#1\endcsname
209 \expandafter\XINT_tofourhex_d\the\numexpr
210 }%
211 \def\XINT_tofourhex_d #1!%
212 {%
213 \expandafter\XINT_tofourhex_e
214 \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
215 #1\xint:
216 }%
217 \def\XINT_tofourhex_e #1\xint:#2\xint:
218 {%
219 \csname XINT_csdth_#1\endcsname
220 \csname XINT_csdth_\the\numexpr #2-\xint_c_xvi*#1\endcsname
221 \iffalse{\fi}%
222 }%
```

We only clean up up to 3 hexadecimal zeros, as output is produced in chunks of 4 hex digits, and (this comment added at 1.4n) as far as I understand the leading chunk can be 0000 only if input was vanishing.

The coding for this simple trimming may not be the most efficient, but this code is executed only once. I remember from 2017 that I had gotten tired to always try to optimize and did not even try to test efficiency.

```
223 \def\XINT_dth_finish !\XINT_dth_tohex!#1#2#3%
224 {%
225 \unless\if#10\xint_dothis{ #1#2#3}\fi
226 \unless\if#20\xint_dothis{ #2#3}\fi
227 \unless\if#30\xint_dothis{ #3}\fi
228 \xint_orthat{ }%
229 }%
```

## 22.6. \xintDecToOct

Added at 1.4n (2025/09/05).

This late extension to the package is imitated from (the *\expanded* updated) *\xintDecToHex*.

It does not share macros with *\xintDecToHex* to same extent as *\xintDecToBin* does (those macros with *\_dthb\_* in their names), because the input gets converted to radix  $8^5$ , not radix  $16^4$ .

Note that we have  $10000 < 8^5 < 100000$  and refer to *\xintDecToHex* for some information on the algorithm.

```
230 \def\xintDecToOct {\romannumeral0\xintdectooc }%
231 \def\xintdectooc #1%
232 {%
233 \expandafter\XINT_dto_checkin\romannumeral`&&@#1\xint:
234 }%
235 \def\XINT_dto_checkin #1%
236 {%
237 \xint_UDsignfork
238 #1\XINT_dto_neg
239 -{\XINT_dto_main #1}%
240 \krof
241 }%
242 \def\XINT_dto_neg {\expandafter-\romannumeral0\XINT_dto_main}%
243 \def\XINT_dto_main #1\xint:
```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

244 {%
245   \expandafter\XINT_dto_finish
246   \romannumeral`&&@\expandafter\XINT_dto_start
247   \romannumeral0\XINT_zeroes_foriv
248   #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
249   #1\xint_bye\XINT_dto_tooct
250 }%
251 \def\XINT_dto_start #1#2#3#4#5%
252 {%
253   \xint_bye#5\XINT_dto_small\xint_bye\XINT_dto_start_a #1#2#3#4#5%
254 }%
255 \def\XINT_dto_small\xint_bye\XINT_dto_start_a #1\xint_bye#2{#2#1!}%
256 \def\XINT_dto_start_a #1#2#3#4#5#6#7#8#9%
257 {%
258   \expandafter\XINT_dto_again\the\numexpr\expandafter\XINT_dto_update
259   \the\numexpr#1#2#3#4%
260   \xint_bye#9\XINT_dto_lastpass\xint_bye
261   #5#6#7#8!\XINT_dto_exclam\relax\XINT_dto_nextfour #9%
262 }%
263 \def\XINT_dto_exclam {!\XINT_dto_exclam\relax
264   \expandafter\XINT_dto_update\the\numexpr}%
265 \def\XINT_dto_update #1!%
266 {%
267   \expandafter\XINT_dto_update_a
268   \the\numexpr (#1+\xint_c_ii^xiv)/\xint_c_ii^xv-\xint_c_i\xint:
269   #1\xint:%
270 }%
271 \def\XINT_dto_update_a #1\xint:#2\xint:#3%
272 {%
273   0000+#1\expandafter#3\the\numexpr#2-#1*\xint_c_ii^xv
274 }%
275 \def\XINT_dto_nextfour #1#2#3#4#5%
276 {%
277   \xint_bye#5\XINT_dto_lastpass\xint_bye
278   #1#2#3#4!\XINT_dto_exclam\relax\XINT_dto_nextfour#5%
279 }%
280 \def\XINT_dto_lastpass\xint_bye #1!#2\xint_bye#3{#1!#3!}%
281 \def\XINT_dto_again #1!#2#3%
282 {%
283   \ifx#3\relax
284     \expandafter\xint_firstoftwo
285   \else
286     \expandafter\xint_secondoftwo
287   \fi
288   {\expandafter\XINT_dto_again
289   \the\numexpr
290   \ifnum #1>\xint_c_
291     \xint_afterfi{\expandafter\XINT_dto_update\the\numexpr#1}%
292   \fi}%
293   {\ifnum #1>\xint_c_ \xint_dothis{#2#1!}\fi\xint_orthat{!#2!}}%
294 }%
295 \def\XINT_dto_tooct

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

296 {%
297   \expandafter\XINT_dto_tooct_a\expanded\XINT_tofiveoct
298 }%
299 \def\XINT_dto_tooct_a{!\XINT_dto_tooct!}%
300 \def\XINT_tofiveoct #1!%
301 {%
302   {\iffalse}\fi\expandafter\XINT_tofiveoct_a
303   \the\numexpr (#1+\xint_c_ii^viii)/\xint_c_ii^ix-\xint_c_i\xint:
304   #1\xint:
305 }%
306 \def\XINT_tofiveoct_a #1\xint:#2\xint:
307 {%
308   \expandafter\XINT_tofiveoct_c
309   \the\numexpr (#1+\xint_c_iv)/\xint_c_viii-\xint_c_i\xint:#1\xint:
310   #2-\xint_c_ii^ix*#1!%
311 }%
312 \def\XINT_tofiveoct_c #1\xint:#2\xint:
313 {%
314   #1\the\numexpr #2-\xint_c_viii*#1\relax
315   \expandafter\XINT_tofiveoct_d\the\numexpr
316 }%
317 \def\XINT_tofiveoct_d #1!%
318 {%
319   \expandafter\XINT_tofiveoct_e
320   \the\numexpr (#1+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i\xint:
321   #1\xint:
322 }%
323 \def\XINT_tofiveoct_e #1\xint:#2\xint:
324 {%
325   #1\expandafter\XINT_tofiveoct_f\the\numexpr #2-\xint_c_ii^vi*#1!%
326 }%
327 \def\XINT_tofiveoct_f #1!%
328 {%
329   \expandafter\XINT_tofiveoct_g
330   \the\numexpr (#1+\xint_c_iv)/\xint_c_viii-\xint_c_i\xint:
331   #1\xint:
332 }%
333 \def\XINT_tofiveoct_g #1\xint:#2\xint:
334 {%
335   #1\the\numexpr #2-\xint_c_viii*#1\iffalse{\fi}%
336 }%

```

We only clean-up up to 4 zero octal digits, as output was produced in chunks of 5 octal digits and as far as this author understands his own code, the leading block can not be vanishing, except when input was itself only with zeros.

```

337 \def\XINT_dto_finish !\XINT_dto_tooct!#1#2#3#4%
338 {%
339   \unless\if#10\xint_dothis{ #1#2#3#4}\fi
340   \unless\if#20\xint_dothis{ #2#3#4}\fi
341   \unless\if#30\xint_dothis{ #3#4}\fi
342   \unless\if#40\xint_dothis{ #4}\fi
343   \xint_orthat{ }%
344 }%

```

## 22.7. \xintDecToBin

An input without leading zeroes gives an output without leading zeroes.

Macros with `_dtobh` in their names are shared with `\xintDecToHex`.

**Modified at 1.2m (2017/07/31).** Complete rewrite in the 1.2 style. Also, 1.2m version is robust against non terminated inputs.

**Modified at 1.2n (2017/08/06).** Increased maximal size from using `\csname`-based expansion.

**Modified at 1.4n (2025/09/05).** Same use of `\expanded` as in `\xintDecToHex`.

```

345 \def\xintDecToBin {\romannumeral0\xintdectobin}%
346 \def\xintdectobin #1%
347 {%
348   \expandafter\XINT_dtb_checkin\romannumeral`&&@#1\xint:
349 }%
350 \def\XINT_dtb_checkin #1%
351 {%
352   \xint_UDsignfork
353     #1\XINT_dtb_neg
354     -{\XINT_dtb_main #1}%
355   \krof
356 }%
357 \def\XINT_dtb_neg {\expandafter-\romannumeral0\XINT_dtb_main}%
358 \def\XINT_dtb_main #1\xint:
359 {%
360   \expandafter\XINT_dtb_finish
361   \romannumeral`&&@\expandafter\XINT_dtb_start
362   \romannumeral0\XINT_zeroes_foriv
363     #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
364   #1\xint_bye\XINT_dtb_tobin
365 }%
366 \def\XINT_dtb_tobin
367 {%
368   \expandafter\XINT_dtb_tobin_a\expanded\XINT_tosixteenbits
369 }%
370 \def\XINT_dtb_tobin_a{!\XINT_dtb_tobin!}%
371 \def\XINT_tosixteenbits #1!%
372 {%
373   {\iffalse}\fi\expandafter\XINT_tosixteenbits_a
374   \the\numexpr (#1+\xint_c_ii^vii)/\xint_c_ii^viii-\xint_c_i\xint:
375   #1\xint:
376 }%
377 \def\XINT_tosixteenbits_a #1\xint:#2\xint:
378 {%
379   \expandafter\XINT_tosixteenbits_c
380   \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
381   #1\xint:
382   #2-\xint_c_ii^viii*#1!%
383 }%
384 \def\XINT_tosixteenbits_c #1\xint:#2\xint:
385 {%
386   \csname XINT_csdtb_#1\endcsname
387   \csname XINT_csdtb_\the\numexpr #2-\xint_c_xvi*#1\endcsname
388   \expandafter\XINT_tosixteenbits_d\the\numexpr

```

```

389 }%
390 \def\XINT_tosixteenbits_d #1!%
391 {%
392   \expandafter\XINT_tosixteenbits_e
393   \the\numexpr (#1+\xint_c_viii)/\xint_c_xvi-\xint_c_i\xint:
394   #1\xint:
395 }%
396 \def\XINT_tosixteenbits_e #1\xint:#2\xint:
397 {%
398   \csname XINT_csdtb_#1\endcsname
399   \csname XINT_csdtb_\the\numexpr #2-\xint_c_xvi*#1\endcsname
400   \iffalse{\fi}%
401 }%
402 \def\XINT_dtb_finish !\XINT_dtb_tobin!#1#2#3#4#5#6#7#8%
403 {%
404   \expandafter\XINT_dtb_finish_a\the\numexpr #1#2#3#4#5#6#7#8\relax
405 }%
406 \def\XINT_dtb_finish_a #1{%
407 \def\XINT_dtb_finish_a ##1##2##3##4##5##6##7##8##9%
408 {%
409   \expandafter#1\the\numexpr ##1##2##3##4##5##6##7##8##9\relax
410 }}\XINT_dtb_finish_a { }%

```

## 22.8. \xintHexToDec

Completely (and belatedly) rewritten at 1.2m in the 1.2 style.

1.2m version robust against non terminated inputs, but there is no primitive from TeX which may generate hexadecimal digits and provoke expansion ahead, afaik, except of course if decimal digits are treated as hexadecimal. This robustness is not on purpose but from need to expand argument and then grab it again. So we do it safely.

Increased maximal size at 1.2n.

1.2m version robust against non terminated inputs.

An input without leading zeroes gives an output without leading zeroes.

```

411 \def\xintHexToDec {\romannumeral0\xinthextodec }%
412 \def\xinthextodec #1%
413 {%
414   \expandafter\XINT_htd_checkin\romannumeral`&&@#1\xint:
415 }%
416 \def\XINT_htd_checkin #1%
417 {%
418   \xint_UDsignfork
419   #1\XINT_htd_neg
420   -{\XINT_htd_main #1}%
421   \krof
422 }%
423 \def\XINT_htd_neg {\expandafter-\romannumeral0\XINT_htd_main}%
424 \def\XINT_htd_main #1\xint:
425 {%
426   \expandafter\XINT_htd_startb
427   \the\numexpr\expandafter\XINT_htd_starta
428   \romannumeral0\XINT_zeroes_foriv
429   #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
430 #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\relax
431 }%
432 \def\xint_htd_starta #1#2#3#4{"#1#2#3#4+\xint_c_x^v!}%
433 \def\xint_htd_startb 1#1%
434 {%
435   \if#10\expandafter\xint_htd_startba\else
436     \expandafter\xint_htd_startbbb
437   \fi 1#1%
438 }%
439 \def\xint_htd_startba 10#1!{\xint_htd_again #1%
440   \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_htd_nextfour}%
441 \def\xint_htd_startbbb 1#1#2!{\xint_htd_again #1!#2%
442   \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xint_htd_nextfour}%

```

It is a bit annoying to grab all to the end here. I had a version, modeled on the 1.2n variant of [\xintDecToHex](#) which solved that problem, but it did not prove much (or at all) faster in my brief testing and it had the defect of a reduced maximal allowed size of the input.

```
443 \def\xint_htd_again #1\xint_htd_nextfour #2%
444 {%
445   \xint_bye #2\xint_htd_finish\xint_bye
446   \expandafter\xint_htd_A\the\numexpr
447   \xint_htd_a #1\xint_htd_nextfour #2%
448 }%
449 \def\xint_htd_a #1!#2!#3!#4!#5!#6!#7!#8!#9!%
450 {%
451   #1\expandafter\xint_htd_update
452   \the\numexpr #2\expandafter\xint_htd_update
453   \the\numexpr #3\expandafter\xint_htd_update
454   \the\numexpr #4\expandafter\xint_htd_update
455   \the\numexpr #5\expandafter\xint_htd_update
456   \the\numexpr #6\expandafter\xint_htd_update
457   \the\numexpr #7\expandafter\xint_htd_update
458   \the\numexpr #8\expandafter\xint_htd_update
459   \the\numexpr #9\expandafter\xint_htd_update
460   \the\numexpr \xint_htd_a
461 }%
462 \def\xint_htd_nextfour #1#2#3#4%
463 {%
464   *\xint_c_ii^xvi+"#1#2#3#4+\xint_c_x^ix\relax\xint_bye!%
465   2!3!4!5!6!7!8!9!\xint_bye\xint_htd_nextfour
466 }%
467 \def\xint_htd_update 1#1#2#3#4#5%
468 {%
469   *\xint_c_ii^xvi+10000#1#2#3#4#5!%
470 }%

```

**Modified at 1.4n (2025/09/05).** `\xint_htd_A` small refactoring to introduce shared macros with the octal conversion routine. Slight loss of efficiency.

```
471 \def\xint_htd_A {\xint_hotd_A\xint_htd_again}%
472 \def\xint_hotd_A #1!#2%
473 {%
474   \if#20\expandafter\xint_hotd_Aa\else
475     \expandafter\xint_hotd_Ab
476   \fi #1!#2%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
477 }%
478 \def\XINT_hotd_Aa #1#2#3#4#5{#1#2#3#4#5!}%
479 \def\XINT_hotd_Ab #1#2#3#4#5#6{#1#2!#3#4#5#6!}%
480 \def\XINT_htd_finish\XINT_bye
481   \expandafter\XINT_htd_A\the\numexpr \XINT_htd_a #1\XINT_htd_nextfour
482 {%
483   \expandafter\XINT_htd_finish_cuz\the\numexpr0\XINT_htd_unsep_loop #1%
484 }%
485 \def\XINT_htd_unsep_loop #1!#2!#3!#4!#5!#6!#7!#8!#9!%
486 {%
487   \expandafter\XINT_unsep_clean
488   \the\numexpr 1#1#2\expandafter\XINT_unsep_clean
489   \the\numexpr 1#3#4\expandafter\XINT_unsep_clean
490   \the\numexpr 1#5#6\expandafter\XINT_unsep_clean
491   \the\numexpr 1#7#8\expandafter\XINT_unsep_clean
492   \the\numexpr 1#9\XINT_htd_unsep_loop_a
493 }%
494 \def\XINT_htd_unsep_loop_a #1!#2!#3!#4!#5!#6!#7!#8!#9!%
495 {%
496   #1\expandafter\XINT_unsep_clean
497   \the\numexpr 1#2#3\expandafter\XINT_unsep_clean
498   \the\numexpr 1#4#5\expandafter\XINT_unsep_clean
499   \the\numexpr 1#6#7\expandafter\XINT_unsep_clean
500   \the\numexpr 1#8#9\XINT_htd_unsep_loop
501 }%
502 \def\XINT_unsep_clean 1{\relax}% also in xintcore
503 \def\XINT_htd_finish_cuz #1{%
504 \def\XINT_htd_finish_cuz ##1##2##3##4##5%
505   {\expandafter#1\the\numexpr ##1##2##3##4##5\relax}%
506 }\XINT_htd_finish_cuz{ }%
```

## 22.9. \xintOctToDec

**Added at 1.4n (2025/09/05).** Yes, the explanations are somewhat lacking. Basically this imitates `\xintxintHexToDec` the main difference being to handle 5 octal digits at a time in place of 4 hexadecimal ones.

```
507 \def\xintOctToDec {\romannumeral0\xintocttodec }%
508 \def\xintocttodec #1%
509 {%
510   \expandafter\XINT_otd_checkin\romannumeral`&&@#1\xint:
511 }%
512 \def\XINT_otd_checkin #1%
513 {%
514   \xint_UDsignfork
515     #1\XINT_otd_neg
516     -{\XINT_otd_main #1}%
517   \krof
518 }%
519 \def\XINT_otd_neg {\expandafter-\romannumeral0\XINT_otd_main}%

```

First we inject leading octal zeroes to make the length a multiple of 5. We do not try to code a direct way as with `\XINT_zeroes_foriv`, in part because as we can't grab 10 by 10, we would have to proceed 5 by 5, which for very long input may prove slower than using `\xintLength` combined with

`\xintReplicate`, as provided by *xintkernel*. However, the author is not motivated enough to do the alternative coding and compare its efficiency with the one here.

```

520 \def\xINT_otd_main #1\xint:
521 {%
522   \expandafter\xINT_otd_startb
523   \the\numexpr\expandafter\xINT_otd_starta_i
524           \romannumeral0\xintlength{#1}\xint:
525   #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\relax
526 }%
527 \def\xINT_otd_starta_i #1\xint:
528 {%
529   \expandafter\xINT_otd_starta
530   \romannumeral\xintreplicate{\xint_c_v*((#1+\xint_c_ii)/\xint_c_v)-#1}{0}%
531 }%

```

Now we grab five octal digits and convert to decimal in a `\numexpr`.

```

532 \def\xINT_otd_starta #1#2#3#4#5{'#1#2#3#4#5+\xint_c_x^v!}%
533 \def\xINT_otd_startb 1#1%
534 {%
535   \if#10\expandafter\xINT_otd_startba\else
536     \expandafter\xINT_otd_startbb
537   \fi 1#1%
538 }%
539 \def\xINT_otd_startba 10#1!\{\xINT_otd_again #1%
540   \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xINT_otd_nextfive}%
541 \def\xINT_otd_startbb 1#1#2!\{\xINT_otd_again #1!#2%
542   \xint_bye!2!3!4!5!6!7!8!9!\xint_bye\xINT_otd_nextfive}%
543 \def\xINT_otd_again #1\xINT_otd_nextfive #2%
544 {%
545   \xint_bye #2\xINT_otd_finish\xint_bye
546   \expandafter\xINT_otd_A\the\numexpr
547   \xINT_otd_a #1\xINT_otd_nextfive #2%
548 }%
549 \def\xINT_otd_a #1!#2!#3!#4!#5!#6!#7!#8!#9!%
550 {%
551   #1\expandafter\xINT_otd_update
552   \the\numexpr #2\expandafter\xINT_otd_update
553   \the\numexpr #3\expandafter\xINT_otd_update
554   \the\numexpr #4\expandafter\xINT_otd_update
555   \the\numexpr #5\expandafter\xINT_otd_update
556   \the\numexpr #6\expandafter\xINT_otd_update
557   \the\numexpr #7\expandafter\xINT_otd_update
558   \the\numexpr #8\expandafter\xINT_otd_update
559   \the\numexpr #9\expandafter\xINT_otd_update
560   \the\numexpr \xINT_otd_a
561 }%
562 \def\xINT_otd_nextfive #1#2#3#4#5%
563 {%
564   *\xint_c_ii^xv+'#1#2#3#4#5+\xint_c_x^ix\relax\xint_bye!%
565   2!3!4!5!6!7!8!9!\xint_bye\xINT_otd_nextfive
566 }%
567 \def\xINT_otd_update 1#1#2#3#4#5%
568 {%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
569      *\xint_c_ii^xv+10000#1#2#3#4#5!%
570 }%
```

We can hook here into the `\xintHexToDec` final sub-routines.

```
571 \def\xINT_otd_A {\XINT_hotd_A\XINT_otd_again}%
572 \def\xINT_otd_finish\xint_bye
573   \expandafter\xINT_otd_A\the\numexpr \XINT_otd_a #1\xINT_otd_nextfive
574 {%
575   \expandafter\xINT_htd_finish_cuz\the\numexpr0\xINT_htd_unsep_loop #1%
576 }%
```

### 22.10. `\xintBinToDec`

Redone entirely for 1.2m. Starts by converting to hexadecimal first.

Increased maximal size at 1.2n.

An input without leading zeroes gives an output without leading zeroes.

Robust against non-terminated input.

```
577 \def\xintBinToDec {\romannumeral0\xintbintodec }%
578 \def\xintbintodec #1%
579 {%
580   \expandafter\xINT_btd_checkin\romannumeral`&&@#1\xint:
581 }%
582 \def\xINT_btd_checkin #1%
583 {%
584   \xint_UDsignfork
585     #1\xINT_btd_N
586     -{\XINT_btd_main #1}%
587   \krof
588 }%
589 \def\xINT_btd_N {\expandafter-\romannumeral0\xINT_btd_main }%
590 \def\xINT_btd_main #1\xint:
591 {%
592   \expandafter\xINT_btd_htd
593   \expanded{\expandafter\xINT_bth_loop
594     \romannumeral0\xINT_zeroes_foriv
595     #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
596     #1nonenone\xint_bye}\xint:
597 }%
598 \def\xINT_btd_htd #1\xint:
599 {%
600   \expandafter\xINT_htd_startb
601   \the\numexpr\expandafter\xINT_htd_starta
602   \romannumeral0\xINT_zeroes_foriv
603   #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
604   #1\xint_bye!2!3!4!5!6!7!8!9!\xint_bye\relax
605 }%
```

### 22.11. `\xintHexToOct`

**Added at 1.4n (2025/09/05).** This is done the lazy way, from hexadecimal to binary to octal. I am simply not motivated enough at this stage to implement a direct conversion (3 hexa digits mapping to 4 octal ones), handle leading zeros, and compare efficiency and size limits with the simple minded one here.

```

606 \def\xintHexToOct {\romannumeral0\xinthextooct }%
607 \def\xinthextooct #1%
608 {%
609     \expandafter\XINT_bto_checkin
610     \romannumeral0\expandafter\XINT_htb_checkin\romannumeral`&&@#1.....\xint_bye
611     \iffalse{\fi}%
612     \xint:
613 }%

```

## 22.12. \xintHexToBin

Completely rewritten for 1.2m.

Only up to three zeros are removed on front of output: if the input had a leading zero, there will be a leading zero (and then possibly 4n of them if inputs had more leading zeroes) on output.

Rewritten again at 1.2n for \csname governed expansion.

**Modified at 1.4n (2025/09/05).** Use of \expanded (quasi globally, in contrast to the local uses made in the decimal conversions).. Dramatic increase of upper limit on the size of input.

As a by-product the initial expansion of the argument #1 is now guarded at the end by (many) full stops, but anyhow I don't know T<sub>E</sub>X construct potentially creating hexadecimal digits and potentially causing expansion of what comes next if not terminated.

```

614 \def\xintHexToBin {\romannumeral0\xinthextobin }%
615 \def\xinthextobin #1%
616 {%
617     \expandafter\XINT_htb_checkin\romannumeral`&&@#1.....\xint_bye
618     \iffalse{\fi}%
619 }%
620 \def\XINT_htb_checkin #1%
621 {%
622     \xint_UDsignfork
623     #1\XINT_htb_N
624     -{\XINT_htb_main #1}%
625     \krof
626 }%
627 \def\XINT_htb_N {\expandafter-\romannumeral0\XINT_htb_main }%
628 \def\XINT_htb_main
629 {%
630     \expandafter\XINT_htb_cuz
631     \expanded{\iffalse}\fi\XINT_htb_loop
632 }%
633 \def\XINT_htb_loop #1#2#3#4#5#6#7#8#9%
634 {%
635     \csname XINT_cshtb_#1\endcsname
636     \csname XINT_cshtb_#2\endcsname
637     \csname XINT_cshtb_#3\endcsname
638     \csname XINT_cshtb_#4\endcsname
639     \csname XINT_cshtb_#5\endcsname
640     \csname XINT_cshtb_#6\endcsname
641     \csname XINT_cshtb_#7\endcsname
642     \csname XINT_cshtb_#8\endcsname
643     \csname XINT_cshtb_#9\endcsname
644     \XINT_htb_loop
645 }%

```



```

646 \expandafter\let\csname XINT_cshtb_.\endcsname\xint_bye
647 \def\xint_htb_cuz #1{%
648 \def\xint_htb_cuz ##1##2##3##4%
649   {\expandafter#1\the\numexpr##1##2##3##4\relax}%
650 }\xint_htb_cuz { }%

```

### 22.13. \xintCHexToBin

The 1.08 macro had same functionality as `\xintHexToBin`, and slightly different code, the 1.2m version has the same code as `\xintHexToBin` except that it does not remove leading zeros from output: if the input had N hexadecimal digits, the output will have exactly 4N binary digits.

Rewritten again at 1.2n for `\csname` governed expansion.

**Modified at 1.4n (2025/09/05).** Kept in sync with new `\xintHexToBin`.

```

651 \def\xintCHexToBin {\romannumeral0\xintchextobin }%
652 \def\xintchextobin #1%
653 {%
654   \expandafter\xint_chtb_checkin\romannumeral`&&@#1.....\xint_bye
655   \iffalse{\fi}%
656 }%
657 \def\xint_chtb_checkin #1%
658 {%
659   \xint_UDsignfork
660     #1\xint_chtb_N
661     -{\xint_chtb_main #1}%
662   \krof
663 }%
664 \def\xint_chtb_N {\expandafter-\romannumeral0\xint_chtb_main }%
665 \def\xint_chtb_main {\expanded{ \iffalse}\fi\xint_htb_loop}%

```

### 22.14. \xintOctToHex

**Added at 1.4n (2025/09/05).** This is done the lazy way, from octal to binary to hexadecimal.

```

666 \def\xintOctToHex {\romannumeral0\xintocttohex }%
667 \def\xintocttohex #1%
668 {%
669   \expandafter\xint_bth_checkin
670   \romannumeral0\expandafter\xint_otb_checkin\romannumeral`&&@#1.....\xint_bye
671   \iffalse{\fi}%
672   \xint:
673 }%

```

### 22.15. \xintBinToHex

**Modified at 1.2m (2017/07/31).** Complete rewrite. Much more efficient but smaller maximal sizes.

**Modified at 1.2n (2017/08/06).** Again redone and now using `\csname` governed expansion: increased maximal size.

Size of output is  $\text{ceil}(\text{size}(\text{input})/4)$ . If the input has leading zeroes, they may exist in the output too. An input without leading zeroes gives an output without leading zeroes.

**Modified at 1.4n (2025/09/05).** Expansion is governed by `\expanded`. Tremendous increase of maximal size. Note that for legacy reasons of the package history, the primary trigger is via `\romannumeral0`, we could use `\expanded` upfront, with some advantages. But well.

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
674 \def\xintBinToHex {\romannumeral0\xintbinto hex }%
675 \def\xintbinto hex #1%
676 {%
677   \expandafter\XINT_bth_checkin\romannumeral`&&@#1\xint:
678 }%
679 \def\XINT_bth_checkin #1%
680 {%
681   \xint_UDsignfork
682     #1\XINT_bth_N
683     -{\XINT_bth_main #1}%
684   \krof
685 }%
686 \def\XINT_bth_N {\expandafter-\romannumeral0\XINT_bth_main }%
687 \def\XINT_bth_main #1\xint:
688 {%
689   \expanded{ \expandafter\XINT_bth_loop
690     \romannumeral0\XINT_zeroes_foriv
691       #1\R{0\R}{00\R}{000\R}\R{0\R}{00\R}{000\R}\R\W
692     #1%
693     nonenone\xint_bye}%
694 }%
695 \def\XINT_bth_loop #1#2#3#4#5#6#7#8%
696 {%
697   \csname XINT_csbth_#1#2#3#4\endcsname
698   \csname XINT_csbth_#5#6#7#8\endcsname
699   \XINT_bth_loop
700 }%
701 \let\XINT_csbth_none\xint_bye
```

## 22.16. \xintOctToBin

Added at 1.4n (2025/09/05).

```
702 \def\xintOctToBin {\romannumeral0\xintocttobin }%
703 \def\xintocttobin #1%
704 {%
705   \expandafter\XINT_otb_checkin\romannumeral`&&@#1.....\xint_bye
706   \iffalse{\fi}%
707 }%
708 \def\XINT_otb_checkin #1%
709 {%
710   \xint_UDsignfork
711     #1\XINT_otb_N
712     -{\XINT_otb_main #1}%
713   \krof
714 }%
715 \def\XINT_otb_N {\expandafter-\romannumeral0\XINT_otb_main }%
716 \def\XINT_otb_main
717 {%
718   \expandafter\XINT_otb_cuz
719   \expanded{\iffalse}\fi \XINT_otb_loop
720 }%
721 \def\XINT_otb_loop #1#2#3#4#5#6#7#8#9%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, xintbinhex, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
722 {%
723     \csname XINT_csotb_#1\endcsname
724     \csname XINT_csotb_#2\endcsname
725     \csname XINT_csotb_#3\endcsname
726     \csname XINT_csotb_#4\endcsname
727     \csname XINT_csotb_#5\endcsname
728     \csname XINT_csotb_#6\endcsname
729     \csname XINT_csotb_#7\endcsname
730     \csname XINT_csotb_#8\endcsname
731     \csname XINT_csotb_#9\endcsname
732     \XINT_otb_loop
733 }%
734 \expandafter\let\csname XINT_csotb_.\endcsname\xint_bye
735 \def\xint_otb_cuz #1{%
736 \def\xint_otb_cuz ##1##2##3%
737     {\expandafter#1\the\numexpr##1##2##3\relax}%
738 }\xint_otb_cuz { }%
```

### 22.17. \xintCOctToBin

Added at 1.4n (2025/09/05).

```
739 \def\xintCOctToBin {\romannumeral0\xintcocttobin }%
740 \def\xintcocttobin #1%
741 {%
742     \expandafter\xint_cotb_checkin\romannumeral`&&@#1.....\xint_bye
743     \iffalse{\fi}%
744 }%
745 \def\xint_cotb_checkin #1%
746 {%
747     \xint_UDsignfork
748         #1\xint_cotb_N
749         -{\xint_cotb_main #1}%
750     \krof
751 }%
752 \def\xint_cotb_N {\expandafter-\romannumeral0\xint_cotb_main }%
753 \def\xint_cotb_main {\expanded{ \iffalse}\fi\xint_otb_loop}%

```

### 22.18. \xintBinToOct

Added at 1.4n (2025/09/05).

```
754 \def\xintBinToOct {\romannumeral0\xintbintoct }%
755 \def\xintbintoct #1%
756 {%
757     \expandafter\xint_bto_checkin\romannumeral`&&@#1\xint:
758 }%
759 \def\xint_bto_checkin #1%
760 {%
761     \xint_UDsignfork
762         #1\xint_bto_N
763         -{\xint_bto_main #1}%
764     \krof
765 }%

```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
766 \def\XINT_bto_N {\expandafter\romannumeral0\XINT_bto_main }%
767 \def\XINT_bto_main #1\xint:
768 {%
769   \expanded{ \expandafter\XINT_bto_loop
770     \romannumeral0\XINT_zeroes_foriii
771       #1\R{0\R}{00\R}\R{0\R}{00\R}\R{0\R}{00\R}\R\W
772     #1%
773   \endendend\xint_bye}%
774 }%
775 \def\XINT_bto_loop #1#2#3#4#5#6#7#8#9%
776 {%
777   \csname XINT_csbto_#1#2#3\endcsname
778   \csname XINT_csbto_#4#5#6\endcsname
779   \csname XINT_csbto_#7#8#9\endcsname
780   \XINT_bto_loop
781 }%
782 \let\XINT_csbto_end\xint_bye
783 \XINTrestorecatcodesendinginput%
```

## 23. Package [xintgcd](#) implementation

.1	Catcodes, $\varepsilon$ -TeX and reload detection . .	420	.5	<code>\xintBezoutAlgorithm</code> . . . . .	426
.2	Package identification . . . . .	421	.6	<code>\xintTypesetEuclideanAlgorithm</code> . . . .	428
.3	<code>\xintBezout</code> . . . . .	421	.7	<code>\xintTypesetBezoutAlgorithm</code> . . . .	428
.4	<code>\xintEuclideanAlgorithm</code> . . . . .	425			

The commenting is currently (2025/09/06) very sparse.

Release [1.09h](#) has modified a bit the `\xintTypesetEuclideanAlgorithm` and `\xintTypesetBezoutAlgorithm` layout with respect to line indentation in particular. And they use the [xinttools](#) `\xintloop` rather than the Plain TeX or  $\mathbb{X}$ 's `\loop`.

Breaking change at [1.2p](#): `\xintBezout{A}{B}` formerly had output  $\{A\}{B}\{U\}{V}\{D\}$  with  $AU-BV=D$ , now it is  $\{U\}{V}\{D\}$  with  $AU+BV=D$ .

From [1.1](#) to [1.3f](#) the package loaded only [xintcore](#). At [1.4](#) it now automatically loads both of [xint](#) and [xinttools](#) (the latter being in fact a requirement of `\xintTypesetEuclideanAlgorithm` and `\xintTypesetBezoutAlgorithm` since [1.09h](#)).



Changed  
at 1.4!

At [1.4](#) `\xintGCD`, `\xintLCM`, `\xintGCDof`, and `\xintLCMof` are removed from the package: they are provided only by [xintfrac](#) and they handle general fractions, not only integers.

The original integer-only macros have been renamed into respectively `\xintiigcd`, `\xintiilcm`, `\xintiigcdof`, and `\xintiilcmof` and got relocated into [xint](#) package.

### 23.1. Catcodes, $\varepsilon$ -TeX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^
11 \def\empty{}\def\space{ }\newlinechar10
12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintgcd.sty\endcsname
14 \expandafter\let\expandafter>w\csname ver@xint.sty\endcsname
15 \expandafter\let\expandafter\t\csname ver@xinttools.sty\endcsname
16 \expandafter\ifx\csname numexpr\endcsname\relax
17 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
18 \immediate\write128{^^JPackage xintgcd Warning:^^J%
19 \space\space\space\space
20 \numexpr not available, aborting input.^^J}%
21 \else
22 \PackageWarningNoLine{xintgcd}{\numexpr not available, aborting input}%
23 \fi
24 \def\z{\endgroup\endinput}%
25 \else

```

```

26 \ifx\x\relax % plain-TeX, first loading of xintgcd.sty
27 \ifx\w\relax % but xint.sty not yet loaded.
28 \expandafter\def\expandafter\z\expandafter{\z\input xint.sty\relax}%
29 \fi
30 \ifx\t\relax % but xinttools.sty not yet loaded.
31 \expandafter\def\expandafter\z\expandafter{\z\input xinttools.sty\relax}%
32 \fi
33 \else
34 \ifx\x\empty % LaTeX, first loading,
35 % variable is initialized, but \ProvidesPackage not yet seen
36 \ifx\w\relax % xint.sty not yet loaded.
37 \expandafter\def\expandafter\z\expandafter{\z\RequirePackage{xint}}%
38 \fi
39 \ifx\t\relax % xinttools.sty not yet loaded.
40 \expandafter\def\expandafter\z\expandafter{\z\RequirePackage{xinttools}}%
41 \fi
42 \else
43 \def\z{\endgroup\endinput}% xintgcd already loaded.
44 \fi
45 \fi
46 \fi
47 \z%
48 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 23.2. Package identification

```

49 \XINT_providespackage
50 \ProvidesPackage{xintgcd}%
51 [2025/09/06 v1.4o Euclidean algorithm with xint package (JFB)]%

```

## 23.3. \xintBezout

`\xintBezout{#1}{#2}` produces  $\{U\}{V\}{D}$  with  $UA+VB=D$ ,  $D = \text{PGCD}(A,B)$  (non-positive), where #1 and #2 f-expand to big integers A and B.

I had not checked this macro for about three years when I realized in January 2017 that `\xintBezout{A}{B}` was buggy for the cases  $A = 0$  or  $B = 0$ . I fixed that blemish in 1.2l but overlooked the other blemish that `\xintBezout{A}{B}` with A multiple of B produced a coefficient U as  $-0$  in place of 0.

Hence I rewrote again for 1.2p. On this occasion I modified the output of the macro to be  $\{U\}{V\}{D}$  with  $AU+BV=D$ , formerly it was  $\{A\}{B\}{U\}{V\}{D}$  with  $AU - BV = D$ . This is quite breaking change!

Note in particular change of sign of V.

I don't know why I had designed this macro to contain  $\{A\}{B\}$  in its output. Perhaps I initially intended to output  $\{A//D\}{B//D\}$  (but forgot), as this is actually possible from outcome of the last iteration, with no need of actually dividing. Current code however arranges to skip this last update, as U and V are already furnished by the iteration prior to realizing that the last non-zero remainder was found.

Also 1.2l raised InvalidOperation if both A and B vanished, but I removed this behaviour at 1.2p.

```

52 \def\xintBezout {\romannumeral0\xintbezout }%
53 \def\xintbezout #1%
54 {%
55 \expandafter\XINT_bezout\expandafter {\romannumeral0\xintnum{#1}}%
56 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

57 \def\XINT_bezout #1#2%
58 {%
59   \expandafter\XINT_bezout_fork \romannumeral0\xintnum{#2}\Z #1\Z
60 }%
61 #3#4 = A, #1#2=B. Micro improvement for 1.21.
62 \def\XINT_bezout_fork #1#2\Z #3#4\Z
63 {%
64   \xint_UDzerosfork
65   #1#3\XINT_bezout_botharezero
66   #10\XINT_bezout_secondiszero
67   #30\XINT_bezout_firstiszero
68   00\xint_UDsignsfork
69   \krof
70   #1#3\XINT_bezout_minusminus % A < 0, B < 0
71   #1-\XINT_bezout_minusplus % A > 0, B < 0
72   #3-\XINT_bezout_plusminus % A < 0, B > 0
73   --\XINT_bezout_plusplus % A > 0, B > 0
74   \krof
75   {#2}{#4}#1#3% #1#2=B, #3#4=A
76 }%
77 \def\XINT_bezout_botharezero #1\krof#2#300{{0}{0}{0}}%
78 \def\XINT_bezout_firstiszero #1\krof#2#3#4#5%
79 {%
80   \xint_UDsignfork
81   #4{{0}{-1}{#2}}%
82   -{{0}{1}{#4#2}}%
83   \krof
84 }%
85 \def\XINT_bezout_secondiszero #1\krof#2#3#4#5%
86 {%
87   \xint_UDsignfork
88   #5{{-1}{0}{#3}}%
89   -{{1}{0}{#5#3}}%
90   \krof
91 }%
92 #4#2= A < 0, #3#1 = B < 0
93 \def\XINT_bezout_minusminus #1#2#3#4%
94 {%
95   \expandafter\XINT_bezout_mm_post
96   \romannumeral0\expandafter\XINT_bezout_preloop_a
97   \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
98 }%
99 \def\XINT_bezout_mm_post #1#2%
100 {%
101   \expandafter\XINT_bezout_mm_postb\expandafter
102   {\romannumeral0\xintiopp{#2}}{\romannumeral0\xintiopp{#1}}%
103 }%
104 \def\XINT_bezout_mm_postb #1#2{\expandafter{#2}{#1}}%
105 minusplus #4#2= A > 0, B < 0
106 \def\XINT_bezout_minusplus #1#2#3#4%
107 {%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, xintgcd, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

105 \expandafter\XINT_bezout_mp_post
106 \romannumeral0\expandafter\XINT_bezout_preloop_a
107 \romannumeral0\XINT_div_prepare {#1}{#4#2}{#1}%
108 }%
109 \def\XINT_bezout_mp_post #1#2%
110 {%
111 \expandafter\xint_exchangetwo_keepbraces\expandafter
112 {\romannumeral0\xintiopp {#2}}{#1}%
113 }%
    plusminus A < 0, B > 0
114 \def\XINT_bezout_plusminus #1#2#3#4%
115 {%
116 \expandafter\XINT_bezout_pm_post
117 \romannumeral0\expandafter\XINT_bezout_preloop_a
118 \romannumeral0\XINT_div_prepare {#3#1}{#2}{#3#1}%
119 }%
120 \def\XINT_bezout_pm_post #1{\expandafter{\romannumeral0\xintiopp{#1}}}%
    plusplus, B = #3#1 > 0, A = #4#2 > 0
121 \def\XINT_bezout_plusplus #1#2#3#4%
122 {%
123 \expandafter\XINT_bezout_preloop_a
124 \romannumeral0\XINT_div_prepare {#3#1}{#4#2}{#3#1}%
125 }%
    n = 0: BA1001 (B, A, e=1, vv, uu, v, u)
    r(1)=B, r(0)=A, après n étapes {r(n+1)}{r(n)}{vv}{uu}{v}{u}
    q(n) quotient de r(n-1) par r(n)
    si reste nul, exit et renvoie U = -e*uu, V = e*vv, A*U+B*V=D
    sinon mise à jour
        vv, v = q * vv + v, vv
        uu, u = q * uu + u, uu
        e = -e
    puis calcul quotient reste et itération
    We arrange for \xintiiMul sub-routine to be called only with positive arguments, thus skipping
    some un-needed sign parsing there. For that though we have to screen out the special cases A
    divides B, or B divides A. And we first want to exchange A and B if A < B. These special cases are
    the only one possibly leading to U or V zero (for A and B positive which is the case here.) Thus the
    general case always leads to non-zero U and V's and assigning a final sign is done simply adding a
    - to one of them, with no fear of producing -0.
126 \def\XINT_bezout_preloop_a #1#2#3%
127 {%
128 \if0#1\xint_dothis\XINT_bezout_preloop_exchange\fi
129 \if0#2\xint_dothis\XINT_bezout_preloop_exit\fi
130 \xint_orthat{\expandafter\XINT_bezout_loop_B}%
131 \romannumeral0\XINT_div_prepare {#2}{#3}{#2}{#1}110%
132 }%
133 \def\XINT_bezout_preloop_exit
134 \romannumeral0\XINT_div_prepare #1#2#3#4#5#6#7%
135 {%
136 {0}{1}{#2}%
137 }%
138 \def\XINT_bezout_preloop_exchange

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, xintgcd, *xintfrac*, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

139 {%
140     \expandafter\xint_exchangetwo_keepbraces
141     \romannumeral0\expandafter\XINT_bezout_preloop_A
142 }%
143 \def\XINT_bezout_preloop_A #1#2#3#4%
144 {%
145     \if0#2\xint_dothis\XINT_bezout_preloop_exit\fi
146     \xint_orthat{\expandafter\XINT_bezout_loop_B}%
147     \romannumeral0\XINT_div_prepare {#2}{#3}{#2}{#1}%
148 }%
149 \def\XINT_bezout_loop_B #1#2%
150 {%
151     \if0#2\expandafter\XINT_bezout_exitA
152     \else\expandafter\XINT_bezout_loop_C
153     \fi {#1}{#2}%
154 }%

```

We use the fact that the `\romannumeral-`0` (or equivalent) done by `\xintiiadd` will absorb the initial space token left by `\XINT_mul_plusplus` in its output.

We arranged for operands here to be always positive which is needed for `\XINT_mul_plusplus` entry point (last time I checked...). Admittedly this kind of optimization is not good for maintenance of code, but I can't resist temptation of limiting the shuffling around of tokens...

```

155 \def\XINT_bezout_loop_C #1#2#3#4#5#6#7%
156 {%
157     \expandafter\XINT_bezout_loop_D\expandafter
158     {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}}{#1\xint:#4\xint:}{#6}}%
159     {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}}{#1\xint:#5\xint:}{#7}}%
160     {#2}{#3}{#4}{#5}%
161 }%
162 \def\XINT_bezout_loop_D #1#2%
163 {%
164     \expandafter\XINT_bezout_loop_E\expandafter{#2}{#1}%
165 }%
166 \def\XINT_bezout_loop_E #1#2#3#4%
167 {%
168     \expandafter\XINT_bezout_loop_b
169     \romannumeral0\XINT_div_prepare {#3}{#4}{#3}{#2}{#1}%
170 }%
171 \def\XINT_bezout_loop_b #1#2%
172 {%
173     \if0#2\expandafter\XINT_bezout_exitA
174     \else\expandafter\XINT_bezout_loop_c
175     \fi {#1}{#2}%
176 }%
177 \def\XINT_bezout_loop_c #1#2#3#4#5#6#7%
178 {%
179     \expandafter\XINT_bezout_loop_d\expandafter
180     {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}}{#1\xint:#4\xint:}{#6}}%
181     {\romannumeral0\xintiiadd{\XINT_mul_plusplus{}}{#1\xint:#5\xint:}{#7}}%
182     {#2}{#3}{#4}{#5}%
183 }%
184 \def\XINT_bezout_loop_d #1#2%
185 {%

```

```

186 \expandafter\XINT_bezout_loop_e\expandafter{#2}{#1}%
187 }%
188 \def\XINT_bezout_loop_e #1#2#3#4%
189 {%
190 \expandafter\XINT_bezout_loop_B
191 \romannumeral0\XINT_div_prepare {#3}{#4}{#3}{#2}{#1}%
192 }%

```

sortir U, V, D mais on a travaillé avec vv, uu, v, u dans cet ordre.

The code is structured so that #4 and #5 are guaranteed non-zero if we exit here, hence we can not create a -0 in output.

```

193 \def\XINT_bezout_exita #1#2#3#4#5#6#7{ {-#5}{#4}{#3}}%
194 \def\XINT_bezout_exita #1#2#3#4#5#6#7{ {-#5}{-#4}{#3}}%

```

### 23.4. \xintEuclideanAlgorithm

Pour Euclide:  $\{N\}\{A\}\{D=r(n)\}\{B\}\{q_1\}\{r_1\}\{q_2\}\{r_2\}\{q_3\}\{r_3\}\dots\{q_N\}\{r_N=0\}$

$u_{<2n} = u_{<2n+3} > u_{<2n+2} > + u_{<2n+4} >$  à la  $n$  ième étape.

Formerly, used `\xintiabs`, but got deprecated at 1.2o.

```

195 \def\xintEuclideanAlgorithm {\romannumeral0\xinteucidealalgorithm }%
196 \def\xinteucidealalgorithm #1%
197 {%
198 \expandafter\XINT_euc\expandafter{\romannumeral0\xintiabs{\xintNum{#1}}}%
199 }%
200 \def\XINT_euc #1#2%
201 {%
202 \expandafter\XINT_euc_fork\romannumeral0\xintiabs{\xintNum{#2}}\Z #1\Z
203 }%
204
205 Ici #3#4=A, #1#2=B
206 \def\XINT_euc_fork #1#2\Z #3#4\Z
207 {%
208 \xint_UDzerofork
209 #1\XINT_euc_BisZero
210 #3\XINT_euc_AisZero
211 0\XINT_euc_a
212 \krof
213 {0}{#1#2}{#3#4}{#3#4}{#1#2}}\Z
214 }%

```

Le {} pour protéger  $\{A\}\{B\}$  si on s'arrête après une étape (B divise A). On va renvoyer:

$\{N\}\{A\}\{D=r(n)\}\{B\}\{q_1\}\{r_1\}\{q_2\}\{r_2\}\{q_3\}\{r_3\}\dots\{q_N\}\{r_N=0\}$

```

213 \def\XINT_euc_AisZero #1#2#3#4#5#6{{1}{0}{#2}{#2}{0}{0}}%
214 \def\XINT_euc_BisZero #1#2#3#4#5#6{{1}{0}{#3}{#3}{0}{0}}%
215 {n}{rn}{an}{qn}{rn}}...{\{A\}\{B\}}\Z
216 a(n) = r(n-1). Pour n=0 on a juste {0}\{B\}\{A\}\{A\}\{B\}}\Z
217 \XINT_div_prepare {u}{v} divise v par u
218 \def\XINT_euc_a #1#2#3%
219 {%
220 \expandafter\XINT_euc_b\the\numexpr #1+\xint_c_i\expandafter.%
221 \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
222 }%

```

```

{#n+1}{#q(n+1)}{#r(n+1)}{#rn}}{#qn}{#rn}}...
220 \def\XINT_euc_b #1.#2#3#4%
221 {%
222     \XINT_euc_c #3\Z {#1}{#3}{#4}{{#2}{#3}}%
223 }%
r(n+1)\Z {#n+1}{#r(n+1)}{#r(n)}{{#q(n+1)}{#r(n+1)}}{{#qn}{#rn}}...
Test si r(n+1) est nul.
224 \def\XINT_euc_c #1#2\Z
225 {%
226     \xint_gob_til_zero #1\XINT_euc_end0\XINT_euc_a
227 }%
{#n+1}{#r(n+1)}{#r(n)}{{#q(n+1)}{#r(n+1)}}...{\Z Ici r(n+1) = 0. On arrête on se prépare à inverser
{#n+1}{#0}{#r(n)}{{#q(n+1)}{#r(n+1)}}...{{#q1}{#r1}}{#A}{#B}}{\Z
On veut renvoyer: {#N=#n+1}{#A}{#D=r(n)}{#B}{#q1}{#r1}{#q2}{#r2}{#q3}{#r3}...{#qN}{#rN=0}
228 \def\XINT_euc_end0\XINT_euc_a #1#2#3#4\Z%
229 {%
230     \expandafter\XINT_euc_end_a
231     \romannumeral0%
232     \XINT_rord_main {#4}{{#1}{#3}}%
233     \xint:
234         \xint_bye\xint_bye\xint_bye\xint_bye
235         \xint_bye\xint_bye\xint_bye\xint_bye
236     \xint:
237 }%
238 \def\XINT_euc_end_a #1#2#3{{#1}{#3}{#2}}%

```

### 23.5. \xintBezoutAlgorithm

Pour Bezout: objectif, renvoyer

$\{N\}\{A\}\{0\}\{1\}\{D=r(n)\}\{B\}\{1\}\{0\}\{q_1\}\{r_1\}\{\alpha_1=q_1\}\{\beta_1=1\}$   
 $\{q_2\}\{r_2\}\{\alpha_2\}\{\beta_2\}\dots\{q_N\}\{r_N=0\}\{\alpha_N=A/D\}\{\beta_N=B/D\}$   
 $\alpha_0=1, \beta_0=0, \alpha(-1)=0, \beta(-1)=1$

```

239 \def\xintBezoutAlgorithm {\romannumeral0\xintbezoutalgorithm}%
240 \def\xintbezoutalgorithm #1%
241 {%
242     \expandafter \XINT_bezalg
243     \expandafter{\romannumeral0\xintiabs{\xintNum{#1}}}%
244 }%
245 \def\XINT_bezalg #1#2%
246 {%
247     \expandafter\XINT_bezalg_fork\romannumeral0\xintiabs{\xintNum{#2}}\Z #1\Z
248 }%
Ici #3#4=A, #1#2=B
249 \def\XINT_bezalg_fork #1#2\Z #3#4\Z
250 {%
251     \xint_UDzerofork
252     #1\XINT_bezalg_BisZero
253     #3\XINT_bezalg_AisZero
254     0\XINT_bezalg_a
255     \krof
256     0{#1#2}{#3#4}1001{{#3#4}{#1#2}}{\Z

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

257 }%
258 \def\XINT_bezalg_AisZero #1#2#3\Z{{1}{0}{0}{1}{#2}{#2}{1}{0}{0}{0}{0}{1}}%
259 \def\XINT_bezalg_BisZero #1#2#3#4\Z{{1}{0}{0}{1}{#3}{#3}{1}{0}{0}{0}{0}{1}}%
    pour préparer l'étape n+1 il faut {n}{r(n)}{r(n-1)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}
    {{q(n)}{r(n)}{alpha(n)}{beta(n)}}... division de #3 par #2
260 \def\XINT_bezalg_a #1#2#3%
261 {%
262     \expandafter\XINT_bezalg_b\the\numexpr #1+\xint_c_i\expandafter.%
263     \romannumeral0\XINT_div_prepare {#2}{#3}{#2}%
264 }%
    {n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}{alpha(n-1)}{beta(n-1)}...
265 \def\XINT_bezalg_b #1.#2#3#4#5#6#7#8%
266 {%
267     \expandafter\XINT_bezalg_c\expandafter
268     {\romannumeral0\xintiiadd {\xintiiMul {#6}{#2}}{#8}}%
269     {\romannumeral0\xintiiadd {\xintiiMul {#5}{#2}}{#7}}%
270     {#1}{#2}{#3}{#4}{#5}{#6}%
271 }%
    {beta(n+1)}{alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{alpha(n)}{beta(n)}
272 \def\XINT_bezalg_c #1#2#3#4#5#6%
273 {%
274     \expandafter\XINT_bezalg_d\expandafter {#2}{#3}{#4}{#5}{#6}{#1}%
275 }%
    {alpha(n+1)}{n+1}{q(n+1)}{r(n+1)}{r(n)}{beta(n+1)}
276 \def\XINT_bezalg_d #1#2#3#4#5#6#7#8%
277 {%
278     \XINT_bezalg_e #4\Z {#2}{#4}{#5}{#1}{#6}{#7}{#8}{{#3}{#4}{#1}{#6}}%
279 }%
    r(n+1)\Z {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}
    {alpha(n)}{beta(n)}{q,r,alpha,beta(n+1)}
    Test si r(n+1) est nul.
280 \def\XINT_bezalg_e #1#2\Z
281 {%
282     \xint_gob_til_zero #1\XINT_bezalg_end0\XINT_bezalg_a
283 }%
    Ici r(n+1) = 0. On arrête on se prépare à inverser.
    {n+1}{r(n+1)}{r(n)}{alpha(n+1)}{beta(n+1)}{alpha(n)}{beta(n)}
    {q,r,alpha,beta(n+1)}... {{A}{B}}{\Z}
    On veut renvoyer
    {N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
    {q2}{r2}{alpha2}{beta2}... {qN}{rN=0}{alphaN=A/D}{betaN=B/D}
284 \def\XINT_bezalg_end0\XINT_bezalg_a #1#2#3#4#5#6#7#8\Z
285 {%
286     \expandafter\XINT_bezalg_end_a
287     \romannumeral0%
288     \XINT_rord_main {}#8{{#1}{#3}}%
289     \xint:
290     \xint_bye\xint_bye\xint_bye\xint_bye
291     \xint_bye\xint_bye\xint_bye\xint_bye
292     \xint:

```

```

293 }%
{N}{D}{A}{B}{q1}{r1}{alpha1=q1}{beta1=1}{q2}{r2}{alpha2}{beta2}
...{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
On veut renvoyer
{N}{A}{0}{1}{D=r(n)}{B}{1}{0}{q1}{r1}{alpha1=q1}{beta1=1}
{q2}{r2}{alpha2}{beta2}...{qN}{rN=0}{alphaN=A/D}{betaN=B/D}
294 \def\XINT_bezalg_end_a #1#2#3#4{{#1}{#3}{0}{1}{#2}{#4}{1}{0}}%

```

## 23.6. \xintTypesetEuclideanAlgorithm

TYPESETTING

Organisation:

$\{N\}\{A\}\{D\}\{B\}\{q1\}\{r1\}\{q2\}\{r2\}\{q3\}\{r3\}...\{qN\}\{rN=0\}$

$U1 = N = \text{nombre d'étapes}$ ,  $U3 = \text{PGCD}$ ,  $U2 = A$ ,  $U4=B$   $q1 = U5$ ,  $q2 = U7 \rightarrow qn = U<2n+3>$ ,  $rn =$

$U<2n+4> bn = rn$ .  $B = r0$ .  $A=r(-1)$

$r(n-2) = q(n)r(n-1)+r(n)$  (n e étape)

$U\{2n\} = U\{2n+3\} \times U\{2n+2\} + U\{2n+4\}$ , n e étape. (avec n entre 1 et N)

1.09h uses `\xintloop`, and `\par` rather than `\endgraf`; and `\par` rather than `\hfill\break`

```

295 \def\xintTypesetEuclideanAlgorithm {%
296   \unless\ifdefined\xintAssignArray
297     \errmessage
298     {xintgcd: package xinttools is required for \string\xintTypesetEuclideanAlgorithm}%
299     \expandafter\xint_gobble_iii
300   \fi
301   \XINT_TypesetEuclideanAlgorithm
302 }%
303 \def\XINT_TypesetEuclideanAlgorithm #1#2%
304 {% l'algo remplace #1 et #2 par |#1| et |#2|
305   \par
306   \begingroup
307     \xintAssignArray\xintEuclideanAlgorithm {#1}{#2}\to\U
308     \edef\A{\U2}\edef\B{\U4}\edef\N{\U1}%
309     \setbox 0 \vbox{\halign {$##$\cr \A\cr \B \cr}}%
310     \count 255 1
311     \xintloop
312       \indent\hbox to \wd 0 {\hfil$\U{\numexpr 2*\count255\relax}$}%
313       ${} = \U{\numexpr 2*\count255 + 3\relax}
314       \times \U{\numexpr 2*\count255 + 2\relax}
315       + \U{\numexpr 2*\count255 + 4\relax}$%
316     \ifnum \count255 < \N
317       \par
318       \advance \count255 1
319     \repeat
320   \endgroup
321 }%

```

## 23.7. \xintTypesetBezoutAlgorithm

Pour Bezout on a:  $\{N\}\{A\}\{0\}\{1\}\{D=r(n)\}\{B\}\{1\}\{0\}\{q1\}\{r1\}\{\alpha1=q1\}\{\beta1=1\}$   
 $\{q2\}\{r2\}\{\alpha2\}\{\beta2\}...\{qN\}\{rN=0\}\{\alphaN=A/D\}\{\betaN=B/D\}$

Donc  $4N+8$  termes:  $U1 = N$ ,  $U2= A$ ,  $U5=D$ ,  $U6=B$ ,  $q1 = U9$ ,  $qn = U\{4n+5\}$ , n au moins 1  
 $rn = U\{4n+6\}$ , n au moins -1

$\alpha(n) = U\{4n+7\}$ ,  $n$  au moins  $-1$

$\beta(n) = U\{4n+8\}$ ,  $n$  au moins  $-1$

1.09h uses `\xintloop`, and `\par` rather than `\endgraf`; and no more `\parindent0pt`

```

322 \def\xintTypesetBezoutAlgorithm {%
323   \unless\ifdefined\xintAssignArray
324     \errmessage
325       {xintgcd: package xinttools is required for \string\xintTypesetBezoutAlgorithm}%
326     \expandafter\xint_gobble_iii
327   \fi
328   \XINT_TypesetBezoutAlgorithm
329 }%
330 \def\XINT_TypesetBezoutAlgorithm #1#2%
331 {%
332   \par
333   \begingroup
334     \xintAssignArray\xintBezoutAlgorithm {#1}{#2}\to\BEZ
335     \edef\A{\BEZ2}\edef\B{\BEZ6}\edef\N{\BEZ1}% A = |#1|, B = |#2|
336     \setbox 0 \vbox{\halign {##$\cr \A\cr \B \cr}}%
337     \count255 1
338     \xintloop
339       \indent\hbox to \wd 0 {\hfil$\BEZ{4*\count255 - 2}$}%
340       ${} = \BEZ{4*\count255 + 5}
341       \times \BEZ{4*\count255 + 2}
342       + \BEZ{4*\count255 + 6}$\hfill\break
343       \hbox to \wd 0 {\hfil$\BEZ{4*\count255 + 7}$}%
344       ${} = \BEZ{4*\count255 + 5}
345       \times \BEZ{4*\count255 + 3}
346       + \BEZ{4*\count255 - 1}$\hfill\break
347       \hbox to \wd 0 {\hfil$\BEZ{4*\count255 + 8}$}%
348       ${} = \BEZ{4*\count255 + 5}
349       \times \BEZ{4*\count255 + 4}
350       + \BEZ{4*\count255 }$
351     \par
352     \ifnum \count255 < \N
353       \advance \count255 1
354     \repeat
355     \edef\U{\BEZ{4*\N + 4}}%
356     \edef\V{\BEZ{4*\N + 3}}%
357     \edef\D{\BEZ5}%
358     \ifodd\N
359       $\U\times\A - \V\times \B = -\D$%
360     \else
361       $\U\times\A - \V\times\B = \D$%
362     \fi
363   \par
364 \endgroup
365 }%
366 \XINTrestorecatcodesendinginput%
```

## 24. Package **xintfrac** implementation

.1	Catcodes, $\varepsilon$ -TeX and reload detection . . .	431	.46	<code>\xintPow</code> . . . . .	465
.2	Package identification . . . . .	432	.47	<code>\xintFac</code> . . . . .	466
.3	<code>\XINT_cntSgnFork</code> . . . . .	432	.48	<code>\xintBinomial</code> . . . . .	466
.4	<code>\xintLen</code> . . . . .	432	.49	<code>\xintPFactorial</code> . . . . .	466
.5	<code>\XINT_outfrac</code> . . . . .	432	.50	<code>\xintPrd</code> . . . . .	467
.6	<code>\XINT_infrac</code> . . . . .	433	.51	<code>\xintDiv</code> . . . . .	467
.7	<code>\XINT_frac_gen</code> . . . . .	435	.52	<code>\xintDivFloor</code> . . . . .	467
.8	<code>\XINT_factortens</code> . . . . .	437	.53	<code>\xintDivTrunc</code> . . . . .	468
.9	<code>\xintEq</code> , <code>\xintNotEq</code> , <code>\xintGt</code> , <code>\xintLt</code> , <code>\xintGtorEq</code> , <code>\xintLtorEq</code> , <code>\xintIsZero</code> , <code>\xintIsNotZero</code> , <code>\xintOdd</code> , <code>\xintEven</code> , <code>\xintifSgn</code> , <code>\xintifCmp</code> , <code>\xintifEq</code> , <code>\xintifGt</code> , <code>\xintifLt</code> , <code>\xintifZero</code> , <code>\xintifNotZero</code> , <code>\xintifOne</code> , <code>\xintifOdd</code> . . . . .	438	.54	<code>\xintDivRound</code> . . . . .	468
.10	<code>\xintRaw</code> . . . . .	440	.55	<code>\xintModTrunc</code> . . . . .	468
.11	<code>\xintRawBraced</code> . . . . .	440	.56	<code>\xintDivMod</code> . . . . .	469
.12	<code>\xintiLogTen</code> . . . . .	440	.57	<code>\xintMod</code> . . . . .	470
.13	<code>\xintPRaw</code> . . . . .	441	.58	<code>\xintIsOne</code> . . . . .	471
.14	<code>\xintSPRaw</code> . . . . .	442	.59	<code>\xintGeq</code> . . . . .	471
.15	<code>\xintFracToSci</code> . . . . .	442	.60	<code>\xintMax</code> . . . . .	472
.16	<code>\xintFracToDecimal</code> . . . . .	442	.61	<code>\xintMaxof</code> . . . . .	473
.17	<code>\xintRawWithZeros</code> . . . . .	442	.62	<code>\xintMin</code> . . . . .	473
.18	<code>\xintDecToString</code> . . . . .	443	.63	<code>\xintMinof</code> . . . . .	474
.19	<code>\xintDecToStringREZ</code> . . . . .	443	.64	<code>\xintCmp</code> . . . . .	474
.20	<code>\xintFloor</code> , <code>\xintiFloor</code> . . . . .	443	.65	<code>\xintAbs</code> . . . . .	476
.21	<code>\xintCeil</code> , <code>\xintiCeil</code> . . . . .	444	.66	<code>\xintOpp</code> . . . . .	476
.22	<code>\xintNumerator</code> . . . . .	444	.67	<code>\xintInv</code> . . . . .	476
.23	<code>\xintDenominator</code> . . . . .	444	.68	<code>\xintSgn</code> . . . . .	477
.24	<code>\xintTeXFrac</code> . . . . .	445	.69	<code>\xintSignBit</code> . . . . .	477
.25	<code>\xintTeXsignedFrac</code> . . . . .	446	.70	<code>\xintGCD</code> . . . . .	477
.26	<code>\xintTeXFromSci</code> . . . . .	446	.71	<code>\xintGCDof</code> . . . . .	478
.27	<code>\xintTeXOver</code> . . . . .	447	.72	<code>\xintLCM</code> . . . . .	479
.28	<code>\xintTeXsignedOver</code> . . . . .	448	.73	<code>\xintLCMof</code> . . . . .	480
.29	<code>\xintREZ</code> . . . . .	448	.74	Floating point macros . . . . .	481
.30	<code>\xintE</code> . . . . .	449	.75	<code>\xintDigits</code> , <code>\xintSetDigits</code> . . . . .	483
.31	<code>\xintIrr</code> , <code>\xintPIrr</code> . . . . .	449	.76	<code>\xintFloat</code> , <code>\xintFloatZero</code> . . . . .	483
.32	<code>\xintifInt</code> . . . . .	451	.77	<code>\xintFloatBraced</code> . . . . .	485
.33	<code>\xintIsInt</code> . . . . .	451	.78	<code>\XINTinFloat</code> , <code>\XINTinFloatS</code> . . . . .	486
.34	<code>\xintJrr</code> . . . . .	451	.79	<code>\XINTFloatiLogTen</code> . . . . .	491
.35	<code>\xintTFrac</code> . . . . .	453	.80	<code>\xintPFloat</code> . . . . .	492
.36	<code>\xintTrunc</code> , <code>\xintiTrunc</code> . . . . .	453	.81	<code>\xintFloatToDecimal</code> . . . . .	496
.37	<code>\xintTTrunc</code> . . . . .	456	.82	<code>\XINTinFloatFrac</code> . . . . .	497
.38	<code>\xintNum</code> , <code>\xintnum</code> . . . . .	456	.83	<code>\xintFloatAdd</code> , <code>\XINTinFloatAdd</code> . . . . .	497
.39	<code>\xintRound</code> , <code>\xintiRound</code> . . . . .	456	.84	<code>\xintFloatSub</code> , <code>\XINTinFloatSub</code> . . . . .	498
.40	<code>\xintXTrunc</code> . . . . .	457	.85	<code>\xintFloatMul</code> , <code>\XINTinFloatMul</code> . . . . .	499
.41	<code>\xintAdd</code> . . . . .	462	.86	<code>\xintFloatSqr</code> , <code>\XINTinFloatSqr</code> . . . . .	499
.42	<code>\xintSub</code> . . . . .	464	.87	<code>\XINTinFloatInv</code> . . . . .	500
.43	<code>\xintSum</code> . . . . .	464	.88	<code>\xintFloatDiv</code> , <code>\XINTinFloatDiv</code> . . . . .	500
.44	<code>\xintMul</code> . . . . .	464	.89	<code>\xintFloatPow</code> , <code>\XINTinFloatPow</code> . . . . .	501
.45	<code>\xintSqr</code> . . . . .	465	.90	<code>\xintFloatPower</code> , <code>\XINTinFloatPower</code> . . . . .	505
			.91	<code>\xintFloatFac</code> , <code>\XINTFloatFac</code> . . . . .	508
			.92	<code>\xintFloatPFactorial</code> , <code>\XINTinFloatPFactorial</code> . . . . .	513
			.93	<code>\xintFloatBinomial</code> , <code>\XINTinFloatBinomial</code> . . . . .	517

.94	\xintFloatSqrt, \XINTinFloatSqrt . .	518	.100	\xintFloatIsInt . . . . .	522
.95	\xintFloatE, \XINTinFloatE . . . . .	520	.101	\xintFloatIntType . . . . .	522
.96	\XINTinFloatMod . . . . .	521	.102	\XINTinFloatdigits, \XINTinFloatSdigits	523
.97	\XINTinFloatDivFloor . . . . .	521	.103	(WIP) \XINTinRandomFloatS, \XINTinRan-	
.98	\XINTinFloatDivMod . . . . .	522		domFloatSdigits . . . . .	523
.99	\xintifFloatInt . . . . .	522	.104	(WIP) \XINTinRandomFloatSixteen . .	524

The commenting is currently (2025/09/06) very sparse.

## 24.1. Catcodes, $\varepsilon$ -T<sub>E</sub>X and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^
11 \def\empty{}\def\space{ }\newlinechar10
12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintfrac.sty\endcsname
14 \expandafter\let\expandafter\w\csname ver@xint.sty\endcsname
15 \expandafter\ifx\csname numexpr\endcsname\relax
16 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17 \immediate\write128{^^JPackage xintfrac Warning:^^J%
18 \space\space\space\space
19 \numexpr not available, aborting input.^^J}%
20 \else
21 \PackageWarningNoLine{xintfrac}{\numexpr not available, aborting input}%
22 \fi
23 \def\z{\endgroup\endinput}%
24 \else
25 \ifx\x\relax % plain-TeX, first loading of xintfrac.sty
26 \ifx\w\relax % but xint.sty not yet loaded.
27 \def\z{\endgroup\input xint.sty\relax}%
28 \fi
29 \else
30 \ifx\x\empty % LaTeX, first loading,
31 % variable is initialized, but \ProvidesPackage not yet seen
32 \ifx\w\relax % xint.sty not yet loaded.
33 \def\z{\endgroup\RequirePackage{xint}}%
34 \fi
35 \else
36 \def\z{\endgroup\endinput}% xintfrac already loaded.
37 \fi
38 \fi
39 \fi
40 \z%
```



```
41 \XINTsetupcatcodes% defined in xintkernel.sty
```

## 24.2. Package identification

```
42 \XINT_providespackage
43 \ProvidesPackage{xintfrac}%
44 [2025/09/06 v1.4o Expandable operations on fractions (JFB)]%
```

## 24.3. \XINT\_cntSgnFork

1.09i. Used internally, #1 must expand to `\m@ne`, `\z@`, or `\@ne` or equivalent. `\XINT_cntSgnFork` does not insert a romannumeral stopper.

```
45 \def\XINT_cntSgnFork #1%
46 {%
47   \ifcase #1\expandafter\xint_secondofthree
48     \or\expandafter\xint_thirdofthree
49     \else\expandafter\xint_firstofthree
50   \fi
51 }%
```

## 24.4. \xintLen

The used formula is disputable, the idea is that  $A/1$  and  $A$  should have same length. Venerable code rewritten for 1.2i, following updates to `\xintLength` in `xintkernel.sty`. And sadly, I forgot on this occasion that this macro is not supposed to count the sign... Fixed in 1.2k.

```
52 \def\xintLen {\romannumeral0\xintlen }%
53 \def\xintlen #1%
54 {%
55   \expandafter\XINT_flen\romannumeral0\XINT_infrac {#1}%
56 }%
57 \def\XINT_flen#1{\def\XINT_flen ##1##2##3%
58 {%
59   \expandafter#1%
60   \the\numexpr \XINT_abs##1+%
61   \XINT_len_fork ##2##3\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
62   \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
63   \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye-\xint_c_i
64   \relax
65 }}\XINT_flen{ }%
```

## 24.5. \XINT\_outfrac

**Modified at 1.06b (2013/05/14).** 1.06b version now outputs  $0/1[0]$  and not  $0[0]$  in case of zero.

More generally all macros have been checked in `xintfrac`, `xintseries`, `xintcfrac`, to make sure the output format for fractions was always  $A/B[n]$ . (except `\xintIrr`, `\xintJrr`, `\xintRawWithZeros`).

Months later (2014/10/22): perhaps I should document what this macro does before I forget? from  $\{e\}\{N\}\{D\}$  it outputs  $N/D[e]$ , checking in passing if  $D=0$  or if  $N=0$ . It also makes sure  $D$  is not  $< 0$ . I am not sure but I don't think there is any place in the code which could call `\XINT_outfrac` with a  $D < 0$ , but I should check.

```
66 \def\XINT_outfrac #1#2#3%
67 {%
68   \ifcase\XINT_cntSgn #3\xint:
```

```

69      \expandafter \XINT_outfrac_divisionbyzero
70  \or
71      \expandafter \XINT_outfrac_P
72  \else
73      \expandafter \XINT_outfrac_N
74  \fi
75  {#2}{#3}[#1]%
76 }%
77 \def\XINT_outfrac_divisionbyzero #1#2[#3]%
78 {%
79     \XINT_signalcondition{DivisionByZero}{Division by zero: #1/#2.}{0/1[0]}%
80 }%
81 \def\XINT_outfrac_P#1{%
82 \def\XINT_outfrac_P ##1##2%
83     {\if0\XINT_Sgn ##1\xint:\expandafter\XINT_outfrac_Zero\fi##1/##2}%
84 }\XINT_outfrac_P{ }%
85 \def\XINT_outfrac_Zero #1[#2]{ 0/1[0]}%
86 \def\XINT_outfrac_N #1#2%
87 {%
88     \expandafter\XINT_outfrac_N_a\expandafter
89     {\romannumeral0\XINT_opp #2}{\romannumeral0\XINT_opp #1}%
90 }%
91 \def\XINT_outfrac_N_a #1#2%
92 {%
93     \expandafter\XINT_outfrac_P\expandafter {#2}{#1}%
94 }%

```

## 24.6. \XINT\_infrac

**Added at 1.03 (2013/04/14).** Parses fraction, scientific notation, etc... and produces  $\{n\}{A}{B}$  corresponding to  $A/B \times 10^n$ . No reduction to smallest terms.

**Modified at 1.07 (2013/05/25).** Extended in 1.07 to accept scientific notation on input. With lowercase e only. The `\xintexpr` parser does accept uppercase E also. Ah, by the way, perhaps I should at least say what this macro does? (belated addition 2014/10/22...), before I forget! It prepares the fraction in the internal format  $\{\text{exponent}\}\{\text{Numerator}\}\{\text{Denominator}\}$  where Denominator is at least 1.

**Modified at 1.2 (2015/10/10).** This venerable macro from the very early days has gotten a lifting for release 1.2. There were two kinds of issues:

- 1) use of `\W`, `\Z`, `\T` delimiters was very poor choice as this could clash with user input,
- 2) the new `\XINT_frac_gen` handles macros (possibly empty) in the input as general as `\A.\Be\C/\_D.\Ee\F`. The earlier version would not have expanded the `\B` or `\E`: digits after decimal mark were constrained to arise from expansion of the first token. Thus the 1.03 original code would have expanded only `\A`, `\D`, `\C`, and `\F` for this input.

This reminded me think I should revisit the remaining earlier portions of code, as I was still learning TeX coding when I wrote them.

Also I thought about parsing even faster the  $A/B[N]$  input, not expanding B, but this turned out to clash with some established uses in the documentation such as `1/\xintiisqr{...}[0]`. For the implementation, careful here about potential brace removals with parameter patterns such as like `#1/#2#3[#4]` for example.

While I was at it 1.2 added `\numexpr` parsing of the N, which earlier was restricted to be only explicit digits. I allowed `[]` with empty N, but the way I did it in 1.2 with `\the\numexpr 0#1` was buggy, as it did not allow #1 to be a `\count` for example or itself a `\numexpr` (although such inputs

were not previously allowed, I later turned out to use them in the code itself, e.g. the float factorial of version 1.2f). The better way would be `\the\numexpr#1+\xint_c_` but 1.2f finally does only `\the\numexpr #1` and #1 is not allowed to be empty.

The 1.2 `\XINT_frac_gen` had two locations with such a problematic `\numexpr 0#1` which I replaced for 1.2f with `\numexpr#1+\xint_c_`.

Regarding calling the macro with an argument `A[<expression>]`, a / in the expression must be suitably hidden for example in `\firstofone` type constructs.

Note: when the numerator is found to be zero `\XINT_infrac` \*always\* returns `{0}{0}{1}`. This behaviour must not change because 1.2g `\xintFloat` and `XINTinFloat` (for example) rely upon it: if the denominator on output is not 1, then `\xintFloat` assumes that the numerator is not zero.

As described in the manual, if the input contains a (final) `[N]` part, it is assumed that it is in the shape `A[N]` or `A/B[N]` with `A` (and `B`) not containing neither decimal mark nor scientific part, moreover `B` must be positive and `A` have at most one minus sign (and no plus sign). Else there will be errors, for example `-0/2[0]` would not be recognized as being zero at this stage and this could cause issues afterwards. When there is no ending `[N]` part, both numerator and denominator will be parsed for the more general format allowing decimal digits and scientific part and possibly multiple leading signs.

**Modified at 1.2l (2017/07/26).** 1.2l fixes frailty of `\XINT_infrac` (hence basically of all `xintfrac` macros) respective to non terminated `\numexpr` input: `\xintRaw{\the\numexpr1}` for example.

The issue was that `\numexpr` sees the / and expands what's next. But even `\numexpr 1//` for example creates an error, and to my mind this is a defect of `\numexpr`. It should be able to trace back and see that / was used as delimiter not as operator. Anyway, I thus fixed this problem belatedly here regarding `\XINT_infrac`.

**Modified at 1.4l (2022/05/29).** Deprecate venerable `\XINT_inFrac`, whose new name is `\xintRawBraced`. Should have been removed then.

**Modified at 1.4n (2025/09/05).** Remove `\XINT_inFrac`.

```

95 \def\xint_infrac #1% the core xintfrac in-parser; triggered by \romannumeral0
96 {%
97   \expandafter\xint_infrac_fork\romannumeral`&&0#1\xint:/\XINT_W[\XINT_W\XINT_T
98 }%
99 \def\xint_infrac_fork #1[#2%
100 {%
101   \xint_UDXINTWfork
102   #2\xint_frac_gen          % input has no brackets [N]
103   \XINT_W\XINT_infrac_res_a % there is some [N], must be strict A[N] or A/B[N] input
104   \krof
105   #1[#2%
106 }%
107 \def\xint_infrac_res_a #1%
108 {%
109   \xint_gob_til_zero #1\XINT_infrac_res_zero 0\XINT_infrac_res_b #1%
110 }%
111 \def\xint_infrac_res_zero 0\XINT_infrac_res_b #1\XINT_T {{0}{0}{1}}%
112 \def\xint_infrac_res_b #1/#2%
113 {%
114   \xint_UDXINTWfork
115   #2\xint_infrac_res_ca      % it was A[N] input
116   \XINT_W\XINT_infrac_res_cb % it was A/B[N] input
117   \krof
118   #1/#2%

```

Note that input exponent is here ignored and forced to be zero.

119 }%

An empty [] is not allowed. (this was authorized in 1.2, removed in 1.2f).

```
120 \def\XINT_infrac_res_ca #1[#2]\xint:/\XINT_W[\XINT_W\XINT_T
121   {\expandafter{\the\numexpr #2}{#1}{1}}%
122 \def\XINT_infrac_res_cb #1/#2[%
123   {\expandafter\XINT_infrac_res_cc\romannumeral`&&@#2~#1[%
124 \def\XINT_infrac_res_cc #1~#2[#3]\xint:/\XINT_W[\XINT_W\XINT_T
125   {\expandafter{\the\numexpr #3}{#2}{#1}}%
```

## 24.7. \XINT\_frac\_gen

**Modified at 1.07 (2013/05/25).** Extended at to recognize and accept scientific notation both at the numerator and (possible) denominator. Only a lowercase e will do here, but uppercase E is possible within an `\xintexpr`..`\relax`

**Modified at 1.2 (2015/10/10).** Completely rewritten. The parsing handles inputs such as `\A.\B_e\C/\D.\Ee\F` where each of `\A`, `\B`, `\D`, and `\E` may need f-expansion and `\C` and `\F` will end up in `\numexpr`.

**Modified at 1.2f (2016/03/12).** 1.2f corrects an issue to allow `\C` and `\F` to be `\count` variable (or expressions with `\numexpr`): 1.2 did a bad `\numexpr0#1` which allowed only explicit digits for expanded #1.

```
126 \def\XINT_frac_gen #1/#2%
127 {%
128   \xint_UDXINTWfork
129   #2\XINT_frac_gen_A      % there was no /
130   \XINT_W\XINT_frac_gen_B % there was a /
131   \krof
132   #1/#2%
133 }%
```

Note that #1 is only expanded so far up to decimal mark or "e".

```
134 \def\XINT_frac_gen_A #1\xint:/\XINT_W [\XINT_W {\XINT_frac_gen_C 0~1!#1ee.\XINT_W }%
135 \def\XINT_frac_gen_B #1/#2\xint:/\XINT_W[%\XINT_W
136 {%
137   \expandafter\XINT_frac_gen_Ba
138   \romannumeral`&&@#2ee.\XINT_W\XINT_Z #1ee.%\XINT_W
139 }%
140 \def\XINT_frac_gen_Ba #1.#2%
141 {%
142   \xint_UDXINTWfork
143   #2\XINT_frac_gen_Bb
144   \XINT_W\XINT_frac_gen_Bc
145   \krof
146   #1.#2%
147 }%
148 \def\XINT_frac_gen_Bb #1e#2e#3\XINT_Z
149   {\expandafter\XINT_frac_gen_C\the\numexpr #2+\xint_c_~#1!}%
150 \def\XINT_frac_gen_Bc #1.#2e%
151 {%
152   \expandafter\XINT_frac_gen_Bd\romannumeral`&&@#2.#1e%
153 }%
154 \def\XINT_frac_gen_Bd #1.#2e#3e#4\XINT_Z
155 {%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

156 \expandafter\XINT_frac_gen_C\the\numexpr #3-%
157 \numexpr\XINT_length_loop
158 #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
159 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
160 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
161 ~#2#1!%
162 }%
163 \def\XINT_frac_gen_C #1!#2.#3%
164 {%
165 \xint_UDXINTWfork
166 #3\XINT_frac_gen_Ca
167 \XINT_W\XINT_frac_gen_Cb
168 \krof
169 #1!#2.#3%
170 }%
171 \def\XINT_frac_gen_Ca #1~#2!#3e#4e#5\XINT_T
172 {%
173 \expandafter\XINT_frac_gen_F\the\numexpr #4-#1\expandafter
174 ~\romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
175 #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z~#3~%
176 }%
177 \def\XINT_frac_gen_Cb #1.#2e%
178 {%
179 \expandafter\XINT_frac_gen_Cc\romannumeral`&&@#2.#1e%
180 }%
181 \def\XINT_frac_gen_Cc #1.#2~#3!#4e#5e#6\XINT_T
182 {%
183 \expandafter\XINT_frac_gen_F\the\numexpr #5-#2-%
184 \numexpr\XINT_length_loop
185 #1\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
186 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
187 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
188 \relax\expandafter~%
189 \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
190 #3\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z
191 ~#4#1~%
192 }%
193 \def\XINT_frac_gen_F #1~#2%
194 {%
195 \xint_UDzerominusfork
196 #2-\XINT_frac_gen_Gdivbyzero
197 0#2{\XINT_frac_gen_G -{}}%
198 0-{\XINT_frac_gen_G {}#2}%
199 \krof #1~%
200 }%
201 \def\XINT_frac_gen_Gdivbyzero #1~#2~%
202 {%
203 \expandafter\XINT_frac_gen_Gdivbyzero_a
204 \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
205 #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:\Z~#1~%
206 }%
207 \def\XINT_frac_gen_Gdivbyzero_a #1~#2~%

```

```

208 {%
209     \XINT_signalcondition{DivisionByZero}{Division by zero: #1/0.}{\{#2\}{#1\}{0}}%
210 }%
211 \def\XINT_frac_gen_G #1#2#3~#4~#5~%
212 {%
213     \expandafter\XINT_frac_gen_Ga
214     \romannumeral0\expandafter\XINT_num_cleanup\the\numexpr\XINT_num_loop
215     #1#5\XINT:\XINT:\XINT:\XINT:\XINT:\XINT:\XINT:\XINT:\XINT:\Z~#3~{#2#4}%
216 }%
217 \def\XINT_frac_gen_Ga #1#2~#3~%
218 {%
219     \XINT_gob_til_zero #1\XINT_frac_gen_zero 0%
220     {#3}{#1#2}%
221 }%
222 \def\XINT_frac_gen_zero 0#1#2#3{\{0\}{0\}{1}}%

```

## 24.8. \XINT\_factortens

This is the core macro for `\xintREZ`. To be used as `\romannumeral0\XINT_factortens{...}`. Output is A.N. (formerly {A}{N}) where A is the integer stripped from trailing zeroes and N is the number of removed zeroes. Only for positive strict integers!

**Modified at 1.3a (2018/03/07).** Completely rewritten at 1.3a to replace a double `\xintReverseOrder` by a direct `\numexpr` governed expansion to the end and back, à la 1.2. I should comment more... and perhaps improve again in future.

Testing shows significant gain at 100 digits or more.

```

223 \def\XINT_factortens #1{\expandafter\XINT_factortens_z
224     \romannumeral0\XINT_factortens_a#1%
225     \XINT_factortens_b123456789.}%
226 \def\XINT_factortens_z.\XINT_factortens_y{ }%
227 \def\XINT_factortens_a #1#2#3#4#5#6#7#8#9%
228     {\expandafter\XINT_factortens_x
229     \the\numexpr 1#1#2#3#4#5#6#7#8#9\XINT_factortens_a}%
230 \def\XINT_factortens_b#1\XINT_factortens_a#2#3.%
231     {.\XINT_factortens_cc 000000000-#2.}%
232 \def\XINT_factortens_x1#1.#2{#2#1}%
233 \def\XINT_factortens_y{.\XINT_factortens_y}%
234 \def\XINT_factortens_cc #1#2#3#4#5#6#7#8#9%
235     {\if#90\xint_dothis
236     {\expandafter\XINT_factortens_d\the\numexpr #8#7#6#5#4#3#2#1\relax
237     \xint_c_i 2345678.}\fi
238     \xint_orthat{\XINT_factortens_yy{#1#2#3#4#5#6#7#8#9}}}%
239 \def\XINT_factortens_yy #1#2.{.\XINT_factortens_y#1.0.}%
240 \def\XINT_factortens_c #1#2#3#4#5#6#7#8#9%
241     {\if#90\xint_dothis
242     {\expandafter\XINT_factortens_d\the\numexpr #8#7#6#5#4#3#2#1\relax
243     \xint_c_i 2345678.}\fi
244     \xint_orthat{.\XINT_factortens_y #1#2#3#4#5#6#7#8#9.}}%
245 \def\XINT_factortens_d #1#2#3#4#5#6#7#8#9%
246     {\if#10\expandafter\XINT_factortens_e\fi
247     \XINT_factortens_f #9#9#8#7#6#5#4#3#2#1.}%
248 \def\XINT_factortens_f #1#2\xint_c_i#3.#4.#5.%
249     {\expandafter\XINT_factortens_g\the\numexpr#1+#5.#3.}%

```

```

250 \def\XINT_factortens_g #1.#2.{.\XINT_factortens_y#2.#1.}%
251 \def\XINT_factortens_e #1..#2.%
252   {\expandafter.\expandafter\XINT_factortens_c
253     \the\numexpr\xint_c_ix+#2.}%

```

## 24.9. `\xintEq`, `\xintNotEq`, `\xintGt`, `\xintLt`, `\xintGtorEq`, `\xintLtorEq`, `\xintIsZero`, `\xintIsNotZero`, `\xintOdd`, `\xintEven`, `\xintifSgn`, `\xintifCmp`, `\xintifEq`, `\xintifGt`, `\xintifLt`, `\xintifZero`, `\xintifNotZero`, `\xintifOne`, `\xintifOdd`

Moved here at 1.3. Formerly these macros were already defined in `xint.sty` or even `xintcore.sty`. They are slim wrappers of macros defined elsewhere in `xintfrac`.

```

254 \def\xintEq    {\romannumeral0\xinteq }%
255 \def\xinteq    #1#2{\xintifeq{#1}{#2}{1}{0}}%
256 \def\xintNotEq#1#2{\romannumeral0\xintifeq {#1}{#2}{0}{1}}%
257 \def\xintGt    {\romannumeral0\xintgt }%
258 \def\xintgt    #1#2{\xintifgt{#1}{#2}{1}{0}}%
259 \def\xintLt    {\romannumeral0\xintlt }%
260 \def\xintlt    #1#2{\xintiflt{#1}{#2}{1}{0}}%
261 \def\xintGtorEq #1#2{\romannumeral0\xintiflt {#1}{#2}{0}{1}}%
262 \def\xintLtorEq #1#2{\romannumeral0\xintifgt {#1}{#2}{0}{1}}%
263 \def\xintIsZero {\romannumeral0\xintiszero }%
264 \def\xintiszero #1{\if0\xintSgn{#1}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%
265 \def\xintIsNotZero{\romannumeral0\xintisnotzero }%
266 \def\xintisnotzero
267   #1{\if0\xintSgn{#1}\xint_afterfi{ 0}\else\xint_afterfi{ 1}\fi}%
268 \def\xintOdd    {\romannumeral0\xintodd }%
269 \def\xintodd    #1%
270 {%
271   \ifodd\xintLDg{\xintNum{#1}} %<- intentional space
272   \xint_afterfi{ 1}%
273   \else
274     \xint_afterfi{ 0}%
275   \fi
276 }%
277 \def\xintEven    {\romannumeral0\xinteven }%
278 \def\xinteven    #1%
279 {%
280   \ifodd\xintLDg{\xintNum{#1}} %<- intentional space
281   \xint_afterfi{ 0}%
282   \else
283     \xint_afterfi{ 1}%
284   \fi
285 }%
286 \def\xintifSgn{\romannumeral0\xintifsgn }%
287 \def\xintifsgn #1%
288 {%
289   \ifcase \xintSgn{#1}
290     \expandafter\xint_stop_atsecondofthree
291     \or\expandafter\xint_stop_atthirdofthree
292     \else\expandafter\xint_stop_atfirstofthree
293   \fi

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
294 }%
295 \def\xintifCmp{\romannumeral0\xintifcmp}%
296 \def\xintifcmp #1#2%
297 {%
298     \ifcase\xintCmp{#1}{#2}
299         \expandafter\xint_stop_atsecondofthree
300         \or\expandafter\xint_stop_atthirdofthree
301         \else\expandafter\xint_stop_atfirstofthree
302     \fi
303 }%
304 \def\xintifEq{\romannumeral0\xintifeq}%
305 \def\xintifeq #1#2%
306 {%
307     \if0\xintCmp{#1}{#2}%
308         \expandafter\xint_stop_atfirstoftwo
309         \else\expandafter\xint_stop_atsecondoftwo
310     \fi
311 }%
312 \def\xintifGt{\romannumeral0\xintifgt}%
313 \def\xintifgt #1#2%
314 {%
315     \if1\xintCmp{#1}{#2}%
316         \expandafter\xint_stop_atfirstoftwo
317         \else\expandafter\xint_stop_atsecondoftwo
318     \fi
319 }%
320 \def\xintifLt{\romannumeral0\xintiflt}%
321 \def\xintiflt #1#2%
322 {%
323     \ifnum\xintCmp{#1}{#2}<\xint_c_
324         \expandafter\xint_stop_atfirstoftwo
325     \else \expandafter\xint_stop_atsecondoftwo
326     \fi
327 }%
328 \def\xintifZero{\romannumeral0\xintifzero}%
329 \def\xintifzero #1%
330 {%
331     \if0\xintSgn{#1}%
332         \expandafter\xint_stop_atfirstoftwo
333     \else
334         \expandafter\xint_stop_atsecondoftwo
335     \fi
336 }%
337 \def\xintifNotZero{\romannumeral0\xintifnotzero}%
338 \def\xintifnotzero #1%
339 {%
340     \if0\xintSgn{#1}%
341         \expandafter\xint_stop_atsecondoftwo
342     \else
343         \expandafter\xint_stop_atfirstoftwo
344     \fi
345 }%
```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
346 \def\xintifOne {\romannumeral0\xintifone }%
347 \def\xintifone #1%
348 {%
349     \if1\xintIsOne{#1}%
350     \expandafter\xint_stop_atfirstoftwo
351     \else
352     \expandafter\xint_stop_atsecondoftwo
353     \fi
354 }%
355 \def\xintifOdd {\romannumeral0\xintifodd }%
356 \def\xintifodd #1%
357 {%
358     \if\xintOdd{#1}1%
359     \expandafter\xint_stop_atfirstoftwo
360     \else
361     \expandafter\xint_stop_atsecondoftwo
362     \fi
363 }%
```

### 24.10. \xintRaw

**Added at 1.07 (2013/05/25).** 1.07: this macro simply prints in a user readable form the fraction after its initial scanning. Useful when put inside braces in an *\xintexpr*, when the input is not yet in the A/B[n] form.

```
364 \def\xintRaw {\romannumeral0\xinraw }%
365 \def\xinraw
366 {%
367     \expandafter\XINT_raw\romannumeral0\XINT_infrac
368 }%
369 \def\XINT_raw #1#2#3{ #2/#3[#1]}%
```

### 24.11. \xintRawBraced

**Added at 1.41 (2022/05/29).** User level interface to core *\romannumeral0\XINT\_infrac*. Replaces *\XINT\_inFrac* which was defined but nowhere used by the xint packages.

```
370 \def\xintRawBraced {\romannumeral0\xinrawbraced }%
371 \let\xinrawbraced \XINT_infrac
```

### 24.12. \xintiLogTen

**Added at 1.3e (2019/04/05).** The exponent a, such that  $10^a \leq \text{abs}(x) < 10^{(a+1)}$ . No rounding done on x, handled as an exact fraction.

```
372 \def\xintiLogTen {\the\numexpr\xintilogten}%
373 \def\xintilogten
374 {%
375     \expandafter\XINT_ilogten\romannumeral0\xinraw
376 }%
377 \def\XINT_ilogten #1%
378 {%
379     \xint_UDzerominusfork
380     0#1\XINT_ilogten_p
381     #1-\XINT_ilogten_z
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
382      0-{\XINT_ilogten_p#1}%
383      \krof
384 }%
385 \def\XINT_ilogten_z #1[#2]{-7FFF8000\relax}%
386 \def\XINT_ilogten_p #1/#2[#3]%
387 {%
388      #3+\expandafter\XINT_ilogten_a
389      \the\numexpr\xintLength{#1}\expandafter.\the\numexpr\xintLength{#2}.#1.#2.%
390 }%
391 \def\XINT_ilogten_a #1.#2.%
392 {%
393      #1-#2\ifnum#1>#2
394      \expandafter\XINT_ilogten_aa
395      \else
396      \expandafter\XINT_ilogten_ab
397      \fi #1.#2.%
398 }%
399 \def\XINT_ilogten_aa #1.#2.#3.#4.%
400 {%
401      \xintiiiflt{#3}{\XINT_dsx_addzerosnofuss{#1-#2}#4;}{-1}{}\relax
402 }%
403 \def\XINT_ilogten_ab #1.#2.#3.#4.%
404 {%
405      \xintiiiflt{\XINT_dsx_addzerosnofuss{#2-#1}#3;}{#4}{-1}{}\relax
406 }%
```

### 24.13. \xintPraw

Added at 1.09b (2013/10/03).

```
407 \def\xintPraw {\romannumeral0\xintpraw }%
408 \def\xintpraw
409 {%
410      \expandafter\XINT_praw\romannumeral0\XINT_infrac
411 }%
412 \def\XINT_praw #1%
413 {%
414      \ifnum #1=\xint_c_ \expandafter\XINT_praw_a\fi \XINT_praw_A {#1}%
415 }%
416 \def\XINT_praw_A #1#2#3%
417 {%
418      \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
419      \else\expandafter\xint_secondoftwo
420      \fi { #2[#1]}{ #2/#3[#1]}%
421 }%
422 \def\XINT_praw_a\XINT_praw_A #1#2#3%
423 {%
424      \if\XINT_isOne{#3}1\expandafter\xint_firstoftwo
425      \else\expandafter\xint_secondoftwo
426      \fi { #2}{ #2/#3}%
427 }%
```

## 24.14. `\xintSPRaw`

This private macro was for internal usage by `\xinttheexpr`. It got moved here at 1.4 and is not used anymore by the package.

It checks if input has a [N] part, if yes uses `\xintPRaw`, else simply lets the input pass through as is.

```
428 \def\xintSPRaw    {\romannumeral0\xintspraw}%
429 \def\xintspraw   #1{\expandafter\XINT_spraw\romannumeral`&&@#1[\W]}%
430 \def\XINT_spraw  #1[#2#3]{\xint_gob_til_W #2\XINT_spraw_a\W\XINT_spraw_p #1[#2#3]}%
431 \def\XINT_spraw_a\W\XINT_spraw_p #1[\W]{ #1}%
432 \def\XINT_spraw_p #1[\W]{\xintpraw {#1}}%
```

## 24.15. `\xintFracToSci`

**Added at 1.41 (2022/05/29).** The macro with this name which was added here at 1.4 then had various changes and finally was moved to `xintexpr` at 1.4k is now called there `\xint_FracToSci_x` and is private. The present macro is public and behaves like the other `xintfrac` macros: f-expandable and accepts general input. Its output is exactly the same as `\xint_FracToSci_x` for same inputs, with the exception of the empty input which `\xintFracToSci` will output as 0 but `\xint_FracToSci_x` as empty. But the latter is not used by `\xinteval` for an empty leaf as it employs then `\xintexprEmptyItem`.

```
433 \def\xintFracToSci{\romannumeral0\xintfractosci}%
434 \def\xintfractosci#1{\expandafter\XINT_fractosci\romannumeral0\xintraw{#1}}%
435 \def\XINT_fractosci#1#2/#3[#4]{\expanded{ %
436   \ifnum#4=\xint_c_ #1#2\else
437     \romannumeral0\expandafter\XINT_pfloat_a_fork\romannumeral0\xintrez{#1#2[#4]}%
438   \fi
439   \if\XINT_isOne{#3}1\else\if#10\else/#3\fi\fi}%
440 }%
```

## 24.16. `\xintFracToDecimal`

**Added at 1.41 (2022/05/29).** The macro with this name which was added at 1.4k to `xintexpr` has been removed. The public variant here behaves like the other `xintfrac` macros: f-expandable and accepts general input.

```
441 \def\xintFracToDecimal{\romannumeral0\xintfractodecimal}%
442 \def\xintfractodecimal#1{\expandafter\XINT_fractodecimal\romannumeral0\xintraw{#1}}%
443 \def\XINT_fractodecimal #1#2/#3[#4]{\expanded{ %
444   \ifnum#4=\xint_c_ #1#2\else
445     \romannumeral0\expandafter\XINT_dectostr\romannumeral0\xintrez{#1#2[#4]}%
446   \fi
447   \if\XINT_isOne{#3}1\else\if#10\else/#3\fi\fi}%
448 }%
```

## 24.17. `\xintRawWithZeros`

This was called `\xintRaw` in versions earlier than 1.07

```
449 \def\xintRawWithZeros {\romannumeral0\xintrawwithzeros}%
450 \def\xintrawwithzeros
451 {%
452   \expandafter\XINT_rawz_fork\romannumeral0\XINT_infrac
453 }%
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
454 \def\XINT_rawz_fork #1%
455 {%
456   \ifnum#1<\xint_c_
457     \expandafter\XINT_rawz_Ba
458   \else
459     \expandafter\XINT_rawz_A
460   \fi
461   #1.%
462 }%
463 \def\XINT_rawz_A #1.#2#3{\XINT_dsx_addzeros{#1}#2;/#3}%
464 \def\XINT_rawz_Ba -#1.#2#3{\expandafter\XINT_rawz_Bb
465   \expandafter{\romannumeral0\XINT_dsx_addzeros{#1}#3;}{#2}}%
466 \def\XINT_rawz_Bb #1#2{ #2/#1}%
```

### 24.18. \xintDecToString

Added at 1.3 (2018/03/01). This is a backport from poexpr 0.4. It is definitely not in final form, consider it to be an unstable macro.

```
467 \def\xintDecToString{\romannumeral0\xintdectostr}%
468 \def\xintdectostr#1{\expandafter\XINT_dectostr\romannumeral0\xintra{#1}}%
469 \def\XINT_dectostr #1/#2[#3]{\xintiiifZero {#1}%
470   \XINT_dectostr_z
471   {\if1\XINT_isOne{#2}\expandafter\XINT_dectostr_a
472     \else\expandafter\XINT_dectostr_b
473   \fi}%
474   #1/#2[#3]}%
475 }%
476 \def\XINT_dectostr_z#1[#2]{ 0}%
477 \def\XINT_dectostr_a#1/#2[#3]{%
478   \ifnum#3<\xint_c_\xint_dothis{\xinttrunc{-#3}{#1[#3]}}\fi
479   \xint_orthat{\xintiie{#1}{#3}}%
480 }%
481 \def\XINT_dectostr_b#1/#2[#3]{% just to handle this somehow
482   \ifnum#3<\xint_c_\xint_dothis{\xinttrunc{-#3}{#1[#3]}/#2}\fi
483   \xint_orthat{\xintiie{#1}{#3}/#2}%
484 }%
```

### 24.19. \xintDecToStringREZ

Added at 1.4e (2021/05/05). And I took this opportunity to improve documentation in manual.

```
485 \def\xintDecToStringREZ{\romannumeral0\xintdectostringrez}%
486 \def\xintdectostringrez#1{\expandafter\XINT_dectostr\romannumeral0\xintrez{#1}}%
```

### 24.20. \xintFloor, \xintiFloor

Added at 1.09a (2013/09/24). 1.1 for [\xintiFloor](#)/[\xintFloor](#). Not efficient if big negative decimal exponent. Also sub-efficient if big positive decimal exponent.

```
487 \def\xintFloor {\romannumeral0\xintfloor }%
488 \def\xintfloor #1% devrais-je faire \xintREZ?
489   {\expandafter\XINT_ifloor \romannumeral0\xintra{withzeros {#1}}./1[0]}%
490 \def\xintiFloor {\romannumeral0\xintifloor }%
491 \def\xintifloor #1%
```

```

492     {\expandafter\XINT_ifloor \romannumeral0\xintraewithzeros {#1}.}%
493 \def\XINT_ifloor #1/#2.{\xintiipro {#1}{#2}}%

```

## 24.21. \xintCeil, \xintiCeil

Added at 1.09a (2013/09/24).

```

494 \def\xintCeil {\romannumeral0\xintceil }%
495 \def\xintceil #1{\xintiio { \xintFloor { \xintOpp{#1} } } }%
496 \def\xintiCeil {\romannumeral0\xinticeil }%
497 \def\xinticeil #1{\xintiio { \xintiFloor { \xintOpp{#1} } } }%

```

## 24.22. \xintNumerator

```

498 \def\xintNumerator {\romannumeral0\xintnumerator }%
499 \def\xintnumerator
500 {%
501     \expandafter\XINT_numer\romannumeral0\XINT_infrac
502 }%
503 \def\XINT_numer #1%
504 {%
505     \ifcase\XINT_cntSgn #1\xint:
506     \expandafter\XINT_numer_B
507     \or
508     \expandafter\XINT_numer_A
509     \else
510     \expandafter\XINT_numer_B
511     \fi
512     {#1}%
513 }%
514 \def\XINT_numer_A #1#2#3{\XINT_dsx_addzeros{#1}{#2};}%
515 \def\XINT_numer_B #1#2#3{ #2}%

```

## 24.23. \xintDenominator

```

516 \def\xintDenominator {\romannumeral0\xintdenominator }%
517 \def\xintdenominator
518 {%
519     \expandafter\XINT_denom_fork\romannumeral0\XINT_infrac
520 }%
521 \def\XINT_denom_fork #1%
522 {%
523     \ifnum#1<\xint_c_
524     \expandafter\XINT_denom_B
525     \else
526     \expandafter\XINT_denom_A
527     \fi
528     #1.%
529 }%
530 \def\XINT_denom_A #1.#2#3{ #3}%
531 \def\XINT_denom_B -#1.#2#3{\XINT_dsx_addzeros{#1}{#3};}%

```

## 24.24. `\xintTeXFrac`

**Added at 1.03 (2013/04/14).** Useless typesetting macro.

**Modified at 1.4g (2021/05/25).** Renamed from `\xintFrac`.

**Modified at 1.4m (2022/06/10).** The old name now raises an error, not a warning.

```

532 \ifdefined\PackageWarning
533 \def\xintfracTeXDeprecation#1#2{%
534 \PackageWarning{xintfrac}{\string#1 is deprecated. Use \string#2\MessageBreak
535 to suppress this warning}#2%
536 }%
537 \else
538 \edef\xintfracTeXDeprecation#1#2{{\newlinechar10
539 \immediate\noexpand\write128{&&JPackage xintfrac Warning: \noexpand\string#1 is
540 deprecated. Use \noexpand\string#2&&J%
541 (xintfrac)\xintReplicate{16}{ }to suppress this warning
542 on input line \noexpand\the\inputlineno.&&J}}#2%
543 }%
544 \fi
545 \ifdefined\PackageError
546 \def\xintfracTeXError#1#2{%
547 \PackageError{xintfrac}{\string#1 has been removed.\MessageBreak
548 Use \string#2 to suppress this error}%
549 {I will fix it for now if you hit the `Return' key.}#2%
550 }%
551 \else
552 \edef\xintfracTeXError#1#2{{\newlinechar10
553 \errhelp{I will fix it for now if you hit the `Return' key.}%
554 \errmessage{Package xintfrac Error: \noexpand\string#1 has been removed.&&J%
555 (xintfrac)\xintReplicate{16}{ }Use \noexpand\string#2 to suppress this error}}#2%
556 }%
557 \fi
558 \def\xintFrac {\xintfracTeXError\xintFrac\xintTeXFrac}%
559 \def\xintTeXFrac{\romannumeral0\xintfrac }%
560 \def\xintfrac #1%
561 {%
562 \expandafter\XINT_fracfrac_A\romannumeral0\XINT_infrac {#1}%
563 }%
564 \def\XINT_fracfrac_A #1{\XINT_fracfrac_B #1\Z }%
565 \catcode\^=7
566 \def\XINT_fracfrac_B #1#2\Z
567 {%
568 \xint_gob_til_zero #1\XINT_fracfrac_C 0\XINT_fracfrac_D {10^{#1#2}}%
569 }%
570 \def\XINT_fracfrac_C 0\XINT_fracfrac_D #1#2#3%
571 {%
572 \if1\XINT_isOne {#3}%
573 \xint_afterfi {\expandafter\xint_stop_atfirstoftwo\xint_gobble_ii }%
574 \fi
575 \space
576 \frac {#2}{#3}%
577 }%
578 \def\XINT_fracfrac_D #1#2#3%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
579 {%
580   \if1\XINT_isOne {#3}\XINT_fracfrac_E\fi
581   \space
582   \frac {#2}{#3}#1%
583 }%
584 \def\XINT_fracfrac_E \fi\space\frac #1#2{\fi \space #1\cdot }%
```

### 24.25. \xintTeXsignedFrac

**Modified at 1.4g (2021/05/25).** Renamed from `\xintSignedFrac`.

**Modified at 1.4m (2022/06/10).** The old name now raises an error, not a warning.

```
585 \def\xintSignedFrac {\xintfracTeXError\xintSignedFrac\xintTeXsignedFrac}%
586 \def\xintTeXsignedFrac{\romannumeral0\xintsignedfrac }%
587 \def\xintsignedfrac #1%
588 {%
589   \expandafter\XINT_sgnfrac_a\romannumeral0\XINT_infrac {#1}%
590 }%
591 \def\XINT_sgnfrac_a #1#2%
592 {%
593   \XINT_sgnfrac_b #2\Z {#1}%
594 }%
595 \def\XINT_sgnfrac_b #1%
596 {%
597   \xint_UDsignfork
598   #1\XINT_sgnfrac_N
599   -{\XINT_sgnfrac_P #1}%
600   \krof
601 }%
602 \def\XINT_sgnfrac_P #1\Z #2%
603 {%
604   \XINT_fracfrac_A {#2}{#1}%
605 }%
606 \def\XINT_sgnfrac_N
607 {%
608   \expandafter-\romannumeral0\XINT_sgnfrac_P
609 }%
```

### 24.26. \xintTeXFromSci

**Added at 1.4g (2021/05/25).** The main problem is how to name this and related macros.

I use `\expanded` here, as `\xintFracToSci` is not f-expandable.

Some complications as I want this to be usable on output of `\xintFracToSci` hence need to handle the case of a /B. After some hesitations I ended with the following which looks reasonable:

- if no scientific part, use `\frac` (or `\over`) for A/B
- if scientific part, postfix /B as `\cdot B^{-1}`

**Modified at 1.4l (2022/05/29).** Suppress external `\expanded`. Keep internal one.

Rename `\xintTeXFromSci` from `\xintTeXfromSci`. Keep deprecated old name for the moment.

Add `\xintTeXFromScifracmacro`. Make it `\protected`.

Nota bene: catcode of `^` is normal one here (else nothing would work).

```
610 \def\xintTeXfromSci{\xintfracTeXDeprecated\xintTeXfromSci\xintTeXFromSci}%
611 \def\xintTeXFromSci#1%
```

```

612 {%
613   \expandafter\XINT_texfromsci\expanded{#1}/\relax/\xint:
614 }%
615 \def\XINT_texfromsci #1/#2#3/#4\xint:
616 {%
617   \XINT_texfromsci_a #1e\relax e\xint:
618   {\ifx\relax#2\xint_dothis\xint_firstofone\fi
619    \xint_orthat{\xintTeXFromScifracmacro{#2#3}}}%
620   {\unless\ifx\relax#2\cdot{#2#3}^{-1}\fi}%
621 }%
622 \def\XINT_texfromsci_a #1e#2#3e#4\xint:#5#6%
623 {%
624   \ifx\relax#2#5{#1}\else#1\cdot10^{#2#3}#6\fi
625 }%
626 \ifdefined\frac
627   \protected\def\xintTeXFromScifracmacro#1#2{\frac{#2}{#1}}%
628 \else
629   \protected\def\xintTeXFromScifracmacro#1#2{{#2\over#1}}%
630 \fi

```

## 24.27. \xintTeXOver

Modified at 1.4g (2021/05/25). Renamed from `\xintFwOver`.

Modified at 1.4m (2022/06/10). The old name now raises an error, not a warning.

```

631 \def\xintFwOver {\xintfracTeXError\xintFwOver\xintTeXOver}%
632 \def\xintTeXOver{\romannumeral0\xintfwover }%
633 \def\xintfwover #1%
634 {%
635   \expandafter\XINT_fwover_A\romannumeral0\XINT_infrac {#1}%
636 }%
637 \def\XINT_fwover_A #1{\XINT_fwover_B #1\Z }%
638 \def\XINT_fwover_B #1#2\Z
639 {%
640   \xint_gob_til_zero #1\XINT_fwover_C 0\XINT_fwover_D {10^{#1#2}}%
641 }%
642 \catcode\^ =11
643 \def\XINT_fwover_C #1#2#3#4#5%
644 {%
645   \if0\XINT_isOne {#5}\xint_afterfi { {#4\over #5}}%
646   \else\xint_afterfi { #4}%
647   \fi
648 }%
649 \def\XINT_fwover_D #1#2#3%
650 {%
651   \if0\XINT_isOne {#3}\xint_afterfi { {#2\over #3}}%
652   \else\xint_afterfi { #2\cdot }%
653   \fi
654   #1%
655 }%

```



## 24.28. `\xintTeXsignedOver`

Modified at 1.4g (2021/05/25). Renamed from `\xintSignedFwOver`.

Modified at 1.4m (2022/06/10). The old name now raises an error, not a warning.

```

656 \def\xintSignedFwOver {\xintfracTeXError\xintSignedFwOver\xintTeXsignedOver}%
657 \def\xintTeXsignedOver{\romannumeral0\xintsignedfwover }%
658 \def\xintsignedfwover #1%
659 {%
660   \expandafter\XINT_sgnfwover_a\romannumeral0\XINT_infrac {#1}%
661 }%
662 \def\XINT_sgnfwover_a #1#2%
663 {%
664   \XINT_sgnfwover_b #2\Z {#1}%
665 }%
666 \def\XINT_sgnfwover_b #1%
667 {%
668   \xint_UDsignfork
669     #1\XINT_sgnfwover_N
670     -{\XINT_sgnfwover_P #1}%
671   \krof
672 }%
673 \def\XINT_sgnfwover_P #1\Z #2%
674 {%
675   \XINT_fwover_A {#2}{#1}%
676 }%
677 \def\XINT_sgnfwover_N
678 {%
679   \expandafter-\romannumeral0\XINT_sgnfwover_P
680 }%
```

## 24.29. `\xintREZ`

Removes trailing zeros from A and B and adjust the N in A/B[N].

The macro really doing the job `\XINT_factortens` was redone at 1.3a. But speed gain really noticeable only beyond about 100 digits.

```

681 \def\xintREZ {\romannumeral0\xintrez }%
682 \def\xintrez
683 {%
684   \expandafter\XINT_rez_A\romannumeral0\XINT_infrac
685 }%
686 \def\XINT_rez_A #1#2%
687 {%
688   \XINT_rez_AB #2\Z {#1}%
689 }%
690 \def\XINT_rez_AB #1%
691 {%
692   \xint_UDzerominusfork
693     #1-\XINT_rez_zero
694     0#1\XINT_rez_neg
695     0-{\XINT_rez_B #1}%
696   \krof
697 }%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
698 \def\XINT_rez_zero #1\Z #2#3{ 0/1[0]}%
699 \def\XINT_rez_neg {\expandafter-\romannumeral0\XINT_rez_B }%
700 \def\XINT_rez_B #1\Z
701 {%
702   \expandafter\XINT_rez_C\romannumeral0\XINT_factortens {#1}%
703 }%
704 \def\XINT_rez_C #1.#2.#3#4%
705 {%
706   \expandafter\XINT_rez_D\romannumeral0\XINT_factortens {#4}#3+#2.#1.%
707 }%
708 \def\XINT_rez_D #1.#2.#3.%
709 {%
710   \expandafter\XINT_rez_E\the\numexpr #3-#2.#1.%
711 }%
712 \def\XINT_rez_E #1.#2.#3.{ #3/#2[#1]}%
```

### 24.30. \xintE

**Added at 1.07 (2013/05/25).** The fraction is the first argument contrarily to `\xintTrunc` and `\xintRound`.

**Modified at 1.1 (2014/10/28).** 1.1 modifies and moves `\xintiE` to `xint.sty`.

```
713 \def\xintE {\romannumeral0\xinte }%
714 \def\xinte #1%
715 {%
716   \expandafter\XINT_e \romannumeral0\XINT_infrac {#1}%
717 }%
718 \def\XINT_e #1#2#3#4%
719 {%
720   \expandafter\XINT_e_end\the\numexpr #1+#4.{#2}{#3}%
721 }%
722 \def\XINT_e_end #1.#2#3{ #2/#3[#1]}%
```

### 24.31. \xintIrr, \xintPIrr

**Modified at 1.04 (2013/04/25).** fixes a buggy `\xintIrr {0}`.

**Modified at 1.05 (2013/05/01).** modifies the initial parsing and post-processing to use `\xintraw` <sub>withzeros</sub> and to more quickly deal with an input denominator equal to 1.

**Modified at 1.08 (2013/06/07).** this version does not remove a /1 denominator.

**Modified at 1.3 (2018/03/01).** added `\xintPIrr` (partial Irr, which ignores the decimal part).

```
723 \def\xintIrr {\romannumeral0\xintirr }%
724 \def\xintPIrr{\romannumeral0\xintpirr }%
725 \def\xintirr #1%
726 {%
727   \expandafter\XINT_irr_start\romannumeral0\xinrawwithzeros {#1}\Z
728 }%
729 \def\xintpirr #1%
730 {%
731   \expandafter\XINT_pirr_start\romannumeral0\xinraw{#1}%
732 }%
733 \def\XINT_irr_start #1#2/#3\Z
734 {%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfraction, xintexpr, xinttrig, xintlog

```

735 \if0\XINT_isOne {#3}%
736 \xint_afterfi
737 {\xint_UDsignfork
738 #1\XINT_irr_negative
739 -{\XINT_irr_nonneg #1}%
740 \krof}%
741 \else
742 \xint_afterfi{\XINT_irr_denomisone #1}%
743 \fi
744 #2\Z {#3}%
745 }%
746 \def\XINT_pirr_start #1#2/#3[%
747 {%
748 \if0\XINT_isOne {#3}%
749 \xint_afterfi
750 {\xint_UDsignfork
751 #1\XINT_irr_negative
752 -{\XINT_irr_nonneg #1}%
753 \krof}%
754 \else
755 \xint_afterfi{\XINT_irr_denomisone #1}%
756 \fi
757 #2\Z {#3}[%
758 }%
759 \def\XINT_irr_denomisone #1\Z #2{ #1/1}% changed in 1.08
760 \def\XINT_irr_negative #1\Z #2{\XINT_irr_D #1\Z #2\Z -}%
761 \def\XINT_irr_nonneg #1\Z #2{\XINT_irr_D #1\Z #2\Z \space}%
762 \def\XINT_irr_D #1#2\Z #3#4\Z
763 {%
764 \xint_UDzerosfork
765 #3#1\XINT_irr_indeterminate
766 #30\XINT_irr_divisionbyzero
767 #10\XINT_irr_zero
768 00\XINT_irr_loop_a
769 \krof
770 {#3#4}{#1#2}{#3#4}{#1#2}%
771 }%
772 \def\XINT_irr_indeterminate #1#2#3#4#5%
773 {%
774 \XINT_signalcondition{DivisionUndefined}{0/0 indeterminate fraction.}{}{ 0/1}%
775 }%
776 \def\XINT_irr_divisionbyzero #1#2#3#4#5%
777 {%
778 \XINT_signalcondition{DivisionByZero}{Division by zero: #5#2/0.}{}{ 0/1}%
779 }%
780 \def\XINT_irr_zero #1#2#3#4#5{ 0/1}% changed in 1.08
781 \def\XINT_irr_loop_a #1#2%
782 {%
783 \expandafter\XINT_irr_loop_d
784 \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
785 }%
786 \def\XINT_irr_loop_d #1#2%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
787 {%
788     \XINT_irr_loop_e #2\Z
789 }%
790 \def\XINT_irr_loop_e #1#2\Z
791 {%
792     \xint_gob_til_zero #1\XINT_irr_loop_exit0\XINT_irr_loop_a {#1#2}%
793 }%
794 \def\XINT_irr_loop_exit0\XINT_irr_loop_a #1#2#3#4%
795 {%
796     \expandafter\XINT_irr_loop_exitb\expandafter
797     {\romannumeral0\xintiique {#3}{#2}}%
798     {\romannumeral0\xintiique {#4}{#2}}%
799 }%
800 \def\XINT_irr_loop_exitb #1#2%
801 {%
802     \expandafter\XINT_irr_finish\expandafter {#2}{#1}%
803 }%
804 \def\XINT_irr_finish #1#2#3{#3#1/#2}% changed in 1.08
```

### 24.32. \xintifInt

```
805 \def\xintifInt {\romannumeral0\xintifint }%
806 \def\xintifint #1{\expandafter\XINT_ifint\romannumeral0\xintraffwithzeros {#1}.}%
807 \def\XINT_ifint #1/#2.%
808 {%
809     \if 0\xintiiRem {#1}{#2}%
810     \expandafter\xint_stop_atfirstoftwo
811     \else
812     \expandafter\xint_stop_atsecondoftwo
813     \fi
814 }%
```

### 24.33. \xintIsInt

Added at 1.3d only, for isint() xintexpr function.

```
815 \def\xintIsInt {\romannumeral0\xintisint }%
816 \def\xintisint #1%
817     {\expandafter\XINT_ifint\romannumeral0\xintraffwithzeros {#1}.10}%

```

### 24.34. \xintJrr

```
818 \def\xintJrr {\romannumeral0\xintjrr }%
819 \def\xintjrr #1%
820 {%
821     \expandafter\XINT_jrr_start\romannumeral0\xintraffwithzeros {#1}\Z
822 }%
823 \def\XINT_jrr_start #1#2/#3\Z
824 {%
825     \if0\XINT_isOne {#3}\xint_afterfi
826     {\xint_UDsignfork
827         #1\XINT_jrr_negative
828         -{\XINT_jrr_nonneg #1}%
829     \krof}%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

830 \else
831 \xint_afterfi{\XINT_jrr_denomisone #1}%
832 \fi
833 #2\Z {#3}%
834 }%
835 \def\XINT_jrr_denomisone #1\Z #2{ #1/1}% changed in 1.08
836 \def\XINT_jrr_negative #1\Z #2{\XINT_jrr_D #1\Z #2\Z -}%
837 \def\XINT_jrr_nonneg #1\Z #2{\XINT_jrr_D #1\Z #2\Z \space}%
838 \def\XINT_jrr_D #1#2\Z #3#4\Z
839 {%
840 \xint_UDzerosfork
841 #3#1\XINT_jrr_indeterminate
842 #30\XINT_jrr_divisionbyzero
843 #10\XINT_jrr_zero
844 00\XINT_jrr_loop_a
845 \krof
846 {#3#4}{#1#2}1001%
847 }%
848 \def\XINT_jrr_indeterminate #1#2#3#4#5#6#7%
849 {%
850 \XINT_signalcondition{DivisionUndefined}{0/0 indeterminate fraction.}{0/1}%
851 }%
852 \def\XINT_jrr_divisionbyzero #1#2#3#4#5#6#7%
853 {%
854 \XINT_signalcondition{DivisionByZero}{Division by zero: #7#2/0.}{0/1}%
855 }%
856 \def\XINT_jrr_zero #1#2#3#4#5#6#7{ 0/1}% changed in 1.08
857 \def\XINT_jrr_loop_a #1#2%
858 {%
859 \expandafter\XINT_jrr_loop_b
860 \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
861 }%
862 \def\XINT_jrr_loop_b #1#2#3#4#5#6#7%
863 {%
864 \expandafter \XINT_jrr_loop_c \expandafter
865 {\romannumeral0\xinttiadd{\XINT_mul_fork #4\xint:#1\xint:}{#6}}%
866 {\romannumeral0\xinttiadd{\XINT_mul_fork #5\xint:#1\xint:}{#7}}%
867 {#2}{#3}{#4}{#5}%
868 }%
869 \def\XINT_jrr_loop_c #1#2%
870 {%
871 \expandafter \XINT_jrr_loop_d \expandafter{#2}{#1}%
872 }%
873 \def\XINT_jrr_loop_d #1#2#3#4%
874 {%
875 \XINT_jrr_loop_e #3\Z {#4}{#2}{#1}%
876 }%
877 \def\XINT_jrr_loop_e #1#2\Z
878 {%
879 \xint_gob_til_zero #1\XINT_jrr_loop_exit0\XINT_jrr_loop_a {#1#2}%
880 }%
881 \def\XINT_jrr_loop_exit0\XINT_jrr_loop_a #1#2#3#4#5#6%

```

```

882 {%
883     \XINT_irr_finish {#3}{#4}%
884 }%

```

## 24.35. \xintTFrac

**Added at 1.09i (2013/12/18).** For `frac` in `\xintexpr`. And `\xintFrac` is already assigned. T for truncation. However, potentially not very efficient with numbers in scientific notations, with big exponents. Will have to think it again some day. I hesitated how to call the macro. Same convention as in maple, but some people reserve fractional part to  $x - \text{floor}(x)$ . Also, not clear if I had to make it negative (or zero) if  $x < 0$ , or rather always positive. There should be in fact such a thing for each rounding function, `trunc`, `round`, `floor`, `ceil`.

```

885 \def\xintTFrac {\romannumeral0\xinttfrac }%
886 \def\xinttfrac #1{\expandafter\XINT_tfrac_fork\romannumeral0\xintraawwithzeros {#1}\Z }%
887 \def\XINT_tfrac_fork #1%
888 {%
889     \xint_UDzerominusfork
890     #1-\XINT_tfrac_zero
891     0#1{\xintiopp\XINT_tfrac_P }%
892     0-{\XINT_tfrac_P #1}%
893     \krof
894 }%
895 \def\XINT_tfrac_zero #1\Z { 0/1[0]}%
896 \def\XINT_tfrac_P #1/#2\Z {\expandafter\XINT_rez_AB
897     \romannumeral0\xintiirem{#1}{#2}\Z {0}{#2}}%

```

## 24.36. \xintTrunc, \xintiTrunc

This of course has a long history. Only showing here some comments.

**Modified at 1.2i (2016/12/13).** 1.2i release notes: ever since its inception this macro was stupid for a decimal input: it did not handle it separately from the general fraction case  $A/B[N]$  with  $B > 1$ , hence ended up doing divisions by powers of ten. But this meant that nesting `\xintTrunc` with itself was very inefficient.

1.2i version is better. However it still handles  $B > 1$ ,  $N < 0$  via adding zeros to  $B$  and dividing with this extended  $B$ . A possibly more efficient approach is implemented in `\xintXTrunc`, but its logic is more complicated, the code is quite longer and making it f-expandable would not shorten it... I decided for the time being to not complicate things here.

**Modified at 1.4a (2020/02/19).**

Adds handling of a negative first argument.

Zero input still gives single digit 0 output as I did not want to complicate the code. But if quantization gives 0, the exponent  $[D]$  will be there. Well actually  $eD$  because of problem that sign of original is preserved in output so we can have  $-0$  and I can not use  $-0[D]$  notation as it is not legal for strict format. So I will use  $-0eD$  hence  $eD$  generally even though this means so slight suboptimality for `trunc()` function in `\xintexpr`.

The idea to give a meaning to negative  $D$  (in the context of optional argument to `\xintiexpr`) was suggested a long time ago by Kpym (October 20, 2015). His suggestion was then to treat it as positive  $D$  but trim trailing zeroes. But since then, there is `\xintDecToString` which can be combined with `\xintREZ`, and I feel matters of formatting output require a whole module (or rather use existing third-party tools), and I decided to opt rather for an operation similar as the `quantize()` of Python Decimal module. I.e. we truncate (or round) to an integer multiple of a given power of 10.

Other reason to decide to do this is that it looks as if I don't even need to understand the original code to hack into its ending via `\XINT_trunc_G` or `\XINT_itrunc_G`. For the latter it looks as if logically I simply have to do nothing. For the former I simply have to add some `eD` postfix.

```

898 \def\xintTrunc  {\romannumeral0\xinttrunc }%
899 \def\xintiTrunc {\romannumeral0\xintitrunc}%
900 \def\xinttrunc #1{\expandafter\XINT_trunc\the\numexpr#1.\XINT_trunc_G}%
901 \def\xintitrunc #1{\expandafter\XINT_trunc\the\numexpr#1.\XINT_itrunc_G}%
902 \def\XINT_trunc #1.#2#3%
903 {%
904   \expandafter\XINT_trunc_a\romannumeral0\XINT_infrac{#3}#1.#2%
905 }%
906 \def\XINT_trunc_a #1#2#3#4.#5%
907 {%
908   \if0\XINT_Sgn#2\xint:\xint_dothis\XINT_trunc_zero\fi
909   \if1\XINT_is_One#3XY\xint_dothis\XINT_trunc_sp_b\fi
910   \xint_orthat\XINT_trunc_b #1+#4.{#2}{#3}#5#4.%
911 }%
912 \def\XINT_trunc_zero #1.#2.{ 0}%
913 \def\XINT_trunc_b     {\expandafter\XINT_trunc_B\the\numexpr}%
914 \def\XINT_trunc_sp_b  {\expandafter\XINT_trunc_sp_B\the\numexpr}%
915 \def\XINT_trunc_B #1%
916 {%
917   \xint_UDsignfork
918     #1\XINT_trunc_C
919     -\XINT_trunc_D
920   \krof #1%
921 }%
922 \def\XINT_trunc_sp_B #1%
923 {%
924   \xint_UDsignfork
925     #1\XINT_trunc_sp_C
926     -\XINT_trunc_sp_D
927   \krof #1%
928 }%
929 \def\XINT_trunc_C -#1.#2#3%
930 {%
931   \expandafter\XINT_trunc_CE
932   \romannumeral0\XINT_dsx_addzeros{#1}#3;.{#2}%
933 }%
934 \def\XINT_trunc_CE #1.#2{\XINT_trunc_E #2.{#1}}%
935 \def\XINT_trunc_sp_C -#1.#2#3{\XINT_trunc_sp_Ca #2.#1}%
936 \def\XINT_trunc_sp_Ca #1%
937 {%
938   \xint_UDsignfork
939     #1{\XINT_trunc_sp_Cb -}%
940     -{\XINT_trunc_sp_Cb \space#1}%
941   \krof
942 }%
943 \def\XINT_trunc_sp_Cb #1#2.#3.%
944 {%
945   \expandafter\XINT_trunc_sp_Cc
946   \romannumeral0\expandafter\XINT_split_fromright_a

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrc*, *xintexpr*, *xinttrig*, *xintlog*

```
947 \the\numexpr#3-\numexpr\XINT_length_loop
948 #2\xint:\xint:\xint:\xint:\xint:\xint:\xint:\xint:
949 \xint_c_viii\xint_c_vii\xint_c_vi\xint_c_v
950 \xint_c_iv\xint_c_iii\xint_c_ii\xint_c_i\xint_c_\xint_bye
951 .#2\xint_bye2345678\xint_bye..#1%
952 }%
953 \def\XINT_trunc_sp_Cc #1%
954 {%
955 \if.#1\xint_dothis{\XINT_trunc_sp_Cd 0.}\fi
956 \xint_orthat {\XINT_trunc_sp_Cd #1}%
957 }%
958 \def\XINT_trunc_sp_Cd #1.#2.#3%
959 {%
960 \XINT_trunc_sp_F #3#1.%
961 }%
962 \def\XINT_trunc_D #1.#2%
963 {%
964 \expandafter\XINT_trunc_E
965 \romannumeral0\XINT_dsx_addzeros {#1}#2;.%
966 }%
967 \def\XINT_trunc_sp_D #1.#2#3%
968 {%
969 \expandafter\XINT_trunc_sp_E
970 \romannumeral0\XINT_dsx_addzeros {#1}#2;.%
971 }%
972 \def\XINT_trunc_E #1%
973 {%
974 \xint_UDsignfork
975 #1{\XINT_trunc_F -}%
976 -{\XINT_trunc_F \space#1}%
977 \krof
978 }%
979 \def\XINT_trunc_sp_E #1%
980 {%
981 \xint_UDsignfork
982 #1{\XINT_trunc_sp_F -}%
983 -{\XINT_trunc_sp_F\space#1}%
984 \krof
985 }%
986 \def\XINT_trunc_F #1#2.#3#4%
987 {\expandafter#4\romannumeral`&&\expandafter\xint_firstoftwo
988 \romannumeral0\XINT_div_prepare {#3}{#2}.#1}%
989 \def\XINT_trunc_sp_F #1#2.#3{#3#2.#1}%
990 \def\XINT_itrunc_G #1#2.#3#4.%
991 {%
992 \if#10\xint_dothis{ 0}\fi
993 \xint_orthat{#3#1}#2%
994 }%
995 \def\XINT_trunc_G #1.#2#3#4.%
996 {%
997 \xint_gob_til_minus#3\XINT_trunc_Hc-%
998 \expandafter\XINT_trunc_H
```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
999 \the\numexpr\romannumeral0\xintlength {#1}-#3#4.#3#4.{#1}#2%
1000 }%
1001 \def\xINT_trunc_Hc-\expandafter\xINT_trunc_H
1002 \the\numexpr\romannumeral0\xintlength #1.-#2.#3#4{#4#3e#2}%
1003 \def\xINT_trunc_H #1.#2.%
1004 {%
1005 \ifnum #1 > \xint_c \xint_dothis{\XINT_trunc_Ha {#2}}\fi
1006 \xint_orthat {\XINT_trunc_Hb {-#1}}% -0,--1,--2, ...
1007 }%
1008 \def\xINT_trunc_Ha%
1009 {%
1010 \expandafter\xINT_trunc_Haa\romannumeral0\xintdecsplit
1011 }%
1012 \def\xINT_trunc_Haa #1#2#3{#3#1.#2}%
1013 \def\xINT_trunc_Hb #1#2#3%
1014 {%
1015 \expandafter #3\expandafter0\expandafter.%
1016 \romannumeral\xintreplicate{#1}0#2%
1017 }%
```

### 24.37. \xintTTrunc

Added at 1.1 (2014/10/28).

```
1018 \def\xintTTrunc {\romannumeral0\xintttrunc }%
1019 \def\xintttrunc {\xintitrunc\xint_c}%
```

### 24.38. \xintNum, \xintnum

```
1020 \let\xintnum \xintttrunc
```

### 24.39. \xintRound, \xintiRound

Modified in 1.2i.

It benefits first of all from the faster [\xintTrunc](#), particularly when the input is already a decimal number (denominator B=1).

And the rounding is now done in 1.2 style (with much delay, sorry), like of the rewritten [\xintInc](#) and [\xintDec](#).

At 1.4a, first argument can be negative. This is handled at [\XINT\\_trunc\\_G](#).

```
1021 \def\xintRound {\romannumeral0\xintround }%
1022 \def\xintiRound {\romannumeral0\xintiround }%
1023 \def\xintround #1{\expandafter\XINT_round\the\numexpr #1.\XINT_round_A}%
1024 \def\xintiround #1{\expandafter\XINT_round\the\numexpr #1.\XINT_iround_A}%
1025 \def\XINT_round #1.{\expandafter\XINT_round_aa\the\numexpr #1+\xint_c.i.#1}%
1026 \def\XINT_round_aa #1.#2.#3#4%
1027 {%
1028 \expandafter\XINT_round_a\romannumeral0\XINT_infrac{#4}#1.#3#2.%
1029 }%
1030 \def\XINT_round_a #1#2#3#4.%
1031 {%
1032 \if0\XINT_Sgn#2\xint:\xint_dothis\XINT_trunc_zero\fi
1033 \if1\XINT_is_One#3XY\xint_dothis\XINT_trunc_sp_b\fi
1034 \xint_orthat\XINT_trunc_b #1+#4.{#2}{#3}%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfraction, xintexpr, xinttrig, xintlog

```

1035 }%
1036 \def\XINT_round_A{\expandafter\XINT_trunc_G\romannumeral0\XINT_round_B}%
1037 \def\XINT_iround_A{\expandafter\XINT_itrunc_G\romannumeral0\XINT_round_B}%
1038 \def\XINT_round_B #1.%
1039     {\XINT_dsrr #1\xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax.}%

```

### 24.40. \xintXTrunc

Added at 1.09j (2014/01/09) [on 2014/01/06]. This is completely expandable but not f-expandable.

Modified at 1.2i (2016/12/13). Rewritten:

- no more use of \xinttiloop from xinttools.sty (replaced by \xintreplicate... from xintkernel.sty),
- no more use in 0>N>-D case of a dummy control sequence name via \csname...\endcsname
- handles better the case of an input already a decimal number

```

1040 \def\xintXTrunc #1%#2%
1041 {%
1042     \expandafter\XINT_xtrunc_a
1043     \the\numexpr #1\expandafter.\romannumeral0\xintraw
1044 }%
1045 \def\XINT_xtrunc_a #1.% ?? faire autre chose
1046 {%
1047     \expandafter\XINT_xtrunc_b\the\numexpr\ifnum#1<\xint_c_i \xint_c_i-\fi #1.%
1048 }%
1049 \def\XINT_xtrunc_b #1.#2{\XINT_xtrunc_c #2{#1}}%
1050 \def\XINT_xtrunc_c #1%
1051 {%
1052     \xint_UDzerominusfork
1053     #1-\XINT_xtrunc_zero
1054     0#1{-\XINT_xtrunc_d {}}%
1055     0-{\XINT_xtrunc_d #1}%
1056     \krof
1057 }%[
1058 \def\XINT_xtrunc_zero #1#2]{0.\romannumeral\xintreplicate{#1}0}%
1059 \def\XINT_xtrunc_d #1#2#3/#4[#5]%
1060 {%
1061     \XINT_xtrunc_prepare_a#4\R\R\R\R\R\R\R {10}0000001\W
1062     !{#4};{#5}{#2}{#1#3}%
1063 }%
1064 \def\XINT_xtrunc_prepare_a #1#2#3#4#5#6#7#8#9%
1065 {%
1066     \xint_gob_til_R #9\XINT_xtrunc_prepare_small\R
1067     \XINT_xtrunc_prepare_b #9%
1068 }%
1069 \def\XINT_xtrunc_prepare_small\R #1!#2;%
1070 {%
1071     \ifcase #2
1072     \or\expandafter\XINT_xtrunc_BisOne
1073     \or\expandafter\XINT_xtrunc_BisTwo
1074     \or
1075     \or\expandafter\XINT_xtrunc_BisFour
1076     \or\expandafter\XINT_xtrunc_BisFive
1077     \or

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfraction, xintexpr, xinttrig, xintlog

```

1078 \or
1079 \or\expandafter\XINT_xtrunc_BisEight
1080 \fi\XINT_xtrunc_BisSmall {#2}%
1081 }%
1082 \def\XINT_xtrunc_BisOne\XINT_xtrunc_BisSmall #1#2#3#4%
1083 {\XINT_xtrunc_sp_e {#2}{#4}{#3}}%
1084 \def\XINT_xtrunc_BisTwo\XINT_xtrunc_BisSmall #1#2#3#4%
1085 {%
1086 \expandafter\XINT_xtrunc_sp_e\expandafter
1087 {\the\numexpr #2-\xint_c_i\expandafter}\expandafter
1088 {\romannumeral0\xintiimul 5{#4}}{#3}%
1089 }%
1090 \def\XINT_xtrunc_BisFour\XINT_xtrunc_BisSmall #1#2#3#4%
1091 {%
1092 \expandafter\XINT_xtrunc_sp_e\expandafter
1093 {\the\numexpr #2-\xint_c_ii\expandafter}\expandafter
1094 {\romannumeral0\xintiimul {25}{#4}}{#3}%
1095 }%
1096 \def\XINT_xtrunc_BisFive\XINT_xtrunc_BisSmall #1#2#3#4%
1097 {%
1098 \expandafter\XINT_xtrunc_sp_e\expandafter
1099 {\the\numexpr #2-\xint_c_i\expandafter}\expandafter
1100 {\romannumeral0\xintdouble {#4}}{#3}%
1101 }%
1102 \def\XINT_xtrunc_BisEight\XINT_xtrunc_BisSmall #1#2#3#4%
1103 {%
1104 \expandafter\XINT_xtrunc_sp_e\expandafter
1105 {\the\numexpr #2-\xint_c_iii\expandafter}\expandafter
1106 {\romannumeral0\xintiimul {125}{#4}}{#3}%
1107 }%
1108 \def\XINT_xtrunc_BisSmall #1%
1109 {%
1110 \expandafter\XINT_xtrunc_e\expandafter
1111 {\expandafter\XINT_xtrunc_small_a
1112 \the\numexpr #1/\xint_c_ii\expandafter
1113 .\the\numexpr \xint_c_x^viii+#1!}%
1114 }%
1115 \def\XINT_xtrunc_small_a #1.#2!#3%
1116 {%
1117 \expandafter\XINT_div_small_b\the\numexpr #1\expandafter
1118 \xint:\the\numexpr #2\expandafter!%
1119 \romannumeral0\XINT_div_small_ba #3\R\R\R\R\R\R\R\R{10}0000001\W
1120 #3\XINT_sepbyviii_Z_end 2345678\relax
1121 }%
1122 \def\XINT_xtrunc_prepare_b
1123 {\expandafter\XINT_xtrunc_prepare_c\romannumeral0\XINT_zeroes_forviii }%
1124 \def\XINT_xtrunc_prepare_c #1!%
1125 {%
1126 \XINT_xtrunc_prepare_d #1.00000000!{#1}%
1127 }%
1128 \def\XINT_xtrunc_prepare_d #1#2#3#4#5#6#7#8#9%
1129 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1130 \expandafter\XINT_xtrunc_prepare_e
1131 \xint_gob_til_dot #1#2#3#4#5#6#7#8#9!%
1132 }%
1133 \def\XINT_xtrunc_prepare_e #1!#2!#3#4%
1134 {%
1135 \XINT_xtrunc_prepare_f #4#3\X {#1}{#3}%
1136 }%
1137 \def\XINT_xtrunc_prepare_f #1#2#3#4#5#6#7#8#9\X
1138 {%
1139 \expandafter\XINT_xtrunc_prepare_g\expandafter
1140 \XINT_div_prepare_g
1141 \the\numexpr #1#2#3#4#5#6#7#8+\xint_c_i\expandafter
1142 \xint:\the\numexpr (#1#2#3#4#5#6#7#8+\xint_c_i)/\xint_c_ii\expandafter
1143 \xint:\the\numexpr #1#2#3#4#5#6#7#8\expandafter
1144 \xint:\romannumeral0\XINT_sepandrev_andcount
1145 #1#2#3#4#5#6#7#8#9\XINT_rsepbyviii_end_A 2345678%
1146 \XINT_rsepbyviii_end_B 2345678\relax\xint_c_ii\xint_c_i
1147 \R\xint:\xint_c_xii \R\xint:\xint_c_x \R\xint:\xint_c_viii \R\xint:\xint_c_vi
1148 \R\xint:\xint_c_iv \R\xint:\xint_c_ii \R\xint:\xint_c_W
1149 \X
1150 }%
1151 \def\XINT_xtrunc_prepare_g #1;{\XINT_xtrunc_e {#1}}%
1152 \def\XINT_xtrunc_e #1#2%
1153 {%
1154 \ifnum #2<\xint_c_
1155 \expandafter\XINT_xtrunc_I
1156 \else
1157 \expandafter\XINT_xtrunc_II
1158 \fi #2\xint:{#1}%
1159 }%
1160 \def\XINT_xtrunc_I -#1\xint:#2#3#4%
1161 {%
1162 \expandafter\XINT_xtrunc_I_a\romannumeral0#2{#4}{#2}{#1}{#3}%
1163 }%
1164 \def\XINT_xtrunc_I_a #1#2#3#4#5%
1165 {%
1166 \expandafter\XINT_xtrunc_I_b\the\numexpr #4-#5\xint:#4\xint:{#5}{#2}{#3}{#1}%
1167 }%
1168 \def\XINT_xtrunc_I_b #1%
1169 {%
1170 \xint_UDsignfork
1171 #1\XINT_xtrunc_IA_c
1172 -\XINT_xtrunc_IB_c
1173 \krof #1%
1174 }%
1175 \def\XINT_xtrunc_IA_c -#1\xint:#2\xint:#3#4#5#6%
1176 {%
1177 \expandafter\XINT_xtrunc_IA_d
1178 \the\numexpr#2-\xintLength{#6}\xint:{#6}%
1179 \expandafter\XINT_xtrunc_IA_xd
1180 \the\numexpr (#1+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i\xint:#1\xint:{#5}{#4}%
1181 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1182 \def\XINT_xtrunc_IA_d #1%
1183 {%
1184     \xint_UDsignfork
1185     #1\XINT_xtrunc_IAA_e
1186     -\XINT_xtrunc_IAB_e
1187     \krof #1%
1188 }%
1189 \def\XINT_xtrunc_IAA_e -#1\xint:#2%
1190 {%
1191     \romannumeral0\XINT_split_fromleft
1192     #1.#2\xint_gobble_i\xint_bye2345678\xint_bye..%
1193 }%
1194 \def\XINT_xtrunc_IAB_e #1\xint:#2%
1195 {%
1196     0.\romannumeral\XINT_rep#1\endcsname0#2%
1197 }%
1198 \def\XINT_xtrunc_IA_xd #1\xint:#2\xint:%
1199 {%
1200     \expandafter\XINT_xtrunc_IA_xe\the\numexpr #2-\xint_c_ii^vi*#1\xint:#1\xint:%
1201 }%
1202 \def\XINT_xtrunc_IA_xe #1\xint:#2\xint:#3#4%
1203 {%
1204     \XINT_xtrunc_loop {#2}{#4}{#3}{#1}%
1205 }%
1206 \def\XINT_xtrunc_IB_c #1\xint:#2\xint:#3#4#5#6%
1207 {%
1208     \expandafter\XINT_xtrunc_IB_d
1209     \romannumeral0\XINT_split_xfork #1.#6\xint_bye2345678\xint_bye..{#3}%
1210 }%
1211 \def\XINT_xtrunc_IB_d #1.#2.#3%
1212 {%
1213     \expandafter\XINT_xtrunc_IA_d\the\numexpr#3-\xintLength {#1}\xint:{#1}%
1214 }%
1215 \def\XINT_xtrunc_II #1\xint:%
1216 {%
1217     \expandafter\XINT_xtrunc_II_a\romannumeral\xintreplicate{#1}0\xint:%
1218 }%
1219 \def\XINT_xtrunc_II_a #1\xint:#2#3#4%
1220 {%
1221     \expandafter\XINT_xtrunc_II_b
1222     \the\numexpr (#3+\xint_c_ii^v)/\xint_c_ii^vi-\xint_c_i\expandafter\xint:%
1223     \the\numexpr #3\expandafter\xint:\romannumeral0#2{#4#1}{#2}%
1224 }%
1225 \def\XINT_xtrunc_II_b #1\xint:#2\xint:%
1226 {%
1227     \expandafter\XINT_xtrunc_II_c\the\numexpr #2-\xint_c_ii^vi*#1\xint:#1\xint:%
1228 }%
1229 \def\XINT_xtrunc_II_c #1\xint:#2\xint:#3#4#5%
1230 {%
1231     #3.\XINT_xtrunc_loop {#2}{#4}{#5}{#1}%
1232 }%
1233 \def\XINT_xtrunc_loop #1%

```

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcffrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

1234 {%
1235 \ifnum #1=\xint_c_ \expandafter\XINT_xtrunc_transition\fi
1236 \expandafter\XINT_xtrunc_loop_a\the\numexpr #1-\xint_c_i\xint:%
1237 }%
1238 \def\XINT_xtrunc_loop_a #1\xint:#2#3%
1239 {%
1240 \expandafter\XINT_xtrunc_loop_b\romannumeral0#3%
1241 {#20000000000000000000000000000000000000000000000000000000000000000000000000000000}%
1242 {#1}{#3}%
1243 }%
1244 \def\XINT_xtrunc_loop_b #1#2#3%
1245 {%
1246 \romannumeral\xintreplicate{\xint_c_ii^vi-\xintLength{#1}}0#1%
1247 \XINT_xtrunc_loop {#3}{#2}%
1248 }%
1249 \def\XINT_xtrunc_transition
1250 \expandafter\XINT_xtrunc_loop_a\the\numexpr #1\xint:#2#3#4%
1251 {%
1252 \ifnum #4=\xint_c_ \expandafter\xint_gobble_vi\fi
1253 \expandafter\XINT_xtrunc_finish\expandafter
1254 {\romannumeral0\XINT_dsx_addzeros{#4}#2;}{#3}{#4}%
1255 }%
1256 \def\XINT_xtrunc_finish #1#2%
1257 {%
1258 \expandafter\XINT_xtrunc_finish_a\romannumeral0#2{#1}%
1259 }%
1260 \def\XINT_xtrunc_finish_a #1#2#3%
1261 {%
1262 \romannumeral\xintreplicate{#3-\xintLength{#1}}0#1%
1263 }%
1264 \def\XINT_xtrunc_sp_e #1%
1265 {%
1266 \ifnum #1<\xint_c_
1267 \expandafter\XINT_xtrunc_sp_I
1268 \else
1269 \expandafter\XINT_xtrunc_sp_II
1270 \fi #1\xint:%
1271 }%
1272 \def\XINT_xtrunc_sp_I -#1\xint:#2#3%
1273 {%
1274 \expandafter\XINT_xtrunc_sp_I_a\the\numexpr #1-#3\xint:#1\xint:{#3}{#2}%
1275 }%
1276 \def\XINT_xtrunc_sp_I_a #1%
1277 {%
1278 \xint_UDsignfork
1279 #1\XINT_xtrunc_sp_IA_b
1280 -\XINT_xtrunc_sp_IB_b
1281 \krof #1%
1282 }%
1283 \def\XINT_xtrunc_sp_IA_b -#1\xint:#2\xint:#3#4%
1284 {%
1285 \expandafter\XINT_xtrunc_sp_IA_c

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrc*, *xintexpr*, *xinttrig*, *xintlog*

```
1286 \the\numexpr#2-\xintLength{#4}\xint:{#4}\romannumeral\xINT_rep#1\endcsname0%
1287 }%
1288 \def\xINT_xtrunc_sp_IA_c #1%
1289 {%
1290 \xint_UDsignfork
1291 #1\xINT_xtrunc_sp_IAA
1292 -\xINT_xtrunc_sp_IAB
1293 \krof #1%
1294 }%
1295 \def\xINT_xtrunc_sp_IAA -#1\xint:#2%
1296 {%
1297 \romannumeral0\xINT_split_fromleft
1298 #1.#2\xint_gobble_i\xint_bye2345678\xint_bye..%
1299 }%
1300 \def\xINT_xtrunc_sp_IAB #1\xint:#2%
1301 {%
1302 0.\romannumeral\xINT_rep#1\endcsname0#2%
1303 }%
1304 \def\xINT_xtrunc_sp_IB_b #1\xint:#2\xint:#3#4%
1305 {%
1306 \expandafter\xINT_xtrunc_sp_IB_c
1307 \romannumeral0\xINT_split_xfork #1.#4\xint_bye2345678\xint_bye..{#3}%
1308 }%
1309 \def\xINT_xtrunc_sp_IB_c #1.#2.#3%
1310 {%
1311 \expandafter\xINT_xtrunc_sp_IA_c\the\numexpr#3-\xintLength {#1}\xint:{#1}%
1312 }%
1313 \def\xINT_xtrunc_sp_II #1\xint:#2#3%
1314 {%
1315 #2\romannumeral\xINT_rep#1\endcsname0.\romannumeral\xINT_rep#3\endcsname0%
1316 }%
```

### 24.41. \xintAdd

**Modified at 1.3 (2018/03/01).** Big change at 1.3:  $a/b+c/d$  uses  $\text{lcm}(b,d)$  as denominator.

```
1317 \def\xintAdd {\romannumeral0\xintadd }%
1318 \def\xintadd #1{\expandafter\xINT_fadd\romannumeral0\xintraw {#1}}%
1319 \def\xINT_fadd #1{\xint_gob_til_zero #1\xINT_fadd_Azero 0\xINT_fadd_a #1}%
1320 \def\xINT_fadd_Azero #1{\xintraw }%
1321 \def\xINT_fadd_a #1/#2[#3]#4%
1322 {\expandafter\xINT_fadd_b\romannumeral0\xintraw {#4}{#3}{#1}{#2}}%
1323 \def\xINT_fadd_b #1{\xint_gob_til_zero #1\xINT_fadd_Bzero 0\xINT_fadd_c #1}%
1324 \def\xINT_fadd_Bzero #1#2#3#4{ #3/#4[#2]}%
1325 \def\xINT_fadd_c #1/#2[#3]#4%
1326 {%
1327 \expandafter\xINT_fadd_Aa\the\numexpr #4-#3.{#3}{#4}{#1}{#2}%
1328 }%
1329 \def\xINT_fadd_Aa #1%
1330 {%
1331 \xint_UDzerominusfork
1332 #1-\xINT_fadd_B
1333 0#1\xINT_fadd_Bb
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfraction, xintexpr, xinttrig, xintlog

```

1334      0-\XINT_fadd_Ba
1335      \krof #1%
1336  }%
1337  \def\XINT_fadd_B    #1.#2#3#4#5#6#7{\XINT_fadd_C {#4}{#5}{#7}{#6}[#3]}%
1338  \def\XINT_fadd_Ba  #1.#2#3#4#5#6#7%
1339  {%
1340      \expandafter\XINT_fadd_C\expandafter
1341          {\romannumeral0\XINT_dsx_addzeros {#1}#6;}%
1342          {#7}{#5}{#4}[#2]}%
1343  }%
1344  \def\XINT_fadd_Bb  -#1.#2#3#4#5#6#7%
1345  {%
1346      \expandafter\XINT_fadd_C\expandafter
1347          {\romannumeral0\XINT_dsx_addzeros {#1}#4;}%
1348          {#5}{#7}{#6}[#3]}%
1349  }%
1350  \def\XINT_fadd_iszero #1[#2]{ 0/1[0]}% ou [#2] originel?
1351  \def\XINT_fadd_C  #1#2#3%
1352  {%
1353      \expandafter\XINT_fadd_D_b
1354      \romannumeral0\XINT_div_prepare{#2}{#3}{#2}{#2}{#3}{#1}%
1355  }%

```

Basically a clone of the `\XINT_irr_loop_a` loop. I should modify the output of `\XINT_div_prepare` perhaps to be optimized for checking if remainder vanishes.

```

1356  \def\XINT_fadd_D_a  #1#2%
1357  {%
1358      \expandafter\XINT_fadd_D_b
1359      \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
1360  }%
1361  \def\XINT_fadd_D_b  #1#2{\XINT_fadd_D_c #2\Z}%
1362  \def\XINT_fadd_D_c  #1#2\Z
1363  {%
1364      \xint_gob_til_zero #1\XINT_fadd_D_exit0\XINT_fadd_D_a {#1#2}%
1365  }%
1366  \def\XINT_fadd_D_exit0\XINT_fadd_D_a  #1#2#3%
1367  {%
1368      \expandafter\XINT_fadd_E
1369      \romannumeral0\xintiitquo {#3}{#2}.{#2}%
1370  }%
1371  \def\XINT_fadd_E  #1.#2#3%
1372  {%
1373      \expandafter\XINT_fadd_F
1374      \romannumeral0\xintiimul{#1}{#3}.{\xintiitquo{#3}{#2}}{#1}%
1375  }%
1376  \def\XINT_fadd_F  #1.#2#3#4#5%
1377  {%
1378      \expandafter\XINT_fadd_G
1379      \romannumeral0\xintiitadd{\xintiimul{#2}{#4}}{\xintiimul{#3}{#5}}/#1%
1380  }%
1381  \def\XINT_fadd_G  #1{%
1382  \def\XINT_fadd_G ##1{\if0##1\expandafter\XINT_fadd_iszero\fi#1##1}%
1383  }\XINT_fadd_G{ }%

```



## 24.42. \xintSub

Modified at 1.3 (2018/03/01). Since 1.3 will use least common multiple of denominators.

```

1384 \def\xintSub {\romannumeral0\xintsub }%
1385 \def\xintsub #1{\expandafter\XINT_fsub\romannumeral0\xintra {#1}}%
1386 \def\XINT_fsub #1{\xint_gob_til_zero #1\XINT_fsub_Azero 0\XINT_fsub_a #1}%
1387 \def\XINT_fsub_Azero #1{\xintopp }%
1388 \def\XINT_fsub_a #1/#2[#3]#4%
1389 {\expandafter\XINT_fsub_b\romannumeral0\xintra {#4}{#3}{#1}{#2}}%
1390 \def\XINT_fsub_b #1{\xint_UDzerominusfork
1391     #1-\XINT_fadd_Bzero
1392     0#1\XINT_fadd_c
1393     0-{\XINT_fadd_c -#1}%
1394 \krof }%
```

## 24.43. \xintSum

There was (not documented anymore since 1.09d, 2013/10/22) a macro `\xintSumExpr`, but it has been deleted at 1.21.

Empty items in the input are not accepted by this macro, but the input may be empty.

Refactored slightly at 1.4. `\XINT_Sum` used in `xintexpr` code.

```

1395 \def\xintSum {\romannumeral0\xintsum }%
1396 \def\xintsum #1{\expandafter\XINT_sum\romannumeral`&&@#1^}%
1397 \def\XINT_Sum{\romannumeral0\XINT_sum}%
1398 \def\XINT_sum#1%
1399 {%
1400     \xint_gob_til_ ^ #1\XINT_sum_empty ^%
1401     \expandafter\XINT_sum_loop\romannumeral0\xintra{#1}\xint:
1402 }%
1403 \def\XINT_sum_empty ^#1\xint:{ 0/1[0]}%
1404 \def\XINT_sum_loop #1\xint:#2%
1405 {%
1406     \xint_gob_til_ ^ #2\XINT_sum_end ^%
1407     \expandafter\XINT_sum_loop
1408     \romannumeral0\xintadd{#1}{\romannumeral0\xintra{#2}}\xint:
1409 }%
1410 \def\XINT_sum_end ^#1\xintadd #2#3\xint:{ #2}%

```

## 24.44. \xintMul

```

1411 \def\xintMul {\romannumeral0\xintmul }%
1412 \def\xintmul #1{\expandafter\XINT_fmula\romannumeral0\xintra {#1}.}%
1413 \def\XINT_fmula #1{\xint_gob_til_zero #1\XINT_fmula_zero 0\XINT_fmula_a #1}%
1414 \def\XINT_fmula_a #1[#2].#3%
1415 {\expandafter\XINT_fmula_b\romannumeral0\xintra {#3}#1[#2.]}%
1416 \def\XINT_fmula_b #1{\xint_gob_til_zero #1\XINT_fmula_zero 0\XINT_fmula_c #1}%
1417 \def\XINT_fmula_c #1/#2[#3]#4/#5[#6.}%
1418 {%
1419     \expandafter\XINT_fmula_d
1420     \expandafter{\the\numexpr #3+#6\expandafter}%
1421     \expandafter{\romannumeral0\xintiimul {#5}{#2}}%
1422     {\romannumeral0\xintiimul {#4}{#1}}%
1423 }%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
1424 \def\XINT_fmula_d #1#2#3%
1425 {%
1426   \expandafter \XINT_fmula_e \expandafter{#3}{#1}{#2}%
1427 }%
1428 \def\XINT_fmula_e #1#2{\XINT_outfrac {#2}{#1}}%
1429 \def\XINT_fmula_zero #1.#2{ 0/1[0]}%
```

### 24.45. \xintSqr

```
1430 \def\xintSqr {\romannumeral0\xintsqr }%
1431 \def\xintsqr #1{\expandafter\XINT_fsqr\romannumeral0\xintra #1}%
1432 \def\XINT_fsqr #1{\xint_gob_til_zero #1\XINT_fsqr_zero 0\XINT_fsqr_a #1}%
1433 \def\XINT_fsqr_a #1/#2[#3]%
1434 {%
1435   \expandafter\XINT_fsqr_b
1436   \expandafter{\the\numexpr #3+#3\expandafter}%
1437   \expandafter{\romannumeral0\xintiisqr {#2}}%
1438   {\romannumeral0\xintiisqr {#1}}%
1439 }%
1440 \def\XINT_fsqr_b #1#2#3{\expandafter \XINT_fmula_e \expandafter{#3}{#1}{#2}}%
1441 \def\XINT_fsqr_zero #1{ 0/1[0]}%
```

### 24.46. \xintPow

1.2f: to be coherent with the "i" convention `\xintiPow` should parse also its exponent via `\xintNum` when `xintfrac.sty` is loaded. This was not the case so far. Cependant le problème est que le fait d'appliquer `\xintNum` rend impossible certains inputs qui auraient pu être gérés par `\numexpr`. Le `\numexpr` externe est ici pour intercepter trop grand input.

```
1442 \def\xintipow #1#2%
1443 {%
1444   \expandafter\xint_pow\the\numexpr \xintNum{#2}\expandafter
1445   .\romannumeral0\xintnum{#1}\xint:
1446 }%
1447 \def\xintPow {\romannumeral0\xintpow }%
1448 \def\xintpow #1%
1449 {%
1450   \expandafter\XINT_fpow\expandafter {\romannumeral0\XINT_infrac {#1}}%
1451 }%
1452 \def\XINT_fpow #1#2%
1453 {%
1454   \expandafter\XINT_fpow_fork\the\numexpr \xintNum{#2}\relax\Z #1%
1455 }%
1456 \def\XINT_fpow_fork #1#2\Z
1457 {%
1458   \xint_UDzerominusfork
1459   #1-\XINT_fpow_zero
1460   0#1\XINT_fpow_neg
1461   0-{\XINT_fpow_pos #1}%
1462   \kroft
1463   {#2}%
1464 }%
1465 \def\XINT_fpow_zero #1#2#3#4{ 1/1[0]}%
1466 \def\XINT_fpow_pos #1#2#3#4#5%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
1467 {%
1468   \expandafter\XINT_fpow_pos_A\expandafter
1469   {\the\numexpr #1#2*#3\expandafter}\expandafter
1470   {\romannumeral0\xintiipow {#5}{#1#2}}%
1471   {\romannumeral0\xintiipow {#4}{#1#2}}%
1472 }%
1473 \def\XINT_fpow_neg #1#2#3#4%
1474 {%
1475   \expandafter\XINT_fpow_pos_A\expandafter
1476   {\the\numexpr -#1*#2\expandafter}\expandafter
1477   {\romannumeral0\xintiipow {#3}{#1}}%
1478   {\romannumeral0\xintiipow {#4}{#1}}%
1479 }%
1480 \def\XINT_fpow_pos_A #1#2#3%
1481 {%
1482   \expandafter\XINT_fpow_pos_B\expandafter {#3}{#1}{#2}%
1483 }%
1484 \def\XINT_fpow_pos_B #1#2{\XINT_outfrac {#2}{#1}}%
```

### 24.47. \xintFac

Factorial coefficients: variant which can be chained with other xintfrac macros.

```
1485 \def\xintFac {\romannumeral0\xintfac}%
1486 \def\xintfac #1{\expandafter\XINT_fac_fork\the\numexpr\xintNum{#1}.[0]}%
```

### 24.48. \xintBinomial

Added at 1.2f (2016/03/12).

```
1487 \def\xintBinomial {\romannumeral0\xintbinomial}%
1488 \def\xintbinomial #1#2%
1489 {%
1490   \expandafter\XINT_binom_pre
1491   \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.[0]%
1492 }%
```

### 24.49. \xintPFactorial

Added at 1.2f (2016/03/12). Partial factorial. For needs of xintexpr.sty.

```
1493 \def\xintipfactorial #1#2%
1494 {%
1495   \expandafter\XINT_pfac_fork
1496   \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.%
1497 }%
1498 \def\xintPFactorial {\romannumeral0\xintpfactorial}%
1499 \def\xintpfactorial #1#2%
1500 {%
1501   \expandafter\XINT_pfac_fork
1502   \the\numexpr\xintNum{#1}\expandafter.\the\numexpr\xintNum{#2}.[0]%
1503 }%
```

## 24.50. \xintPrd

Refactored at 1.4. After some hesitation the routine still does not try to detect on the fly a zero item, to abort the loop. Indeed this would add some overhead generally (as we need normalizing the item before checking if it vanishes hence we must then grab things once more).

```

1504 \def\xintPrd {\romannumeral0\xintprd }%
1505 \def\xintprd #1{\expandafter\XINT_prd\romannumeral`&&@#1^}%
1506 \def\XINT_Prd{\romannumeral0\XINT_prd}%
1507 \def\XINT_prd#1%
1508 {%
1509     \xint_gob_til_ ^ #1\XINT_prd_empty ^%
1510     \expandafter\XINT_prd_loop\romannumeral0\xintra{#1}\xint:
1511 }%
1512 \def\XINT_prd_empty ^#1\xint:{ 1/1[0]}%
1513 \def\XINT_prd_loop #1\xint:#2%
1514 {%
1515     \xint_gob_til_ ^ #2\XINT_prd_end ^%
1516     \expandafter\XINT_prd_loop
1517     \romannumeral0\xintmul{#1}{\romannumeral0\xintra{#2}}\xint:
1518 }%
1519 \def\XINT_prd_end ^#1\xintmul #2#3\xint:{ #2}%

```

## 24.51. \xintDiv

```

1520 \def\xintDiv {\romannumeral0\xintdiv }%
1521 \def\xintdiv #1%
1522 {%
1523     \expandafter\XINT_fdiv\expandafter {\romannumeral0\XINT_infrac {#1}}%
1524 }%
1525 \def\XINT_fdiv #1#2%
1526     {\expandafter\XINT_fdiv_A\romannumeral0\XINT_infrac {#2}#1}%
1527 \def\XINT_fdiv_A #1#2#3#4#5#6%
1528 {%
1529     \expandafter\XINT_fdiv_B
1530     \expandafter{\the\numexpr #4-#1\expandafter}%
1531     \expandafter{\romannumeral0\xintiimul {#2}{#6}}%
1532     {\romannumeral0\xintiimul {#3}{#5}}%
1533 }%
1534 \def\XINT_fdiv_B #1#2#3%
1535 {%
1536     \expandafter\XINT_fdiv_C
1537     \expandafter{#3}{#1}{#2}%
1538 }%
1539 \def\XINT_fdiv_C #1#2{\XINT_outfrac {#2}{#1}}%

```

## 24.52. \xintDivFloor

Added at 1.1 (2014/10/28). Changed at 1.2p to not append /1[0] ending but rather output a big integer in strict format, like *\xintDivTrunc* and *\xintDivRound*.

```

1540 \def\xintDivFloor {\romannumeral0\xintdivfloor }%
1541 \def\xintdivfloor #1#2{\xintifloor{\xintDiv {#1}{#2}}}%

```

## 24.53. \xintDivTrunc

Added at 1.1 (2014/10/28).

```
1542 \def\xintDivTrunc    {\romannumeral0\xintdivtrunc }%
1543 \def\xintdivtrunc #1#2{\xintttrunc {\xintDiv {#1}{#2}}}%
```

## 24.54. \xintDivRound

1.1

```
1544 \def\xintDivRound    {\romannumeral0\xintdivround }%
1545 \def\xintdivround #1#2{\xintiround 0{\xintDiv {#1}{#2}}}%
```

## 24.55. \xintModTrunc

Added at 1.1 (2014/10/28). `\xintModTrunc {q1}{q2}` computes  $q1 - q2 * t(q1/q2)$  with  $t(q1/q2)$  equal to the truncated division of two fractions  $q1$  and  $q2$ .

Its former name, prior to 1.2p, was `\xintMod`.

Modified at 1.3 (2018/03/01). At 1.3, uses least common multiple denominator, like `\xintMod` (next).

```
1546 \def\xintModTrunc {\romannumeral0\xintmodtrunc }%
1547 \def\xintmodtrunc #1{\expandafter\XINT_modtrunc_a\romannumeral0\xintra{#1}.}%
1548 \def\XINT_modtrunc_a #1#2.#3%
1549   {\expandafter\XINT_modtrunc_b\expandafter #1\romannumeral0\xintra{#3}#2.}%
1550 \def\XINT_modtrunc_b #1#2% #1 de A, #2 de B.
1551 {%
1552   \if0#2\xint_dothis{\XINT_modtrunc_divbyzero #1#2}\fi
1553   \if0#1\xint_dothis\XINT_modtrunc_aiszero\fi
1554   \if-#2\xint_dothis{\XINT_modtrunc_bneg #1}\fi
1555   \xint_orthat{\XINT_modtrunc_bpos #1#2}%
1556 }%
1557 \def\XINT_modtrunc_divbyzero #1#2[#3]#4.%
1558 {%
1559   \XINT_signalcondition{DivisionByZero}{Division by zero: #1#4/(#2[#3]).}{\{ 0/1[0]}}%
1560 }%
1561 \def\XINT_modtrunc_aiszero #1.{ 0/1[0]}%
1562 \def\XINT_modtrunc_bneg #1%
1563 {%
1564   \xint_UDsignfork
1565     #1{\xintiopp\XINT_modtrunc_pos {}}%
1566     -{\XINT_modtrunc_pos #1}%
1567   \krof
1568 }%
1569 \def\XINT_modtrunc_bpos #1%
1570 {%
1571   \xint_UDsignfork
1572     #1{\xintiopp\XINT_modtrunc_pos {}}%
1573     -{\XINT_modtrunc_pos #1}%
1574   \krof
1575 }%
```

Attention. This crucially uses that `xint`'s `\xintiiE{x}{e}` is defined to return  $x$  unchanged if  $e$  is negative (and  $x$  extended by  $e$  zeroes if  $e \geq 0$ ).

```

1576 \def\XINT_modtrunc_pos #1#2/#3[#4]#5/#6[#7].%
1577 {%
1578   \expandafter\XINT_modtrunc_pos_a
1579   \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
1580   \romannumeral0\expandafter\XINT_mod_D_b
1581   \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%
1582   {#1#5}{#7-#4}{#2}{#4-#7}%
1583 }%
1584 \def\XINT_modtrunc_pos_a #1.#2#3#4{\xintiirem {#3}{#4}/#2[#1]}%

```

## 24.56. \xintDivMod

**Added at 1.2p (2017/12/05).** `\xintDivMod{q1}{q2}` outputs `{floor(q1/q2)}{q1 - q2*floor(q1/q2)}`.

Attention that it relies on `\xintiiE{x}{e}` returning `x` if `e < 0`.

**Modified at 1.3 (2018/03/01).** Modified (like `\xintAdd` and `\xintSub`) at 1.3 to use a l.c.m for final denominator of the "mod" part.

```

1585 \def\xintDivMod {\romannumeral0\xintdivmod }%
1586 \def\xintdivmod #1{\expandafter\XINT_divmod_a\romannumeral0\xintraw{#1}.}%
1587 \def\XINT_divmod_a #1#2.#3%
1588   {\expandafter\XINT_divmod_b\expandafter #1\romannumeral0\xintraw{#3}#2.}%
1589 \def\XINT_divmod_b #1#2% #1 de A, #2 de B.
1590 {%
1591   \if0#2\xint_dothis{\XINT_divmod_divbyzero #1#2}\fi
1592   \if0#1\xint_dothis\XINT_divmod_aiszero\fi
1593   \if-#2\xint_dothis{\XINT_divmod_bneg #1}\fi
1594   \xint_orthat{\XINT_divmod_bpos #1#2}%
1595 }%
1596 \def\XINT_divmod_divbyzero #1#2[#3]#4.%
1597 {%
1598   \XINT_signalcondition{DivisionByZero}{Division by zero: #1#4/(#2[#3]).}{}%
1599   {{0}{0/1[0]}}% à revoir...
1600 }%
1601 \def\XINT_divmod_aiszero #1.{{0}{0/1[0]}}%
1602 \def\XINT_divmod_bneg #1% f // -g = (-f) // g, f % -g = -((-f) % g)
1603 {%
1604   \expandafter\XINT_divmod_bneg_finish
1605   \romannumeral0\xint_UDsignfork
1606   #1{\XINT_divmod_bpos {}}%
1607   -{\XINT_divmod_bpos {-#1}}%
1608   \krof
1609 }%
1610 \def\XINT_divmod_bneg_finish#1#2%
1611 {%
1612   \expandafter\xint_exchangetwo_keepbraces\expandafter
1613   {\romannumeral0\xintiiopp#2}{#1}%
1614 }%
1615 \def\XINT_divmod_bpos #1#2/#3[#4]#5/#6[#7].%
1616 {%
1617   \expandafter\XINT_divmod_bpos_a
1618   \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
1619   \romannumeral0\expandafter\XINT_mod_D_b
1620   \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%

```

```

1621      {#1#5}{#7-#4}{#2}{#4-#7}%
1622 }%
1623 \def\XINT_divmod_bpos_a #1.#2#3#4%
1624 {%
1625     \expandafter\XINT_divmod_bpos_finish
1626     \romannumeral0\xintiidevision{#3}{#4}{/#2[#1]}%
1627 }%
1628 \def\XINT_divmod_bpos_finish #1#2#3{#1}{#2#3}%

```

## 24.57. \xintMod

**Added at 1.2p (2017/12/05).** `\xintMod{q1}{q2}` computes  $q1 - q2 \cdot \text{floor}(q1/q2)$ . Attention that it relies on `\xintiiE{x}{e}` returning  $x$  if  $e < 0$ .

Prior to 1.2p, that macro had the meaning now attributed to `\xintModTrunc`.

**Modified at 1.3 (2018/03/01).** Modified (like `\xintAdd` and `\xintSub`) at 1.3 to use a l.c.m for final denominator.

```

1629 \def\xintMod {\romannumeral0\xintmod}%
1630 \def\xintmod #1{\expandafter\XINT_mod_a\romannumeral0\xintraw{#1}.}%
1631 \def\XINT_mod_a #1#2.#3%
1632     {\expandafter\XINT_mod_b\expandafter #1\romannumeral0\xintraw{#3}#2.}%
1633 \def\XINT_mod_b #1#2% #1 de A, #2 de B.
1634 {%
1635     \if0#2\xint_dothis{\XINT_mod_divbyzero #1#2}\fi
1636     \if0#1\xint_dothis\XINT_mod_aiszero\fi
1637     \if-#2\xint_dothis{\XINT_mod_bneg #1}\fi
1638     \xint_orthat{\XINT_mod_bpos #1#2}%
1639 }%

```

Attention to not move ModTrunc code beyond that point.

```

1640 \let\XINT_mod_divbyzero\XINT_modtrunc_divbyzero
1641 \let\XINT_mod_aiszero \XINT_modtrunc_aiszero
1642 \def\XINT_mod_bneg #1% f % -g = -((-f) % g), for g > 0
1643 {%
1644     \xintiiopp\xint_UDsignfork
1645     #1{\XINT_mod_bpos {}}%
1646     -{\XINT_mod_bpos {-#1}}%
1647     \krof
1648 }%
1649 \def\XINT_mod_bpos #1#2/#3[#4]#5/#6[#7].%
1650 {%
1651     \expandafter\XINT_mod_bpos_a
1652     \the\numexpr\ifnum#7>#4 #4\else #7\fi\expandafter.%
1653     \romannumeral0\expandafter\XINT_mod_D_b
1654     \romannumeral0\XINT_div_prepare{#3}{#6}{#3}{#3}{#6}%
1655     {#1#5}{#7-#4}{#2}{#4-#7}%
1656 }%
1657 \def\XINT_mod_D_a #1#2%
1658 {%
1659     \expandafter\XINT_mod_D_b
1660     \romannumeral0\XINT_div_prepare {#1}{#2}{#1}%
1661 }%
1662 \def\XINT_mod_D_b #1#2{\XINT_mod_D_c #2\Z}%
1663 \def\XINT_mod_D_c #1#2\Z

```

```

1664 {%
1665     \xint_gob_til_zero #1\XINT_mod_D_exit0\XINT_mod_D_a {#1#2}%
1666 }%
1667 \def\XINT_mod_D_exit0\XINT_mod_D_a #1#2#3%
1668 {%
1669     \expandafter\XINT_mod_E
1670     \romannumeral0\xintiiquo {#3}{#2}.{#2}%
1671 }%
1672 \def\XINT_mod_E #1.#2#3%
1673 {%
1674     \expandafter\XINT_mod_F
1675     \romannumeral0\xintiimul{#1}{#3}.{\xintiiQuo{#3}{#2}}{#1}%
1676 }%
1677 \def\XINT_mod_F #1.#2#3#4#5#6#7%
1678 {%
1679     {#1}{\xintiiE{\xintiiMul{#4}{#3}}{#5}}%
1680     {\xintiiE{\xintiiMul{#6}{#2}}{#7}}%
1681 }%
1682 \def\XINT_mod_bpos_a #1.#2#3#4{\xintiirem {#3}{#4}/#2[#1]}%

```

## 24.58. \xintIsOne

**Added at 1.09a (2013/09/24).** Could be more efficient. For fractions with big powers of tens, it is better to use `\xintCmp{f}{1}`. Restyled in 1.09i.

```

1683 \def\xintIsOne {\romannumeral0\xintisone }%
1684 \def\xintisone #1{\expandafter\XINT_fracisone
1685     \romannumeral0\xintrawwithzeros{#1}\Z }%
1686 \def\XINT_fracisone #1/#2\Z
1687     {\if0\xintiiCmp {#1}{#2}\xint_afterfi{ 1}\else\xint_afterfi{ 0}\fi}%

```

## 24.59. \xintGeq

```

1688 \def\xintGeq {\romannumeral0\xintgeq }%
1689 \def\xintgeq #1%
1690 {%
1691     \expandafter\XINT_fgeq\expandafter {\romannumeral0\xintabs {#1}}%
1692 }%
1693 \def\XINT_fgeq #1#2%
1694 {%
1695     \expandafter\XINT_fgeq_A \romannumeral0\xintabs {#2}#1%
1696 }%
1697 \def\XINT_fgeq_A #1%
1698 {%
1699     \xint_gob_til_zero #1\XINT_fgeq_Zii 0%
1700     \XINT_fgeq_B #1%
1701 }%
1702 \def\XINT_fgeq_Zii 0\XINT_fgeq_B #1[#2]#3[#4]{ 1}%
1703 \def\XINT_fgeq_B #1/#2[#3]#4#5/#6[#7]%
1704 {%
1705     \xint_gob_til_zero #4\XINT_fgeq_Zi 0%
1706     \expandafter\XINT_fgeq_C\expandafter
1707     {\the\numexpr #7-#3\expandafter}\expandafter

```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

1708     {\romannumeral0\xintiimul {#4#5}{#2}}%
1709     {\romannumeral0\xintiimul {#6}{#1}}%
1710 }%
1711 \def\xINT_fgeq_Zi 0#1#2#3#4#5#6#7{ 0}%
1712 \def\xINT_fgeq_C #1#2#3%
1713 {%
1714     \expandafter\xINT_fgeq_D\expandafter
1715     {#3}{#1}{#2}%
1716 }%
1717 \def\xINT_fgeq_D #1#2#3%
1718 {%
1719     \expandafter\xINT_cntSgnFork\romannumeral`&&\expandafter\xINT_cntSgn
1720     \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\xint:
1721     { 0}{\xINT_fgeq_E #2\Z {#3}{#1}}{ 1}%
1722 }%
1723 \def\xINT_fgeq_E #1%
1724 {%
1725     \xint_UDsignfork
1726     #1\xINT_fgeq_Fd
1727     -{\xINT_fgeq_Fn #1}%
1728     \krof
1729 }%
1730 \def\xINT_fgeq_Fd #1\Z #2#3%
1731 {%
1732     \expandafter\xINT_fgeq_Fe
1733     \romannumeral0\xINT_dsx_addzeros {#1}{#3;\xint:#2\xint:
1734 }%
1735 \def\xINT_fgeq_Fe #1\xint:#2#3\xint:{\xINT_geq_plusplus #2#1\xint:#3\xint:}%
1736 \def\xINT_fgeq_Fn #1\Z #2#3%
1737 {%
1738     \expandafter\xINT_fgeq_Fo
1739     \romannumeral0\xINT_dsx_addzeros {#1}{#2;\xint:#3\xint:
1740 }%
1741 \def\xINT_fgeq_Fo #1#2\xint:#3\xint:{\xINT_geq_plusplus #1#3\xint:#2\xint:}%

```

## 24.60. \xintMax

```

1742 \def\xintMax {\romannumeral0\xintmax }%
1743 \def\xintmax #1%
1744 {%
1745     \expandafter\xINT_fmax\expandafter {\romannumeral0\xintraw {#1}}%
1746 }%
1747 \def\xINT_fmax #1#2%
1748 {%
1749     \expandafter\xINT_fmax_A\romannumeral0\xintraw {#2}#1%
1750 }%
1751 \def\xINT_fmax_A #1#2/#3[#4]#5#6/#7[#8]%
1752 {%
1753     \xint_UDsignsfork
1754     #1#5\xINT_fmax_minusminus
1755     -#5\xINT_fmax_firstneg
1756     #1-\xINT_fmax_secondneg
1757     --\xINT_fmax_nonneg_a

```

```

1758 \krof
1759 #1#5{#2/#3[#4]}{#6/#7[#8]}%
1760 }%
1761 \def\XINT_fmax_minusminus --%
1762 {\expandafter\romannumeral0\XINT_fmin_nonneg_b }%
1763 \def\XINT_fmax_firstneg #1-#2#3{ #1#2}%
1764 \def\XINT_fmax_secondneg -#1#2#3{ #1#3}%
1765 \def\XINT_fmax_nonneg_a #1#2#3#4%
1766 {%
1767 \XINT_fmax_nonneg_b {#1#3}{#2#4}%
1768 }%
1769 \def\XINT_fmax_nonneg_b #1#2%
1770 {%
1771 \if0\romannumeral0\XINT_fgeq_A #1#2%
1772 \xint_afterfi{ #1}%
1773 \else \xint_afterfi{ #2}%
1774 \fi
1775 }%

```

## 24.61. \xintMaxof

1.21 protects \xintMaxof against items with non terminated \the\numexpr expressions.

1.4 renders the macro compatible with an empty argument and it also defines an accessor \XINT\_ Maxof suitable for xintexpr usage (formerly xintexpr had its own macro handling comma separated values, but it changed internal representation at 1.4).

```

1776 \def\xintMaxof {\romannumeral0\xintmaxof }%
1777 \def\xintmaxof #1{\expandafter\XINT_maxof\romannumeral`&&@#1^}%
1778 \def\XINT_Maxof{\romannumeral0\XINT_maxof}%
1779 \def\XINT_maxof#1%
1780 {%
1781 \xint_gob_til_ ^ #1\XINT_maxof_empty ^%
1782 \expandafter\XINT_maxof_loop\romannumeral0\xintraw{#1}\xint:
1783 }%
1784 \def\XINT_maxof_empty ^#1\xint:{ 0/1[0]}%
1785 \def\XINT_maxof_loop #1\xint:#2%
1786 {%
1787 \xint_gob_til_ ^ #2\XINT_maxof_e ^%
1788 \expandafter\XINT_maxof_loop
1789 \romannumeral0\xintmax{#1}{\romannumeral0\xintraw{#2}}\xint:
1790 }%
1791 \def\XINT_maxof_e ^#1\xintmax #2#3\xint:{ #2}%

```

## 24.62. \xintMin

```

1792 \def\xintMin {\romannumeral0\xintmin }%
1793 \def\xintmin #1%
1794 {%
1795 \expandafter\XINT_fmin\expandafter {\romannumeral0\xintraw {#1}}%
1796 }%
1797 \def\XINT_fmin #1#2%
1798 {%
1799 \expandafter\XINT_fmin_A\romannumeral0\xintraw {#2}#1%
1800 }%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrc*, *xintexpr*, *xinttrig*, *xintlog*

```

1801 \def\XINT_fmin_A #1#2/#3[#4]#5#6/#7[#8]%
1802 {%
1803   \xint_UDsignsfork
1804     #1#5\XINT_fmin_minusminus
1805     -#5\XINT_fmin_firstneg
1806     #1-\XINT_fmin_secondneg
1807     --\XINT_fmin_nonneg_a
1808   \krof
1809   #1#5{#2/#3[#4]}{#6/#7[#8]}%
1810 }%
1811 \def\XINT_fmin_minusminus --%
1812   {\expandafter-\romannumeral0\XINT_fmax_nonneg_b }%
1813 \def\XINT_fmin_firstneg #1-#2#3{ -#3}%
1814 \def\XINT_fmin_secondneg -#1#2#3{ -#2}%
1815 \def\XINT_fmin_nonneg_a #1#2#3#4%
1816 {%
1817   \XINT_fmin_nonneg_b {#1#3}{#2#4}%
1818 }%
1819 \def\XINT_fmin_nonneg_b #1#2%
1820 {%
1821   \if0\romannumeral0\XINT_fgeq_A #1#2%
1822     \xint_afterfi{ #2}%
1823   \else \xint_afterfi{ #1}%
1824   \fi
1825 }%

```

### 24.63. \xintMinof

1.21 protects `\xintMinof` against items with non terminated `\the\numexpr` expressions.  
 1.4 version is compatible with an empty input (empty items are handled as zero).

```

1826 \def\xintMinof {\romannumeral0\xintminof }%
1827 \def\xintminof #1{\expandafter\XINT_minof\romannumeral`&&@#1^}%
1828 \def\XINT_Minof{\romannumeral0\XINT_minof}%
1829 \def\XINT_minof#1%
1830 {%
1831   \xint_gob_til_^ #1\XINT_minof_empty ^%
1832   \expandafter\XINT_minof_loop\romannumeral0\xintraw{#1}\xint:
1833 }%
1834 \def\XINT_minof_empty ^#1\xint:{ 0/1[0]}%
1835 \def\XINT_minof_loop #1\xint:#2%
1836 {%
1837   \xint_gob_til_^ #2\XINT_minof_e ^%
1838   \expandafter\XINT_minof_loop\romannumeral0\xintmin{#1}{\romannumeral0\xintraw{#2}}\xint:
1839 }%
1840 \def\XINT_minof_e ^#1\xintmin #2#3\xint:{ #2}%

```

### 24.64. \xintCmp

```

1841 \def\xintCmp {\romannumeral0\xintcmp }%
1842 \def\xintcmp #1%
1843 {%
1844   \expandafter\XINT_fcmp\expandafter {\romannumeral0\xintraw {#1}}%
1845 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

1846 \def\XINT_fcmp #1#2%
1847 {%
1848   \expandafter\XINT_fcmp_A\romannumeral0\xintra {#2}#1%
1849 }%
1850 \def\XINT_fcmp_A #1#2/#3[#4]#5#6/#7[#8]%
1851 {%
1852   \xint_UDsignsfork
1853     #1#5\XINT_fcmp_minusminus
1854     -#5\XINT_fcmp_firstneg
1855     #1-\XINT_fcmp_secondneg
1856     --\XINT_fcmp_nonneg_a
1857   \krof
1858   #1#5{#2/#3[#4]}{#6/#7[#8]}%
1859 }%
1860 \def\XINT_fcmp_minusminus --#1#2{\XINT_fcmp_B #2#1}%
1861 \def\XINT_fcmp_firstneg #1-#2#3{ -1}%
1862 \def\XINT_fcmp_secondneg -#1#2#3{ 1}%
1863 \def\XINT_fcmp_nonneg_a #1#2%
1864 {%
1865   \xint_UDzerosfork
1866     #1#2\XINT_fcmp_zerozero
1867     0#2\XINT_fcmp_firstzero
1868     #10\XINT_fcmp_secondzero
1869     00\XINT_fcmp_pos
1870   \krof
1871   #1#2%
1872 }%
1873 \def\XINT_fcmp_zerozero #1#2#3#4{ 0}%
1874 \def\XINT_fcmp_firstzero #1#2#3#4{ -1}%
1875 \def\XINT_fcmp_secondzero #1#2#3#4{ 1}%
1876 \def\XINT_fcmp_pos #1#2#3#4%
1877 {%
1878   \XINT_fcmp_B #1#3#2#4%
1879 }%
1880 \def\XINT_fcmp_B #1/#2[#3]#4/#5[#6]%
1881 {%
1882   \expandafter\XINT_fcmp_C\expandafter
1883   {\the\numexpr #6-#3\expandafter}\expandafter
1884   {\romannumeral0\xintiimul {#4}{#2}}%
1885   {\romannumeral0\xintiimul {#5}{#1}}%
1886 }%
1887 \def\XINT_fcmp_C #1#2#3%
1888 {%
1889   \expandafter\XINT_fcmp_D\expandafter
1890   {#3}{#1}{#2}%
1891 }%
1892 \def\XINT_fcmp_D #1#2#3%
1893 {%
1894   \expandafter\XINT_cntSgnFork\romannumeral`&&\expandafter\XINT_cntSgn
1895   \the\numexpr #2+\xintLength{#3}-\xintLength{#1}\relax\xint:
1896   { -1}{\XINT_fcmp_E #2\Z {#3}{#1}}{ 1}%
1897 }%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
1898 \def\XINT_fcmp_E #1%
1899 {%
1900     \xint_UDsignfork
1901     #1\XINT_fcmp_Fd
1902     -{\XINT_fcmp_Fn #1}%
1903     \krof
1904 }%
1905 \def\XINT_fcmp_Fd #1\Z #2#3%
1906 {%
1907     \expandafter\XINT_fcmp_Fe
1908     \romannumeral0\XINT_dsx_addzeros {#1}#3;\xint:#2\xint:
1909 }%
1910 \def\XINT_fcmp_Fe #1\xint:#2#3\xint:{\XINT_cmp_plusplus #2#1\xint:#3\xint:}%
1911 \def\XINT_fcmp_Fn #1\Z #2#3%
1912 {%
1913     \expandafter\XINT_fcmp_Fo
1914     \romannumeral0\XINT_dsx_addzeros {#1}#2;\xint:#3\xint:
1915 }%
1916 \def\XINT_fcmp_Fo #1#2\xint:#3\xint:{\XINT_cmp_plusplus #1#3\xint:#2\xint:}%

```

### 24.65. \xintAbs

```
1917 \def\xintAbs {\romannumeral0\xintabs }%
1918 \def\xintabs #1{\expandafter\XINT_abs\romannumeral0\xintra #1}%

```

### 24.66. \xintOpp

```
1919 \def\xintOpp {\romannumeral0\xintopp }%
1920 \def\xintopp #1{\expandafter\XINT_opp\romannumeral0\xintra #1}%

```

### 24.67. \xintInv

Modified at 1.3d (2019/01/06).

```
1921 \def\xintInv {\romannumeral0\xintinv }%
1922 \def\xintinv #1{\expandafter\XINT_inv\romannumeral0\xintra #1}%
1923 \def\XINT_inv #1%
1924 {%
1925     \xint_UDzerominusfork
1926     #1-\XINT_inv_iszero
1927     0#1\XINT_inv_a
1928     0-{\XINT_inv_a {}}%
1929     \krof #1%
1930 }%
1931 \def\XINT_inv_iszero #1]%
1932     {\XINT_signalcondition{DivisionByZero}{Inverse of zero: inv(#1)}.}{0/1[0]}%
1933 \def\XINT_inv_a #1#2/#3[#4#5]%
1934 {%
1935     \xint_UDzerominusfork
1936     #4-\XINT_inv_expiszero
1937     0#4\XINT_inv_b
1938     0-{\XINT_inv_b -#4}%
1939     \krof #5.{#1#3/#2}%

```

```

1940 }%
1941 \def\XINT_inv_expiszero #1.#2{ #2[0]}%
1942 \def\XINT_inv_b #1.#2{ #2[#1]}%

```

## 24.68. \xintSgn

```

1943 \def\xintSgn {\romannumeral0\xintsfn }%
1944 \def\xintsfn #1{\expandafter\XINT_sgn\romannumeral0\xintraw {#1}\xint:}%

```

## 24.69. \xintSignBit

Added at 1.41 (2022/05/29).

```

1945 \def\xintSignBit {\romannumeral0\xintsignbit }%
1946 \def\xintsignbit #1{\expandafter\XINT_signbit\romannumeral0\xintraw {#1}\xint:}%
1947 \def\XINT_signbit #1#2\xint:
1948 {%
1949   \xint_UDzerominusfork
1950   #1-{ 0}%
1951   0#1{ 1}%
1952   0-{ 0}%
1953   \krof
1954 }%

```

## 24.70. \xintGCD

**Modified at 1.4 (2020/01/31).** They replace the former [xintgcd](#) macros of the same names which truncated to integers their arguments. Fraction-producing [gcd\(\)](#) and [lcm\(\)](#) functions were available since 1.3d [xintexpr](#), with non-public support macros handling comma separated values.

**Modified at 1.4d (2021/03/29).** Somewhat strangely [\xintGCD](#) was formerly [\xintGCDof](#) used with only two arguments, as the latter directly implemented a fraction gcd algorithm using [\xintMod](#) repeatedly for two arguments.

Now [\xintGCD](#) contains the pairwise gcd routine and [\xintGCDof](#) is only a wrapper. And the pairwise gcd is reduced to integer-only computations to hopefully reduce fraction overhead.

Each input is filtered via [\xintPIrr](#) and [\xintREZ](#) to reduce size of manipulate integers in algebra.

But hesitation about applying [\xintPIrr](#) to output, and/or [\xintREZ](#). (as it is applied on input).

But as the code is now used for fractional lcm's we actually need to do some reduction of output else lcm's of integers will not be necessarily printed by [\xinteval](#) as integers.

Well finally I apply [\xintIrr](#) (but not [\xintREZ](#) to output). Hesitations here (thinking of inputs with large [n] parts, the output will have many zeros). So I do this only for the user macro but the core routine as used by [\xintGCDof](#) will not do it.

Also at 1.4d the code uses [\expanded](#).

```

1955 \def\xintGCD {\romannumeral0\xintgcd}%
1956 \def\xintgcd #1%
1957 {%
1958   \expandafter\XINT_fgcd_in
1959   \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
1960 }%
1961 \def\XINT_fgcd_in #1#2\xint:#3%
1962 {%
1963   \expandafter\XINT_fgcd_out
1964   \romannumeral0\expandafter\XINT_fgcd_chkzeros\expandafter#1%

```

```

1965 \romannumeral0\xintrez{\xintPIrr{\xintAbs{#3}}}\xint:#1#2\xint:
1966 }%
1967 \def\xINT_fgcd_out#1[#2]{\xintirr{#1[#2]}[0]}%
1968 \def\xINT_fgcd_chkzeros #1#2%
1969 {%
1970 \xint_UDzerofork
1971 #1\xINT_fgcd_aiszero
1972 #2\xINT_fgcd_biszero
1973 0\xINT_fgcd_main
1974 \krof #2%
1975 }%
1976 \def\xINT_fgcd_aiszero #1\xint:#2\xint:{ #1}%
1977 \def\xINT_fgcd_biszero #1\xint:#2\xint:{ #2}%
1978 \def\xINT_fgcd_main #1/#2[#3]\xint:#4/#5[#6]\xint:
1979 {%
1980 \expandafter\xINT_fgcd_a
1981 \romannumeral0\xINT_gcd_loop #2\xint:#5\xint:\xint:
1982 #2\xint:#5\xint:#1\xint:#4\xint:#3.#6.%
1983 }%
1984 \def\xINT_fgcd_a #1\xint:#2\xint:
1985 {%
1986 \expandafter\xINT_fgcd_b
1987 \romannumeral0\xintiico{#2}{#1}\xint:#1\xint:#2\xint:
1988 }%
1989 \def\xINT_fgcd_b #1\xint:#2\xint:#3\xint:#4\xint:#5\xint:#6\xint:#7.#8.%
1990 {%
1991 \expanded{%
1992 \xintiigcd{\xintiiE{\xintiiMul{#5}{\xintiiQuo{#4}{#2}}}{#7-#8}}%
1993 {\xintiiE{\xintiiMul{#6}{#1}}{#8-#7}}%
1994 /\xintiiMul{#1}{#4}%
1995 [\ifnum#7>#8 #8\else #7\fi]%
1996 }%
1997 }%

```

## 24.71. \xintGCDof

**Modified at 1.4 (2020/01/31).** This inherits from former non public [xintexpr](#) macro called [\xintGCDof:csv](#), which handled comma separated items.

It handles fractions presented as braced items and is the support macro for the `gcd()` function in [\xintexpr](#) and [\xintfloatexpr](#). The support macro for the `gcd()` function in [\xintiexpr](#) is [\xintiigCDof](#), from [xint](#).

An empty input is allowed but I have some hesitations on the return value of 1.

**Modified at 1.4d (2021/03/29).** Sadly the 1.4 version had multiple problems:

- broken if first argument vanished,
- broken if some argument was not in strict format, for example had leading chains of signs or zeros ([\xintGCDof{2}{03}](#)). This bug originates in the fact the original macro was used only in [xintexpr](#) sanitized context.

Also, output is now always an irreducible fraction (ending with [\[0\]](#)).

```

1998 \def\xintGCDof {\romannumeral0\xintgcdof}%
1999 \def\xintgcdof #1{\expandafter\xINT_fgcdof\romannumeral`&&@#1^}%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```

2000 \def\XINT_GCDof{\romannumeral0\XINT_fgcdof}%
2001 \def\XINT_fgcdof #1%
2002 {%
2003   \expandafter\XINT_fgcdof_chkempty\romannumeral`&&@#1\xint:
2004 }%
2005 \def\XINT_fgcdof_chkempty #1%
2006 {%
2007   \xint_gob_til_#1\XINT_fgcdof_empty ^\XINT_fgcdof_in #1%
2008 }%
2009 \def\XINT_fgcdof_empty #1\xint:{ 1/1[0]}% hesitation, should it be infinity? 0?
2010 \def\XINT_fgcdof_in #1\xint:
2011 {%
2012   \expandafter\XINT_fgcd_out
2013   \romannumeral0\expandafter\XINT_fgcdof_loop
2014   \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
2015 }%
2016 \def\XINT_fgcdof_loop #1\xint:#2%
2017 {%
2018   \expandafter\XINT_fgcdof_chkend\romannumeral`&&@#2\xint:#1\xint:\xint:
2019 }%
2020 \def\XINT_fgcdof_chkend #1%
2021 {%
2022   \xint_gob_til_#1\XINT_fgcdof_end ^\XINT_fgcdof_loop_pair #1%
2023 }%
2024 \def\XINT_fgcdof_end #1\xint:#2\xint:\xint:{ #2}%
2025 \def\XINT_fgcdof_loop_pair #1\xint:#2%
2026 {%
2027   \expandafter\XINT_fgcdof_loop
2028   \romannumeral0\expandafter\XINT_fgcd_chkzeros\expandafter#2%
2029   \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:#2%
2030 }%

```

### 24.72. \xintLCM

Same comments as for \xintGCD. Entirely redone for 1.4d. Well, actually we can express it in terms of fractional gcd.

```

2031 \def\xintLCM {\romannumeral0\xintlcm}%
2032 \def\xintlcm #1%
2033 {%
2034   \expandafter\XINT_flcm_in
2035   \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
2036 }%
2037 \def\XINT_flcm_in #1#2\xint:#3%
2038 {%
2039   \expandafter\XINT_fgcd_out
2040   \romannumeral0\expandafter\XINT_flcm_chkzeros\expandafter#1%
2041   \romannumeral0\xintrez{\xintPIrr{\xintAbs{#3}}}\xint:#1#2\xint:
2042 }%
2043 \def\XINT_flcm_chkzeros #1#2%
2044 {%
2045   \xint_UDzerofork
2046   #1\XINT_flcm_zero

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrc*, *xintexpr*, *xinttrig*, *xintlog*

```
2047      #2\XINT_flcm_zero
2048      0\XINT_flcm_main
2049      \krof #2%
2050 }%
2051 \def\XINT_flcm_zero #1\xint:#2\xint:{ 0/1[0]}%
2052 \def\XINT_flcm_main #1/#2[#3]\xint:#4/#5[#6]\xint:
2053 {%
2054     \xintinv
2055     {%
2056         \romannumeral0\XINT_fgcd_main #2/#1[-#3]\xint:#5/#4[-#6]\xint:
2057     }%
2058 }%
```

### 24.73. \xintLCMof

See comments for `\xintGCDof`. `xint` provides the integer only `\xintiilCMof`.

**Modified at 1.4d (2021/03/29).** Sadly, although a public `xintfrac` macro, it did not (since 1.4) sanitize its arguments like other `xintfrac` macros.

```
2059 \def\xintLCMof {\romannumeral0\xintlcmof}%
2060 \def\xintlcmof #1{\expandafter\XINT_flcmof\romannumeral`&&@#1^}%
2061 \def\XINT_LCMof{\romannumeral0\XINT_flcmof}%
2062 \def\XINT_flcmof #1%
2063 {%
2064     \expandafter\XINT_flcmof_chkempty\romannumeral`&&@#1\xint:
2065 }%
2066 \def\XINT_flcmof_chkempty #1%
2067 {%
2068     \xint_gob_til_`^#1\XINT_flcmof_empty ^\XINT_flcmof_in #1%
2069 }%
2070 \def\XINT_flcmof_empty #1\xint:{ 0/1[0]}% hesitation
2071 \def\XINT_flcmof_in #1\xint:
2072 {%
2073     \expandafter\XINT_fgcd_out
2074     \romannumeral0\expandafter\XINT_flcmof_loop
2075     \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:
2076 }%
2077 \def\XINT_flcmof_loop #1\xint:#2%
2078 {%
2079     \expandafter\XINT_flcmof_chkend\romannumeral`&&@#2\xint:#1\xint:\xint:
2080 }%
2081 \def\XINT_flcmof_chkend #1%
2082 {%
2083     \xint_gob_til_`^#1\XINT_flcmof_end ^\XINT_flcmof_loop_pair #1%
2084 }%
2085 \def\XINT_flcmof_end #1\xint:#2\xint:\xint:{ #2}%
2086 \def\XINT_flcmof_loop_pair #1\xint:#2%
2087 {%
2088     \expandafter\XINT_flcmof_chkzero
2089     \romannumeral0\expandafter\XINT_flcm_chkzeros\expandafter#2%
2090     \romannumeral0\xintrez{\xintPIrr{\xintAbs{#1}}}\xint:#2%
2091 }%
2092 \def\XINT_flcmof_chkzero #1%
```

```

2093 {%
2094     \xint_gob_til_zero#1\XINT_flgcmof_zero0\XINT_flgcmof_loop#1%
2095 }%
2096 \def\XINT_flgcmof_zero#1^{ 0/1[0]}%

```

## 24.74. Floating point macros

For a long time the float routines dating back to releases 1.07/1.08a (May–June 2013) were not modified.

Since 1.2f (March 2016) the four operations first round their arguments to `\xinttheDigits`-floats (or `P`-floats), not `(\xinttheDigits+2)`-floats or `(P+2)`-floats as was the case with earlier releases.

The four operations addition, subtraction, multiplication, division have always produced the correct rounding of the theoretical exact value to `P` or `\xinttheDigits` digits when the inputs are decimal numbers with at most `P` digits, and arbitrary decimal exponent part.

From 1.08a to 1.2j, `\xintFloat` (and `\XINTinFloat` which is used to parse inputs to other float macros) handled a fractional input `A/B` via an initial replacement to `A'/B'` where `A'` and `B'` were `A` and `B` truncated to `Q+2` digits (where asked-for precision is `Q`), and then they correctly rounded `A/B` to `Q` digits. But this meant that this rounding of the input could differ (by up to one unit in the last place) from the correct rounding of the original `A/B` to the asked-for number of digits (which until 1.2f in uses as auxiliary to the macros for the basic operations was 2 more than the prevailing precision).

Since 1.2k all inputs are correctly rounded to the asked-for number of digits (this was, I think, the case in the 1.07 release -- there are no code comments -- but was, afaicr, not very efficiently done, and this is why the 1.08a release opted for truncation of the numerator and denominator.)

Notice that in float expressions, the `/` is treated as operator, hence the above discussion makes a difference only for the special input form `qfloat(A/B)` or for an `\xintexpr A/B\relax` embedded in the float expression, with `A` or `B` having more digits than the prevailing float precision.

Internally there is no inner representation of `P`-floats as such !!!!!

The input parser will again compute the length of the mantissa on each use !!! This is obviously something that must be improved upon before implementation of higher functions.

Currently, special tricks are used to quickly recognize inputs having no denominators, or fractions whose numerators and denominators are not too long compared to the target precision `P`, and in particular `P`-floats or quotients of two such.

Another long-standing issue is that float multiplication will first compute the `2P` or `2P-1` digits of the exact product, and then round it to `P` digits. This is sub-optimal for large `P` particularly as the multiplication algorithm is basically the schoolbook one, hence worse than quadratic in the `TEX` implementation which has extra cost of fetching long sequences of tokens.

Changes at 1.4e (done 2021/04/15; undone 2021/04/29)

Macros named `\XINTinFloat<name>` are not public user-level but were designed a long time ago for `\xintfloatexpr` context as a very preliminary step towards attempting to preserve some internal format, here `A[N]` type.

When `<name>` is lowercased it means it needs a `\romannumeral0` trigger (`\XINTinfloatS` keeps an uppercase `S`).

Most were coded to check for an optional argument `[D]`, and to use `D=\XINTdigits` in its place if absent but it turned out only `\XINTinfloatpow`, `\XINTinfloatmul`, `\XINTinfloatadd` were actually used with an optional argument and this happened only in macros from the very old `xintseries.sty`, so I changed all of them to not check for optional argument `[D]` anymore, keeping only some private

interface for the `xintseries.sty` use case. Some required being used with [D], some still had names ending in "digits" indicating they would use `\XINTdigits` always.

Indeed basically all algebra is done "exactly" and the [D] governs rules of float-rounding on input and output.

During development of 1.4e we fleetingly experimented with letting the value used in place of D be `\XINTdigitsx` to 1.4e, i.e. `\XINTdigits` with guard digits, a situation which was motivated by the implementation of trigonometrical functions at high level, i.e. using `\xintdeffloatfunc` which had no mechanism to make intermediate calculations with guard digits.

Simply doing everything "as is" but with 2 guard digits proved very good (surprisingly efficient, even) to the trigonometrical functions. However using them systematically raises many issues (for example, the correct rounding at P digits is destroyed if we obtain it a D=P+2 then round from P+2 to P digits so we definitely can not do this as default, so some interface is needed to define intermediate functions only using such guard digits and keeping them in their output).

Finally, an approach limited to the `xinttrig.sty` scope was used and I removed all `\XINTdigits`, `x` related matters from 1.4e. But this left some modifications of the interfaces of the "float" macros here which this list tries to document, mainly for the author's benefit.

Macros always using `\XINTdigits` and now not allowing [P] option

`\XINTinFloatAdd`  
`\XINTinFloatSub`  
`\XINTinFloatMul`  
`\XINTinFloatSqr`  
`\XINTinFloatInv`  
`\XINTinFloatDiv`  
`\XINTinFloatPow`  
`\XINTinFloatPower`  
`\XINTinFloatPFactorial`  
`\XINTinFloatBinomial`

Macros which already did not allow [P] option prior to 1.4e refactoring

`\XINTinFloatFrac` (renamed from `\XINTinFloatFracdigits`)  
`\XINTinFloatE`  
`\XINTinFloatMod`  
`\XINTinFloatDivFloor`  
`\XINTinFloatDivMod`

Macros requiring a [P]. Some of the "\_wopt" named macros are renamings of macros formerly requiring [P].

`\XINTinFloat`  
`\XINTinFloatS`  
`\XINTFloatiLogTen`  
`\XINTinRandomFloatS` (this one has only the [P] mandatory argument)  
`\XINTinFloatFac`  
`\XINTinFloatSqrt`  
`\XINTinFloatAdd_wopt`, `\XINTinfloatadd_wopt`  
`\XINTinFloatSub_wopt`, `\XINTinfloatsub_wopt`  
`\XINTinFloatMul_wopt`, `\XINTinfloatmul_wopt`  
`\XINTinFloatSqr_wopt`  
`\XINTinfloatpow_wopt` (not `FloatPow`)  
`\XINTinFloatDiv_wopt`  
`\XINTinFloatInv_wopt`

Specially named macros indicating usage of `\XINTdigits`

`\XINTinFloatdigits`  
`\XINTinFloatSdigits`  
`\XINTFloatiLogTendigits`

```

\XINTinRandomFloatSdigits
\XINTinFloatFacdigits
\XINTinFloatSqrtdigits

```

## 24.75. \xintDigits, \xintSetDigits

Modified at 1.3 (2018/03/01). 1.3f allows `\xintDigits=` in place of `\xintDigits:=` syntax. It defines `\xintDigits*[:]=` which reloads `xinttrig.sty`. Perhaps this should be default, well.

During 1.4e development I added an interface for guard digits, but I decided to drop inclusion from 1.4e release because there were pending issues both in documentation and functionalities for which I did not have time left.

1.4e fixes the issue that `\xinttheDigits` could not be used in the right hand side of `\xintDigits[*][:]=...` or inside the argument to `\xintSetDigits`.

```

2097 \mathchardef\XINTdigits 16
2098 \chardef\XINTguarddigits 0
2099 \def\xinttheDigits {\number\XINTdigits}%
2100 %\def\xinttheGuardDigits{\number\XINTguarddigits}%
2101 \def\xinttheGuardDigits{0}% in case used in some of my test files
2102 \def\xintDigits #1={\afterassignment\xintDigits_i\mathchardef\XINT_digits=}%
2103 \def\xintDigits_i#1%
2104 {%
2105     \let\XINTdigits\XINT_digits
2106 }%
2107 \def\xintSetDigits #1%
2108 {%
2109     \mathchardef\XINT_digits=\numexpr#1\relax
2110     \let\XINTdigits=\XINT_digits
2111 }%

```

## 24.76. \xintFloat, \xintFloatZero

1.2f and 1.2g brought some refactoring which resulted in faster treatment of decimal inputs. 1.2i dropped use of some old routines dating back to pre 1.2 era in favor of more modern `\xintDSRr` for rounding. Then 1.2k improves again the handling of denominators B with few digits.

But the main change with 1.2k is a complete rewrite of the  $B > 1$  case in order to achieve again correct rounding in all cases.

The original version from 1.07 (May 2013) computed the exact rounding to P digits for all inputs. But from 1.08 on (June 2013), the macro handled A/B input by first truncating both A and B to at most P+2 digits. This meant that decimal input (arbitrarily long, with scientific part) was correctly rounded, but in case of fractional input there could be up to 0.6 unit in the last place difference of the produced rounding to the input, hence the output could differ from the correct rounding.

Example with 16 digits (the default): `\xintFloat {1/17597472569900621233}`

with `xintfrac 1.07`: 5.682634230727187e-20

with `xintfrac 1.08b--1.2j`: 5.682634230727188e-20

with `xintfrac 1.2k`: 5.682634230727187e-20

The exact value is 5.682634230727187499924124...e-20, showing that 1.07 and 1.2k produce the correct rounding.

Currently the code ends in a more costly branch in about 1 case among 500, where it does some extra operations (a multiplication in particular). There is a free parameter delta (here set at 4), I have yet to make some numerical explorations, to see if it could be favorable to set it to a higher value (with delta=5, there is only 1 exceptional case in 5000, etc...).

I have always hesitated about the policy of printing  $10.00\dots 0$  in case of rounding upwards to the next power of ten. Already since 1.2f `\XINTinFloat` always produced a mantissa with exactly  $P$  digits (except for the zero value). Starting with 1.2k, `\xintFloat` drops this habit of printing  $10.00\dots 0$  in such cases. Side note: the rounding-up detection worked when the input  $A/B$  was with numerator  $A$  and denominator  $B$  having each less than  $P+2$  digits, or with  $B=1$ , else, it could happen that the output was a power of ten but not detected to be a rounding up of the original fraction. The value was ok, but printed  $1.0\dots 0eN$  with  $P-1$  zeroes, not  $10.0\dots 0e(N-1)$ .

I decided it was not worth the effort to enhance the algorithm to detect with 100% fiability all cases of rounding up to next power of ten, hence 1.2k dropped this.

To avoid duplication of code, and any extra burden on `\XINTinFloat`, which is the macro used internally by the float macros for parsing their inputs, we simply make now `\xintFloat` a wrapper of `\XINTinFloat`.

```

2112 \def\xintFloatZero{0.0e0}% 1.4k breaking change. Replaces hard-coded 0.e0
2113 \def\xintFloat {\romannumeral0\xintfloat}%
2114 \def\xintfloat #1{\XINT_float_chkopt #1\xint:}%
2115 \def\XINT_float_chkopt #1%
2116 {%
2117   \ifx [#1\expandafter\XINT_float_opt
2118     \else\expandafter\XINT_float_noopt
2119   \fi #1%
2120}%
2121 \def\XINT_float_noopt #1\xint:%
2122 {%
2123   \expandafter\XINT_float_post
2124   \romannumeral0\XINTinfloat[\XINTdigits]{#1}\XINTdigits.%
2125}%
2126 \def\XINT_float_opt [\xint:
2127 {%
2128   \expandafter\XINT_float_opt_a\the\numexpr
2129}%
2130 \def\XINT_float_opt_a #1]#2%
2131 {%
2132   \expandafter\XINT_float_post
2133   \romannumeral0\XINTinfloat[#1]{#2}#1.%
2134}%
2135 \def\XINT_float_post #1%
2136 {%
2137   \xint_UDzerominusfork
2138   #1-\XINT_float_zero
2139   0#1\XINT_float_neg
2140   0-\XINT_float_pos
2141   \krof #1%
2142}%[
2143 \def\XINT_float_zero #1]#2.{\expanded{ \xintFloatZero}}%
2144 \def\XINT_float_neg-{\expandafter-\romannumeral0\XINT_float_pos}%
2145 \def\XINT_float_pos #1#2[#3]#4.%
2146 {%
2147   \expandafter\XINT_float_pos_done\the\numexpr#3+#4-\xint_c_i.#1.#2;%
2148}%
2149 \def\XINT_float_pos_done #1.#2;{ #2e#1}%

```

## 24.77. \xintFloatBraced

**Added at 1.41 (2022/05/29).** Je ne le fais pas comme un wrapper au-dessus de `\xintFloat` car c'est pénible avec argument optionnel donc finalement on est obligé de rajouter overhead comme ici.

Hésitation si on obéit à `\xintFloatZero` ou pas. Finalement non.

Hésitation si on renvoie avec séparateur décimal ou pas.

Hésitation si on met l'exposant scientifique en premier.

Hésitation si on sépare le signe pour le mettre en premier.

Hésitation si on renvoie un exposant pour mantisse normalisée ou pas normalisée.

Finalement je décide {signe}{exposant}{mantisse sans point décimal}. Avec en fait 0 ou 1 pour signe (mais ce sign bit mais ça n'a pas grand sens en décimal...). Non finalement mantisse avec point décimal.

```

2150 \def\xintFloatBraced{\romannumeral0\xintfloatbraced }%
2151 \def\xintfloatbraced#1{\XINT_floatbr_chkopt #1\xint:}%
2152 \def\XINT_floatbr_chkopt #1%
2153 {%
2154     \ifx [#1\expandafter\XINT_floatbr_opt
2155         \else\expandafter\XINT_floatbr_noopt
2156     \fi #1%
2157 }%
2158 \def\XINT_floatbr_noopt #1\xint:%
2159 {%
2160     \expandafter\XINT_floatbr_post
2161     \romannumeral0\XINTinfloat[\XINTdigits]{#1}\XINTdigits.%
2162 }%
2163 \def\XINT_floatbr_opt [\xint:
2164 {%
2165     \expandafter\XINT_floatbr_opt_a\the\numexpr
2166 }%
2167 \def\XINT_floatbr_opt_a #1]#2%
2168 {%
2169     \expandafter\XINT_floatbr_post
2170     \romannumeral0\XINTinfloat[#1]{#2}#1.%
2171 }%
2172 \def\XINT_floatbr_post #1%
2173 {%
2174     \xint_UDzerominusfork
2175     #1-\XINT_floatbr_zero
2176     0#1\XINT_floatbr_neg
2177     0-\XINT_floatbr_pos
2178     \krof #1%
2179 }%

```

Hésitation à faire

```
\def\XINT_floatbr_zero #1]#2.{\expandafter\XINT_floatbr_zero_a\xintFloatZero e0e\relax}
```

```
\def\XINT_floatbr_zero_a#1e#2e#3\relax{{#1}{#2}}
```

Finalement non. Et même je décide de renvoyer autant de zéros que P. De plus depuis j'ai opté pour {sign bit}{exposant}{mantisse} Hésitation si mantisse avec ou sans le séparateur décimal. Est-ce que je devrais mettre plutôt -0+ au début?

```

2180 \def\XINT_floatbr_zero #1]#2.{\expanded{{0}{0}{0.\xintReplicate{#2-\xint_c_i}0}}}%
2181 \def\XINT_floatbr_neg-{\expandafter\XINT_floatbr_neg_a\romannumeral0\XINT_floatbr_pos}%
2182 \def\XINT_floatbr_neg_a#1{{1}}%
2183 \def\XINT_floatbr_pos #1#2[#3]#4.%

```

```

2184 {%
2185     \expanded{{0}}{\the\numexpr#3+#4-\xint_c_i}}{#1.#2}%
2186 }%

```

## 24.78. \XINTinFloat, \XINTinFloatS

This routine is like [\xintFloat](#) but produces an output of the shape A[N] which is then parsed faster as input to other float macros. Float operations in [\xintfloatexpr](#)...[\relax](#) use internally this format.

It must be used in form [\XINTinFloat](#)[P]{f}: the optional [P] is mandatory.

Since 1.2f, the mantissa always has exactly P digits even in case of rounding up to next power of ten. This simplifies other routines.

(but the zero value must always be checked for, as it outputs 0[0])

1.2g added a variant [\XINTinFloatS](#) which, in case of decimal input with less than the asked for precision P will not add extra zeros to the mantissa. For example it may output 2[0] even if P=500, rather than the canonical representation 200...000[-499]. This is how [\xintFloatMul](#) and [\xintFloatDiv](#) parse their inputs, which speeds-up follow-up processing. But [\xintFloatAdd](#) and [\xintFloatSub](#) still use [\XINTinFloat](#) for parsing their inputs; anyway this will have to be changed again when inner structure will carry upfront at least the length of mantissa as data.

Each time [\XINTinFloat](#) is called it at least computes a length. Naturally if we had some format for floats that would be dispensed of...

something like <letterP><length of mantissa>.mantissa.exponent, etc... not yet.

Since 1.2k, [\XINTinFloat](#) always correctly rounds its argument, even if it is a fraction with very big numerator and denominator. See the discussion of [\xintFloat](#).

```

2187 \def\xintFloat {\romannumeral0\xintinfloat }%
2188 \def\xintinfloat
2189     {\expandafter\xint_infloat_clean\romannumeral0\xint_infloat}%

```

Attention que ici le fait que l'on grabbe #1 est important car il pourrait y avoir un zéro (en particulier dans le cas où input est nul).

```

2190 \def\xint_infloat_clean #1%
2191     {\if #1!\xint_dothis\xint_infloat_clean_a\fi\xint_orthat{ }#1}%

```

Ici on ajoute les zeros pour faire exactement avec P chiffres. Car le #1 = P - L avec L la longueur de #2, (ou plutôt de abs(#2), car ici le #2 peut avoir un signe) et L < P

```

2192 \def\xint_infloat_clean_a !#1.#2[#3]%
2193 {%
2194     \expandafter\xint_infloat_done
2195     \the\numexpr #3-#1\expandafter.%
2196     \romannumeral0\xint_dsx_addzeros {#1}#2;%
2197 }%
2198 \def\xint_infloat_done #1.#2;{ #2[#1]}%

```

variant which allows output with shorter mantissas.

```

2199 \def\xintFloatS {\romannumeral0\xintinfloatS}%
2200 \def\xintinfloatS
2201     {\expandafter\xint_infloatS_clean\romannumeral0\xint_infloat}%
2202 \def\xint_infloatS_clean #1%
2203     {\if #1!\xint_dothis\xint_infloatS_clean_a\fi\xint_orthat{ }#1}%
2204 \def\xint_infloatS_clean_a !#1.{ }%

```

début de la routine proprement dite, l'argument optionnel est obligatoire.

```

2205 \def\xint_infloat [#1]##2%
2206 {%
2207     \expandafter\xint_infloat_a\the\numexpr #1\expandafter.%

```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
2208 \romannumeral0\XINT_infrac% {#2}%
2209 }%
      #1=P, #2=n, #3=A, #4=B.
2210 \def\XINT_infloat_a #1.#2#3#4%
2211 {%
      micro boost au lieu d'utiliser \XINT_isOne{#4}, mais pas bon style.
2212 \if1\XINT_is_One#4XY%
2213 \expandafter\XINT_infloat_sp
2214 \else\expandafter\XINT_infloat_fork
2215 \fi #3.{#1}{#2}{#4}%
2216 }%
      Special quick treatment of B=1 case (1.2f then again 1.2g.)
      maintenant: A.{P}{N}{1} Il est possible que A soit nul.
2217 \def\XINT_infloat_sp #1%
2218 {%
2219 \xint_UDzerominusfork
2220 #1-\XINT_infloat_spzero
2221 0#1\XINT_infloat_spneg
2222 0-\XINT_infloat_sppos
2223 \krof #1%
2224 }%
      Attention surtout pas 0/1[0] ici.
2225 \def\XINT_infloat_spzero 0.#1#2#3{ 0[0]}%
2226 \def\XINT_infloat_spneg-%
2227 {\expandafter\XINT_infloat_spnegend\romannumeral0\XINT_infloat_sppos}%
2228 \def\XINT_infloat_spnegend #1%
2229 {\if#1!\expandafter\XINT_infloat_spneg_needzeros\fi -#1}%
2230 \def\XINT_infloat_spneg_needzeros -!#1.{!#1.-}%
      in: A.{P}{N}{1}
      out: P-L.A.P.N.
2231 \def\XINT_infloat_sppos #1.#2#3#4%
2232 {%
2233 \expandafter\XINT_infloat_sp_b\the\numexpr#2-\xintLength{#1}.#1.#2.#3.%
2234 }%
      #1= P-L. Si c'est positif ou nul il faut retrancher #1 à l'exposant, et ajouter autant de zéros.
      On regarde premier token. P-L.A.P.N.
2235 \def\XINT_infloat_sp_b #1%
2236 {%
2237 \xint_UDzerominusfork
2238 #1-\XINT_infloat_sp_quick
2239 0#1\XINT_infloat_sp_c
2240 0-\XINT_infloat_sp_needzeros
2241 \krof #1%
2242 }%
      Ici P=L. Le cas usuel dans \xintfloatexpr.
2243 \def\XINT_infloat_sp_quick 0.#1.#2.#3.{ #1[#3]}%
      Ici #1=P-L est >0. L'exposant sera N-(P-L). #2=A. #3=P. #4=N.
      18 mars 2016. En fait dans certains contextes il est sous-optimal d'ajouter les zéros. Par
      exemple quand c'est appelé par la multiplication ou la division, c'est idiot de convertir 2 en
```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

200000...00000[-499]. Donc je redéfinis addzeros en needzeros. Si on appelle sous la forme `\XINTinFloatS`, on ne fait pas l'addition de zeros.

```

2244 \def\XINT_infloat_sp_needzeros #1.#2.#3.#4.{!#1.#2[#4]}%
      L-P=#1.A=#2#3.P=#4.N=#5.
      Ici P<L. Il va falloir arrondir. Attention si on va à la puissance de 10 suivante. En #1 on a
      L-P qui est >0. L'exposant final sera N+L-P, sauf dans le cas spécial, il sera alors N+L-P+1.
      L'ajustement final est fait par \XINT_infloat_Y.
2245 \def\XINT_infloat_sp_c -#1.#2#3.#4.#5.%
2246 {%
2247   \expandafter\XINT_infloat_Y
2248   \the\numexpr #5+#1\expandafter.%
2249   \romannumeral0\expandafter\XINT_infloat_sp_round
2250   \romannumeral0\XINT_split_fromleft
2251   (\xint_c_i+#4).#2#3\xint_bye2345678\xint_bye..#2%
2252 }%
2253 \def\XINT_infloat_sp_round #1.#2.%
2254 {%
2255   \XINT_dsrr#1\xint_bye\xint_Bye3456789\xint_bye/\xint_c_x\relax.%
2256 }%

      General branch for A/B with B>1 inputs. It achieves correct rounding always since 1.2k (done
      January 2, 2017.) This branch is never taken for A=0 because \XINT_infrac will have returned B=1
      then.
2257 \def\XINT_infloat_fork #1%
2258 {%
2259   \xint_UDsignfork
2260   #1\XINT_infloat_J
2261   -\XINT_infloat_K
2262   \krof #1%
2263 }%
2264 \def\XINT_infloat_J-{\expandafter-\romannumeral0\XINT_infloat_K}%
      A.{P}{n}{B} avec B>1.
2265 \def\XINT_infloat_K #1.#2%
2266 {%
2267   \expandafter\XINT_infloat_L
2268   \the\numexpr\xintLength{#1}\expandafter.\the\numexpr #2+\xint_c_iv.{#1}{#2}%
2269 }%
      |A|.P+4.{A}{P}{n}{B}. We check if A already has length <= P+4.
2270 \def\XINT_infloat_L #1.#2.%
2271 {%
2272   \ifnum #1>#2
2273     \expandafter\XINT_infloat_Ma
2274   \else
2275     \expandafter\XINT_infloat_Mb
2276   \fi #1.#2.%
2277 }%
      |A|.P+4.{A}{P}{n}{B}. We will keep only the first P+4 digits of A, denoted A'' in what follows.
      output: u=-0.A''.junk.P+4.|A|. {A}{P}{n}{B}
2278 \def\XINT_infloat_Ma #1.#2.#3%
2279 {%
2280   \expandafter\XINT_infloat_MtoN\expandafter-\expandafter0\expandafter.%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

2281 \romannumeral0\XINT_split_fromleft#2.#3\xint_bye2345678\xint_bye..%
2282 #2.#1.{#3}%
2283 }%
    |A|.P+4.{A}{P}{n}{B}.
    Here A is short. We set  $u = P+4 - |A|$ , and  $A' = A$  ( $A' = 10^u A$ )
    output:  $u.A'..P+4. |A|. \{A\} \{P\} \{n\} \{B\}$ 
2284 \def\XINT_infloat_Mb #1.#2.#3%
2285 {%
2286 \expandafter\XINT_infloat_MtoN\the\numexpr#2-#1.%
2287 #3..#2.#1.{#3}%
2288 }%
    input  $u.A'..junk.P+4. |A|. \{A\} \{P\} \{n\} \{B\}$ 
    output  $|B|.P+4. \{B\} u.A'..P. |A|. n. \{A\} \{B\}$ 
2289 \def\XINT_infloat_MtoN #1.#2.#3.#4.#5.#6#7#8#9%
2290 {%
2291 \expandafter\XINT_infloat_N
2292 \the\numexpr\xintLength{#9}.#4.{#9}#1.#2.#7.#5.#8.{#6}{#9}%
2293 }%
2294 \def\XINT_infloat_N #1.#2.%
2295 {%
2296 \ifnum #1>#2
2297 \expandafter\XINT_infloat_Oa
2298 \else
2299 \expandafter\XINT_infloat_Ob
2300 \fi #1.#2.%
2301 }%
    input  $|B|.P+4. \{B\} u.A'..P. |A|. n. \{A\} \{B\}$ 
    output  $v=-0.B'..junk. |B|. u.A'..P. |A|. n. \{A\} \{B\}$ 
2302 \def\XINT_infloat_Oa #1.#2.#3%
2303 {%
2304 \expandafter\XINT_infloat_P\expandafter-\expandafter0\expandafter.%
2305 \romannumeral0\XINT_split_fromleft#2.#3\xint_bye2345678\xint_bye..%
2306 #1.%
2307 }%
    output  $v=P+4-|B| \geq 0.B'..junk. |B|. u.A'..P. |A|. n. \{A\} \{B\}$ 
2308 \def\XINT_infloat_Ob #1.#2.#3%
2309 {%
2310 \expandafter\XINT_infloat_P\the\numexpr#2-#1.#3..#1.%
2311 }%
    input  $v.B'..junk. |B|. u.A'..P. |A|. n. \{A\} \{B\}$ 
    output  $Q1.P. |B|. |A|. n. \{A\} \{B\}$ 
     $Q1 = \text{division euclidienne de } A'..10^{u-v+P+3} \text{ par } B'.$ 
    Special detection of cases with A and B both having length at most P+4: this will happen when
    called from \xintFloatDiv as A and B (produced then via \XINTinFloatS) will have at most P digits.
    We then only need integer division with P+1 extra zeros, not P+3.
2312 \def\XINT_infloat_P #1#2.#3.#4.#5.#6#7.#8.#9.%
2313 {%
2314 \csname XINT_infloat_Q\if-#1\else\if-#6\else q\fi\fi\expandafter\endcsname
2315 \romannumeral0\xintiiquo
2316 {\romannumeral0\XINT_dsx_addzerosnofuss

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

2317     {#6#7-#1#2+#9+\xint_c_iii\if-#1\else\if-#6\else-\xint_c_ii\fi\fi}#8;}%
2318     {#3}.#9.#5.%
2319 }%
    «quick» branch.
2320 \def\xINT_infloat_Qq #1.#2.%
2321 {%
2322     \expandafter\xINT_infloat_Rq
2323     \romannumeral0\xINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..#2.%
2324 }%
2325 \def\xINT_infloat_Rq #1.#2#3.%
2326 {%
2327     \ifnum#2<\xint_c_v
2328         \expandafter\xINT_infloat_SEq
2329     \else\expandafter\xINT_infloat_SUP
2330     \fi
2331     {\if.#3.\xint_c_\else\xint_c_i\fi}#1.%
2332 }%
    standard branch which will have to handle undecided rounding, if too close to a mid-value.
2333 \def\xINT_infloat_Q #1.#2.%
2334 {%
2335     \expandafter\xINT_infloat_R
2336     \romannumeral0\xINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..#2.%
2337 }%
2338 \def\xINT_infloat_R #1.#2#3#4#5.%
2339 {%
2340     \if.#5.\expandafter\xINT_infloat_Sa\else\expandafter\xINT_infloat_Sb\fi
2341     #2#3#4#5.#1.%
2342 }%
    trailing digits.Q.P.|B|.|A|.n.{A}{B}
    #1=trailing digits (they may have leading zeros.)
2343 \def\xINT_infloat_Sa #1.%
2344 {%
2345     \ifnum#1>500 \xint_dothis\xINT_infloat_SUP\fi
2346     \ifnum#1<499 \xint_dothis\xINT_infloat_SEq\fi
2347     \xint_orthat\xINT_infloat_X\xint_c_
2348 }%
2349 \def\xINT_infloat_Sb #1.%
2350 {%
2351     \ifnum#1>5009 \xint_dothis\xINT_infloat_SUP\fi
2352     \ifnum#1<4990 \xint_dothis\xINT_infloat_SEq\fi
2353     \xint_orthat\xINT_infloat_X\xint_c_i
2354 }%
    epsilon #2=Q.#3=P.#4=|B|. #5=|A|. #6=n.{A}{B}
    exposant final est n+|A|-|B|-P+epsilon
2355 \def\xINT_infloat_SEq #1#2.#3.#4.#5.#6.#7#8%
2356 {%
2357     \expandafter\xINT_infloat_SY
2358     \the\numexpr #6+#5-#4-#3+#1.#2.%
2359 }%
2360 \def\xINT_infloat_SY #1.#2.{ #2[#1]}%
```

initial digit #2 put aside to check for case of rounding up to next power of ten, which will need adjustment of mantissa and exponent.

```
2361 \def\XINT_infloat_SUP #1#2#3.#4.#5.#6.#7.#8#9%
2362 {%
2363   \expandafter\XINT_infloat_Y
2364   \the\numexpr#7+#6-#5-#4+#1\expandafter.%
2365   \romannumeral0\xintinc{#2#3}.#2%
2366 }%
epsilon Q.P. |B|. |A|.n.{A}{B}
```

`\xintDSH{-x}{U}` multiplies U by  $10^x$ . When x is negative, this means it truncates (i.e. it drops the last -x digits).

We don't try to optimize too much macro calls here, the odds are 2 per 1000 for this branch to be taken. Perhaps in future I will use higher free parameter d, which currently is set at 4.

#1=epsilon, #2#3=Q, #4=P, #5=|B|, #6=|A|, #7=n, #8=A, #9=B

```
2367 \def\XINT_infloat_X #1#2#3.#4.#5.#6.#7.#8#9%
2368 {%
2369   \expandafter\XINT_infloat_Y
2370   \the\numexpr #7+#6-#5-#4+#1\expandafter.%
2371   \romannumeral`&&\romannumeral0\xintiiiflt
2372   {\xintDSH{#6-#5-#4+#1}{\xintDouble{#8}}}%
2373   {\xintiiMul{\xintInc{\xintDouble{#2#3}}}{#9}}%
2374   \xint_firstofone
2375   \xintinc{#2#3}.#2%
2376 }%
```

check for rounding up to next power of ten.

```
2377 \def\XINT_infloat_Y #1{%
2378 \def\XINT_infloat_Y ##1.##2##3.##4%
2379 {%
2380   \if##49\if##21\expandafter\expandafter\expandafter\XINT_infloat_Z\fi\fi
2381   #1##2##3[##1]%
2382 }}\XINT_infloat_Y{ }%
#1=1, #2=0.
2383 \def\XINT_infloat_Z #1#2#3[#4]%
2384 {%
2385   \expandafter\XINT_infloat_ZZ\the\numexpr#4+\xint_c_i.#3.%
2386 }%
2387 \def\XINT_infloat_ZZ #1.#2.{ 1#2[#1]}%
```

## 24.79. \XINTFloatiLogTen

**Added at 1.3e (2019/04/05).** Le comportement pour un input nul est non encore finalisé. Il changera lorsque NaN, +Inf, -Inf existeront.

The optional argument [#1] is in fact mandatory and #1 is not pre-expanded in a `\numexpr`.

The return value here  $2^{31}-2^{15}$  is highly undecided.

```
2388 \def\XINTFloatiLogTen {\the\numexpr\XINTfloatilogten}%
2389 \def\XINTfloatilogten [#1]#2%
2390   {\expandafter\XINT_floatilogten\romannumeral0\XINT_infloat[#1]{#2}#1.%}
2391 \def\XINTFloatiLogTendigits{\the\numexpr\XINTfloatilogten[\XINTdigits]}%
2392 \def\XINT_floatilogten #1{%
2393   \if #10\xint_dothis\XINT_floatilogten_z\fi
```

```

2394 \if #1!\xint_dothis\xINT_floatilogten_a\fi
2395 \xint_orthat\xINT_floatilogten_b #1%
2396 }%
2397 \def\xINT_floatilogten_z 0[0]#1.{-7FFF8000\relax}%
2398 \def\xINT_floatilogten_a !#1.#2[#3]#4.{#3-#1+#4-\xint_c_i\relax}%
2399 \def\xINT_floatilogten_b #1[#2]#3.{#2+#3-\xint_c_i\relax}%

```

## 24.80. \xintPFloat

Added at 1.1 (2014/10/28).

Modified at 1.4e (2021/05/05).

`xint` has not yet incorporated a general formatter as it was not a priority during development and external solutions exist (I did not check for a while but I think LaTeX3 has implemented a general formatter in the `printf` or Python `".format"` spirit)

But when one starts using really the package, especially in an interactive way (`xintsession 2021`), one needs the default output to be as nice as possible.

The `\xintPFloat` macro was added at 1.1 as a "prettifying printer" for floats, basically influenced by Maple.

The rules were:

0. The input is float-rounded to either `Digits` or the optional argument
1. zero is printed as "0."
2. `x.yz...eK` is printed "as is" if  $K > 5$  or  $K < -5$ .
3. if  $-5 \leq K \leq 5$ , fixed point decimal notation is used.
4. in cases 2. and 3., no trimming of trailing zeroes.

1.4b added `\xintPFloatE` to customize whether to use `e` or `E`.

1.4e, with some hesitation, decided to make a breaking change and to modify the behaviour.

The new rules:

0. The input is float-rounded to either `Digits` or the optional argument
1. zero is printed as `0.0`
2. `x.yz...eK` is printed in decimal fixed point if  $-4 \leq K \leq +5$  (notice the change, formerly  $K \leq -5$  used fixed point notation in output) else it is printed in scientific notation
3. trailing zeros of the mantissa are trimmed always
4. in case of decimal fixed point for an integer, there is a trailing `".0"`
5. in case of scientific notation with a one-digit trimmed mantissa there is an added `".0"` too

Further, `\xintPFloatE` can now also be redefined as a macro with a parameter delimited by a full stop, with the full stop also in its output as terminator. It would then grab the scientific exponent `K` as explicit digit possibly prefixed by a minus sign. The macro must be f-expandable.

The macro `\xintPFloat_wopt` is only there for a micro gain as the package does

```
\let\xintfloatexprPrintOne\xintPFloat_wopt
```

as it knows it will be used always with a `[P]` argument in the `xintexpr.sty` context.

Modified at 1.4k (2022/05/18). Addition of customization via `\xintPFloatZero`, `\xintPFloatLengt`, `hOneSuffix`, `\xintPFloatNoSciEmax`, `\xintPFloatNoSciEmin` which replace formerly hard-coded behaviour.

Breaking change to not add `".0"` suffix to integers (when scientific notation dropped) or to one-digit mantissas.

In my own practice I started being annoyed by the automatic trimming of zeros added at 1.4e.

This change had been influenced by using Python in interactive mode which since 3.1 prints floats (in decimal conversion) choosing the shortest string. In particular it trims trailing zeros, and it drops the scientific notation in favor of decimal notation for something like  $-4 \leq K \leq 15$ , with `K` the scientific exponent.

At 1.4e I was still influenced by my experience with Maple and did for  $-4 \leq K \leq 5$ . Not very well thought anyhow (one may wish to use decimal notation when sending things to PostScript, so perhaps I should have kept with -5).

But, the main problem is with trimming trailing zeros: although in interactive sessions, this has its logic, as soon as one does tables with numbers, dropping a trailing zero upsets alignments or creates visual holes compared to other lines and this is in the end very annoying.

After much hesitation, I decided to slightly modify only the former behaviour: trimming only if that removes at least 4 zeros. I had also experimented with another condition: trimmed mantissas should be at most 6 digits (for example) wide, else use no trimming.

Threshold customizable via `\xintPFloatMinTrimmed`.

**Modified at 1.4l (2022/05/29).** The 1.4k check for canceling the trimming of trailing zeros took over priority over the later check for being an integer when decimal fixed point notation was used (or being only with a one-digit trimmed mantissa). In particular if user set `\xintPFloatMinTrimmed` to the value of Digits (or P) to avoid trimming it also prevented recognition of some integers (but not all). Fixed at 1.4l

```

2400 \def\xintPFloatE{e}%
2401 \def\xintPFloatNoSciEmax{\xint_c_v}% 1e6 uses sci.not.
2402 \def\xintPFloatNoSciEmin{-\xint_c_iv}% 1e-5 uses sci.not.
2403 \def\xintPFloatIntSuffix{}%
2404 \def\xintPFloatLengthOneSuffix{}%
2405 \def\xintPFloatZero{0}%
2406 \def\xintPFloatMinTrimmed{\xint_c_iv}%
2407 \def\xintPFloat {\romannumeral0\xintpfloat }%
2408 \def\xintpfloat #1{\XINT_pfloat_chkopt #1\xint:}%
2409 \def\xintPFloat_wopt[#1]#2%
2410 {%
2411     \romannumeral0\expandafter\XINT_pfloat
2412     \romannumeral0\XINTinfloatS[#1]{#2}#1.%
2413 }%
2414 \def\XINT_pfloat_chkopt #1%
2415 {%
2416     \ifx [#1\expandafter\XINT_pfloat_opt
2417         \else\expandafter\XINT_pfloat_noopt
2418     \fi #1%
2419 }%
2420 \def\XINT_pfloat_noopt #1\xint:%
2421 {%
2422     \expandafter\XINT_pfloat\romannumeral0\XINTinfloatS[\XINTdigits]{#1}%
2423     \XINTdigits.%
2424 }%
2425 \def\XINT_pfloat_opt [\xint:{\expandafter\XINT_pfloat_opt_a\the\numexpr}%
2426 \def\XINT_pfloat_opt_a #1]#2%
2427 {%
2428     \expandafter\XINT_pfloat\romannumeral0\XINTinfloatS[#1]{#2}%
2429     #1.%
2430 }%
2431 \def\XINT_pfloat#1]%
2432 {%
2433     \expandafter\XINT_pfloat_fork\romannumeral0\xintrez{#1}%
2434 }%
2435 \def\XINT_pfloat_fork#1%
2436 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

2437 \xint_UDzerominusfork
2438 #1-\XINT_pfloat_zero
2439 0#1\XINT_pfloat_neg
2440 0-\XINT_pfloat_pos
2441 \krof #1%
2442 }%
2443 \def\XINT_pfloat_zero#1#2.{\expanded{ \xintPFloatZero}}%
2444 \def\XINT_pfloat_neg-{\expandafter-\romannumeral0\XINT_pfloat_pos}%
2445 \def\XINT_pfloat_pos#1/1[#2]#3.%
2446 {%
2447 \expandafter\XINT_pfloat_aa\the\numexpr\xintLength{#1}.%
2448 #3.#2.#1.%
2449 }%

#1 est la longueur de la mantisse trimmée
#2 est Digits ou P
Si #2-#1 < MinTrimmed, on se prépare à peut-être remettre les trailing zeros
On teste pour #2=#1, car c'est le cas le plus fréquent (mais est-ce une bonne idée) car on sait
qu'alors il n'y a pas de trailing zéros donc on va direct vers \XINT_pfloat_a.

2450 \def\XINT_pfloat_aa #1.#2.%
2451 {%
2452 \unless\ifnum\xintPFloatMinTrimmed>\numexpr#2-#1\relax
2453 \xint_dothis\XINT_pfloat_a\fi
2454 \ifnum#2>#1 \xint_dothis{\XINT_pfloat_i #2.}\fi
2455 \xint_orthat\XINT_pfloat_a #1.%
2456 }%

Needed for \xintFracToSci, which uses old pre 1.4k interface, where the P parameter was not stored
for counting how many zeros were trimmed. \xintFracToSci trims always.

2457 \def\XINT_pfloat_a_fork#1%
2458 {%
2459 \xint_UDzerominusfork
2460 #1-\XINT_pfloat_a_zero
2461 0#1\XINT_pfloat_a_neg
2462 0-\XINT_pfloat_a_pos
2463 \krof #1%
2464 }%
2465 \def\XINT_pfloat_a_zero#1#2.{\expanded{ \xintPFloatZero}}%
2466 \def\XINT_pfloat_a_neg-{\expandafter-\romannumeral0\XINT_pfloat_a_pos}%
2467 \def\XINT_pfloat_a_pos#1/1[#2]#3.%
2468 {%
2469 \expandafter\XINT_pfloat_a\the\numexpr\xintLength{#1}.#2.#1.%
2470 }%

#1 est P > #2 mais peut être encore sous la forme \XINTdigits
#2 est la longueur de la mantisse trimmée
#3 est l'exposant non normalisé
#4 est la mantisse

On reconstitue les trailing zéros à remettre éventuellement.

2471 \def\XINT_pfloat_i #1.#2.%#3.#4.%
2472 {%
2473 \expandafter\XINT_pfloat_j\romannumeral\xintreplicate{#1-#2}0.#2.%
2474 }%

#1 est les trailing zeros à remettre peut-être

```

#2 est la longueur de la mantisse trimmée  
 #3#4 est l'exposant N pour mantisse trimmée entière  
 #5 serait la mantisse trimmée

On calcule l'exposant scientifique.

La façon bizarre de mettre #3 est liée aux versions anciennes de la macro, héritage conservé pour minimiser effort d'adaptation.

```
2475 \def\XINT_pfloat_j #1.#2.#3#4.##5.
2476 {%
2477   \expandafter\XINT_pfloat_b\the\numexpr#2+#3#4-\xint_c_i.%
2478   #3#2.#1.%
2479 }%
```

#1 est la longueur de la mantisse trimmée  
 #2#3 est l'exposant N pour mantisse trimmée  
 #4 serait la mantisse

On calcule l'exposant scientifique. On est arrivé ici dans une branche où on n'a pas besoin de remettre les zéros trimmés donc on positionne un dernier argument vide pour `\XINT_pfloat_b`

```
2480 \def\XINT_pfloat_a #1.#2#3.##4.
2481 {%
2482   \expandafter\XINT_pfloat_b\the\numexpr#1+#2#3-\xint_c_i.%
2483   #2#1..%
2484 }%
```

#1 est l'exposant scientifique K  
 #2 est le signe ou premier chiffre de l'exposant N pour mantisse trimmée  
 #3 serait la longueur de la mantisse trimmée  
 #4 serait les trailing zéros  
 #5 serait la mantisse trimmée

On va vers `\XINT_float_P` lorsque l'on n'utilise pas la notation scientifique, mais qu'on a besoin de chiffres non nuls fractionnaires, et vers `\XINT_float_Ps` si on n'en a pas besoin.

On va vers `\XINT_pfloat_N` lorsque l'on n'utilise pas la notation scientifique et que l'exposant scientifique était strictement négatif.

```
2485 \def\XINT_pfloat_b #1.#2%#3.#4.#5.
2486 {%
2487   \ifnum \xintPFloatNoSciEmax<#1 \xint_dothis\XINT_pfloat_sci\fi
2488   \ifnum \xintPFloatNoSciEmin>#1 \xint_dothis\XINT_pfloat_sci\fi
2489   \ifnum #1<\xint_c_ \xint_dothis\XINT_pfloat_N\fi
2490   \if-#2\xint_dothis\XINT_pfloat_P\fi
2491   \xint_orthat\XINT_pfloat_Ps
2492   #1.%
2493 }%
```

#1 is the scientific exponent, #2 is the length of the trimmed mantissa, #3 are the trailing zeros, #4 is the trimmed integer mantissa

`\xintPFloatE` can be replaced by any f-expandable macro with a dot-delimited argument which produces a dot-delimited output.

```
2494 \def\XINT_pfloat_sci #1.#2.%
2495 {%
2496   \ifnum#2=\xint_c_i\expandafter\XINT_pfloat_sci_i\expandafter\fi
2497   \expandafter\XINT_pfloat_sci_a\romannumeral`&&@\xintPFloatE #1.%
2498 }%
2499 \def\XINT_pfloat_sci_a #1.#2.#3#4.{ #3.#4#2#1}%
#1#2=\fi\XINT_pfloat_sci_a
```



1-digit mantissa, hesitation between d.0eK or deK Finally at 1.4k, `\xintPFloatLengthOneSuffix` for customization.

```

2500 \def\xINT_pfloat_sci_i #1#2#3.#4.#5.{\expanded{#1 #5\xintPFloatLengthOneSuffix}#3}%
    #1=sci.exp. K, #2=mant. wd L, #3=trailing zéros, #4=trimmed mantissa
    For _N, #1 is at most -1, for _P, #1 is at least 0. For _P there will be fractional digits, and
    #1+1 digits before the mark.
2501 \def\xINT_pfloat_N#1.#2.#3.#4.%
2502 {%
2503     \expandafter\xINT_pfloat_N_e\romannumeral\xintreplicate{-#1}{0}#4#3%
2504 }%
2505 \def\xINT_pfloat_N_e 0{ 0.}%
    #1=sci.exp. K, #2=mant. wd L, #3=trailing zéros, #4=trimmed mantissa
    Abusive usage of internal \XINT_split_fromleft_a. It means using x = -1 - #1 in \xintDecSplit
    from xint.sty. We benefit also with the way \xintDecSplit is built upon \XINT_split_fromleft with
    a final clean-up which here we can shortcut via using terminator "\xint_bye." not "\xint_bye.."
2506 \def\xINT_pfloat_P #1.#2.#3.#4.%
2507 {%
2508     \expandafter\xINT_split_fromleft_a
2509     \the\numexpr\xint_c_vii-#1.#4\xint_bye2345678\xint_bye.#3%
2510 }%
    Here we have an integer so we only need to postfix the trimmed mantissa #4 with #1+1-#2 zeros
    (#1=sci exp., #2=trimmed mantissa width). Less cumbersome to do that with \expanded. And the
    trailing zeros #3 ignored here.
2511 \def\xINT_pfloat_Ps #1.#2.#3.#4.%
2512 {%
2513     \expanded{ #4%
2514     \romannumeral\xintreplicate{#1+\xint_c_i-#2}{0}\xintPFloatIntSuffix}%
2515 }%
```

## 24.81. \xintFloatToDecimal

Added at 1.4k (2022/05/18).

```

2516 \def\xintFloatToDecimal {\romannumeral0\xintfloattodecimal }%
2517 \def\xintfloattodecimal #1{\XINT_floattodec_chkopt #1\xint:}%
2518 \def\xINT_floattodec_chkopt #1%
2519 {%
2520     \ifx [#1\expandafter\xINT_floattodec_opt
2521         \else\expandafter\xINT_floattodec_noopt
2522     \fi #1%
2523 }%
2524 \def\xINT_floattodec_noopt #1\xint:%
2525 {%
2526     \expandafter\xINT_floattodec\romannumeral0\xINTinfloatS[\XINTdigits]{#1}%
2527 }%
2528 \def\xINT_floattodec_opt [\xint:#1]%
2529 {%
2530     \expandafter\xINT_floattodec\romannumeral0\xINTinfloatS[#1]%
2531 }%
```

Temptation to try to use direct access to lower entry points from `\xintREZ`, but it dates back from very early days and uses old `\Z` delimiters (same remarks for the code jumping from `\xintFracToSci` to `\xintrez`)

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
2532 \def\XINT_floattodec#1]%
2533 {%
2534     \expandafter\XINT_dectostr\romannumeral0\xintrez{#1}}%
2535 }%
```

### 24.82. \XINTinFloatFrac

Added at 1.09i (2013/12/18).

For `frac` function in `\xintfloatexpr`. This version computes exactly from the input the fractional part and then only converts it into a float with the asked-for number of digits. I will have to think it again some day, certainly.

**Modified at 1.1 (2014/10/28).** 1.1 removes optional argument for which there was anyhow no interface, for technical reasons having to do with `\xintNewExpr`.

**Modified at 1.1a (2014/11/07).** 1.1a renames the macro as `\XINTinFloatFracdigits` (from `\XINTinFloatFrac`) to be synchronous with the `\XINTinFloatSqrt` and `\XINTinFloat` habits related to `\xintNewExpr` context and issues with macro names.

**Modified at 1.4e (2021/05/05).** 1.4e renames it back to `\XINTinFloatFrac` because of all such similarly named macros also using `\XINTdigits` forcedly.

```
2536 \def\XINTinFloatFrac {\romannumeral0\XINTinfloatfrac}%
2537 \def\XINTinfloatfrac #1%
2538 {%
2539     \expandafter\XINT_infloatfrac_a\expandafter {\romannumeral0\xinttfraction{#1}}%
2540 }%
2541 \def\XINT_infloatfrac_a {\XINTinfloat[\XINTdigits]}%
```

### 24.83. \xintFloatAdd, \XINTinFloatAdd

First included in release 1.07.

1.09ka improved a bit the efficiency. However the add, sub, mul, div routines were provisory and supposed to be revised soon.

Which didn't happen until 1.2f. Now, the inputs are first rounded to P digits, not P+2 as earlier.

See general introduction for important changes at 1.4e relative to the `\XINTinFloat<name>` macros.

```
2542 \def\xintFloatAdd {\romannumeral0\xintfloatadd}%
2543 \def\xintfloatadd #1{\XINT_fladd_chkopt \xintfloat #1\xint:}%
2544 \def\XINTinFloatAdd{\romannumeral0\XINTinfloatadd}%
2545 \def\XINTinfloatadd{\XINT_fladd_opt_a\XINTdigits.\XINTinfloatS}%
2546 \def\XINTinFloatAdd_wopt{\romannumeral0\XINTinfloatadd_wopt}%
2547 \def\XINTinfloatadd_wopt[#1]{\expandafter\XINT_fladd_opt_a\the\numexpr#1.\XINTinfloatS}%
2548 \def\XINT_fladd_chkopt #1#2%
2549 {%
2550     \ifx [#2\expandafter\XINT_fladd_opt
2551         \else\expandafter\XINT_fladd_noopt
2552     \fi #1#2%
2553 }%
2554 \def\XINT_fladd_noopt #1#2\xint:#3%
2555 {%
2556     #1[\XINTdigits]%
2557     {\expandafter\XINT_FL_add_a
2558         \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.{#3}}%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
2559 }%
2560 \def\XINT_fladd_opt #1[\xint:#2]%#3#4%
2561 {%
2562   \expandafter\XINT_fladd_opt_a\the\numexpr #2.#1%
2563 }%
2564 \def\XINT_fladd_opt_a #1.#2#3#4%
2565 {%
2566   #2[#1]{\expandafter\XINT_FL_add_a\romannumeral0\XINTinfloat[#1]{#3}#1.{#4}}%
2567 }%
2568 \def\XINT_FL_add_a #1%
2569 {%
2570   \xint_gob_til_zero #1\XINT_FL_add_zero 0\XINT_FL_add_b #1%
2571 }%
2572 \def\XINT_FL_add_zero #1.#2{#2}%[[
2573 \def\XINT_FL_add_b #1]#2.#3%
2574 {%
2575   \expandafter\XINT_FL_add_c\romannumeral0\XINTinfloat[#2]{#3}#2.#1%
2576 }%
2577 \def\XINT_FL_add_c #1%
2578 {%
2579   \xint_gob_til_zero #1\XINT_FL_add_zero 0\XINT_FL_add_d #1%
2580 }%
2581 \def\XINT_FL_add_d #1[#2]#3.#4[#5]%
2582 {%
2583   \ifnum\numexpr #2-#3-#5>\xint_c_\xint_dothis\xint_firstoftwo\fi
2584   \ifnum\numexpr #5-#3-#2>\xint_c_\xint_dothis\xint_secondoftwo\fi
2585   \xint_orthat\xintAdd {#1[#2]}{#4[#5]}%
2586 }%
```

### 24.84. \xintFloatSub, \XINTinFloatSub

Added at 1.07 (2013/05/25).

Modified at 1.2f (2016/03/12). Starting with 1.2f the arguments undergo an initial rounding to the target precision P not P+2.

```
2587 \def\xintFloatSub {\romannumeral0\xintfloatsub}%
2588 \def\xintfloatsub #1{\XINT_flsb_chkopt \xintfloat #1\xint:}%
2589 \def\XINTinFloatSub{\romannumeral0\XINTinfloatsub}%
2590 \def\XINTinfloatsub{\XINT_flsb_opt_a\XINTdigits.\XINTinfloatS}%
2591 \def\XINTinFloatSub_wopt{\romannumeral0\XINTinfloatsub_wopt}%
2592 \def\XINTinfloatsub_wopt[#1]{\expandafter\XINT_flsb_opt_a\the\numexpr#1.\XINTinfloatS}%
2593 \def\XINT_flsb_chkopt #1#2%
2594 {%
2595   \ifx [#2\expandafter\XINT_flsb_opt
2596     \else\expandafter\XINT_flsb_noopt
2597   \fi #1#2%
2598 }%
2599 \def\XINT_flsb_noopt #1#2\xint:#3%
2600 {%
2601   #1[\XINTdigits]%
2602   {\expandafter\XINT_FL_add_a
2603     \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.{\xintOpp{#3}}}%
2604 }%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
2605 \def\XINT_flsb_opt #1[\xint:#2]%#3#4%
2606 {%
2607   \expandafter\XINT_flsb_opt_a\the\numexpr #2.#1%
2608 }%
2609 \def\XINT_flsb_opt_a #1.#2#3#4%
2610 {%
2611   #2[#1]{\expandafter\XINT_FL_add_a\romannumeral0\XINTinfloat[#1]{#3}#1.{\xintOpp{#4}}}%
2612 }%
```

### 24.85. \xintFloatMul, \XINTinFloatMul

Added at 1.07 (2013/05/25).

Modified at 1.2d (2015/11/18). Starting with 1.2f the arguments are rounded to the target precision P not P+2.

Modified at 1.2g (2016/03/19). 1.2g handles the inputs via \XINTinFloatS which will be more efficient when the precision is large and the input is for example a small constant like 2.

```
2613 \def\xintFloatMul {\romannumeral0\xintfloatmul}%
2614 \def\xintfloatmul #1{\XINT_flmul_chkopt \xintfloat #1\xint:}%
2615 \def\XINTinFloatMul{\romannumeral0\XINTinfloatmul}%
2616 \def\XINTinfloatmul{\XINT_flmul_opt_a\XINTdigits.\XINTinfloatS}%
2617 \def\XINTinFloatMul_wopt{\romannumeral0\XINTinfloatmul_wopt}%
2618 \def\XINTinfloatmul_wopt[#1]{\expandafter\XINT_flmul_opt_a\the\numexpr#1.\XINTinfloatS}%
2619 \def\XINT_flmul_chkopt #1#2%
2620 {%
2621   \ifx [#2\expandafter\XINT_flmul_opt
2622     \else\expandafter\XINT_flmul_noopt
2623   \fi #1#2%
2624 }%
2625 \def\XINT_flmul_noopt #1#2\xint:#3%
2626 {%
2627   #1[\XINTdigits]%
2628   {\expandafter\XINT_FL_mul_a
2629     \romannumeral0\XINTinfloatS[\XINTdigits]{#2}\XINTdigits.{#3}}%
2630 }%
2631 \def\XINT_flmul_opt #1[\xint:#2]%#3#4%
2632 {%
2633   \expandafter\XINT_flmul_opt_a\the\numexpr #2.#1%
2634 }%
2635 \def\XINT_flmul_opt_a #1.#2#3#4%
2636 {%
2637   #2[#1]{\expandafter\XINT_FL_mul_a\romannumeral0\XINTinfloatS[#1]{#3}#1.{#4}}%
2638 }%
2639 \def\XINT_FL_mul_a #1[#2]#3.#4%
2640 {%
2641   \expandafter\XINT_FL_mul_b\romannumeral0\XINTinfloatS[#3]{#4}#1[#2]%
2642 }%
2643 \def\XINT_FL_mul_b #1[#2]#3[#4]{\xintiiMul{#3}{#1}/1[#4+#2]}%
```

### 24.86. \xintFloatSqr, \XINTinFloatSqr

Added at 1.4e (2021/05/05). Strangely \xintFloatSqr had never been defined so far.

An `\XINTinFloatSqr{#1}` was defined in `xintexpr.sty` directly as `\XINTinFloatMul[\XINTdigit_s]{#1}{#1}`, to support the `sqr()` function. The `{#1}{#1}` causes no problem as `#1` in this context is always pre-expanded so we don't need to worry about this, and the `\xintdeffloatfunc` mechanism should hopefully take care to add the needed argument pre-expansion if need be.

Anyway let's do this finally properly here.

```

2644 \def\xintFloatSqr  {\romannumeral0\xintfloatsqr}%
2645 \def\xintfloatsqr  #1{\XINT_flqr_chkopt \xintfloat #1\xint:}%
2646 \def\XINTinFloatSqr{\romannumeral0\XINTinfloatsqr}%
2647 \def\XINTinfloatsqr{\XINT_flqr_opt_a\XINTdigits.\XINTinfloatS}%
2648 \def\XINT_flqr_chkopt #1#2%
2649 {%
2650   \ifx [#2\expandafter\XINT_flqr_opt
2651     \else\expandafter\XINT_flqr_noopt
2652   \fi  #1#2%
2653 }%
2654 \def\XINT_flqr_noopt #1#2\xint:
2655 {%
2656   #1[\XINTdigits]%
2657   {\expandafter\XINT_FL_sqr_a\romannumeral0\XINTinfloatS[\XINTdigits]{#2}}%
2658 }%
2659 \def\XINT_flqr_opt #1[\xint:#2]%
2660 {%
2661   \expandafter\XINT_flqr_opt_a\the\numexpr #2.#1%
2662 }%
2663 \def\XINT_flqr_opt_a #1.#2#3%
2664 {%
2665   #2[#1]{\expandafter\XINT_FL_sqr_a\romannumeral0\XINTinfloatS[#1]{#3}}%
2666 }%
2667 \def\XINT_FL_sqr_a #1[#2]{\xintiisqr{#1}/1[#2+#2]}%
2668 \def\XINTinFloatSqr_wopt[#1]#2{\XINTinFloatS[#1]{\expandafter\XINT_FL_sqr_a\romannumeral0\XINTinfloatS[#1]

```

## 24.87. `\XINTinFloatInv`

**Added at 1.3e (2019/04/05).** Added belatedly at 1.3e, to support `inv()` function. We use Short output, for rare `inv(\xintexpr 1/3\relax)` case. I need to think the whole thing out at some later date.

```

2669 \def\XINTinFloatInv#1{\XINTinFloatS[\XINTdigits]{\xintInv{#1}}}%
2670 \def\XINTinFloatInv_wopt[#1]#2{\XINTinFloatS[#1]{\xintInv{#2}}}%

```

## 24.88. `\xintFloatDiv`, `\XINTinFloatDiv`

**Added at 1.07 (2013/05/25).**

**Modified at 1.2f (2016/03/12).** Starting with 1.2f the arguments are rounded to the target precision  $P$  not  $P+2$ .

**Modified at 1.2g (2016/03/19).** 1.2g handles the inputs via `\XINTinFloatS` which will be more efficient when the precision is large and the input is for example a small constant like 2.

The actual rounding of the quotient is handled via `\xintfloat` (or `\XINTinfloatS`).

**Modified at 1.2k (2017/01/06).** 1.2k does the same kind of improvement in `\XINT_FL_div_b` as for multiplication: earlier code was unnecessarily high level.

```

2671 \def\xintFloatDiv  {\romannumeral0\xintfloatdiv}%
2672 \def\xintfloatdiv  #1{\XINT_fldiv_chkopt \xintfloat #1\xint:}%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfac*, *xintexpr*, *xinttrig*, *xintlog*

```
2673 \def\XINTinFloatDiv{\romannumeral0\XINTinfloatdiv}%
2674 \def\XINTinfloatdiv{\XINT_fldiv_opt_a\XINTdigits.\XINTinfloatS}%
2675 \def\XINTinFloatDiv_wopt[#1]{\romannumeral0\XINT_fldiv_opt_a#1.\XINTinfloatS}%
2676 \def\XINT_fldiv_chkopt #1#2%
2677 {%
2678     \ifx [#2\expandafter\XINT_fldiv_opt
2679     \else\expandafter\XINT_fldiv_noopt
2680     \fi #1#2%
2681 }%
    1.4g adds here intercept of second argument being zero, else a low level error will arise at later
    stage from the the fall-back value returned by core iidivision being 0 and not having expected
    number of digits at \XINT_infloat_Qq and split from left returning some empty value breaking the
    \ifnum test in \XINT_infloat_Rq.
2682 \def\XINT_fldiv_noopt #1#2\xint:#3%
2683 {%
2684     #1[\XINTdigits]%
2685     {\expandafter\XINT_FL_div_aa
2686     \romannumeral0\XINTinfloatS[\XINTdigits]{#3}\XINTdigits.{#2}}%
2687 }%
2688 \def\XINT_FL_div_aa #1%
2689 {%
2690     \xint_gob_til_zero#1\XINT_FL_div_Bzero0\XINT_FL_div_a #1%
2691 }%
2692 \def\XINT_FL_div_Bzero0\XINT_FL_div_a#1[#2]#3.#4%
2693 {%
2694     \XINT_signalcondition{DivisionByZero}{Division by zero (#1[#2]) of #4}{0[0]}%
2695 }%
2696 \def\XINT_fldiv_opt #1[\xint:#2]#3#4%
2697 {%
2698     \expandafter\XINT_fldiv_opt_a\the\numexpr #2.#1%
2699 }%
    Also here added early check at 1.4g if divisor is zero.
2700 \def\XINT_fldiv_opt_a #1.#2#3#4%
2701 {%
2702     #2[#1]{\expandafter\XINT_FL_div_aa\romannumeral0\XINTinfloatS[#1]{#4}#1.{#3}}%
2703 }%
2704 \def\XINT_FL_div_a #1[#2]#3.#4%
2705 {%
2706     \expandafter\XINT_FL_div_b\romannumeral0\XINTinfloatS[#3]{#4}/#1e#2%
2707 }%
2708 \def\XINT_FL_div_b #1[#2]{#1e#2}%
```

### 24.89. \xintFloatPow, \XINTinFloatPow

Added at 1.07 (2013/05/25).

1.09j has re-organized the core loop.

2015/12/07. I have hesitated to map ^ in expressions to `\xintFloatPow` rather than `\xintFloatPower`. But for 1.234567890123456 to the power 2145678912 with P=16, using Pow rather than Power seems to bring only about 5% gain.

This routine requires the exponent x to be compatible with `\numexpr` parsing.

**Modified at 1.2f (2016/03/12).** 1.2f has rewritten the code for better efficiency. Also, now the argument A for  $A^x$  is first rounded to P digits before switching to the increased working precision (which depends upon x).

```

2709 \def\xintFloatPow {\romannumeral0\xintfloatpow}%
2710 \def\xintfloatpow #1{\XINT_flpow_chkopt \xintfloat #1\xint:}%
2711 \def\XINTinFloatPow{\romannumeral0\XINTinfloatpow }%
2712 \def\XINTinfloatpow{\XINT_flpow_opt_a\XINTdigits.\XINTinfloatS}%
2713 \def\XINTinfloatpow_wopt[#1]{\expandafter\XINT_flpow_opt_a\the\numexpr#1.\XINTinfloatS}%
2714 \def\XINT_flpow_chkopt #1#2%
2715 {%
2716   \ifx [#2\expandafter\XINT_flpow_opt
2717     \else\expandafter\XINT_flpow_noopt
2718   \fi
2719   #1#2%
2720 }%
2721 \def\XINT_flpow_noopt #1#2\xint:#3%
2722 {%
2723   \expandafter\XINT_flpow_checkB_a
2724   \the\numexpr #3.\XINTdigits.{#2}{#1[\XINTdigits]]}%
2725 }%
2726 \def\XINT_flpow_opt #1[\xint:#2]%
2727 {%
2728   \expandafter\XINT_flpow_opt_a\the\numexpr #2.#1%
2729 }%
2730 \def\XINT_flpow_opt_a #1.#2#3#4%
2731 {%
2732   \expandafter\XINT_flpow_checkB_a\the\numexpr #4.#1.{#3}{#2[#1]]}%
2733 }%
2734 \def\XINT_flpow_checkB_a #1%
2735 {%
2736   \xint_UDzerominusfork
2737   #1-\XINT_flpow_BisZero
2738   0#1{\XINT_flpow_checkB_b -}%
2739   0-{\XINT_flpow_checkB_b }{#1}%
2740   \kroft
2741 }%
2742 \def\XINT_flpow_BisZero .#1.#2#3{#3{1[0]}}%
2743 \def\XINT_flpow_checkB_b #1#2.#3.%
2744 {%
2745   \expandafter\XINT_flpow_checkB_c
2746   \the\numexpr\xintLength{#2}+\xint_c_iii.#3.#2.{#1}%
2747 }%
2748 \def\XINT_flpow_checkB_c #1.#2.%
2749 {%
2750   \expandafter\XINT_flpow_checkB_d\the\numexpr#1+#2.#1.#2.%
2751 }%

```

1.2f rounds input to P digits, first.

```

2752 \def\XINT_flpow_checkB_d #1.#2.#3.#4.#5#6%
2753 {%
2754   \expandafter \XINT_flpow_aa
2755   \romannumeral0\XINTinfloat [#3]{#6}{#2}{#1}{#4}{#5}%
2756 }%

```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

2757 \def\XINT_flpow_aa #1[#2]#3%
2758 {%
2759     \expandafter\XINT_flpow_ab\the\numexpr #2-#3\expandafter.%
2760     \romannumeral\XINT_rep #3\endcsname0.#1.%
2761 }%
2762 \def\XINT_flpow_ab #1.#2.#3.{\XINT_flpow_a #3#2[#1]}%
2763 \def\XINT_flpow_a #1%
2764 {%
2765     \xint_UDzerominusfork
2766     #1-\XINT_flpow_zero
2767     0#1{\XINT_flpow_b \iftrue}%
2768     0-{\XINT_flpow_b \iffalse#1}%
2769     \krof
2770 }%
2771 \def\XINT_flpow_zero #1[#2]#3#4#5#6%
2772 {%
2773     #6{\if 1#51\xint_dothis {0[0]}\fi
2774         \xint_orthat
2775         {\XINT_signalcondition{DivisionByZero}{0 raised to power -#4.}}{0[0]}}%
2776     }%
2777 }%
2778 \def\XINT_flpow_b #1#2[#3]#4#5%
2779 {%
2780     \XINT_flpow_loopI #5.#3.#2.#4.{#1\ifodd #5 \xint_c_i\fi\fi}%
2781 }%
2782 \def\XINT_flpow_truncate #1.#2.#3.%
2783 {%
2784     \expandafter\XINT_flpow_truncate_a
2785     \romannumeral0\XINT_split_fromleft
2786     #3.#2\xint_bye2345678\xint_bye..#1.#3.%
2787 }%
2788 \def\XINT_flpow_truncate_a #1.#2.#3.{#3+\xintLength{#2}.#1.}%
2789 \def\XINT_flpow_loopI #1.%
2790 {%
2791     \ifnum #1=\xint_c_i\expandafter\XINT_flpow_ItoIII\fi
2792     \ifodd #1
2793         \expandafter\XINT_flpow_loopI_odd
2794     \else
2795         \expandafter\XINT_flpow_loopI_even
2796     \fi
2797     #1.%
2798 }%
2799 \def\XINT_flpow_ItoIII\ifodd #1\fi #2.#3.#4.#5.#6%
2800 {%
2801     \expandafter\XINT_flpow_III\the\numexpr #6+\xint_c_.#3.#4.#5.%
2802 }%
2803 \def\XINT_flpow_loopI_even #1.#2.#3.##4.%
2804 {%
2805     \expandafter\XINT_flpow_loopI
2806     \the\numexpr #1/\xint_c_ii\expandafter.%
2807     \the\numexpr\expandafter\XINT_flpow_truncate
2808     \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.%

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

2809 }%
2810 \def\XINT_flpow_loopI_odd #1.#2.#3.#4.%
2811 {%
2812   \expandafter\XINT_flpow_loopII
2813   \the\numexpr #1/\xint_c_ii-\xint_c_i\expandafter.%
2814   \the\numexpr\expandafter\XINT_flpow_truncate
2815   \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#2.#3.%
2816 }%
2817 \def\XINT_flpow_loopII #1.%
2818 {%
2819   \ifnum #1 = \xint_c_i\expandafter\XINT_flpow_IItoIII\fi
2820   \ifodd #1
2821     \expandafter\XINT_flpow_loopII_odd
2822   \else
2823     \expandafter\XINT_flpow_loopII_even
2824   \fi
2825   #1.%
2826 }%
2827 \def\XINT_flpow_loopII_even #1.#2.#3.##4.%
2828 {%
2829   \expandafter\XINT_flpow_loopII
2830   \the\numexpr #1/\xint_c_ii\expandafter.%
2831   \the\numexpr\expandafter\XINT_flpow_truncate
2832   \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.%
2833 }%
2834 \def\XINT_flpow_loopII_odd #1.#2.#3.#4.#5.#6.%
2835 {%
2836   \expandafter\XINT_flpow_loopII_odda
2837   \the\numexpr\expandafter\XINT_flpow_truncate
2838   \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
2839   #1.#2.#3.%
2840 }%
2841 \def\XINT_flpow_loopII_odda #1.#2.#3.#4.#5.#6.%
2842 {%
2843   \expandafter\XINT_flpow_loopII
2844   \the\numexpr #4/\xint_c_ii-\xint_c_i\expandafter.%
2845   \the\numexpr\expandafter\XINT_flpow_truncate
2846   \the\numexpr\xint_c_ii*#5\expandafter.\romannumeral0\xintiisqr{#6}.#3.%
2847   #1.#2.%
2848 }%
2849 \def\XINT_flpow_IItoIII\ifodd #1\fi #2.#3.#4.#5.#6.#7.#8%
2850 {%
2851   \expandafter\XINT_flpow_III\the\numexpr #8+\xint_c_\expandafter.%
2852   \the\numexpr\expandafter\XINT_flpow_truncate
2853   \the\numexpr#3+#6\expandafter.\romannumeral0\xintiimul{#4}{#7}.#5.%
2854 }%

```

This ending is common with `\xintFloatPower`.

In the case of negative exponent we need to inverse the Q-digits mantissa. This requires no special attention now as 1.2k's `\xintFloat` does correct rounding of fractions hence it is easy to bound the total error. It can be checked that the algorithm after final rounding to the target precision computes a value Z whose distance to the exact theoretical will be less than 0.52 ulp(Z) (and worst cases can only be slightly worse than 0.51 ulp(Z)).

In the case of the half-integer exponent (only via the expression interface,) the computation (which proceeds via `\XINTinFloatPowerH`) ends with a square root. This square root extraction is done with 3 guard digits (the power operations were done with more.) Then the value is rounded to the target precision. There is thus this rounding to 3 guard digits (in the case of negative exponent the reciprocal is computed before the square-root), then the square root is (computed with exact rounding for these 3 guard digits), and then there is the final rounding of this to the target precision. The total error (for positive as well as negative exponent) has been estimated to at worst possibly exceed slightly  $0.5125 \text{ ulp}(Z)$ , and at any rate it is less than  $0.52 \text{ ulp}(Z)$ .

```

2855 \def\XINT_flpow_III #1.#2.#3.#4.#5%
2856 {%
2857   \expandafter\XINT_flpow_IIIend
2858   \xint_UDsignfork
2859     #5{{1/#3[-#2]}}}%
2860     -{{#3[#2]}}}%
2861   \krof #1%
2862 }%
2863 \def\XINT_flpow_IIIend #1#2#3%
2864   {#3{\if#21\xint_afterfi{\expandafter-\romannumeral`&&@\fi#1}}}%

```

## 24.90. `\xintFloatPower`, `\XINTinFloatPower`

**Added at 1.07 (2013/05/25).** The core loop has been re-organized in 1.09j for some slight efficiency gain. The exponent B is given to `\xintNum`. The `^` in expressions is mapped to this routine.

**Modified at 1.2f (2016/03/12).** Same modifications as in `\xintFloatPow` for 1.2f.

1.2f `\XINTinFloatPowerH` (now moved to `xintlog`, and renamed). It truncated the exponent to an integer of half-integer, and in the latter case use Square-root extraction. At 1.2k this was improved as 1.2f stupidly rounded to Digits before, not after the square root extraction, 1.2k kept 3 guard digits for this last step. And the initial step was changed to a rounding rather than truncating.

**Modified at 1.4e (2021/05/05).** Until 1.4e this `\XINTinFloatPowerH` was the macro for  $a^b$  in expressions, but of course it behaved strangely for  $b$  not an integer or an half-integer! At 1.4e, the non-integer, non-half-integer exponents will be handled via `log10()` and `pow10()` support macros, see `xintlog`. The code has now been relocated there.

```

2865 \def\xintFloatPower {\romannumeral0\xintfloatpower}%
2866 \def\xintfloatpower #1{\XINT_flpower_chkopt \xintfloat #1\xint:}%
2867 \def\XINTinFloatPower{\romannumeral0\XINTinfloatpower}%
2868 \def\XINTinfloatpower{\XINT_flpower_opt_a\XINTdigits.\XINTinfloatS}%

```

Start of macro. Check for optional argument.

```

2869 \def\XINT_flpower_chkopt #1#2%
2870 {%
2871   \ifx [#2\expandafter\XINT_flpower_opt
2872   \else\expandafter\XINT_flpower_noopt
2873   \fi
2874   #1#2%
2875 }%
2876 \def\XINT_flpower_noopt #1#2\xint:#3%
2877 {%
2878   \expandafter\XINT_flpower_checkB_a
2879   \romannumeral0\xintnum{#3}.\XINTdigits.{#2}{#1[\XINTdigits]}%
2880 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

2881 \def\XINT_flpower_opt #1[\xint:#2]%
2882 {%
2883   \expandafter\XINT_flpower_opt_a\the\numexpr #2.#1%
2884 }%
2885 \def\XINT_flpower_opt_a #1.#2#3#4%
2886 {%
2887   \expandafter\XINT_flpower_checkB_a
2888   \romannumeral0\xintnum{#4}.#1.{#3}{#2[#1]}%
2889 }%
2890 \def\XINT_flpower_checkB_a #1%
2891 {%
2892   \xint_UDzerominusfork
2893   #1-\XINT_flpower_BisZero 0}%
2894   0#1{\XINT_flpower_checkB_b -}%
2895   0-\XINT_flpower_checkB_b {}#1}%
2896   \krof
2897 }%
2898 \def\XINT_flpower_BisZero 0.#1.#2#3{#3{1[0]}}%
2899 \def\XINT_flpower_checkB_b #1#2.#3.%
2900 {%
2901   \expandafter\XINT_flpower_checkB_c
2902   \the\numexpr\xintLength{#2}+\xint_c_iii.#3.#2.{#1}%
2903 }%
2904 \def\XINT_flpower_checkB_c #1.#2.%
2905 {%
2906   \expandafter\XINT_flpower_checkB_d\the\numexpr#1+#2.#1.#2.%
2907 }%
2908 \def\XINT_flpower_checkB_d #1.#2.#3.#4.#5#6%
2909 {%
2910   \expandafter \XINT_flpower_aa
2911   \romannumeral0\XINTinfloat [#3]{#6}{#2}{#1}{#4}{#5}%
2912 }%
2913 \def\XINT_flpower_aa #1[#2]#3%
2914 {%
2915   \expandafter\XINT_flpower_ab\the\numexpr #2-#3\expandafter.%
2916   \romannumeral\XINT_rep #3\endcsname0.#1.%
2917 }%
2918 \def\XINT_flpower_ab #1.#2.#3.{\XINT_flpower_a #3#2[#1]}%
2919 \def\XINT_flpower_a #1%
2920 {%
2921   \xint_UDzerominusfork
2922   #1-\XINT_flpow_zero
2923   0#1{\XINT_flpower_b \iftrue}%
2924   0-\XINT_flpower_b \iffalse#1}%
2925   \krof
2926 }%
2927 \def\XINT_flpower_b #1#2[#3]#4#5%
2928 {%
2929   \XINT_flpower_loopI #5.#3.#2.#4.{#1\xintiiOdd{#5}\fi}%
2930 }%
2931 \def\XINT_flpower_loopI #1.%
2932 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

2933 \if1\XINT_isOne {#1}\xint_dothis\XINT_flpower_ItoIII\fi
2934 \ifodd\xintLDg{#1} %<- intentional space
2935 \xint_dothis{\expandafter\XINT_flpower_loopI_odd}\fi
2936 \xint_orthat{\expandafter\XINT_flpower_loopI_even}%
2937 \romannumeral0\XINT_half
2938 #1\xint_bye\xint_Bye345678\xint_bye
2939 *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax.%
2940 }%
2941 \def\XINT_flpower_ItoIII #1.#2.#3.#4.#5%
2942 {%
2943 \expandafter\XINT_flpow_III\the\numexpr #5+\xint_c_.#2.#3.#4.%
2944 }%
2945 \def\XINT_flpower_loopI_even #1.#2.#3.#4.%
2946 {%
2947 \expandafter\XINT_flpower_toloopI
2948 \the\numexpr\expandafter\XINT_flpow_truncate
2949 \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#1.%
2950 }%
2951 \def\XINT_flpower_toloopI #1.#2.#3.#4.{\XINT_flpower_loopI #4.#1.#2.#3}%
2952 \def\XINT_flpower_loopI_odd #1.#2.#3.#4.%
2953 {%
2954 \expandafter\XINT_flpower_toloopII
2955 \the\numexpr\expandafter\XINT_flpow_truncate
2956 \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.%
2957 #1.#2.#3.%
2958 }%
2959 \def\XINT_flpower_toloopII #1.#2.#3.#4.{\XINT_flpower_loopII #4.#1.#2.#3}%
2960 \def\XINT_flpower_loopII #1.%
2961 {%
2962 \if1\XINT_isOne{#1}\xint_dothis\XINT_flpower_ItoIII\fi
2963 \ifodd\xintLDg{#1} %<- intentional space
2964 \xint_dothis{\expandafter\XINT_flpower_loopII_odd}\fi
2965 \xint_orthat{\expandafter\XINT_flpower_loopII_even}%
2966 \romannumeral0\XINT_half#1\xint_bye\xint_Bye345678\xint_bye
2967 *\xint_c_v+\xint_c_v)/\xint_c_x-\xint_c_i\relax.%
2968 }%
2969 \def\XINT_flpower_loopII_even #1.#2.#3.#4.%
2970 {%
2971 \expandafter\XINT_flpower_toloopII
2972 \the\numexpr\expandafter\XINT_flpow_truncate
2973 \the\numexpr\xint_c_ii*#2\expandafter.\romannumeral0\xintiisqr{#3}.#4.#1.%
2974 }%
2975 \def\XINT_flpower_loopII_odd #1.#2.#3.#4.#5.#6.%
2976 {%
2977 \expandafter\XINT_flpower_loopII_odda
2978 \the\numexpr\expandafter\XINT_flpow_truncate
2979 \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
2980 #1.#2.#3.%
2981 }%
2982 \def\XINT_flpower_loopII_odda #1.#2.#3.#4.#5.#6.%
2983 {%
2984 \expandafter\XINT_flpower_toloopII

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
2985 \the\numexpr\expandafter\XINT_flpow_truncate
2986 \the\numexpr\xint_c_ii*#5\expandafter.\romannumeral0\xintiisqr{#6}.#3.%
2987 #4.#1.#2.%
2988 }%
2989 \def\XINT_flpower_IItoIII #1.#2.#3.#4.#5.#6.#7%
2990 {%
2991 \expandafter\XINT_flpow_III\the\numexpr #7+\xint_c_\expandafter.%
2992 \the\numexpr\expandafter\XINT_flpow_truncate
2993 \the\numexpr#2+#5\expandafter.\romannumeral0\xintiimul{#3}{#6}.#4.%
2994 }%
```

### 24.91. \xintFloatFac, \XINTFloatFac

Added at 1.2 (2015/10/10).

```
2995 \def\xintFloatFac {\romannumeral0\xintfloatfac}%
2996 \def\xintfloatfac #1{\XINT_flfac_chkopt \xintfloat #1\xint:}%
2997 \def\XINTinFloatFac{\romannumeral0\XINTinfloatfac}%
2998 \def\XINTinfloatfac[#1]{\expandafter\XINT_flfac_opt_a\the\numexpr#1.\XINTinfloatS}%
2999 \def\XINTinFloatFacdigits{\romannumeral0\XINT_flfac_opt_a\XINTdigits.\XINTinfloatS}%
3000 \def\XINT_flfac_chkopt #1#2%
3001 {%
3002 \ifx [#2\expandafter\XINT_flfac_opt
3003 \else\expandafter\XINT_flfac_noopt
3004 \fi
3005 #1#2%
3006 }%
3007 \def\XINT_flfac_noopt #1#2\xint:
3008 {%
3009 \expandafter\XINT_FL_fac_fork_a
3010 \the\numexpr \xintNum{#2}.\xint_c_i \XINTdigits\XINT_FL_fac_out{#1[\XINTdigits]]}%
3011 }%
3012 \def\XINT_flfac_opt #1[\xint:#2]%
3013 {%
3014 \expandafter\XINT_flfac_opt_a\the\numexpr #2.#1%
3015 }%
3016 \def\XINT_flfac_opt_a #1.#2#3%
3017 {%
3018 \expandafter\XINT_FL_fac_fork_a\the\numexpr \xintNum{#3}.\xint_c_i {#1}\XINT_FL_fac_out{#2[#1]]}%
3019 }%
3020 \def\XINT_FL_fac_fork_a #1%
3021 {%
3022 \xint_UDzerominusfork
3023 #1-\XINT_FL_fac_iszero
3024 0#1\XINT_FL_fac_isneg
3025 0-{\XINT_FL_fac_fork_b #1}%
3026 \krof
3027 }%
3028 \def\XINT_FL_fac_iszero #1.#2#3#4#5{#5{1[0]}}%
3029 1.2f XINT_FL_fac_isneg returns 0, earlier versions used 1 here.
3029 \def\XINT_FL_fac_isneg #1.#2#3#4#5%
3030 {%
3031 #5{\XINT_signalcondition{InvalidOperation}}
```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

3032             {Factorial argument is negative: -#1.}{ 0[0]}}%
3033 }%
3034 \def\XINT_FL_fac_fork_b #1.%
3035 {%
3036     \ifnum #1>\xint_c_x^viii_mone\xint_dothis\XINT_FL_fac_toobig\fi
3037     \ifnum #1>\xint_c_x^iv\xint_dothis\XINT_FL_fac_vbig \fi
3038     \ifnum #1>465 \xint_dothis\XINT_FL_fac_big\fi
3039     \ifnum #1>101 \xint_dothis\XINT_FL_fac_med\fi
3040     \xint_orthat\XINT_FL_fac_small
3041     #1.%
3042 }%
3043 \def\XINT_FL_fac_toobig #1.#2#3#4#5%
3044 {%
3045     #5{\XINT_signalcondition{InvalidOperation}}
3046     {Factorial argument is too large: #1>=10^8.}{ 0[0]}}%
3047 }%

```

Computations are done with  $Q$  blocks of eight digits. When a multiplication has a carry, hence creates  $Q+1$  blocks, the least significant one is dropped. The goal is to compute an approximate value  $X'$  to the exact value  $X$ , such that the final relative error  $(X-X')/X$  will be at most  $10^{-P-1}$  with  $P$  the desired precision. Then, when we round  $X'$  to  $X''$  with  $P$  significant digits, we can prove that the absolute error  $|X-X''|$  is bounded (strictly) by  $0.6 \text{ ulp}(X'')$ . (ulp= unit in the last (significant) place). Let  $N$  be the number of such operations, the formula for  $Q$  deduces from the previous explanations is that  $8Q$  should be at least  $P+9+k$ , with  $k$  the number of digits of  $N$  (in base 10). Note that 1.2 version used  $P+10+k$ , for 1.2f I reduced to  $P+9+k$ . Also,  $k$  should be the number of digits of the number  $N$  of multiplications done, hence for  $n \leq 10000$  we can take  $N=n/2$ , or  $N/3$ , or  $N/4$ . This is rounded above by `numexpr` and always an overestimate of the actual number of approximate multiplications done (the first ones are exact). (vérifier ce que je raconte, j'ai la flemme là).

We then want  $\text{ceil}((P+k+n)/8)$ . Using `\numexpr` rounding division (ARRRRRGGGHHHH), if  $m$  is a positive integer,  $\text{ceil}(m/8)$  can be computed as  $(m+3)/8$ . Thus with  $m=P+10+k$ , this gives  $Q \leftarrow (P+13+k)/8$ . The routine actually computes  $8(Q-1)$  for use in `\XINT_FL_fac_addzeros`.

With 1.2f the formula is  $m=P+9+k$ ,  $Q \leftarrow (P+12+k)/8$ , and we use now  $4=12-8$  rather than the earlier  $5=13-8$ . Whatever happens, the value computed in `\XINT_FL_fac_increaseP` is at least 8. There will always be an extra block.

Note: with Digits:=32; Maple gives for 200!:

```
> factorial(200.);
```

```
375
```

```
0.78865786736479050355236321393218 10
```

My 1.2f routine (and also 1.2) outputs:

```
7.8865786736479050355236321393219e374
```

and this is the correct rounding because for 40 digits it computes

```
7.886578673647905035523632139321850622951e374
```

Maple's result (contrarily to `xint`) is thus not the correct rounding but still it is less than  $0.6 \text{ ulp}$  wrong.

```

3048 \def\XINT_FL_fac_vbig
3049     {\expandafter\XINT_FL_fac_vbigloop_a
3050     \the\numexpr \XINT_FL_fac_increaseP \xint_c_i }%
3051 \def\XINT_FL_fac_big
3052     {\expandafter\XINT_FL_fac_bigloop_a
3053     \the\numexpr \XINT_FL_fac_increaseP \xint_c_ii }%
3054 \def\XINT_FL_fac_med
3055     {\expandafter\XINT_FL_fac_medloop_a

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

3056 \the\numexpr \XINT_FL_fac_increasP \xint_c_iii }%
3057 \def\XINT_FL_fac_small
3058 {\expandafter\XINT_FL_fac_smallloop_a
3059 \the\numexpr \XINT_FL_fac_increasP \xint_c_iv }%
3060 \def\XINT_FL_fac_increasP #1#2.#3#4%
3061 {%
3062 #2\expandafter.\the\numexpr\xint_c_viii*%
3063 ((\xint_c_iv+#4+\expandafter\XINT_FL_fac_countdigits
3064 \the\numexpr #2/(#1*#3)\relax 87654321\Z)/\xint_c_viii).%
3065 }%
3066 \def\XINT_FL_fac_countdigits #1#2#3#4#5#6#7#8{\XINT_FL_fac_countdone }%
3067 \def\XINT_FL_fac_countdone #1#2\Z {#1}%
3068 \def\XINT_FL_fac_out #1;![#2]#3%
3069 {#3{\romannumeral0\XINT_mul_out
3070 #1;!\R!1\R!1\R!1\R!1\R!%
3071 1\R!1\R!1\R!1\R!1\R!\W [#2]}}%
3072 \def\XINT_FL_fac_vbigloop_a #1.#2.%
3073 {%
3074 \XINT_FL_fac_bigloop_a \xint_c_x^iv.#2.%
3075 {\expandafter\XINT_FL_fac_vbigloop_loop\the\numexpr 100010001\expandafter.%
3076 \the\numexpr \xint_c_x^viii+#1.}%
3077 }%
3078 \def\XINT_FL_fac_vbigloop_loop #1.#2.%
3079 {%
3080 \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
3081 \expandafter\XINT_FL_fac_vbigloop_loop
3082 \the\numexpr #1+\xint_c_i\expandafter.%
3083 \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_mul #1!%
3084 }%
3085 \def\XINT_FL_fac_bigloop_a #1.%
3086 {%
3087 \expandafter\XINT_FL_fac_bigloop_b \the\numexpr
3088 #1+\xint_c_i-\xint_c_ii*((#1-464)/\xint_c_ii).#1.%
3089 }%
3090 \def\XINT_FL_fac_bigloop_b #1.#2.#3.%
3091 {%
3092 \expandafter\XINT_FL_fac_medloop_a
3093 \the\numexpr #1-\xint_c_i.#3.{\XINT_FL_fac_bigloop_loop #1.#2.}%
3094 }%
3095 \def\XINT_FL_fac_bigloop_loop #1.#2.%
3096 {%
3097 \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
3098 \expandafter\XINT_FL_fac_bigloop_loop
3099 \the\numexpr #1+\xint_c_ii\expandafter.%
3100 \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_bigloop_mul #1!%
3101 }%
3102 \def\XINT_FL_fac_bigloop_mul #1!%
3103 {%
3104 \expandafter\XINT_FL_fac_mul
3105 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
3106 }%
3107 \def\XINT_FL_fac_medloop_a #1.%

```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```
3108 {%
3109     \expandafter\XINT_FL_fac_medloop_b
3110         \the\numexpr #1+\xint_c_i-\xint_c_iii*((#1-100)/\xint_c_iii).#1.%
3111 }%
3112 \def\XINT_FL_fac_medloop_b #1.#2.#3.%
3113 {%
3114     \expandafter\XINT_FL_fac_smallloop_a
3115         \the\numexpr #1-\xint_c_i.#3.{\XINT_FL_fac_medloop_loop #1.#2.}%
3116 }%
3117 \def\XINT_FL_fac_medloop_loop #1.#2.%
3118 {%
3119     \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
3120     \expandafter\XINT_FL_fac_medloop_loop
3121     \the\numexpr #1+\xint_c_iii\expandafter.%
3122     \the\numexpr #2\expandafter.\the\numexpr\XINT_FL_fac_medloop_mul #1!%
3123 }%
3124 \def\XINT_FL_fac_medloop_mul #1!%
3125 {%
3126     \expandafter\XINT_FL_fac_mul
3127     \the\numexpr
3128         \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
3129 }%
3130 \def\XINT_FL_fac_smallloop_a #1.%
3131 {%
3132     \csname
3133         XINT_FL_fac_smallloop_\the\numexpr #1-\xint_c_iv*(#1/\xint_c_iv)\relax
3134     \endcsname #1.%
3135 }%
3136 \expandafter\def\csname XINT_FL_fac_smallloop_1\endcsname #1.#2.%
3137 {%
3138     \XINT_FL_fac_addzeros #2.100000001!.{2.#1.}{#2}%
3139 }%
3140 \expandafter\def\csname XINT_FL_fac_smallloop_-2\endcsname #1.#2.%
3141 {%
3142     \XINT_FL_fac_addzeros #2.100000002!.{3.#1.}{#2}%
3143 }%
3144 \expandafter\def\csname XINT_FL_fac_smallloop_-1\endcsname #1.#2.%
3145 {%
3146     \XINT_FL_fac_addzeros #2.100000006!.{4.#1.}{#2}%
3147 }%
3148 \expandafter\def\csname XINT_FL_fac_smallloop_0\endcsname #1.#2.%
3149 {%
3150     \XINT_FL_fac_addzeros #2.100000024!.{5.#1.}{#2}%
3151 }%
3152 \def\XINT_FL_fac_addzeros #1.%
3153 {%
3154     \ifnum #1=\xint_c_viii \expandafter\XINT_FL_fac_addzeros_exit\fi
3155     \expandafter\XINT_FL_fac_addzeros
3156     \the\numexpr #1-\xint_c_viii.100000000!%
3157 }
```

We will manipulate by successive \*small\* multiplications Q blocks 1<8d>!, terminated by 1;! . We need a custom small multiplication which tells us when it has create a new block, and the least



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

significant one should be dropped.

```
3158 \def\XINT_FL_fac_addzeros_exit #1.#2.#3#4{\XINT_FL_fac_smallloop_loop #3#21;![ -#4]}%
3159 \def\XINT_FL_fac_smallloop_loop #1.#2.%
3160 {%
3161   \ifnum #1>#2 \expandafter\XINT_FL_fac_loop_exit\fi
3162   \expandafter\XINT_FL_fac_smallloop_loop
3163   \the\numexpr #1+\xint_c_iv\expandafter.%
3164   \the\numexpr #2\expandafter.\romannumeral0\XINT_FL_fac_smallloop_mul #1!%
3165 }%
3166 \def\XINT_FL_fac_smallloop_mul #1!%
3167 {%
3168   \expandafter\XINT_FL_fac_mul
3169   \the\numexpr
3170     \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
3171 }%[[
3172 \def\XINT_FL_fac_loop_exit #1!#2]#3{#3#2]}%
3173 \def\XINT_FL_fac_mul 1#1!%
3174   {\expandafter\XINT_FL_fac_mul_a\the\numexpr\XINT_FL_fac_smallmul 10!{#1}}%
3175 \def\XINT_FL_fac_mul_a #1-#2%
3176 {%
3177   \if#21\xint_afterfi{\expandafter\space\xint_gob_til_exclam}\else
3178   \expandafter\space\fi #11; !%
3179 }%
3180 \def\XINT_FL_fac_minimulwc_a #1#2#3#4#5!#6#7#8#9%
3181 {%
3182   \XINT_FL_fac_minimulwc_b {#1#2#3#4}{#5}{#6#7#8#9}%
3183 }%
3184 \def\XINT_FL_fac_minimulwc_b #1#2#3#4!#5%
3185 {%
3186   \expandafter\XINT_FL_fac_minimulwc_c
3187   \the\numexpr \xint_c_x^ix+#5+#2*#4!{{#1}{#2}{#3}{#4}}%
3188 }%
3189 \def\XINT_FL_fac_minimulwc_c 1#1#2#3#4#5#6!#7%
3190 {%
3191   \expandafter\XINT_FL_fac_minimulwc_d {#1#2#3#4#5}#7{#6}%
3192 }%
3193 \def\XINT_FL_fac_minimulwc_d #1#2#3#4#5%
3194 {%
3195   \expandafter\XINT_FL_fac_minimulwc_e
3196   \the\numexpr \xint_c_x^ix+#1+#2*#5+#3*#4!{#2}{#4}%
3197 }%
3198 \def\XINT_FL_fac_minimulwc_e 1#1#2#3#4#5#6!#7#8#9%
3199 {%
3200   1#6#9\expandafter!%
3201   \the\numexpr\expandafter\XINT_FL_fac_smallmul
3202   \the\numexpr \xint_c_x^viii+#1#2#3#4#5+#7*#8!%
3203 }%
3204 \def\XINT_FL_fac_smallmul 1#1!#21#3!%
3205 {%
3206   \xint_gob_til_sc #3\XINT_FL_fac_smallmul_end;%
3207   \XINT_FL_fac_minimulwc_a #2!#3!{#1}{#2}%
3208 }%
```

This is the crucial ending. I note that I used here an `\ifnum` test rather than the `gob_til_eightzeroes` thing. Actually for eight digits there is much less difference than for only four.

The "carry" situation is marked by a final `!-1` rather than `!-2` for no-carry. (a `\numexpr` must be stopped, and leaving a `-` as delimiter is good as it will not arise earlier.)

```

3209 \def\XINT_FL_fac_smallmul_end;\XINT_FL_fac_minimulwc_a #1!!#2#3[#4]%
3210 {%
3211   \ifnum #2=\xint_c_
3212     \expandafter\xint_firstoftwo\else
3213     \expandafter\xint_secondoftwo
3214   \fi
3215   {-2\relax[#4]}%
3216   {1#2\expandafter!\expandafter-\expandafter1\expandafter
3217     [\the\numexpr #4+\xint_c_viii]}%
3218 }%
```

## 24.92. `\xintFloatPFactorial`, `\XINTinFloatPFactorial`

**Added at 1.2f (2016/03/12) [on 2015/11/29].** Partial factorial `pfactorial(a,b)=(a+1)...``b`, only for non-negative integers with `a<=b<10^8`.

**Modified at 1.2h (2016/11/20).** Now avoids raising `\xintError:OutOfRangePFac` if the condition `0<=a<=b<10^8` is violated. Same as for `\xintiiPFactorial`.

**Modified at 1.4e (2021/05/05).** 1.4e extends the precision in floating point context adding some overhead but well.

```

3219 \def\xintFloatPFactorial {\romannumeral0\xintfloatpfactorial}%
3220 \def\xintfloatpfactorial #1{\XINT_flpfac_chkopt \xintfloat #1\xint:}%
3221 \def\XINTinFloatPFactorial{\romannumeral0\XINTinfloatpfactorial}%
3222 \def\XINTinfloatpfactorial{\XINT_flpfac_opt_a\XINTdigits.\XINTinfloatS}%
3223 \def\XINT_flpfac_chkopt #1#2%
3224 {%
3225   \ifx [#2\expandafter\XINT_flpfac_opt
3226     \else\expandafter\XINT_flpfac_noopt
3227   \fi
3228   #1#2%
3229 }%
3230 \def\XINT_flpfac_noopt #1#2\xint:#3%
3231 {%
3232   \expandafter\XINT_FL_pfac_fork
3233   \the\numexpr \xintNum{#2}\expandafter.%
3234   \the\numexpr \xintNum{#3}.\xint_c_i{\XINTdigits}{#1[\XINTdigits]}%
3235 }%
3236 \def\XINT_flpfac_opt #1[\xint:#2]%
3237 {%
3238   \expandafter\XINT_flpfac_opt_a\the\numexpr #2.#1%
3239 }%
3240 \def\XINT_flpfac_opt_a #1.#2#3#4%
3241 {%
3242   \expandafter\XINT_FL_pfac_fork
3243   \the\numexpr \xintNum{#3}\expandafter.%
3244   \the\numexpr \xintNum{#4}.\xint_c_i{#1}{#2[#1]}%
3245 }%
3246 \def\XINT_FL_pfac_fork #1#2.#3#4.%
3247 {%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

3248 \unless\ifnum #1#2<#3#4 \xint_dothis\XINT_FL_pfac_one\fi
3249 \if-#3\xint_dothis\XINT_FL_pfac_neg \fi
3250 \if-#1\xint_dothis\XINT_FL_pfac_zero\fi
3251 \ifnum #3#4>\xint_c_x^viii_mone\xint_dothis\XINT_FL_pfac_outofrange\fi
3252 \xint_orthat \XINT_FL_pfac_increaseP #1#2.#3#4.%
3253 }%
3254 \def\XINT_FL_pfac_outofrange #1.#2.#3#4#5%
3255 {%
3256     #5{\XINT_signalcondition{InvalidOperation}
3257         {pFactorial with too large argument: #2 >= 10^8.}}{ 0[0]}}%
3258 }%
3259 \def\XINT_FL_pfac_one #1.#2.#3#4#5{#5{1[0]}}%
3260 \def\XINT_FL_pfac_zero #1.#2.#3#4#5{#5{0[0]}}%
3261 \def\XINT_FL_pfac_neg -#1.-#2.%
3262 {%
3263     \ifnum #1>\xint_c_x^viii\xint_dothis\XINT_FL_pfac_outofrange\fi
3264     \xint_orthat {%
3265         \ifodd\numexpr#2-#1\relax\xint_afterfi{\expandafter-\romannumeral`&&@}\fi
3266         \expandafter\XINT_FL_pfac_increaseP}%
3267     \the\numexpr #2-\xint_c_i\expandafter.\the\numexpr#1-\xint_c_i.%
3268 }%

```

See the comments for `\XINT_FL_pfac_increaseP`. Case of  $b=a+1$  should be filtered out perhaps. We only needed here to copy the `\xintPFactorial` macros and re-use `\XINT_FL_fac_mul/\XINT_FL_fac_out`. Had to modify a bit `\XINT_FL_pfac_addzeroes`. We can enter here directly with #3 equal to specify the precision (the calculated value before final rounding has a relative error less than  $\#3 \cdot 10^{-\#4-1}$ ), and #5 would hold the macro doing the final rounding (or truncating, if I make a `FloatTrunc` available) to a given number of digits, possibly not #4. By default the #3 is 1, but `FloatBinomial` calls it with #3=4.

```

3269 \def\XINT_FL_pfac_increaseP #1.#2.#3#4%
3270 {%
3271     \expandafter\XINT_FL_pfac_a
3272     \the\numexpr \xint_c_viii*((\xint_c_iv+#4+\expandafter
3273         \XINT_FL_fac_countdigits\the\numexpr (#2-#1-\xint_c_i)%
3274         /\ifnum #2>\xint_c_x^iv #3\else(#3*\xint_c_ii)\fi\relax
3275         87654321\Z)/\xint_c_viii).#1.#2.%
3276 }%
3277 \def\XINT_FL_pfac_a #1.#2.#3.%
3278 {%
3279     \expandafter\XINT_FL_pfac_b\the\numexpr \xint_c_i+#2\expandafter.%
3280     \the\numexpr#3\expandafter.%
3281     \romannumeral0\XINT_FL_pfac_addzeroes #1.100000001!1;![ -#1]%
3282 }%
3283 \def\XINT_FL_pfac_addzeroes #1.%
3284 {%
3285     \ifnum #1=\xint_c_viii \expandafter\XINT_FL_pfac_addzeroes_exit\fi
3286     \expandafter\XINT_FL_pfac_addzeroes\the\numexpr #1-\xint_c_viii.100000000!%
3287 }%
3288 \def\XINT_FL_pfac_addzeroes_exit #1.{ }%
3289 \def\XINT_FL_pfac_b #1.%
3290 {%
3291     \ifnum #1>9999 \xint_dothis\XINT_FL_pfac_vbigloop \fi
3292     \ifnum #1>463 \xint_dothis\XINT_FL_pfac_bigloop \fi

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

3293     \ifnum #1>98     \xint_dothis\XINT_FL_pfac_medloop     \fi
3294                     \xint_orthat\XINT_FL_pfac_smallloop #1.%
3295 }%
3296 \def\XINT_FL_pfac_smallloop #1.#2.%
3297 {%
3298     \ifcase\numexpr #2-#1\relax
3299         \expandafter\XINT_FL_pfac_end_
3300     \or \expandafter\XINT_FL_pfac_end_i
3301     \or \expandafter\XINT_FL_pfac_end_ii
3302     \or \expandafter\XINT_FL_pfac_end_iii
3303     \else\expandafter\XINT_FL_pfac_smallloop_a
3304     \fi #1.#2.%
3305 }%
3306 \def\XINT_FL_pfac_smallloop_a #1.#2.%
3307 {%
3308     \expandafter\XINT_FL_pfac_smallloop_b
3309     \the\numexpr #1+\xint_c_iv\expandafter.%
3310     \the\numexpr #2\expandafter.%
3311     \romannumeral0\expandafter\XINT_FL_fac_mul
3312     \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
3313 }%
3314 \def\XINT_FL_pfac_smallloop_b #1.%
3315 {%
3316     \ifnum #1>98     \expandafter\XINT_FL_pfac_medloop     \else
3317                     \expandafter\XINT_FL_pfac_smallloop \fi #1.%
3318 }%
3319 \def\XINT_FL_pfac_medloop #1.#2.%
3320 {%
3321     \ifcase\numexpr #2-#1\relax
3322         \expandafter\XINT_FL_pfac_end_
3323     \or \expandafter\XINT_FL_pfac_end_i
3324     \or \expandafter\XINT_FL_pfac_end_ii
3325     \else\expandafter\XINT_FL_pfac_medloop_a
3326     \fi #1.#2.%
3327 }%
3328 \def\XINT_FL_pfac_medloop_a #1.#2.%
3329 {%
3330     \expandafter\XINT_FL_pfac_medloop_b
3331     \the\numexpr #1+\xint_c_iii\expandafter.%
3332     \the\numexpr #2\expandafter.%
3333     \romannumeral0\expandafter\XINT_FL_fac_mul
3334     \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
3335 }%
3336 \def\XINT_FL_pfac_medloop_b #1.%
3337 {%
3338     \ifnum #1>463 \expandafter\XINT_FL_pfac_bigloop     \else
3339                     \expandafter\XINT_FL_pfac_medloop     \fi #1.%
3340 }%
3341 \def\XINT_FL_pfac_bigloop #1.#2.%
3342 {%
3343     \ifcase\numexpr #2-#1\relax
3344         \expandafter\XINT_FL_pfac_end_

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
3345 \or \expandafter\XINT_FL_pfac_end_i
3346 \else\expandafter\XINT_FL_pfac_bigloop_a
3347 \fi #1.#2.%
3348 }%
3349 \def\XINT_FL_pfac_bigloop_a #1.#2.%
3350 {%
3351 \expandafter\XINT_FL_pfac_bigloop_b
3352 \the\numexpr #1+\xint_c_ii\expandafter.%
3353 \the\numexpr #2\expandafter.%
3354 \romannumeral0\expandafter\XINT_FL_fac_mul
3355 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
3356 }%
3357 \def\XINT_FL_pfac_bigloop_b #1.%
3358 {%
3359 \ifnum #1>9999 \expandafter\XINT_FL_pfac_vbigloop \else
3360 \expandafter\XINT_FL_pfac_bigloop \fi #1.%
3361 }%
3362 \def\XINT_FL_pfac_vbigloop #1.#2.%
3363 {%
3364 \ifnum #2=#1
3365 \expandafter\XINT_FL_pfac_end_
3366 \else\expandafter\XINT_FL_pfac_vbigloop_a
3367 \fi #1.#2.%
3368 }%
3369 \def\XINT_FL_pfac_vbigloop_a #1.#2.%
3370 {%
3371 \expandafter\XINT_FL_pfac_vbigloop
3372 \the\numexpr #1+\xint_c_i\expandafter.%
3373 \the\numexpr #2\expandafter.%
3374 \romannumeral0\expandafter\XINT_FL_fac_mul
3375 \the\numexpr\xint_c_x^viii+#1!%
3376 }%
3377 \def\XINT_FL_pfac_end_iii #1.#2.%
3378 {%
3379 \expandafter\XINT_FL_fac_out
3380 \romannumeral0\expandafter\XINT_FL_fac_mul
3381 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)*(#1+\xint_c_iii)!%
3382 }%
3383 \def\XINT_FL_pfac_end_ii #1.#2.%
3384 {%
3385 \expandafter\XINT_FL_fac_out
3386 \romannumeral0\expandafter\XINT_FL_fac_mul
3387 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)*(#1+\xint_c_ii)!%
3388 }%
3389 \def\XINT_FL_pfac_end_i #1.#2.%
3390 {%
3391 \expandafter\XINT_FL_fac_out
3392 \romannumeral0\expandafter\XINT_FL_fac_mul
3393 \the\numexpr \xint_c_x^viii+#1*(#1+\xint_c_i)!%
3394 }%
3395 \def\XINT_FL_pfac_end_ #1.#2.%
3396 {%
```

```

3397 \expandafter\XINT_FL_fac_out
3398 \romannumeral0\expandafter\XINT_FL_fac_mul
3399 \the\numexpr \xint_c_x^viii+1!%
3400 }%

```

### 24.93. \xintFloatBinomial, \XINTinFloatBinomial

**Added at 1.2f (2016/03/12) [on 2015/12/01].** We compute  $\text{binomial}(x,y)$  as  $\text{pfac}(x-y,x)/y!$ , where the numerator and denominator are computed with a relative error at most  $4.10^{-P-2}$ , then rounded (once I have a float truncation, I will use truncation rather) to  $P+3$  digits, and finally the quotient is correctly rounded to  $P$  digits. This will guarantee that the exact value  $X$  differs from the computed one  $Y$  by at most  $0.6 \text{ ulp}(Y)$ .

**Modified at 1.2h (2016/11/20).** As for [\xintiiBinomial](#), hard to understand why last year I coded this to raise an error if  $y < 0$  or  $y > x$  ! The question of the Gamma function is for another occasion, here  $x$  and  $y$  must be (small) integers.

1.4e: same remarks as for factorial and partial factorial about added overhead due to extra guard digits.

```

3401 \def\xintFloatBinomial {\romannumeral0\xintfloatbinomial}%
3402 \def\xintfloatbinomial #1{\XINT_flbinom_chkopt \xintfloat #1\xint:}%
3403 \def\XINTinFloatBinomial{\romannumeral0\XINTinfloatbinomial}%
3404 \def\XINTinfloatbinomial{\XINT_flbinom_opt\XINTinfloatS[\xint:\XINTdigits]}%
3405 \def\XINT_flbinom_chkopt #1#2%
3406 {%
3407   \ifx [#2\expandafter\XINT_flbinom_opt
3408     \else\expandafter\XINT_flbinom_noopt
3409   \fi #1#2%
3410 }%
3411 \def\XINT_flbinom_noopt #1#2\xint:#3%
3412 {%
3413   \expandafter\XINT_FL_binom_a
3414   \the\numexpr\xintNum{#2}\expandafter.\the\numexpr\xintNum{#3}.\XINTdigits.#1%
3415 }%
3416 \def\XINT_flbinom_opt #1[\xint:#2]#3#4%
3417 {%
3418   \expandafter\XINT_FL_binom_a
3419   \the\numexpr\xintNum{#3}\expandafter.\the\numexpr\xintNum{#4}\expandafter.%
3420   \the\numexpr #2.#1%
3421 }%
3422 \def\XINT_FL_binom_a #1.#2.%
3423 {%
3424   \expandafter\XINT_FL_binom_fork \the\numexpr #1-#2.#2.#1.%
3425 }%
3426 \def\XINT_FL_binom_fork #1#2.#3#4.#5#6.%
3427 {%
3428   \if-#5\xint_dothis \XINT_FL_binom_neg\fi
3429   \if-#1\xint_dothis \XINT_FL_binom_zero\fi
3430   \if-#3\xint_dothis \XINT_FL_binom_zero\fi
3431   \if0#1\xint_dothis \XINT_FL_binom_one\fi
3432   \if0#3\xint_dothis \XINT_FL_binom_one\fi
3433   \ifnum #5#6>\xint_c_x^viii_mone \xint_dothis\XINT_FL_binom_toobig\fi
3434   \ifnum #1#2>#3#4 \xint_dothis\XINT_FL_binom_ab \fi
3435   \xint_orthat\XINT_FL_binom_aa

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```
3436      #1#2.#3#4.#5#6.%
3437 }%
3438 \def\XINT_FL_binom_neg #1.#2.#3.#4.#5%
3439 {%
3440     #5[#4]{\XINT_signalcondition{InvalidOperation}
3441             {Binomial with negative argument: #3.}{0[0]}}%
3442 }%
3443 \def\XINT_FL_binom_toobig #1.#2.#3.#4.#5%
3444 {%
3445     #5[#4]{\XINT_signalcondition{InvalidOperation}
3446             {Binomial with too large argument: #3 >= 10^8.}{0[0]}}%
3447 }%
3448 \def\XINT_FL_binom_one #1.#2.#3.#4.#5{#5[#4]{1[0]}}%
3449 \def\XINT_FL_binom_zero #1.#2.#3.#4.#5{#5[#4]{0[0]}}%
3450 \def\XINT_FL_binom_aa #1.#2.#3.#4.#5%
3451 {%
3452     #5[#4]{\xintDiv{\XINT_FL_pfac_increaseP
3453                   #2.#3.\xint_c_iv{#4+\xint_c_i}{\XINTinfloat[#4+\xint_c_iii]}}%
3454             {\XINT_FL_fac_fork_b
3455             #1.\xint_c_iv{#4+\xint_c_i}\XINT_FL_fac_out{\XINTinfloat[#4+\xint_c_iii]}}}%
3456 }%
3457 \def\XINT_FL_binom_ab #1.#2.#3.#4.#5%
3458 {%
3459     #5[#4]{\xintDiv{\XINT_FL_pfac_increaseP
3460                   #1.#3.\xint_c_iv{#4+\xint_c_i}{\XINTinfloat[#4+\xint_c_iii]}}%
3461             {\XINT_FL_fac_fork_b
3462             #2.\xint_c_iv{#4+\xint_c_i}\XINT_FL_fac_out{\XINTinfloat[#4+\xint_c_iii]}}}%
3463 }%
```

## 24.94. \xintFloatSqrt, \XINTinFloatSqrt

Added at 1.08 (2013/06/07).

Modified at 1.2f (2016/03/12).

The float version was developed at the same time as the integer one and even a bit earlier. As a result the integer variant had some sub-optimal parts. Anyway, for 1.2f I have rewritten the integer variant, and the float variant delegates all preparatory work for it until the last step. In particular the very low precisions are not penalized anymore from doing computations for at least 17 or 18 digits. Both the large and small precisions give quite shorter computation times.

Also, after examining more closely the achieved precision I decided to extend the float version in order for it to obtain the correct rounding (for inputs already of at most P digits with P the precision) of the theoretical exact value.

Beyond about 500 digits of precision the efficiency decreases swiftly, as is the case generally speaking with *xintcore*/*xint*/*xintfrac* arithmetic macros.

Final note: with 1.2f the input is always first rounded to P significant places.

```
3464 \def\xintFloatSqrt {\romannumeral0\xintfloatsqrt}%
3465 \def\xintfloatsqrt #1{\XINT_flsqrt_chkopt \xintfloat #1\xint:}%
3466 \def\XINTinFloatSqrt{\romannumeral0\XINTinfloatsqrt}%
3467 \def\XINTinfloatsqrt[#1]{\expandafter\XINT_flsqrt_opt_a\the\numexpr#1.\XINTinfloatS}%
3468 \def\XINTinFloatSqrtdigits{\romannumeral0\XINT_flsqrt_opt_a\XINTdigits.\XINTinfloatS}%
3469 \def\XINT_flsqrt_chkopt #1#2%
3470 {%
3471     \ifx [#2\expandafter\XINT_flsqrt_opt
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

3472     \else\expandafter\XINT_flsqrt_noopt
3473     \fi #1#2%
3474 }%
3475 \def\XINT_flsqrt_noopt #1#2\xint:%
3476 {%
3477     \expandafter\XINT_FL_sqrt_a
3478         \romannumeral0\XINTinfloat[\XINTdigits]{#2}\XINTdigits.#1%
3479 }%
3480 \def\XINT_flsqrt_opt #1[\xint:#2]#3%
3481 {%
3482     \expandafter\XINT_flsqrt_opt_a\the\numexpr #2.#1%
3483 }%
3484 \def\XINT_flsqrt_opt_a #1.#2#3%
3485 {%
3486     \expandafter\XINT_FL_sqrt_a\romannumeral0\XINTinfloat[#1]{#3}#1.#2%
3487 }%
3488 \def\XINT_FL_sqrt_a #1%
3489 {%
3490     \xint_UDzerominusfork
3491     #1-\XINT_FL_sqrt_iszero
3492     0#1\XINT_FL_sqrt_isneg
3493     0-\XINT_FL_sqrt_pos #1}%
3494 \kroft
3495 }%[
3496 \def\XINT_FL_sqrt_iszero #1]#2.#3{#3[#2]{0[0]}}%
3497 \def\XINT_FL_sqrt_isneg #1]#2.#3%
3498 {%
3499     #3[#2]{\XINT_signalcondition{InvalidOperation}
3500         {Square root of negative: -#1].}{#3}{0[0]}}%
3501 }%
3502 \def\XINT_FL_sqrt_pos #1[#2]#3.%
3503 {%
3504     \expandafter\XINT_flsqrt
3505     \the\numexpr #3\ifodd #2 \xint_dothis {+\xint_c_iii.(#2+\xint_c_i).0}\fi
3506     \xint_orthat {+\xint_c_ii.#2.{}}#100.#3.%
3507 }%
3508 \def\XINT_flsqrt #1.#2.%
3509 {%
3510     \expandafter\XINT_flsqrt_a
3511     \the\numexpr #2/\xint_c_ii-(#1-\xint_c_i)/\xint_c_ii.#1.%
3512 }%
3513 \def\XINT_flsqrt_a #1.#2.#3#4.#5.%
3514 {%
3515     \expandafter\XINT_flsqrt_b
3516     \the\numexpr (#2-\xint_c_i)/\xint_c_ii\expandafter.%
3517     \romannumeral0\XINT_sqrt_start #2.#4#3.#5.#2.#4#3.#5.#1.%
3518 }%
3519 \def\XINT_flsqrt_b #1.#2#3%
3520 {%
3521     \expandafter\XINT_flsqrt_c
3522     \romannumeral0\xintiisub
3523     {\XINT_dsx_addzeros {#1}#2;}%

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfrac*, *xintexpr*, *xinttrig*, *xintlog*

```

3524     {\xintiiDivRound{\XINT_dsx_addzeros {#1}#3;}}%
3525         {\XINT_dbl#2\xint_bye2345678\xint_bye*\xint_c_ii\relax}}}%
3526 }%
3527 \def\XINT_flsqrt_c #1.#2.%
3528 {%
3529     \expandafter\XINT_flsqrt_d
3530     \romannumeral0\XINT_split_fromleft#2.#1\xint_bye2345678\xint_bye..%
3531 }%
3532 \def\XINT_flsqrt_d #1.#2#3.%
3533 {%
3534     \ifnum #2=\xint_c_v
3535     \expandafter\XINT_flsqrt_f\else\expandafter\XINT_flsqrt_finish\fi
3536     #2#3.#1.%
3537 }%
3538 \def\XINT_flsqrt_finish #1#2.#3.#4.#5.#6.#7.#8{#8[#6]{#3#1[#7]}}%
3539 \def\XINT_flsqrt_f 5#1.%
3540     {\expandafter\XINT_flsqrt_g\romannumeral0\xintinum{#1}\relax.}%
3541 \def\XINT_flsqrt_g #1#2#3.{\if\relax#2\xint_dothis{\XINT_flsqrt_h #1}\fi
3542     \xint_orthat{\XINT_flsqrt_finish 5.}}%
3543 \def\XINT_flsqrt_h #1{\ifnum #1<\xint_c_iii\xint_dothis{\XINT_flsqrt_again}\fi
3544     \xint_orthat{\XINT_flsqrt_finish 5.}}%
3545 \def\XINT_flsqrt_again #1.#2.%
3546 {%
3547     \expandafter\XINT_flsqrt_again_a\the\numexpr #2+\xint_c_viii.%
3548 }%
3549 \def\XINT_flsqrt_again_a #1.#2.#3.%
3550 {%
3551     \expandafter\XINT_flsqrt_b
3552     \the\numexpr (#1-\xint_c_i)/\xint_c_ii\expandafter.%
3553     \romannumeral0\XINT_sqrt_start #1.#200000000.#3.%
3554     #1.#200000000.#3.%
3555 }%

```

### 24.95. \xintFloatE, \XINTinFloatE

Added at 1.07 (2013/05/25). The fraction is the first argument contrarily to `\xintTrunc` and `\xintRound`.

Attention to `\XINTinFloatE`: it is for use by `xintexpr`. With input 0 it produces on output an 0[N], not 0[0].

```

3556 \def\xintFloatE {\romannumeral0\xintfloate }%
3557 \def\xintfloate #1{\XINT_floate_chkopt #1\xint:}%
3558 \def\XINT_floate_chkopt #1%
3559 {%
3560     \ifx [#1\expandafter\XINT_floate_opt
3561     \else\expandafter\XINT_floate_noopt
3562     \fi #1%
3563 }%
3564 \def\XINT_floate_noopt #1\xint:%
3565 {%
3566     \expandafter\XINT_floate_post
3567     \romannumeral0\XINTinfloat[\XINTdigits]{#1}\XINTdigits.%
3568 }%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, xintfrac, *xintseries*, *xintcfraction*, *xintexpr*, *xinttrig*, *xintlog*

```
3569 \def\XINT_floate_opt [\xint:#1]%
3570 {%
3571     \expandafter\XINT_floate_opt_a\the\numexpr #1.%
3572 }%
3573 \def\XINT_floate_opt_a #1.#2%
3574 {%
3575     \expandafter\XINT_floate_post
3576     \romannumeral0\XINTinfloat[#1]{#2}#1.%
3577 }%
3578 \def\XINT_floate_post #1%
3579 {%
3580     \xint_UDzerominusfork
3581     #1-\XINT_floate_zero
3582     0#1\XINT_floate_neg
3583     0-\XINT_floate_pos
3584     \krof #1%
3585 }%[
3586 \def\XINT_floate_zero #1]#2.#3{ 0.e0}%
3587 \def\XINT_floate_neg-{\expandafter-\romannumeral0\XINT_floate_pos}%
3588 \def\XINT_floate_pos #1#2[#3]#4.#5%
3589 {%
3590     \expandafter\XINT_float_pos_done\the\numexpr#3+#4+#5-\xint_c_i.#1.#2;%
3591 }%
3592 \def\XINTinFloatE {\romannumeral0\XINTinfloat}%
3593 \def\XINTinfloat
3594     {\expandafter\XINT_infloat\romannumeral0\XINTinfloat[\XINTdigits]}%
3595 \def\XINT_infloat #1[#2]#3%
3596     {\expandafter\XINT_infloat_end\the\numexpr #3+#2.{#1}}%
3597 \def\XINT_infloat_end #1.#2{ #2[#1]}%
```

### 24.96. \XINTinFloatMod

Added at 1.1 (2014/10/28). Pour emploi dans *xintexpr*. Code shortened at 1.2p.

```
3598 \def\XINTinFloatMod {\romannumeral0\XINTinfloatmod [\XINTdigits]}%
3599 \def\XINTinfloatmod [#1]#2#3%
3600 {%
3601     \XINTinfloat[#1]{\xintMod
3602         {\romannumeral0\XINTinfloat[#1]{#2}}%
3603         {\romannumeral0\XINTinfloat[#1]{#3}}}%
3604 }%
```

### 24.97. \XINTinFloatDivFloor

Added at 1.2p (2017/12/05). Formerly // and /: in *\xintfloatexpr* used *\xintDivFloor* and *\xintMod*, hence did not round their operands to float precision beforehand.

```
3605 \def\XINTinFloatDivFloor {\romannumeral0\XINTinfloatdivfloor [\XINTdigits]}%
3606 \def\XINTinfloatdivfloor [#1]#2#3%
3607 {%
3608     \xintdivfloor
3609     {\romannumeral0\XINTinfloat[#1]{#2}}%
3610     {\romannumeral0\XINTinfloat[#1]{#3}}}%
3611 }%
```

## 24.98. \XINTinFloatDivMod

**Added at 1.2p (2017/12/05).** Pour emploi dans `xintexpr`, donc je ne prends pas la peine de faire l'expansion du modulo, qui se produira dans le `\csname`.

Hésitation sur le quotient, faut-il l'arrondir immédiatement ? Finalement non, le produire comme un integer.

Breaking change at 1.4 as output format is not comma separated anymore. Attention also that it uses `\expanded`.

No time now at the time of completion of the big 1.4 rewrite of `xintexpr` to test whether code efficiency here can be improved to expand the second item of output.

```

3612 \def\XINTinFloatDivMod {\romannumeral0\XINTinfloatdivmod [\XINTdigits]}%
3613 \def\XINTinfloatdivmod [#1]#2#3%
3614 {%
3615   \expandafter\XINT_infloatdivmod
3616   \romannumeral0\xintdivmod
3617       {\romannumeral0\XINTinfloat[#1]{#2}}%
3618       {\romannumeral0\XINTinfloat[#1]{#3}}%
3619   [#1]}%
3620 }%
3621 \def\XINT_infloatdivmod #1#2#3{\expanded{[#1]{\XINTinFloat[#3]{#2}}}}%

```

## 24.99. \xintifFloatInt

**Added at 1.3a (2018/03/07).** For `ifint()` function in `\xintfloatexpr`.

```

3622 \def\xintifFloatInt {\romannumeral0\xintiffloatint}%
3623 \def\xintiffloatint #1{\expandafter\XINT_iffloatint
3624   \romannumeral0\xintrez{\XINTinFloatS[\XINTdigits]{#1}}}%
3625 \def\XINT_iffloatint #1#2/1[#3]%
3626 {%
3627   \if 0#1\xint_dothis\xint_stop_atfirstoftwo\fi
3628   \ifnum#3<\xint_c\xint_dothis\xint_stop_atsecondoftwo\fi
3629   \xint_orthat\xint_stop_atfirstoftwo
3630 }%

```

## 24.100. \xintFloatIsInt

**Added at 1.3d (2019/01/06).** For `isint()` function in `\xintfloatexpr`.

```

3631 \def\xintFloatIsInt {\romannumeral0\xintfloatisint}%
3632 \def\xintfloatisint #1{\expandafter\XINT_iffloatint
3633   \romannumeral0\xintrez{\XINTinFloatS[\XINTdigits]{#1}}10}%

```

## 24.101. \xintFloatIntType

**Added at 1.4e (2021/05/05).** For fractional powers. Expands to `\xint_c_mone` if argument is not an integer, to `\xint_c_` if it is an even integer and to `\xint_c_i` if it is an odd integer.

```

3634 \def\xintFloatIntType {\romannumeral`&&\xintfloatinttype}%
3635 \def\xintfloatinttype #1%
3636 {%
3637   \expandafter\XINT_floatinttype
3638   \romannumeral0\xintrez{\XINTinFloatS[\XINTdigits]{#1}}%
3639 }%

```

```

3640 \def\XINT_floatinttype #1#2/1[#3]%
3641 {%
3642     \if 0#1\xint_dothis\xint_c_\fi
3643     \ifnum#3<\xint_c_\xint_dothis\xint_c_mone\fi
3644     \ifnum#3>\xint_c_\xint_dothis\xint_c_\fi
3645     \ifodd\xintLDg{#1#2} \xint_dothis\xint_c_i\fi
3646     \xint_orthat\xint_c_
3647 }%

```

## 24.102. \XINTinFloatdigits, \XINTinFloatSdigits

```

3648 \def\XINTinFloatdigits {\XINTinFloat [\XINTdigits]}%
3649 \def\XINTinFloatSdigits{\XINTinFloatS[\XINTdigits]}%

```

## 24.103. (WIP) \XINTinRandomFloatS, \XINTinRandomFloatSdigits

Added at 1.3b (2018/05/18). Support for random() function.

Thus as it is a priori only for xintexpr usage, it expands inside `\csname` context, but as we need to get rid of initial zeros we use `\xintRandomDigits` not `\xintXRandomDigits` (`\expanded` would have a use case here).

And anyway as we want to be able to use random() in `\xintdefunc/\xintNewExpr`, it is good to have f-expandable macros, so we add the small overhead to make it f-expandable.

We don't have to be very efficient in removing leading zeroes, as there is only 10% chance for each successive one. Besides we use (current) internal storage format of the type A[N], where A is not required to be with `\xintDigits` digits, so N will simply be `-\xintDigits` and needs no adjustment.

In case we use in future with #1 something else than `\xintDigits` we do the 0-(#1) construct.

I had some qualms about doing a random float like this which means that when there are leading zeros in the random digits the (virtual) mantissa ends up with trailing zeros. That did not feel right but I checked random() in Python (which of course uses radix 2), and indeed this is what happens there.

```

3650 \def\XINTinRandomFloatS{\romannumeral0\XINTinrandomfloatS}%
3651 \def\XINTinRandomFloatSdigits{\XINTinRandomFloatS[\XINTdigits]}%
3652 \def\XINTinrandomfloatS[#1]%
3653 {%
3654     \expandafter\XINT_inrandomfloatS\the\numexpr\xint_c_-(#1)\xint:
3655 }%
3656 \def\XINT_inrandomfloatS-#1\xint:
3657 {%
3658     \expandafter\XINT_inrandomfloatS_a
3659     \romannumeral0\xintrandomdigits{#1}[-#1]%
3660 }%

```

We add one macro to handle a tiny bit faster 90% of cases, after all we also use one extra macro for the completely improbable all 0 case.

```

3661 \def\XINT_inrandomfloatS_a#1%
3662 {%
3663     \if#10\xint_dothis{\XINT_inrandomfloatS_b}\fi
3664     \xint_orthat{ #1}%
3665 }%[
3666 \def\XINT_inrandomfloatS_b#1%
3667 {%
3668     \if#1[\xint_dothis{\XINT_inrandomfloatS_zero}\fi% ]

```

```

3669 \if#10\xint_dothis{\XINT_inrandomfloatS_b}\fi
3670 \xint_orthat{ #1}%
3671 }%[
3672 \def\xint_inrandomfloatS_zero#1]{ 0[0]}%

```

## 24.104. (WIP) \XINTinRandomFloatSixteen

Added at 1.3b (2018/05/18). Support for grand() function.

```

3673 \def\xint_inRandomFloatSixteen%
3674 {%
3675 \romannumeral0\expandafter\xint_inrandomfloatS_a
3676 \romannumeral`&&\expandafter\xint_eightrandomdigits
3677 \romannumeral`&&\xint_eightrandomdigits[-16]%
3678 }%
3679 \let\xint_inFloatMaxof\xint_Maxof
3680 \let\xint_inFloatMinof\xint_Minof
3681 \let\xint_inFloatSum\xint_Sum
3682 \let\xint_inFloatPrd\xint_Prd
3683 \xint_restorecatcodesendinginput%

```

## 25. Package *xintseries* implementation

.1	Catcodes, $\varepsilon$ -TeX and reload detection . .	525	.7	<code>\xintRationalSeries</code> . . . . .	528
.2	Package identification . . . . .	526	.8	<code>\xintRationalSeriesX</code> . . . . .	529
.3	<code>\xintSeries</code> . . . . .	526	.9	<code>\xintFxFtPowerSeries</code> . . . . .	530
.4	<code>\xintiSeries</code> . . . . .	526	.10	<code>\xintFxFtPowerSeriesX</code> . . . . .	531
.5	<code>\xintPowerSeries</code> . . . . .	527	.11	<code>\xintFloatPowerSeries</code> . . . . .	531
.6	<code>\xintPowerSeriesX</code> . . . . .	528	.12	<code>\xintFloatPowerSeriesX</code> . . . . .	533

The commenting is currently (2025/09/06) very sparse.

### 25.1. Catcodes, $\varepsilon$ -TeX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^
11 \def\empty{}\def\space{ }\newlinechar10
12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintseries.sty\endcsname
14 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
15 \expandafter\ifx\csname numexpr\endcsname\relax
16 \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
17 \immediate\write128{^^JPackage xintseries Warning:^^J}%
18 \space\space\space\space
19 \numexpr not available, aborting input.^^J}%
20 \else
21 \PackageWarningNoLine{xintseries}{\numexpr not available, aborting input}%
22 \fi
23 \def\z{\endgroup\endinput}%
24 \else
25 \ifx\x\relax % plain-TeX, first loading of xintseries.sty
26 \ifx\w\relax % but xintfrac.sty not yet loaded.
27 \def\z{\endgroup\input xintfrac.sty\relax}%
28 \fi
29 \else
30 \ifx\x\empty % LaTeX, first loading,
31 % variable is initialized, but \ProvidesPackage not yet seen
32 \ifx\w\relax % xintfrac.sty not yet loaded.
33 \def\z{\endgroup\RequirePackage{xintfrac}}%
34 \fi
35 \else
36 \def\z{\endgroup\endinput}% xintseries already loaded.
37 \fi
38 \fi

```

```

39 \fi
40 \z%
41 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 25.2. Package identification

```

42 \XINT_providespackage
43 \ProvidesPackage{xintseries}%
44 [2025/09/06 v1.4o Expandable partial sums with xint package (JFB)]%

```

## 25.3. \xintSeries

```

45 \def\xintSeries {\romannumeral0\xintseries }%
46 \def\xintseries #1#2%
47 {%
48   \expandafter\XINT_series\expandafter
49   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
50 }%
51 \def\XINT_series #1#2#3%
52 {%
53   \ifnum #2<#1
54     \xint_afterfi { 0/1[0]}%
55   \else
56     \xint_afterfi {\XINT_series_loop {#1}{0}{#2}{#3}}%
57   \fi
58 }%
59 \def\XINT_series_loop #1#2#3#4%
60 {%
61   \ifnum #3>#1 \else \XINT_series_exit \fi
62   \expandafter\XINT_series_loop\expandafter
63   {\the\numexpr #1+1\expandafter }\expandafter
64   {\romannumeral0\xintadd {#2}{#4{#1}}}%
65   {#3}{#4}%
66 }%
67 \def\XINT_series_exit \fi #1#2#3#4#5#6#7#8%
68 {%
69   \fi\xint_gobble_ii #6%
70 }%

```

## 25.4. \xintiSeries

```

71 \def\xintiSeries {\romannumeral0\xintiseries }%
72 \def\xintiseries #1#2%
73 {%
74   \expandafter\XINT_iseries\expandafter
75   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
76 }%
77 \def\XINT_iseries #1#2#3%
78 {%
79   \ifnum #2<#1
80     \xint_afterfi { 0}%
81   \else
82     \xint_afterfi {\XINT_iseries_loop {#1}{0}{#2}{#3}}%

```

```

83   \fi
84 }%
85 \def\XINT_iseries_loop #1#2#3#4%
86 {%
87   \ifnum #3>#1 \else \XINT_iseries_exit \fi
88   \expandafter\XINT_iseries_loop\expandafter
89   {\the\numexpr #1+1\expandafter }\expandafter
90   {\romannumeral0\xintiadd {#2}{#4{#1}}}%
91   {#3}{#4}%
92 }%
93 \def\XINT_iseries_exit \fi #1#2#3#4#5#6#7#8%
94 {%
95   \fi\xint_gobble_ii #6%
96 }%

```

## 25.5. \xintPowerSeries

The 1.03 version was very lame and created a build-up of denominators. (this was at a time `\xintAdd` always multiplied denominators, by the way) The Horner scheme for polynomial evaluation is used in 1.04, this cures the denominator problem and drastically improves the efficiency of the macro. Modified in 1.06 to give the indices first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```

97 \def\xintPowerSeries {\romannumeral0\xintpowerseries }%
98 \def\xintpowerseries #1#2%
99 {%
100   \expandafter\XINT_powseries\expandafter
101   {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
102 }%
103 \def\XINT_powseries #1#2#3#4%
104 {%
105   \ifnum #2<#1
106     \xint_afterfi { 0/1[0]}%
107   \else
108     \xint_afterfi
109     {\XINT_powseries_loop_i {#3}{#2}}{#1}{#2}{#3}{#4}%
110   \fi
111 }%
112 \def\XINT_powseries_loop_i #1#2#3#4#5%
113 {%
114   \ifnum #3>#2 \else\XINT_powseries_exit_i\fi
115   \expandafter\XINT_powseries_loop_ii\expandafter
116   {\the\numexpr #3-1\expandafter}\expandafter
117   {\romannumeral0\xintmul {#1}{#5}}{#2}{#4}{#5}%
118 }%
119 \def\XINT_powseries_loop_ii #1#2#3#4%
120 {%
121   \expandafter\XINT_powseries_loop_i\expandafter
122   {\romannumeral0\xintadd {#4{#1}}{#2}}{#3}{#1}{#4}%
123 }%
124 \def\XINT_powseries_exit_i\fi #1#2#3#4#5#6#7#8#9%
125 {%

```



```

126 \fi \XINT_powseries_exit_ii #6{#7}%
127 }%
128 \def\XINT_powseries_exit_ii #1#2#3#4#5#6%
129 {%
130 \xintmul{\xintPow {#5}{#6}}{#4}%
131 }%

```

## 25.6. \xintPowerSeriesX

Same as [\xintPowerSeries](#) except for the initial expansion of the  $x$  parameter. Modified in 1.06 to give the indices first to a [\numexpr](#) rather than expanding twice. I just use [\the\numexpr](#) and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```

132 \def\xintPowerSeriesX {\romannumeral0\xintpowerseriesx }%
133 \def\xintpowerseriesx #1#2%
134 {%
135 \expandafter\XINT_powseriesx\expandafter
136 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
137 }%
138 \def\XINT_powseriesx #1#2#3#4%
139 {%
140 \ifnum #2<#1
141 \xint_afterfi { 0/1[0]}%
142 \else
143 \xint_afterfi
144 {\expandafter\XINT_powseriesx_pre\expandafter
145 {\romannumeral`&&@#4}{#1}{#2}{#3}%
146 }%
147 \fi
148 }%
149 \def\XINT_powseriesx_pre #1#2#3#4%
150 {%
151 \XINT_powseries_loop_i {#4}{#3}{#2}{#3}{#4}{#1}%
152 }%

```

## 25.7. \xintRationalSeries

This computes  $F(a)+\dots+F(b)$  on the basis of the value of  $F(a)$  and the ratios  $F(n)/F(n-1)$ . As in [\xintPowerSeries](#) we use an iterative scheme which has the great advantage to avoid denominator build-up. This makes exact computations possible with exponential type series, which would be completely inaccessible to [\xintSeries](#). #1= $a$ , #2= $b$ , #3= $F(a)$ , #4=ratio function Modified in 1.06 to give the indices first to a [\numexpr](#) rather than expanding twice. I just use [\the\numexpr](#) and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```

153 \def\xintRationalSeries {\romannumeral0\xintratseries }%
154 \def\xintratseries #1#2%
155 {%
156 \expandafter\XINT_ratseries\expandafter
157 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
158 }%
159 \def\XINT_ratseries #1#2#3#4%
160 {%

```

```

161 \ifnum #2<#1
162 \xint_afterfi { 0/1[0]}%
163 \else
164 \xint_afterfi
165 {\XINT_ratseries_loop {#2}{1}{#1}{#4}{#3}}%
166 \fi
167 }%
168 \def\XINT_ratseries_loop #1#2#3#4%
169 {%
170 \ifnum #1>#3 \else\XINT_ratseries_exit_i\fi
171 \expandafter\XINT_ratseries_loop\expandafter
172 {\the\numexpr #1-1\expandafter}\expandafter
173 {\romannumeral0\xintadd {1}{\xintMul {#2}{#4{#1}}}{#3}{#4}}%
174 }%
175 \def\XINT_ratseries_exit_i\fi #1#2#3#4#5#6#7#8%
176 {%
177 \fi \XINT_ratseries_exit_ii #6%
178 }%
179 \def\XINT_ratseries_exit_ii #1#2#3#4#5%
180 {%
181 \XINT_ratseries_exit_iii #5%
182 }%
183 \def\XINT_ratseries_exit_iii #1#2#3#4%
184 {%
185 \xintmul{#2}{#4}%
186 }%

```

## 25.8. \xintRationalSeriesX

*a*,*b*,*initial*,*ratiofunction*,*x*

This computes  $F(a,x)+\dots+F(b,x)$  on the basis of the value of  $F(a,x)$  and the ratios  $F(n,x)/F(n-1,x)$ . The argument *x* is first expanded and it is the value resulting from this which is used then throughout. The initial term  $F(a,x)$  must be defined as one-parameter macro which will be given *x*. Modified in 1.06 to give the indices first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```

187 \def\xintRationalSeriesX {\romannumeral0\xintratseriesx }%
188 \def\xintratseriesx #1#2%
189 {%
190 \expandafter\XINT_ratseriesx\expandafter
191 {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
192 }%
193 \def\XINT_ratseriesx #1#2#3#4#5%
194 {%
195 \ifnum #2<#1
196 \xint_afterfi { 0/1[0]}%
197 \else
198 \xint_afterfi
199 {\expandafter\XINT_ratseriesx_pre\expandafter
200 {\romannumeral`&&@#5}{#2}{#1}{#4}{#3}}%
201 }%
202 \fi

```

```

203 }%
204 \def\XINT_ratseriesx_pre #1#2#3#4#5%
205 {%
206     \XINT_ratseries_loop {#2}{1}{#3}{#4{#1}}{#5{#1}}%
207 }%

```

## 25.9. \xintFxpPowerSeries

I am not too happy with this piece of code. Will make it more economical another day. Modified in 1.06 to give the indices first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that. 1.08a: forgot last time some optimization from the change to `\numexpr`.

```

208 \def\xintFxpPowerSeries {\romannumeral0\xintfxptpowerseries }%
209 \def\xintfxptpowerseries #1#2%
210 {%
211     \expandafter\XINT_fppowseries\expandafter
212     {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
213 }%
214 \def\XINT_fppowseries #1#2#3#4#5%
215 {%
216     \ifnum #2<#1
217         \xint_afterfi { 0}%
218     \else
219         \xint_afterfi
220         {\expandafter\XINT_fppowseries_loop_pre\expandafter
221          {\romannumeral0\xinttrunc {#5}{\xintPow {#4}{#1}}}%
222          {#1}{#4}{#2}{#3}{#5}%
223         }%
224     \fi
225 }%
226 \def\XINT_fppowseries_loop_pre #1#2#3#4#5#6%
227 {%
228     \ifnum #4>#2 \else\XINT_fppowseries_dont_i \fi
229     \expandafter\XINT_fppowseries_loop_i\expandafter
230     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
231     {\romannumeral0\xintitrunc {#6}{\xintMul {#5{#2}}{#1}}}%
232     {#1}{#3}{#4}{#5}{#6}%
233 }%
234 \def\XINT_fppowseries_dont_i \fi\expandafter\XINT_fppowseries_loop_i
235     {\fi \expandafter\XINT_fppowseries_dont_ii }%
236 \def\XINT_fppowseries_dont_ii #1#2#3#4#5#6#7{\xinttrunc {#7}{#2[-#7]}}%
237 \def\XINT_fppowseries_loop_i #1#2#3#4#5#6#7%
238 {%
239     \ifnum #5>#1 \else \XINT_fppowseries_exit_i \fi
240     \expandafter\XINT_fppowseries_loop_ii\expandafter
241     {\romannumeral0\xinttrunc {#7}{\xintMul {#3}{#4}}}%
242     {#1}{#4}{#2}{#5}{#6}{#7}%
243 }%
244 \def\XINT_fppowseries_loop_ii #1#2#3#4#5#6#7%
245 {%
246     \expandafter\XINT_fppowseries_loop_i\expandafter
247     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter

```

```

248     {\romannumeral0\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}%
249     {#1}{#3}{#5}{#6}{#7}%
250 }%
251 \def\xINT_fppowseries_exit_i\fi\expandafter\xINT_fppowseries_loop_ii
252     {\fi \expandafter\xINT_fppowseries_exit_ii }%
253 \def\xINT_fppowseries_exit_ii #1#2#3#4#5#6#7%
254 {%
255     \xinttrunc {#7}
256     {\xintiiadd {#4}{\xintiTrunc {#7}{\xintMul {#6{#2}}{#1}}}{-#7}}%
257 }%

```

## 25.10. \xintFxpPtPowerSeriesX

a,b,coeff,x,D

Modified in 1.06 to give the indices first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that. 1.08a adds the forgotten optimization following that previous change.

```

258 \def\xintFxpPtPowerSeriesX {\romannumeral0\xintfxptpowerseriesx }%
259 \def\xintfxptpowerseriesx #1#2%
260 {%
261     \expandafter\xINT_fppowseriesx\expandafter
262     {\the\numexpr #1\expandafter}\expandafter{\the\numexpr #2}%
263 }%
264 \def\xINT_fppowseriesx #1#2#3#4#5%
265 {%
266     \ifnum #2<#1
267         \xint_afterfi { 0}%
268     \else
269         \xint_afterfi
270         {\expandafter \xINT_fppowseriesx_pre \expandafter
271         {\romannumeral`&&@#4}{#1}{#2}{#3}{#5}%
272         }%
273     \fi
274 }%
275 \def\xINT_fppowseriesx_pre #1#2#3#4#5%
276 {%
277     \expandafter\xINT_fppowseries_loop_pre\expandafter
278     {\romannumeral0\xinttrunc {#5}{\xintPow {#1}{#2}}}%
279     {#2}{#1}{#3}{#4}{#5}%
280 }%

```

## 25.11. \xintFloatPowerSeries

1.08a. I still have to re-visit `\xintFxpPtPowerSeries`; temporarily I just adapted the code to the case of floats.

Usage of new names `\XINTinfloatpow_wopt` `\XINTinfloatmul_wopt`, `\XINTinfloatadd_wopt` to track `xintfrac.sty` changes at 1.4e.

```

281 \def\xintFloatPowerSeries {\romannumeral0\xintfloatpowerseries }%
282 \def\xintfloatpowerseries #1{\XINT_flpowseries_chkopt #1\xint:}%
283 \def\xINT_flpowseries_chkopt #1%
284 {%
285     \ifx [#1\expandafter\xINT_flpowseries_opt

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

286     \else\expandafter\XINT_flpowseries_noopt
287     \fi
288     #1%
289 }%
290 \def\XINT_flpowseries_noopt #1\xint:#2%
291 {%
292     \expandafter\XINT_flpowseries\expandafter
293     {\the\numexpr #1\expandafter}\expandafter
294     {\the\numexpr #2}\XINTdigits
295 }%
296 \def\XINT_flpowseries_opt [\xint:#1]#2#3%
297 {%
298     \expandafter\XINT_flpowseries\expandafter
299     {\the\numexpr #2\expandafter}\expandafter
300     {\the\numexpr #3\expandafter}{\the\numexpr #1}%
301 }%
302 \def\XINT_flpowseries #1#2#3#4#5%
303 {%
304     \ifnum #2<#1
305         \xint_afterfi { 0.e0}%
306     \else
307         \xint_afterfi
308         {\expandafter\XINT_flpowseries_loop_pre\expandafter
309          {\romannumeral0\XINTinfloatpow_wopt[#3]{#5}{#1}}%
310          {#1}{#5}{#2}{#4}{#3}%
311         }%
312     \fi
313 }%
314 \def\XINT_flpowseries_loop_pre #1#2#3#4#5#6%
315 {%
316     \ifnum #4>#2 \else\XINT_flpowseries_dont_i \fi
317     \expandafter\XINT_flpowseries_loop_i\expandafter
318     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
319     {\romannumeral0\XINTinfloatmul_wopt[#6]{#5}{#2}}{#1}%
320     {#1}{#3}{#4}{#5}{#6}%
321 }%
322 \def\XINT_flpowseries_dont_i \fi\expandafter\XINT_flpowseries_loop_i
323 {\fi \expandafter\XINT_flpowseries_dont_ii }%
324 \def\XINT_flpowseries_dont_ii #1#2#3#4#5#6#7{\xintfloat [#7]{#2}}%
325 \def\XINT_flpowseries_loop_i #1#2#3#4#5#6#7%
326 {%
327     \ifnum #5>#1 \else \XINT_flpowseries_exit_i \fi
328     \expandafter\XINT_flpowseries_loop_ii\expandafter
329     {\romannumeral0\XINTinfloatmul_wopt[#7]{#3}{#4}}%
330     {#1}{#4}{#2}{#5}{#6}{#7}%
331 }%
332 \def\XINT_flpowseries_loop_ii #1#2#3#4#5#6#7%
333 {%
334     \expandafter\XINT_flpowseries_loop_i\expandafter
335     {\the\numexpr #2+\xint_c_i\expandafter}\expandafter
336     {\romannumeral0\XINTinfloatadd_wopt[#7]{#4}%
337      {\XINTinfloatmul_wopt[#7]{#6}{#2}}{#1}}}%

```

```

338     {#1}{#3}{#5}{#6}{#7}%
339 }%
340 \def\XINT_flpowseries_exit_i\fi\expandafter\XINT_flpowseries_loop_ii
341     {\fi \expandafter\XINT_flpowseries_exit_ii }%
342 \def\XINT_flpowseries_exit_ii #1#2#3#4#5#6#7%
343 {%
344     \xintfloatadd[#7]{#4}{\XINTinfloatmul_wopt[#7]{#6}{#2}}{#1}}%
345 }%

```

## 25.12. \xintFloatPowerSeriesX

1.08a

See `\xintFloatPowerSeries` for 1.4e comments.

```

346 \def\xintFloatPowerSeriesX {\romannumeral0\xintfloatpowerseriesx }%
347 \def\xintfloatpowerseriesx #1{\XINT_flpowseriesx_chkopt #1\xint:}%
348 \def\XINT_flpowseriesx_chkopt #1%
349 {%
350     \ifx [#1\expandafter\XINT_flpowseriesx_opt
351         \else\expandafter\XINT_flpowseriesx_noopt
352     \fi
353     #1%
354 }%
355 \def\XINT_flpowseriesx_noopt #1\xint:#2%
356 {%
357     \expandafter\XINT_flpowseriesx\expandafter
358     {\the\numexpr #1\expandafter}\expandafter
359     {\the\numexpr #2}\XINTdigits
360 }%
361 \def\XINT_flpowseriesx_opt [\xint:#1]#2#3%
362 {%
363     \expandafter\XINT_flpowseriesx\expandafter
364     {\the\numexpr #2\expandafter}\expandafter
365     {\the\numexpr #3\expandafter}{\the\numexpr #1}%
366 }%
367 \def\XINT_flpowseriesx #1#2#3#4#5%
368 {%
369     \ifnum #2<#1
370         \xint_afterfi { 0.e0}%
371     \else
372         \xint_afterfi
373         {\expandafter \XINT_flpowseriesx_pre \expandafter
374         {\romannumeral`&&@#5}{#1}{#2}{#4}{#3}%
375         }%
376     \fi
377 }%
378 \def\XINT_flpowseriesx_pre #1#2#3#4#5%
379 {%
380     \expandafter\XINT_flpowseries_loop_pre\expandafter
381     {\romannumeral0\XINTinfloatpow_wopt[#5]{#1}{#2}}%
382     {#2}{#1}{#3}{#4}{#5}%
383 }%
384 \XINTrestorecatcodesendinginput%

```

## 26. Package *xintcfrac* implementation

.1	Catcodes, $\varepsilon$ -TeX and reload detection . . .	534	.16	<code>\xintiGctoF</code> . . . . .	546
.2	Package identification . . . . .	535	.17	<code>\xintCtoCv</code> , <code>\xintCstoCv</code> . . . . .	547
.3	<code>\xintCFrac</code> . . . . .	535	.18	<code>\xintiCstoCv</code> . . . . .	548
.4	<code>\xintGCFrac</code> . . . . .	537	.19	<code>\xintGctoCv</code> . . . . .	549
.5	<code>\xintGGCFrac</code> . . . . .	538	.20	<code>\xintiGctoCv</code> . . . . .	550
.6	<code>\xintGctoGCx</code> . . . . .	539	.21	<code>\xintFtoCv</code> . . . . .	551
.7	<code>\xintFtoCs</code> . . . . .	539	.22	<code>\xintFtoCCv</code> . . . . .	551
.8	<code>\xintFtoCx</code> . . . . .	540	.23	<code>\xintCntoF</code> . . . . .	552
.9	<code>\xintFtoC</code> . . . . .	541	.24	<code>\xintGcntoF</code> . . . . .	552
.10	<code>\xintFtoGC</code> . . . . .	541	.25	<code>\xintCntoCs</code> . . . . .	553
.11	<code>\xintFGtoC</code> . . . . .	541	.26	<code>\xintCntoGC</code> . . . . .	554
.12	<code>\xintFtoCC</code> . . . . .	542	.27	<code>\xintGcntoGC</code> . . . . .	554
.13	<code>\xintCtoF</code> , <code>\xintCstoF</code> . . . . .	543	.28	<code>\xintCstoGC</code> . . . . .	555
.14	<code>\xintiCstoF</code> . . . . .	544	.29	<code>\xintGctoGC</code> . . . . .	555
.15	<code>\xintGctoF</code> . . . . .	545			

The commenting is currently (2025/09/06) very sparse. Release 1.09m (2014/02/26) has modified a few things: `\xintFtoCs` and `\xintCntoCs` insert spaces after the commas, `\xintCstoF` and `\xintCstoCv` authorize spaces in the input also before the commas, `\xintCntoCs` does not brace the produced coefficients, new macros `\xintFtoC`, `\xintCtoF`, `\xintCtoCv`, `\xintFGtoC`, and `\xintGGCFrac`.

The macros `\xintCstoF` and `\xintCstoCv` use *xinttools*'s `\xintCSVtoList`. Formerly, it was up to user to load *xinttools* to enable these macros. Starting at 1.4n the loading is automatic.

### 26.1. Catcodes, $\varepsilon$ -TeX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2   \catcode13=5      % ^^M
3   \endlinechar=13 %
4   \catcode123=1     % {
5   \catcode125=2     % }
6   \catcode64=11     % @
7   \catcode44=12     % ,
8   \catcode46=12     % .
9   \catcode58=12     % :
10  \catcode94=7      % ^
11  \def\empty{}\def\space{ }\newlinechar10
12  \def\z{\endgroup}%
13  \expandafter\let\expandafter\x\csname ver@xintcfrac.sty\endcsname
14  \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
15  \expandafter\let\expandafter\t\csname ver@xinttools.sty\endcsname
16  \expandafter\ifx\csname numexpr\endcsname\relax
17    \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
18      \immediate\write128{^^JPackage xintcfrac Warning:^^J}%
19      \space\space\space\space
20      \numexpr not available, aborting input.^^J}%
21  \else
22    \PackageWarningNoLine{xintcfrac}{\numexpr not available, aborting input}%
23  \fi
24  \def\z{\endgroup\endinput}%

```

```

25 \else
26   \ifx\x\relax % not LaTeX, first loading of xintcfrac.sty
27   \ifx\w\relax % but xintfrac.sty not yet loaded.
28     \expandafter\def\expandafter\z\expandafter
29       {\z\input xintfrac.sty\relax}%
30   \fi
31   \ifx\t\relax % but xinttools.sty not yet loaded.
32     \expandafter\def\expandafter\z\expandafter
33       {\z\input xinttools.sty\relax}%
34   \fi
35 \else
36   \ifx\x\empty % LaTeX, first loading,
37   % variable is initialized, but \ProvidesPackage not yet seen
38   \ifx\w\relax % xintfrac not yet loaded.
39     \expandafter\def\expandafter\z\expandafter
40       {\z\RequirePackage{xintfrac}}%
41   \fi
42   \ifx\t\relax % xinttools not yet loaded.
43     \expandafter\def\expandafter\z\expandafter
44       {\z\RequirePackage{xinttools}}%
45   \fi
46 \else
47   \def\z{\endgroup\endinput}% xintcfrac already loaded.
48 \fi
49 \fi
50 \fi
51 \z%
52 \XINTsetupcatcodes% defined in xintkernel.sty

```

## 26.2. Package identification

```

53 \XINTprovidespackage
54 \ProvidesPackage{xintcfrac}%
55 [2025/09/06 v1.4o Expandable continued fractions with xint package (JFB)]%

```

## 26.3. \xintCFrac

```

56 \def\xintCFrac {\romannumeral0\xintcfrac }%
57 \def\xintcfrac #1%
58 {%
59   \XINT_cfrac_opt_a #1\xint:
60 }%
61 \def\XINT_cfrac_opt_a #1%
62 {%
63   \ifx[#1\XINT_cfrac_opt_b\fi \XINT_cfrac_noopt #1%
64 }%
65 \def\XINT_cfrac_noopt #1\xint:
66 {%
67   \expandafter\XINT_cfrac_A\romannumeral0\xintrawwithzeros {#1}\Z
68   \relax\relax
69 }%
70 \def\XINT_cfrac_opt_b\fi\XINT_cfrac_noopt [\xint:#1]%
71 {%

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```

72   \fi\csname XINT_cfrac_opt#1\endcsname
73 }%
74 \def\XINT_cfrac_optl #1%
75 {%
76   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
77   \relax\hfill
78 }%
79 \def\XINT_cfrac_optc #1%
80 {%
81   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
82   \relax\relax
83 }%
84 \def\XINT_cfrac_optr #1%
85 {%
86   \expandafter\XINT_cfrac_A\romannumeral0\xintraewithzeros {#1}\Z
87   \hfill\relax
88 }%
89 \def\XINT_cfrac_A #1/#2\Z
90 {%
91   \expandafter\XINT_cfrac_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
92 }%
93 \def\XINT_cfrac_B #1#2%
94 {%
95   \XINT_cfrac_C #2\Z {#1}%
96 }%
97 \def\XINT_cfrac_C #1%
98 {%
99   \xint_gob_til_zero #1\XINT_cfrac_integer 0\XINT_cfrac_D #1%
100 }%
101 \def\XINT_cfrac_integer 0\XINT_cfrac_D 0#1\Z #2#3#4#5{ #2}%
102 \def\XINT_cfrac_D #1\Z #2#3{\XINT_cfrac_loop_a {#1}{#3}{#1}{#2}}%
103 \def\XINT_cfrac_loop_a
104 {%
105   \expandafter\XINT_cfrac_loop_d\romannumeral0\XINT_div_prepare
106 }%
107 \def\XINT_cfrac_loop_d #1#2%
108 {%
109   \XINT_cfrac_loop_e #2.{#1}%
110 }%
111 \def\XINT_cfrac_loop_e #1%
112 {%
113   \xint_gob_til_zero #1\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1%
114 }%
115 \def\XINT_cfrac_loop_f #1.#2#3#4%
116 {%
117   \XINT_cfrac_loop_a {#1}{#3}{#1}{#2#4}%
118 }%
119 \def\xint_cfrac_loop_exit0\XINT_cfrac_loop_f #1.#2#3#4#5#6%
120   {\XINT_cfrac_T #5#6{#2}#4\Z }%
121 \def\XINT_cfrac_T #1#2#3#4%
122 {%
123   \xint_gob_til_Z #4\XINT_cfrac_end\Z\XINT_cfrac_T #1#2{#4+\cfrac{#1#2}{#3}}%

```

```

124 }%
125 \def\XINT_cfrac_end\Z\XINT_cfrac_T #1#2#3%
126 {%
127     \XINT_cfrac_end_b #3%
128 }%
129 \def\XINT_cfrac_end_b \Z+\cfrac#1#2{ #2}%

```

## 26.4. \xintGCFrac

Updated at 1.4g to follow-up on renaming of `\xintFrac` into `\xintTeXFrac`.

```

130 \def\xintGCFrac {\romannumeral0\xintgcfrac }%
131 \def\xintgcfrac #1{\XINT_gcfrac_opt_a #1\xint:}%
132 \def\XINT_gcfrac_opt_a #1%
133 {%
134     \ifx[#1\XINT_gcfrac_opt_b\fi \XINT_gcfrac_noopt #1%
135 }%
136 \def\XINT_gcfrac_noopt #1\xint:%
137 {%
138     \XINT_gcfrac #1+!\relax\relax
139 }%
140 \def\XINT_gcfrac_opt_b\fi\XINT_gcfrac_noopt [\xint:#1]%
141 {%
142     \fi\csname XINT_gcfrac_opt#1\endcsname
143 }%
144 \def\XINT_gcfrac_optl #1%
145 {%
146     \XINT_gcfrac #1+!\relax\hfill
147 }%
148 \def\XINT_gcfrac_optc #1%
149 {%
150     \XINT_gcfrac #1+!\relax\relax
151 }%
152 \def\XINT_gcfrac_optr #1%
153 {%
154     \XINT_gcfrac #1+!\hfill\relax
155 }%
156 \def\XINT_gcfrac
157 {%
158     \expandafter\XINT_gcfrac_enter\romannumeral`&&@%
159 }%
160 \def\XINT_gcfrac_enter {\XINT_gcfrac_loop {}}%
161 \def\XINT_gcfrac_loop #1#2+#3/%
162 {%
163     \xint_gob_til_exclam #3\XINT_gcfrac_endloop!%
164     \XINT_gcfrac_loop {{#3}{#2}#1}%
165 }%
166 \def\XINT_gcfrac_endloop!\XINT_gcfrac_loop #1#2#3%
167 {%
168     \XINT_gcfrac_T #2#3#1!!%
169 }%
170 \def\XINT_gcfrac_T #1#2#3#4{\XINT_gcfrac_U #1#2{\xintTeXFrac{#4}}}%
171 \def\XINT_gcfrac_U #1#2#3#4#5%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

172 {%
173     \xint_gob_til_exclam #5\XINT_gcfrac_end!\XINT_gcfrac_U
174         #1#2{\xintTeXFrac{#5}%
175             \ifcase\xintSgn{#4}
176             +\or+\else-\fi
177             \cfrac{#1\xintTeXFrac{\xintAbs{#4}}{#2}{#3}}%
178 }%
179 \def\XINT_gcfrac_end!\XINT_gcfrac_U #1#2#3%
180 {%
181     \XINT_gcfrac_end_b #3%
182 }%
183 \def\XINT_gcfrac_end_b #1\cfrac#2#3{ #3}%

```

## 26.5. \xintGGCFrac

New with 1.09m

```

184 \def\xintGGCFrac {\romannumeral0\xintggcfrac }%
185 \def\xintggcfrac #1{\XINT_ggcfrac_opt_a #1\xint:}%
186 \def\XINT_ggcfrac_opt_a #1%
187 {%
188     \ifx[#1\XINT_ggcfrac_opt_b\fi \XINT_ggcfrac_noopt #1%
189 }%
190 \def\XINT_ggcfrac_noopt #1\xint:
191 {%
192     \XINT_ggcfrac #1+!\relax\relax
193 }%
194 \def\XINT_ggcfrac_opt_b\fi\XINT_ggcfrac_noopt [\xint:#1]%
195 {%
196     \fi\csname XINT_ggcfrac_opt#1\endcsname
197 }%
198 \def\XINT_ggcfrac_optl #1%
199 {%
200     \XINT_ggcfrac #1+!\relax\hfill
201 }%
202 \def\XINT_ggcfrac_optc #1%
203 {%
204     \XINT_ggcfrac #1+!\relax\relax
205 }%
206 \def\XINT_ggcfrac_optr #1%
207 {%
208     \XINT_ggcfrac #1+!\hfill\relax
209 }%
210 \def\XINT_ggcfrac
211 {%
212     \expandafter\XINT_ggcfrac_enter\romannumeral`&&@%
213 }%
214 \def\XINT_ggcfrac_enter {\XINT_ggcfrac_loop {}}%
215 \def\XINT_ggcfrac_loop #1#2+#3/%
216 {%
217     \xint_gob_til_exclam #3\XINT_ggcfrac_endloop!%
218     \XINT_ggcfrac_loop {{#3}{#2}{#1}}%
219 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
220 \def\XINT_ggcfrac_endloop!\XINT_ggcfrac_loop #1#2#3%
221 {%
222     \XINT_ggcfrac_T #2#3#1!!%
223 }%
224 \def\XINT_ggcfrac_T #1#2#3#4{\XINT_ggcfrac_U #1#2{#4}}%
225 \def\XINT_ggcfrac_U #1#2#3#4#5%
226 {%
227     \xint_gob_til_exclam #5\XINT_ggcfrac_end!\XINT_ggcfrac_U
228         #1#2{#5+\cfrac{#1#4#2}{#3}}%
229 }%
230 \def\XINT_ggcfrac_end!\XINT_ggcfrac_U #1#2#3%
231 {%
232     \XINT_ggcfrac_end_b #3%
233 }%
234 \def\XINT_ggcfrac_end_b #1\cfrac#2#3{ #3}%
```

### 26.6. \xintGctoGCx

```
235 \def\xintGctoGCx {\romannumeral0\xintgctogcx }%
236 \def\xintgctogcx #1#2#3%
237 {%
238     \expandafter\XINT_gctgcx_start\expandafter {\romannumeral`&@#3}{#1}{#2}%
239 }%
240 \def\XINT_gctgcx_start #1#2#3{\XINT_gctgcx_loop_a {}{#2}{#3}#1+!/}%
241 \def\XINT_gctgcx_loop_a #1#2#3#4+#5/%
242 {%
243     \xint_gob_til_exclam #5\XINT_gctgcx_end!%
244     \XINT_gctgcx_loop_b {#1{#4}}{#2{#5}#3}{#2}{#3}%
245 }%
246 \def\XINT_gctgcx_loop_b #1#2%
247 {%
248     \XINT_gctgcx_loop_a {#1#2}%
249 }%
250 \def\XINT_gctgcx_end!\XINT_gctgcx_loop_b #1#2#3#4{ #1}%
```

### 26.7. \xintFtoCs

Modified in 1.09m: a space is added after the inserted commas.

```
251 \def\xintFtoCs {\romannumeral0\xintftocs }%
252 \def\xintftocs #1%
253 {%
254     \expandafter\XINT_ftc_A\romannumeral0\xintrawwithzeros {#1}\Z
255 }%
256 \def\XINT_ftc_A #1/#2\Z
257 {%
258     \expandafter\XINT_ftc_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
259 }%
260 \def\XINT_ftc_B #1#2%
261 {%
262     \XINT_ftc_C #2.{#1}%
263 }%
264 \def\XINT_ftc_C #1%
265 {%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
266 \xint_gob_til_zero #1\xINT_ftc_integer 0\xINT_ftc_D #1%
267 }%
268 \def\xINT_ftc_integer 0\xINT_ftc_D 0#1.#2#3{ #2}%
269 \def\xINT_ftc_D #1.#2#3{\xINT_ftc_loop_a {#1}{#3}{#1}{#2, }}% 1.09m adds a space
270 \def\xINT_ftc_loop_a
271 {%
272 \expandafter\xINT_ftc_loop_d\romannumeral0\xINT_div_prepare
273 }%
274 \def\xINT_ftc_loop_d #1#2%
275 {%
276 \xINT_ftc_loop_e #2.{#1}%
277 }%
278 \def\xINT_ftc_loop_e #1%
279 {%
280 \xint_gob_til_zero #1\xint_ftc_loop_exit0\xINT_ftc_loop_f #1%
281 }%
282 \def\xINT_ftc_loop_f #1.#2#3#4%
283 {%
284 \xINT_ftc_loop_a {#1}{#3}{#1}{#4#2, }}% 1.09m has an added space here
285 }%
286 \def\xint_ftc_loop_exit0\xINT_ftc_loop_f #1.#2#3#4{ #4#2}%
```

## 26.8. \xintFtoCx

```
287 \def\xintFtoCx {\romannumeral0\xintftocx }%
288 \def\xintftocx #1#2%
289 {%
290 \expandafter\xINT_ftcx_A\romannumeral0\xintrawwithzeros {#2}\Z {#1}%
291 }%
292 \def\xINT_ftcx_A #1/#2\Z
293 {%
294 \expandafter\xINT_ftcx_B\romannumeral0\xintiidivision {#1}{#2}{#2}%
295 }%
296 \def\xINT_ftcx_B #1#2%
297 {%
298 \xINT_ftcx_C #2.{#1}%
299 }%
300 \def\xINT_ftcx_C #1%
301 {%
302 \xint_gob_til_zero #1\xINT_ftcx_integer 0\xINT_ftcx_D #1%
303 }%
304 \def\xINT_ftcx_integer 0\xINT_ftcx_D 0#1.#2#3#4{ #2}%
305 \def\xINT_ftcx_D #1.#2#3#4{\xINT_ftcx_loop_a {#1}{#3}{#1}{#2}{#4}{#4}}%
306 \def\xINT_ftcx_loop_a
307 {%
308 \expandafter\xINT_ftcx_loop_d\romannumeral0\xINT_div_prepare
309 }%
310 \def\xINT_ftcx_loop_d #1#2%
311 {%
312 \xINT_ftcx_loop_e #2.{#1}%
313 }%
314 \def\xINT_ftcx_loop_e #1%
315 {%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
316 \xint_gob_til_zero #1\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1%
317 }%
318 \def\XINT_ftcx_loop_f #1.#2#3#4#5%
319 {%
320 \XINT_ftcx_loop_a {#1}{#3}{#1}{#4{#2}#5}{#5}%
321 }%
322 \def\xint_ftcx_loop_exit0\XINT_ftcx_loop_f #1.#2#3#4#5{ #4{#2}}%
```

### 26.9. \xintFtoC

New in 1.09m: this is the same as `\xintFtoCx` with empty separator. I had temporarily during preparation of 1.09m removed braces from `\xintFtoCx`, but I recalled later why that was useful (see doc), thus let's just here do `\xintFtoCx {}`

```
323 \def\xintFtoC {\romannumeral0\xintftoc }%
324 \def\xintftoc {\xintftocx {}}%
```

### 26.10. \xintFtoGC

```
325 \def\xintFtoGC {\romannumeral0\xintftogc }%
326 \def\xintftogc {\xintftocx {+1/}}%
```

### 26.11. \xintFGtoC

New with 1.09m of 2014/02/26. Computes the common initial coefficients for the two fractions *f* and *g*, and outputs them as a sequence of braced items.

```
327 \def\xintFGtoC {\romannumeral0\xintfgtoc}%
328 \def\xintfgtoc#1%
329 {%
330 \expandafter\XINT_fgtc_a\romannumeral0\xintraewithzeros {#1}\Z
331 }%
332 \def\XINT_fgtc_a #1/#2\Z #3%
333 {%
334 \expandafter\XINT_fgtc_b\romannumeral0\xintraewithzeros {#3}\Z #1/#2\Z { }%
335 }%
336 \def\XINT_fgtc_b #1/#2\Z
337 {%
338 \expandafter\XINT_fgtc_c\romannumeral0\xintiidivision {#1}{#2}{#2}%
339 }%
340 \def\XINT_fgtc_c #1#2#3#4/#5\Z
341 {%
342 \expandafter\XINT_fgtc_d\romannumeral0\xintiidivision
343 {#4}{#5}{#5}{#1}{#2}{#3}%
344 }%
345 \def\XINT_fgtc_d #1#2#3#4%#5#6#7%
346 {%
347 \xintifEq {#1}{#4}{\XINT_fgtc_da {#1}{#2}{#3}{#4}}%
348 {\xint_thirdofthree}%
349 }%
350 \def\XINT_fgtc_da #1#2#3#4#5#6#7%
351 {%
352 \XINT_fgtc_e {#2}{#5}{#3}{#6}{#7{#1}}%
353 }%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
354 \def\XINT_fgtc_e #1%
355 {%
356     \xintiiifZero {#1}\expandafter\xint_firstofone\xint_gobble_iii}%
357     {\XINT_fgtc_f {#1}}%
358 }%
359 \def\XINT_fgtc_f #1#2%
360 {%
361     \xintiiifZero {#2}\xint_thirdofthree}{\XINT_fgtc_g {#1}{#2}}%
362 }%
363 \def\XINT_fgtc_g #1#2#3%
364 {%
365     \expandafter\XINT_fgtc_h\romannumeral0\XINT_div_prepare {#1}{#3}{#1}{#2}%
366 }%
367 \def\XINT_fgtc_h #1#2#3#4#5%
368 {%
369     \expandafter\XINT_fgtc_d\romannumeral0\XINT_div_prepare
370     {#4}{#5}{#4}{#1}{#2}{#3}%
371 }%
```

## 26.12. \xintFtoCC

```
372 \def\xintFtoCC {\romannumeral0\xintftocc }%
373 \def\xintftocc #1%
374 {%
375     \expandafter\XINT_ftcc_A\expandafter {\romannumeral0\xintrawwithzeros {#1}}%
376 }%
377 \def\XINT_ftcc_A #1%
378 {%
379     \expandafter\XINT_ftcc_B
380     \romannumeral0\xintrawwithzeros {\xintAdd {1/2[0]}{#1[0]}}\Z {#1[0]}%
381 }%
382 \def\XINT_ftcc_B #1/#2\Z
383 {%
384     \expandafter\XINT_ftcc_C\expandafter {\romannumeral0\xintiiquo {#1}{#2}}%
385 }%
386 \def\XINT_ftcc_C #1#2%
387 {%
388     \expandafter\XINT_ftcc_D\romannumeral0\xintsub {#2}{#1}\Z {#1}%
389 }%
390 \def\XINT_ftcc_D #1%
391 {%
392     \xint_UDzerominusfork
393     #1-\XINT_ftcc_integer
394     0#1\XINT_ftcc_En
395     0-{\XINT_ftcc_Ep #1}%
396     \krof
397 }%
398 \def\XINT_ftcc_Ep #1\Z #2%
399 {%
400     \expandafter\XINT_ftcc_loop_a\expandafter
401     {\romannumeral0\xintdiv {1[0]}{#1}}{#2+1}%
402 }%
403 \def\XINT_ftcc_En #1\Z #2%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```

404 {%
405     \expandafter\XINT_ftcc_loop_a\expandafter
406     {\romannumeral0\xintdiv {1[0]}\{#1}\{#2+-1/}%
407 }%
408 \def\XINT_ftcc_integer #1\Z #2{ #2}%
409 \def\XINT_ftcc_loop_a #1%
410 {%
411     \expandafter\XINT_ftcc_loop_b
412     \romannumeral0\xintraewithzeros {\xintAdd {1/2[0]}\{#1}\Z {#1}%
413 }%
414 \def\XINT_ftcc_loop_b #1/#2\Z
415 {%
416     \expandafter\XINT_ftcc_loop_c\expandafter
417     {\romannumeral0\xintiigo {#1}\{#2}%
418 }%
419 \def\XINT_ftcc_loop_c #1#2%
420 {%
421     \expandafter\XINT_ftcc_loop_d
422     \romannumeral0\xintsub {#2}\{#1[0]\Z {#1}%
423 }%
424 \def\XINT_ftcc_loop_d #1%
425 {%
426     \xint_UDzerominusfork
427     #1-\XINT_ftcc_end
428     0#1\XINT_ftcc_loop_N
429     0-\XINT_ftcc_loop_P #1}%
430 \krof
431 }%
432 \def\XINT_ftcc_end #1\Z #2#3{ #3#2}%
433 \def\XINT_ftcc_loop_P #1\Z #2#3%
434 {%
435     \expandafter\XINT_ftcc_loop_a\expandafter
436     {\romannumeral0\xintdiv {1[0]}\{#1}\{#3#2+1/}%
437 }%
438 \def\XINT_ftcc_loop_N #1\Z #2#3%
439 {%
440     \expandafter\XINT_ftcc_loop_a\expandafter
441     {\romannumeral0\xintdiv {1[0]}\{#1}\{#3#2+-1/}%
442 }%

```

### 26.13. \xintCtoF, \xintCstoF

1.09m uses *xinttools*'s `\xintCSVtolist` on the argument of `\xintCstoF` to allow spaces also before the commas. And the original `\xintCstoF` code became the one of the new `\xintCtoF` dealing with a braced rather than comma separated list.

```

443 \def\xintCstoF {\romannumeral0\xintcstof }%
444 \def\xintcstof #1%
445 {%
446     \expandafter\XINT_ctf_prep \romannumeral0\xintcsvtolist{#1}!%
447 }%
448 \def\xintCtoF {\romannumeral0\xintctof }%
449 \def\xintctof #1%

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```

450 {%
451   \expandafter\XINT_ctf_prep \romannumeral`&&@#1!%
452 }%
453 \def\XINT_ctf_prep
454 {%
455   \XINT_ctf_loop_a 1001%
456 }%
457 \def\XINT_ctf_loop_a #1#2#3#4#5%
458 {%
459   \xint_gob_til_exclam #5\XINT_ctf_end!%
460   \expandafter\XINT_ctf_loop_b
461   \romannumeral0\xintraewithzeros {#5}.{#1}{#2}{#3}{#4}%
462 }%
463 \def\XINT_ctf_loop_b #1/#2.#3#4#5#6%
464 {%
465   \expandafter\XINT_ctf_loop_c\expandafter
466   {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
467   {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
468   {\romannumeral0\xintiadd {\XINT_mul_fork #2\xint:#6\xint:}%
469                               {\XINT_mul_fork #1\xint:#4\xint:}}%
470   {\romannumeral0\xintiadd {\XINT_mul_fork #2\xint:#5\xint:}%
471                               {\XINT_mul_fork #1\xint:#3\xint:}}%
472 }%
473 \def\XINT_ctf_loop_c #1#2%
474 {%
475   \expandafter\XINT_ctf_loop_d\expandafter {\expandafter{#2}{#1}}%
476 }%
477 \def\XINT_ctf_loop_d #1#2%
478 {%
479   \expandafter\XINT_ctf_loop_e\expandafter {\expandafter{#2}#1}%
480 }%
481 \def\XINT_ctf_loop_e #1#2%
482 {%
483   \expandafter\XINT_ctf_loop_a\expandafter{#2}#1%
484 }%
485 \def\XINT_ctf_end #1.#2#3#4#5{\xintraewithzeros {#2/#3}}% 1.09b removes [0]

```

## 26.14. \xintiCstoF

```

486 \def\xintiCstoF {\romannumeral0\xinticstof }%
487 \def\xinticstof #1%
488 {%
489   \expandafter\XINT_icstf_prep \romannumeral`&&@#1,!,%
490 }%
491 \def\XINT_icstf_prep
492 {%
493   \XINT_icstf_loop_a 1001%
494 }%
495 \def\XINT_icstf_loop_a #1#2#3#4#5,%
496 {%
497   \xint_gob_til_exclam #5\XINT_icstf_end!%
498   \expandafter
499   \XINT_icstf_loop_b \romannumeral`&&@#5.{#1}{#2}{#3}{#4}%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
500 }%
501 \def\XINT_icstf_loop_b #1.#2#3#4#5%
502 {%
503     \expandafter\XINT_icstf_loop_c\expandafter
504     {\romannumeral0\xintiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
505     {\romannumeral0\xintiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
506     {#2}{#3}%
507 }%
508 \def\XINT_icstf_loop_c #1#2%
509 {%
510     \expandafter\XINT_icstf_loop_a\expandafter {#2}{#1}%
511 }%
512 \def\XINT_icstf_end#1.#2#3#4#5{\xintrawithzeros {#2/#3}}% 1.09b removes [0]
```

## 26.15. \xintGctoF

```
513 \def\xintGctoF {\romannumeral0\xintgctof }%
514 \def\xintgctof #1%
515 {%
516     \expandafter\XINT_gctf_prep \romannumeral`&&@#1+!/ %
517 }%
518 \def\XINT_gctf_prep
519 {%
520     \XINT_gctf_loop_a 1001%
521 }%
522 \def\XINT_gctf_loop_a #1#2#3#4#5+%
523 {%
524     \expandafter\XINT_gctf_loop_b
525     \romannumeral0\xintrawithzeros {#5}.{#1}{#2}{#3}{#4}%
526 }%
527 \def\XINT_gctf_loop_b #1/#2.#3#4#5#6%
528 {%
529     \expandafter\XINT_gctf_loop_c\expandafter
530     {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
531     {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
532     {\romannumeral0\xintiadd {\XINT_mul_fork #2\xint:#6\xint:}%
533     {\XINT_mul_fork #1\xint:#4\xint:}}%
534     {\romannumeral0\xintiadd {\XINT_mul_fork #2\xint:#5\xint:}%
535     {\XINT_mul_fork #1\xint:#3\xint:}}%
536 }%
537 \def\XINT_gctf_loop_c #1#2%
538 {%
539     \expandafter\XINT_gctf_loop_d\expandafter {\expandafter{#2}{#1}}%
540 }%
541 \def\XINT_gctf_loop_d #1#2%
542 {%
543     \expandafter\XINT_gctf_loop_e\expandafter {\expandafter{#2}{#1}}%
544 }%
545 \def\XINT_gctf_loop_e #1#2%
546 {%
547     \expandafter\XINT_gctf_loop_f\expandafter {\expandafter{#2}{#1}}%
548 }%
549 \def\XINT_gctf_loop_f #1#2/%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
550 {%
551     \xint_gob_til_exclam #2\XINT_gctf_end!%
552     \expandafter\XINT_gctf_loop_g
553     \romannumeral0\xintraewithzeros {#2}.#1%
554 }%
555 \def\XINT_gctf_loop_g #1/#2.#3#4#5#6%
556 {%
557     \expandafter\XINT_gctf_loop_h\expandafter
558     {\romannumeral0\XINT_mul_fork #1\xint:#6\xint:}%
559     {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
560     {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
561     {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
562 }%
563 \def\XINT_gctf_loop_h #1#2%
564 {%
565     \expandafter\XINT_gctf_loop_i\expandafter {\expandafter{#2}{#1}}%
566 }%
567 \def\XINT_gctf_loop_i #1#2%
568 {%
569     \expandafter\XINT_gctf_loop_j\expandafter {\expandafter{#2}{#1}}%
570 }%
571 \def\XINT_gctf_loop_j #1#2%
572 {%
573     \expandafter\XINT_gctf_loop_a\expandafter {#2}{#1}%
574 }%
575 \def\XINT_gctf_end #1.#2#3#4#5{\xintraewithzeros {#2/#3}}% 1.09b removes [0]
```

### 26.16. \xintiGctoF

```
576 \def\xintiGctoF {\romannumeral0\xintigctof }%
577 \def\xintigctof #1%
578 {%
579     \expandafter\XINT_igctf_prep \romannumeral`&&@#1+!/ %
580 }%
581 \def\XINT_igctf_prep
582 {%
583     \XINT_igctf_loop_a 1001%
584 }%
585 \def\XINT_igctf_loop_a #1#2#3#4#5+%
586 {%
587     \expandafter\XINT_igctf_loop_b
588     \romannumeral`&&@#5.{#1}{#2}{#3}{#4}%
589 }%
590 \def\XINT_igctf_loop_b #1.#2#3#4#5%
591 {%
592     \expandafter\XINT_igctf_loop_c\expandafter
593     {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
594     {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
595     {#2}{#3}%
596 }%
597 \def\XINT_igctf_loop_c #1#2%
598 {%
599     \expandafter\XINT_igctf_loop_f\expandafter {\expandafter{#2}{#1}}%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
600 }%
601 \def\XINT_igctf_loop_f #1#2#3#4/%
602 {%
603     \xint_gob_til_exclam #4\XINT_igctf_end!%
604     \expandafter\XINT_igctf_loop_g
605     \romannumeral`&&@#4.{#2}{#3}#1%
606 }%
607 \def\XINT_igctf_loop_g #1.#2#3%
608 {%
609     \expandafter\XINT_igctf_loop_h\expandafter
610     {\romannumeral0\XINT_mul_fork #1\xint:#3\xint:}%
611     {\romannumeral0\XINT_mul_fork #1\xint:#2\xint:}%
612 }%
613 \def\XINT_igctf_loop_h #1#2%
614 {%
615     \expandafter\XINT_igctf_loop_i\expandafter {#2}{#1}%
616 }%
617 \def\XINT_igctf_loop_i #1#2#3#4%
618 {%
619     \XINT_igctf_loop_a {#3}{#4}{#1}{#2}%
620 }%
621 \def\XINT_igctf_end #1.#2#3#4#5{\xintrawwithzeros {#4/#5}}% 1.09b removes [0]
```

### 26.17. \xintCtoCv, \xintCstoCv

1.09m uses *xinttools*'s `\xintCSVtoList` on the argument of `\xintCstoCv` to allow spaces also before the commas. The original `\xintCstoCv` code became the one of the new `\xintCtoF` dealing with a braced rather than comma separated list.

```
622 \def\xintCstoCv {\romannumeral0\xintcstocv }%
623 \def\xintcstocv #1%
624 {%
625     \expandafter\XINT_ctcv_prep\romannumeral0\xintcsvtolist{#1}!%
626 }%
627 \def\xintCtoCv {\romannumeral0\xintctocv }%
628 \def\xintctocv #1%
629 {%
630     \expandafter\XINT_ctcv_prep\romannumeral`&&@#1!%
631 }%
632 \def\XINT_ctcv_prep
633 {%
634     \XINT_ctcv_loop_a {}1001%
635 }%
636 \def\XINT_ctcv_loop_a #1#2#3#4#5#6%
637 {%
638     \xint_gob_til_exclam #6\XINT_ctcv_end!%
639     \expandafter\XINT_ctcv_loop_b
640     \romannumeral0\xintrawwithzeros {#6}. {#2}{#3}{#4}{#5}{#1}%
641 }%
642 \def\XINT_ctcv_loop_b #1/#2.#3#4#5#6%
643 {%
644     \expandafter\XINT_ctcv_loop_c\expandafter
645     {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
646 }
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```

646     {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
647     {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
648                               {\XINT_mul_fork #1\xint:#4\xint:}}}%
649     {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
650                               {\XINT_mul_fork #1\xint:#3\xint:}}}%
651 }%
652 \def\XINT_ctcv_loop_c #1#2%
653 {%
654     \expandafter\XINT_ctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
655 }%
656 \def\XINT_ctcv_loop_d #1#2%
657 {%
658     \expandafter\XINT_ctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
659 }%
660 \def\XINT_ctcv_loop_e #1#2%
661 {%
662     \expandafter\XINT_ctcv_loop_f\expandafter{#2}{#1}%
663 }%
664 \def\XINT_ctcv_loop_f #1#2#3#4#5%
665 {%
666     \expandafter\XINT_ctcv_loop_g\expandafter
667     {\romannumeral0\xintrawwithzeros {#1/#2}}{#5}{#1}{#2}{#3}{#4}%
668 }%
669 \def\XINT_ctcv_loop_g #1#2{\XINT_ctcv_loop_a {#2}{#1}}}% 1.09b removes [0]
670 \def\XINT_ctcv_end #1.#2#3#4#5#6{ #6}%

```

### 26.18. \xintiCstoCv

```

671 \def\xintiCstoCv {\romannumeral0\xinticstocv }%
672 \def\xinticstocv #1%
673 {%
674     \expandafter\XINT_icstcv_prep \romannumeral`&&@#1,!,%
675 }%
676 \def\XINT_icstcv_prep
677 {%
678     \XINT_icstcv_loop_a {}1001%
679 }%
680 \def\XINT_icstcv_loop_a #1#2#3#4#5#6,%
681 {%
682     \xint_gob_til_exclam #6\XINT_icstcv_end!%
683     \expandafter
684     \XINT_icstcv_loop_b \romannumeral`&&@#6.{#2}{#3}{#4}{#5}{#1}%
685 }%
686 \def\XINT_icstcv_loop_b #1.#2#3#4#5%
687 {%
688     \expandafter\XINT_icstcv_loop_c\expandafter
689     {\romannumeral0\xintiiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
690     {\romannumeral0\xintiiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%
691     {{#2}{#3}}}%
692 }%
693 \def\XINT_icstcv_loop_c #1#2%
694 {%
695     \expandafter\XINT_icstcv_loop_d\expandafter {#2}{#1}%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, xintcfrac, *xintexpr*, *xinttrig*, *xintlog*

```
696 }%
697 \def\XINT_icstcv_loop_d #1#2%
698 {%
699     \expandafter\XINT_icstcv_loop_e\expandafter
700     {\romannumeral0\xintraewithzeros {#1/#2}}{#{#1}{#2}}%
701 }%
702 \def\XINT_icstcv_loop_e #1#2#3#4{\XINT_icstcv_loop_a {#4{#1}}#2#3}%
703 \def\XINT_icstcv_end #1.#2#3#4#5#6{ #6}% 1.09b removes [0]
```

### 26.19. \xintGctoCv

```
704 \def\xintGctoCv {\romannumeral0\xintgctocv }%
705 \def\xintgctocv #1%
706 {%
707     \expandafter\XINT_gctcv_prep \romannumeral`&&@#1+!/ %
708 }%
709 \def\XINT_gctcv_prep
710 {%
711     \XINT_gctcv_loop_a {}1001%
712 }%
713 \def\XINT_gctcv_loop_a #1#2#3#4#5#6+%
714 {%
715     \expandafter\XINT_gctcv_loop_b
716     \romannumeral0\xintraewithzeros {#6}.#{#2}{#3}{#4}{#5}{#1}%
717 }%
718 \def\XINT_gctcv_loop_b #1/#2.#3#4#5#6%
719 {%
720     \expandafter\XINT_gctcv_loop_c\expandafter
721     {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
722     {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
723     {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#6\xint:}%
724     {\XINT_mul_fork #1\xint:#4\xint:}}%
725     {\romannumeral0\xintiiadd {\XINT_mul_fork #2\xint:#5\xint:}%
726     {\XINT_mul_fork #1\xint:#3\xint:}}%
727 }%
728 \def\XINT_gctcv_loop_c #1#2%
729 {%
730     \expandafter\XINT_gctcv_loop_d\expandafter {\expandafter{#2}{#1}}%
731 }%
732 \def\XINT_gctcv_loop_d #1#2%
733 {%
734     \expandafter\XINT_gctcv_loop_e\expandafter {\expandafter{#2}{#1}}%
735 }%
736 \def\XINT_gctcv_loop_e #1#2%
737 {%
738     \expandafter\XINT_gctcv_loop_f\expandafter {#2}#1%
739 }%
740 \def\XINT_gctcv_loop_f #1#2%
741 {%
742     \expandafter\XINT_gctcv_loop_g\expandafter
743     {\romannumeral0\xintraewithzeros {#1/#2}}{#{#1}{#2}}%
744 }%
745 \def\XINT_gctcv_loop_g #1#2#3#4%
```

```

746 {%
747   \XINT_gctcv_loop_h {#4{#1}}{#2#3}% 1.09b removes [0]
748 }%
749 \def\XINT_gctcv_loop_h #1#2#3/%
750 {%
751   \xint_gob_til_exclam #3\XINT_gctcv_end!%
752   \expandafter\XINT_gctcv_loop_i
753   \romannumeral0\xintraewithzeros {#3}.#2{#1}%
754 }%
755 \def\XINT_gctcv_loop_i #1/#2.#3#4#5#6%
756 {%
757   \expandafter\XINT_gctcv_loop_j\expandafter
758   {\romannumeral0\XINT_mul_fork #1\xint:#6\xint:}%
759   {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
760   {\romannumeral0\XINT_mul_fork #2\xint:#4\xint:}%
761   {\romannumeral0\XINT_mul_fork #2\xint:#3\xint:}%
762 }%
763 \def\XINT_gctcv_loop_j #1#2%
764 {%
765   \expandafter\XINT_gctcv_loop_k\expandafter {\expandafter{#2}{#1}}%
766 }%
767 \def\XINT_gctcv_loop_k #1#2%
768 {%
769   \expandafter\XINT_gctcv_loop_l\expandafter {\expandafter{#2}#1}%
770 }%
771 \def\XINT_gctcv_loop_l #1#2%
772 {%
773   \expandafter\XINT_gctcv_loop_m\expandafter {\expandafter{#2}#1}%
774 }%
775 \def\XINT_gctcv_loop_m #1#2{\XINT_gctcv_loop_a {#2}#1}%
776 \def\XINT_gctcv_end #1.#2#3#4#5#6{ #6}%

```

## 26.20. \xintiGctoCv

```

777 \def\xintiGctoCv {\romannumeral0\xintigctocv }%
778 \def\xintigctocv #1%
779 {%
780   \expandafter\XINT_igctcv_prep \romannumeral`&&@#1+!/;%
781 }%
782 \def\XINT_igctcv_prep
783 {%
784   \XINT_igctcv_loop_a {}1001%
785 }%
786 \def\XINT_igctcv_loop_a #1#2#3#4#5#6+%
787 {%
788   \expandafter\XINT_igctcv_loop_b
789   \romannumeral`&&@#6.{#2}{#3}{#4}{#5}{#1}%
790 }%
791 \def\XINT_igctcv_loop_b #1.#2#3#4#5%
792 {%
793   \expandafter\XINT_igctcv_loop_c\expandafter
794   {\romannumeral0\xintiadd {#5}{\XINT_mul_fork #1\xint:#3\xint:}}%
795   {\romannumeral0\xintiadd {#4}{\XINT_mul_fork #1\xint:#2\xint:}}%

```

```

796     {{#2}{#3}}}%
797 }%
798 \def\XINT_igctcv_loop_c #1#2%
799 {%
800     \expandafter\XINT_igctcv_loop_f\expandafter {\expandafter{#2}{#1}}%
801 }%
802 \def\XINT_igctcv_loop_f #1#2#3#4/%
803 {%
804     \xint_gob_til_exclam #4\XINT_igctcv_end_a!%
805     \expandafter\XINT_igctcv_loop_g
806     \romannumeral`&&@#4.#1#2{#3}%
807 }%
808 \def\XINT_igctcv_loop_g #1.#2#3#4#5%
809 {%
810     \expandafter\XINT_igctcv_loop_h\expandafter
811     {\romannumeral0\XINT_mul_fork #1\xint:#5\xint:}%
812     {\romannumeral0\XINT_mul_fork #1\xint:#4\xint:}%
813     {{#2}{#3}}}%
814 }%
815 \def\XINT_igctcv_loop_h #1#2%
816 {%
817     \expandafter\XINT_igctcv_loop_i\expandafter {\expandafter{#2}{#1}}%
818 }%
819 \def\XINT_igctcv_loop_i #1#2{\XINT_igctcv_loop_k #2{#2#1}}%
820 \def\XINT_igctcv_loop_k #1#2%
821 {%
822     \expandafter\XINT_igctcv_loop_l\expandafter
823     {\romannumeral0\xintrawwithzeros {#1/#2}}%
824 }%
825 \def\XINT_igctcv_loop_l #1#2#3{\XINT_igctcv_loop_a {#3{#1}}#2}%1.09i removes [0]
826 \def\XINT_igctcv_end_a #1.#2#3#4#5%
827 {%
828     \expandafter\XINT_igctcv_end_b\expandafter
829     {\romannumeral0\xintrawwithzeros {#2/#3}}%
830 }%
831 \def\XINT_igctcv_end_b #1#2{ #2{#1}}% 1.09b removes [0]

```

## 26.21. \xintFtoCv

Still uses \xinticstocv \xintFtoCs rather than \xintctocv \xintFtoC.

```

832 \def\xintFtoCv {\romannumeral0\xintftocv }%
833 \def\xintftocv #1%
834 {%
835     \xinticstocv {\xintFtoCs {#1}}%
836 }%

```

## 26.22. \xintFtoCCv

```

837 \def\xintFtoCCv {\romannumeral0\xintftoccv }%
838 \def\xintftoccv #1%
839 {%
840     \xintigctocv {\xintFtoCC {#1}}%
841 }%

```



### 26.23. \xintCntoF

Modified in 1.06 to give the N first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that.

```

842 \def\xintCntoF {\romannumeral0\xintcntof}%
843 \def\xintcntof #1%
844 {%
845   \expandafter\XINT_cntf\expandafter {\the\numexpr #1}%
846 }%
847 \def\XINT_cntf #1#2%
848 {%
849   \ifnum #1>\xint_c_
850     \xint_afterfi {\expandafter\XINT_cntf_loop\expandafter
851                   {\the\numexpr #1-1\expandafter}\expandafter
852                   {\romannumeral`&&@#2{#1}}{#2}}}%
853   \else
854     \xint_afterfi
855       {\ifnum #1=\xint_c_
856         \xint_afterfi {\expandafter\space \romannumeral`&&@#2{0}}}%
857       \else \xint_afterfi { }% 1.09m now returns nothing.
858       \fi}%
859   \fi
860 }%
861 \def\XINT_cntf_loop #1#2#3%
862 {%
863   \ifnum #1>\xint_c_ \else \XINT_cntf_exit \fi
864   \expandafter\XINT_cntf_loop\expandafter
865   {\the\numexpr #1-1\expandafter }\expandafter
866   {\romannumeral0\xintadd {\xintDiv {1[0]}{#2}}{#3{#1}}}%
867   {#3}%
868 }%
869 \def\XINT_cntf_exit \fi
870   \expandafter\XINT_cntf_loop\expandafter
871   #1\expandafter #2#3%
872 {%
873   \fi\xint_gobble_ii #2%
874 }%

```

### 26.24. \xintGCntoF

Modified in 1.06 to give the N argument first to a `\numexpr` rather than expanding twice. I just use `\the\numexpr` and maintain the previous code after that.

```

875 \def\xintGCntoF {\romannumeral0\xintgcntof}%
876 \def\xintgcntof #1%
877 {%
878   \expandafter\XINT_gcntf\expandafter {\the\numexpr #1}%
879 }%
880 \def\XINT_gcntf #1#2#3%
881 {%
882   \ifnum #1>\xint_c_
883     \xint_afterfi {\expandafter\XINT_gcntf_loop\expandafter
884                   {\the\numexpr #1-1\expandafter}\expandafter
885                   {\romannumeral`&&@#2{#1}}{#2}{#3}}}%

```

```

886 \else
887 \xint_afterfi
888 {\ifnum #1=\xint_c_
889 \xint_afterfi {\expandafter\space\romannumeral`&&@#2{0}}}%
890 \else \xint_afterfi { }% 1.09m now returns nothing rather than 0/1[0]
891 \fi}%
892 \fi
893 }%
894 \def\xINT_gcntf_loop #1#2#3#4%
895 {%
896 \ifnum #1>\xint_c_ \else \XINT_gcntf_exit \fi
897 \expandafter\xINT_gcntf_loop\expandafter
898 {\the\numexpr #1-1\expandafter }\expandafter
899 {\romannumeral0\xintadd {\xintDiv {#4{#1}}{#2}}{#3{#1}}}%
900 {#3}{#4}}%
901 }%
902 \def\xINT_gcntf_exit \fi
903 \expandafter\xINT_gcntf_loop\expandafter
904 #1\expandafter #2#3#4%
905 {%
906 \fi\xint_gobble_ii #2%
907 }%

```

## 26.25. \xintCntoCs

Modified in 1.09m: added spaces after the commas in the produced list. Moreover the coefficients are not braced anymore. A slight induced limitation is that the macro argument should not contain some explicit comma (cf. `\XINT_cntcs_exit_b`), hence `\xintCntoCs {macro,}` with `\def\macro,ro,#1{<stuff>}` would crash. Not a very serious limitation, I believe.

```

908 \def\xintCntoCs {\romannumeral0\xintcntocs }%
909 \def\xintcntocs #1%
910 {%
911 \expandafter\xINT_cntcs\expandafter {\the\numexpr #1}%
912 }%
913 \def\xINT_cntcs #1#2%
914 {%
915 \ifnum #1<0
916 \xint_afterfi { }% 1.09i: a 0/1[0] was here, now the macro returns nothing
917 \else
918 \xint_afterfi {\expandafter\xINT_cntcs_loop\expandafter
919 {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
920 {\romannumeral`&&@#2{#1}}{#2}}% produced coeff not braced
921 \fi
922 }%
923 \def\xINT_cntcs_loop #1#2#3%
924 {%
925 \ifnum #1>-\xint_c_i \else \XINT_cntcs_exit \fi
926 \expandafter\xINT_cntcs_loop\expandafter
927 {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
928 {\romannumeral`&&@#3{#1}, #2}{#3}% space added, 1.09m
929 }%
930 \def\xINT_cntcs_exit \fi

```

```

931 \expandafter\XINT_cntcs_loop\expandafter
932 #1\expandafter #2#3%
933 {%
934 \fi\XINT_cntcs_exit_b #2%
935 }%
936 \def\XINT_cntcs_exit_b #1,{}% romannumeral stopping space already there

```

## 26.26. \xintCntoGC

Modified in 1.06 to give the N first to a `\numexpr` rather than expanding twice. I just use `\the\nuumexpr` and maintain the previous code after that.

1.09m maintains the braces, as the coeff are allowed to be fraction and the slash can not be naked in the GC format, contrarily to what happens in `\xintCntoCs`. Also the separators given to `\xintGctoGCx` may then fetch the coefficients as argument, as they are braced.

```

937 \def\xintCntoGC {\romannumeral0\xintcntogc }%
938 \def\xintcntogc #1%
939 {%
940 \expandafter\XINT_cntgc\expandafter {\the\numexpr #1}%
941 }%
942 \def\XINT_cntgc #1#2%
943 {%
944 \ifnum #1<0
945 \xint_afterfi { }% 1.09i there was as strange 0/1[0] here, removed
946 \else
947 \xint_afterfi {\expandafter\XINT_cntgc_loop\expandafter
948 {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
949 {\expandafter{\romannumeral`&&@#2{#1}}{#2}}}%
950 \fi
951 }%
952 \def\XINT_cntgc_loop #1#2#3%
953 {%
954 \ifnum #1>-\xint_c_i \else \XINT_cntgc_exit \fi
955 \expandafter\XINT_cntgc_loop\expandafter
956 {\the\numexpr #1-\xint_c_i\expandafter }\expandafter
957 {\expandafter{\romannumeral`&&@#3{#1}}+1/#2}{#3}%
958 }%
959 \def\XINT_cntgc_exit \fi
960 \expandafter\XINT_cntgc_loop\expandafter
961 #1\expandafter #2#3%
962 {%
963 \fi\XINT_cntgc_exit_b #2%
964 }%
965 \def\XINT_cntgc_exit_b #1+1/{ }%

```

## 26.27. \xintGCntoGC

Modified in 1.06 to give the N first to a `\numexpr` rather than expanding twice. I just use `\the\nuumexpr` and maintain the previous code after that.

```

966 \def\xintGCntoGC {\romannumeral0\xintgcntogc }%
967 \def\xintgcntogc #1%
968 {%
969 \expandafter\XINT_gcntgc\expandafter {\the\numexpr #1}%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

970 }%
971 \def\XINT_gcntgc #1#2#3%
972 {%
973   \ifnum #1<0
974     \xint_afterfi { }% 1.09i now returns nothing
975   \else
976     \xint_afterfi {\expandafter\XINT_gcntgc_loop\expandafter
977                   {\the\numexpr #1-\xint_c_i\expandafter}\expandafter
978                   {\expandafter{\romannumeral`&&@#2{#1}}{#2}{#3}}}%
979   \fi
980 }%
981 \def\XINT_gcntgc_loop #1#2#3#4%
982 {%
983   \ifnum #1>-\xint_c_i \else \XINT_gcntgc_exit \fi
984   \expandafter\XINT_gcntgc_loop_b\expandafter
985   {\expandafter{\romannumeral`&&@#4{#1}}/#2}{#3{#1}}{#1}{#3}{#4}%
986 }%
987 \def\XINT_gcntgc_loop_b #1#2#3%
988 {%
989   \expandafter\XINT_gcntgc_loop\expandafter
990   {\the\numexpr #3-\xint_c_i \expandafter}\expandafter
991   {\expandafter{\romannumeral`&&@#2}+#1}%
992 }%
993 \def\XINT_gcntgc_exit \fi
994   \expandafter\XINT_gcntgc_loop_b\expandafter #1#2#3#4#5%
995 {%
996   \fi\XINT_gcntgc_exit_b #1%
997 }%
998 \def\XINT_gcntgc_exit_b #1/{ }%

```

### 26.28. \xintCstoGC

```

999 \def\xintCstoGC {\romannumeral0\xintcstogc }%
1000 \def\xintcstogc #1%
1001 {%
1002   \expandafter\XINT_cstc_prep \romannumeral`&&@#1,!,%
1003 }%
1004 \def\XINT_cstc_prep #1,{\XINT_cstc_loop_a {{#1}}}%
1005 \def\XINT_cstc_loop_a #1#2,%
1006 {%
1007   \xint_gob_til_exclam #2\XINT_cstc_end!%
1008   \XINT_cstc_loop_b {#1}{#2}%
1009 }%
1010 \def\XINT_cstc_loop_b #1#2{\XINT_cstc_loop_a {#1+1/{#2}}}%
1011 \def\XINT_cstc_end!\XINT_cstc_loop_b #1#2{ #1}%

```

### 26.29. \xintGctoGC

```

1012 \def\xintGctoGC {\romannumeral0\xintgctogc }%
1013 \def\xintgctogc #1%
1014 {%
1015   \expandafter\XINT_gctgc_start \romannumeral`&&@#1+!/,%
1016 }%

```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1017 \def\XINT_gctgc_start {\XINT_gctgc_loop_a {}}%
1018 \def\XINT_gctgc_loop_a #1#2+#3/%
1019 {%
1020     \xint_gob_til_exclam #3\XINT_gctgc_end!%
1021     \expandafter\XINT_gctgc_loop_b\expandafter
1022     {\romannumeral`&&@#2}{#3}{#1}%
1023 }%
1024 \def\XINT_gctgc_loop_b #1#2%
1025 {%
1026     \expandafter\XINT_gctgc_loop_c\expandafter
1027     {\romannumeral`&&@#2}{#1}%
1028 }%
1029 \def\XINT_gctgc_loop_c #1#2#3%
1030 {%
1031     \XINT_gctgc_loop_a {#3{#2}+{#1}/}%
1032 }%
1033 \def\XINT_gctgc_end!\expandafter\XINT_gctgc_loop_b
1034 {%
1035     \expandafter\XINT_gctgc_end_b
1036 }%
1037 \def\XINT_gctgc_end_b #1#2#3{ #3{#1}}%
1038 \XINTrestorecatcodesendinginput%
```

## 27. Package xintexpr implementation

This is release 1.4o of 2025/09/06.

### Contents

27.1	READ ME! Important warnings and explanations relative to the status of the code source at the time of the 1.4 release . . . . .	559
27.2	Old comments . . . . .	561
27.3	Catcodes, $\varepsilon$ -T <sub>E</sub> X and reload detection . . . . .	561
27.4	Package identification . . . . .	563
27.5	<code>\XINTfstop</code> . . . . .	563
27.6	<code>\xintDigits*</code> , <code>\xintSetDigits*</code> , <code>\xintreloadscilibs</code> . . . . .	563
27.7	<code>\XINTdigitsormax</code> . . . . .	564
27.8	Support for output and transform of nested braced contents as core data type . . . . .	564
27.8.1	Bracketed list rendering with prettifying of leaves from nested braced contents . . . . .	564
27.8.2	Flattening nested braced contents . . . . .	565
27.8.3	Braced contents rendering via a T <sub>E</sub> X alignment with prettifying of leaves . . . . .	565
27.8.4	Transforming all leaves within nested braced contents . . . . .	566
27.9	Top level user T <sub>E</sub> X interface: <code>\xinteval</code> , <code>\xintfloateval</code> , <code>\xintiieval</code> . . . . .	567
27.9.1	<code>\xintexpr</code> , <code>\xintiexpr</code> , <code>\xintfloatexpr</code> , <code>\xintiieexpr</code> . . . . .	568
27.9.2	<code>\XINT_expr_wrap</code> , <code>\XINT_iiexpr_wrap</code> , <code>\XINT_flexpr_wrap</code> . . . . .	570
27.9.3	<code>\XINTexprprint</code> , <code>\XINTiexprprint</code> , <code>\XINTiiexprprint</code> , <code>\XINTflexprprint</code> . . . . .	570
27.9.4	<code>\xintthe</code> , <code>\xintthealign</code> , <code>\xinttheexpr</code> , <code>\xinttheiexpr</code> , <code>\xintthefloatexpr</code> , <code>\xinttheiieexpr</code> . . . . .	570
27.9.5	<code>\xintbareeval</code> , <code>\xintbarefloateval</code> , <code>\xintbareiieval</code> . . . . .	571
27.9.6	<code>\xintthebareeval</code> , <code>\xintthebarefloateval</code> , <code>\xintthebareiieval</code> . . . . .	571
27.9.7	<code>\xinteval</code> , <code>\xintiieval</code> , <code>\xintfloateval</code> , <code>\xintiieval</code> . . . . .	572
27.9.8	<code>\xintboolexpr</code> , <code>\XINT_boolexpr_print</code> , <code>\xinttheboolexpr</code> . . . . .	573
27.9.9	<code>\xintifboolexpr</code> , <code>\xintifboolfloatexpr</code> , <code>\xintifbooliieexpr</code> . . . . .	573
27.9.10	<code>\xintifsgnexpr</code> , <code>\xintifsgnfloatexpr</code> , <code>\xintifsgniieexpr</code> . . . . .	574
27.9.11	<code>\xint_FracToSci_x</code> . . . . .	574
27.9.12	Small bits we have to put somewhere . . . . .	575
	<code>\xintthecoords</code> . . . . .	575
	<code>\xintthespaceseparated</code> . . . . .	576
27.10	Hooks into the numeric parser for usage by the <code>\xintdeffunc</code> symbolic parser . . . . .	576
27.11	<code>\XINT_expr_getnext</code> : fetch some value then an operator and present them to last waiter with the found operator precedence, then the operator, then the value . . . . .	578
27.12	<code>\XINT_expr_startint</code> . . . . .	582
27.12.1	Integral part (skipping zeroes) . . . . .	583
27.12.2	Fractional part . . . . .	585
27.12.3	Scientific notation . . . . .	586
27.12.4	Hexadecimal numbers . . . . .	587
27.12.5	Octal numbers . . . . .	590
27.12.6	Binary numbers . . . . .	592
27.12.7	<code>\XINT_expr_startfunc</code> : collecting names of functions and variables . . . . .	593
27.12.8	<code>\XINT_expr_func</code> : dispatch to variable replacement or to function execution . . . . .	594
27.13	<code>\XINT_expr_op_`</code> : launch function or pseudo-function, or evaluate variable and insert operator of multiplication in front of parenthesized contents . . . . .	595
27.14	<code>\XINT_expr_op_</code> : replace a variable by its value and then fetch next operator . . . . .	596
27.15	<code>\XINT_expr_getop</code> : fetch the next operator or closing parenthesis or end of expression . . . . .	597
27.16	Expansion spanning; opening and closing parentheses . . . . .	600
27.17	The comma as binary operator . . . . .	602

27.18	The minus as prefix operator of variable precedence level . . . . .	603
27.19	The * as Python-like «unpacking» prefix operator . . . . .	604
27.20	Infix operators . . . . .	604
27.20.1	&&,   , //, /:, +, -, *, /, ^, **, 'and', 'or', 'xor', and 'mod' . . . . .	605
27.20.2	.., ..[, and ].. for a..b and a..[b]..c syntax . . . . .	607
27.20.3	<, >, ==, <=, >=, != with Python-like chaining . . . . .	609
27.20.4	Support macros for .., ..[ and ].. . . . .	610
	\xintSeq:tl:x . . . . .	611
	\xintiiSeq:tl:x . . . . .	611
	\xintSeqA, \xintiiSeqA . . . . .	612
	\xintSeqB:tl:x . . . . .	612
	\xintiiSeqB:tl:x . . . . .	613
27.21	Square brackets [] both as a container and a Python slicer . . . . .	613
27.21.1	[...] as «oneple» constructor . . . . .	613
27.21.2	[...] brackets and : operator for NumPy-like slicing and item indexing syntax . . . . .	614
27.21.3	Macro layer implementing indexing and slicing . . . . .	616
27.22	Support for raw A/B[N] . . . . .	620
27.23	? as two-way and ?? as three-way «short-circuit» conditionals . . . . .	621
27.24	! as postfix factorial operator . . . . .	622
27.25	User defined variables . . . . .	622
27.25.1	\xintdefvar, \xintdefiivar, \xintdeffloatvar . . . . .	622
27.25.2	\xintunassignvar . . . . .	626
27.26	Support for dummy variables . . . . .	627
27.26.1	\xintnewdummy . . . . .	627
27.26.2	\xintensuredummy, \xintrestorevariable . . . . .	628
27.26.3	Checking (without expansion) that a symbolic expression contains correctly nested parentheses . . . . .	629
27.26.4	Fetching balanced expressions E1, E2 and a variable name Name from E1, Name=E2) . . . . .	630
27.26.5	Fetching a balanced expression delimited by a semi-colon . . . . .	630
27.26.6	Low-level support for omit and abort keywords, the break() function, the n++ construct and the semi-colon as used in the syntax of seq(), add(), mul(), iter(), rseq(), iterr(), rrseq(), subsm(), subsn(), ndseq(), ndmap() . . . . .	631
	The n++ construct . . . . .	631
	The break() function . . . . .	631
	The omit and abort keywords . . . . .	631
	The semi-colon . . . . .	632
27.26.7	Reserved dummy variables @, @1, @2, @3, @4, @@, @@(1), ..., @@@, @@@(1), ... for recursions . . . . .	632
27.27	Pseudo-functions involving dummy variables and generating scalars or sequences . . . . .	633
27.27.1	Comments . . . . .	633
27.27.2	subsn(): substitution of one variable . . . . .	635
27.27.3	subsm(): simultaneous independent substitutions . . . . .	636
27.27.4	subsn(): leaner syntax for nesting (possibly dependent) substitutions . . . . .	637
27.27.5	seq(): sequences from assigning values to a dummy variable . . . . .	639
27.27.6	iter() . . . . .	640
27.27.7	add(), mul() . . . . .	641
27.27.8	rseq() . . . . .	642
27.27.9	iterr() . . . . .	643
27.27.10	rrseq() . . . . .	644
27.28	Pseudo-functions related to N-dimensional hypercubic lists . . . . .	645
27.28.1	ndseq() . . . . .	645
27.28.2	ndmap() . . . . .	646

27.28.3	<code>ndfillraw()</code> . . . . .	648
27.29	Other pseudo-functions: <code>bool()</code> , <code>togl()</code> , <code>protect()</code> , <code>qraw()</code> , <code>qint()</code> , <code>qfrac()</code> , <code>qfloat()</code> , <code>qrand()</code> , <code>random()</code> , <code>rbit()</code> . . . . .	648
27.30	Regular built-in functions: <code>num()</code> , <code>reduce()</code> , <code>preduce()</code> , <code>abs()</code> , <code>sgn()</code> , <code>frac()</code> , <code>floor()</code> , <code>ceil()</code> , <code>sqr()</code> , <code>?</code> , <code>!</code> , <code>not()</code> , <code>odd()</code> , <code>even()</code> , <code>isint()</code> , <code>isone()</code> , <code>factorial()</code> , <code>sqrt()</code> , <code>sqrtr()</code> , <code>inv()</code> , <code>round()</code> , <code>trunc()</code> , <code>float()</code> , <code>sfloat()</code> , <code>ilog10()</code> , <code>divmod()</code> , <code>mod()</code> , <code>binomial()</code> , <code>pfactorial()</code> , <code>randrange()</code> , <code>iquo()</code> , <code>irem()</code> , <code>gcd()</code> , <code>lcm()</code> , <code>max()</code> , <code>min()</code> , <code>`+`()</code> , <code>`*`()</code> , <code>all()</code> , <code>any()</code> , <code>xor()</code> , <code>len()</code> , <code>first()</code> , <code>last()</code> , <code>reversed()</code> , <code>if()</code> , <code>ifint()</code> , <code>ifone()</code> , <code>ifsgn()</code> , <code>nuple()</code> , <code>unpack()</code> , <code>flat()</code> and <code>zip()</code> . . . . .	649
27.31	User declared functions . . . . .	663
27.31.1	<code>\xintdeffunc</code> , <code>\xintdefiifunc</code> , <code>\xintdeffloatfunc</code> . . . . .	664
27.31.2	<code>\xintdefufunc</code> , <code>\xintdefiufunc</code> , <code>\xintdeffloatufunc</code> . . . . .	667
27.31.3	<code>\xintunassignexprfunc</code> , <code>\xintunassigniexprfunc</code> , <code>\xintunassignfloatexprfunc</code> . . . . .	668
27.31.4	<code>\xintNewFunction</code> . . . . .	668
27.31.5	Mysterious stuff . . . . .	670
27.31.6	<code>\XINT_expr_redefinemacros</code> . . . . .	682
27.31.7	<code>\xintNewExpr</code> , <code>\xintNewIExpr</code> , <code>\xintNewFloatExpr</code> , <code>\xintNewIIExpr</code> . . . . .	683
27.31.8	<code>\xintexprSafeCatcodes</code> , <code>\xintexprRestoreCatcodes</code> . . . . .	686
27.32	Matters related to loading log and trig libraries . . . . .	687

## 27.1. READ ME! Important warnings and explanations relative to the status of the code source at the time of the 1.4 release

At release 1.4 the `csname` encapsulation of intermediate evaluations during parsing of expressions is dropped, and `xintexpr` requires the `\expanded` primitive. This means that there is no more impact on the string pool. And as internal storage now uses simply core `\TeX` syntax with braces rather than comma separated items inside a `csname` dummy control sequence, it became much easier to let the `[...]` syntax be associated to a true internal type of «tuple» or «list».

The output of `\xintexpr` (after `\romannumeral0` or `\romannumeral-`0` triggered expansion or double expansion) is thus modified at 1.4. It now looks like this:<sup>76</sup>

```
\XINTfstop \XINTexprprint .{{<number>}} in simplest case
\XINTfstop \XINTexprprint .{{...}}...{{...}} in general case
```

where `...` stands for nested braces ultimately ending in `{<num. rep.>}` leaves. The `<num. rep.>` stands for some internal representation of numeric data. It may be empty, and currently as well as probably in future uses only catcode 12 tokens (no spaces currently).

`{{}}` corresponds (in input as in output) to `[]`. The external TeX braces also serve as set-theoretical braces. The comma is concatenation, so for example `[]`, `[]` will become `{{}}{{}}`, or rather `{{}}` if sub-unit of something else.

The associated vocabulary is explained in the user manual and we avoid too much duplication here. `xintfrac` numerical macros receiving an empty argument usually handle it as being 0, but this is not the case of the `xintcore` macros supporting `\xintiexpr`, they usually break if exercised on some empty argument.

The above expansion result `\XINTfstop \XINTexprprint .{{<num1>}}{{<num2>}}...` uses only normal catcodes: the backslash, regular braces, ascii letters and catcode 12 characters. Scientific notation is internally converted to raw `xintfrac` representation `[N]`.

Additional data may be located before the dot; this is the case only for `\xintfloatexpr` currently. As `xintexpr` actually defines three parsers `\xintexpr`, `\xintiexpr` and `\xintfloatexpr` but tries to share as much code as possible, some overhead is induced to fit all into the same mold.

<sup>76</sup> The `\XINTfstop` was a simple `\def \XINTfstop{ \noexpand \XINTfstop }` at 1.4 but has been converted into a fancier token at 1.5, to avoid the “blinking” effect when submitted to finitely many expansions à la `\expandafter` as first token of a replacement text.



`\XINTfstop` stops `\romannumeral-`0` (or `0`) type spanned expansion, and is invariant under `\edef`, but simply disappears in typesetting context.<sup>77</sup> It is thus now legal to use `\xintexpr` directly in typesetting flow.

`\XINTexprprint` is `\protected`.

The f-expansion of an `\xintexpr` <expression>`\relax` is a complete expansion, i.e. one whose result remains invariant under `\edef`.

`\xintthe\xintexpr` <expression>`\relax` or `\xinteval`{<expression>} serve as formerly to deliver the explicit digits, or more exactly some prettifying view of the actual <internal number representation>.

Nested contents like this

```
{\1}{\2}{3}{\4}{5}{6}}{9}}
```

will get delivered using nested square brackets like that

```
1, [2, 3, [4, 5, 6]], 9
```

and as conversely `\xintexpr` 1, [2, 3, [4, 5, 6]], 9`\relax` expands to

```
\XINTfstop \XINTexprprint .{\1}{\2}{3}{\4}{5}{6}}{9}}
```

we obtain the gratifying result that

```
\xinteval{1, [2, 3, [4, 5, 6]], 9}
```

expands to

```
1, [2, 3, [4, 5, 6]], 9
```

See user manual for explanations on the plasticity of `\xintexpr` syntax regarding functions with multiple arguments, and the 1.4 «unpacking» Python-like `*` prefix operator.

I have suppressed (from the public dtx) many big chunks of comments. Some became obsolete and need to be updated, others are currently of value only to the author as a historical record.

ATTENTION! As the removal process itself took too much time, I ended up leaving as is many comments which are obsoleted and wrong to various degrees after the 1.4 release. Precedence levels of operators have all been doubled to make room for new constructs

Even comments added during 1.4 developement may now be obsolete because the preparation of 1.4 took a few weeks and that's enough of duration to provide the author many chances to contradict in the code what has been already commented upon.

Thus don't believe (fully) anything which is said here!

Warning: in text below and also in left-over old comments I may refer to «until» and «op» macros; due to the change of data storage at 1.4, I needed to refactor a bit the way expansion is controlled, and the situation now is mainly governed by «op», «exec», «check-» and «checkp» macros the latter three replacing the two «until\_a» and «until\_b» of former code. This allows to diminish the number of times an accumulated result will be grabbed in order to propagate expansion to its right. Formerly this was not an issue because such things were only a single token! I do not describe here how this is all articulated but it is not hard to see it from the code (the hardest thing in all such matter was in 2013 to actually write how the expansion would be initially launched because to do that one basically has to understand the mechanism in its whole and such things are not easy to develop piecemeal). Another thing to keep in mind is that operators in truth have a left precedence (i.e. the precedence they show to operators arising earlier) and a right precedence (which determines how they react to operators coming after them from the right). Only the first one is usually encapsulated in a chardef, the second one is most of the times identical to the first one and if not it is only virtual but implemented via `\ifcase` of `\ifnum` branching. A final remark is that some things are achieved by special «op» macros, which are a favorite tool to hack into the normal regular flow of things, via injection of special syntax elements. I did not rename these macros for avoiding too large git diffs, and besides the nice thing is that the 1.4 refactoring minimally had to modify them, and all hacky things using them kept on working with

<sup>77</sup> It is not `\let` to `\relax` as it must be distinguished from it in `\ifx` tests.

not a single modification. And a post-scriptum is that advanced features crucially exploit injecting sub-`\xintexpr`-expressions, as all is expandable there is no real «context» (only a minimal one) which one would have to perhaps store and restore and doing this sub-expression injection is rather cheap and efficient operation.

## 27.2. Old comments

These general comments were last updated at the end of the 1.09x series in 2014. The principles remain in place to this day but refer to [CHANGES.html](#) for some significant evolutions since.

The first version was released in June 2013. I was greatly helped in this task of writing an expandable parser of infix operations by the comments provided in `l3fp-parse.dtx` (in its version as available in April-May 2013). One will recognize in particular the idea of the ``until'` macros; I have not looked into the actual `l3fp` code beyond the very useful comments provided in its documentation.

A main worry was that my data has no a priori bound on its size; to keep the code reasonably efficient, I experimented with a technique of storing and retrieving data expandably as *names* of control sequences. Intermediate computation results are stored as control sequences `\.=a/b[n]`.

Roughly speaking, the parser mechanism is as follows: at any given time the last found ``operator'` has its associated `until` macro awaiting some news from the token flow; first `getnext` expands forward in the hope to construct some number, which may come from a parenthesized sub-expression, from some braced material, or from a digit by digit scan. After this number has been formed the next operator is looked for by the `getop` macro. Once `getop` has finished its job, `until` is presented with three tokens: the first one is the precedence level of the new found operator (which may be an end of expression marker), the second is the operator character token (earlier versions had here already some macro name, but in order to keep as much common code to `expr` and `floatexpr` common as possible, this was modified) of the new found operator, and the third one is the newly found number (which was encountered just before the new operator).

The `until` macro of the earlier operator examines the precedence level of the new found one, and either executes the earlier operator (in the case of a binary operation, with the found number and a previously stored one) or it delays execution, giving the hand to the `until` macro of the operator having been found of higher precedence.

A minus sign acting as prefix gets converted into a (unary) operator inheriting the precedence level of the previous operator.

Once the end of the expression is found (it has to be marked by a `\relax`) the final result is output as four tokens (five tokens since 1.09j) the first one a catcode 11 exclamation mark, the second one an error generating macro, the third one is a protection mechanism, the fourth one a printing macro and the fifth is `\.=a/b[n]`. The prefix `\xintthe` makes the output printable by killing the first three tokens.

## 27.3. Catcodes, $\varepsilon$ -TeX and reload detection

The code for reload detection was initially copied from HEIKO OBERDIEK's packages, then modified.

The method for catcodes was also initially directly inspired by these packages.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode44=12 % ,
8 \catcode46=12 % .
9 \catcode58=12 % :
10 \catcode94=7 % ^

```

```

11 \def\empty{}\def\space{ }\newlinechar10
12 \def\z{\endgroup}%
13 \expandafter\let\expandafter\x\csname ver@xintexpr.sty\endcsname
14 \expandafter\let\expandafter\w\csname ver@xintfrac.sty\endcsname
15 \expandafter\let\expandafter\t\csname ver@xinttools.sty\endcsname
16 \expandafter\let\expandafter\b\csname ver@xintbinhex.sty\endcsname
17 % I assume engines do not exist providing \expanded but not \numexpr
18 \expandafter\ifx\csname expanded\endcsname\relax
19   \expandafter\ifx\csname PackageWarningNoLine\endcsname\relax
20     \immediate\write128{^^JPackage xintexpr Warning:^^J%
21       \space\space\space\space
22       \expanded not available, aborting input.^^J}%
23   \else
24     \PackageWarningNoLine{xintexpr}{\expanded not available, aborting input}%
25   \fi
26 \def\z{\endgroup\endinput}%
27 \else
28   \ifx\x\relax % not LaTeX (perhaps Plain+miniltx), first loading of xintexpr.sty
29     \ifx\w\relax % but xintfrac.sty not yet loaded (made miniltx robust 2022/06/09)
30       \expandafter\def\expandafter\z\expandafter
31         {\z\input xintfrac.sty\relax}%
32     \fi
33     \ifx\t\relax % but xinttools.sty not yet loaded.
34       \expandafter\def\expandafter\z\expandafter
35         {\z\input xinttools.sty\relax}%
36     \fi
37     \ifx\b\relax % but xintbinhex.sty not yet loaded.
38       \expandafter\def\expandafter\z\expandafter
39         {\z\input xintbinhex.sty\relax}%
40     \fi
41   \else
42     \ifx\x\empty % LaTeX, first loading,
43     % variable is initialized, but \ProvidesPackage not yet seen
44     \ifx\w\relax % xintfrac not yet loaded.
45       \expandafter\def\expandafter\z\expandafter
46         {\z\RequirePackage{xintfrac}}%
47     \fi
48     \ifx\t\relax % xinttools not yet loaded.
49       \expandafter\def\expandafter\z\expandafter
50         {\z\RequirePackage{xinttools}}%
51     \fi
52     \ifx\b\relax % xintbinhex not yet loaded.
53       \expandafter\def\expandafter\z\expandafter
54         {\z\RequirePackage{xintbinhex}}%
55     \fi
56   \else
57     \def\z{\endgroup\endinput}% xintexpr already loaded.
58   \fi
59 \fi
60 \fi
61 \z%
62 \XINTsetupcatcodes%

```

## 27.4. Package identification

The `!` is of catcode LETTER for most of the duration of the package. Prior to 1.4 it was the first token in the complete expansion of `\xintexpr...\relax` (i.e. without `\xintthe` prefix). It has since lost to `\XINTfstop` this prominent rôle, but its usage in various places of the code as delimiter in otherwise catcode 12 contexts has stuck.

```
63 \XINT_providespackage
64 \ProvidesPackage{xintexpr}%
65 [2025/09/06 v1.4o Expandable expression parser (JFB)]%
66 \catcode`! 11
```

## 27.5. `\XINTfstop`

Added at 1.4 (2020/01/31).

This replaced in 2019 the legacy usage of a catcode 11 exclamation mark to signal the start of a sub-expression (it was followed by some other tokens).

From 2022 (aiming at 1.5) to 2025 (releasing 1.4n) I had replaced the definition used here with a fancier one originating in a ``fake'' `\relax` from `unravel`. I think in 2022, `unravel` used a frozen `let` to `\the`, which I had transferred here as a definition of `\XINTfstop`, now reverted. When I checked again `unravel` in 2025, I saw a definition of the type

```
\expandafter\let\expandafter\foo\noexpand\foo
```

If I did that with `\foo` being `\XINTfstop` then `\XINTfstop` would test equal via `\ifx` to the `unravel` or any such `\foo` (if `\foo` is at first undefined)...

The inconvenience with our more naive `\XINTfstop` here, is that expanding a finite number of times `\xintexpr...\relax` depends on parity of number of expansions. Not really a problem as far as I can tell. The advantage is that `\ifx` will never confuse it with something else.

```
67 \def\XINTfstop{\noexpand\XINTfstop}%
```

## 27.6. `\xintDigits*`, `\xintSetDigits*`, `\xintreloadscilibs`

1.3f. 1.4e added some `\xintGuardDigits` and `\XINTdigitsex` mechanism but it was finally removed, due to pending issues of user interface, functionality, and documentation (the worst part) for whose resolution no time was left.

```
68 \def\xintreloadscilibs{\xintreloadxintlog\xintreloadxinttrig}%
69 \def\xintDigits {\futurelet\XINT_token\xintDigits_i}%
70 \def\xintDigits_i#1={\afterassignment\xintDigits_j\mathchardef\XINT_digits=}%
71 \def\xintDigits_j#1%
72 {%
73   \let\XINTdigits=\XINT_digits
74   \ifx*\XINT_token\expandafter\xintreloadscilibs\fi
75 }%
76 \let\xintfracSetDigits\xintSetDigits
77 \def\xintSetDigits#1#{\if\relax\detokenize{#1}\relax\expandafter\xintfracSetDigits
78   \else\expandafter\xintSetDigits_a\fi}%
79 \def\xintSetDigits_a#1%
80 {%
81   \mathchardef\XINT_digits=\numexpr#1\relax
82   \let\XINTdigits\XINT_digits
83   \xintreloadscilibs
84 }%
```

## 27.7. \XINTdigitsormax

1.4f. To not let xintlog and xinttrig work with, and produce, long mantissas exceeding the supported range for accuracy of the math functions. The official maximal value is 62, let's set the cut-off at 64.

A priori, no need for `\expandafter`, always ends up expanded in `\numexpr` (I saw also in an `\edef` in xinttrig as argument to `\xintReplicate` prior to its `\numexpr`).

```
85 \def\XINTdigitsormax{\ifnum\XINTdigits>\xint_c_ii^vi\xint_c_ii^vi\else\XINTdigits\fi}%
```

## 27.8. Support for output and transform of nested braced contents as core data type

New at 1.4, of course. The former `\csname.=...\endcsname` encapsulation technique made very difficult implementation of nested structures.

### 27.8.1. Bracketed list rendering with prettifying of leaves from nested braced contents

**Added at 1.4 (2020/01/31).** The braces in `\XINT:expr:toblistwith` are there because there is an `\expanded` trigger.

**Modified at 1.4d (2021/03/29).** Add support for the polexpr 0.8 polynomial type. See `\XINT:expr:toblist_a`.

**Modified at 1.4l (2022/05/29).** Let `\XINT:expr:toblist_b` use `#1{#2}` with regular `{` and `}` in case macro `#1` is `\protected` and things are output to external file where former `#1<#2>` would end up with catcode 12 `<` and `>`.

```
86 \def\XINT:expr:toblistwith#1#2%
87 {%
88   {\expandafter\XINT:expr:toblist_checkempty
89     \expanded{\noexpand#1!\expandafter}\detokenize{#2}^}%
90 }%
91 \def\XINT:expr:toblist_checkempty #1!#2%
92 {%
93   \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:toblist_a\fi
94   #1!#2%
95 }%
96 \catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
97 \def\XINT:expr:toblist_a #1{#2%
98 <%
99   \if{#2\xint_dothis<[\XINT:expr:toblist_a]>\fi
100   \if P#2\xint_dothis<[\XINT:expr:toblist_pol]>\fi
101   \xint_orthat\XINT:expr:toblist_b #1#2%
102 >%
103 \def\XINT:expr:toblist_pol #1!#2.{#3}}%
104 <%
105   pol([\XINT:expr:toblist_b #1!#3]^)]\XINT:expr:toblist_c #1!}%
106 >%
107 \catcode`{ 1 \catcode`} 2
108 \def\XINT:expr:toblist_b #1{%
109 \def\XINT:expr:toblist_b ##1!##2#1%
110 {%
111   \if\relax##2\relax\xintexprEmptyItem\else##1{##2}\fi\XINT:expr:toblist_c ##1!#1%
112 }}\catcode`{ 12 \catcode`} 12 \XINT:expr:toblist_b<>%
113 \def\XINT:expr:toblist_c #1{#2%
```

```

114 <%
115   \if ^#2\xint_dothis<\xint_gob_til_^>\fi
116   \if{#2\xint_dothis<, \XINT:expr:toblist_a>\fi
117   \xint_orthat<]\XINT:expr:toblist_c>#1#2%
118 >%
119 \catcode`{ 1 \catcode`} 2 \catcode`< 12 \catcode`> 12

```

### 27.8.2. Flattening nested braced contents

1.4b I hesitated whether using this technique or some variation of the method of the ListSel macros. I chose this one which I downscaled from tobrlistwith, I will revisit later. I only have a few minutes right now.

Call form is `\expanded\XINT:expr:flatten`

See `\XINT_expr_func_flat`. I hesitated with «flattened», but short names are faster parsed.

```

120 \def\XINT:expr:flatten#1%
121 {%
122   {\expanded\XINT:expr:flatten_checkempty\detokenize{#1}^}%
123 }%
124 \def\XINT:expr:flatten_checkempty #1%
125 {%
126   \if ^#1\expanded\xint_gobble_i\else\expanded\XINT:expr:flatten_a\fi
127   #1%
128 }%
129 \begingroup
130 \catcode`[ 1 \catcode`] 2 \lccode`[`{ \lccode`] `}
131 \catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
132 \lowercase<\endgroup
133 \def\XINT:expr:flatten_a {#1%
134 <%
135   \if{#1\xint_dothis<\XINT:expr:flatten_a>\fi
136   \xint_orthat\XINT:expr:flatten_b #1%
137 >%
138 \def\XINT:expr:flatten_b #1}%
139 <%
140   [#1]\XINT:expr:flatten_c }%
141 >%
142 \def\XINT:expr:flatten_c }#1%
143 <%
144   \if ^#1\xint_dothis<\xint_gobble_i>\fi
145   \if{#1\xint_dothis<\XINT:expr:flatten_a>\fi
146   \xint_orthat<\XINT:expr:flatten_c>#1%
147 >%
148 >% back to normal catcodes

```

### 27.8.3. Braced contents rendering via a TeX alignment with prettifying of leaves

#### 1.4.

Breaking change at 1.4a as helper macros were renamed and their meanings refactored: no more `\xintexpralignninnercomma` or `\xintexpralignnoutercomma` but `\xintexpralignninnersep`, etc...

At 1.4c I remove the `\protected` from `\xintexpralignend`. I had made note a year ago that it served nothing. Let's trust myself on this one (risky one year later!).

```

149 \catcode`& 4

```

```

150 \protected\def\xintexpralignbegin      {\halign\bgroup\tabskip2ex\hfil##&&##\hfil\cr}%
151 \def\xintexpralignend                  {\crcr\egroup}%
152 \protected\def\xintexpralignlinesep    {\,\cr}%
153 \protected\def\xintexpralignleftbracket {[%
154 \protected\def\xintexpralignrightbracket {]}%
155 \protected\def\xintexpralignleftsep     {\&}%
156 \protected\def\xintexpralignrightsep    {\&}%
157 \protected\def\xintexpraligninnersep    {\,&}%
158 \catcode`& 7
159 \def\XINT:expr:toalignwith#1#2%
160 {%
161     {\expandafter\XINT:expr:toalign_checkempty
162     \expanded{\noexpand#1!\expandafter}\detokenize{#2}^\expandafter}%
163     \xintexpralignend
164 }%
165 \def\XINT:expr:toalign_checkempty #1!#2%
166 {%
167     \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:toalign_a\fi
168     #1!#2%
169 }%
170 \catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
171 \def\XINT:expr:toalign_a #1{#2%
172 <%
173     \if{#2\xint_dothis<\xintexpralignleftbracket\XINT:expr:toalign_a>\fi
174     \xint_orthat<\xintexpralignleftsep\XINT:expr:toalign_b>#1#2%
175 >%
176 \def\XINT:expr:toalign_b #1!#2}%
177 <%
178     \if\relax#2\relax\xintexprEmptyItem\else#1<#2>\fi\XINT:expr:toalign_c #1!}%
179 >%
180 \def\XINT:expr:toalign_c #1{#2%
181 <%
182     \if ^#2\xint_dothis<\xint_gob_til_^\fi
183     \if {#2\xint_dothis<\xintexpraligninnersep\XINT:expr:toalign_A>\fi
184     \xint_orthat<\xintexpralignrightsep\xintexpralignrightbracket\XINT:expr:toalign_C>#1#2%
185 >%
186 \def\XINT:expr:toalign_A #1{#2%
187 <%
188     \if{#2\xint_dothis<\xintexpralignleftbracket\XINT:expr:toalign_A>\fi
189     \xint_orthat\XINT:expr:toalign_b #1#2%
190 >%
191 \def\XINT:expr:toalign_C #1{#2%
192 <%
193     \if ^#2\xint_dothis<\xint_gob_til_^\fi
194     \if {#2\xint_dothis<\xintexpralignlinesep\XINT:expr:toalign_a>\fi
195     \xint_orthat<\xintexpralignrightbracket\XINT:expr:toalign_C>#1#2%
196 >%
197 \catcode`{ 1 \catcode`} 2 \catcode`< 12 \catcode`> 12

```

#### 27.8.4. Transforming all leaves within nested braced contents

1.4. Leaves must be of catcode 12... This is currently not a constraint (or rather not a new constraint) for xintexpr because formerly anyhow all data went through csname encapsulation and



extraction via string.

In order to share code with the functioning of universal functions, which will be allowed to transform a number into an ople, the applied macro is supposed to apply one level of bracing to its output. Thus to apply this with an `xintfrac` macro such as `\xintiRound{0}` one needs first to define a wrapper which will expand it inside an added brace pair:

```
\def\foo#1{{\xintiRound{0}{#1}}}
```

As the things will expand inside expanded, propagating expansion is not an issue.

This code is used by `\xintexpr` and `\xintfloatexpr` in case of optional argument and by the «Universal functions».

Comment at 1.41: this seems to be used only at private package level, else I should modify `\XINT:expr:mapwithin_b` like I did with `\XINT:expr:toblist_b` to use regular braces in case the applied macro is `\protected` and things end up in external file.

```
198 \def\XINT:expr:mapwithin#1#2%
199 {%
200   {{\expandafter\XINT:expr:mapwithin_checkempty
201     \expanded{\noexpand#1!\expandafter}\detokenize{#2}^}}%
202 }%
203 \def\XINT:expr:mapwithin_checkempty #1#2%
204 {%
205   \if ^#2\expandafter\xint_gob_til_^\else\expandafter\XINT:expr:mapwithin_a\fi
206   #1!#2%
207 }%
208 \begingroup
209 \catcode`[ 1 \catcode`] 2 \lccode`[`{ \lccode`]` }
210 \catcode`< 1 \catcode`> 2 \catcode`{ 12 \catcode`} 12
211 \lowercase<\endgroup
212 \def\XINT:expr:mapwithin_a #1#2%
213 <%
214   \if{#2\xint_dothis<[iffalse]\fi\XINT:expr:mapwithin_a>\fi%
215   \xint_orthat\XINT:expr:mapwithin_b #1#2%
216 >%
217 \def\XINT:expr:mapwithin_b #1#2}%
218 <%
219   #1<#2>\XINT:expr:mapwithin_c #1!}%
220 >%
221 \def\XINT:expr:mapwithin_c #1#2%
222 <%
223   \if ^#2\xint_dothis<\xint_gob_til_^\fi
224   \if{#2\xint_dothis<\XINT:expr:mapwithin_a>\fi%
225   \xint_orthat<iffalse[\fi]\XINT:expr:mapwithin_c>#1#2%
226 >%
227 >% back to normal catcodes
```

## 27.9. Top level user T<sub>E</sub>X interface: `\xinteval`, `\xintfloateval`, `\xintiieval`

27.9.1	<code>\xintexpr</code> , <code>\xintiexpr</code> , <code>\xintfloatexpr</code> , <code>\xintiiepr</code> . . . . .	568
27.9.2	<code>\XINT_expr_wrap</code> , <code>\XINT_iexpr_wrap</code> , <code>\XINT_flexpr_wrap</code> . . . . .	570
27.9.3	<code>\XINTexprprint</code> , <code>\XINTiexprprint</code> , <code>\XINTiieprprint</code> , <code>\XINTflexprprint</code> . . . . .	570
27.9.4	<code>\xintthe</code> , <code>\xintthealign</code> , <code>\xinttheexpr</code> , <code>\xinttheiexpr</code> , <code>\xintthefloatexpr</code> , <code>\xint-</code> <code>theiiepr</code> . . . . .	570
27.9.5	<code>\xintbareeval</code> , <code>\xintbarefloateval</code> , <code>\xintbareieval</code> . . . . .	571
27.9.6	<code>\xintthebareeval</code> , <code>\xintthebarefloateval</code> , <code>\xintthebareieval</code> . . . . .	571



27.9.7	<code>\xinteval</code> , <code>\xintieval</code> , <code>\xintfloateval</code> , <code>\xintiieval</code> . . . . .	572
27.9.8	<code>\xintboolexpr</code> , <code>\XINT_boolexpr_print</code> , <code>\xinttheboolexpr</code> . . . . .	573
27.9.9	<code>\xintifboolexpr</code> , <code>\xintifboolfloatexpr</code> , <code>\xintifbooliiexpr</code> . . . . .	573
27.9.10	<code>\xintifsgnexpr</code> , <code>\xintifsgnfloatexpr</code> , <code>\xintifsgniiexpr</code> . . . . .	574
27.9.11	<code>\xint_FracToSci_x</code> . . . . .	574
27.9.12	Small bits we have to put somewhere . . . . .	575
	<code>\xintthecoords</code> . . . . .	575
	<code>\xintthespaceparated</code> . . . . .	576

### 27.9.1. `\xintexpr`, `\xintiexpr`, `\xintfloatexpr`, `\xintiexpr`

`\xintiexpr` and `\xintfloatexpr` have an optional argument since 1.1.

ATTENTION! 1.3d renamed `\xinteval` to `\xintexpr` etc...

Usage of `\xintiRound{0}` for `\xintiexpr` without optional [D] means that `\xintiexpr` ... `\relax` wrapper can be used to insert rounded-to-integers values in `\xintiexpr` context: no post-fix [0] which would break it.

1.4a add support for the optional argument [D] for `\xintiexpr` being negative D, with same meaning as the 1.4a modified `\xintRound` from `xintfrac.sty`.

`\xintiexpr` mechanism was refactored at 1.4e so that rounding due to [D] optional argument uses raw format, not fixed point format on output, delegating fixed point conversion to an `\XINTiexprprint` now separated from `\XINTexprprint`.

In case of negative [D], `\xintiexpr` [D]...`\relax` internally has the [0] post-fix so it can not be inserted as sub-expression in `\xintiexpr` without a `num()` or `\xintiexpr` ...`\relax` (extra) wrapper.

Modified at 1.4o (2025/09/06).

While preparing 1.4n I had made changes to tame Babel active characters via a `\csname` trick. But at last minute I thought I had a way requiring not so many ``top level'' changes, but this was a grave logic error on my part, due to having a bit forgotten the way things are done here, and due also to external circumstances I released too hastily a 1.4n which turned out to be faulty (except for `\xintiieval`). Any test other than simply evaluating 3! with ! Babel active would have revealed it but I was planning to add test units after (!) release (by the way one has to be careful to do such tests after a `\noindent` because the French active ! in vertical mode does things not breaking the parser. Anyway, here I do it right.

So this is why for example `\xintexpr` has a `\csname` construct.

```

228 \def\xintexpr      {\romannumeral0\xintexpr    }%
229 \def\xintiexpr     {\romannumeral0\xintiexpr    }%
230 \def\xintfloatexpr {\romannumeral0\xintfloatexpr}%
231 \def\xintiexpr     {\romannumeral0\xintiexpr    }%
232 \def\xintexpr      {\csname XINT_expr_wrap\expandafter\endcsname
233                    \romannumeral0\xintbareeval }%
234 \def\xintiexpr     {\csname XINT_iiexpr_wrap\expandafter\endcsname
235                    \romannumeral0\xintbareiieval }%
236 \def\xintiexpr     #1%
237 {%
238   \ifx [#1\expandafter\XINT_iexpr_withopt\else\expandafter\XINT_iexpr_noopt
239   \fi #1%
240 }%
241 \def\XINT_iexpr_noopt
242 {%
243   \csname XINT_iexpr_iiround\expandafter\endcsname\romannumeral0\xintbareeval
244 }%
245 \def\XINT_iexpr_iiround

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

246 {%
247   \expandafter\XINT_expr_wrap
248   \expanded
249   \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTiRoundzero_braced}%
250 }%
251 \def\XINTiRoundzero_braced#1{{\xintiRound{0}{#1}}}%
252 \def\XINT_iexpr_withopt [#1]%
253 {%
254   \csname XINT_iexpr_round\expandafter\endcsname
255   \the\numexpr \xint_zapspace #1 \xint_gobble_i\expandafter.%
256   \romannumeral0\xintbareeval
257 }%
258 \def\XINT_iexpr_round #1.%
259 {%
260   \ifnum#1=\xint_c_\xint_dothis{\XINT_iexpr_iiround}\fi
261   \xint_orthat{\XINT_iexpr_round_a #1.}%
262 }%
263 \def\XINT_iexpr_round_a #1.%
264 {%
265   \expandafter\XINT_iexpr_wrap
266   \expanded
267   \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTiRound_braced{#1}}%
268 }%
269 \def\XINTiRound_braced#1#2%
270   {{\xintiRound{#1}{#2}}[\the\numexpr\ifnum#1<\xint_c_i0\else-#1\fi]]}%
271 \def\xintfloatexpro #1%
272 {%
273   \ifx [#1\expandafter\XINT_flexpr_withopt\else\expandafter\XINT_flexpr_noopt
274   \fi #1%
275 }%
276 \def\XINT_flexpr_noopt
277 {%
278   \csname XINT_flexpr_wrap\expandafter\endcsname
279   \the\numexpr\XINTdigits\expandafter.%
280   \romannumeral0\xintbarefloateval
281 }%
282 \def\XINT_flexpr_withopt [#1]%
283 {%
284   \csname XINT_flexpr_withopt_a\expandafter\endcsname
285   \the\numexpr\xint_zapspace #1 \xint_gobble_i\expandafter.%
286   \romannumeral0\xintbarefloateval
287 }%
288 \def\XINT_flexpr_withopt_a #1#2.%
289 {%
290   \expandafter\XINT_flexpr_withopt_b\the\numexpr\if#1-\XINTdigits\fi#1#2.%
291 }%
292 \def\XINT_flexpr_withopt_b #1.%
293 {%
294   \expandafter\XINT_flexpr_wrap
295   \the\numexpr#1\expandafter.%
296   \expanded
297   \XINT:NEhook:x:mapwithin\XINT:expr:mapwithin{\XINTinFloat_braced[#1]}%

```

```

298 }%
299 \def\XINTinFloat_braced[#1]#2{{\XINTinFloat[#1]{#2}}}%

```

### 27.9.2. \XINT\_expr\_wrap, \XINT\_iexpr\_wrap, \XINT\_flexpr\_wrap

1.3e removes some leading space tokens which served nothing. There is no `\XINT_iexpr_wrap`, because `\XINT_expr_wrap` is used directly.

1.4e has `\XINT_iexpr_wrap` separated from `\XINT_expr_wrap`, thus simplifying internal matters as output printer for `\xintexpr` will not have to handle fixed point input but only extended-raw type input (i.e. A, A/B, A[N] or A/B[N]).

```

300 \def\XINT_expr_wrap    {\XINTfstop\XINTexprprint.}%
301 \def\XINT_iexpr_wrap  {\XINTfstop\XINTiexprprint.}%
302 \def\XINT_iiexpr_wrap {\XINTfstop\XINTiiexprprint.}%
303 \def\XINT_flexpr_wrap {\XINTfstop\XINTflexprprint}%

```

### 27.9.3. \XINTexprprint, \XINTiexprprint, \XINTiiexprprint, \XINTflexprprint

Modified at 1.4 (2020/01/31). Requires `\expanded`.

Modified at 1.4e (2021/05/05). 1.4e has a breaking change of `\XINTflexprprint` and `\xintfloatexprPrintOne` which now requires `\xintfloatexprPrintOne[D]{x}` syntax, with first argument in brackets.

Modified at 1.4l (2022/05/29). The code does `\let\xintexprPrintOne\xint_FracToSci_x` but the latter is not yet defined so this is delayed until `\xint_FracToSci_x` definition.

Modified at 1.4m (2022/06/10). `\xintboolexprPrintOne` outputs `true` or `false`, not `True` or `False`. By the way (undocumented) all four keywords `true`, `false`, `True`, `False` are recognized as genuine variables since 1.4i.

```

304 \protected\def\XINTexprprint.%
305   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintexprPrintOne}%
306 \protected\def\XINTiexprprint.%
307   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiexprPrintOne}%
308 \let\xintiexprPrintOne\xintDecToString
309 \def\xintexprEmptyItem{[]}%
310 \protected\def\XINTiiexprprint.%
311   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiiexprPrintOne}%
312 \let\xintiiexprPrintOne\xint_firstofone
313 \protected\def\XINTflexprprint #1.%
314   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith{\xintfloatexprPrintOne[#1]}}%
315 \let\xintfloatexprPrintOne\xintPFloat_wopt
316 \protected\def\XINTboolexprprint.%
317   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintboolexprPrintOne}%
318 \def\xintboolexprPrintOne#1{\xintiifNotZero{#1}{true}{false}}%

```

### 27.9.4. \xintthe, \xintthealign, \xinttheexpr, \xinttheiexpr, \xintthefloatexpr, \xinttheiexpr

The reason why `\xinttheiexpr` et `\xintthefloatexpr` are handled differently is that they admit an optional argument which acts via a custom «printing» stage.

We exploit here that `\expanded` expands forward until finding an implicit or explicit brace, and that this expansion overrules `\protected` macros, forcing them to expand, similarly as `\romannumer` expands `\protected` macros, and contrarily to what happens \*within\* the actual `\expanded` scope. I discovered this fact by testing (with pdfTeX) and I don't know where this is documented apart from the source code of the relevant engines. This is useful to us because there are contexts where we will want to apply a complete expansion before printing, but in purely numerical context this is not needed (if I converted correctly after dropping at 1.4 the `\csname` governed expansions;

however I rely at various places on the fact that the xint macros are f-expandable, so I have tried to not use zillions of expanded all over the place), hence it is not needed to add the expansion overhead by default. But the `\expanded` here will allow `\xintNewExpr` to create macro with suitable modification or the printing step, via some hook rather than having to duplicate all macros here with some new «NE» meaning (aliasing does not work or causes big issues due to desire to support `\xinteval` also in «NE» context as sub-constituent. The `\XINT:NEhook:x:toblist` is something else which serves to achieve this support of \*sub\* `\xinteval`, it serves nothing for the actual produced macros. For `\xintdeffunc`, things are simpler, but still we support the [N] optional argument of `\xintiexpr` and `\xintfloatexpr`, which required some work...

The `\expanded` upfront ensures `\xintthe` mechanism does expand completely in two steps.

The fact that `\xintthe` grabs a #1 is legacy and I am not sure why. I vaguely remember thinking the overhead was minimal because #1 will simply be one token such as `\xintexpr` and that it could potentially be useful. Of course `\xintthealign` imitates `\xintthe` in that matter.

**Modified at 1.40 (2025/09/06).** Again here some `\csname`'s to handle babel-active characters.

```

319 \def\xintthe      #1{\expanded\expandafter\xint_gobble_i\romannumeral`&&@#1}%
320 \def\xintthealign #1{\expanded\expandafter\xintexpralignbegin
321   \expanded\expandafter\XINT:expr:toalignwith
322   \romannumeral0\expandafter\expandafter\expandafter\expandafter
323   \expandafter\expandafter\expandafter\xint_gob_andstop_ii
324   \expandafter\xint_gobble_i\romannumeral`&&@#1}%
325 \def\xinttheexpr
326   {\expanded\csname XINTexprprint\expandafter\endcsname
327   \expandafter.\romannumeral0\xintbareeval}%
328 \def\xinttheiexpr
329   {\expanded\csname XINTiexprprint\expandafter\endcsname
330   \expandafter.\romannumeral0\xintbareiieval}%

```

No need at 1.40 of `\csname`'s for Babel here, as `\xintiexpr` and `\xintfloatexpr` are already taken care off.

```

331 \def\xinttheiexpr
332   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintiexpr}%
333 \def\xintthefloatexpr
334   {\expanded\expandafter\xint_gobble_i\romannumeral`&&@\xintfloatexpr}%

```

#### 27.9.5. `\xintbareeval`, `\xintbarefloateval`, `\xintbareiieval`

**Modified at 1.4 (2020/01/31).** one expansion step added via `\XINT_expr_start`, `\XINT_iexpr_start`, `\XINT_flexpr_start`. Trigger is expected to be via `\romannumeral`^^@` or `\romannumeral0`.

For the benefit of those who like the author while finishing 1.4n looked in vain for where the ```_start` macros were defined, this documentation was updated so that one can find the location via clicking on their names in the previous paragraph. Spoiler: they are in [subsection 27.16](#). This was hidden because the three are defined in an `\xintFor` in one-go.

```

335 \def\xintbareeval      {\XINT_expr_start }%
336 \def\xintbarefloateval{\XINT_flexpr_start}%
337 \def\xintbareiieval    {\XINT_iexpr_start}%

```

#### 27.9.6. `\xintthebareeval`, `\xintthebarefloateval`, `\xintthebareiieval`

For matters of `\XINT_NewFunc`

**Modified at 1.40 (2025/09/06).** The matter of Babel active here too, although they are used for `\xintdeffunc` and `\xintNewExpr` which sanitize catcodes. But I already edited the user documentation to explain the case of contexts where changing catcodes is impossible, and would be too time-costly to re-edit that. So let's add some overhead.

```

338 \def\xINT_expr_unlock      {\expandafter\xint_firstofone\romannumeral`&&@}%
339 \def\xintthebareeval
340   {\romannumeral0\csname xint_stop_atfirstofone\expandafter\endcsname
341     \romannumeral0\xintbareeval}%
342 \def\xintthebareiieval
343   {\romannumeral0\csname xint_stop_atfirstofone\expandafter\endcsname
344     \romannumeral0\xintbareiieval}%
345 \def\xintthebarefloateval
346   {\romannumeral0\csname xint_stop_atfirstofone\expandafter\endcsname
347     \romannumeral0\xintbarefloateval}%
348 \def\xintthebareroundedfloateval
349 {%
350   \romannumeral0\csname xintthebareroundedfloateval_a\expandafter\endcsname
351   \romannumeral0\xintbarefloateval
352 }%
353 \def\xintthebareroundedfloateval_a
354 {%
355   \expandafter\xint_stop_atfirstofone
356   \expanded\xINT:NEhook:x:mapwithin\xINT:expr:mapwithin{\XINTinFloatSdigits_braced}%
357 }%
358 \def\xINTinFloatSdigits_braced#1{{\XINTinFloatS[\XINTdigits]{#1}}}%

```

### 27.9.7. \xinteval, \xintieval, \xintfloateval, \xintiieval

Refactored at 1.4.

The `\expanded` upfront ensures `\xinteval` still expands completely in two steps. No `\romannumeral` trigger here, in relation to the fact that `\XINTexprprint` is not f-expandable, only e-expandable.

(and attention that `\xintexpr\relax` is now legal, and an empty ople can be produced in output also from `\xintexpr [17][1]\relax` for example)

Modified at 1.4k (2022/05/18).

The `\xintieval` and `\xintfloateval` optional bracketed argument can now be located outside the braces... took me years to finally make the step towards LaTeX users expectations for the interface.

Modified at 1.4o (2025/09/06). Some `\csname`'s for the Babel-active feature which was botched at 1.4n.

```

359 \def\xinteval #1%
360   {\expanded\csname XINTexprprint\expandafter\endcsname
361     \expandafter.\romannumeral0\xintbareeval#1\relax}%
362 \def\xintieval{\expanded\expandafter\xint_ieval_chkopt\string}%
363 \def\xint_ieval_chkopt #1%
364 {%
365   \ifx [#1\expandafter\xint_ieval_opt
366     \else\expandafter\xint_ieval_noopt
367     \fi #1%
368 }%
369 \def\xint_ieval_opt [#1]#2%
370   {\expandafter\xint_gobble_i\romannumeral`&&\xintiexpr[#1]#2\relax}%
371 \def\xint_ieval_noopt #1{\expandafter\xint_ieval\expandafter{\iffalse}\fi}%
372 \def\xint_ieval#1%
373   {\expandafter\xint_gobble_i\romannumeral`&&\xintiexpr#1\relax}%
374 \def\xintfloateval {\expanded\expandafter\xint_floateval_chkopt\string}%
375 \def\xint_floateval_chkopt #1%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

```
376 {%
377   \ifx [#1\expandafter\xint_floateval_opt
378     \else\expandafter\xint_floateval_noopt
379   \fi #1%
380 }%
381 \def\xint_floateval_opt [#1]#2%
382   {\expandafter\xint_gobble_i\romannumeral`&&\xintfloatexpr[#1]#2\relax}%
383 \def\xint_floateval_noopt #1{\expandafter\xint_floateval\expandafter{\iffalse}\fi}%
384 \def\xint_floateval#1%
385   {\expandafter\xint_gobble_i\romannumeral`&&\xintfloatexpr#1\relax}%
```

Modified at 1.4n (2025/09/05). `\xintiieval` admits optional arguments `[h]`, `[o]` and `[b]`. The way this is implemented tames Babel active characters.

```
386 \def\xintiieval #1#{\expanded\xintiieval_a{#1}}%
387 \def\xintiieval_a#1#2%
388 {%
389   \csname XINTiexprprint\xint_zapspace #1 \xint_gobble_i
390   \expandafter\endcsname
391   \expandafter.\romannumeral0\xintbareieval#2\relax
392 }%
393 \expandafter\def\csname XINTiexprprint[h]\endcsname.%
394   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiexprPrintOneHex}%
395 \expandafter\def\csname XINTiexprprint[o]\endcsname.%
396   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiexprPrintOneOct}%
397 \expandafter\def\csname XINTiexprprint[b]\endcsname.%
398   {\XINT:NEhook:x:toblist\XINT:expr:toblistwith\xintiexprPrintOneBin}%
399 \expandafter\let\csname XINTiexprprint[]\endcsname\XINTiexprprint
400 \let\xintiexprPrintOneHex\xintDecToHex
401 \let\xintiexprPrintOneOct\xintDecToOct
402 \let\xintiexprPrintOneBin\xintDecToBin
```

### 27.9.8. `\xintboolexpr`, `\XINT_boolexpr_print`, `\xinttheboolexpr`

ATTENTION! 1.3d renamed `\xinteval` to `\xintexpro` etc...

Attention, the conversion to 1 or 0 is done only by the print macro. Perhaps I should force it also inside raw result.

`\changed{1.4o}` Some `|\csname|` for `|\xinttheboolexpr|` for Babel.

```
403 \def\xintboolexpr
404 {%
405   \romannumeral0\expandafter\XINT_boolexpr_done\romannumeral0\xintexpro
406 }%
407 \def\XINT_boolexpr_done #1.{\XINTfstop\XINTboolexprprint.}%
408 \def\xinttheboolexpr
409 {%
410   \expanded\csname XINTboolexprprint\expandafter\endcsname
411   \expandafter.\romannumeral0\xintbareeval
412 }%
```

### 27.9.9. `\xintifboolexpr`, `\xintifboolfloatexpr`, `\xintifboolliexpr`

They do not accept comma separated expressions input.

```
413 \def\xintifboolexpr #1{\romannumeral0\xintiifnotzero {\xinttheexpr #1\relax}}%
414 \def\xintifboolfloatexpr #1{\romannumeral0\xintiifnotzero {\xintthefloatexpr #1\relax}}%
```



```
415 \def\xintifboolliexpr #1{\romannumeral0\xintiiifnotzero {\xinttheiexpr #1\relax}}%
```

### 27.9.10. \xintifsgnexpr, \xintifsgnfloatexpr, \xintifsgniexpr

Modified at 1.3d (2019/01/06). They do not accept comma separated expressions.

```
416 \def\xintifsgnexpr #1{\romannumeral0\xintiiifsgn {\xinttheexpr #1\relax}}%
417 \def\xintifsgnfloatexpr #1{\romannumeral0\xintiiifsgn {\xintthefloatexpr #1\relax}}%
418 \def\xintifsgniexpr #1{\romannumeral0\xintiiifsgn {\xinttheiexpr #1\relax}}%
```

### 27.9.11. \xint\_FracToSci\_x

Added at 1.4 (2020/01/31). Under the name of [\xintFracToSci](#) and defined in [xintfrac](#).

Modified at 1.4e (2021/05/05). Refactored and much simplified

It only needs to be x-expandable, and indeed the implementation here is only x-expandable.

Finally for 1.4e release I modify. This is breaking change for all [\xinteval](#) output in case of scientific notation: it will not be with an integer mantissa, but with regular scientific notation, using the same rules as [\xintPFloat](#).

Of course no float rounding! Also, as [0] will always or almost always be present from an [\xinteval](#), we want then to use integer not scientific notation. But expression contained decimal fixed point input, or uses scientific functions, then probably the N will not be zero and this will trigger usage of scientific notation in output.

Implementing these changes sort of ruin our previous efforts to minimize grabbing the argument, but well. So the rules now are

Input: A, A/B, A[N], A/B[N]

Output: A, A/B, A if N=0, A/B if N≠0

If N is not zero, scientific notation like [\xintPFloat](#), i.e. behaviour like [\xintfloateval](#) apart from the rounding to significands of width Digits. At 1.4k, trimming of zeros from A is always done, i.e. the [\xintPFloatMinTrimmed](#) is ignored to keep behaviour unchanged. Trailing zeros of B are kept as is.

The zero gives 0, except in A[N] and A/B[N] cases, it may give 0.0

Modified at 1.4k (2022/05/18). Moved from [xintfrac](#) to [xintexpr](#).

Modified at 1.4l (2022/05/29). Renamed to [\xint\\_FracToSci\\_x](#) to make it private and provide in [xintfrac](#) another [\xintFracToSci](#) with same output but which behaves like other macros there: f-expandable and accepting the whole range of inputs accepted by the [xintfrac](#) public macros.

The private x-expandable macro here will have an empty output for an empty input but is never used for an empty input (see [\xintexprEmptyItem](#)).

```
419 \def\xint_FracToSci_x #1{\expandafter\xint_FracToSci_x\romannumeral`&&@#1/\W[\R}%
420 \def\xint_FracToSci_x #1/#2#3[#4%
421 {%
422   \xint_gob_til_W #2\xint_FracToSci_x_noslash\W
423   \xint_gob_til_R #4\xint_FracToSci_x_slash_noN\R
424   \xint_FracToSci_x_slash_N #1/#2#3[#4%
425 }%
426 \def\xint_FracToSci_x_noslash#1\xint_FracToSci_x_slash_N #2[#3%
427 {%
428   \xint_gob_til_R #3\xint_FracToSci_x_noslash_noN\R
429   \xint_FracToSci_x_noslash_N #2[#3%
430 }%
431 \def\xint_FracToSci_x_noslash_noN\R\xint_FracToSci_x_noslash_N #1/\W[\R{#1}%
432 \def\xint_FracToSci_x_noslash_N #1[#2]/\W[\R%
433 {%
```

```

434 \ifnum#2=\xint_c_ #1\else
435 \romannumeral0\expandafter\XINT_pfloat_a_fork\romannumeral0\xintrez{#1[#2]}%
436 \fi
437 }%
438 \def\XINT_FracToSci_x_slash_noN\R\XINT_FracToSci_x_slash_N #1#2/#3/\W[\R%
439 {%
440 #1\xint_gob_til_zero#1\expandafter\iffalse\xint_gobble_ii0\iftrue
441 #2\if\XINT_isOne{#3}1\else/#3\fi\fi
442 }%
443 \def\XINT_FracToSci_x_slash_N #1#2/#3[#4]/\W[\R%
444 {%
445 \ifnum#4=\xint_c_ #1#2\else
446 \romannumeral0\expandafter\XINT_pfloat_a_fork\romannumeral0\xintrez{#1#2[#4]}%
447 \fi
448 \if\XINT_isOne{#3}1\else\if#10\else/#3\fi\fi
449 }%
450 \let\xintexprPrintOne\xint_FracToSci_x

```

### 27.9.12. Small bits we have to put somewhere

Some renaming and modifications here with release 1.2 to switch from using chains of `\romannumeral-`0` in order to gather numbers, possibly hexadecimal, to using a `\csname` governed expansion. In this way no more limit at 5000 digits, and besides this is a logical move because the `\xintexpr` parser is already based on `\csname...\endcsname` storage of numbers as one token.

The limitation at 5000 digits didn't worry me too much because it was not very realistic to launch computations with thousands of digits... such computations are still slow with 1.2 but less so now. Chains or `\romannumeral` are still used for the gathering of function names and other stuff which I have half-forgotten because the parser does many things.

In the earlier versions we used the `lockscan` macro after a chain of `\romannumeral-`0` had ended gathering digits; this uses has been replaced by direct processing inside a `\csname...\endcsname` and the macro is kept only for matters of dummy variables.

Currently, the parsing of hexadecimal numbers needs two nested `\csname...\endcsname`, first to gather the letters (possibly with a hexadecimal fractional part), and in a second stage to apply `\xintHexToDec` to do the actual conversion. This should be faster than updating on the fly the number (which would be hard for the fraction part...).

```

451 \def\XINT_embrace#1{{#1}}%
452 \def\xint_gob_til_! #1!{% ! with catcode 11
453 \def\xintError:noopening
454 {%
455 \XINT_expandableerror{Extra ). This is serious and prospects are bleak.}%
456 }%

```

**\xintthecoords** 1.1 Wraps up an even number of comma separated items into pairs of TikZ coordinates; for use in the following way:

```
coordinates {\xintthecoords\xintfloatexpr ... \relax}
```

The crazyness with the `\csname` and `unlock` is due to TikZ somewhat STRANGE control of the TOTAL number of expansions which should not exceed the very low value of 100 !! As we implemented `\XINT_thecoords_b` in an "inline" style for efficiency, we need to hide its expansions.

Not to be used as `\xintthecoords\xintthefloatexpr`, only as `\xintthecoords\xintfloatexpr` (or `\xintiexpr` etc...). Perhaps `\xintthecoords` could make an extra check, but one should not accustom users to too loose requirements!

```

457 \def\xintthecoords#1%

```



```

458 {\romannumeral`&&@\expandafter\XINT_thecoords_a\romannumeral0#1}%
459 \def\XINT_thecoords_a #1#2.#3% #2.=\XINTfloatprint<digits>. etc...
460 {\expanded{\expandafter\XINT_thecoords_b\expanded#2.{#3},!,!,^}%
461 \def\XINT_thecoords_b #1#2,#3#4,%
462 {\xint_gob_til_! #3\XINT_thecoords_c ! (#1#2, #3#4)\XINT_thecoords_b }%
463 \def\XINT_thecoords_c #1^{}%

```

**\xintthespaceseparated** 1.4a This is a utility macro which was distributed previously separately for usage with PSTricks **\listplot**

```

464 \def\xintthespaceseparated#1%
465 {\expanded\expandafter\xintthespaceseparated_a\romannumeral0#1}%
466 \def\xintthespaceseparated_a #1#2.#3%
467 {\{\expandafter\xintthespaceseparated_b\expanded#2.{#3},!,!,!,!,!,!,!,!,^}%
468 \def\xintthespaceseparated_b #1,#2,#3,#4,#5,#6,#7,#8,#9,%
469 {\xint_gob_til_! #9\xintthespaceseparated_c !%
470 #1#2#3#4#5#6#7#8#9%
471 \xintthespaceseparated_b}%

```

1.4c I add a space here to stop the **\romannumeral`&&@** in case of empty input. But this space induces an extra un-needed space token after 9, 18, 27,... items before the last group of less than 9 items.

Fix (at 1.4h) is simple because I already use **\expanded** anyhow: I don't need at all the **\romannumeral`&&@** which was first in **\xintthespaceseparated**, let's move the first **\expanded** which was in **\xintthespaceseparated\_a** to **\xintthespaceseparated**, and remove the extra space here in **\_c**.

(alternative would have been to put the space after #1 and accept a systematic trailing space, at least it is more aesthetic).

Again, I did have a test file, but it was not incorporated in my test suite, so I discovered the problem accidentally by compiling all files in an archive.

```

472 \def\xintthespaceseparated_c !#1!#2^{#1}%

```

## 27.10. Hooks into the numeric parser for usage by the **\xintdeffunc** symbolic parser

This is new with 1.3 and considerably refactored at 1.4. See «Mysterious stuff».

```

473 \let\XINT:NEhook:f:one:from:one\expandafter
474 \let\XINT:NEhook:f:one:from:one:direct\empty
475 \let\XINT:NEhook:f:one:from:two\expandafter
476 \let\XINT:NEhook:f:one:from:two:direct\empty
477 \let\XINT:NEhook:x:one:from:two\empty
478 \let\XINT:NEhook:f:one:and:opt:direct \empty
479 \let\XINT:NEhook:f:tacitzeroifone:direct \empty
480 \let\XINT:NEhook:f:iitacitzeroifone:direct \empty
481 \let\XINT:NEhook:x:select:obey\empty
482 \let\XINT:NEhook:x:listsels\empty
483 \let\XINT:NEhook:f:reverse\empty

```

At 1.4 it was **\def\XINT:NEhook:f:from:delim:u #1#2^{#1#2^}** which was trick to allow automatic unpacking of a nutple argument to multi-arguments functions such as **gcd()** or **max()**. But this sacrificed the usage with a single numeric argument.

**Modified at 1.4i (2021/06/11).** More sophisticated code to check if the argument ople was actually a single number. Notice that this forces numeric types to actually use catcode 12 tokens, and **polexpr** diverges a bit using **P**, but actually always testing with **\if not \ifx**.

This is used by **gcd()**, **lcm()**, **max()**, **min()**, **`+`()**, **`\*`()**, **all()**, **any()**, **xor()**.

The nil and None will give the same result due to the initial brace stripping done by `\XINT:NEhook:f:from:delim:u` (there was even a prior brace stripping to provide the #2 which is empty here for the nil and {} for the None).

```

484 \def\XINT:NEhook:f:from:delim:u #1#2^%
485 {%
486   \expandafter\XINT_foof_checkifnumber\expandafter#1\string#2^%
487 }%
488 \def\XINT_foof_checkifnumber#1#2%
489 {%
490   \expandafter#1%
491   \romannumeral0\expanded{\if ^#2^else
492     \if\bggroup#2\noexpand\XINT_foof_no\else
493     \noexpand\XINT_foof_yes#2\fi\fi}%
494 }%
495 \def\XINT_foof_yes#1^{{#1}^}%
496 \def\XINT_foof_no{\expandafter{\iffalse}\fi}%

```

**Modified at 1.4i (2021/06/11).** Same changes as for the other multiple arguments functions, making them again usable with a single numeric input.

Was at 1.4 `\def\XINT:NEhook:f:noeval:from:braced:u#1#2^{#1{#2}}` which is not compatible with a single numeric input.

Used by `len()`, `first()`, `last()` but it is a potential implementation bug that the three share this as the location where expansion takes places is one level deeper for the support macro of `len()`.

The None is here handled as nil, i.e. it is unpacked, which is fine as the documentations says nuples are unpacked.

```

497 \def\XINT:NEhook:f:LFL #1{\expandafter#1\expandafter}%
498 \def\XINT:NEhook:r:check #1^%
499 {%
500   \expandafter\XINT:NEhook:r:check_a\string#1^%
501 }%
502 \let\XINT:NEsaved:r:check \XINT:NEhook:r:check
503 \def\XINT:NEhook:r:check_a #1%
504 {%
505   \if ^#1\xint_dothis\xint_c_\fi
506   \if\bggroup#1\xint_dothis\XINT:NEhook:r:check_no\fi
507   \xint_orthat{\XINT:NEhook:r:check_yes#1}%
508 }%
509 \def\XINT:NEhook:r:check_no
510 {%
511   \expandafter\XINT:NEhook:r:check_no_b
512   \expandafter\xint_c_\expandafter{\iffalse}\fi
513 }%
514 \def\XINT:NEhook:r:check_no_b#1^{#1}%
515 \def\XINT:NEhook:r:check_yes#1^{\xint_c_{#1}}%
516 \let\XINT:NEhook:branch\expandafter
517 \let\XINT:NEhook:seqx\empty
518 \let\XINT:NEhook:iter\expandafter
519 \let\XINT:NEhook:opx\empty
520 \let\XINT:NEhook:rseq\expandafter
521 \let\XINT:NEhook:iterr\expandafter
522 \let\XINT:NEhook:rrseq\expandafter
523 \let\XINT:NEhook:x:toblist\empty
524 \let\XINT:NEhook:x:mapwithin\empty

```

525 \let\XINT:NEhook:x:ndmapx\empty

## 27.11. \XINT\_expr\_getnext: fetch some value then an operator and present them to last waiter with the found operator precedence, then the operator, then the value

Big change in 1.1, no attempt to detect braced stuff anymore as the [N] notation is implemented otherwise. Now, braces should not be used at all; one level removed, then `\romannumeral`0` expansion.

Refactored at 1.4 to put expansion of `\XINT_expr_gettop` after the fetched number, thus avoiding it to have to fetch it (which could happen then multiple times, it was not really important when it was only one token in pre-1.4 `xintexpr`).

Allow `\xintexpr\relax` at 1.4.

Refactored at 1.4 the articulation `\XINT_expr_getnext/XINT_expr_func/XINT_expr_gettop`. For some legacy reason the first token picked by `getnext` was soon turned to catcode 12 The next ones after the first were not a priori stringified but the first token was, and this made allowing things such as `\xintexpr\relax`, `\xintexpr,,\relax`, `[]`, `1+()`, `[:]` etc... complicated and requiring each time specific measures.

The `\expandafter` chain in `\XINT_expr_put_op_first` is an overhead related to an 1.4 attempt, the "varvalue" mechanism. I.e.: expansion of `\XINT_expr_var_foo` is `{\XINT_expr_varvalue_foo}` and then for example `\XINT_expr_varvalue_foo` expands to `{4/1[0]}`. The mechanism was originally conceived to have only one token with idea its makes things faster. But the `xintfrac` macros break with syntax such as `\xintMul\foo\bar` and `\foo` expansion giving braces. So at 1.4c I added here these `\expandafter`, but this is REALLY not satisfactory because the `\expandafter` are needed it seems only for this variable "varvalue" mechanism.

See also the discussion of `\XINT_expr_op__` which distinguishes variables from functions.

After a 1.4g refactoring it would be possible to drop here the `\expandafter` if the `\XINT_expr_var__foo` macro was defined to f-expand to {actual expanded value (as ople)} for example explicit `{{3}}`. I have to balance the relative weights of doing always the `\expandafter` but they are needed only for the case the value was encapsulated in a variable, and of never doing the `\expandafter` and ensure f-expansion of the `_var_foo` gives explicit value (now that the refactoring let it be f-expanded, and the case of fake variables omit and abort in particular was safely separated instead of being treated like other and imposing restrictions on general variable handling), and then there is the overhead of possibly moving around many digits in the #1 of `\XINT_expr_put_op_first`.

```
526 \def\XINT_expr_getnext #1%
527 {%
528   \expandafter\XINT_expr_put_op_first\romannumeral`&&@%
529   \expandafter\XINT_expr_getnext_a\romannumeral`&&@#1%
530 }%
531 \def\XINT_expr_put_op_first #1#2#3{\expandafter#2\expandafter#3\expandafter{#1}}%
532 \def\XINT_expr_getnext_a #1%
533 {%
534   \ifx\relax #1\xint_dothis\XINT_expr_foundprematureend\fi
535   \ifx\XINTfstop#1\xint_dothis\XINT_expr_subexpr\fi
536   \ifcat\relax#1\xint_dothis\XINT_expr_countetc\fi
537   \xint_orthat{}\XINT_expr_getnextfork #1%
538 }%
539 \def\XINT_expr_foundprematureend\XINT_expr_getnextfork #1{{}\xint_c\relax}%
540 \def\XINT_expr_subexpr #1.#2%
541 {%
542   \expanded{\xint_noexpd{{#2}}\expandafter}\romannumeral`&&\XINT_expr_getop
543 }%
```

The "fetch as number" must be avoided for those cases where `\number` can not be hoped to need only one token. Hence the list of exceptions.

**Modified at 1.2 (2015/10/10).** Add `\ht`, `\dp`, `\wd` and the eTeX font related primitives.

**Modified at 1.4 (2020/01/31).** Refactor for readability to avoid huge amounts of `\fi`'s.

**Modified at 1.4g (2021/05/25).** Check also for `\catcode` so that `\xinteval{\catcode`@}` does not crash. But later I observed that `\fpeval{\catcode`@}` does crash, so in retrospect I wonder if I should have added the overhead.

**Modified at 1.4n (2025/09/05).** Added `\_catcode` to the list for OpTeX. This means though that an undefined token with other engines will be expanded via `\number...`

```

544 \def\xINT_expr_countetc\xINT_expr_getnextfork#1%
545 {%
546   \if0\ifx\count#1\fi
547   \ifx\numexpr#1\fi
548   \ifx\catcode#1\fi
549   \ifx\dimen#1\fi
550   \ifx\dimexpr#1\fi
551   \ifx\skip#1\fi
552   \ifx\glueexpr#1\fi
553   \ifx\fontdimen#1\fi
554   \ifx\ht#1\fi
555   \ifx\dp#1\fi
556   \ifx\wd#1\fi
557   \ifx\fontcharht#1\fi
558   \ifx\fontcharwd#1\fi
559   \ifx\fontchardp#1\fi
560   \ifx\fontcharic#1\fi
561   \ifx\_catcode#1\fi
562   0\expandafter\xINT_expr_fetch_as_number\fi
563 \expandafter\xINT_expr_getnext_a\number #1%
564 }%
565 \def\xINT_expr_fetch_as_number
566   \expandafter\xINT_expr_getnext_a\number #1%
567 {%
568   \expanded{{{ \number#1}}\expandafter}\romannumeral`&&\xINT_expr_getop
569 }%
```

This is the key initial dispatch component. It has been refactored at 1.4g to give priority to identifying letter and digit tokens first. It thus combines former `\XINT_expr_getnextfork`, `\XINT_expr_scan_nbr_or_func` and `\XINT_expr_scanfunc`. A branch of the latter having become `\XINT_expr_startfunc`. The handling of non-catcode 11 underscore `_` has changed: it is now skipped completely like the `+`. Formerly it would cause an infinite loop because it triggered first insertion of a nil variable, (being confused with a possible operator at a location where one looks for a value), then tacit multiplication (being now interpreted as starting some name), and then it came back to `getnextfork` creating loop. The `@` of catcode 12 could have caused the same issue if it was not handled especially because it is used in the syntax as special variable for recursion hence was recognized even if of catcode 12. Anyway I could have handled the `_` like the `@`, to avoid this problem of infinite loop with a non-letter underscore used as first character but decided finally to have it be ignored (it is already ignored if among digits, but it can be a constituent of a function of variable name). It is not ignored of course if of catcode 11. It may then start a variable or function name, but only for use by the package (by `polexpr` for example), not by users.

Then the matter is handed over to specialized routines: gathering digits of a number (inclusive of a decimal mark, an exponential part) or letters of a function or variable. And we have to

intercept some tokens to implement various functionalities.

In each *dothis/orthat* structure, the first encountered branches are usually handled slower than the next, because `\if..\fi` test cost less than grabbing tokens. The exception is in the first one where letters pass through slightly faster than digits, presumably because the `\ifnum` test is more costly. Prior to this 1.4g refactoring the case of a starting letter of a variable or function name was handled last, it is now handled first. Now, this is only first letter...

Here are the various possibilities in order that they appear below (the indicative order of speed of treatment is given as a number).

- 1 tokens of catcode letter start a variable or function name
- 2 digits (I apply `\string` for the test, but I will have to review, it seems natural anyhow to require digits to be of catcode 12 and this is in fact basically done by the package, `\n2umexpr` does not work if not the case.),
- 7 support for Python-like \* "unpacking" unary operator (added at 1.4),
- 6 support for [ as opener for the [...] nutple constructor (1.4),
- 5 support for the minus as unary operator of variable precedence,
- 4 support for @ as first character of special variables even if not letter,
- 3 support for opening parentheses (possibly triggering tacit multiplication),
- 13 support for skipping over ignored + character,
- 12 support for numbers starting with a decimal point,
- 11 support for the ``+`()` and ``*`()` functions,
- 10 support for the `!()` function,
- 9 support for the `?()` function,
- 8 support for " for input of hexadecimal numbers and (1.4n) of ' for octal numbers.
- 17 support for `\xintdeffunc` via special handling of # token,
- 16 support for ignoring \_ if not of catcode 11 and at start of numbers or names (this 1.4g change fixes `\xinteval{4}` creating infinite loop)
- 15 support for inserting "nil" in front of operators, as needed in particular for the Python slicing syntax. This covers the comma, the :, the ] and the ) and also the ; although I don't think using ; to delimit nil is licit.
- 14 support for inserting 0 as missing value if / or ^ are encountered directly. This 1.4g changes avoids `\xinteval{/3}` causing unrecoverable low level errors from `\xintDiv` receiving only one argument.

I did not see here other bad syntax to protect.

The handling of "nil" insertion penalizes Python slicing but anyway time differences in the 14-15-16-17 group are less than 5%. The alternative will be to do some positive test for the targets (:, ], the comma and closing parenthesis) and do this in the prior group but this then penalizes others. Anyway. This is all negligible compared to actual computations...

Note: the above may not be in sync with code as it is extremely time-consuming to maintain correspondence in case of re-factoring.

```

570 \def\xINT_expr_getnextfork #1%
571 {%
572   \ifcat a#1\xint_dothis\xINT_expr_startfunc\fi
573   \ifnum \xint_c_ix<1\string#1 \xint_dothis\xINT_expr_startint\fi
574   \xint_orthat\xINT_expr_getnextfork_a #1%
575 }%
576 \def\xINT_expr_getnextfork_a #1%
577 {%
578   \if#1*\xint_dothis {{}}\xint_c_ii^v 0}\fi
579   \if#1[\xint_dothis {{}}\xint_c_ii^v \XINT_expr_itself_obracket}\fi
580   \if#1-\xint_dothis {{}}{-}\fi
581   \if#1@\xint_dothis{\XINT_expr_startfunc @}\fi
582   \if#1(\xint_dothis {{}}\xint_c_ii^v {})\fi

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

583 \xint_orthat{\XINT_expr_getnextfork_b#1}%
584 }%
585 \catcode96 11 % `
586 \def\xINT_expr_getnextfork_b #1%
587 {%
588 \if#1+\xint_dothis \XINT_expr_getnext_a\fi
589 \if#1.\xint_dothis \XINT_expr_startdec\fi
590 \if#1'\xint_dothis {\XINT_expr_onliteral_}\fi
591 \if#1!\xint_dothis {\XINT_expr_startfunc !}\fi
592 \if#1?\xint_dothis {\XINT_expr_startfunc ?}\fi
593 \if#1'\xint_dothis \XINT_expr_startoct\fi
594 \if#1"\xint_dothis \XINT_expr_starthex\fi
595 \xint_orthat{\XINT_expr_getnextfork_c#1}%
596 }%
597 \def\xINT_tmpa #1{%
598 \def\xINT_expr_getnextfork_c ##1%
599 {%
600 \if##1#1\xint_dothis \XINT_expr_getmacropar\fi
601 \if##1.\xint_dothis \XINT_expr_getnext_a\fi
602 \if0\if##1/1\fi\if##1^1\fi0\xint_dothis{\XINT_expr_insertnil##1}\fi
603 \xint_orthat{\XINT_expr_missing_arg##1}%
604 }%
605 }\expandafter\xINT_tmpa\string#%

```

The ` syntax is here used for special constructs like `+`(..), `\*`(..) where + or \* will be treated as functions. Current implementation picks only one token (could have been braced stuff), here it will be + or \*, and via `\XINT_expr_op_`` this then becomes a suitable `\XINT_{expr|iiexpr|flexpr}_func_+` (or \*). Documentation says to use `+`(..), but `+`(..) is also valid. The opening parenthesis must be there, it is not allowed to require some expansion.

```

606 \def\xINT_expr_onliteral_` #1#2({{#1}\xint_c_ii^v `}%
607 \catcode96 12 % `

```

Prior to 1.4g, I was using a `\lowercase` technique to insert the catcode 12 #, but this is a bit risky when one does not ensure a priori control of all lccodes.

```

608 \def\xINT_tmpa #1{%
609 \def\xINT_expr_getmacropar ##1%
610 {%
611 \expandafter{\expandafter{\expandafter#1\expandafter
612 ##1\expandafter}\expandafter}\romannumeral`&&\XINT_expr_getop
613 }%
614 }\expandafter\xINT_tmpa\string#%
615 \def\xINT_expr_insertnil #1%
616 {%
617 \expandafter{\expandafter}\romannumeral`&&\XINT_expr_getop_a#1%
618 }%
619 \def\xINT_expr_missing_arg#1%
620 {%
621 \expanded{\XINT_expandableerror
622 {Expected a value, got nothing before `#1'. Inserting 0.}{0}}\expandafter}%
623 \romannumeral`&&\XINT_expr_getop_a#1%
624 }%

```



**27.12. \XINT\_expr\_startint**

27.12.1	Integral part (skipping zeroes) . . . . .	583
27.12.2	Fractional part . . . . .	585
27.12.3	Scientific notation . . . . .	586
27.12.4	Hexadecimal numbers . . . . .	587
27.12.5	Octal numbers . . . . .	590
27.12.6	Binary numbers . . . . .	592
27.12.7	\XINT_expr_startfunc: collecting names of functions and variables . . . . .	593
27.12.8	\XINT_expr_func: dispatch to variable replacement or to function execution . . . . .	594

Following comments are in part OBSOLETE as the code was refactored at some stage to use `\expand`ed.

1.2 release has replaced chains of `\romannumeral-`0` by `\csname` governed expansion. Thus there is no more the limit at about 5000 digits for parsed numbers.

In order to avoid having to lock and unlock in succession to handle the scientific part and adjust the exponent according to the number of digits of the decimal part, the parsing of this decimal part counts on the fly the number of digits it encounters.

There is some slight annoyance with `\xintiexpr` which should never be given a `[n]` inside its `\csname.=<digits>\endcsname` storage of numbers (because its arithmetic uses the `ii` macros which know nothing about the `[N]` notation). Hence if the parser has only seen digits when hitting something else than the dot or `e` (or `E`), it will not insert a `[0]`. Thus we very slightly compromise the efficiency of `\xintexpr` and `\xintfloatexpr` in order to be able to share the same code with `\xintiexpr`.

Indeed, the parser at this location is completely common to all, it does not know if it is working inside `\xintexpr` or `\xintiexpr`. On the other hand if a dot or a `e` (or `E`) is met, then the (common) parser has no scruples ending this number with a `[n]`, this will provoke an error later if that was within an `\xintiexpr`, as soon as an arithmetic macro is used.

As the gathered numbers have no spaces, no pluses, no minuses, the only remaining issue is with leading zeroes, which are discarded on the fly. The hexadecimal numbers leading zeroes are stripped in a second stage by the `\xintHexToDec` macro.

With 1.2, `\xinttheexpr . \relax` does not work anymore (it did in earlier releases). There must be digits either before or after the decimal mark. Thus both `\xinttheexpr 1.\relax` and `\xinttheexpr .1\relax` are legal.

Attention at this location `#1` was of catcode 12 in all versions prior to 1.4.

We assume anyhow that catcodes of digits are 12...

Style of location of the braces around replacement text varies, but is kept as is to avoid wasting time on cosmetics.

**Modified at 1.4n (2025/09/05).** Support for the `0b`, `0o` and `0x` prefixes! Implementation of this extension of the syntax was very easy. I may have thought of it briefly in the past but presumed it would break things.

This change seems to only break stuff such as `+0bar+3` with `bar` a variable being interpreted `+0*b ar+3` (tacit multiplication) and will cause the parser to raise then an error message, but this is not very serious breaking change.

Supporting `0b` meant that like for octal 1.4n needed to add the needed support macros, which opens this TODO: perhaps try to share code between binary, octal, hexadecimal more than currently.

```

625 \def\XINT_expr_startint #1%
626 {%
627   \if #10\expandafter\XINT_expr_gobz_a\else\expandafter\XINT_expr_scanint_a\fi #1%
628 }%
629 \def\XINT_expr_scanint_a #1#2%
630   {\expanded\bgroup{{\iffalse}}\fi #1% spare a \string
631   \expandafter\XINT_expr_scanint_main\romannumeral`&&@#2}%

```

```

632 \def\XINT_expr_gobz_a #1#2%
633   {\expandafter\XINT_expr_gobz_b\romannumeral`&&@#2}%
    Perhaps use \if and not \ifx tests here?
634 \def\XINT_expr_gobz_b #1%
635 {%
636   \ifx b#1\xint_dothis \XINT_expr_startbin \fi
637   \ifx o#1\xint_dothis \XINT_expr_startoct \fi
638   \ifx x#1\xint_dothis \XINT_expr_starthex \fi
639   \xint_orthat {\XINT_expr_gobz_c #1}%
640 }%
641 \def\XINT_expr_gobz_c #1%
642   {\expanded\bgroup{{\iffalse}}}\fi
643   \expandafter\XINT_expr_gobz_scanint_main#1}%
644 \def\XINT_expr_startdec #1%
645   {\expanded\bgroup{{\iffalse}}}\fi
646   \expandafter\XINT_expr_scandec_a\romannumeral`&&@#1}%

```

### 27.12.1. Integral part (skipping zeroes)

1.2 has modified the code to give highest priority to digits, the accelerating impact is non-negligeable. I don't think the doubled `\string` is a serious penalty.

(reference to `\string` is obsolete: it is only used in the test but the tokens are not submitted to `\string` anymore)

```

647 \def\XINT_expr_scanint_main #1%
648 {%
649   \ifcat \relax #1\expandafter\XINT_expr_scanint_hit_cs \fi
650   \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanint_next\fi
651   #1\XINT_expr_scanint_again
652 }%
653 \def\XINT_expr_scanint_again #1%
654 {%
655   \expandafter\XINT_expr_scanint_main\romannumeral`&&@#1%
656 }%

```

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```

657 \def\XINT_expr_scanint_hit_cs \ifnum#1\fi#2\XINT_expr_scanint_again
658 {%
659   \iffalse{{\fi}}\expandafter\romannumeral`&&@\XINT_expr_getop_a#2%
660 }%

```

With 1.2d the tacit multiplication in front of a variable name or function name is now done with a higher precedence, intermediate between the common one of `*` and `/` and the one of `^`. Thus `x/2y` is like `x/(2y)`, but `x^2y` is like `x^2*y` and `2y!` is not `(2y)!` but `2*y!`.

Finally, 1.2d has moved away from the `_scan` macros all the business of the tacit multiplication in one unique place via `\XINT_expr_getop`. For this, the ending token is not first given to `\string` as was done earlier before handing over back control to `\XINT_expr_getop`. Earlier we had to identify the catcode 11 ! signaling a sub-expression here. With no `\string` applied we can do it in `\XINT_expr_getop`. As a corollary of this displacement, parsing of big numbers should be a tiny bit faster now.

Extended for 1.2l to ignore underscore character `_` if encountered within digits; so it can serve as separator for better readability.

It is not obvious at 1.4 to support `[]` for three things: packing, slicing, ... and raw `xintfrac` syntax `A/B[N]`. The only good way would be to actually really separate completely `\xintexpr`,



`\xintfloatexpr` and `\xintiexpr` code which would allow to handle both / and [] from A/B[N] as we handle e and E. But triplicating the code is something I need to think about. It is not possible as in pre 1.4 to consider [ only as an operator of same precedence as multiplication and division which was the way we did this, but we can use the technique of fake operators. Thus we intercept hitting a [ here, which is not too much of a problem as anyhow we dropped temporarily  $3*[1,2,3]+5$  syntax so we don't have to worry that  $3[1,2,3]$  should do tacit multiplication. I think only way in future will be to really separate the code of the three parsers (or drop entirely support for A/B[N]; as 1.4 has modified output of `\xinteval` to not use this notation this is not too dramatic).

Anyway we find a way to inject here the former handling of [N], which will use a delimited macro to directly fetch until the closing]. We do still need some fake operator because A/B[N] is (A/B) times  $10^N$  and the /B is allowed to be missing. We hack this using the \$ which is not used currently as operator elsewhere in the syntax and need to hook into `\XINT_expr_getop_b`. No finally I use the null char. It must be of catcode 12.

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```

661 \def\xINT_expr_scanint_next #1\xINT_expr_scanint_again
662 {%
663   \if [#1\xint_dothis\xINT_expr_rawxintfrac\fi
664   \if _#1\xint_dothis\xINT_expr_scanint_again\fi
665   \if e#1\xint_dothis{[\the\numexpr0\xINT_expr_scanexp_a +]\fi
666   \if E#1\xint_dothis{[\the\numexpr0\xINT_expr_scanexp_a +]\fi
667   \if .#1\xint_dothis{\XINT_expr_startdec_a .}\fi
668   \xint_orthat
669   {\iffalse{[\fi]]\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
670 }%
671 \def\xINT_expr_rawxintfrac
672 {%
673   \iffalse{[\fi]]\expandafter}\csname XINT_expr_precedence_&&\endcsname&&%
674 }%
675 \def\xINT_expr_gobz_scanint_main #1%
676 {%
677   \ifcat \relax #1\expandafter\xINT_expr_gobz_scanint_hit_cs\fi
678   \ifnum\xint_c_x<1\string#1 \else\expandafter\xINT_expr_gobz_scanint_next\fi
679   #1\xINT_expr_scanint_again
680 }%
681 \def\xINT_expr_gobz_scanint_again #1%
682 {%
683   \expandafter\xINT_expr_gobz_scanint_main\romannumeral`&&#1%
684 }%

```

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```

685 \def\xINT_expr_gobz_scanint_hit_cs\ifnum#1\fi#2\xINT_expr_scanint_again
686 {%
687   0\iffalse{[\fi]]\expandafter}\romannumeral`&&\XINT_expr_getop_a#2%
688 }%
689 \def\xINT_expr_gobz_scanint_next #1\xINT_expr_scanint_again
690 {%
691   \if [#1\xint_dothis{\expandafter0\xINT_expr_rawxintfrac}\fi
692   \if _#1\xint_dothis\xINT_expr_gobz_scanint_again\fi
693   \if e#1\xint_dothis{0[\the\numexpr0\xINT_expr_scanexp_a +]\fi
694   \if E#1\xint_dothis{0[\the\numexpr0\xINT_expr_scanexp_a +]\fi
695   \if .#1\xint_dothis{\XINT_expr_gobz_startdec_a .}\fi
696   \if 0#1\xint_dothis\xINT_expr_gobz_scanint_again\fi
697   \xint_orthat

```

```

698     {0\iffalse{{{fi}}}\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
699 }%

```

### 27.12.2. Fractional part

Annoying duplication of code to allow 0. as input.

1.2a corrects a very bad bug in 1.2 `\XINT_expr_gobz_scandec_b` which should have stripped leading zeroes in the fractional part but didn't; as a result `\xinttheexpr 0.01\relax` returned `0 =:-(((` Thanks to Kroum Tzanev who reported the issue. Does it improve things if I say the bug was introduced in 1.2, it wasn't present before ?

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```

700 \def\XINT_expr_startdec_a .#1%
701 {%
702     \expandafter\XINT_expr_scandec_a\romannumeral`&&#1%
703 }%
704 \def\XINT_expr_scandec_a #1%
705 {%
706     \if .#1\xint_dothis{\iffalse{{{fi}}}\expandafter}%
707         \romannumeral`&&\XINT_expr_getop_a..}\fi
708     \xint_orthat {\XINT_expr_scandec_main 0.#1}%
709 }%
710 \def\XINT_expr_gobz_startdec_a .#1%
711 {%
712     \expandafter\XINT_expr_gobz_scandec_a\romannumeral`&&#1%
713 }%
714 \def\XINT_expr_gobz_scandec_a #1%
715 {%
716     \if .#1\xint_dothis
717     {0\iffalse{{{fi}}}\expandafter}\romannumeral`&&\XINT_expr_getop_a..}\fi
718     \xint_orthat {\XINT_expr_gobz_scandec_main 0.#1}%
719 }%
720 \def\XINT_expr_scandec_main #1.#2%
721 {%
722     \ifcat \relax #2\expandafter\XINT_expr_scandec_hit_cs\fi
723     \ifnum\xint_c_ix<1\string#2 \else\expandafter\XINT_expr_scandec_next\fi
724     #2\expandafter\XINT_expr_scandec_again\the\numexpr #1-\xint_c_i.%
725 }%
726 \def\XINT_expr_scandec_again #1.#2%
727 {%
728     \expandafter\XINT_expr_scandec_main
729     \the\numexpr #1\expandafter.\romannumeral`&&#2%
730 }%

```

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```

731 \def\XINT_expr_scandec_hit_cs\ifnum#1\fi
732     #2\expandafter\XINT_expr_scandec_again\the\numexpr#3-\xint_c_i.%
733 {%
734     [#3]\iffalse{{{fi}}}\expandafter}\romannumeral`&&\XINT_expr_getop_a#2%
735 }%
736 \def\XINT_expr_scandec_next #1#2\the\numexpr#3-\xint_c_i.%
737 {%
738     \if _#1\xint_dothis{\XINT_expr_scandec_again#3.}\fi
739     \if e#1\xint_dothis{[\the\numexpr#3\XINT_expr_scanexp_a +}\fi

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```
740 \if E#1\xint_dothis{[\the\numexpr#3\XINT_expr_scanexp_a +]\fi
741 \xint_orthat
742 {[#3]\iffalse{[\fi]}\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
743 }%
744 \def\XINT_expr_gobz_scandec_main #1.#2%
745 {%
746 \ifcat \relax #2\expandafter\XINT_expr_gobz_scandec_hit_cs\fi
747 \ifnum\xint_c_ix<1\string#2 \else\expandafter\XINT_expr_gobz_scandec_next\fi
748 \if0#2\expandafter\xint_firstoftwo\else\expandafter\xint_secondoftwo\fi
749 {\expandafter\XINT_expr_gobz_scandec_main}%
750 [#2\expandafter\XINT_expr_scandec_again]\the\numexpr#1-\xint_c_i.%
751 }%
```

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```
752 \def\XINT_expr_gobz_scandec_hit_cs \ifnum#1\fi\if0#2#3\xint_c_i.%
753 {%
754 0[0]\iffalse{[\fi]}\expandafter}\romannumeral`&&\XINT_expr_getop_a#2%
755 }%
756 \def\XINT_expr_gobz_scandec_next\if0#1#2\fi #3\numexpr#4-\xint_c_i.%
757 {%
758 \if _#1\xint_dothis{\XINT_expr_gobz_scandec_main #4.}\fi
759 \if e#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +]\fi
760 \if E#1\xint_dothis{0[\the\numexpr0\XINT_expr_scanexp_a +]\fi
761 \xint_orthat
762 {0[0]\iffalse{[\fi]}\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
763 }%
```

### 27.12.3. Scientific notation

Some pluses and minuses are allowed at the start of the scientific part, however not later, and no parenthesis.

ATTENTION! `1e\numexpr2+3\relax` or `1e\xintiexpr i\relax`, `i=1..5` are not allowed and `1e1\numexpr 2\relax` does `1e1 * \numexpr2\relax`. Use `\the\numexpr`, `\xinttheiexpr`, etc...

```
764 \def\XINT_expr_scanexp_a #1#2%
765 {%
766 #1\expandafter\XINT_expr_scanexp_main\romannumeral`&&#2%
767 }%
768 \def\XINT_expr_scanexp_main #1%
769 {%
770 \ifcat \relax #1\expandafter\XINT_expr_scanexp_hit_cs\fi
771 \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanexp_next\fi
772 #1\XINT_expr_scanexp_again
773 }%
774 \def\XINT_expr_scanexp_again #1%
775 {%
776 \expandafter\XINT_expr_scanexp_main_b\romannumeral`&&#1%
777 }%
```

1.4f had `_getop` here, but let's jump directly to `_getop_a`.

```
778 \def\XINT_expr_scanexp_hit_cs\ifnum#1\fi#2\XINT_expr_scanexp_again
779 {%
780 ]\iffalse{[\fi]}\expandafter}\romannumeral`&&\XINT_expr_getop_a#2%
781 }%
782 \def\XINT_expr_scanexp_next #1\XINT_expr_scanexp_again
```

```

783 {%
784   \if _#1\xint_dothis \XINT_expr_scanexp_again \fi
785   \if +#1\xint_dothis {\XINT_expr_scanexp_a +}\fi
786   \if -#1\xint_dothis {\XINT_expr_scanexp_a -}\fi
787   \xint_orthat
788   {} \iffalse{{{\fi}}\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
789 }%
790 \def\XINT_expr_scanexp_main_b #1%
791 {%
792   \ifcat \relax #1\expandafter\XINT_expr_scanexp_hit_cs_b\fi
793   \ifnum\xint_c_ix<1\string#1 \else\expandafter\XINT_expr_scanexp_next_b\fi
794   #1\XINT_expr_scanexp_again_b
795 }%

1.4f had _getop here, but let's jump directly to _getop_a.
796 \def\XINT_expr_scanexp_hit_cs_b\ifnum#1\fi#2\XINT_expr_scanexp_again_b
797 {%
798   {} \iffalse{{{\fi}}\expandafter}\romannumeral`&&\XINT_expr_getop_a#2}%
799 }%
800 \def\XINT_expr_scanexp_again_b #1%
801 {%
802   \expandafter\XINT_expr_scanexp_main_b\romannumeral`&&#1%
803 }%
804 \def\XINT_expr_scanexp_next_b #1\XINT_expr_scanexp_again_b
805 {%
806   \if _#1\xint_dothis\XINT_expr_scanexp_again\fi
807   \xint_orthat
808   {} \iffalse{{{\fi}}\expandafter}\romannumeral`&&\XINT_expr_getop_a#1}%
809 }%

```

#### 27.12.4. Hexadecimal numbers

1.2d has moved most of the handling of tacit multiplication to `\XINT_expr_getop`, but we have to do some of it here, because we apply `\string` before calling `\XINT_expr_scanhexI_aa`. I do not insert the `*` in `\XINT_expr_scanhexI_a`, because it is its higher precedence variant which will be expected, to do the same as when a non-hexadecimal number prefixes a sub-expression. Tacit multiplication in front of variable or function names will not work (because of this `\string`).

Extended for 1.2l to ignore underscore character `_` if encountered within digits.

(some above remarks have been obsoleted for some long time, no more applied `\string` since 1.4)

Notice that internal representation adds a `[N]` part only in case input used "DDD.dddd form, for compatibility with `\xintiexpr` which is not compatible with such internal representation.

At 1.4g a very long-standing bug was fixed: input such as `"\foo` broke the parser because (incredibly) the `\foo` token was picked up unexpanded and ended up as is in an `\ifcat` !

Another long-standing bug was fixed at 1.4g: contrarily to the decimal case, here in the hexadecimal input leading zeros were not trimmed. This was ok, because formerly `\xintHexToDec` trimmed leading zeros, but at 1.2m 2017/07/31 `xintbinhex.sty` was modified and this ceased being the case. But I forgot to upgrade the parser here at that time. Leading zeros would in many circumstances (presence of a fractional part, or `\xintiexpr` context) lead to wrong results. Leading zeros are now trimmed during input.

```

810 \def\XINT_expr_hex_in #1.#2#3;%
811 {%
812   \expanded{{{\if#2>%
813     \xintHexToDec{#1}%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

814 \else
815 \xintiiMul{\xintiiPow{625}{\xintLength{#3}}}{\xintHexToDec{#1#3}}%
816 [\the\numexpr-4*\xintLength{#3}]%
817 \fi}}\expandafter\romannumeral`&&\XINT_expr_getop
818 }%

```

Let's not forget to grab-expand next token first as is normal rule of operation. Formerly called `\XINT_expr_scanhex_I` and had " upfront.

```

819 \def\XINT_expr_starthex #1%
820 {%
821 \expandafter\XINT_expr_hex_in\expanded\bgroup
822 \expandafter\XINT_expr_scanhexIgobz_a\romannumeral`&&#1%
823 }%
824 \def\XINT_expr_scanhexIgobz_a #1%
825 {%
826 \ifcat #1\relax
827 0.>;\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
828 \XINT_expr_scanhexIgobz_aa #1%
829 }%
830 \def\XINT_expr_scanhexIgobz_aa #1%
831 {%
832 \if\ifnum`#1>`0
833 \ifnum`#1>`9
834 \ifnum`#1>`@
835 \ifnum`#1>`F
836 0\else1\fi\else0\fi\else1\fi\else0\fi 1%
837 \xint_dothis\XINT_expr_scanhexI_b
838 \fi
839 \if 0#1\xint_dothis\XINT_expr_scanhexIgobz_bgob\fi
840 \if _#1\xint_dothis\XINT_expr_scanhexIgobz_bgob\fi
841 \if .#1\xint_dothis\XINT_expr_scanhexIgobz_toII\fi
842 \xint_orthat
843 {\XINT_expandableerror
844 {Expected hexadecimal digit, `_', or `.'. got `#1'. Using `0'.}%
845 0.>;\iffalse{\fi}}%
846 #1%
847 }%
848 \def\XINT_expr_scanhexIgobz_bgob #1#2%
849 {%
850 \expandafter\XINT_expr_scanhexIgobz_a\romannumeral`&&#2%
851 }%
852 \def\XINT_expr_scanhexIgobz_toII .#1%
853 {%
854 0..\expandafter\XINT_expr_scanhexII_a\romannumeral`&&#1%
855 }%
856 \def\XINT_expr_scanhexI_a #1%
857 {%
858 \ifcat #1\relax
859 .>;\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
860 \XINT_expr_scanhexI_aa #1%
861 }%
862 \def\XINT_expr_scanhexI_aa #1%
863 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

864 \if\ifnum`#1>`/
865 \ifnum`#1>`9
866 \ifnum`#1>`@
867 \ifnum`#1>`F
868 0\else1\fi\else0\fi\else1\fi\else0\fi 1%
869 \expandafter\XINT_expr_scanhexI_b
870 \else
871 \if _#1\xint_dothis{\expandafter\XINT_expr_scanhexI_bgob}\fi
872 \if .#1\xint_dothis{\expandafter\XINT_expr_scanhexI_toII}\fi
873 \xint_orthat {.;\iffalse{\fi\expandafter}}}%
874 \fi
875 #1%
876 }%
877 \def\XINT_expr_scanhexI_b #1#2%
878 {%
879 #1\expandafter\XINT_expr_scanhexI_a\romannumeral`&&@#2%
880 }%
881 \def\XINT_expr_scanhexI_bgob #1#2%
882 {%
883 \expandafter\XINT_expr_scanhexI_a\romannumeral`&&@#2%
884 }%
885 \def\XINT_expr_scanhexI_toII .#1%
886 {%
887 ..\expandafter\XINT_expr_scanhexII_a\romannumeral`&&@#1%
888 }%
889 \def\XINT_expr_scanhexII_a #1%
890 {%
891 \ifcat #1\relax\xint_dothis{;\iffalse{\fi}#1}\fi
892 \xint_orthat {\XINT_expr_scanhexII_aa #1}%
893 }%
894 \def\XINT_expr_scanhexII_aa #1%
895 {%
896 \if\ifnum`#1>`/
897 \ifnum`#1>`9
898 \ifnum`#1>`@
899 \ifnum`#1>`F
900 0\else1\fi\else0\fi\else1\fi\else0\fi 1%
901 \expandafter\XINT_expr_scanhexII_b
902 \else
903 \if _#1\xint_dothis{\expandafter\XINT_expr_scanhexII_bgob}\fi
904 \xint_orthat{;\iffalse{\fi\expandafter}}}%
905 \fi
906 #1%
907 }%
908 \def\XINT_expr_scanhexII_b #1#2%
909 {%
910 #1\expandafter\XINT_expr_scanhexII_a\romannumeral`&&@#2%
911 }%
912 \def\XINT_expr_scanhexII_bgob #1#2%
913 {%
914 \expandafter\XINT_expr_scanhexII_a\romannumeral`&&@#2%
915 }%

```

### 27.12.5. Octal numbers

Added at 1.4n (2025/09/05).

The parsing goes exactly as with hexadecimal inputs, except that we require an octal digit when expected.

TODO: think about perhaps sharing code between binary, octal, and hexadecimal.

```

916 \def\XINT_expr_oct_in #1.#2#3;%
917 {%
918   \expanded{{{if#2>%
919     \xintOctToDec{#1}%
920   \else
921     \xintiiMul{\xintiiPow{125}{\xintLength{#3}}}{\xintOctToDec{#1#3}}%
922     [\the\numexpr-3*\xintLength{#3}]%
923   \fi}}\expandafter}\romannumeral`&&\XINT_expr_getop
924 }%
925 %   \begin{macrocode}
926 \def\XINT_expr_startoct #1%
927 {%
928   \expandafter\XINT_expr_oct_in\expanded\bgroup
929   \expandafter\XINT_expr_scanoctIglobz_a\romannumeral`&&#1%
930 }%
931 \def\XINT_expr_scanoctIglobz_a #1%
932 {%
933   \ifcat #1\relax
934     0.>.\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
935   \XINT_expr_scanoctIglobz_aa #1%
936 }%

```

Wondering if a bunch of `\if` tests comparing with 0 to 7 would not be more efficient here. But testing would take some time I am not motivated enough to use for that. In the code, also added at 1.4n for parsing binary input, I do this, but of course then it was rather more natural to do. But I did not test either comparing with style as here.

```

937 \def\XINT_expr_scanoctIglobz_aa #1%
938 {%
939   \if\ifnum`#1>`0 \ifnum`#1>`7 0\else1\fi\else0\fi 1%
940   \xint_dothis\XINT_expr_scanoctI_b
941   \fi
942   \if 0#1\xint_dothis\XINT_expr_scanoctIglobz_bgob\fi
943   \if _#1\xint_dothis\XINT_expr_scanoctIglobz_bgob\fi
944   \if .#1\xint_dothis\XINT_expr_scanoctIglobz_toII\fi
945   \xint_orthat
946   {\XINT_expandableerror
947     {Expected an octal digit, `_', or `.'. Got `#1'. Using `0'.}%
948     0.>.\iffalse{\fi}}%
949   #1%
950 }%
951 \def\XINT_expr_scanoctIglobz_bgob #1#2%
952 {%
953   \expandafter\XINT_expr_scanoctIglobz_a\romannumeral`&&#2%
954 }%
955 \def\XINT_expr_scanoctIglobz_toII .#1%
956 {%
957   0..\expandafter\XINT_expr_scanoctII_a\romannumeral`&&#1%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
958 }%
959 \def\XINT_expr_scanoctI_a #1%
960 {%
961     \ifcat #1\relax
962         .>;\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
963     \XINT_expr_scanoctI_aa #1%
964 }%
965 \def\XINT_expr_scanoctI_aa #1%
966 {%
967     \if\ifnum`#1>`/ \ifnum`#1>`7 0\else1\fi\else0\fi 1%
968     \expandafter\XINT_expr_scanoctI_b
969     \else
970     \if _#1\xint_dothis{\expandafter\XINT_expr_scanoctI_bgob}\fi
971     \if .#1\xint_dothis{\expandafter\XINT_expr_scanoctI_toII}\fi
972     \xint_orthat {.>;\iffalse{\fi\expandafter}}%
973     \fi
974     #1%
975 }%
976 \def\XINT_expr_scanoctI_b #1#2%
977 {%
978     #1\expandafter\XINT_expr_scanoctI_a\romannumeral`&&@#2%
979 }%
980 \def\XINT_expr_scanoctI_bgob #1#2%
981 {%
982     \expandafter\XINT_expr_scanoctI_a\romannumeral`&&@#2%
983 }%
984 \def\XINT_expr_scanoctI_toII .#1%
985 {%
986     ..\expandafter\XINT_expr_scanoctII_a\romannumeral`&&@#1%
987 }%
988 \def\XINT_expr_scanoctII_a #1%
989 {%
990     \ifcat #1\relax\xint_dothis{;\iffalse{\fi}#1}\fi
991     \xint_orthat {\XINT_expr_scanoctII_aa #1}%
992 }%
993 \def\XINT_expr_scanoctII_aa #1%
994 {%
995     \if\ifnum`#1>`/ \ifnum`#1>`7 0\else1\fi\else0\fi 1%
996     \expandafter\XINT_expr_scanoctII_b
997     \else
998     \if _#1\xint_dothis{\expandafter\XINT_expr_scanoctII_bgob}\fi
999     \xint_orthat{;\iffalse{\fi\expandafter}}%
1000     \fi
1001     #1%
1002 }%
1003 \def\XINT_expr_scanoctII_b #1#2%
1004 {%
1005     #1\expandafter\XINT_expr_scanoctII_a\romannumeral`&&@#2%
1006 }%
1007 \def\XINT_expr_scanoctII_bgob #1#2%
1008 {%
1009     \expandafter\XINT_expr_scanoctII_a\romannumeral`&&@#2%
```



1010 }%

### 27.12.6. Binary numbers

**Added at 1.4n (2025/09/05).** Analogous to hexadecimal or octal with some simpler tests for digits.

```

1011 \def\XINT_expr_bin_in #1.#2#3;%
1012 {%
1013   \expanded{{\if#2>%
1014     \xintBinToDec{#1}%
1015   \else
1016     \xintiiMul{\xintiiPow{5}{\xintLength{#3}}}{\xintBinToDec{#1#3}}%
1017     [\the\numexpr-\xintLength{#3}]%
1018   \fi}}\expandafter\romannumeral`&&\XINT_expr_getop
1019 }%
1020 %   \begin{macrocode}
1021 \def\XINT_expr_startbin #1%
1022 {%
1023   \expandafter\XINT_expr_bin_in\expanded\bgroup
1024   \expandafter\XINT_expr_scanbinIgobz_a\romannumeral`&&#1%
1025 }%
1026 \def\XINT_expr_scanbinIgobz_a #1%
1027 {%
1028   \ifcat #1\relax
1029     0.>;\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
1030   \XINT_expr_scanbinIgobz_aa #1%
1031 }%
1032 \def\XINT_expr_scanbinIgobz_aa #1%
1033 {%
1034   \if_#1\xint_dothis\XINT_expr_scanbinIgobz_bgob\fi
1035   \if.#1\xint_dothis\XINT_expr_scanbinIgobz_toII\fi
1036   \if 0#1\xint_dothis\XINT_expr_scanbinIgobz_bgob\fi
1037   \if 1#1\xint_dothis\XINT_expr_scanbinI_b\fi
1038   \xint_orthat
1039   {\XINT_expandableerror
1040     {Expected a binary digit, `_', or `.'. Got `#1'. Using `0'.}%
1041     0.>;\iffalse{\fi}}%
1042   #1%
1043 }%
1044 \def\XINT_expr_scanbinIgobz_bgob #1#2%
1045 {%
1046   \expandafter\XINT_expr_scanbinIgobz_a\romannumeral`&&#2%
1047 }%
1048 \def\XINT_expr_scanbinIgobz_toII .#1%
1049 {%
1050   0..\expandafter\XINT_expr_scanbinII_a\romannumeral`&&#1%
1051 }%
1052 \def\XINT_expr_scanbinI_a #1%
1053 {%
1054   \ifcat #1\relax
1055     .>;\iffalse{\fi\expandafter}\expandafter\xint_gobble_i\fi
1056   \XINT_expr_scanbinI_aa #1%
1057 }%
```

```

1058 \def\xINT_expr_scanbinI_aa #1%
1059 {%
1060   \if_#1\xint_dothis\xINT_expr_scanbinI_bgob\fi
1061   \if_#1\xint_dothis\xINT_expr_scanbinI_toII\fi
1062   \if_0#1\xint_dothis\xINT_expr_scanbinI_b\fi
1063   \if_1#1\xint_dothis\xINT_expr_scanbinI_b\fi
1064   \xint_orthat {.>;\iffalse{\fi}}%
1065   #1%
1066 }%
1067 \def\xINT_expr_scanbinI_b #1#2%
1068 {%
1069   #1\expandafter\xINT_expr_scanbinI_a\romannumeral`&&@#2%
1070 }%
1071 \def\xINT_expr_scanbinI_bgob #1#2%
1072 {%
1073   \expandafter\xINT_expr_scanbinI_a\romannumeral`&&@#2%
1074 }%
1075 \def\xINT_expr_scanbinI_toII .#1%
1076 {%
1077   ..\expandafter\xINT_expr_scanbinII_a\romannumeral`&&@#1%
1078 }%
1079 \def\xINT_expr_scanbinII_a #1%
1080 {%
1081   \ifcat #1\relax\xint_dothis{;\iffalse{\fi}#1}\fi
1082   \xint_orthat {\xINT_expr_scanbinII_aa #1}%
1083 }%
1084 \def\xINT_expr_scanbinII_aa #1%
1085 {%
1086   \if_#1\xint_dothis\xINT_expr_scanbinII_bgob\fi
1087   \if_0#1\xint_dothis\xINT_expr_scanbinII_b\fi
1088   \if_1#1\xint_dothis\xINT_expr_scanbinII_b\fi
1089   \xint_orthat{;\iffalse{\fi}}%
1090   #1%
1091 }%
1092 \def\xINT_expr_scanbinII_b #1#2%
1093 {%
1094   #1\expandafter\xINT_expr_scanbinII_a\romannumeral`&&@#2%
1095 }%
1096 \def\xINT_expr_scanbinII_bgob #1#2%
1097 {%
1098   \expandafter\xINT_expr_scanbinII_a\romannumeral`&&@#2%
1099 }%

```

### 27.12.7. \xINT\_expr\_startfunc: collecting names of functions and variables

At 1.4 the first token left over has not been submitted to `\string`. We also know it is not a control sequence. So we can test catcode to identify if operator is found. And it is allowed to hit some operator such as a closing parenthesis we will then insert the «nil» value (edited: which however will cause certain breakage of the infix binary operators: I notice I did not insert None `{{}}` but nil `{}`, perhaps by oversight).

There was prior to 1.4 solely the dispatch in `\xINT_expr_scanfunc_b` but now we do it immediately and issue `\xINT_expr_func` only in certain cases.

Comments here have been removed because 1.4g did a refactoring and renamed `\xINT_expr_scanfunc`

to `\XINT_expr_startfunc`, moving half of it earlier inside the `getnextfork` macros.

```
1100 \def\XINT_expr_startfunc #1%
1101   {\expandafter\XINT_expr_func\expanded\bgroup#1\XINT_expr_scanfunc_a}%
1102 \def\XINT_expr_scanfunc_a #1%
1103 {%
1104   \expandafter\XINT_expr_scanfunc_b\romannumeral`&&@#1%
1105 }%
```

This handles: 1) (indirectly) tacit multiplication by a variable in front a of sub-expression, 2) (indirectly) tacit multiplication in front of a `\count` etc..., 3) functions which are recognized via an encountered opening parenthesis (but later this must be disambiguated from variables with tacit multiplication) 4) 5) 6) 7) acceptable components of a variable or function names: @, underscore, digits, letters (or chars of category code letter.)

The short lived 1.2d which followed the even shorter lived 1.2c managed to introduce a bug here as it removed the check for catcode 11 !, which must be recognized if ! is not to be taken as part of a variable name. Don't know what I was thinking, it was the time when I was moving the handling of tacit multiplication entirely to the `\XINT_expr_getop` side. Fixed in 1.2e.

I almost decided to remove the `\ifcat\relax` test whose rôle is to avoid the `\string#1` to do something bad is the escape char is a digit! Perhaps I will remove it at some point ! I truly almost did it, but also the case of no escape char is a problem (`\string\0`, if `\0` is a count ...)

The (indirectly) above means that via `\XINT_expr_func` then `\XINT_expr_op__` one goes back to `\XINT_expr_getop` then `\XINT_expr_getop_b` which is the location where tacit multiplication is now centralized. This makes the treatment of tacit multiplication for situations such as `<variable>\count` or `<variable>xintexpr.\relax`, perhaps a bit sub-optimal, but first the variable name must be gathered, second the variable must expand to its value.

```
1106 \def\XINT_expr_scanfunc_b #1%
1107 {%
1108   \ifcat \relax#1\xint_dothis{\iffalse{\fi}(_#1)\fi
1109   \if (#1\xint_dothis{\iffalse{\fi}(`)\fi
1110   \if 1\ifcat a#10\fi
1111     \ifnum\xint_c_ix<1\string#1 0\fi
1112     \if @#10\fi
1113     \if _#10\fi
1114     1%
1115     \xint_dothis{\iffalse{\fi}(_#1)\fi
1116     \xint_orthat {#1\XINT_expr_scanfunc_a}%
1117 }%
```

### 27.12.8. `\XINT_expr_func`: dispatch to variable replacement or to function execution

Comments written 2015/11/12: earlier there was an `\ifcsname` test for checking if we had a variable in front of a (, for tacit multiplication for example in `x(y+z(x+w))` to work. But after I had implemented functions (that was yesterday...), I had the problem if was impossible to re-declare a variable name such as "f" as a function name. The problem is that here we can not test if the function is available because we don't know if we are in `expr`, `iiexpr` or `floatexpr`. The `\xint_c_ii^v` causes all fetching operations to stop and control is handed over to the routines which will be `expr`, `iiexpr` ou `floatexpr` specific, i.e. the `\XINT_{expr|iiexpr|flexpr}_op_{`|_}` which are invoked by the `until_<op>_b` macros earlier in the stream. Functions may exist for one but not the two other parsers. Variables are declared via one parser and usable in the others, but naturally `\xintiexpr` has its restrictions.

Thinking about this again I decided to treat a priori cases such as `x(...)` as functions, after having assigned to each variable a low-weight macro which will convert this into `_getop\.=<value`

of `x>*(...)`. To activate that macro at the right time I could for this exploit the "onliteral" intercept, which is parser independent (1.2c).

This led to me necessarily to rewrite partially the `seq`, `add`, `mul`, `subs`, `iter` ... routines as now the variables fetch only one token. I think the thing is more efficient.

1.2c had `\def\XINT_expr_func #1(#2{\xint_c_ii^v #2{#1}}`

In `\XINT_expr_func` the #2 is `_` if #1 must be a variable name, or #2=``` if #1 must be either a function name or possibly a variable name which will then have to be followed by tacit multiplication before the opening parenthesis.

The `\xint_c_ii^v` is there because `_op_`` must know in which parser it works. Dispendious for `_`. Hence I modify for 1.2d.

```
1118 \def\XINT_expr_func #1(#2{\if _#2\xint_dothis{\XINT_expr_op_{#1}}\fi
1119      \xint_orthat{#1}\xint_c_ii^v #2}}%
```

### 27.13. `\XINT_expr_op_``: launch function or pseudo-function, or evaluate variable and insert operator of multiplication in front of parenthesized contents

The "onliteral" intercepts is for `bool`, `togl`, `protect`, ... but also for `add`, `mul`, `seq`, etc... Genuine functions have `expr`, `iiexpr` and `flexpr` versions (or only one or two of the three) and trigger here the use of the suitable parser-dependant form. The former (pseudo functions and functions handling dummy variables) first trigger a parser independent mechanism.

With 1.2c "onliteral" is also used to disambiguate a variable followed by an opening parenthesis from a function and then apply tacit multiplication. However as I use only a `\ifcsname` test, in order to be able to re-define a variable as function, I move the check for being a function first. Each variable name now has its `onliteral_<name>` associated macro. This used to be decided much earlier at the time of `\XINT_expr_func`.

The advantage of 1.2c code is that the same name can be used for a variable or a function.

**Modified at 1.4i (2021/06/11).** The 1.2c abuse of «onliteral» for both tacit multiplication in front of an opening parenthesis and «generic» functions or pseudo-functions meant that the latter were vulnerable against user redefinition of a function name as a variable name. This applied to `subs`, `subsm`, `subsn`, `seq`, `add`, `mul`, `ndseq`, `ndmap`, `ndfillraw`, `bool`, `togl`, `protect`, `qint`, `qfrac`, `qfloat`, `qraw`, `random`, `qrand`, `rbit` and the most susceptible in real life was probably "seq".

Now variables have an associated «var\*» named macro, not «onliteral».

In passing I refactor here in a `\romannumeral` inspired way how `\csname` and TeX booleans are intertwined, minimizing `\expandafter` usage.

```
1120 \def\XINT_tmpa #1#2#3{%
1121   \def #1##1%
1122     {%
1123       \csname
1124         XINT_\ifcsname XINT_#3_func_##1\endcsname
1125         #3_func_##1\expandafter\endcsname\romannumeral`&&\expandafter#2%
1126       \romannumeral\else
1127       \ifcsname XINT_expr_onliteral_##1\endcsname
1128         expr_onliteral_##1\expandafter\endcsname\romannumeral
1129       \else
1130       \ifcsname XINT_expr_var*_##1\endcsname
1131         expr_var*_##1\expandafter\endcsname\romannumeral
1132       \else
1133         #3_func_\XINT_expr_unknown_function {##1}%
1134         \expandafter\endcsname\romannumeral`&&\expandafter#2%
1135       \romannumeral
1136     \fi\fi\fi\xint_c_
```

```

1137 }%
1138 }%
1139 \xintFor #1 in {expr,flexpr,iiexpr} \do {%
1140     \expandafter\XINT_tmpa
1141         \csname XINT_#1_op_\` \expandafter\endcsname
1142         \csname XINT_#1_oparen\endcsname
1143         {#1}%
1144 }%
1145 \def\XINT_expr_unknown_function #1%
1146     {\XINT_expandableerror{`#1' is unknown, say `Isome_func' or I use 0.}}%
1147 \def\XINT_expr_func_ #1#2#3{#1#2{{0}}}%
1148 \let\XINT_flexpr_func_\XINT_expr_func_
1149 \let\XINT_iiexpr_func_\XINT_expr_func_

```

## 27.14. \XINT\_expr\_op\_\_: replace a variable by its value and then fetch next operator

The 1.1 mechanism for `\XINT_expr_var_<varname>` has been modified in 1.2c. The `<varname>` associated macro is now only expanded once, not twice. We arrive here via `\XINT_expr_func`.

At 1.4 `\XINT_expr_getop` is launched with accumulated result on its left. But the omit and abort keywords are implemented via fake variables which rely on possibility to modify incoming upfront tokens. If we did here something such as

```
_var_#1\expandafter\endcsname\romannumeral`^^@\XINT_expr_getop
```

the premature expansion of `getop` would break the `var_omit` and `var_abort` mechanism. Thus we revert to former code which locates an `\XINT_expr_getop` (call it `_legacy`) before the tokens from the variable expansion (in `xintexpr < 1.4` the normal variables expanded to a single token so the overhead was not serious) so we can expand fake variables first.

Abusing variables to manipulate the incoming token stream is a bit bad, usually I prefer functions for this (such as the `break()` function) but then I have to define 3 macros for the 3 parsers.

This trick of fake variables puts thus a general overhead at various locations, and the situation here is REALLY not satisfactory. But 1.4 has (had) to be released now.

Even if I could put the `\csname XINT_expr_var_foo\endcsname` upfront, which would then be f-expanded, this would still need `\XINT_expr_put_op_first` to use its `\expandafter`'s as long as `\XINT_expr_var_foo` expands to `{\XINT_expr_varvalue_foo}` with a not-yet expanded `\XINT_expr_varvalue_foo`.

I could let `\XINT_expr_var_foo` expand to `\expandafter{\XINT_expr_varvalue_foo}` allowing then (if it gets f-expanded) probably to drop the `\expandafter` in `\XINT_expr_put_op_first`. But I can not consider this option in the form

```
_var_foo\expandafter\endcsname\romannumeral`^^@\XINT_expr_getop
```

until the issue with fake variables such as `omit` and `abort` which must act before `\XINT_expr_getop` has some workaround. This could be implemented here with some extra branch, i.e. there would not be some `\XINT_expr_var_omit` but something else filtered out in the `\else` branch here.

The above comments mention only `omit` and `abort`, but the case of real dummy variables also needs consideration.

At 1.4g, I test first for existence of `\XINT_expr_onliteral_foo`.

Updated for 1.4i: now rather existence of `\XINT_expr_var*_foo` is tested.

This is a trick which allows to distinguish actual or dummy variables from really fake variables `omit` and `abort` (must check if there are others). For the real or dummy variables we can trigger the expansion of the `\XINT_expr_getop` before the one of the variable. I could test `vor varvalue_foo` but this applies only to real variables not dummy variables. Actual and dummy variables are thus handled slightly faster at 1.4g as there is less induced moving around (the `\expandafter` chain in `\XINT_expr_put_op_first` still applies at this stage, as I have not yet re-examined the `var/varvalue`

mechanism). And the test for `var_foo` is moved directly inside the `\csname` construct in the `\else` branch which now handles together fake variables and non-existing variables.

I only have to make sure dummy variables are really safe being handled this way with the `getop` action having being done before they expand, but it looks ok. Attention it is crucial that if `\XINT_expr_getop` finds a `\relax` it inserts `\xint_c_\relax` so the `\relax` token is still there!

With this refactoring the `\XINT_expr_getop_legacy` is applied only in case of non-existent variables or fake variables omit/abort or things such as `nil`, `None`, `false`, `true`, `False`, `True`.

If user in interactive mode fixes the variable name, the `\XINT_expr_var_foo` expanded once with `deliver {\XINT_expr_varvalue_foo}` (if not dummy), and the braces are maintained by `\XINT_expr_getop_legacy`.

```

1150 \def\XINT_expr_op__ #1% op__ with two _'s
1151 {%
1152   \ifcsname XINT_expr_var*_{#1}\endcsname
1153     \csname XINT_expr_var*_{#1}\expandafter\endcsname
1154     \romannumeral`&&\expandafter\XINT_expr_getop
1155   \else
1156     \expandafter\expandafter\expandafter\XINT_expr_getop_legacy
1157     \csname XINT_expr_var*_{#1}\endcsname
1158     \ifcsname XINT_expr_var*_{#1}\endcsname#1\else\XINT_expr_unknown_variable{#1}\fi
1159   \expandafter\endcsname
1160 \fi
1161 }%
1162 \def\XINT_expr_unknown_variable #1%
1163   {\XINT_expandableerror {`#1' unknown, say `Isome_var' or I use 0.}}%
1164 \def\XINT_expr_var*_{#1}%
1165 \let\XINT_flexpr_op__ \XINT_expr_op__
1166 \let\XINT_iexpr_op__ \XINT_expr_op__
1167 \def\XINT_expr_getop_legacy #1%
1168 {%
1169   \expanded{\xint_noexpd{#1}\expandafter}\romannumeral`&&\XINT_expr_getop
1170 }%
```

## 27.15. `\XINT_expr_getop`: fetch the next operator or closing parenthesis or end of expression

Release 1.1 implements multi-character operators.

1.2d adds tacit multiplication also in front of variable or functions names starting with a letter, not only a `@` or a `_` as was already the case. This is for  $(x+y)z$  situations. It also applies higher precedence in cases like  $x/2y$  or  $x/2@$ , or  $x/2\max(3,5)$ , or  $x/2\xintexpr 3\relax$ .

In fact, finally I decide that all sorts of tacit multiplication will always use the higher precedence.

Indeed I hesitated somewhat: with the current code one does not know if `\XINT_expr_getop` as invoked after a closing parenthesis or because a number parsing ended, and I felt distinguishing the two was unneeded extra stuff. This means cases like  $(a+b)/(c+d)(e+f)$  will first multiply the last two parenthesized terms.

1.2q adds tacit multiplication in cases such as  $(1+1)3$  or  $5!7!$

1.4 has simplified coding here as `\XINT_expr_getop` expansion happens at a time when a fetched value has already being stored.

Prior to 1.4g there was an `\if_{#1}\xint_dothis\xint_secondofthree\fi` because the `_` can be used to start names, for private use by package (for example by `polexpr`). But this test was silly because these usages are only with a `_` of catcode 11. And allowing non-catcode 11 `_` also to trigger

tacit multiplication caused an infinite loop in collaboration with `\XINT_expr_scanfunc`, see explanations there (now removed after refactoring, see `\XINT_expr_startfunc`).

The situation with the `@` is different because we must allow it even as catcode 12 as a name, as it used in the syntax and must work the same if of catcode 11 or 12. No infinite loop because it is filtered out by one of the `\XINT_expr_getnextfork` macros.

The check for `:` to send it to thirdofthree "getop" branch is needed, last time I checked, because during some part of at least `\xintdeffunc`, some scantokens are done which need to work with the `:` of catcode 11, and it would be misconstrued to start a name if not filtered out.

```

1171 \def\XINT_expr_getop #1%
1172 {%
1173   \expandafter\XINT_expr_getop_a\romannumeral`&&@#1%
1174 }%
1175 \catcode`* 11
1176 \def\XINT_expr_getop_a #1%
1177 {%
1178   \ifx \relax #1\xint_dothis\xint_firstofthree\fi
1179   \ifcat \relax #1\xint_dothis\xint_secondofthree\fi
1180   \ifnum\xint_c_ix<1\string#1 \xint_dothis\xint_secondofthree\fi
1181   \if :#1\xint_dothis \xint_thirdofthree\fi
1182   \if @#1\xint_dothis \xint_secondofthree\fi
1183   \if (#1\xint_dothis \xint_secondofthree\fi %)
1184   \ifcat a#1\xint_dothis \xint_secondofthree\fi
1185   \xint_orthat \xint_thirdofthree

```

Formerly `\XINT_expr_foundedend` as firstofthree but at 1.4g let's simply insert `\xint_c_` as the #1 is `\relax` (and anyhow a place-holder according to remark in definition of `\XINT_expr_foundedend`

```

1186   \xint_c_

```

Tacit multiplication with higher precedence. Formerly `\XINT_expr_precedence_***` was used, renamed to `\XINT_expr_prec_tacit` at 1.4g in case a backport is done of the `\bnumdefinfix` from `bnumexpr`.

```

1187   {\XINT_expr_prec_tacit *}%

```

This is only location which jumps to `\XINT_expr_getop_b`. At 1.4f and perhaps for old legacy reasons this was `\expandafter\XINT_expr_getop_b \string#1` but I see no reason now for applying `\string` to #1. Removed at 1.4g. And the #1 now moved out of the secondofthree and thirdofthree branches.

```

1188   \XINT_expr_getop_b
1189   #1%
1190 }%
1191 \catcode`* 12

```

`\relax` is a place holder here. At 1.4g, we don't use `\XINT_expr_foundedend` anymore in `\XINT_expr_getop_a` which was slightly refactored, but it is used elsewhere.

Attention that keeping a `\relax` around if `\XINT_expr_getop` hits it is crucial to good functioning of dummy variables after 1.4g refactoring of `\XINT_expr_op__`, the `\relax` being used as delimiter by dummy variables, and `\XINT_expr_getop` is now expanded before the variable itself does its thing.

```

1192 \def\XINT_expr_foundedend {\xint_c_ \relax}%

```

`?` is a very special operator with top precedence which will check if the next token is another `?`, while avoiding removing a brace pair from token stream due to its syntax. Pre 1.1 releases used `:` rather than `??`, but we need `:` for Python like slices of lists.

null char is used as hack to implement `A/B[N]` raw input at 1.4. See also `\XINT_expr_scanint_c`.



Memo: 1.4g, the token fetched by `\XINT_expr_getop_b` has not anymore been previously submitted in `\XINT_expr_getop_a` to `\string`.

```

1193 \def\XINT_expr_getop_b#1{\def\XINT_expr_getop_b ##1%
1194 {%
1195     \if &&@##1\xint_dothis{#1&&@}\fi
1196     \if '##1\xint_dothis{\XINT_expr_binopwrdrd }\fi
1197     \if ?##1\xint_dothis{\XINT_expr_precedence_? ?}\fi
1198     \xint_orthat {\XINT_expr_scanop_a ##1}%
1199 }}\expandafter\XINT_expr_getop_b\csname XINT_expr_precedence_&&\endcsname
1200 \def\XINT_expr_binopwrdrd #1'%
1201 {%
1202     \expandafter\XINT_expr_foundop_a
1203     \csname XINT_expr_itself_\xint_zapspaces #1 \xint_gobble_i\endcsname
1204 }%
1205 \def\XINT_expr_scanop_a #1#2%
1206 {%
1207     \expandafter\XINT_expr_scanop_b\expandafter#1\romannumeral`&&@#2%
1208 }%

```

Multi-character operators have an associated `itself` macro at each stage of decomposition starting at two characters. Here, nothing imposes to the operator characters not to be of catcode letter, this constraint applies only on the first character and is done via `\XINT_expr_getop_a`, to handle in particular tacit multiplication in front of variable or function names.

But it would be dangerous to allow letters in operator characters, again due to existence of variables and functions, and anyhow there is no user interface to add such custom operators. However in `bnumexpr`, such a constraint does not exist.

I don't worry too much about efficiency here... and at 1.4g I have re-written for code readability only. Once we see that `#1#2` is not a candidate to be or start an operator, we need to check if single-character operator `#1` is really an operator and this is done via the existence of the precedence token.

Unfortunately the 1.4g refactoring of the scanop macros had a bad bug: `\XINT_expr_scanop_c` inserted `\romannumeral`^^@` in stream but did not grab a token first so a space would stop the `\romannumeral` and then the `#2` in `\XINT_expr_scanop_d` was not pre-expanded and ended up alone in `\ifcat`. It is too distant in the past the time when I wrote the core of `xintexpr` in 2013... older and dumber now.

```

1209 \def\XINT_expr_scanop_b #1#2%
1210 {%
1211     \unless\ifcat#2\relax
1212         \ifcsname XINT_expr_itself_#1#2\endcsname
1213             \XINT_expr_scanop_c
1214         \fi\fi
1215     \XINT_expr_foundop_a #1#2%
1216 }%
1217 \def\XINT_expr_scanop_c #1#2#3#4#5#6% #1#2=\fi\fi
1218 {%
1219     #1#2%
1220     \expandafter\XINT_expr_scanop_d\csname XINT_expr_itself_#4#5\expandafter\endcsname
1221     \romannumeral`&&@#6%
1222 }%
1223 \def\XINT_expr_scanop_d #1#2%
1224 {%
1225     \unless\ifcat#2\relax
1226         \ifcsname XINT_expr_itself_#1#2\endcsname

```



```

1227         \XINT_expr_scanop_c
1228     \fi\fi
1229     \XINT_expr_foundop #1#2%
1230 }%
1231 \def\XINT_expr_foundop_a #1%
1232 {%
1233     \ifcsname XINT_expr_precedence_#1\endcsname
1234         \csname XINT_expr_precedence_#1\expandafter\endcsname
1235         \expandafter #1%
1236     \else
1237         \expandafter\XINT_expr_getop\romannumeral`&&@%
1238         \xint_afterfi{\XINT_expandableerror
1239             {Expected an operator but got `#1'. Ignoring.}}%
1240     \fi
1241 }%
1242 \def\XINT_expr_foundop #1{\csname XINT_expr_precedence_#1\endcsname #1}%

```

## 27.16. Expansion spanning; opening and closing parentheses

This is also where `\XINT_expr_start`, `\XINT_iiexpr_start` and `\XINT_flexpr_start` are defined.

These comments apply to all definitions coming next relative to execution of operations from parsing of syntax.

Refactored (and unified) at 1.4. In particular the 1.4 scheme uses `op`, `exec`, `check-`, and `checkp`. Formerly it was `until_a` (`check-`) and `until_b` (now split into `checkp` and `exec`).

This way neither `check-` nor `checkp` have to grab the accumulated number so far (top of stack if you like) and besides one never has to go back to `check-` from `checkp` (and neither from `check-`).

Prior to 1.4, accumulated intermediate results were stored as one token, but now we have to use `\expanded` to propagate expansion beyond possibly arbitrary long braced nested data. With the 1.4 refactoring we do this only once and only grab a second time the data if we actually have to act upon it.

Version 1.1 had a hack inside the `until` macros for handling the `omit` and `abort` in iterations over dummy variables. This has been removed by 1.2c, see the subsection where `omit` and `abort` are discussed.

Exceptionally, the `check-` is here abbreviated to `check`.

```

1243 \catcode`~ 11
1244 \def\XINT_tmpa #1#2#3#4#5#6%
1245 {%
1246     \def#1% \XINT_expr_start, \XINT_iiexpr_start, \XINT_flexpr_start
1247     {%
1248         \expandafter#2\romannumeral`&&@\XINT_expr_getnext
1249     }%
1250     \def#2##1% check
1251     {%
1252         \xint_UDsignfork
1253         ##1{\expandafter#3\romannumeral`&&@#4}%
1254         -{#3##1}%
1255     \krof
1256 }%
1257 \def#3##1##2% checkp
1258 {%
1259     \ifcase ##1%
1260         \expandafter\XINT_expr_done

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

1261      \or\expandafter#5%
1262      \else
1263      \expandafter#3\romannumeral`&&\csname XINT_#6_op_##2\expandafter\endcsname
1264      \fi
1265  }%
1266  \def#5%
1267  {%
1268      \XINT_expandableerror
1269      {Extra ) removed. Hit <return>, fingers crossed.}%
1270      \expandafter#2\romannumeral`&&\expandafter\XINT_expr_put_op_first
1271      \romannumeral`&&\XINT_expr_getop_legacy
1272  }%
1273 }%
1274 \let\XINT_expr_done\space
1275 \xintFor #1 in {expr,fexpr,iiexpr} \do {%
1276     \expandafter\XINT_tmpa
1277     \csname XINT_#1_start\expandafter\endcsname
1278     \csname XINT_#1_check\expandafter\endcsname
1279     \csname XINT_#1_checkp\expandafter\endcsname
1280     \csname XINT_#1_op_-xii\expandafter\endcsname
1281     \csname XINT_#1_extra_\expandafter\endcsname
1282     {#1}%
1283 }%

```

Here also we take some shortcuts relative to general philosophy and have no explicit exec macro.

```

1284 \def\XINT_tmpa #1#2#3#4#5#6#7%
1285 {%
1286     \def #1##1% op_(
1287     {%
1288         \expandafter #4\romannumeral`&&\XINT_expr_getnext
1289     }%
1290     \def #2##1% op_)
1291     {%
1292         \expanded{\xint_noxpd{\XINT_expr_put_op_first{##1}}\expandafter}%
1293         \romannumeral`&&\XINT_expr_getop
1294     }%
1295     \def #3% oparen
1296     {%
1297         \expandafter #4\romannumeral`&&\XINT_expr_getnext
1298     }%
1299     \def #4##1% check-
1300     {%
1301         \xint_UDsignfork
1302         ##1{\expandafter#5\romannumeral`&&#6}%
1303         -{#5##1}%
1304         \krof
1305     }%
1306     \def #5##1##2% checkp
1307     {%
1308         \ifcase ##1\expandafter\XINT_expr_missing_
1309         \or \csname XINT_#7_op_##2\expandafter\endcsname
1310         \else
1311         \expandafter #5\romannumeral`&&\csname XINT_#7_op_##2\expandafter\endcsname

```

```

1312     \fi
1313 }%
1314 }%
1315 \def\XINT_expr_missing_
1316   {\XINT_expandableerror{End of expression found, but some ) was missing there.}%
1317   \xint_c_ \XINT_expr_done }%
1318 \xintFor #1 in {expr,flexpr,iiexpr} \do {%
1319   \expandafter\XINT_tmpa
1320   \csname XINT_#1_op_(\expandafter\endcsname
1321   \csname XINT_#1_op_)\expandafter\endcsname
1322   \csname XINT_#1_oparen\expandafter\endcsname
1323   \csname XINT_#1_check_)\expandafter\endcsname
1324   \csname XINT_#1_checkp_)\expandafter\endcsname
1325   \csname XINT_#1_op_-xii\endcsname
1326   {#1}%
1327 }%
1328 \let\XINT_expr_precedence_)\xint_c_i
1329 \catcode` ) 12

```

## 27.17. The comma as binary operator

New with 1.09a. Refactored at 1.4.

```

1330 \def\XINT_tmpa #1#2#3#4#5#6%
1331 {%
1332   \def #1##1% \XINT_expr_op_,
1333   {%
1334     \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1335     \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1336   }%
1337   \def #2##1##2##3##4{##2##3{##1##4}}% \XINT_expr_exec_,
1338   \def #3##1% \XINT_expr_check_-,
1339   {%
1340     \xint_UDsignfork
1341     ##1{\expandafter#4\romannumeral`&&#5}%
1342     -{#4##1}%
1343     \krof
1344   }%
1345   \def #4##1##2% \XINT_expr_checkp_,
1346   {%
1347     \ifnum ##1>\xint_c_iii
1348       \expandafter#4%
1349       \romannumeral`&&\csname XINT_#6_op_#2\expandafter\endcsname
1350     \else
1351       \expandafter##1\expandafter##2%
1352     \fi
1353   }%
1354 }%
1355 \xintFor #1 in {expr,flexpr,iiexpr} \do {%
1356 \expandafter\XINT_tmpa
1357   \csname XINT_#1_op_,\expandafter\endcsname
1358   \csname XINT_#1_exec_,\expandafter\endcsname
1359   \csname XINT_#1_check_-, \expandafter\endcsname

```

```

1360 \csname XINT_#1_checkp_ \expandafter\endcsname
1361 \csname XINT_#1_op_-xii\endcsname {#1}%
1362 }%
1363 \expandafter\let\csname XINT_expr_precedence_ \endcsname\xint_c_iii

```

## 27.18. The minus as prefix operator of variable precedence level

Inherits the precedence level of the previous infix operator, if the latter has at least the precedence level of binary + and -, i.e. currently 12.

Refactored at 1.4.

At 1.4g I belatedly observe that I have been defining architecture for op\_-xvi but such operator can never be created, because there are no infix operators of precedence level 16. Perhaps in the past this was really needed? But now such 16 is precedence level of tacit multiplication which is implemented simply by the `\XINT_expr_prec_tacit` token, there is no macro check-\*\*\* which would need an op\_-xvi.

For the record: at least one scenario exists which creates tacit multiplication in front of a unary -, it is `2\count0` which first generates tacit multiplication then applies `\number` to `\count0`, but the operator is still \*, so this triggers only `\XINT_expr_op_-xiv`, not `-xvi`.

At 1.4g we need 17 and not 18 anymore as the precedence of unary minus following power operators ^ and \*\*. The needed `\xint_c_xvii` creation was added to `xintkernel.sty`.

```

1364 \def\XINT_tmppb #1#2#3#4#5#6#7%
1365 {%
1366 \def #1% \XINT_expr_op_-<level>
1367 {%
1368 \expandafter #2\romannumeral`&&\expandafter#3%
1369 \romannumeral`&&\XINT_expr_getnext
1370 }%
1371 \def #2##1##2##3% \XINT_expr_exec_-<level>
1372 {%
1373 \expandafter ##1\expandafter ##2\expandafter
1374 {%
1375 \romannumeral`&&\XINT:NEhook:f:one:from:one
1376 {\romannumeral`&&#7##3}%
1377 }%
1378 }%
1379 \def #3##1% \XINT_expr_check_-<level>
1380 {%
1381 \xint_UDsignfork
1382 ##1{\expandafter #4\romannumeral`&&#1}%
1383 -{#4##1}%
1384 \krof
1385 }%
1386 \def #4##1##2% \XINT_expr_checkp_-<level>
1387 {%
1388 \ifnum ##1>#5%
1389 \expandafter #4%
1390 \romannumeral`&&\csname XINT_#6_op_##2\expandafter\endcsname
1391 \else
1392 \expandafter ##1\expandafter ##2%
1393 \fi
1394 }%
1395 }%

```

```

1396 \def\XINT_tmpa #1#2#3%
1397 {%
1398   \expandafter\XINT_tmppb
1399   \csname XINT_#1_op_#3\expandafter\endcsname
1400   \csname XINT_#1_exec_#3\expandafter\endcsname
1401   \csname XINT_#1_check_#3\expandafter\endcsname
1402   \csname XINT_#1_checkp_#3\expandafter\endcsname
1403   \csname xint_c_#3\endcsname {#1}#2%
1404 }%
1405 \xintApplyInline{\XINT_tmpa {expr}\xintOpp}{\xii\xiv\xvii}%
1406 \xintApplyInline{\XINT_tmpa {fexpr}\xintOpp}{\xii\xiv\xvii}%
1407 \xintApplyInline{\XINT_tmpa {iexpr}\xintiiOpp}{\xii\xiv\xvii}%

```

## 27.19. The \* as Python-like «unpacking» prefix operator

New with 1.4. Prior to 1.4 the internal data structure was the one of `\csname` encapsulated comma separated numbers. No hierarchical structure was (easily) possible. At 1.4, we can use TeX braces because there is no detokenization to catcode 12.

```

1408 \def\XINT_tmpa#1#2#3%
1409 {%
1410   \def#1#1{\expandafter#2\romannumeral`&&\XINT_expr_getnext}%
1411   \def#2#1#2%
1412   {%
1413     \ifnum ##1>\xint_c_xx
1414       \expandafter #2%
1415       \romannumeral`&&\csname XINT_#3_op_##2\expandafter\endcsname
1416     \else
1417       \expandafter##1\expandafter##2\romannumeral0\expandafter\XINT:NEhook:unpack
1418     \fi
1419   }%
1420 }%
1421 \def\XINT:NEhook:unpack{\xint_stop_atfirstofone}%
1422 \xintFor* #1 in {\expr}{fexpr}{iexpr}:
1423   {\expandafter\XINT_tmpa\csname XINT_#1_op_0\expandafter\endcsname
1424     \csname XINT_#1_until_unpack\endcsname {#1}}%

```

## 27.20. Infix operators

27.20.1	&&,   , //, /:, +, -, *, /, ^, **, 'and', 'or', 'xor', and 'mod'	605
27.20.2	.., ..[, and ].. for a..b and a..[b]..c syntax	607
27.20.3	<, >, ==, <=, >=, != with Python-like chaining	609
27.20.4	Support macros for .., ..[ and ]..	610
	\xintSeq:tl:x	611
	\xintiiSeq:tl:x	611
	\xintSeqA, \xintiiSeqA	612
	\xintSeqB:tl:x	612
	\xintiiSeqB:tl:x	613

1.2d adds the \*\*\* for tying via tacit multiplication, for example  $x/2y$ . Actually I don't need the `_itself` mechanism for \*\*\*, only a precedence.

At 1.4b we must make sure that the ! in expansion of `\XINT_expr_itself_!` is of catcode 12 and not of catcode 11. This is because implementation of chaining of comparison operators proceeds

via inserting the `itself` macro directly into upcoming token stream, whereas formerly such `itself` macros would be expanded only in a `\csname...\endcsname` context.

```

1425 \catcode\& 12 \catcode\! 12
1426 \xintFor* #1 in {==}{!=}{<=}{>=}{&&}{||}{//}{/}{:}{.}{..}{[]}{.}{[]}{..}}%
1427     \do {\expandafter\def\csname XINT_expr_itself_#1\endcsname {#1}}%
1428 \catcode\& 7 \catcode\! 11

```

### 27.20.1. &&, ||, //, /:, +, -, \*, /, ^, \*\*, 'and', 'or', 'xor', and 'mod'

At 1.4g I finally decide to enact the switch to right associativity for the power operators `^` and `**`.

This goes via inserting into the `checkp` macros not anymore the precedence `chardef` token (which now only serves as left precedence, inserted in the token stream) but in its place an `\xint_c_<roman>` token holding the right precedence. Which is also transmitted to spanned unary minus operators.

Here only levels 12, 14, and 17 are created as right precedences.

#6 and #7 got permuted and the new #7 is directly a control sequence. Also #3 and #4 are now integers which need `\romannumeral`. The change in `\XINT_expr_defbin_c` does not propagate as it is re-defined shortly thereafter.

```

1429 \def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8%
1430 {%
1431     \def #1#1% \XINT_expr_op_<op>
1432     {%
1433         \expanded{\xint_noexpd{#2{#1}}\expandafter}%
1434         \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1435     }%
1436     \def #2#1#2#3#4% \XINT_expr_exec_<op>
1437     {%
1438         \expandafter##2\expandafter##3\expandafter
1439         {\romannumeral`&&\XINT:NEhook:f:one:from:two{\romannumeral`&&@#7#1#4}}%
1440     }%
1441     \def #3#1% \XINT_expr_check_-_<op>
1442     {%
1443         \xint_UDsignfork
1444         ##1{\expandafter#4\romannumeral`&&@#5}%
1445         -{#4#1}%
1446         \krof
1447     }%
1448     \def #4#1#2% \XINT_expr_checkp_<op>
1449     {%
1450         \ifnum #1>#6%
1451             \expandafter#4%
1452             \romannumeral`&&\csname XINT_#8_op_#2\endcsname\expandafter\endcsname
1453         \else
1454             \expandafter #1\expandafter #2%
1455         \fi
1456     }%
1457 }%
1458 \def\XINT_expr_defbin_b #1#2#3#4#5%
1459 {%
1460     \expandafter\XINT_expr_defbin_c
1461     \csname XINT_#1_op_#2\endcsname\expandafter\endcsname

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

```

1462 \csname XINT_#1_exec_#2\expandafter\endcsname
1463 \csname XINT_#1_check_#2\expandafter\endcsname
1464 \csname XINT_#1_checkp_#2\expandafter\endcsname
1465 \csname XINT_#1_op_-\romannumeral\ifnum#4>12 #4\else12\fi\expandafter\endcsname
1466 \csname xint_c_\romannumeral#4\endcsname
1467 #5%
1468 {#1}%
1469 \expandafter % done 3 times but well
1470 \let\csname XINT_expr_precedence_#2\expandafter\endcsname
1471 \csname xint_c_\romannumeral#3\endcsname
1472 }%
1473 \XINT_expr_defbin_b {expr} {||} {6} {6} \xintOR
1474 \XINT_expr_defbin_b {fexpr}{||} {6} {6} \xintOR
1475 \XINT_expr_defbin_b {iiexpr}{||} {6} {6} \xintOR
1476 \catcode`& 12
1477 \XINT_expr_defbin_b {expr} {&&} {8} {8} \xintAND
1478 \XINT_expr_defbin_b {fexpr}{&&} {8} {8} \xintAND
1479 \XINT_expr_defbin_b {iiexpr}{&&} {8} {8} \xintAND
1480 \catcode`& 7
1481 \XINT_expr_defbin_b {expr} {xor}{6} {6} \xintXOR
1482 \XINT_expr_defbin_b {fexpr}{xor}{6} {6} \xintXOR
1483 \XINT_expr_defbin_b {iiexpr}{xor}{6} {6} \xintXOR
1484 \XINT_expr_defbin_b {expr} {//} {14}{14}\xintDivFloor
1485 \XINT_expr_defbin_b {fexpr}{//} {14}{14}\XINTinFloatDivFloor
1486 \XINT_expr_defbin_b {iiexpr}{//} {14}{14}\xintiiDivFloor
1487 \XINT_expr_defbin_b {expr} {/:} {14}{14}\xintMod
1488 \XINT_expr_defbin_b {fexpr}{/:} {14}{14}\XINTinFloatMod
1489 \XINT_expr_defbin_b {iiexpr}{/:} {14}{14}\xintiiMod
1490 \XINT_expr_defbin_b {expr} + {12}{12}\xintAdd
1491 \XINT_expr_defbin_b {fexpr} + {12}{12}\XINTinFloatAdd
1492 \XINT_expr_defbin_b {iiexpr} + {12}{12}\xintiiAdd
1493 \XINT_expr_defbin_b {expr} - {12}{12}\xintSub
1494 \XINT_expr_defbin_b {fexpr} - {12}{12}\XINTinFloatSub
1495 \XINT_expr_defbin_b {iiexpr} - {12}{12}\xintiiSub
1496 \XINT_expr_defbin_b {expr} * {14}{14}\xintMul
1497 \XINT_expr_defbin_b {fexpr} * {14}{14}\XINTinFloatMul
1498 \XINT_expr_defbin_b {iiexpr} * {14}{14}\xintiiMul
1499 \let\XINT_expr_prec_tacit \xint_c_xvi
1500 \XINT_expr_defbin_b {expr} / {14}{14}\xintDiv
1501 \XINT_expr_defbin_b {fexpr} / {14}{14}\XINTinFloatDiv
1502 \XINT_expr_defbin_b {iiexpr} / {14}{14}\xintiiDivRound

```

At 1.4g, right associativity is implemented via a lowered right precedence here.

```

1503 \XINT_expr_defbin_b {expr} ^ {18}{17}\xintPow
1504 \XINT_expr_defbin_b {fexpr} ^ {18}{17}\XINTinFloatSciPow
1505 \XINT_expr_defbin_b {iiexpr} ^ {18}{17}\xintiiPow

```

1.4g This is a trick (which was in old version of bnumexpr, I wonder why I did not have it here) but it will make error messages in case of \*\*<token> confusing. The ^ here is of catcode 11 but it does not matter.

```

1506 \expandafter\def\csname XINT_expr_itself_**\endcsname{^}%
1507 \catcode`& 12

```

For this which contributes to implementing 'and', 'or', etc... see `\XINT_expr_binopwrdr`.

```

1508 \xintFor #1 in {and,or,xor,mod} \do
1509 {%
1510   \expandafter\def\csname XINT_expr_itself_#1\endcsname {#1}%
1511 }%
1512 \expandafter\let\csname XINT_expr_precedence_and\expandafter\endcsname
1513   \csname XINT_expr_precedence_&\endcsname
1514 \expandafter\let\csname XINT_expr_precedence_or\expandafter\endcsname
1515   \csname XINT_expr_precedence_||\endcsname
1516 \expandafter\let\csname XINT_expr_precedence_mod\expandafter\endcsname
1517   \csname XINT_expr_precedence_/\endcsname
1518 \xintFor #1 in {expr, flexpr, iexpr} \do
1519 {%
1520   \expandafter\let\csname XINT_#1_op_and\expandafter\endcsname
1521     \csname XINT_#1_op_&\endcsname
1522   \expandafter\let\csname XINT_#1_op_or\expandafter\endcsname
1523     \csname XINT_#1_op_||\endcsname
1524   \expandafter\let\csname XINT_#1_op_mod\expandafter\endcsname
1525     \csname XINT_#1_op_/\endcsname
1526 }%
1527 \catcode`& 7

```

### 27.20.2. .., ..[, and ].. for a..b and a..[b]..c syntax

The 1.4 `exec_...` macros (which do no further expansion!) had silly `\expandafter` doing nothing for the sole reason of sharing a common `\XINT_expr_defbin_c` as used previously for the `+`, `-` etc... operators. At 1.4b we take the time to set things straight and do other similar simplifications.

```

1528 \def\XINT_expr_defbin_c #1#2#3#4#5#6#7%
1529 {%
1530   \def #1##1% \XINT_expr_op_..[
1531   {%
1532     \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1533     \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1534   }%
1535   \def #2##1##2##3##4% \XINT_expr_exec_..[
1536   {%
1537     ##2##3{{##1##4}}%
1538   }%
1539   \def #3##1% \XINT_expr_check_...[
1540   {%
1541     \xint_UDsignfork
1542     ##1{\expandafter#4\romannumeral`&&#5}%
1543     -{#4##1}%
1544     \krof
1545   }%
1546   \def #4##1##2% \XINT_expr_checkp_..[
1547   {%
1548     \ifnum ##1>#6%
1549       \expandafter#4%
1550       \romannumeral`&&\csname XINT_#7_op_#2\expandafter\endcsname
1551     \else
1552       \expandafter ##1\expandafter ##2%
1553     \fi
1554   }%

```



## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
1555 }%
1556 \def\XINT_expr_defbin_b #1%
1557 {%
1558   \expandafter\XINT_expr_defbin_c
1559   \csname XINT_#1_op_..[\expandafter\endcsname
1560   \csname XINT_#1_exec_..[\expandafter\endcsname
1561   \csname XINT_#1_check-..[\expandafter\endcsname
1562   \csname XINT_#1_checkp_..[\expandafter\endcsname
1563   \csname XINT_#1_op_-xi\expandafter\endcsname
1564   \csname XINT_expr_precedence_..[\endcsname
1565   {#1}%
1566 }%
1567 \XINT_expr_defbin_b {expr}%
1568 \XINT_expr_defbin_b {fexpr}%
1569 \XINT_expr_defbin_b {iexpr}%
1570 \expandafter\let\csname XINT_expr_precedence_..[\endcsname\xint_c_vi
1571 \def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8%
1572 {%
1573   \def #1##1% \XINT_expr_op_<op>
1574   {%
1575     \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1576     \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1577   }%
1578   \def #2##1##2##3##4% \XINT_expr_exec_<op>
1579   {%
1580     \expandafter##2\expandafter##3\expanded
1581     {{\XINT:NEhook:x:one:from:two#8##1##4}}%
1582   }%
1583   \def #3##1% \XINT_expr_check_-<op>
1584   {%
1585     \xint_UDsignfork
1586     ##1{\expandafter#4\romannumeral`&&#5}%
1587     -{#4##1}%
1588     \krof
1589   }%
1590   \def #4##1##2% \XINT_expr_checkp_<op>
1591   {%
1592     \ifnum ##1>#6%
1593       \expandafter#4%
1594       \romannumeral`&&\csname XINT_#7_op_##2\expandafter\endcsname
1595     \else
1596       \expandafter ##1\expandafter ##2%
1597     \fi
1598   }%
1599 }%
1600 \def\XINT_expr_defbin_b #1#2#3%
1601 {%
1602   \expandafter\XINT_expr_defbin_c
1603   \csname XINT_#1_op_#2\expandafter\endcsname
1604   \csname XINT_#1_exec_#2\expandafter\endcsname
1605   \csname XINT_#1_check_-#2\expandafter\endcsname
1606   \csname XINT_#1_checkp_#2\expandafter\endcsname
```

```

1607 \csname XINT_#1_op_-xii\expandafter\endcsname
1608 \csname XINT_expr_precedence_#2\endcsname
1609 {#1}#3%
1610 \expandafter\let
1611 \csname XINT_expr_precedence_#2\expandafter\endcsname\xint_c_vi
1612 }%
1613 \XINT_expr_defbin_b {expr} {..}\xintSeq:tl:x
1614 \XINT_expr_defbin_b {fexpr} {..}\xintSeq:tl:x
1615 \XINT_expr_defbin_b {iexpr} {..}\xintiSeq:tl:x
1616 \XINT_expr_defbin_b {expr} {[]}\xintSeqB:tl:x
1617 \XINT_expr_defbin_b {fexpr}{[]}\xintSeqB:tl:x
1618 \XINT_expr_defbin_b {iexpr}{[]}\xintiSeqB:tl:x

```

### 27.20.3. <, >, ==, <=, >=, != with Python-like chaining

1.4b This is preliminary implementation of chaining of comparison operators like Python and (I think) l3fp do. I am not too happy with how many times the (second) operand (already evaluated) is fetched.

```

1619 \def\XINT_expr_defbin_d #1#2%
1620 {%
1621 \def #1##1##2##3##4% \XINT_expr_exec_<op>
1622 {%
1623 \expandafter##2\expandafter##3\expandafter
1624 {\romannumeral`&&\XINT:NEhook:f:one:from:two{\romannumeral`&&@#2##1##4}}%
1625 }%
1626 }%
1627 \def\XINT_expr_defbin_c #1#2#3#4#5#6#7#8#9%
1628 {%
1629 \def #1##1% \XINT_expr_op_<op>
1630 {%
1631 \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1632 \romannumeral`&&\expandafter#7%
1633 \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1634 }%
1635 \def #3##1% \XINT_expr_check_<op>
1636 {%
1637 \xint_UDsignfork
1638 ##1{\expandafter#4\romannumeral`&&@#5}%
1639 -{#4##1}%
1640 \krof
1641 }%
1642 \def #4##1##2% \XINT_expr_checkp_<op>
1643 {%
1644 \ifnum ##1>#6%
1645 \expandafter#4%
1646 \romannumeral`&&\csname XINT_#9_op_##2\expandafter\endcsname
1647 \else
1648 \expandafter ##1\expandafter ##2%
1649 \fi
1650 }%
1651 \let #6\xint_c_x
1652 \def #7##1% \XINT_expr_checkc_<op>
1653 {%

```

```

1654 \ifnum ##1=\xint_c_x\expandafter#8\fi ##1%
1655 }%
1656 \edef #8##1##2##3% \XINT_expr_execc_<op>
1657 {%
1658 \csname XINT_#9_precedence_\string&\string&\endcsname
1659 \expandafter\noexpand\csname XINT_#9_itself_\string&\string&\endcsname
1660 {##3}%
1661 \XINTfstop.{##3}##2%
1662 }%
1663 \XINT_expr_defbin_d #2% \XINT_expr_exec_<op>
1664 }%
1665 \def\XINT_expr_defbin_b #1#2#3%
1666 {%
1667 \expandafter\XINT_expr_defbin_c
1668 \csname XINT_#1_op_#2\expandafter\endcsname
1669 \csname XINT_#1_exec_#2\expandafter\endcsname
1670 \csname XINT_#1_check_#2\expandafter\endcsname
1671 \csname XINT_#1_checkp_#2\expandafter\endcsname
1672 \csname XINT_#1_op_#2\expandafter\endcsname
1673 \csname XINT_expr_precedence_#2\expandafter\endcsname
1674 \csname XINT_#1_checkc_#2\expandafter\endcsname
1675 \csname XINT_#1_execc_#2\endcsname
1676 {#1}%#3%
1677 }%

```

Attention that third token here is left in stream by `defbin_b`, then also by `defbin_c` and is picked up as #2 of `defbin_d`. Had to work around TeX accepting only 9 arguments. Why did it not start counting at #0 like all decent mathematicians do?

```

1678 \XINT_expr_defbin_b {expr} <\xintLt
1679 \XINT_expr_defbin_b {flexpr}<\xintLt
1680 \XINT_expr_defbin_b {iiexpr}<\xintiilt
1681 \XINT_expr_defbin_b {expr} >\xintGt
1682 \XINT_expr_defbin_b {flexpr}>\xintGt
1683 \XINT_expr_defbin_b {iiexpr}>\xintiiGt
1684 \XINT_expr_defbin_b {expr} {==}\xintEq
1685 \XINT_expr_defbin_b {flexpr}{==}\xintEq
1686 \XINT_expr_defbin_b {iiexpr}{==}\xintiiEq
1687 \XINT_expr_defbin_b {expr} {<=}\xintLtorEq
1688 \XINT_expr_defbin_b {flexpr}{<=}\xintLtorEq
1689 \XINT_expr_defbin_b {iiexpr}{<=}\xintiiLtorEq
1690 \XINT_expr_defbin_b {expr} {>=}\xintGtorEq
1691 \XINT_expr_defbin_b {flexpr}{>=}\xintGtorEq
1692 \XINT_expr_defbin_b {iiexpr}{>=}\xintiiGtorEq
1693 \XINT_expr_defbin_b {expr} {!=}\xintNotEq
1694 \XINT_expr_defbin_b {flexpr}{!=}\xintNotEq
1695 \XINT_expr_defbin_b {iiexpr}{!=}\xintiiNotEq

```

#### 27.20.4. Support macros for .., ..[ and ]..

\xintSeq:tl:x . . . . .	611
\xintiiSeq:tl:x . . . . .	611
\xintSeqA, \xintiiSeqA . . . . .	612
\xintSeqB:tl:x . . . . .	612

\xintiiSeqB:tl:x . . . . . 613

**\xintSeq:tl:x** Commence par remplacer a par ceil(a) et b par floor(b) et renvoie ensuite les entiers entre les deux, possiblement en décroissant, et extrémités comprises. Si a=b est non entier en obtient donc ceil(a) et floor(a). Ne renvoie jamais une liste vide.

Note: le a..b dans **\xintfloatexpr** utilise cette routine.

```

1696 \def\xintSeq:tl:x #1#2%
1697 {%
1698   \expandafter\xINT_Seq:tl:x
1699   \the\numexpr \xintiCeil{#1}\expandafter.\the\numexpr \xintiFloor{#2}.%
1700 }%
1701 \def\xINT_Seq:tl:x #1.#2.%
1702 {%
1703   \ifnum #2=#1 \xint_dothis\xINT_Seq:tl:x_z\fi
1704   \ifnum #2<#1 \xint_dothis\xINT_Seq:tl:x_n\fi
1705   \xint_orthat\xINT_Seq:tl:x_p
1706   #1.#2.%
1707 }%
1708 \def\xINT_Seq:tl:x_z #1.#2.{#{#1/1[0]}}%
1709 \def\xINT_Seq:tl:x_p #1.#2.%
1710 {%
1711   {#{#1/1[0]}}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
1712   \expandafter\xINT_Seq:tl:x_p \the\numexpr #1+\xint_c_i.#2.%
1713 }%
1714 \def\xINT_Seq:tl:x_n #1.#2.%
1715 {%
1716   {#{#1/1[0]}}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
1717   \expandafter\xINT_Seq:tl:x_n \the\numexpr #1-\xint_c_i.#2.%
1718 }%
1719 \def\xINT_Seq:tl:x_e#1#2.#3.{#{#1}}%

```

**\xintiiSeq:tl:x**

```

1720 \def\xintiiSeq:tl:x #1#2%
1721 {%
1722   \expandafter\xINT_iiSeq:tl:x
1723   \the\numexpr \xintiCeil{#1}\expandafter.\the\numexpr \xintiFloor{#2}.%
1724 }%
1725 \def\xINT_iiSeq:tl:x #1.#2.%
1726 {%
1727   \ifnum #2=#1 \xint_dothis\xINT_iiSeq:tl:x_z\fi
1728   \ifnum #2<#1 \xint_dothis\xINT_iiSeq:tl:x_n\fi
1729   \xint_orthat\xINT_iiSeq:tl:x_p
1730   #1.#2.%
1731 }%
1732 \def\xINT_iiSeq:tl:x_z #1.#2.{#{#1}}%
1733 \def\xINT_iiSeq:tl:x_p #1.#2.%
1734 {%
1735   {#{#1}}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
1736   \expandafter\xINT_iiSeq:tl:x_p \the\numexpr #1+\xint_c_i.#2.%
1737 }%
1738 \def\xINT_iiSeq:tl:x_n #1.#2.%
1739 {%

```

```
1740 {#1}\ifnum #1=#2 \XINT_Seq:tl:x_e\fi
1741 \expandafter\XINT_iiSeq:tl:x_n \the\numexpr #1-\xint_c_i.#2.%
1742 }%
```

Contrarily to `a..b` which is limited to small integers, this works with `a`, `b`, and `d` (big) fractions. It will produce a «nil» list, if `a>b` and `d<0` or `a<b` and `d>0`.

#### `\xintSeqA`, `\xintiiSeqA`

```
1743 \def\xintSeqA      {\expandafter\XINT_SeqA\romannumeral0\xintra}
1744 \def\xintiiSeqA    #1{\expandafter\XINT_iiSeqA\romannumeral`&&@#1;}
1745 \def\XINT_SeqA    #1]#2{\expandafter\XINT_SeqA_a\romannumeral0\xintra {#2}#1]}
1746 \def\XINT_iiSeqA#1;#2{\expandafter\XINT_SeqA_a\romannumeral`&&@#2;#1;}
1747 \def\XINT_SeqA_a #1{\xint_UDzerominusfork
1748                      #1-{z}%
1749                      0#1{n}%
1750                      0-{p}%
1751 \krof #1}%
```

`\xintSeqB:tl:x` At 1.4, delayed expansion of start and step done here and not before, for matters of `\xintdefunc` and «NEhooks».

The float variant at 1.4 is made identical to the exact variant. I.e. stepping is exact and comparison to the range limit too. But recall that `a/b` input will be converted to a float. To handle `1/3` step for example still better to use `\xintexpr 1..1/3..10\relax` for example inside the `\xintfloateval`.

```
1752 \def\xintSeqB:tl:x #1{\expandafter\XINT_SeqB:tl:x\romannumeral`&&@\xintSeqA#1}%
1753 \def\XINT_SeqB:tl:x #1{\csname XINT_SeqB#1:tl:x\endcsname}%
1754 \def\XINT_SeqBz:tl:x #1]#2]#3{{#2}}}%
1755 \def\XINT_SeqBp:tl:x #1]#2]#3%
1756     {\expandafter\XINT_SeqBp:tl:x_a\romannumeral0\xintra{#3}#2]#1]}
1757 \def\XINT_SeqBp:tl:x_a #1]#2]#3}%
1758 {%
1759     \xintifCmp{#1}]{#2}}%
1760     {{{#2}}}{#{2}}\expandafter\XINT_SeqBp:tl:x_b\romannumeral0\xintadd{#3}]{#2}]{#1}#3}}%
1761 }%
1762 \def\XINT_SeqBp:tl:x_b #1]#2]#3}%
1763 {%
1764     \xintifCmp{#1}]{#2}}%
1765     {#{1}}\expandafter\XINT_SeqBp:tl:x_b\romannumeral0\xintadd{#3}]{#1}]{#2}]{#3}]{#{1}}}%
1766 }%
1767 \def\XINT_SeqBn:tl:x #1]#2]#3%
1768     {\expandafter\XINT_SeqBn:tl:x_a\romannumeral0\xintra{#3}#2]#1]}
1769 \def\XINT_SeqBn:tl:x_a #1]#2]#3}%
1770 {%
1771     \xintifCmp{#1}]{#2}}%
1772     {#{2}}\expandafter\XINT_SeqBn:tl:x_b\romannumeral0\xintadd{#3}]{#2}]{#1}#3}]{#{2}}}%
1773 }%
1774 \def\XINT_SeqBn:tl:x_b #1]#2]#3}%
1775 {%
1776     \xintifCmp{#1}]{#2}}%
1777     {{{#1}}}{#{1}}\expandafter\XINT_SeqBn:tl:x_b\romannumeral0\xintadd{#3}]{#1}]{#2}]{#3}}%
1778 }%
```

**\xintiiSeqB:tl:x**

```

1779 \def\xintiiSeqB:tl:x #1{\expandafter\XINT_iiSeqB:tl:x\romannumeral`&&\xintiiSeqA#1}%
1780 \def\XINT_iiSeqB:tl:x #1{\csname XINT_iiSeqB#1:tl:x\endcsname}%
1781 \def\XINT_iiSeqBz:tl:x #1;#2;#3{{#2}}%
1782 \def\XINT_iiSeqBp:tl:x #1;#2;#3{\expandafter\XINT_iiSeqBp:tl:x_a\romannumeral`&&@#3;#2;#1;%
1783 \def\XINT_iiSeqBp:tl:x_a #1;#2;#3;%
1784 {%
1785     \xintiifCmp{#1}{#2}%
1786     {{{#2}}}{#2}\expandafter\XINT_iiSeqBp:tl:x_b\romannumeral0\xintiadd{#3}{#2};#1;#3;%
1787 }%
1788 \def\XINT_iiSeqBp:tl:x_b #1;#2;#3;%
1789 {%
1790     \xintiifCmp{#1}{#2}%
1791     {#1}\expandafter\XINT_iiSeqBp:tl:x_b\romannumeral0\xintiadd{#3}{#1};#2;#3;{{{#1}}}%
1792 }%
1793 \def\XINT_iiSeqBn:tl:x #1;#2;#3{\expandafter\XINT_iiSeqBn:tl:x_a\romannumeral`&&@#3;#2;#1;%
1794 \def\XINT_iiSeqBn:tl:x_a #1;#2;#3;%
1795 {%
1796     \xintiifCmp{#1}{#2}%
1797     {#2}\expandafter\XINT_iiSeqBn:tl:x_b\romannumeral0\xintiadd{#3}{#2};#1;#3;{{{#2}}}%
1798 }%
1799 \def\XINT_iiSeqBn:tl:x_b #1;#2;#3;%
1800 {%
1801     \xintiifCmp{#1}{#2}%
1802     {{{#1}}}{#1}\expandafter\XINT_iiSeqBn:tl:x_b\romannumeral0\xintiadd{#3}{#1};#2;#3;%
1803 }%

```

**27.21. Square brackets [ ] both as a container and a Python slicer**

Refactored at 1.4

The architecture allows to implement separately a «left» and a «right» precedence and this is crucial.

27.21.1	[...] as «oneple» constructor . . . . .	613
27.21.2	[...] brackets and : operator for NumPy-like slicing and item indexing syntax . . . . .	614
27.21.3	Macro layer implementing indexing and slicing . . . . .	616

**27.21.1. [...] as «oneple» constructor**

In the definition of `\XINT_expr_op_obracket` the parameter is trash `{}`. The `[` is intercepted by the `getnextfork` and handled via the `\xint_c_ii^v` highest precedence trick to get `op_obracket` executed.

```

1804 \def\XINT_expr_itself_obracket{obracket}%
1805 \catcode`[ 11 \catcode`\[ 11
1806 \def\XINT_expr_defbin_c #1#2#3#4#5#6%
1807 {%
1808     \def #1##1%
1809     {%
1810         \expandafter#3\romannumeral`&&\XINT_expr_getnext
1811     }%
1812     \def #2##1% op_]
1813     {%
1814         \expanded{\xint_noxpd{\XINT_expr_put_op_first{{{#1}}}}\expandafter}%

```

```

1815     \romannumeral`&&@\XINT_expr_getop
1816 }%
1817 \def #3##1%  until_cbracket_a
1818 {%
1819     \xint_UDsignfork
1820     ##1{\expandafter#4\romannumeral`&&@#5}% #5 = op_-xii
1821     -{#4##1}%
1822     \krof
1823 }%
1824 \def #4##1##2%  until_cbracket_b
1825 {%
1826     \ifcase ##1\expandafter\XINT_expr_missing_]
1827     \or \expandafter\XINT_expr_missing_]
1828     \or \expandafter#2%
1829     \else
1830     \expandafter #4%
1831     \romannumeral`&&\csname XINT_#6_op_##2\expandafter\endcsname
1832     \fi
1833 }%
1834 }%
1835 \def\XINT_expr_defbin_b #1%
1836 {%
1837     \expandafter\XINT_expr_defbin_c
1838     \csname XINT_#1_op_obracket\expandafter\endcsname
1839     \csname XINT_#1_op_]\expandafter\endcsname
1840     \csname XINT_#1_until_cbracket_a\expandafter\endcsname
1841     \csname XINT_#1_until_cbracket_b\expandafter\endcsname
1842     \csname XINT_#1_op_-xii\endcsname
1843     {#1}%
1844 }%
1845 \XINT_expr_defbin_b {expr}%
1846 \XINT_expr_defbin_b {flexpr}%
1847 \XINT_expr_defbin_b {iiexpr}%
1848 \def\XINT_expr_missing_]
1849     {\XINT_expandableerror{0oops, looks like we are missing a ]. Aborting!}%
1850     \xint_c_ \XINT_expr_done}%
1851 \let\XINT_expr_precedence_]\xint_c_ii

```

### 27.21.2. [...] brackets and : operator for NumPy-like slicing and item indexing syntax

The opening bracket [ for the nutple constructor is filtered out by `\XINT_expr_getnextfork` and becomes «obracket» which behaves with precedence level 2. For the [...] Python slicer on the other hand, a real operator [ is defined with precedence level 4 (it must be higher than precedence level of commas) on its right and maximal precedence on its left.

Important: although slicing and indexing shares many rules with Python/NumPy there are some significant differences: in particular there can not be any out-of-range error generated, slicing applies also to «oples» and not only to «nutple», and nested lists do not have to have their leaves at a constant depth. See the user manual.

Currently, NumPy-like nested (basic) slicing is implemented, i.e [a:b, c:d, N, e:f, M] type syntax with Python rules regarding negative integers. This is parsed as an expression and can arise from expansion or contain calculations.

Currently stepping, Ellipsis, and simultaneous multi-index extracting are not yet implemented. There are some subtle things here with possibility of variables been passed by reference.

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

1852 \def\XINT_expr_defbin_c #1#2#3#4#5#6%
1853 {%
1854   \def #1##1% \XINT_expr_op_[
1855   {%
1856     \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1857     \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1858   }%
1859   \def #2##1##2##3##4% \XINT_expr_exec_]
1860   {%
1861     \expandafter\XINT_expr_put_op_first
1862     \expanded
1863     {%
1864       {\XINT:NEhook:x:lists\XINT_ListSel_top ##1__##4&({##1}\expandafter}%
1865       \expandafter
1866     }%
1867     \romannumeral`&&\XINT_expr_getop
1868   }%
1869   \def #3##1% \XINT_expr_check-]
1870   {%
1871     \xint_UDsignfork
1872     ##1{\expandafter#4\romannumeral`&&@#5}%
1873     -{#4##1}%
1874     \krof
1875   }%
1876   \def #4##1##2% \XINT_expr_checkp_]
1877   {%
1878     \ifcase ##1\XINT_expr_missing_]
1879     \or \XINT_expr_missing_]
1880     \or \expandafter##1\expandafter##2%
1881     \else \expandafter#4%
1882       \romannumeral`&&\csname XINT_#6_op_##2\expandafter\endcsname
1883     \fi
1884   }%
1885 }%
1886 \let\XINT_expr_precedence_ \xint_c_xx
1887 \def\XINT_expr_defbin_b #1%
1888 {%
1889   \expandafter\XINT_expr_defbin_c
1890   \csname XINT_#1_op_\expandafter\endcsname
1891   \csname XINT_#1_exec_\expandafter\endcsname
1892   \csname XINT_#1_check-_\expandafter\endcsname
1893   \csname XINT_#1_checkp_\expandafter\endcsname
1894   \csname XINT_#1_op_-xii\endcsname
1895   {#1}%
1896 }%
1897 \XINT_expr_defbin_b {expr}%
1898 \XINT_expr_defbin_b {flexpr}%
1899 \XINT_expr_defbin_b {iiexpr}%
1900 \catcode` ] 12 \catcode` [ 12

```

At 1.4 the getnext, scanint, scanfunc, getop chain got revisited to trigger automatic insertion of the nil variable if needed, without having in situations like here to define operators to support «[:» or «:]». And as we want to implement nested slicing à la NumPy, we would have had to handle



also «:»,» for example. Thus here we simply have to define the sole operator «:» and it will be some sort of inert joiner preparing a slicing spec.

```

1901 \def\XINT_expr_defbin_c #1#2#3#4#5#6%
1902 {%
1903   \def #1##1% \XINT_expr_op_:
1904     {%
1905       \expanded{\xint_noexpd{#2{##1}}\expandafter}%
1906       \romannumeral`&&\expandafter#3\romannumeral`&&\XINT_expr_getnext
1907     }%
1908   \def #2##1##2##3##4% \XINT_expr_exec_:
1909     {%
1910       ##2##3{:##1{0};##4:_}%
1911     }%
1912   \def #3##1% \XINT_expr_check-_:
1913     {\xint_UDsignfork
1914       ##1{\expandafter#4\romannumeral`&&@#5}%
1915       -{#4##1}%
1916     \krof
1917   }%
1918   \def #4##1##2% \XINT_expr_checkp_:
1919     {%
1920       \ifnum ##1>\XINT_expr_precedence_:
1921         \expandafter #4\romannumeral`&&@%
1922         \csname XINT_#6_op_##2\expandafter\endcsname
1923       \else
1924         \expandafter##1\expandafter##2%
1925       \fi
1926     }%
1927   }%
1928 \let\XINT_expr_precedence_:\xint_c_vi
1929 \def\XINT_expr_defbin_b #1%
1930 {%
1931   \expandafter\XINT_expr_defbin_c
1932   \csname XINT_#1_op_:\expandafter\endcsname
1933   \csname XINT_#1_exec_:\expandafter\endcsname
1934   \csname XINT_#1_check-:\expandafter\endcsname
1935   \csname XINT_#1_checkp_:\expandafter\endcsname
1936   \csname XINT_#1_op_-xii\endcsname {#1}%
1937 }%
1938 \XINT_expr_defbin_b {expr}%
1939 \XINT_expr_defbin_b {fexpr}%
1940 \XINT_expr_defbin_b {iexpr}%

```

### 27.21.3. Macro layer implementing indexing and slicing

*xintexpr* applies slicing not only to «objects» (which can be passed as arguments to functions) but also to «oples».

Our «nlists» are not necessarily regular N-dimensional arrays à la NumPy. Leaves can be at arbitrary depths. If we were handling regular «ndarrays», we could proceed a bit differently.

For the related explanations, refer to the user manual.

Notice that currently the code uses f-expandable (and not using `\expanded`) macros `\xintApply`, `\xintApplyUnbraced`, `\xintKeep`, `\xintTrim`, `\xintNthOne` from *xinttools*.

But the whole expansion happens inside an `\expanded` context, so possibly some gain could be achieved with x-expandable variants (`xintexpr < 1.4` had an `\xintKeep:x:csv`).

I coded `\xintApply:x` and `\xintApplyUnbraced:x` in *xinttools*, Brief testing indicated they were perhaps a bit better for 5x5x5x5 and 15x15x15x15 arrays of 8 digits numbers and for 30x30x15 with 16 digits numbers: say 1% gain... this seems to raise to between 4% and 5% for 400x400 array of 1 digit...

Currently sticking with old macros.

```

1941 \def\xINT_ListSel_deeper #1%
1942 {%
1943   \if :#1\xint_dothis\xINT_ListSel_slice_next\fi
1944   \xint_orthat {\XINT_ListSel_extract_next {#1}}%
1945 }%
1946 \def\xINT_ListSel_slice_next #1(%
1947 {%
1948   \xintApply{\XINT_ListSel_recurse{:#1}}%
1949 }%
1950 \def\xINT_ListSel_extract_next #1(%
1951 {%
1952   \xintApplyUnbraced{\XINT_ListSel_recurse{#1}}%
1953 }%
1954 \def\xINT_ListSel_recurse #1#2%
1955 {%
1956   \XINT_ListSel_check #2__#1({#2})\expandafter\empty\empty
1957 }%
1958 \def\xINT_ListSel_check{\expandafter\xINT_ListSel_check_a \string}%
1959 \def\xINT_ListSel_check_a #1%
1960 {%
1961   \if #1\bgroup\xint_dothis\xINT_ListSel_check_is_ok\fi
1962   \xint_orthat\xINT_ListSel_check_leaf
1963 }%
1964 \def\xINT_ListSel_check_leaf #1\expandafter{\expandafter}%
1965 \def\xINT_ListSel_check_is_ok
1966 {%
1967   \expandafter\xINT_ListSel_check_is_ok_a\expandafter{\string}%
1968 }%
1969 \def\xINT_ListSel_check_is_ok_a #1__#2%
1970 {%
1971   \if :#2\xint_dothis{\XINT_ListSel_slice}\fi
1972   \xint_orthat {\XINT_ListSel_nthone {#2}}%
1973 }%
1974 \def\xINT_ListSel_top #1#2%
1975 {%
1976   \if _\noexpand#2%
1977     \expandafter\xINT_ListSel_top_one_or_none\string#1.\else
1978     \expandafter\xINT_ListSel_top_at_least_two\fi
1979 }%
1980 \def\xINT_ListSel_top_at_least_two #1__{\XINT_ListSel_top_ople}%
1981 \def\xINT_ListSel_top_one_or_none #1%
1982 {%
1983   \if #1_\xint_dothis\xINT_ListSel_top_nil\fi
1984   \if #1.\xint_dothis\xINT_ListSel_top_nutple_a\fi
1985   \if #1\bgroup\xint_dothis\xINT_ListSel_top_nutple\fi

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

1986 \xint_orthat\xINT_ListSel_top_number
1987 }%
1988 \def\xINT_ListSel_top_nil #1\expandafter#2\expandafter{\fi\expandafter}%
1989 \def\xINT_ListSel_top_nutple
1990 {%
1991 \expandafter\xINT_ListSel_top_nutple_a\expandafter{\string}%
1992 }%
1993 \def\xINT_ListSel_top_nutple_a #1_#2#3(#4%
1994 {%
1995 \fi\if :#2\xint_dothis{{\xINT_ListSel_slice #3(#4}}\fi
1996 \xint_orthat {\xINT_ListSel_nthone {#2}#3(#4}%
1997 }%
1998 \def\xINT_ListSel_top_number #1_{\fi\xINT_ListSel_top_ople}%
1999 \def\xINT_ListSel_top_ople #1%
2000 {%
2001 \if :#1\xint_dothis\xINT_ListSel_slice\fi
2002 \xint_orthat {\xINT_ListSel_nthone {#1}}%
2003 }%
2004 \def\xINT_ListSel_slice #1%
2005 {%
2006 \expandafter\xINT_ListSel_slice_a \expandafter{\romannumeral0\xintnum{#1}}%
2007 }%
2008 \def\xINT_ListSel_slice_a #1#2;#3#4%
2009 {%
2010 \if _#4\expandafter\xINT_ListSel_s_b
2011 \else\expandafter\xINT_ListSel_slice_b\fi
2012 #1;#3%
2013 }%
2014 \def\xINT_ListSel_s_b #1#2;#3#4%
2015 {%
2016 \if &#4\expandafter\xINT_ListSel_s_last\fi
2017 \xINT_ListSel_s_c #1{#1#2}{#4}%
2018 }%
2019 \def\xINT_ListSel_s_last\xINT_ListSel_s_c #1#2#3(#4%
2020 {%
2021 \if-#1\expandafter\xintKeep\else\expandafter\xintTrim\fi {#2}{#4}%
2022 }%
2023 \def\xINT_ListSel_s_c #1#2#3(#4%
2024 {%
2025 \expandafter\xINT_ListSel_deeper
2026 \expanded{\xint_noxpd{#3}(\expandafter)\expandafter{%
2027 \romannumeral0%
2028 \if-#1\expandafter\xintkeep\else\expandafter\xinttrim\fi {#2}{#4}}%
2029 }%

```

`\xintNthElt` from `xinttools` (knowingly) strips one level of braces when fetching `kth` «item» from `{v1}...{vN}`. If we expand `{\xintNthElt{k}}{{v1}...{vN}}` (notice external braces):

if `k` is out of range we end up with `{}`

if `k` is in range and the `kth` braced item was `{}` we end up with `{}`

if `k` is in range and the `kth` braced item was `{17}` we end up with `{17}`

Problem is that individual numbers such as 17 are stored `{{17}}`. So we must have one more brace pair and in the first two cases we end up with `{{}}`. But in the first case we should end up with the empty ople `{}`, not the empty bracketed ople `{{}}`.

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

I have thus added `\xintNthOne` to *xinttools* which does not strip brace pair from an extracted item.

Attention: `\XINT_nthoney_a` does no expansion on second argument. But here arguments are either numerical or already expanded. Normally.

```
2030 \def\xintNthOne #1#2%
2031 {%
2032   \if &#2\expandafter\xintNthOne_last\fi
2033   \XINT_ListSel_nthone_a {#1}{#2}%
2034 }%
2035 \def\xintNthOne_a #1#2(#3%
2036 {%
2037   \expandafter\xintNthOne_deeper
2038   \expanded{\xint_noexpd{#2}(\expandafter)\expandafter{%
2039     \romannumeral0\expandafter\xintNthOne_a\the\numexpr\xintNum{#1}.{#3}%
2040 }%
2041 \def\xintNthOne_last\xintNthOne_a #1#2(%#3%
2042 {%
2043   \romannumeral0\expandafter\xintNthOne_a\the\numexpr\xintNum{#1}.{#3}%
2044 }%
```

The macros here are basically f-expandable and use the f-expandable `\xintKeep` and `\xintTrim`. Prior to xint 1.4, there was here an x-expandable `\xintKeep:x:csv` dealing with comma separated items, for time being we make do with our f-expandable toolkit.

```
2045 \def\xintNthOne_slice_b #1;#2;#3%
2046 {%
2047   \if &#3\expandafter\xintNthOne_slice_last\fi
2048   \expandafter\xintNthOne_slice_c \expandafter{\romannumeral0\xintnum{#2}};#1;{#3}%
2049 }%
2050 \def\xintNthOne_slice_last\expandafter\xintNthOne_slice_c #1;#2;#3(%#4
2051 {%
2052   \expandafter\xintNthOne_slice_last_c #1;#2;#{#4}
2053 }%
```

**Modified at 1.4n (2025/09/05).** Compatibility with LuaMetaTeX regarding `;` not usable as `\numexpr` delimiter. Fortunately I had mostly used `\xint:` or a dot in the past throughout the code base, so not many locations needed adjustments. Here I simply replaced all semi-colons by fullstops starting here with how `\XINT_ListSel_slice_d` receives arguments. Fortunately `\xint_gob_til_sc` was not used around here.

```
2054 \def\xintNthOne_slice_last_c #1;#2;#3%
2055 {%
2056   \romannumeral0\xintNthOne_slice_d #2.#1.{#3}%
2057 }%
2058 \def\xintNthOne_slice_c #1;#2;#3(%#4%
2059 {%
2060   \expandafter\xintNthOne_deeper
2061   \expanded{\xint_noexpd{#3}(\expandafter)\expandafter{%
2062     \romannumeral0\xintNthOne_slice_d #2.#1.{#4}%
2063 }%
2064 \def\xintNthOne_slice_d #1#2.#3#4.%
2065 {%
2066   \xint_UDsignsfork
2067     #1#3\xintNthOne_N:N
2068     #1-\xintNthOne_N:P
```

```

2069     -#3\XINT_ListSel_P:N
2070     --\XINT_ListSel_P:P
2071     \krof #1#2.#3#4.%
2072 }%
2073 \def\XINT_ListSel_P:P #1.#2.#3%
2074 {%
2075     \unless\ifnum #1<#2 \expandafter\xint_gob_andstop_iii\fi
2076     \xintkeep{#2-#1}{\xintTrim{#1}{#3}}%
2077 }%
2078 \def\XINT_ListSel_N:N #1.#2.#3%
2079 {%
2080     \expandafter\XINT_ListSel_N:N_a
2081     \the\numexpr #2-#1\expandafter.\the\numexpr#1+\xintLength{#3}.#3}%
2082 }%
2083 \def\XINT_ListSel_N:N_a #1.#2.#3%
2084 {%
2085     \unless\ifnum #1>\xint_c_ \expandafter\xint_gob_andstop_iii\fi
2086     \xintkeep{#1}{\xintTrim{\ifnum#2<\xint_c_\xint_c_\else#2\fi}{#3}}%
2087 }%
2088 \def\XINT_ListSel_N:P #1.#2.#3%
2089 {%
2090     \expandafter\XINT_ListSel_N:P_a
2091     \the\numexpr #1+\xintLength{#3}.#2.#3}%
2092 }%
2093 \def\XINT_ListSel_N:P_a #1#2.%
2094 {\if -#1\expandafter\XINT_ListSel_O:P\fi\XINT_ListSel_P:P #1#2.}%
2095 \def\XINT_ListSel_O:P\XINT_ListSel_P:P #1.{\XINT_ListSel_P:P 0.}%
2096 \def\XINT_ListSel_P:N #1.#2.#3%
2097 {%
2098     \expandafter\XINT_ListSel_P:N_a
2099     \the\numexpr #2+\xintLength{#3}.#1.#3}%
2100 }%
2101 \def\XINT_ListSel_P:N_a #1#2.#3.%
2102 {\if -#1\expandafter\XINT_ListSel_P:O\fi\XINT_ListSel_P:P #3.#1#2.}%
2103 \def\XINT_ListSel_P:O\XINT_ListSel_P:P #1.#2.{\XINT_ListSel_P:P #1.0.}%

```

## 27.22. Support for raw A/B[N]

Releases earlier than 1.1 required the use of braces around A/B[N] input. The [N] is now implemented directly. \*BUT\* this uses a delimited macro! thus N is not allowed to be itself an expression (I could add it...). `\xintE`, `\xintiE`, and `\XINTinFloatE` all put #2 in a `\numexpr`. But attention to the fact that `\numexpr` stops at spaces separating digits: `\the\numexpr 3 + 7 9\relax` gives 109`\relax` !! Hence we have to be careful.

`\numexpr` will not handle catcode 11 digits, but adding a `\detokenize` will suddenly make illicit for N to rely on macro expansion.

At 1.4, [ is already overloaded and it is not easy to support this. We do this by a kludge maintaining more or less former (very not efficient) way but using \$ sign which is free for time being. No, finally I use the null character, should be safe enough! (I hesitated about using R with catcode 12).

As for ? operator we needed to hack into `\XINT_expr_getop_b` for intercepting that pseudo operator. See also `\XINT_expr_scanint_c` (`\XINT_expr_rawxintfrac`).

```

2104 \catcode0 11

```

```

2105 \let\XINT_expr_precedence_&&@ \xint_c_xiv
2106 \def\XINT_expr_op_&&@ #1#2]%
2107 {%
2108     \expandafter\XINT_expr_put_op_first
2109     \expanded{{{xintE#1{xint_zapspaces #2 \xint_gobble_i}}}%
2110     \expandafter}\romannumeral`&&@\XINT_expr_getop
2111 }%
2112 \def\XINT_iiexpr_op_&&@ #1#2]%
2113 {%
2114     \expandafter\XINT_expr_put_op_first
2115     \expanded{{{xintiE#1{xint_zapspaces #2 \xint_gobble_i}}}%
2116     \expandafter}\romannumeral`&&@\XINT_expr_getop
2117 }%
2118 \def\XINT_flexpr_op_&&@ #1#2]%
2119 {%
2120     \expandafter\XINT_expr_put_op_first
2121     \expanded{{{XINTinFloatE#1{xint_zapspaces #2 \xint_gobble_i}}}%
2122     \expandafter}\romannumeral`&&@\XINT_expr_getop
2123 }%
2124 \catcode0 12

```

## 27.23. ? as two-way and ?? as three-way «short-circuit» conditionals

Comments undergoing reconstruction.

```

2125 \let\XINT_expr_precedence_? \xint_c_xx
2126 \catcode`- 11
2127 \def\XINT_expr_op_? {\XINT_expr_op__? \XINT_expr_op_-xii}%
2128 \def\XINT_flexpr_op_? {\XINT_expr_op__? \XINT_flexpr_op_-xii}%
2129 \def\XINT_iiexpr_op_? {\XINT_expr_op__? \XINT_iiexpr_op_-xii}%
2130 \catcode`- 12
2131 \def\XINT_expr_op__? #1#2#3%
2132     {\XINT_expr_op__?_a #3!\xint_bye\XINT_expr_exec_? {#1}{#2}{#3}}%
2133 \def\XINT_expr_op__?_a #1{\expandafter\XINT_expr_op__?_b\detokenize{#1}}%
2134 \def\XINT_expr_op__?_b #1%
2135     {\if ?#1\expandafter\XINT_expr_op__?_c\else\expandafter\xint_bye\fi }%
2136 \def\XINT_expr_op__?_c #1{\xint_gob_til_! #1\XINT_expr_op_?? !\xint_bye}%
2137 \def\XINT_expr_op_?? !\xint_bye\xint_bye\XINT_expr_exec_? {\XINT_expr_exec_??}%
2138 \catcode`- 11
2139 \def\XINT_expr_exec_? #1#2%
2140 {%
2141     \expandafter\XINT_expr_check_-_after?\expandafter#1%
2142     \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifnotzero#2%
2143 }%
2144 \def\XINT_expr_exec_?? #1#2#3%
2145 {%
2146     \expandafter\XINT_expr_check_-_after?\expandafter#1%
2147     \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifsgn#2%
2148 }%
2149 \def\XINT_expr_check_-_after? #1{%
2150 \def\XINT_expr_check_-_after? ##1##2%
2151 {%
2152     \xint_UDsignfork

```

```

2153      ##2{##1}%
2154      #1{##2}%
2155      \krof
2156  }}\expandafter\XINT_expr_check-_after?\string -%
2157  \catcode`- 12

```

## 27.24. ! as postfix factorial operator

```

2158 \let\XINT_expr_precedence_! \xint_c_xx
2159 \def\XINT_expr_op_! #1%
2160 {%
2161   \expandafter\XINT_expr_put_op_first
2162   \expanded{{\romannumeral`&&\XINT:NEhook:f:one:from:one
2163     {\romannumeral`&&\xintFac#1}}\expandafter}\romannumeral`&&\XINT_expr_getop
2164 }%
2165 \def\XINT_flexpr_op_! #1%
2166 {%
2167   \expandafter\XINT_expr_put_op_first
2168   \expanded{{\romannumeral`&&\XINT:NEhook:f:one:from:one
2169     {\romannumeral`&&\XINTinFloatFacdigits#1}}\expandafter}%
2170   \romannumeral`&&\XINT_expr_getop
2171 }%
2172 \def\XINT_iiexpr_op_! #1%
2173 {%
2174   \expandafter\XINT_expr_put_op_first
2175   \expanded{{\romannumeral`&&\XINT:NEhook:f:one:from:one
2176     {\romannumeral`&&\xintiiFac#1}}\expandafter}\romannumeral`&&\XINT_expr_getop
2177 }%

```

At 1.4g, fix for input "x! == y" via a fake operator !=. The ! is of catcode 11 but this does not matter here. The definition of `\XINT_expr_itself_!=` is required by the functioning of the scanop macros.

We don't have to worry about "x! = y" as the single-character Boolean comparison = operator has been removed from syntax. Fixing it would have required obeying space tokens when parsing operators. For "x! == y" case, obeying space tokens would not solve "x!==y" input case anyhow.

```

2178 \expandafter
2179 \def\csname XINT_expr_precedence_!=\expandafter\endcsname
2180   \csname XINT_expr_itself_!=\endcsname {\XINT_expr_precedence_! !=}%
2181 \expandafter\def\csname XINT_expr_itself_!=\endcsname{!=}%

```

## 27.25. User defined variables

27.25.1	<code>\xintdefvar</code> , <code>\xintdefiivar</code> , <code>\xintdeffloatvar</code>	622
27.25.2	<code>\xintunassignvar</code>	626

### 27.25.1. `\xintdefvar`, `\xintdefiivar`, `\xintdeffloatvar`

Modified at 1.1 (2014/10/28).

Modified at 1.2p (2017/12/05). Extends `\xintdefvar` et al. to accept simultaneous assignments to multiple variables.

Modified at 1.3c (2018/06/17). Use `\xintexprSafeCatcodes` (to palliate issue with active semi-colon from Babel+French if in body of a  $\LaTeX$  document).



And allow usage with both syntaxes `name:=expr;` or `name=expr;`. Also the colon may have catcode 11, 12, or 13 with no issue. Variable names may contain letters, digits, underscores, and must not start with a digit. Names starting with @ or an underscore are reserved.

- currently @, @1, @2, @3, and @4 are reserved because they have special meanings for use in iterations,
- @@, @@@, @@@@ are also reserved but are technically functions, not variables: a user may possibly define @@ as a variable name, but if it is followed by parentheses, the function interpretation will be applied (rather than the variable interpretation followed by a tacit multiplication),
- since 1.21, the underscore \_ may be used as separator of digits in long numbers. Hence a variable whose name starts with \_ will not play well with the mechanism of tacit multiplication of variables by numbers: the underscore will be removed from input stream by the number scanner, thus creating an undefined or wrong variable name, or none at all if the variable name was an initial \_ followed by digits.

Note that the optional argument [P] as usable with `\xintfloatexpr` is **not** supported by `\xintdeffloatvar`. One must do `\xintdeffloatvar foo = \xintfloatexpr[16] blabla \relax;` to achieve the effect.

**Modified at 1.4 (2020/01/31).** The expression will be fetched up to final semi-colon in a manner allowing inner semi-colons as used in the `iter()`, `rseq()`, `subsm()`, `subsn()` etc... syntax. They don't need to be hidden within a braced pair anymore.

**Modified at 1.4 (2020/01/31).** Automatic unpacking in case of simultaneous assignments if the expression evaluates to a nutple.

Notes (added much later on 2021/06/10 during preparation of 1.4i):

1. the code did not try to intercept illicit syntax such as `\xintdefvar a,b,c:=<number>;`. It blindly «unpacked» the number handling it as if it was a nutple. The extended functionality added at 1.4i requires to check for such a situation, as the syntax is not illicit anymore.
2. the code was broken in case the expression to evaluate was an oples of length 10 or more, due to a silly mistake at some point during 1.4 development which replaced some `\ifnum` by an `\if`, perhaps due to mental confusion with the fact that functions can have at most 9 arguments, but here the code is about defining variables. Anyway this got fixed as corollary to the 1.4i extension.

**Modified at 1.4c (2021/02/20).** One year later I realized I had broken tacit multiplication for situations such as `variable(1+2)`. As hinted at in comments above before 1.4 release I had been doing some deep refactoring here, which I cancelled almost completely in the end... but not quite, and as a result there was a problem that some macro holding braced contents was expanded to late, once it was in old core routines of `xintfrac` not expecting other things than digits. I do an emergency bugfix here with some `\expandafter`'s but I don't have the code in my brain at this time, and don't have the luxury now to invest into it. Let's hope this does not induce breakage elsewhere, and that the February 2020 1.4 did not break something else.

**Modified at 1.4e (2021/05/05).** Modifies `\xintdeffloatvar` to round to the prevailing precision (formerly, any operation would induce rounding, but in case of things such as `\xintdeffloatvar foo:=\xintexpr 1/100!\relax;` there was no automatic rounding. One could use `0+` syntax to trigger it, and for oples, some trick like `\xintfloatexpr[\XINTdigits]...\relax` extra wrapper.

**Modified at 1.4g (2021/05/25).** The `\expandafter\expandafter\expandafter` et al. chain which was kept by `\XINT_expr_defvar_one_b` for expanding only at time of use the `\XINT_expr_var_foo` in `\XINT_expr_onliteral_foo` were senseless overhead added at 1.4c. This is used only for real variables, not dummy variables or fake variables and it is simpler to have the `\XINT_expr_var_foo` pre-expanded. So let's use some `\edef` here.



The `\XINT_expr_onliteral_foo` is expanded as result of action of `\XINT_expr_op_`` (or `\XINT_flexp_r_op_``, `\XINT_iiexpr_op_``) which itself was triggered consuming already an `\XINT_expr_put_op_first`, so its expansion has to produce tokens as expected after `\XINT_expr_put_op_first`: `<precedence token><op token>{expanded value}`.

**Modified at 1.4i (2021/06/11).** Implement extended notion of simultaneous assignments: if there are more variables than values, define the extra variables to be nil. If there are less variables than values let the last variable be defined as the ope concatenating all non reclaimed values.

If there are at least two variables, the right hand side, if it turns out to be a nutple, is (as since 1.4) automatically unpacked, then the above rules apply.

**Modified at 1.4i (2021/06/11).** Fix the long-standing «seq renaming bug» via a change here of the name of auxiliary macro. Previously «onliteral\_<varname>» now «var\*\_<varname>». I hesitated with using «var\_varname\*» rather.

Hesitated adding `\XINT_expr_letvar_one` (motivation: case of simultaneous assignments leading to defining «nil» variables). Finally, no.

```

2182 \catcode`* 11
2183 \def\XINT_expr_defvar_one #1#2%
2184 {%
2185     \XINT_global
2186     \expandafter\edef\csname XINT_expr_varvalue_#1\endcsname {#2}%
2187     \XINT_expr_defvar_one_b {#1}%
2188 }%
2189 \def\XINT_expr_defvar_one_b #1%
2190 {%
2191     \XINT_global
2192     \expandafter\edef\csname XINT_expr_var_#1\endcsname
2193     {{\expandafter\noexpand\csname XINT_expr_varvalue_#1\endcsname}}%
2194     \XINT_global
2195     \expandafter\edef\csname XINT_expr_var*_#1\endcsname
2196     {\XINT_expr_prec_tacit *\csname XINT_expr_var_#1\endcsname}%
2197     \ifxintverbose\xintMessage{xintexpr}{Info}%
2198     {Variable #1 \ifxintglobaldefs globally \fi
2199     defined with value \csname XINT_expr_varvalue_#1\endcsname.}%
2200     \fi
2201 }%
2202 \catcode`* 12
2203 \catcode`~ 13
2204 \catcode`: 12
2205 \def\XINT_expr_defvar_getname #1:#2~%
2206 {%
2207     \endgroup
2208     \def\XINT_defvar_tmpa{#1}\edef\XINT_defvar_tmpc{\xintCSVLength{#1}}%
2209 }%
2210 \def\XINT_expr_defvar #1#2%
2211 {%
2212     \def\XINT_defvar_tmpa{#2}%
2213     \expandafter\XINT_expr_defvar_a\expanded{\xint_noxpd{#1}}\expandafter}%
2214     \romannumeral\XINT_expr_fetch_to_semicolon
2215 }%
2216 \def\XINT_expr_defvar_a #1#2%
2217 {%
2218     \xintexprRestoreCatcodes

```

Maybe SafeCatcodes was without effect because the colon and the rest are from some earlier macro definition. Give a safe definition to active colon (even if in math mode with a math active colon..).

The `\XINT_expr_defvar_getname` closes the group opened here.

```

2219 \begingroup\lccode`~: \lowercase{\let~}\empty
2220 \edef\XINT_defvar_tmpa{\XINT_defvar_tmpa}%
2221 \edef\XINT_defvar_tmpa{\xint_zapspace_o\XINT_defvar_tmpa}%
2222 \expandafter\XINT_expr_defvar_getname
2223     \detokenize\expandafter{\XINT_defvar_tmpa}:~%
2224 \ifcase\XINT_defvar_tmpc\space
2225     \xintMessage {xintexpr}{Error}
2226     {Aborting: not allowed to declare variable with empty name.}%
2227 \or
2228     \XINT_global
2229     \expandafter
2230     \edef\cename XINT_expr_varvalue\XINT_defvar_tmpa\endcsname{#1#2\relax}%
2231     \XINT_expr_defvar_one_b\XINT_defvar_tmpa
2232 \else
2233     \edef\XINT_defvar_tmpb{#1#2\relax}%
2234     \edef\XINT_defvar_tmpe{\expandafter\xintLength\expandafter{\XINT_defvar_tmpb}}%
2235     \ifnum\XINT_defvar_tmpe=\xint_c_i
2236         \oodef\XINT_defvar_tmpe{\expandafter\xint_firstofone\XINT_defvar_tmpe}%
2237         \if0\expandafter\expandafter\expandafter\XINT_defvar_checkifnutple
2238             \expandafter\string\XINT_defvar_tmpe _\xint_bye
2239         \oodef\XINT_defvar_tmpe{\expandafter{\XINT_defvar_tmpe}}%
2240     \else
2241         \edef\XINT_defvar_tmpe{\expandafter\xintLength\expandafter{\XINT_defvar_tmpe}}%
2242     \fi
2243 \fi
2244 \xintAssignArray\xintCSVtoList\XINT_defvar_tmpe\to\XINT_defvar_tmpevar
2245 \def\XINT_defvar_tmpe{1}%
2246 \expandafter\XINT_expr_defvar_multiple\XINT_defvar_tmpe\relax
2247 \fi
2248 }%
2249 \def\XINT_defvar_checkifnutple#1%
2250 {%
2251     \if#1_1\fi
2252     \if#1\bgroup1\fi
2253     0\xint_bye
2254 }%
2255 \def\XINT_expr_defvar_multiple
2256 {%
2257     \ifnum\XINT_defvar_tmpe<\XINT_defvar_tmpe\space
2258         \expandafter\XINT_expr_defvar_multiple_one
2259     \else
2260         \expandafter\XINT_expr_defvar_multiple_last\expandafter\empty
2261     \fi
2262 }%
2263 \def\XINT_expr_defvar_multiple_one
2264 {%
2265     \ifnum\XINT_defvar_tmpe>\XINT_defvar_tmpe\space
2266         \expandafter\XINT_expr_defvar_one

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```

2267      \csname XINT_defvar_tmpvar\XINT_defvar_tmpe\endcsname{}%
2268      \edef\XINT_defvar_tmpe{\the\numexpr\XINT_defvar_tmpe+1}%
2269      \expandafter\XINT_expr_defvar_multiple
2270  \else
2271      \expandafter\XINT_expr_defvar_multiple_one_a
2272  \fi
2273 }%
2274 \def\XINT_expr_defvar_multiple_one_a #1%
2275 {%
2276     \expandafter\XINT_expr_defvar_one
2277     \csname XINT_defvar_tmpvar\XINT_defvar_tmpe\endcsname{{#1}}%
2278     \edef\XINT_defvar_tmpe{\the\numexpr\XINT_defvar_tmpe+1}%
2279     \XINT_expr_defvar_multiple
2280 }%
2281 \def\XINT_expr_defvar_multiple_last #1\relax
2282 {%
2283     \expandafter\XINT_expr_defvar_one
2284     \csname XINT_defvar_tmpvar\XINT_defvar_tmpe\endcsname{#1}%
2285     \xintRelaxArray\XINT_defvar_tmpvar
2286     \let\XINT_defvar_tmpe\empty
2287     \let\XINT_defvar_tmpe\empty
2288     \let\XINT_defvar_tmpe\empty
2289     \let\XINT_defvar_tmpe\empty
2290     \let\XINT_defvar_tmpe\empty
2291 }%
2292 \catcode`~ 3
2293 \catcode`: 11

```

This SafeCatcodes is mainly in the hope that semi-colon ending the expression can still be sanitized.

Pre 1.4e definition:

```

\def\xintdeffloatvar      {\xintexprSafeCatcodes\xintdeffloatvar_a}
\def\xintdeffloatvar_a #1={\XINT_expr_defvar\xintthebarefloateval{#1}}

```

This would keep the value (or values) with extra digits, now. If this is actually wanted one can use `\xintdefvar foo:=\xintfloatexpr...\relax`; syntax, but recalling that only operations trigger the rounding inside `\xintfloatexpr`. Some tricks are needed for no operations case if multiple or nested values. But for a single one, one can use simply the `float()` function.

```

2294 \def\xintdefvar      {\xintexprSafeCatcodes\xintdefvar_a}%
2295 \def\xintdefvar_a#1={\XINT_expr_defvar\xintthebareeval{#1}}%
2296 \def\xintdefiivar      {\xintexprSafeCatcodes\xintdefiivar_a}%
2297 \def\xintdefiivar_a#1={\XINT_expr_defvar\xintthebareiieval{#1}}%
2298 \def\xintdeffloatvar      {\xintexprSafeCatcodes\xintdeffloatvar_a}%
2299 \def\xintdeffloatvar_a #1={\XINT_expr_defvar\xintthebareroundedfloateval{#1}}%

```

### 27.25.2. \xintunassignvar

Modified at 1.2e (2015/11/22).

Modified at 1.3d (2019/01/06). Embarrassingly I had for a long time a misunderstanding of `\ifcs_name` (let's blame its documentation) and I was not aware that it chooses FALSE branch if tested control sequence has been `\let` to `\undefined`... So earlier version didn't do the right thing (and had another bug: failure to protect `\.=0` from expansion).

The `\ifcsname` tests are done in `\XINT_expr_op__` and `\XINT_expr_op``.

Modified at 1.4i (2021/06/11). Track `s/onliteral/var*/` change in macro names.

## 27.26. Support for dummy variables

### 27.26.1. \xintnewdummy

1.4 adds multi-letter names as usable dummy variables!

```

2323 \catcode`* 11
2324 \def\XINT_expr_makedummy #1%
2325 {%
2326   \edef\XINT_tmpa{\xint_zapspaces #1 \xint_gobble_i}%
2327   \ifcsname XINT_expr_var_\XINT_tmpa\endcsname

```

```

2328 \XINT_global
2329 \expandafter\let\csname XINT_expr_var_\XINT_tmpa/old\expandafter\endcsname
2330 \csname XINT_expr_var_\XINT_tmpa\expandafter\endcsname
2331 \fi
2332 \ifcsname XINT_expr_var*_\XINT_tmpa\endcsname
2333 \XINT_global
2334 \expandafter\let\csname XINT_expr_var*_\XINT_tmpa/old\expandafter\endcsname
2335 \csname XINT_expr_var*_\XINT_tmpa\expandafter\endcsname
2336 \fi
2337 \expandafter\XINT_global
2338 \expanded
2339 {\edef\expandafter\noexpand
2340 \csname XINT_expr_var_\XINT_tmpa\endcsname ##1\relax !\XINT_tmpa##2}%
2341 {{##2}##1\relax !\XINT_tmpa{##2}}%
2342 \expandafter\XINT_global
2343 \expanded
2344 {\edef\expandafter\noexpand
2345 \csname XINT_expr_var*_\XINT_tmpa\endcsname ##1\relax !\XINT_tmpa##2}%
2346 {\XINT_expr_prec_tacit *{##2}({##1\relax !\XINT_tmpa{##2}})%
2347 }%
2348 \xintApplyUnbraced \XINT_expr_makedummy {abcdefghijklmnopqrstuvwxyz}%
2349 \xintApplyUnbraced \XINT_expr_makedummy {ABCDEFGHIJKLMNOPQRSTUVWXYZ}%
2350 \def\xintnewdummy #1{%
2351 \XINT_expr_makedummy{#1}%
2352 \ifxintverbose\xintMessage {xintexpr}{Info}%
2353 {\XINT_tmpa\space now
2354 \ifxintglobaldefs globally \fi usable as dummy variable.}%
2355 \fi
2356 }%
2357 \catcode`* 12

```

The **nil** variable was need in **xint < 1.4** (with some other meaning) in places the syntax could not allow emptiness, such as **,,** and other things, but at **1.4** meaning as changed.

The other variables are new with **1.4**. Don't use the **None**, it is tentative, and may be input as **[]**.

Refactored at **1.4i** to define them as really genuine variables, i.e. also with associated **var\*** macros involved in tacit multiplication (even though it will be broken with **nil**, and with **None** in **\xintiexpr**). No real reason, because **\XINT\_expr\_op\_\_** managed them fine even in absence of **var\*** macros.

```

2358 \XINT_expr_defvar_one{nil}{}%
2359 \XINT_expr_defvar_one{None}{}% ? tentative
2360 \XINT_expr_defvar_one{false}{}% Maple, TeX
2361 \XINT_expr_defvar_one{true}{}%
2362 \XINT_expr_defvar_one{False}{}% Python
2363 \XINT_expr_defvar_one{True}{}%

```

### 27.26.2. \xintensuredummy, \xintrestorevariable

1.3e **\xintensuredummy** differs from **\xintnewdummy** only in the informational message... Attention that this is not meant to be nested.

1.4 fixes that the message mentioned non-existent **\xintrestoredummy** (real name was **\xintrest\_orelettervar** and renames the latter to **\xintrestorevariable** as it applies also to multi-letter names.

```

2364 \def\xintensuredummy #1{%
2365   \XINT_expr_makedummy{#1}%
2366   \ifxintverbose\xintMessage {xintexpr}{Info}%
2367     {\XINT_tmpa\space now
2368     \ifxintglobaldefs globally \fi usable as dummy variable.&&J
2369     Issue \string\xintrestorevariable{\XINT_tmpa} to restore former meaning.}%
2370   \fi
2371 }%
2372 \def\xintrestorevariablesilently #1{%
2373   \edef\XINT_tmpa{\xint_zapspaces #1 \xint_gobble_i}%
2374   \ifcsname XINT_expr_var_\XINT_tmpa/old\endcsname
2375     \XINT_global
2376     \expandafter\let\csname XINT_expr_var_\XINT_tmpa\expandafter\endcsname
2377       \csname XINT_expr_var_\XINT_tmpa/old\expandafter\endcsname
2378   \fi
2379   \ifcsname XINT_expr_var*_\XINT_tmpa/old\endcsname
2380     \XINT_global
2381     \expandafter\let\csname XINT_expr_var*_\XINT_tmpa\expandafter\endcsname
2382       \csname XINT_expr_var*_\XINT_tmpa/old\expandafter\endcsname
2383   \fi
2384 }%
2385 \def\xintrestorevariable #1{%
2386   \xintrestorevariablesilently {#1}%
2387   \ifxintverbose\xintMessage {xintexpr}{Info}%
2388     {\XINT_tmpa\space
2389     \ifxintglobaldefs globally \fi restored to its earlier status, if any.}%
2390   \fi
2391 }%

```

### 27.26.3. Checking (without expansion) that a symbolic expression contains correctly nested parentheses

Expands to `\xint_c_mone` in case a closing `)` had no opening `(` matching it, to `\@ne` if opening `(` had no closing `)` matching it, to `\z@` if expression was balanced. Call it as:

`\XINT_isbalanced_a \relax #1(\xint_bye)\xint_bye`

This is legacy f-expandable code not using `\expanded` even at 1.4.

```

2392 \def\XINT_isbalanced_a #1({\XINT_isbalanced_b #1)\xint_bye }%
2393 \def\XINT_isbalanced_b #1)#2%
2394   {\xint_bye #2\XINT_isbalanced_c\xint_bye\XINT_isbalanced_error }%
2395   if #2 is not \xint_bye, a ) was found, but there was no (. Hence error -> -1
2396 \def\XINT_isbalanced_error #1)\xint_bye {\xint_c_mone}%
2397   #2 was \xint_bye, was there a ) in original #1?
2398 \def\XINT_isbalanced_c\xint_bye\XINT_isbalanced_error #1%
2399   {\xint_bye #1\XINT_isbalanced_yes\xint_bye\XINT_isbalanced_d #1}%
2400   #1 is \xint_bye, there was never ( nor ) in original #1, hence OK.
2401 \def\XINT_isbalanced_yes\xint_bye\XINT_isbalanced_d\xint_bye )\xint_bye {\xint_c_}%
2402   #1 is not \xint_bye, there was indeed a ( in original #1. We check if we see a ). If we do, we then
2403   loop until no ( nor ) is to be found.
2404 \def\XINT_isbalanced_d #1)#2%
2405   {\xint_bye #2\XINT_isbalanced_no\xint_bye\XINT_isbalanced_a #1#2}%

```

#2 was `\xint_bye`, we did not find a closing `)` in original #1. Error.

```
2401 \def\xINT_isbalanced_no\xint_bye #1\xint_bye\xint_bye {\xint_c_i }%
```

#### 27.26.4. Fetching balanced expressions E1, E2 and a variable name Name from E1, Name=E2)

Multi-letter dummy variables added at 1.4.

```
2402 \def\xINT_expr_fetch_E_comma_V_equal_E_a #1#2,%
2403 {%
2404   \ifcase\xINT_isbalanced_a \relax #1#2(\xint_bye)\xint_bye
2405     \expandafter\xINT_expr_fetch_E_comma_V_equal_E_c
2406     \or\expandafter\xINT_expr_fetch_E_comma_V_equal_E_b
2407     \else\expandafter\xintError:noopening
2408   \fi {#1#2},%
2409 }%
2410 \def\xINT_expr_fetch_E_comma_V_equal_E_b #1,%
2411   {\xINT_expr_fetch_E_comma_V_equal_E_a {#1},}%
2412 \def\xINT_expr_fetch_E_comma_V_equal_E_c #1,#2#3=%
2413 {%
2414   \expandafter\xINT_expr_fetch_E_comma_V_equal_E_d\expandafter
2415   {\expanded{{\xint_zapspaces #2#3 \xint_gobble_i}}{#1}}}%
2416 }%
2417 \def\xINT_expr_fetch_E_comma_V_equal_E_d #1#2#3)%
2418 {%
2419   \ifcase\xINT_isbalanced_a \relax #2#3(\xint_bye)\xint_bye
2420     \or\expandafter\xINT_expr_fetch_E_comma_V_equal_E_e
2421     \else\expandafter\xintError:noopening
2422   \fi
2423   {#1}{#2#3}%
2424 }%
2425 \def\xINT_expr_fetch_E_comma_V_equal_E_e #1#2%
2426   {\xINT_expr_fetch_E_comma_V_equal_E_d {#1}{#2}}%
```

#### 27.26.5. Fetching a balanced expression delimited by a semi-colon

1.4. For `subsn()` leaner syntax of nested substitutions.

Will also serve to `\xintdeffunc`, to not have to hide inner semi-colons in for example an `iter()` from `\xintdeffunc`.

Adding brace removal protection for no serious reason, anyhow the `xintexpr` parsers always removes braces when moving forward, but well.

Trigger by `\romannumeral\xINT_expr_fetch_to_semicolon` upfront.

```
2427 \def\xINT_expr_fetch_to_semicolon {\xINT_expr_fetch_to_semicolon_a {} \empty}%
2428 \def\xINT_expr_fetch_to_semicolon_a #1#2;%
2429 {%
2430   \ifcase\xINT_isbalanced_a \relax #1#2(\xint_bye)\xint_bye
2431     \xint_dothis{\expandafter\xINT_expr_fetch_to_semicolon_c}%
2432     \or\xint_dothis{\expandafter\xINT_expr_fetch_to_semicolon_b}%
2433     \else\expandafter\xintError:noopening
2434   \fi\xint_orthat{\}\expandafter{#2}{#1}%
2435 }%
2436 \def\xINT_expr_fetch_to_semicolon_b #1#2{\xINT_expr_fetch_to_semicolon_a {#2#1;} \empty}%
2437 \def\xINT_expr_fetch_to_semicolon_c #1#2{\xint_c_{#2#1}}%
```



### 27.26.6. Low-level support for omit and abort keywords, the break() function, the n++ construct and the semi-colon as used in the syntax of seq(), add(), mul(), iter(), rseq(), iterr(), rrseq(), subsm(), subsn(), ndseq(), ndmap()

There is some clever play simply based on setting suitable precedence levels combined with special meanings given to op macros.

The special !? internal operator is a helper for omit and abort keywords in list generators.

Prior to 1.4 support for +[, \*[, ..., ]+, ]\*, had some elements here.

**The n++ construct** 1.1 2014/10/29 did `\expandafter\.=+\xintiCeil` which transformed it into `\romannumeral0\xinticeil`, which seems a bit weird. This exploited the fact that dummy variables macros could back then pick braced material (which in the case at hand here ended being `{\romannumeral0\xinticeil...}` and were submitted to two expansions. The result of this was to provide a not value which got expanded only in the first loop of the `:_A` and following macros of `seq`, `iter`, `rseq`, etc...

Anyhow with 1.2c I have changed the implementation of dummy variables which now need to fetch a single locked token, which they do not expand.

The `\xintiCeil` appears a bit dispendious, but I need the starting value in a `\numexpr` compatible form in the iteration loops.

```
2438 \expandafter\def\csname XINT_expr_itself_++\endcsname {++}%
2439 \expandafter\def\csname XINT_expr_itself_++\endcsname {++}}%
2440 \expandafter\let\csname XINT_expr_precedence_++\endcsname \xint_c_i
2441 \xintFor #1 in {expr,flexpr,iiexpr} \do {%
2442     \expandafter\def\csname XINT_#1_op_++\endcsname ##1##2\relax
2443     {\expandafter\XINT_expr_founded
2444         \expanded{{+{\XINT:NEhook:f:one:from:one:direct\xintiCeil##1}}}%
2445     }%
2446 }%
```

**The break() function** break is a true function, the parsing via expansion of the enclosed material proceeds via `_oparen` macros as with any other function.

```
2447 \catcode`? 3
2448 \def\XINT_expr_func_break #1#2#3{#1#2{?#3}}%
2449 \catcode`? 11
2450 \let\XINT_flexpr_func_break \XINT_expr_func_break
2451 \let\XINT_iiexpr_func_break \XINT_expr_func_break
```

**The omit and abort keywords** Comments are currently undergoing reconstruction.

The mechanism is somewhat complex. The operator !? will fetch a dummy value ! or ^ which is then recognized into the loops implementing the various `seq` etc... construct using dummy variables and implement omit and abort.

In May 2021 I realized that the January 2020 1.4 had broken omit and abort if used inside a `subs()`. The definition

```
\edef\XINT_expr_var_omit #1\relax !{\string !?!\relax !}
```

conflicted with the 1.4 refactoring of «subs» and similar things which had replaced formerly clean-up macros (of ! and what's next, as in now defunct `\def\XINT_expr_subx:_end #1!#2#3{#1}` which was involved in `subs` mechanism, and by the way would be incompatible with multi-letter dummy variables) by usage of an `\iffalse` as in `"\relax\iffalse\relax !"` to delimit a sub-expression, which was supposed to be clever (the `"\relax !"` being delimiter for dummy variables).

This `\iffalse` from `subs` mechanism ended up being gobbled by omit/abort thus inducing breakage.

Grabbing `\relax #2!` would be a fix but looks a bit dangerous, as there can be a subexpression after the omit or abort bringing its own `\relax`, although this is very very unlikely.



I considered to modify the dummy variables delimiter from `\relax !` to `\xint_Bye !` for example but got afraid from the ramifications, as all structures handling dummy variables would have needed refactoring.

So finally things here remain unchanged and the refactoring to fix this breakage was done in `\XINT_allexpr_subsx` (and also `subsm`). Done at 1.4h. See `\XINT_allexpr_subsx` for comments.

```
2452 \edef\xintexpr_var_omit #1\relax !{1\string !?! \relax !}%
2453 \edef\xintexpr_var_abort #1\relax !{1\string !?^ \relax !}%
2454 \def\xintexpr_itself_! ? {!}%
2455 \def\xintexpr_op_! ? #1#2\relax{\xintexpr_foundend{#2}}%
2456 \let\xintexpr_iiexpr_op_! ? \xintexpr_op_! ?
2457 \let\xintexpr_flexpr_op_! ? \xintexpr_op_! ?
2458 \let\xintexpr_precedence_! ? \xint_c_iv
```

**The semi-colon** Obsolete comments undergoing re-construction

```
2459 \xintFor #1 in {expr, flexpr, iiexpr} \do {%
2460   \expandafter\def\csname XINT_#1_op_;\endcsname {\xint_c_i ;}%
2461 }%
2462 \expandafter\let\csname XINT_expr_precedence_;\endcsname\xint_c_i
2463 \expandafter\def\csname XINT_expr_itself_;\endcsname {};%
2464 \expandafter\let\csname XINT_expr_precedence_;\endcsname\xint_c_i
```

#### 27.26.7. Reserved dummy variables @, @1, @2, @3, @4, @@, @@(1), ..., @@@, @@@(1), ... for recursions

Comments currently under reconstruction.

1.4 breaking change: @ and @1 behave differently and one can not use @ in place of @1 in `iterr()` and `rrseq()`. Formerly @ and @1 had the same definition.

Brace stripping in `\XINT_expr_func_@` is prevented by some ending 0 or other token see `iterr()` and `rrseq()` code.

For the record, the ~ and ? have catcode 3 in this code.

```
2465 \catcode`* 11
2466 \def\xintexpr_var_@ #1~#2{{#2}#1~{#2}}%
2467 \def\xintexpr_var*_@ #1~#2{\xintexpr_prec_tacit *{#2}{#1~{#2}}}%
2468 \expandafter
2469 \def\csname XINT_expr_var_@1\endcsname #1~#2{{#2}#1~{#2}}%
2470 \expandafter
2471 \def\csname XINT_expr_var_@2\endcsname #1~#2#3{{#3}#1~{#2}{#3}}%
2472 \expandafter
2473 \def\csname XINT_expr_var_@3\endcsname #1~#2#3#4{{#4}#1~{#2}{#3}{#4}}%
2474 \expandafter
2475 \def\csname XINT_expr_var_@4\endcsname #1~#2#3#4#5{{#5}#1~{#2}{#3}{#4}{#5}}%
2476 \expandafter\def\csname XINT_expr_var*_@1\endcsname #1~#2%
2477   {\xintexpr_prec_tacit *{#2}{#1~{#2}}}%
2478 \expandafter\def\csname XINT_expr_var*_@2\endcsname #1~#2#3%
2479   {\xintexpr_prec_tacit *{#3}{#1~{#2}{#3}}}%
2480 \expandafter\def\csname XINT_expr_var*_@3\endcsname #1~#2#3#4%
2481   {\xintexpr_prec_tacit *{#4}{#1~{#2}{#3}{#4}}}%
2482 \expandafter\def\csname XINT_expr_var*_@4\endcsname #1~#2#3#4#5%
2483   {\xintexpr_prec_tacit *{#5}{#1~{#2}{#3}{#4}{#5}}}%
2484 \catcode`* 12
2485 \catcode`? 3
2486 \def\xintexpr_func_@@ #1#2#3#4~#5?%
```

```

2487 {%
2488   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2489     \romannumeral0\xintntheltnoexpand{\xintNum#3}{#5}}}\#4~#5?%
2490 }%
2491 \def\xINT_expr_func_@@@ #1#2#3#4~#5~#6?%
2492 {%
2493   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2494     \romannumeral0\xintntheltnoexpand{\xintNum#3}{#6}}}\#4~#5~#6?%
2495 }%
2496 \def\xINT_expr_func_@@@ #1#2#3#4~#5~#6~#7?%
2497 {%
2498   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2499     \romannumeral0\xintntheltnoexpand{\xintNum#3}{#7}}}\#4~#5~#6~#7?%
2500 }%
2501 \let\xINT_flexpr_func_@@\xINT_expr_func_@@
2502 \let\xINT_flexpr_func_@@@\xINT_expr_func_@@@
2503 \let\xINT_flexpr_func_@@@@\xINT_expr_func_@@@
2504 \def\xINT_iexpr_func_@@ #1#2#3#4~#5?%
2505 {%
2506   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2507     \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#5}}}\#4~#5?%
2508 }%
2509 \def\xINT_iexpr_func_@@@ #1#2#3#4~#5~#6?%
2510 {%
2511   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2512     \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#6}}}\#4~#5~#6?%
2513 }%
2514 \def\xINT_iexpr_func_@@@ #1#2#3#4~#5~#6~#7?%
2515 {%
2516   \expandafter#1\expandafter#2\expandafter{\expandafter{%
2517     \romannumeral0\xintntheltnoexpand{\xint_firstofone#3}{#7}}}\#4~#5~#6~#7?%
2518 }%
2519 \catcode`? 11

```

## 27.27. Pseudo-functions involving dummy variables and generating scalars or sequences

27.27.1	Comments . . . . .	633
27.27.2	subs(): substitution of one variable . . . . .	635
27.27.3	subsm(): simultaneous independent substitutions . . . . .	636
27.27.4	subsn(): leaner syntax for nesting (possibly dependent) substitutions . . . . .	637
27.27.5	seq(): sequences from assigning values to a dummy variable . . . . .	639
27.27.6	iter() . . . . .	640
27.27.7	add(), mul() . . . . .	641
27.27.8	rseq() . . . . .	642
27.27.9	iterr() . . . . .	643
27.27.10	rrseq() . . . . .	644

### 27.27.1. Comments

Comments added 2020/01/16.

The mechanism for «seq» is the following. When the parser encounters «seq», which means it parsed these letters and encountered (from expansion) an opening parenthesis, the \xINT\_expr\_func

mechanism triggers the «`» operator which realizes that «seq» is a pseudo-function (there is no `_func_seq`) and thus spans the `\XINT_expr_onliteral_seq` macro (currently this means however that the knowledge of which parser we are in is lost, see comments of `\XINT_expr_op`` code). The latter will use delimited macros and parenthesis check to fetch (without any expansion), the symbolic expression `ExprSeq` to evaluate, the `Name` (now possibly multi-letter) of the variable and the expression `ExprValues` to evaluate which will give the values to assign to the dummy variable `Name`. It then positions upstream `ExprValues` suitably terminated (see next) and after it `{{Name}}{ExprSeq}}`. Then it inserts a second call to the «`» operator with now «seqx» as argument hence the appropriate «{,fl,ii}expr\_func\_seqx» macros gets executed. The general way function macros work is that first all their arguments are evaluated via a call not to `\xintbare{,float,ii}eval` but to the suitable `\XINT_{expr,flexpr,iiexpr}_oparen` core macro which does almost same excepts it expects a final closing parenthesis (of course allowing nested parenthesis in-between) and stops there. Here, this closing parenthesis got positioned deliberately with a `\relax` after it, so the parser, which always after having gathered a value looks ahead to find the next operator, thinks it has hit the end of the expression and as result inserts a `\xint_c_` (i.e. `\z@`) token for precedence level and a dummy `\relax` token (place-holder for a non-existing operator). Generally speaking «func\_foo» macros expect to be executed with three parameters #1#2#3, #1 = precedence, #2 = operator, #3 = values (call it «args») i.e. the fully evaluated list of all its arguments. The special «func\_seqx» and cousins know that the first two tokens are trash and they now proceed forward, having thus lying before them upstream the values to loop over, now fully evaluated, and `{{Name}}{ExprSeq}}`. It then positions appropriately `ExprSeq` inside a sub-expression and after it, following suitable delimiter, `Name` and the evaluated values to assign to `Name`.

Dummy variables are essentially simply delimited macros where the delimiter is the variable name preceded by a `\relax` token and a catcode 11 exclamation point. Thus the various «subsx», «seqx», «iterx» position the tokens appropriately and launch suitable loops.

All of this nests well, inner «seq»'s (or more often in practice «subs»'s) being allowed to refer to the dummy variables used by outer «seq»'s because the outer «seq»'s have the values to assign to their variables evaluated first and their `ExprSeq` evaluated last. For inner dummy variables to be able to refer to outer dummy variables the author must be careful of course to not use in the implementation braces { and } which would break dummy variables to fetch values beyond the closing brace.

The above «seq» mechanism was done around June 15-25th 2014 at the time of the transition from 1.09n to 1.1 but already in October 2014 I made a note that I had a hard time to understand it again:

« [START OF YEAR 2014 COMMENTS]

All of seq, add, mul, rseq, etc... (actually all of the extensive changes from `xintexpr` 1.09n to 1.1) was done around June 15-25th 2014, but the problem is that I did not document the code enough, and I had a hard time understanding in October what I had done in June. Despite the lesson, again being short on time, I do not document enough my current understanding of the innards of the beast...

I added subs, and iter in October (also the `[:n]`, `[n:]` list extractors), proving I did at least understand a bit (or rather could imitate) my earlier code (but don't ask me to explain `\xintNewExpr` !)

The `\XINT_expr_fetch_E_comma_V_equal_E_a` parses: "expression, variable=list)" (when it is called the opening ( has been swallowed, and it looks for the ending one.) Both expression and list may themselves contain parentheses and commas, we allow nesting. For example "`x^2,x=1..10`", at the end of `seq_a` we have `{variable{expression}}{list}`, in this example `{x{x^2}}{1..10}`, or more complicated "`seq(add(y,y=1..x),x=1..10)`" will work too. The variable is a single lowercase Latin letter.

The complications with `\xint_c_ii^v` in `seq_f` is for the recurrent thing that we don't know in what type of expressions we are, hence we must move back up, with some loss of efficiency (superfluous check for minus sign, etc...). But the code manages simultaneously `expr`, `flexpr` and `iiexpr`.

[END OF YEAR 2014 OLD COMMENTS]»

On Jeudi 16 janvier 2020 à 15:13:32 I finally did the documentation as above.

The case of «iter», «rseq», «iterr», «rrseq» differs slightly because the initial values need evaluation. This is done by genuine functions `\XINT_<parser>_func_iter` etc... (there was no `\XINT_<parser>_func_seq`). The trick is via the semi-colon ; which is a genuine operator having the precedence of a closing parenthesis and whose action is only to stop expansion. Thus this first step of gathering the initial values is done as part of the regular expansion job of the parser not using delimited macros and the ; can be hidden in braces {} because the three parsers when moving forward remove one level of braces always. Thus `\XINT_<parser>_func_seq` simply hand over to `\XINT_allexpr_iter` which will then trigger the fetching without expansion of `ExprIter`, `Name=ExprValues` as described previously for «seq».

With 1.4, multi-letter names for dummy variables are allowed.

Also there is the additional 1.4 ambition to make the whole thing parsable by `\xintNewExpr/\xintdeffunc`. This is done by checking if all is numerical, because the omit, abort and break() mechanisms have no translation into macros, and the only solution for symbolic material is to simply keep it as is, so that expansion will again activate the xintexpr parsers. At 1.4 this approach is fine although the initial goals of `\xintNewExpr/\xintdeffunc` was to completely replace the parsers (whose storage method hit the string pool formerly) by macros. Now that 1.4 does not impact the string pool we can make `\xintdeffunc` much more powerful but it will not be a construct using only xintfrac macros, it will still be partially the `\xintexpr` etc... parsers in such cases.

Got simpler with 1.2c as now the dummy variable fetches an already encapsulated value, which is anyhow the form in which we get it.

Refactored at 1.4 using `\expanded` rather than `\csname`.

And support for multi-letter variables, which means function declarations can now use multi-letter variables !

### 27.27.2. subs(): substitution of one variable

```
2520 \def\XINT_expr_onliteral_subs
2521 {%
2522   \expandafter\XINT_allexpr_subsx_f
2523   \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a }%
2524 }%
2525 \def\XINT_allexpr_subsx_f #1#2{\xint_c_ii^v`{subsx}#2)\relax #1}%
2526 \def\XINT_expr_func_subsx #1#2{\XINT_allexpr_subsx \xintbareeval }%
2527 \def\XINT_flexpr_func_subsx #1#2{\XINT_allexpr_subsx \xintbarefloateval}%
2528 \def\XINT_iexpr_func_subsx #1#2{\XINT_allexpr_subsx \xintbareieeval }%
```

#2 is the value to assign to the dummy variable #3 is the dummy variable name (possibly multi-letter), #4 is the expression to evaluate

1.4 was doing something clever to get rid of the ! and tokens following it, via an `\iffalse...\fi` which erased them and propagated the expansion to trigger the getopt:

```
\expanded\bgroup\romannumeral0#1#4\relax \iffalse\relax !#3{#2}{\fi\expandafter}
```

But sadly, with a delay of more than one year later (right after having released 1.4g) I realized that this had broken omit and abort if inside a subs. As omit and abort would clean all up to `\relax !`, this meant here swallowing in particular the above `\iffalse`, leaving a dangling `\fi`. I had the files which show this bug already at time of 1.4 release but did not compile them, and they were not included in my test suite.

I hesitated with modifying the delimiter from `"\relax !<varname>"` (catcode 11 !) to `"\relax \xint_Bye<varname>"` for the dummy variables which would have allowed some trickery using `\xint_Bye...\xint_bye` clean-up but got afraid from the breakage potential of such refactoring with many induced changes.

A variant like this:

```
\def\XINT_allexpr_subsx #1#2#3#4
```

```

{
  \expandafter\XINT_expr_clean_and_put_op_first
  \expanded
  {\romannumeral0#1#4\relax !#3{#2}\xint:\expandafter}\romannumeral`&&\XINT_expr_getop
}
\def\XINT_expr_clean_and_put_op_first #1#2\xint:#3#4{#3#4{#1}}

```

breaks nesting: the braces make variables encountered in #4 unable to match their definition. This would work:

```

\def\XINT_allexpr_subsx #1#2#3#4
{
  \expandafter\XINT_allexpr_subsx_clean\romannumeral0#1#4\relax !#3{#2}\xint:
}
\def\XINT_allexpr_subsx_clean #1#2\xint:
{
  \expandafter\XINT_expr_put_op_first
  \expanded{\xint_noexpd{#1}}\expandafter}\romannumeral`&&\XINT_expr_getop
}

```

(not tested).

But in the end I decided to simply fix the first envisioned code above. This accepts expansion of supposedly inert #3{#2}. There is again the `\iffalse` but it is moved to the right. This change limits possibly hacky future developments. Done at 1.4h (2021/01/27).

No need for the `\expandafter`'s from `\XINT_expr_put_op_first` in `\XINT_expr_clean_and_put_op_first`.

```

2529 \def\XINT_allexpr_subsx #1#2#3#4%
2530 {%
2531   \expandafter\XINT_expr_clean_and_put_op_first
2532   \expanded
2533   \bgroup\romannumeral0#1#4\relax !#3{#2}\xint:\iffalse{\fi\expandafter}%
2534   \romannumeral`&&\XINT_expr_getop
2535 }%
2536 \def\XINT_expr_clean_and_put_op_first #1#2\xint:#3#4{#3#4{#1}}%

```

### 27.27.3. subsm(): simultaneous independent substitutions

New with 1.4. Globally the `var1=expr1; var2=expr2; var2=expr3; ...` part can arise from expansion, except that once a semi-colon has been found (from expansion) the `varK=` thing following it must be there. And as for `subs()` the final parenthesis must be there from the start.

```

2537 \def\XINT_expr_onliteral_subsm
2538 {%
2539   \expandafter\XINT_allexpr_subsm_f
2540   \romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {}%
2541 }%
2542 \def\XINT_allexpr_subsm_f #1#2{\xint_c_ii^v`{subsmx}#2)\relax #1}%
2543 \def\XINT_expr_func_subsmx
2544 {%
2545   \expandafter\XINT_allexpr_subsmx\expandafter\xintbareeval
2546   \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_expr_oparen
2547 }%
2548 \def\XINT_flexpr_func_subsmx
2549 {%
2550   \expandafter\XINT_allexpr_subsmx\expandafter\xintbarefloateval
2551   \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_flexpr_oparen

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```

2552 }%
2553 \def\XINT_iiexpr_func_subsmx
2554 {%
2555     \expandafter\XINT_allexpr_subsmx\expandafter\xintbareiieval
2556     \expanded\bgroup{\iffalse}\fi\XINT_allexpr_subsm_A\XINT_iiexpr_oparen
2557 }%
2558 \def\XINT_allexpr_subsm_A #1#2#3%
2559 {%
2560     \ifx#2\xint_c_
2561         \expandafter\XINT_allexpr_subsm_done
2562     \else
2563         \expandafter\XINT_allexpr_subsm_B
2564     \fi #1%
2565 }%
2566 \def\XINT_allexpr_subsm_B #1#2#3#4=%
2567 {%
2568     {#2}\relax !\xint_zapspaces#3#4 \xint_gobble_i
2569     \expandafter\XINT_allexpr_subsm_A\expandafter#1\romannumeral`&&@#1%
2570 }%
2571 \def\XINT_allexpr_subsm_done #1#2{{#2}\iffalse{{\fi}}}%
2572 \def\XINT_allexpr_subsm_A #1#2#3#4%
2573 {%
2574     \expandafter\XINT_expr_clean_and_put_op_first
2575     \expanded
2576     \bgroup\romannumeral0#1#4\relax !#3#2\xint:\iffalse{\fi\expandafter}%
2577     \romannumeral`&&\XINT_expr_getop
2578 }%

```

#1 = \xintbareeval, or \xintbarefloateval or \xintbareiieval  
 #2 = evaluation of last variable assignment

Refactored at 1.4h as for \XINT\_allexpr\_subsx, see comments there related to the omit/abort co-  
 nundrum.

### 27.27.4. subsn(): leaner syntax for nesting (possibly dependent) substitutions

New with 1.4. 2020/01/24

```

2579 \def\XINT_expr_onliteral_subsn
2580 {%
2581     \expandafter\XINT_allexpr_subsn_f
2582     \romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {%}%
2583 }%
2584 \def\XINT_allexpr_subsn_f #1{\XINT_allexpr_subsn_g #1}%

```

#1 = Name1  
 #2 = Expression in all variables which is to evaluate  
 #3 = all the stuff after Name1 = and up to final parenthesis

This one needed no reactoring at 1.4h to fix the omit/abort problem, as there was no \iffalse..\fi  
 clean-up: the clean-up is done directly via \XINT\_allexpr\_subsnx\_J.

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

I only added usage of `\XINT_expr_put_op_first_noexpand`. There may be other locations where it could be used, but I can't afford now reviewing usage. For next release after 1.4h bugfix.

```
2585 \def\XINT_allexpr_subsn_g #1#2#3%
2586 {%
2587   \expandafter\XINT_allexpr_subsn_h
2588   \expanded\bgroup{\iffalse}\fi\expandafter\XINT_allexpr_subsn_B
2589   \romannumeral\XINT_expr_fetch_to_semicolon #1=#3;\hbox=;;^{#2}%
2590 }%
2591 \def\XINT_allexpr_subsn_B #1{\XINT_allexpr_subsn_C #1\vbox}%
2592 \def\XINT_allexpr_subsn_C #1#2=#3\vbox
2593 {%
2594   \ifx\hbox#1\iffalse{\fi}\expandafter\else
2595   {\xint_zapspaces #1#2 \xint_gobble_i};\xint_noxp{{#3}}}%
2596   \expandafter\XINT_allexpr_subsn_B
2597   \romannumeral\expandafter\XINT_expr_fetch_to_semicolon\fi
2598 }%
2599 \def\XINT_allexpr_subsn_h
2600 {%
2601   \xint_c_ii^v `{subsnx}\romannumeral0\xintreverseorder
2602 }%
2603 \def\XINT_expr_func_subsnx #1#2#3#4#5;#6%
2604 {%
2605   \xint_gob_til^ #6\XINT_allexpr_subsnx_H ^%
2606   \expandafter\XINT_allexpr_subsnx\expandafter
2607   \xintbareeval\romannumeral0\xintbareeval #5\relax !#4{#3}\xintundefined
2608   {\relax !#4{#3}\relax !#6}%
2609 }%
2610 \def\XINT_iiexpr_func_subsnx #1#2#3#4#5;#6%
2611 {%
2612   \xint_gob_til^ #6\XINT_allexpr_subsnx_H ^%
2613   \expandafter\XINT_allexpr_subsnx\expandafter
2614   \xintbareiieval\romannumeral0\xintbareiieval #5\relax !#4{#3}\xintundefined
2615   {\relax !#4{#3}\relax !#6}%
2616 }%
2617 \def\XINT_flexpr_func_subsnx #1#2#3#4#5;#6%
2618 {%
2619   \xint_gob_til^ #6\XINT_allexpr_subsnx_H ^%
2620   \expandafter\XINT_allexpr_subsnx\expandafter
2621   \xintbarefloateval\romannumeral0\xintbarefloateval #5\relax !#4{#3}\xintundefined
2622   {\relax !#4{#3}\relax !#6}%
2623 }%
2624 \def\XINT_allexpr_subsnx #1#2!#3\xintundefined#4#5;#6%
2625 {%
2626   \xint_gob_til^ #6\XINT_allexpr_subsnx_I ^%
2627   \expandafter\XINT_allexpr_subsnx\expandafter
2628   #1\romannumeral0#1#5\relax !#4{#2}\xintundefined
2629   {\relax !#4{#2}\relax !#6}%
2630 }%
2631 \def\XINT_allexpr_subsnx_H ^#1\romannumeral0#2#3!#4\xintundefined #5#6%
2632 {%
2633   \expandafter\XINT_allexpr_subsnx_J\romannumeral0#2#6#5%
2634 }%
```



```

2635 \def\XINT_allexpr_subsnx_I ^#1\romannumeral0#2#3\xintundefined #4#5%
2636 {%
2637   \expandafter\XINT_allexpr_subsnx_J\romannumeral0#2#5#4%
2638 }%
2639 \def\XINT_allexpr_subsnx_J #1#2^%
2640 {%
2641   \expandafter\XINT_expr_put_op_first_noexpand
2642   \expanded{\xint_noexpd{#1}}\expandafter\romannumeral`&&\XINT_expr_getop
2643 }%
2644 \def\XINT_expr_put_op_first_noexpand#1#2#3{#2#3{#1}}%

```

### 27.27.5. seq(): sequences from assigning values to a dummy variable

In seq\_f, the #2 is the ExprValues expression which needs evaluation to provide the values to the dummy variable and #1 is {Name}{ExprSeq} where Name is the name of dummy variable and {ExprSeq} the expression which will have to be evaluated.

```

2645 \def\XINT_allexpr_seq_f #1#2{\xint_c_ii^v `{seqx}#2)\relax #1}%
2646 \def\XINT_expr_onliteral_seq
2647   {\expandafter\XINT_allexpr_seq_f\romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
2648 \def\XINT_expr_func_seqx #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbareeval }%
2649 \def\XINT_flexpr_func_seqx #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbarefloateval}%
2650 \def\XINT_iexpr_func_seqx #1#2{\XINT:NEhook:seqx\XINT_allexpr_seqx\xintbareiieval }%
2651 \def\XINT_allexpr_seqx #1#2#3#4%
2652 {%
2653   \expandafter\XINT_expr_put_op_first
2654   \expanded \bgroup {\iffalse}\fi\XINT_expr_seq:_b {#1#4\relax !#3}#2^%
2655   \XINT_expr_cb_and_getop
2656 }%
2657 \def\XINT_expr_cb_and_getop{\iffalse{\fi\expandafter}\romannumeral`&&\XINT_expr_getop}%

```

Comments undergoing reconstruction.

```

2658 \catcode`? 3
2659 \def\XINT_expr_seq:_b #1#2%
2660 {%
2661   \ifx +#2\xint_dothis\XINT_expr_seq:_Ca\fi
2662   \ifx !#2!\xint_dothis\XINT_expr_seq:_noop\fi
2663   \ifx ^#2\xint_dothis\XINT_expr_seq:_end\fi
2664   \xint_orthat{\XINT_expr_seq:_c}{#2}{#1}%
2665 }%
2666 \def\XINT_expr_seq:_noop #1{\XINT_expr_seq:_b }%
2667 \def\XINT_expr_seq:_end #1#2{\iffalse{\fi}}%
2668 \def\XINT_expr_seq:_c #1#2{\expandafter\XINT_expr_seq:_d\romannumeral0#2{{#1}}{#2}}%
2669 \def\XINT_expr_seq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_seq:_abort\fi
2670   \ifx ?#1\xint_dothis\XINT_expr_seq:_break\fi
2671   \ifx !#1\xint_dothis\XINT_expr_seq:_omit\fi
2672   \xint_orthat{\XINT_expr_seq:_goon }{#1}}%
2673 \def\XINT_expr_seq:_abort #1!#2^{\iffalse{\fi}}%
2674 \def\XINT_expr_seq:_break #1!#2^{#1\iffalse{\fi}}%
2675 \def\XINT_expr_seq:_omit #1!#2#{\expandafter\XINT_expr_seq:_b\xint_gobble_i}%
2676 \def\XINT_expr_seq:_goon #1!#2#{#1\expandafter\XINT_expr_seq:_b\xint_gobble_i}%
2677 \def\XINT_expr_seq:_Ca #1#2#3{\XINT_expr_seq:_Cc#3.#2}%
2678 \def\XINT_expr_seq:_Cb #1{\expandafter\XINT_expr_seq:_Cc\the\numexpr#1+\xint_c_i.}%
2679 \def\XINT_expr_seq:_Cc #1.#2{\expandafter\XINT_expr_seq:_D\romannumeral0#2{{#1}}{#1}{#2}}%

```



```

2680 \def\XINT_expr_seq:_D #1{\ifx ^#1\xint_dothis\XINT_expr_seq:_abort\fi
2681           \ifx ?#1\xint_dothis\XINT_expr_seq:_break\fi
2682           \ifx !#1\xint_dothis\XINT_expr_seq:_Omit\fi
2683           \xint_orthat{\XINT_expr_seq:_Goon {#1}}}%
2684 \def\XINT_expr_seq:_Omit #1!#2#\expandafter\XINT_expr_seq:_Cb\xint_gobble_i}%
2685 \def\XINT_expr_seq:_Goon #1!#2#\expandafter\XINT_expr_seq:_Cb\xint_gobble_i}%

```

### 27.27.6. iter()

Prior to 1.2g, the `iter` keyword was what is now called `iterr`, analogous with `rrseq`. Somehow I forgot an `iter` functioning like `rseq` with the sole difference of printing only the last iteration. Both `rseq` and `iter` work well with list selectors, as `@` refers to the whole comma separated sequence of the initial values. I have thus deliberately done the backwards incompatible renaming of `iter` to `iterr`, and the new `iter`.

To understand the tokens which are presented to `\XINT_allexpr_iter` it is needed to check elsewhere in the source code how the `;` hack is done.

The `#2` in `\XINT_allexpr_iter` is `\xint_c_i` from the `;` hack. Formerly (`xint < 1.4`) there was no such token. The change is motivated to using `;` also in `subsm()` syntax.

```

2686 \def\XINT_expr_func_iter {\XINT_allexpr_iter \xintbareeval }%
2687 \def\XINT_flexpr_func_iter {\XINT_allexpr_iter \xintbarefloateval }%
2688 \def\XINT_iiexpr_func_iter {\XINT_allexpr_iter \xintbareiieval }%
2689 \def\XINT_allexpr_iter #1#2#3#4%
2690 {%
2691   \expandafter\XINT_expr_iterx
2692   \expandafter#1\expanded{\xint_noexpd{{#4}}\expandafter}%
2693   \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a }%
2694 }%
2695 \def\XINT_expr_iterx #1#2#3#4%
2696 {%
2697   \XINT:NEhook:iter\XINT_expr_itery\romannumeral0#1(#4)\relax {#2}#3#1%
2698 }%
2699 \def\XINT_expr_itery #1#2#3#4#5%
2700 {%
2701   \expandafter\XINT_expr_put_op_first
2702   \expanded \bgroup {\iffalse}\fi
2703   \XINT_expr_iter:_b {#5#4}\relax !#3#1^~{#2}\XINT_expr_cb_and_getop
2704 }%
2705 \def\XINT_expr_iter:_b #1#2%
2706 {%
2707   \ifx +#2\xint_dothis\XINT_expr_iter:_Ca\fi
2708   \ifx !#2!\xint_dothis\XINT_expr_iter:_noop\fi
2709   \ifx ^#2\xint_dothis\XINT_expr_iter:_end\fi
2710   \xint_orthat{\XINT_expr_iter:_c}{#2}{#1}%
2711 }%
2712 \def\XINT_expr_iter:_noop #1{\XINT_expr_iter:_b }%
2713 \def\XINT_expr_iter:_end #1#2~#3{#3\iffalse{\fi}}%
2714 \def\XINT_expr_iter:_c #1#2{\expandafter\XINT_expr_iter:_d\romannumeral0#2{{#1}}{{#2}}}%
2715 \def\XINT_expr_iter:_d #1{\ifx ^#1\xint_dothis\XINT_expr_iter:_abort\fi
2716           \ifx ?#1\xint_dothis\XINT_expr_iter:_break\fi
2717           \ifx !#1\xint_dothis\XINT_expr_iter:_omit\fi
2718           \xint_orthat{\XINT_expr_iter:_goon {#1}}}%
2719 \def\XINT_expr_iter:_abort #1!#2^~#3{#3\iffalse{\fi}}%
2720 \def\XINT_expr_iter:_break #1!#2^~#3{#1\iffalse{\fi}}%

```

```

2721 \def\XINT_expr_iter:_omit #1!#2#\expandafter\XINT_expr_iter:_b\xint_gobble_i}%
2722 \def\XINT_expr_iter:_goon #1!#2#\XINT_expr_iter:_goon_a {#1}}%
2723 \def\XINT_expr_iter:_goon_a #1#2#3~#4{\XINT_expr_iter:_b #3~{#1}}%
2724 \def\XINT_expr_iter:_Ca #1#2#3{\XINT_expr_iter:_Cc#3.{#2}}%
2725 \def\XINT_expr_iter:_Cb #1{\expandafter\XINT_expr_iter:_Cc\the\numexpr#1+\xint_c_i.}%
2726 \def\XINT_expr_iter:_Cc #1.#2{\expandafter\XINT_expr_iter:_D\romannumeral0#2{{#1}}{{#1}}{{#2}}%
2727 \def\XINT_expr_iter:_D #1{\ifx ^#1\xint_dothis\XINT_expr_iter:_abort\fi
2728 \ifx ?#1\xint_dothis\XINT_expr_iter:_break\fi
2729 \ifx !#1\xint_dothis\XINT_expr_iter:_Omit\fi
2730 \xint_orthat{\XINT_expr_iter:_Goon {#1}}}%
2731 \def\XINT_expr_iter:_Omit #1!#2#\expandafter\XINT_expr_iter:_Cb\xint_gobble_i}%
2732 \def\XINT_expr_iter:_Goon #1!#2#\XINT_expr_iter:_Goon_a {#1}}%
2733 \def\XINT_expr_iter:_Goon_a #1#2#3~#4{\XINT_expr_iter:_Cb #3~{#1}}%

```

### 27.27.7. add(), mul()

Comments under reconstruction.

These were a bit anomalous as they did not implement omit and abort keyword and the break() function (and per force then neither the n++ syntax).

At 1.4 they are simply mapped to using adequately iter(). Thus, there is small loss in efficiency, but supporting omit, abort and break is important. Using dedicated macros here would have caused also slight efficiency drop. Simpler to remove the old approach.

```

2734 \def\XINT_expr_onliteral_add
2735 {\expandafter\XINT_allexpr_add_f\romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
2736 \def\XINT_allexpr_add_f #1#2{\xint_c_ii^v `{opx}#2)\relax #1{+}{0}}%
2737 \def\XINT_expr_onliteral_mul
2738 {\expandafter\XINT_allexpr_mul_f\romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
2739 \def\XINT_allexpr_mul_f #1#2{\xint_c_ii^v `{opx}#2)\relax #1{*}{1}}%
2740 \def\XINT_expr_func_opx {\XINT:NEhook:opx \XINT_allexpr_opx \xintbareeval }%
2741 \def\XINT_flexpr_func_opx {\XINT:NEhook:opx \XINT_allexpr_opx \xintbarefloateval}%
2742 \def\XINT_iexpr_func_opx {\XINT:NEhook:opx \XINT_allexpr_opx \xintbareieval }%

```

1.4a In case of usage of omit (did I not test it? obviously I didn't as neither omit nor abort could work; and break neither), 1.4 code using (#6) syntax caused a (somewhat misleading) «missing » error message which originated in the #6. This is non-obvious problem (perhaps explained why prior to 1.4 I had not added support for omit and break() to add() and mul()...

Allowing () is not enough as it would have to be 0 or 1 depending on whether we are using add() or mul(). Hence the somewhat complicated detour (relying on precise way var\_omit and var\_abort work) via \XINT\_allexpr\_opx\_ifnotomitted.

\break() has special meaning here as it is used as last operand, not as last value. The code is very unsatisfactory and inefficient but this is hotfix for 1.4a.

```

2743 \def\XINT_allexpr_opx #1#2#3#4#5#6#7#8%
2744 {%
2745 \expandafter\XINT_expr_put_op_first
2746 \expanded \bgroup {\iffalse}\fi
2747 \XINT_expr_iter:_b {#1%
2748 \expandafter\XINT_allexpr_opx_ifnotomitted
2749 \romannumeral0#1#6\relax#7@relax !#5}#4^~{{#8}}\XINT_expr_cb_and_getop
2750 }%
2751 \def\XINT_allexpr_opx_ifnotomitted #1%
2752 {%
2753 \ifx !#1\xint_dothis{@\relax}\fi
2754 \ifx ^#1\xint_dothis{\XINTfstop. ^\relax}\fi

```

```

2755 \if ?\xintFirstItem{#1}\xint_dothis{\XINT_allexpr_opx_break{#1}}\fi
2756 \xint_orthat{\XINTfstop.{#1}}%
2757 }%
2758 \def\XINT_allexpr_opx_break #1#2\relax
2759 {%
2760 break(\expandafter\XINTfstop\expandafter.\expandafter{\xint_gobble_i#1#2})\relax
2761 }%

```

### 27.27.8. rseq()

When func\_rseq has its turn, initial segment has been scanned by oparen, the ; mimicking the rôle of a closing parenthesis, and stopping further expansion (and leaving a \xint\_c\_i left-over token since 1.4). The ; is discovered during standard parsing mode, it may be for example {} or arise from expansion as rseq does not use a delimited macro to locate it.

```

2762 \def\XINT_expr_func_rseq {\XINT_allexpr_rseq \xintbareeval }%
2763 \def\XINT_flexpr_func_rseq {\XINT_allexpr_rseq \xintbarefloateval }%
2764 \def\XINT_iiexpr_func_rseq {\XINT_allexpr_rseq \xintbareiieval }%
2765 \def\XINT_allexpr_rseq #1#2#3#4%
2766 {%
2767 \expandafter\XINT_expr_rseqx
2768 \expandafter #1\expanded{\xint_noxp{#4}}\expandafter}%
2769 \romannumeral &&@\XINT_expr_fetch_E_comma_V_equal_E_a {}%
2770 }%
2771 \def\XINT_expr_rseqx #1#2#3#4%
2772 {%
2773 \XINT:NEhook:rseq \XINT_expr_rseq\romannumeral0#1(#4)\relax {#2}#3#1%
2774 }%
2775 \def\XINT_expr_rseqy #1#2#3#4#5%
2776 {%
2777 \expandafter\XINT_expr_put_op_first
2778 \expanded \bgroup {\iffalse}\fi
2779 #2%
2780 \XINT_expr_rseq:_b {#5#4}\relax !#3#1^~{#2}\XINT_expr_cb_and_getop
2781 }%
2782 \def\XINT_expr_rseq:_b #1#2%
2783 {%
2784 \ifx +#2\xint_dothis\XINT_expr_rseq:_Ca\fi
2785 \ifx !#2!\xint_dothis\XINT_expr_rseq:_noop\fi
2786 \ifx ^#2\xint_dothis\XINT_expr_rseq:_end\fi
2787 \xint_orthat{\XINT_expr_rseq:_c}{#2}{#1}%
2788 }%
2789 \def\XINT_expr_rseq:_noop #1{\XINT_expr_rseq:_b }%
2790 \def\XINT_expr_rseq:_end #1#2~#3{\iffalse\fi}%
2791 \def\XINT_expr_rseq:_c #1#2{\expandafter\XINT_expr_rseq:_d\romannumeral0#2{{#1}}{#2}}%
2792 \def\XINT_expr_rseq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_rseq:_abort\fi
2793 \ifx ?#1\xint_dothis\XINT_expr_rseq:_break\fi
2794 \ifx !#1\xint_dothis\XINT_expr_rseq:_omit\fi
2795 \xint_orthat{\XINT_expr_rseq:_goon {#1}}}%
2796 \def\XINT_expr_rseq:_abort #1!#2^~#3{\iffalse\fi}%
2797 \def\XINT_expr_rseq:_break #1!#2^~#3{#1\iffalse\fi}%
2798 \def\XINT_expr_rseq:_omit #1!#2#{\expandafter\XINT_expr_rseq:_b\xint_gobble_i}%
2799 \def\XINT_expr_rseq:_goon #1!#2#{\XINT_expr_rseq:_goon_a {#1}}%
2800 \def\XINT_expr_rseq:_goon_a #1#2#3~#4{#1\XINT_expr_rseq:_b #3~{#1}}%

```

```

2801 \def\XINT_expr_rseq:_Ca #1#2#3{\XINT_expr_rseq:_Cc#3.{#2}}%
2802 \def\XINT_expr_rseq:_Cb #1{\expandafter\XINT_expr_rseq:_Cc\the\numexpr#1+\xint_c_i.}%
2803 \def\XINT_expr_rseq:_Cc #1.#2{\expandafter\XINT_expr_rseq:_D\romannumeral0#2{{#1}}{{#1}}{{#2}}}%
2804 \def\XINT_expr_rseq:_D #1{\ifx ^#1\xint_dothis\XINT_expr_rseq:_abort\fi
2805         \ifx ?#1\xint_dothis\XINT_expr_rseq:_break\fi
2806         \ifx !#1\xint_dothis\XINT_expr_rseq:_omit\fi
2807         \xint_orthat{\XINT_expr_rseq:_Goon {#1}}}%
2808 \def\XINT_expr_rseq:_omit #1!#2#{\expandafter\XINT_expr_rseq:_Cb\xint_gobble_i}%
2809 \def\XINT_expr_rseq:_Goon #1!#2#{\XINT_expr_rseq:_Goon_a {#1}}%
2810 \def\XINT_expr_rseq:_Goon_a #1#2#3~#4{#1\XINT_expr_rseq:_Cb #3~{#1}}%

```

### 27.27.9. iterr()

ATTENTION! at 1.4 the @ and @1 are not synonymous anymore. One *must* use @1 in iterr() context.

```

2811 \def\XINT_expr_func_iterr {\XINT_allexpr_iterr \xintbareeval }%
2812 \def\XINT_flexpr_func_iterr {\XINT_allexpr_iterr \xintbarefloateval }%
2813 \def\XINT_iiexpr_func_iterr {\XINT_allexpr_iterr \xintbareiieval }%
2814 \def\XINT_allexpr_iterr #1#2#3#4%
2815 {%
2816     \expandafter\XINT_expr_iterrx
2817     \expandafter #1\expanded{{\xintRevWithBraces{#4}}\expandafter}%
2818     \romannumeral &&@\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
2819 }%
2820 \def\XINT_expr_iterrx #1#2#3#4%
2821 {%
2822     \XINT:NEhook:iterr\XINT_expr_iterr\romannumeral0#1(#4)\relax {#2}#3#1%
2823 }%
2824 \def\XINT_expr_iterr #1#2#3#4#5%
2825 {%
2826     \expandafter\XINT_expr_put_op_first
2827     \expanded \bgroup {\iffalse}\fi
2828     \XINT_expr_iterr:_b {#5#4}\relax !#3#1^~#20?\XINT_expr_cb_and_getop
2829 }%
2830 \def\XINT_expr_iterr:_b #1#2%
2831 {%
2832     \ifx +#2\xint_dothis\XINT_expr_iterr:_Ca\fi
2833     \ifx !#2!\xint_dothis\XINT_expr_iterr:_noop\fi
2834     \ifx ^#2\xint_dothis\XINT_expr_iterr:_end\fi
2835     \xint_orthat{\XINT_expr_iterr:_c}{#2}{#1}%
2836 }%
2837 \def\XINT_expr_iterr:_noop #1{\XINT_expr_iterr:_b }%
2838 \def\XINT_expr_iterr:_end #1!#2~#3#4?{{#3}\iffalse{\fi}}%
2839 \def\XINT_expr_iterr:_c #1#2{\expandafter\XINT_expr_iterr:_d\romannumeral0#2{{#1}}{{#2}}}%
2840 \def\XINT_expr_iterr:_d #1{\ifx ^#1\xint_dothis\XINT_expr_iterr:_abort\fi
2841         \ifx ?#1\xint_dothis\XINT_expr_iterr:_break\fi
2842         \ifx !#1\xint_dothis\XINT_expr_iterr:_omit\fi
2843         \xint_orthat{\XINT_expr_iterr:_goon {#1}}}%
2844 \def\XINT_expr_iterr:_abort #1!#2^~#3?{\iffalse{\fi}}%
2845 \def\XINT_expr_iterr:_break #1!#2^~#3?{#1\iffalse{\fi}}%
2846 \def\XINT_expr_iterr:_omit #1!#2#{\expandafter\XINT_expr_iterr:_b\xint_gobble_i}%
2847 \def\XINT_expr_iterr:_goon #1!#2#{\XINT_expr_iterr:_goon_a{#1}}%
2848 \def\XINT_expr_iterr:_goon_a #1#2#3~#4?%
2849 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
2850 \expandafter\XINT_expr_iterr:_b \expanded{\xint_noxpd{#3~}\xintTrim{-2}{#1#4}}0?%
2851 }%
2852 \def\XINT_expr_iterr:_Ca #1#2#3{\XINT_expr_iterr:_Cc#3.{#2}}%
2853 \def\XINT_expr_iterr:_Cb #1{\expandafter\XINT_expr_iterr:_Cc\the\numexpr#1+\xint_c_i.}%
2854 \def\XINT_expr_iterr:_Cc #1.#2%
2855 {\expandafter\XINT_expr_iterr:_D\romannumeral0#2{{#1}}{{#1}}{{#2}}%
2856 \def\XINT_expr_iterr:_D #1{\ifx ^#1\xint_dothis\XINT_expr_iterr:_abort\fi
2857 \ifx ?#1\xint_dothis\XINT_expr_iterr:_break\fi
2858 \ifx !#1\xint_dothis\XINT_expr_iterr:_Omit\fi
2859 \xint_orthat{\XINT_expr_iterr:_Goon {#1}}}%
2860 \def\XINT_expr_iterr:_Omit #1!#2#{\expandafter\XINT_expr_iterr:_Cb\xint_gobble_i}%
2861 \def\XINT_expr_iterr:_Goon #1!#2#{\XINT_expr_iterr:_Goon_a{#1}}%
2862 \def\XINT_expr_iterr:_Goon_a #1#2#3~#4?%
2863 {%
2864 \expandafter\XINT_expr_iterr:_Cb \expanded{\xint_noxpd{#3~}\xintTrim{-2}{#1#4}}0?%
2865 }%
```

### 27.27.10. rrseq()

When func\_rrseq has its turn, initial segment has been scanned by oparen, the ; mimicking the rôle of a closing parenthesis, and stopping further expansion. #2 = \xint\_c\_i and #3 are left-over trash.

```
2866 \def\XINT_expr_func_rrseq {\XINT_allexpr_rrseq \xintbareeval }%
2867 \def\XINT_flexpr_func_rrseq {\XINT_allexpr_rrseq \xintbarefloateval }%
2868 \def\XINT_iiexpr_func_rrseq {\XINT_allexpr_rrseq \xintbareiieval }%
2869 \def\XINT_allexpr_rrseq #1#2#3#4%
2870 {%
2871 \expandafter\XINT_expr_rrseq\expandafter#1\expanded
2872 {\xint_noxpd{{#4}}{\xintRevWithBraces{#4}}\expandafter}%
2873 \romannumeral`&&\XINT_expr_fetch_E_comma_V_equal_E_a {}%
2874 }%
2875 \def\XINT_expr_rrseqx #1#2#3#4#5%
2876 {%
2877 \XINT:NEhook:rrseq\XINT_expr_rrseq\romannumeral0#1(#5)\relax {#2}{#3}#4#1%
2878 }%
2879 \def\XINT_expr_rrseqy #1#2#3#4#5#6%
2880 {%
2881 \expandafter\XINT_expr_put_op_first
2882 \expanded \bgroup {\iffalse}\fi
2883 #2\XINT_expr_rrseq:_b {#6#5}\relax !#4#1^~#30?\XINT_expr_cb_and_getop
2884 }%
2885 \def\XINT_expr_rrseq:_b #1#2%
2886 {%
2887 \ifx +#2\xint_dothis\XINT_expr_rrseq:_Ca\fi
2888 \ifx !#2!\xint_dothis\XINT_expr_rrseq:_noop\fi
2889 \ifx ^#2\xint_dothis\XINT_expr_rrseq:_end\fi
2890 \xint_orthat{\XINT_expr_rrseq:_c}{#2}{#1}%
2891 }%
2892 \def\XINT_expr_rrseq:_noop #1{\XINT_expr_rrseq:_b }%
2893 \def\XINT_expr_rrseq:_end #1#2~#3?{\iffalse{\fi}}%
2894 \def\XINT_expr_rrseq:_c #1#2{\expandafter\XINT_expr_rrseq:_d\romannumeral0#2{{#1}}{{#2}}%
2895 \def\XINT_expr_rrseq:_d #1{\ifx ^#1\xint_dothis\XINT_expr_rrseq:_abort\fi
2896 \ifx ?#1\xint_dothis\XINT_expr_rrseq:_break\fi
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```

2897 \ifx !#1\xint_dothis\xINT_expr_rrseq:_omit\fi
2898 \xint_orthat{\XINT_expr_rrseq:_goon {#1}}}%
2899 \def\xINT_expr_rrseq:_abort #1!#2^~#3?{\iffalse{\fi}}%
2900 \def\xINT_expr_rrseq:_break #1!#2^~#3?{#1\iffalse{\fi}}%
2901 \def\xINT_expr_rrseq:_omit #1!#2#\xexpandafter\xINT_expr_rrseq:_b\xint_gobble_i}%
2902 \def\xINT_expr_rrseq:_goon #1!#2#\XINT_expr_rrseq:_goon_a {#1}}%
2903 \def\xINT_expr_rrseq:_goon_a #1#2#3~#4?%
2904 {%
2905 #1\xexpandafter\xINT_expr_rrseq:_b\xexpanded{\xint_noxpd{#3~}\xintTrim{-2}{#1#4}}0?%
2906 }%
2907 \def\xINT_expr_rrseq:_Ca #1#2#3{\XINT_expr_rrseq:_Cc#3.{#2}}%
2908 \def\xINT_expr_rrseq:_Cb #1{\xexpandafter\xINT_expr_rrseq:_Cc\the\numexpr#1+\xint_c_i.}%
2909 \def\xINT_expr_rrseq:_Cc #1.#2%
2910 {\xexpandafter\xINT_expr_rrseq:_D\romannumeral0#2{{#1}}{{#1}}{{#2}}}%
2911 \def\xINT_expr_rrseq:_D #1{\ifx ^#1\xint_dothis\xINT_expr_rrseq:_abort\fi
2912 \ifx ?#1\xint_dothis\xINT_expr_rrseq:_break\fi
2913 \ifx !#1\xint_dothis\xINT_expr_rrseq:_Omit\fi
2914 \xint_orthat{\XINT_expr_rrseq:_Goon {#1}}}%
2915 \def\xINT_expr_rrseq:_Omit #1!#2#\xexpandafter\xINT_expr_rrseq:_Cb\xint_gobble_i}%
2916 \def\xINT_expr_rrseq:_Goon #1!#2#\XINT_expr_rrseq:_Goon_a {#1}}%
2917 \def\xINT_expr_rrseq:_Goon_a #1#2#3~#4?%
2918 {%
2919 #1\xexpandafter\xINT_expr_rrseq:_Cb\xexpanded{\xint_noxpd{#3~}\xintTrim{-2}{#1#4}}0?%
2920 }%
2921 \catcode`? 11

```

## 27.28. Pseudo-functions related to N-dimensional hypercubic lists

### 27.28.1. ndseq()

New with 1.4. 2020/01/23. It is derived from subsm() but instead of evaluating one expression according to one value per variable, it constructs a nested bracketed seq... this means the expression is parsed each time ! Anyway, proof of concept. Nota Bene : omit, abort, break() work !

```

2922 \def\xINT_expr_onliteral_ndseq
2923 {%
2924 \expandafter\xINT_allexpr_ndseq_f
2925 \romannumeral`&&@\XINT_expr_fetch_E_comma_V_equal_E_a {}}%
2926 }%
2927 \def\xINT_allexpr_ndseq_f #1#2{\xint_c_ii^v `{ndseqx#2)\relax #1}%
2928 \def\xINT_expr_func_ndseqx
2929 {%
2930 \expandafter\xINT_allexpr_ndseqx\expandafter\xintbareeval
2931 \expandafter{\romannumeral0\expandafter\xint_gobble_i|string}%
2932 \expandafter\xintrevwithbraces
2933 \expanded\bgroup{\iffalse}\fi\xINT_allexpr_ndseq_A\xINT_expr_oparen
2934 }%
2935 \def\xINT_flexpr_func_ndseqx
2936 {%
2937 \expandafter\xINT_allexpr_ndseqx\expandafter\xintbarefloateval
2938 \expandafter{\romannumeral0\expandafter\xint_gobble_i|string}%
2939 \expandafter\xintrevwithbraces
2940 \expanded\bgroup{\iffalse}\fi\xINT_allexpr_ndseq_A\xINT_flexpr_oparen

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```
2941 }%
2942 \def\XINT_iiexpr_func_ndseqx
2943 {%
2944   \expandafter\XINT_allexpr_ndseqx\expandafter\xintbareiieval
2945   \expandafter{\romannumeral0\expandafter\xint_gobble_i|string}%
2946   \expandafter\xintrevwithbraces
2947   \expanded\bgroup{\iffalse}\fi\XINT_allexpr_ndseq_A\XINT_iiexpr_oparen
2948 }%
2949 \def\XINT_allexpr_ndseq_A #1#2#3%
2950 {%
2951   \ifx#2\xint_c_
2952     \expandafter\XINT_allexpr_ndseq_C
2953   \else
2954     \expandafter\XINT_allexpr_ndseq_B
2955   \fi #1%
2956 }%
2957 \def\XINT_allexpr_ndseq_B #1#2#3#4=%
2958 {%
2959   {#2}{\xint_zapspace#3#4 \xint_gobble_i}%
2960   \expandafter\XINT_allexpr_ndseq_A\expandafter#1\romannumeral`&&@#1%
2961 }%
2962 \def\XINT_allexpr_ndseq_C #1#2{{#2}\iffalse{{\fi}}}%
2963 \def\XINT_allexpr_ndseqx #1#2#3#4%
2964 {%
2965   \expandafter\XINT_expr_put_op_first
2966   \expanded
2967   \bgroup
2968   \romannumeral0#1\empty
2969   \expanded{\xintReplicate{\xintLength{{#3}#2}/2}{[seq(%
2970     \xint_noxpd{#4}%
2971     \XINT_allexpr_ndseqx_a #2{#3}^^%
2972     ]}%
2973   \relax
2974   \iffalse{\fi\expandafter}\romannumeral`&&@\XINT_expr_getop
2975 }%
2976 \def\XINT_allexpr_ndseqx_a #1#2%
2977 {%
2978   \xint_gob_til^ #1\XINT_allexpr_ndseqx_e ^%
2979   \xint_noxpd{, #2=\XINTfstop.{#1}]]}\XINT_allexpr_ndseqx_a
2980 }%
2981 \def\XINT_allexpr_ndseqx_e ^#1\XINT_allexpr_ndseqx_a{}}%
```

### 27.28.2. ndmap()

New with 1.4. 2020/01/24.

```
2982 \def\XINT_expr_onliteral_ndmap #1,{\xint_c_ii^v `{ndmapx}\XINTfstop.{#1}};%
2983 \def\XINT_expr_func_ndmapx #1#2#3%
2984 {%
2985   \expandafter\XINT_allexpr_ndmapx
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

2986 \csname XINT_expr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
2987 \XINT_expr_oparen
2988 }%
2989 \def\XINT_flexpr_func_ndmapx #1#2#3%
2990 {%
2991 \expandafter\XINT_allexpr_ndmapx
2992 \csname XINT_flexpr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
2993 \XINT_flexpr_oparen
2994 }%
2995 \def\XINT_iiexpr_func_ndmapx #1#2#3%
2996 {%
2997 \expandafter\XINT_allexpr_ndmapx
2998 \csname XINT_iiexpr_func_\xint_zapspaces #3 \xint_gobble_i\endcsname
2999 \XINT_iiexpr_oparen
3000 }%
3001 \def\XINT_allexpr_ndmapx #1#2%
3002 {%
3003 \expandafter\XINT_expr_put_op_first
3004 \expanded\bgroup{\iffalse}\fi
3005 \expanded
3006 {\noexpand\XINT:NEhook:x:ndmapx
3007 \noexpand\XINT_allexpr_ndmapx_a
3008 \noexpand#1}\expandafter}%
3009 \expanded\bgroup\expandafter\XINT_allexpr_ndmap_A
3010 \expandafter#2\romannumeral`&&@#2%
3011 }%
3012 \def\XINT_allexpr_ndmap_A #1#2#3%
3013 {%
3014 \ifx#3;%
3015 \expandafter\XINT_allexpr_ndmap_B
3016 \else
3017 \xint_afterfi{\XINT_allexpr_ndmap_C#2#3}%
3018 \fi #1%
3019 }%
3020 \def\XINT_allexpr_ndmap_B #1#2%
3021 {%
3022 {#2}\expandafter\XINT_allexpr_ndmap_A\expandafter#1\romannumeral`&&@#1%
3023 }%
3024 \def\XINT_allexpr_ndmap_C #1#2#3#4%
3025 {%
3026 {#4}^\relax\iffalse{{{\fi}}}%#1#2%
3027 }%
3028 \def\XINT_allexpr_ndmapx_a #1#2#3%
3029 {%
3030 \xint_gob_til^ #3\XINT_allexpr_ndmapx_l ^%
3031 \XINT_allexpr_ndmapx_b #1{#2}{#3}%
3032 }%
3033 \def\XINT_allexpr_ndmapx_l ^#1\XINT_allexpr_ndmapx_b #2#3#4\relax
3034 {%
3035 #2\empty\xint_firstofone{#3}%
3036 }%
3037 \def\XINT_allexpr_ndmapx_b #1#2#3#4\relax

```



```

3038 {%
3039     {\iffalse}\fi\XINT_allexpr_ndmapx_c {#4\relax}#1{#2}#3^%
3040 }%
3041 \def\XINT_allexpr_ndmapx_c #1#2#3#4%
3042 {%
3043     \xint_gob_til_^ #4\XINT_allexpr_ndmapx_e ^%
3044     \XINT_allexpr_ndmapx_a #2{#3{#4}}#1%
3045     \XINT_allexpr_ndmapx_c {#1}#2{#3}%
3046 }%
3047 \def\XINT_allexpr_ndmapx_e ^#1\XINT_allexpr_ndmapx_c
3048     {\iffalse{\fi}\xint_gobble_iii}%

```

### 27.28.3. ndfillraw()

New with 1.4. 2020/01/24. J'hésite à autoriser un #1 quelconque, ou plutôt à le wrapper dans un `\xintbareval`. Mais il faut alors distinguer les trois. De toute façon les variables ne marcheraient pas donc j'hésite à mettre un wrapper automatique. Mais ce n'est pas bien d'autoriser l'injection de choses quelconques.

Pour des choses comme `ndfillraw(\xintRandomBit,[10,10])`.

Je n'aime pas le nom !. Le changer. `ndconst`? Surtout je n'aime pas que dans le premier argument il faut rajouter explicitement si nécessaire `\xintiexpr` wrap.

```

3049 \def\XINT_expr_onliteral_ndfillraw #1,{\xint_c_ii^v `{ndfillrawx}\XINTfstop.{{#1}},}%
3050 \def\XINT_expr_func_ndfillrawx #1#2#3%
3051 {%
3052     \expandafter#1\expandafter#2\expanded{{{XINT_allexpr_ndfillrawx_a #3}}}%
3053 }%
3054 \let\XINT_iiexpr_func_ndfillrawx\XINT_expr_func_ndfillrawx
3055 \let\XINT_flexpr_func_ndfillrawx\XINT_expr_func_ndfillrawx
3056 \def\XINT_allexpr_ndfillrawx_a #1#2%
3057 {%
3058     \expandafter\XINT_allexpr_ndfillrawx_b
3059     \romannumeral0\xintApply{\xintNum}{#2}^{\relax {#1}}%
3060 }%
3061 \def\XINT_allexpr_ndfillrawx_b #1#2\relax#3%
3062 {%
3063     \xint_gob_til_^ #1\XINT_allexpr_ndfillrawx_c ^%
3064     \xintReplicate{#1}{{XINT_allexpr_ndfillrawx_b #2\relax {#3}}}%
3065 }%
3066 \def\XINT_allexpr_ndfillrawx_c ^\xintReplicate #1#2%
3067 {%
3068     \expandafter\XINT_allexpr_ndfillrawx_d\xint_firstofone #2%
3069 }%
3070 \def\XINT_allexpr_ndfillrawx_d\XINT_allexpr_ndfillrawx_b \relax #1{#1}%

```

### 27.29. Other pseudo-functions: `bool()`, `togl()`, `protect()`, `qraw()`, `qint()`, `qfrac()`, `qfloat()`, `qrand()`, `random()`, `rbit()`

`bool`, `togl` and `protect` use delimited macros. They are not true functions, they turn off the parser to gather their "variable".

Modified at 1.2 (2015/10/10). Adds `qint()`, `qfrac()`, `qfloat()`.

Modified at 1.3c (2018/06/17). Adds `qraw()`. Useful to limit impact on  $\TeX$  memory from abuse of `\csname`'s storage when generating many comma separated values from a loop.

Modified at 1.3e (2019/04/05). `qfloat()` keeps a short mantissa if possible.

They allow the user to hand over quickly a big number to the parser, spaces not immediately removed but should be harmless in general. The `qraw()` does no post-processing at all apart complete expansion, useful for comma-separated values, but must be obedient to (non really documented) expected format. Each uses a delimited macro, the closing parenthesis can not emerge from expansion.

1.3b. `random()`, `grand()` Function-like syntax but with no argument currently, so let's use fast parsing which requires though the closing parenthesis to be explicit.

Attention that `qraw()` which pre-supposes knowledge of internal storage model is fragile and may break at any release.

1.4 adds `rbit()`. Short for random bit.

```

3071 \def\XINT_expr_onliteral_bool #1)%
3072   {\expandafter\XINT_expr_put_op_first\expanded{{{xintBool{#1}}}\expandafter
3073     }\romannumeral`&&\XINT_expr_getop}%
3074 \def\XINT_expr_onliteral_togl #1)%
3075   {\expandafter\XINT_expr_put_op_first\expanded{{{xintToggle{#1}}}\expandafter
3076     }\romannumeral`&&\XINT_expr_getop}%
3077 \def\XINT_expr_onliteral_protect #1)%
3078   {\expandafter\XINT_expr_put_op_first\expanded{{{detokenize{#1}}}\expandafter
3079     }\romannumeral`&&\XINT_expr_getop}%
3080 \def\XINT_expr_onliteral_qint #1)%
3081   {\expandafter\XINT_expr_put_op_first\expanded{{{xintiNum{#1}}}\expandafter
3082     }\romannumeral`&&\XINT_expr_getop}%
3083 \def\XINT_expr_onliteral_qfrac #1)%
3084   {\expandafter\XINT_expr_put_op_first\expanded{{{xintRaw{#1}}}\expandafter
3085     }\romannumeral`&&\XINT_expr_getop}%
3086 \def\XINT_expr_onliteral_qfloat #1)%
3087   {\expandafter\XINT_expr_put_op_first\expanded{{{XINTinFloatSdigits{#1}}}\expandafter
3088     }\romannumeral`&&\XINT_expr_getop}%
3089 \def\XINT_expr_onliteral_qraw #1)%
3090   {\expandafter\XINT_expr_put_op_first\expanded{{{#1}}\expandafter
3091     }\romannumeral`&&\XINT_expr_getop}%
3092 \def\XINT_expr_onliteral_random #1)%
3093   {\expandafter\XINT_expr_put_op_first\expanded{{{XINTinRandomFloatSdigits}}\expandafter
3094     }\romannumeral`&&\XINT_expr_getop}%
3095 \def\XINT_expr_onliteral_grand #1)%
3096   {\expandafter\XINT_expr_put_op_first\expanded{{{XINTinRandomFloatSixteen}}\expandafter
3097     }\romannumeral`&&\XINT_expr_getop}%
3098 \def\XINT_expr_onliteral_rbit #1)%
3099   {\expandafter\XINT_expr_put_op_first\expanded{{{xintRandBit}}\expandafter
3100     }\romannumeral`&&\XINT_expr_getop}%

```

27.30. Regular built-in functions: `num()`, `reduce()`, `preduce()`, `abs()`, `sgn()`, `frac()`, `floor()`, `ceil()`, `sqr()`, `?`, `!`, `not()`, `odd()`, `even()`, `isint()`, `isone()`, `factorial()`, `sqrt()`, `sqrtr()`, `inv()`, `round()`, `trunc()`, `float()`, `sfloat()`, `ilog10()`, `divmod()`, `mod()`, `binomial()`, `pfactorial()`, `randrange()`, `iquo()`, `irem()`, `gcd()`, `lcm()`, `max()`, `min()`, ``+`()`, ``*`()`, `all()`, `any()`, `xor()`, `len()`, `first()`, `last()`, `reversed()`, `if()`, `ifint()`, `ifone()`, `ifsgn()`, `nuple()`, `unpack()`, `flat()` and `zip()`

```

3101 \def\XINT:expr:f:one:and:opt #1#2#3!#4#5%

```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

```
3102 {%
3103     \if\relax#3\relax\expandafter\xint_firstoftwo\else
3104         \expandafter\xint_secondoftwo\fi
3105     {#4}{#5[\xintNum{#2}]}{#1}%
3106 }%
3107 \def\xint:expr:f:tacitzeroifone #1#2#3!#4#5%
3108 {%
3109     \if\relax#3\relax\expandafter\xint_firstoftwo\else
3110         \expandafter\xint_secondoftwo\fi
3111     {#4{0}}{#5{\xintNum{#2}}}{#1}%
3112 }%
3113 \def\xint:expr:f:iitacitzeroifone #1#2#3!#4%
3114 {%
3115     \if\relax#3\relax\expandafter\xint_firstoftwo\else
3116         \expandafter\xint_secondoftwo\fi
3117     {#4{0}}{#4{#2}}{#1}%
3118 }%
3119 \def\xint:expr_func_num #1#2#3%
3120 {%
3121     \expandafter #1\expandafter #2\expandafter{%
3122         \romannumeral`&&\xint:NEhook:f:one:from:one
3123         {\romannumeral`&&\xintNum#3}}%
3124 }%
3125 \let\xint:flexpr_func_num\xint:expr_func_num
3126 \let\xint:iiexpr_func_num\xint:expr_func_num
3127 \def\xint:expr_func_reduce #1#2#3%
3128 {%
3129     \expandafter #1\expandafter #2\expandafter{%
3130         \romannumeral`&&\xint:NEhook:f:one:from:one
3131         {\romannumeral`&&\xintIrr#3}}%
3132 }%
3133 \let\xint:flexpr_func_reduce\xint:expr_func_reduce
3134 \def\xint:expr_func_preduce #1#2#3%
3135 {%
3136     \expandafter #1\expandafter #2\expandafter{%
3137         \romannumeral`&&\xint:NEhook:f:one:from:one
3138         {\romannumeral`&&\xintPIrr#3}}%
3139 }%
3140 \let\xint:flexpr_func_preduce\xint:expr_func_preduce
3141 \def\xint:expr_func_abs #1#2#3%
3142 {%
3143     \expandafter #1\expandafter #2\expandafter{%
3144         \romannumeral`&&\xint:NEhook:f:one:from:one
3145         {\romannumeral`&&\xintAbs#3}}%
3146 }%
3147 \let\xint:flexpr_func_abs\xint:expr_func_abs
3148 \def\xint:iiexpr_func_abs #1#2#3%
3149 {%
3150     \expandafter #1\expandafter #2\expandafter{%
3151         \romannumeral`&&\xint:NEhook:f:one:from:one
3152         {\romannumeral`&&\xintiiAbs#3}}%
3153 }%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```

3154 \def\XINT_expr_func_sgn #1#2#3%
3155 {%
3156   \expandafter #1\expandafter #2\expandafter{%
3157     \romannumeral`&&\XINT:NEhook:f:one:from:one
3158     {\romannumeral`&&\xintSgn#3}}%
3159 }%
3160 \let\XINT_flexpr_func_sgn\XINT_expr_func_sgn
3161 \def\XINT_iiexpr_func_sgn #1#2#3%
3162 {%
3163   \expandafter #1\expandafter #2\expandafter{%
3164     \romannumeral`&&\XINT:NEhook:f:one:from:one
3165     {\romannumeral`&&\xintiSgn#3}}%
3166 }%
3167 \def\XINT_expr_func_frac #1#2#3%
3168 {%
3169   \expandafter #1\expandafter #2\expandafter{%
3170     \romannumeral`&&\XINT:NEhook:f:one:from:one
3171     {\romannumeral`&&\xintTFrac#3}}%
3172 }%
3173 \def\XINT_flexpr_func_frac #1#2#3%
3174 {%
3175   \expandafter #1\expandafter #2\expandafter{%
3176     \romannumeral`&&\XINT:NEhook:f:one:from:one
3177     {\romannumeral`&&\XINTinFloatFrac#3}}%
3178 }%
3179 \def\XINT_expr_func_floor #1#2#3%
3180 {%
3181   \expandafter #1\expandafter #2\expandafter{%
3182     \romannumeral`&&\XINT:NEhook:f:one:from:one
3183     {\romannumeral`&&\xintFloor#3}}%
3184 }%
3185 \let\XINT_flexpr_func_floor\XINT_expr_func_floor
3186 \def\XINT_iiexpr_func_floor #1#2#3%
3187 {%
3188   \expandafter #1\expandafter #2\expandafter{%
3189     \romannumeral`&&\XINT:NEhook:f:one:from:one
3190     {\romannumeral`&&\xintiFloor#3}}%
3191 }%
3192 \def\XINT_expr_func_ceil #1#2#3%
3193 {%
3194   \expandafter #1\expandafter #2\expandafter{%
3195     \romannumeral`&&\XINT:NEhook:f:one:from:one
3196     {\romannumeral`&&\xintCeil#3}}%
3197 }%
3198 \let\XINT_flexpr_func_ceil\XINT_expr_func_ceil
3199 \def\XINT_iiexpr_func_ceil #1#2#3%
3200 {%
3201   \expandafter #1\expandafter #2\expandafter{%
3202     \romannumeral`&&\XINT:NEhook:f:one:from:one

```

The floor and ceil functions in `\xintiexpr` require `protect(a/b)` or, better, `\qfrac(a/b)`; else the / will be executed first and do an integer rounded division.

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```
3203     {\romannumeral`&&\xintiCeil#3}}%
3204 }%
3205 \def\xINT_expr_func_sqr #1#2#3%
3206 {%
3207     \expandafter #1\expandafter #2\expandafter{%
3208     \romannumeral`&&\XINT:NEhook:f:one:from:one
3209     {\romannumeral`&&\xintSqr#3}}%
3210 }%
3211 \def\xINT_flexpr_func_sqr #1#2#3%
3212 {%
3213     \expandafter #1\expandafter #2\expandafter{%
3214     \romannumeral`&&\XINT:NEhook:f:one:from:one
3215     {\romannumeral`&&\XINTinFloatSqr#3}}%
3216 }%
3217 \def\xINT_iiexpr_func_sqr #1#2#3%
3218 {%
3219     \expandafter #1\expandafter #2\expandafter{%
3220     \romannumeral`&&\XINT:NEhook:f:one:from:one
3221     {\romannumeral`&&\xintiiSqr#3}}%
3222 }%
3223 \def\xINT_expr_func_? #1#2#3%
3224 {%
3225     \expandafter #1\expandafter #2\expandafter{%
3226     \romannumeral`&&\XINT:NEhook:f:one:from:one
3227     {\romannumeral`&&\xintiiIsNotZero#3}}%
3228 }%
3229 \let\xINT_flexpr_func_? \XINT_expr_func_?
3230 \let\xINT_iiexpr_func_? \XINT_expr_func_?
3231 \def\xINT_expr_func_! #1#2#3%
3232 {%
3233     \expandafter #1\expandafter #2\expandafter{%
3234     \romannumeral`&&\XINT:NEhook:f:one:from:one
3235     {\romannumeral`&&\xintiiIsZero#3}}%
3236 }%
3237 \let\xINT_flexpr_func_! \XINT_expr_func_!
3238 \let\xINT_iiexpr_func_! \XINT_expr_func_!
3239 \def\xINT_expr_func_not #1#2#3%
3240 {%
3241     \expandafter #1\expandafter #2\expandafter{%
3242     \romannumeral`&&\XINT:NEhook:f:one:from:one
3243     {\romannumeral`&&\xintiiIsZero#3}}%
3244 }%
3245 \let\xINT_flexpr_func_not \XINT_expr_func_not
3246 \let\xINT_iiexpr_func_not \XINT_expr_func_not
3247 \def\xINT_expr_func_odd #1#2#3%
3248 {%
3249     \expandafter #1\expandafter #2\expandafter{%
3250     \romannumeral`&&\XINT:NEhook:f:one:from:one
3251     {\romannumeral`&&\xintOdd#3}}%
3252 }%
3253 \let\xINT_flexpr_func_odd\xINT_expr_func_odd
3254 \def\xINT_iiexpr_func_odd #1#2#3%
```

```

3255 {%
3256     \expandafter #1\expandafter #2\expandafter{%
3257     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3258     {\romannumeral`&&@\xintiiOdd#3}}%
3259 }%
3260 \def\XINT_expr_func_even #1#2#3%
3261 {%
3262     \expandafter #1\expandafter #2\expandafter{%
3263     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3264     {\romannumeral`&&@\xintEven#3}}%
3265 }%
3266 \let\XINT_flexpr_func_even\XINT_expr_func_even
3267 \def\XINT_iiexpr_func_even #1#2#3%
3268 {%
3269     \expandafter #1\expandafter #2\expandafter{%
3270     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3271     {\romannumeral`&&@\xintiiEven#3}}%
3272 }%
3273 \def\XINT_expr_func_isint #1#2#3%
3274 {%
3275     \expandafter #1\expandafter #2\expandafter{%
3276     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3277     {\romannumeral`&&@\xintIsInt#3}}%
3278 }%
3279 \def\XINT_flexpr_func_isint #1#2#3%
3280 {%
3281     \expandafter #1\expandafter #2\expandafter{%
3282     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3283     {\romannumeral`&&@\xintFloatIsInt#3}}%
3284 }%
3285 \let\XINT_iiexpr_func_isint\XINT_expr_func_isint % ? perhaps rather always 1
3286 \def\XINT_expr_func_ison #1#2#3%
3287 {%
3288     \expandafter #1\expandafter #2\expandafter{%
3289     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3290     {\romannumeral`&&@\xintIsOne#3}}%
3291 }%
3292 \let\XINT_flexpr_func_ison\XINT_expr_func_ison
3293 \def\XINT_iiexpr_func_ison #1#2#3%
3294 {%
3295     \expandafter #1\expandafter #2\expandafter{%
3296     \romannumeral`&&@\XINT:NEhook:f:one:from:one
3297     {\romannumeral`&&@\xintiiIsOne#3}}%
3298 }%
3299 \def\XINT_expr_func_factorial #1#2#3%
3300 {%
3301     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3302     \romannumeral`&&@\XINT:NEhook:f:one:and:opt:direct
3303     \XINT:expr:f:one:and:opt #3,! \xintFac\XINTinFloatFac
3304     }}%
3305 }%
3306 \def\XINT_flexpr_func_factorial #1#2#3%

```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
3307 {%
3308     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3309     \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3310     \XINT:expr:f:one:and:opt#3,! \XINTinFloatFacdigits\XINTinFloatFac
3311     }}%
3312 }%
3313 \def\XINT_iiexpr_func_factorial #1#2#3%
3314 {%
3315     \expandafter #1\expandafter #2\expandafter{%
3316     \romannumeral`&&\XINT:NEhook:f:one:from:one
3317     {\romannumeral`&&\xintiiFac#3}}%
3318 }%
3319 \def\XINT_expr_func_sqrt #1#2#3%
3320 {%
3321     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3322     \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3323     \XINT:expr:f:one:and:opt #3,! \XINTinFloatSqrtdigits\XINTinFloatSqrt
3324     }}%
3325 }%
3326 \let\XINT_flexpr_func_sqrt\XINT_expr_func_sqrt
3327 \def\XINT_iiexpr_func_sqrt #1#2#3%
3328 {%
3329     \expandafter #1\expandafter #2\expandafter{%
3330     \romannumeral`&&\XINT:NEhook:f:one:from:one
3331     {\romannumeral`&&\xintiiSqrt#3}}%
3332 }%
3333 \def\XINT_iiexpr_func_sqrtr #1#2#3%
3334 {%
3335     \expandafter #1\expandafter #2\expandafter{%
3336     \romannumeral`&&\XINT:NEhook:f:one:from:one
3337     {\romannumeral`&&\xintiiSqrtr#3}}%
3338 }%
3339 \def\XINT_expr_func_inv #1#2#3%
3340 {%
3341     \expandafter #1\expandafter #2\expandafter{%
3342     \romannumeral`&&\XINT:NEhook:f:one:from:one
3343     {\romannumeral`&&\xintInv#3}}%
3344 }%
3345 \def\XINT_flexpr_func_inv #1#2#3%
3346 {%
3347     \expandafter #1\expandafter #2\expandafter{%
3348     \romannumeral`&&\XINT:NEhook:f:one:from:one
3349     {\romannumeral`&&\XINTinFloatInv#3}}%
3350 }%
3351 \def\XINT_expr_func_round #1#2#3%
3352 {%
3353     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3354     \romannumeral`&&\XINT:NEhook:f:tacitzeroifone:direct
3355     \XINT:expr:f:tacitzeroifone #3,! \xintiRound\xintRound
3356     }}%
3357 }%
3358 \let\XINT_flexpr_func_round\XINT_expr_func_round
```



## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

3359 \def\XINT_iiexpr_func_round #1#2#3%
3360 {%
3361     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3362         \romannumeral`&&\XINT:NEhook:f:iitacitzeroifone:direct
3363         \XINT:expr:f:iitacitzeroifone #3,! \xintiRound
3364     }}%
3365}%
3366 \def\XINT_expr_func_trunc #1#2#3%
3367 {%
3368     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3369         \romannumeral`&&\XINT:NEhook:f:tacitzeroifone:direct
3370         \XINT:expr:f:tacitzeroifone #3,! \xintiTrunc\xintTrunc
3371     }}%
3372}%
3373 \let\XINT_flexpr_func_trunc\XINT_expr_func_trunc
3374 \def\XINT_iiexpr_func_trunc #1#2#3%
3375 {%
3376     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3377         \romannumeral`&&\XINT:NEhook:f:iitacitzeroifone:direct
3378         \XINT:expr:f:iitacitzeroifone #3,! \xintiTrunc
3379     }}%
3380}%

```

Hesitation at 1.3e about using `\XINTinFloatSdigits` and `\XINTinFloatS`. Finally I add a `sfloat()` function. It helps for `xinttrig.sty`.

```

3381 \def\XINT_expr_func_float #1#2#3%
3382 {%
3383     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3384         \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3385         \XINT:expr:f:one:and:opt #3,! \XINTinFloatdigits\XINTinFloat
3386     }}%
3387}%
3388 \let\XINT_flexpr_func_float\XINT_expr_func_float

```

`float_()` was added at 1.4, as a shortcut alias to `float()` skipping the check for an optional second argument. This is useful to transfer function definitions between `\xintexpr` and `\xintfloatexpr` contexts.

No need for a similar shortcut for `sfloat()` as currently used in `xinttrig.sty` to go from `float` to `expr`: as it is used there as `sfloat(x)` with dummy `x`, it sees there is no optional argument, contrarily to for example `float(\xintexpr...\relax)` which has to allow for the inner expression to expand to an ople with two items, so does not know in which branch it is at time of definiion.

After some hesitation at 1.4e regarding guard digits mechanism the `float_()` got renamed to `float_dgt()`, but then renamed back to `float_()` to avoid a breaking change and having to document it.

Nevertheless the documentation of 1.4e mentioned `float_dgt()`... but it was still `float_()`... now changed into `float_dgt()` for real at 1.4f.

1.4f also adds private `float_dgtormax` and `sfloat_dgtormax` for matters of `xinttrig`.

```

3389 \def\XINT_expr_func_float_dgt #1#2#3%
3390 {%
3391     \expandafter #1\expandafter #2\expandafter{%
3392         \romannumeral`&&\XINT:NEhook:f:one:from:one
3393         {\romannumeral`&&\XINTinFloatdigits#3}}%
3394}%
3395 \let\XINT_flexpr_func_float_dgt\XINT_expr_func_float_dgt

```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfac*, xintexpr, *xinttrig*, *xintlog*

```

3396 % no \XINT_iiexpr_func_float_dgt
3397 \def\XINT_expr_func_float_dgtormax #1#2#3%
3398 {%
3399     \expandafter #1\expandafter #2\expandafter{%
3400     \romannumeral`&&\XINT:NEhook:f:one:from:one
3401     {\romannumeral`&&\XINTinFloatdigitsormax#3}}%
3402 }%
3403 \let\XINT_flexpr_func_float_dgtormax\XINT_expr_func_float_dgtormax
3404 \def\XINT_expr_func_sfloat #1#2#3%
3405 {%
3406     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3407     \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3408     \XINT:expr:f:one:and:opt #3,! \XINTinFloatSdigits\XINTinFloatS
3409     }}%
3410 }%
3411 \let\XINT_flexpr_func_sfloat\XINT_expr_func_sfloat
3412 % no \XINT_iiexpr_func_sfloat
3413 \def\XINT_expr_func_sfloat_dgtormax #1#2#3%
3414 {%
3415     \expandafter #1\expandafter #2\expandafter{%
3416     \romannumeral`&&\XINT:NEhook:f:one:from:one
3417     {\romannumeral`&&\XINTinFloatSdigitsormax#3}}%
3418 }%
3419 \let\XINT_flexpr_func_sfloat_dgtormax\XINT_expr_func_sfloat_dgtormax
3420 \expandafter\def\csname XINT_expr_func_ilog10\endcsname #1#2#3%
3421 {%
3422     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3423     \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3424     \XINT:expr:f:one:and:opt #3,! \xintiLogTen\XINTFloatiLogTen
3425     }}%
3426 }%
3427 \expandafter\def\csname XINT_flexpr_func_ilog10\endcsname #1#2#3%
3428 {%
3429     \expandafter #1\expandafter #2\expandafter{\expandafter{%
3430     \romannumeral`&&\XINT:NEhook:f:one:and:opt:direct
3431     \XINT:expr:f:one:and:opt #3,! \XINTFloatiLogTendigits\XINTFloatiLogTen
3432     }}%
3433 }%
3434 \expandafter\def\csname XINT_iiexpr_func_ilog10\endcsname #1#2#3%
3435 {%
3436     \expandafter #1\expandafter #2\expandafter{%
3437     \romannumeral`&&\XINT:NEhook:f:one:from:one
3438     {\romannumeral`&&\xintiLogTen#3}}%
3439 }%
3440 \def\XINT_expr_func_divmod #1#2#3%
3441 {%
3442     \expandafter #1\expandafter #2\expandafter{\romannumeral`&&%
3443     \XINT:NEhook:f:one:from:two
3444     {\romannumeral`&&\xintDivMod #3}}%
3445 }%
3446 \def\XINT_flexpr_func_divmod #1#2#3%
3447 {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```
3448 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3449 \XINT:NEhook:f:one:from:two
3450 {\romannumeral`&&@\XINTinFloatDivMod #3}}%
3451 }%
3452 \def\XINT_iiexpr_func_divmod #1#2#3%
3453 {%
3454 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3455 \XINT:NEhook:f:one:from:two
3456 {\romannumeral`&&@\xintiDivMod #3}}%
3457 }%
3458 \def\XINT_expr_func_mod #1#2#3%
3459 {%
3460 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3461 \XINT:NEhook:f:one:from:two
3462 {\romannumeral`&&@\xintMod#3}}%
3463 }%
3464 \def\XINT_flexpr_func_mod #1#2#3%
3465 {%
3466 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3467 \XINT:NEhook:f:one:from:two
3468 {\romannumeral`&&@\XINTinFloatMod#3}}%
3469 }%
3470 \def\XINT_iiexpr_func_mod #1#2#3%
3471 {%
3472 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3473 \XINT:NEhook:f:one:from:two
3474 {\romannumeral`&&@\xintiMod#3}}%
3475 }%
3476 \def\XINT_expr_func_binomial #1#2#3%
3477 {%
3478 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3479 \XINT:NEhook:f:one:from:two
3480 {\romannumeral`&&@\xintBinomial #3}}%
3481 }%
3482 \def\XINT_flexpr_func_binomial #1#2#3%
3483 {%
3484 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3485 \XINT:NEhook:f:one:from:two
3486 {\romannumeral`&&@\XINTinFloatBinomial #3}}%
3487 }%
3488 \def\XINT_iiexpr_func_binomial #1#2#3%
3489 {%
3490 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3491 \XINT:NEhook:f:one:from:two
3492 {\romannumeral`&&@\xintiBinomial #3}}%
3493 }%
3494 \def\XINT_expr_func_pfactorial #1#2#3%
3495 {%
3496 \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3497 \XINT:NEhook:f:one:from:two
3498 {\romannumeral`&&@\xintPFactorial #3}}%
3499 }%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```
3500 \def\XINT_flexpr_func_pfactorial #1#2#3%
3501 {%
3502     \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3503     \XINT:NEhook:f:one:from:two
3504     {\romannumeral`&&\XINTinFloatPFactorial #3}}%
3505 }%
3506 \def\XINT_iiexpr_func_pfactorial #1#2#3%
3507 {%
3508     \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3509     \XINT:NEhook:f:one:from:two
3510     {\romannumeral`&&\xintiiPFactorial #3}}%
3511 }%
3512 \def\XINT_expr_func_randrange #1#2#3%
3513 {%
3514     \expandafter #1\expandafter #2\expanded{{{%
3515     \XINT:expr:randrange #3,!%
3516     }}}%
3517 }%
3518 \let\XINT_flexpr_func_randrange\XINT_expr_func_randrange
3519 \def\XINT_iiexpr_func_randrange #1#2#3%
3520 {%
3521     \expandafter #1\expandafter #2\expanded{{{%
3522     \XINT:iiexpr:randrange #3,!%
3523     }}}%
3524 }%
3525 \def\XINT:expr:randrange #1#2#3!%
3526 {%
3527     \if\relax#3\relax\expandafter\xint_firstoftwo\else
3528         \expandafter\xint_secondoftwo\fi
3529     {\xintiiRandRange{\XINT:NEhook:f:one:from:one:direct\xintNum{#1}}}%
3530     {\xintiiRandRangeAtoB{\XINT:NEhook:f:one:from:one:direct\xintNum{#1}}}%
3531         {\XINT:NEhook:f:one:from:one:direct\xintNum{#2}}}%
3532     }%
3533 }%
3534 \def\XINT:iiexpr:randrange #1#2#3!%
3535 {%
3536     \if\relax#3\relax\expandafter\xint_firstoftwo\else
3537         \expandafter\xint_secondoftwo\fi
3538     {\xintiiRandRange{#1}}%
3539     {\xintiiRandRangeAtoB{#1}{#2}}%
3540 }%
3541 \def\XINT_iiexpr_func_iquo #1#2#3%
3542 {%
3543     \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3544     \XINT:NEhook:f:one:from:two
3545     {\romannumeral`&&\xintiiQuo #3}}%
3546 }%
3547 \def\XINT_iiexpr_func_irem #1#2#3%
3548 {%
3549     \expandafter #1\expandafter #2\expandafter{\romannumeral`&&@%
3550     \XINT:NEhook:f:one:from:two
3551     {\romannumeral`&&\xintiiRem #3}}%
```

## TOC

[TOC](#), [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfrc](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```
3552 }%
3553 \def\XINT_expr_func_gcd #1#2#3%
3554 {%
3555     \expandafter #1\expandafter #2\expandafter{\expandafter
3556     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_GCDof#3^}}}%
3557 }%
3558 \let\XINT_flexpr_func_gcd\XINT_expr_func_gcd
3559 \def\XINT_iexpr_func_gcd #1#2#3%
3560 {%
3561     \expandafter #1\expandafter #2\expandafter{\expandafter
3562     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iigCDof#3^}}}%
3563 }%
3564 \def\XINT_expr_func_lcm #1#2#3%
3565 {%
3566     \expandafter #1\expandafter #2\expandafter{\expandafter
3567     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_LCMof#3^}}}%
3568 }%
3569 \let\XINT_flexpr_func_lcm\XINT_expr_func_lcm
3570 \def\XINT_iexpr_func_lcm #1#2#3%
3571 {%
3572     \expandafter #1\expandafter #2\expandafter{\expandafter
3573     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iilCMof#3^}}}%
3574 }%
3575 \def\XINT_expr_func_max #1#2#3%
3576 {%
3577     \expandafter #1\expandafter #2\expandafter{\expandafter
3578     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_Maxof#3^}}}%
3579 }%
3580 \def\XINT_iexpr_func_max #1#2#3%
3581 {%
3582     \expandafter #1\expandafter #2\expandafter{\expandafter
3583     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iimaxof#3^}}}%
3584 }%
3585 \def\XINT_flexpr_func_max #1#2#3%
3586 {%
3587     \expandafter #1\expandafter #2\expandafter{\expandafter
3588     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINTinFloatMaxof#3^}}}%
3589 }%
3590 \def\XINT_expr_func_min #1#2#3%
3591 {%
3592     \expandafter #1\expandafter #2\expandafter{\expandafter
3593     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_Minof#3^}}}%
3594 }%
3595 \def\XINT_iexpr_func_min #1#2#3%
3596 {%
3597     \expandafter #1\expandafter #2\expandafter{\expandafter
3598     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iiminof#3^}}}%
3599 }%
3600 \def\XINT_flexpr_func_min #1#2#3%
3601 {%
3602     \expandafter #1\expandafter #2\expandafter{\expandafter
3603     {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINTinFloatMinof#3^}}}%
3604 }
```

```

3604 }%
3605 \expandafter
3606 \def\csname XINT_expr_func_+\endcsname #1#2#3%
3607 {%
3608     \expandafter #1\expandafter #2\expandafter{\expandafter
3609         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_Sum#3^}}}%
3610 }%
3611 \expandafter
3612 \def\csname XINT_flexpr_func_+\endcsname #1#2#3%
3613 {%
3614     \expandafter #1\expandafter #2\expandafter{\expandafter
3615         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINTinFloatSum#3^}}}%
3616 }%
3617 \expandafter
3618 \def\csname XINT_iiexpr_func_+\endcsname #1#2#3%
3619 {%
3620     \expandafter #1\expandafter #2\expandafter{\expandafter
3621         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iiSum#3^}}}%
3622 }%
3623 \expandafter
3624 \def\csname XINT_expr_func_*\endcsname #1#2#3%
3625 {%
3626     \expandafter #1\expandafter #2\expandafter{\expandafter
3627         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_Prd#3^}}}%
3628 }%
3629 \expandafter
3630 \def\csname XINT_flexpr_func_*\endcsname #1#2#3%
3631 {%
3632     \expandafter #1\expandafter #2\expandafter{\expandafter
3633         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINTinFloatPrd#3^}}}%
3634 }%
3635 \expandafter
3636 \def\csname XINT_iiexpr_func_*\endcsname #1#2#3%
3637 {%
3638     \expandafter #1\expandafter #2\expandafter{\expandafter
3639         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_iiPrd#3^}}}%
3640 }%
3641 \def\XINT_expr_func_all #1#2#3%
3642 {%
3643     \expandafter #1\expandafter #2\expandafter{\expandafter
3644         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_ANDof#3^}}}%
3645 }%
3646 \let\XINT_flexpr_func_all\XINT_expr_func_all
3647 \let\XINT_iiexpr_func_all\XINT_expr_func_all
3648 \def\XINT_expr_func_any #1#2#3%
3649 {%
3650     \expandafter #1\expandafter #2\expandafter{\expandafter
3651         {\romannumeral`&&\XINT:NEhook:f:from:delim:u\XINT_ORof#3^}}}%
3652 }%
3653 \let\XINT_flexpr_func_any\XINT_expr_func_any
3654 \let\XINT_iiexpr_func_any\XINT_expr_func_any
3655 \def\XINT_expr_func_xor #1#2#3%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
3656 {%
3657     \expandafter #1\expandafter #2\expandafter{\expandafter
3658     {\romannumeral`&&@\XINT:NEhook:f:from:delim:u\XINT_XORof#3^}}}%
3659 }%
3660 \let\XINT_flexpr_func_xor\XINT_expr_func_xor
3661 \let\XINT_iiexpr_func_xor\XINT_expr_func_xor
3662 \def\XINT_expr_func_len #1#2#3%
3663 {%
3664     \expandafter#1\expandafter#2\expandafter{\expandafter{%
3665     \romannumeral`&&@\XINT:NEhook:f:LFL\xintLength
3666     {\romannumeral\XINT:NEhook:r:check#3^}}}%
3667 }%
3668 }%
3669 \let\XINT_flexpr_func_len \XINT_expr_func_len
3670 \let\XINT_iiexpr_func_len \XINT_expr_func_len
3671 \def\XINT_expr_func_first #1#2#3%
3672 {%
3673     \expandafter #1\expandafter #2\expandafter{%
3674     \romannumeral`&&@\XINT:NEhook:f:LFL\xintFirstOne
3675     {\romannumeral\XINT:NEhook:r:check#3^}}}%
3676 }%
3677 }%
3678 \let\XINT_flexpr_func_first\XINT_expr_func_first
3679 \let\XINT_iiexpr_func_first\XINT_expr_func_first
3680 \def\XINT_expr_func_last #1#2#3%
3681 {%
3682     \expandafter #1\expandafter #2\expandafter{%
3683     \romannumeral`&&@\XINT:NEhook:f:LFL\xintLastOne
3684     {\romannumeral\XINT:NEhook:r:check#3^}}}%
3685 }%
3686 }%
3687 \let\XINT_flexpr_func_last\XINT_expr_func_last
3688 \let\XINT_iiexpr_func_last\XINT_expr_func_last
3689 \def\XINT_expr_func_reversed #1#2#3%
3690 {%
3691     \expandafter #1\expandafter #2\expandafter{%
3692     \romannumeral`&&@\XINT:NEhook:f:reverse\XINT_expr_reverse
3693     #3^^#3\xint:\xint:\xint:\xint:
3694     \xint:\xint:\xint:\xint:\xint_bye
3695     }%
3696 }%
3697 \def\XINT_expr_reverse #1#2%
3698 {%
3699     \if ^\noexpand#2%
3700         \expandafter\XINT_expr_reverse:_one_or_none\string#1.%
3701     \else
3702         \expandafter\XINT_expr_reverse:_at_least_two
3703     \fi
3704 }%
3705 \def\XINT_expr_reverse:_at_least_two #1^^{\XINT_revwbr_loop {}}}%
3706 \def\XINT_expr_reverse:_one_or_none #1%
3707 {%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

3708 \if #1\bgroup\xint_dothis\XINT_expr_reverse:_nutple\fi
3709 \if #1^\xint_dothis\XINT_expr_reverse:_nil\fi
3710 \xint_orthat\XINT_expr_reverse:_leaf
3711 }%
3712 \edef\XINT_expr_reverse:_nil #1\xint_bye{\noexpand\fi\space}%
3713 \def\XINT_expr_reverse:_leaf#1\fi #2\xint:#3\xint_bye{\fi\xint_gob_andstop_i#2}%
3714 \def\XINT_expr_reverse:_nutple%
3715 {%
3716 \expandafter\XINT_expr_reverse:_nutple_a\expandafter{\string}%
3717 }%
3718 \def\XINT_expr_reverse:_nutple_a #1^#2\xint:#3\xint_bye
3719 {%
3720 \fi\expandafter
3721 {\romannumeral0\XINT_revwbr_loop{}}#2\xint:#3\xint_bye}%
3722 }%
3723 \let\XINT_flexpr_func_reversed\XINT_expr_func_reversed
3724 \let\XINT_iiexpr_func_reversed\XINT_expr_func_reversed
3725 \def\XINT_expr_func_if #1#2#3%
3726 {%
3727 \expandafter #1\expandafter #2\expandafter{%
3728 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintiiifNotZero #3}}%
3729 }%
3730 \let\XINT_flexpr_func_if\XINT_expr_func_if
3731 \let\XINT_iiexpr_func_if\XINT_expr_func_if
3732 \def\XINT_expr_func_ifint #1#2#3%
3733 {%
3734 \expandafter #1\expandafter #2\expandafter{%
3735 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintifInt #3}}%
3736 }%
3737 \let\XINT_iiexpr_func_ifint\XINT_expr_func_ifint
3738 \def\XINT_flexpr_func_ifint #1#2#3%
3739 {%
3740 \expandafter #1\expandafter #2\expandafter{%
3741 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintifFloatInt #3}}%
3742 }%
3743 \def\XINT_expr_func_ifone #1#2#3%
3744 {%
3745 \expandafter #1\expandafter #2\expandafter{%
3746 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintifOne #3}}%
3747 }%
3748 \let\XINT_flexpr_func_ifone\XINT_expr_func_ifone
3749 \def\XINT_iiexpr_func_ifone #1#2#3%
3750 {%
3751 \expandafter #1\expandafter #2\expandafter{%
3752 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintiiifOne #3}}%
3753 }%
3754 \def\XINT_expr_func_ifsgn #1#2#3%
3755 {%
3756 \expandafter #1\expandafter #2\expandafter{%
3757 \romannumeral`&&\XINT:NEhook:branch{\romannumeral`&&\xintiiifSgn #3}}%
3758 }%
3759 \let\XINT_flexpr_func_ifsgn\XINT_expr_func_ifsgn

```

```

3760 \let\XINT_iiexpr_func_ifsgn\XINT_expr_func_ifsgn
3761 \def\XINT_expr_func_nuple #1#2#3{#1#2{{#3}}}%
3762 \let\XINT_flexpr_func_nuple\XINT_expr_func_nuple
3763 \let\XINT_iiexpr_func_nuple\XINT_expr_func_nuple
3764 \def\XINT_expr_func_unpack #1#2%#3%
3765     {\expandafter#1\expandafter#2\romannumeral0\XINT:NEhook:unpack}%
3766 \let\XINT_flexpr_func_unpack\XINT_expr_func_unpack
3767 \let\XINT_iiexpr_func_unpack\XINT_expr_func_unpack
3768 \def\XINT_expr_func_flat #1#2%#3%
3769 {%
3770     \expandafter#1\expandafter#2\expanded
3771     \XINT:NEhook:x:flatten\XINT:expr:flatten
3772 }%
3773 \let\XINT_flexpr_func_flat\XINT_expr_func_flat
3774 \let\XINT_iiexpr_func_flat\XINT_expr_func_flat
3775 \let\XINT:NEhook:x:flatten\empty
3776 \def\XINT_expr_func_zip #1#2%#3%
3777 {%
3778     \expandafter#1\expandafter#2\romannumeral`&&@%
3779     \XINT:NEhook:x:zip\XINT:expr:zip
3780 }%
3781 \let\XINT_flexpr_func_zip\XINT_expr_func_zip
3782 \let\XINT_iiexpr_func_zip\XINT_expr_func_zip
3783 \let\XINT:NEhook:x:zip\empty
3784 \def\XINT:expr:zip#1{\expandafter{\expanded\XINT_zip_A#1\xint_bye\xint_bye}}%

```

## 27.31. User declared functions

It is possible that the author actually does understand at this time the `\xintNewExpr`/`\xintdeffunc` refactored code and mechanisms for the first time since 2014: past evolutions such as the 2018 1.3 refactoring were done a bit in the fog (although they did accomplish a crucial step).

The 1.4 version of function and macro definitions is much more powerful than 1.3 one. But the mechanisms such as «omit», «abort» and «break()» in `iter()` et al. can't be translated into much else than their actual code when they potentially have to apply to non-numeric only context. The 1.4 `\xintdeffunc` is thus apparently able to digest them but its pre-parsing benefits are limited compared to simply assigning such parts of an expression to a mock-function created by `\xintNewFunction` (which creates simply a TeX macro from its substitution expression in macro parameters and add syntactic sugar to let it appear to `\xintexpr` as a genuine «function» although nothing of the syntax has really been pre-parsed.)

At 1.4 fetching the expression up to final semi-colon is done using `\XINT_expr_fetch_to_semicolon`, hence semi-colons arising in the syntax do not need to be hidden inside braces.

27.31.1	<code>\xintdeffunc</code> , <code>\xintdefiifunc</code> , <code>\xintdeffloatfunc</code>	664
27.31.2	<code>\xintdefufunc</code> , <code>\xintdefiifunc</code> , <code>\xintdeffloatufunc</code>	667
27.31.3	<code>\xintunassignexprfunc</code> , <code>\xintunassigniexprfunc</code> , <code>\xintunassignfloatexprfunc</code>	668
27.31.4	<code>\xintNewFunction</code>	668
27.31.5	Mysterious stuff	670
27.31.6	<code>\XINT_expr_redefinmacros</code>	682
27.31.7	<code>\xintNewExpr</code> , <code>\xintNewIExpr</code> , <code>\xintNewFloatExpr</code> , <code>\xintNewIIExpr</code>	683
27.31.8	<code>\xintexprSafeCatcodes</code> , <code>\xintexprRestoreCatcodes</code>	686



**27.31.1. `\xintdeffunc`, `\xintdefiifunc`, `\xintdeffloatfunc`**

**Modified at 1.2c (2015/11/16).** Note: it is possible to have same name assigned both to a variable and a function: things such as `add(f(f), f=1..10)` are possible.

**Modified at 1.2c (2015/11/16).** Function names first expanded then detokenized and cleaned of spaces.

**Modified at 1.2e (2015/11/22).** No `\detokenize` anymore on the function names. And `#1(#2)#3=#4` parameter pattern to avoid to have to worry if a `:` is there and it is active.

**Modified at 1.2f (2016/03/12).** La macro associée à la fonction ne débute plus par un `\romannumberal`, car de toute façon elle est pour emploi dans `\csname..\endcsname`.

**Modified at 1.2f (2016/03/12).** Comma separated expressions allowed (formerly this required using parenthesis `\xintdeffunc foo(x,...):=(..., ..., ...)`;

**Modified at 1.3c (2018/06/17).** Usage of `\xintexprSafeCatcodes` to be compatible with an active semi-colon at time of use; the colon was not a problem (see `##3`) already.

**Modified at 1.3e (2019/04/05).** `\xintdefefunc` variant added for functions which will expand completely if used with numeric arguments in other function definitions. They can't be used for recursive definitions.

Their functionality was merged into `\xintdeffunc` et al. at 1.4. The original macros were removed at 1.4m.

**Modified at 1.4 (2020/01/31).** Multi-letter variables can be used (with no prior declaration)

**Modified at 1.4 (2020/01/31).** The new internal data model has caused many worries initially (such as whether to allow functions with «ople» outputs in contrast to «numbers» or «nutples») but in the end all is simpler again and the refactoring of `?` and `??` in function definitions allows to fuse inert functions (allowing recursive definitions) and expanding functions (expanding completely if with numeric arguments) into a single entity.

A special situation is with functions of no variables. In that case it will be handled as an inert entity, else they would not be different from variables.

**Modified at 1.4 (2020/01/31).** Addition de la syntaxe déclarative `\xintdeffunc foo(a,b,...,*z) = ...;`

**Modified at 1.4m (2022/06/10).** Removal of the `\xintdefefunc` et al. macros deprecated at 1.4.

```

3785 \def\XINT_tmpa #1#2#3#4#5%
3786 {%
3787   \def #1##1(##2)##3={%
3788     \edef\XINT_deffunc_tmpa {##1}%
3789     \edef\XINT_deffunc_tmpa {\xint_zapspaces_o \XINT_deffunc_tmpa}%
3790     \def\XINT_deffunc_tmpb {0}%
3791     \edef\XINT_deffunc_tmpd {##2}%
3792     \edef\XINT_deffunc_tmpd {\xint_zapspaces_o\XINT_deffunc_tmpd}%
3793     \def\XINT_deffunc_tmpe {0}%
3794     \expandafter#5\romannumeral\XINT_expr_fetch_to_semicolon
3795   }% end of \xintdeffunc_a definition
3796   \def#5##1{%
3797     \def\XINT_deffunc_tmpe{##1}%
3798     \ifnum\xintLength:f:csv{\XINT_deffunc_tmpe}>\xint_c_
3799       \xintFor #####1 in {\XINT_deffunc_tmpe}\do
3800       {%
3801         \xintifForFirst{\let\XINT_deffunc_tmpe\empty}{}%
3802         \def\XINT_deffunc_tmpef{#####1}%
3803         \if*\xintFirstItem{#####1}%
3804           \xintifForLast

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

3805     {%
3806     \def\XINT_deffunc_tmpe{1}%
3807     \edef\XINT_deffunc_tmpf{\xintTrim{1}{####1}}%
3808     }%
3809     {%
3810     \edef\XINT_deffunc_tmpf{\xintTrim{1}{####1}}%
3811     \xintMessage{xintexpr}{Error}
3812     {Only the last positional argument can be variadic. Trimmed ####1 to
3813     \XINT_deffunc_tmpf}%
3814     }%
3815     \fi
3816     \XINT_expr_makedummy{\XINT_deffunc_tmpf}%
3817     \edef\XINT_deffunc_tmpd{\XINT_deffunc_tmpd{\XINT_deffunc_tmpf}}%
3818     \edef\XINT_deffunc_tmpb {\the\numexpr\XINT_deffunc_tmpb+\xint_c_i}%
3819     \edef\XINT_deffunc_tmpe {subs(\xint_noexpd\expandafter{\XINT_deffunc_tmpe},%
3820     \XINT_deffunc_tmpf=#####\XINT_deffunc_tmpb)}}%
3821     }%
3822     \fi

```

Logic at 1.4 is simplified here compared to earlier releases.

Modified at 1.4n (2025/09/05). Usage of \xintmeaning wrapper of engine's \meaning. See near \XINT\_NewExpr for explanations.

```

3823     \ifcase\XINT_deffunc_tmpb\space
3824     \expandafter\XINT_expr_defuserfunc_none\csname
3825     \else
3826     \expandafter\XINT_expr_defuserfunc\csname
3827     \fi
3828     XINT_#2_func_\XINT_deffunc_tmpe\expandafter\endcsname
3829     \csname XINT_#2_userfunc_\XINT_deffunc_tmpe\expandafter\endcsname
3830     \expandafter{\XINT_deffunc_tmpe}{#2}%
3831     \expandafter#3\csname XINT_#2_userfunc_\XINT_deffunc_tmpe\endcsname
3832     [\XINT_deffunc_tmpe]{\XINT_deffunc_tmpe}%
3833     \ifxintverbose\xintMessage {xintexpr}{Info}%
3834     {Function \XINT_deffunc_tmpe\space for \string\xint #4 parser
3835     associated to \string\XINT_#2_userfunc_\XINT_deffunc_tmpe\space
3836     with \ifxintglobaldefs global \fi meaning \expandafter\xintmeaning
3837     \csname XINT_#2_userfunc_\XINT_deffunc_tmpe\endcsname}%
3838     \fi
3839     \xintFor* ####1 in {\XINT_deffunc_tmpd}:{\xintrestorevariablesilently{####1}}%
3840     \xintexprRestoreCatcodes
3841     }% end of \xintdeffunc_b definition
3842     }%
3843     \def\xintdeffunc {\xintexprSafeCatcodes\xintdeffunc_a}%
3844     \def\xintdefiifunc {\xintexprSafeCatcodes\xintdefiifunc_a}%
3845     \def\xintdeffloatfunc {\xintexprSafeCatcodes\xintdeffloatfunc_a}%
3846     \XINT_tmpe\xintdeffunc_a {expr} \XINT_NewFunc {expr}\xintdeffunc_b
3847     \XINT_tmpe\xintdefiifunc_a {iiexpr}\XINT_NewIIFunc {iiexpr}\xintdefiifunc_b
3848     \XINT_tmpe\xintdeffloatfunc_a{flexpr}\XINT_NewFloatFunc{floatexpr}\xintdeffloatfunc_b
3849     \def\XINT_expr_defuserfunc_none #1#2#3#4%
3850     {%
3851     \XINT_global
3852     \def #1##1##2##3%
3853     {%

```

```

3854     \expandafter##1\expandafter##2\expanded{%
3855     {\XINT:NEhook:usernoargfunc\csname XINT_#4_userfunc_#3\endcsname}%
3856     }%
3857 }%
3858 }%
3859 \let\XINT:NEhook:usernoargfunc \empty
3860 \def\XINT_expr_defuserfunc #1#2#3#4%
3861 {%
3862     \if0\XINT_deffunc_tmpe
3863     \XINT_global
3864     \def #1##1##2%##3%
3865     {%
3866         \expandafter ##1\expandafter##2\expanded\bgroup{\iffalse}\fi
3867         \XINT:NEhook:userfunc{XINT_#4_userfunc_#3}#2%##3%
3868     }%
3869     \else

```

Last argument in the call signature is variadic (was prefixed by \*).

```

3870     \def #1##1{%
3871     \XINT_global\def #1####1####2%####3%
3872     {%
3873         \expandafter #####1\expandafter#####2\expanded\bgroup{\iffalse}\fi
3874         \XINT:NEhook:userfunc:argv{##1}{XINT_#4_userfunc_#3}#2%####3%
3875     }}\expandafter#1\expandafter{\the\numexpr\XINT_deffunc_tmpe-1}%
3876     \fi
3877 }%

```

Deliberate brace stripping of #3 to reveal the elements of the ople, which may be atoms i.e. numeric data such as {1}, or again oples, which means that the corresponding item was a nutple, for example it came from input syntax such as foo(1, 2, [1, 2], 3), so (up to details of raw encoding) {1}{2}{{1}{2}}{3}, which gives 4 braced arguments to macro #2.

```

3878 \def\XINT:NEhook:userfunc #1#2#3{#2#3\iffalse{{\fi}}}%

```

Here #1 indicates the number k-1 of standard positional arguments of the call signature, the kth and last one having been declared of variadic type. The braces around `\xintTrim{#1}{#4}` have the effect to gather all these remaining elements to provide a single one to the TeX macro.

For example input was foo(1,2,3,4,5) and call signature was foo(a,b,\*z). Then #4 will fetch {{1}{2}}{3}{4}{5}}, with one level of brace removal. We will have `\xintKeep{2}{{1}{2}}{3}{4}{5}}` which produces {1}{2}. Then `{\xintTrim{2}{{1}{2}}{3}{4}{5}}` which produces {{3}{4}{5}}. So the macro will be used as `\macro{1}{2}{{3}{4}{5}}` having been declared as a macro with 3 arguments.

The above comments were added in June 2021 but the code was done on January 19, 2020 for 1.4.

Note on June 10, 2021: at core level `\XINT_NewFunc` is used which is derived from `\XINT_NewExpr` which has always prepared TeX macros with non-delimited parameters. A refactoring could add a final delimiter, for example `\relax`. The macro with 3 arguments would be defined as `\def\macro#1#2#3\relax{...}` for example. Then we could transfer to TeX core processing what is achieved here via `\xintKeep/\xintTrim`, of course adding efficiency, via insertion of the delimiter. In the case of foo(1,2,3,4,5) we would have the #3 of delimited `\macro` fetch {3}{4}{5}, no brace removal, which is equivalent to current situation fetching {{3}{4}{5}} with brace removal. But let's see in case of foo(1,2,3) then. This would lead to delimited `\macro{1}{2}{3}\relax` and #3 will fetch {3}, removing one brace pair. Whereas current non-delimited `\macro` is used as `\macro{1}{2}{{3}}` from the Keep/Trim, then #3 fetches {{3}}, removing one brace pair. Not the same thing. So it seems there is a stumbling-block here to adopt such an alternative method, in relation with brace removal. Rather relieved in fact, as my head starts spinning in ople world. Seems better to stop thinking about doing something like that, and what it would imply as consequences for user

declarative interface also. Oples are dangerous to mental health, let's stick with one-ples: « named arguments in function body declaration must stand for one-ples », even the last one, although a priori it could be envisioned if foo has been declared with call signature (x,y,z) and is used with more items that z is mapped to the ople of extra elements beyond the first two ones. For my sanity I stick with my January 2020 concept of (x,y,\*z) which makes z stand for a nutple always and having to be used as such in the function body (possibly unpacked there using \*z).

```
3879 \def\XINT:NEhook:userfunc:argv #1#2#3#4%
3880   {\expandafter#3\expanded{\xintKeep{#1}{#4}{\xintTrim{#1}{#4}}}\iffalse{\fi}}%
```

### 27.31.2. \xintdefufunc, \xintdefiufunc, \xintdeffloatufunc

Added at 1.4

Modified at 1.4k (2022/05/18). [\xintexprRestoreCatcodes](#) was in only one branch of [\xint\\_defufunc\\_b](#), and as a result sanitization of catcodes via [\xintexprSafeCatcodes](#) was never reverted. That the bug remained unseen and in particular did not break compilation of user manual (where the | must be active), was a sort of unhappy miracle due to the | ending up recovering its active catcode from some ulterior [\xintdefiufunc](#) whose Safe/Restore behaved as described in the user manual, i.e. it did a restore to the state before the first unpaired Safe, and this miraculous recovery happened before breakage had happened, by sheer luck, or rather lack of luck, else I would have seen and solved the problem two years ago...

```
3881 \def\XINT_tmpa #1#2#3#4#5#6%
3882 {%
3883   \def #1##1(##2)##3={%
3884     \edef\XINT_defufunc_tmpa {##1}%
3885     \edef\XINT_defufunc_tmpa {\xint_zapspaces_o \XINT_defufunc_tmpa}%
3886     \edef\XINT_defufunc_tmpd {##2}%
3887     \edef\XINT_defufunc_tmpd {\xint_zapspaces_o\XINT_defufunc_tmpd}%
3888     \expandafter#5\romannumeral\XINT_expr_fetch_to_semicolon
3889   }% end of \xint_defufunc_a
3890   \def#5##1{%
3891     \def\XINT_defufunc_tmpc{##1}%
3892     \ifnum\xintLength:f:csv{\XINT_defufunc_tmpd}=\xint_c_i
3893       \expandafter#6%
3894     \else
3895       \xintMessage {xintexpr}{ERROR}
3896       {Universal functions must be functions of one argument only,
3897        but the declaration of \XINT_defufunc_tmpa\space
3898        has \xintLength:f:csv{\XINT_defufunc_tmpd} of them. Canceled.}%
3899       \xintexprRestoreCatcodes
3900     \fi
3901   }% end of \xint_defufunc_b
3902 \def #6{%
3903   \XINT_expr_makedummy{\XINT_defufunc_tmpd}%
3904   \edef\XINT_defufunc_tmpc {subs(\xint_noexpd\expandafter{\XINT_defufunc_tmpc},%
3905     \XINT_defufunc_tmpd=#####1)}%
3906   \expandafter\XINT_expr_defuserufunc
3907   \csname XINT_#2_func_\XINT_defufunc_tmpa\expandafter\endcsname
3908   \csname XINT_#2_userufunc_\XINT_defufunc_tmpa\expandafter\endcsname
3909   \expandafter{\XINT_defufunc_tmpa}{#2}%
3910   \expandafter#3\csname XINT_#2_userufunc_\XINT_defufunc_tmpa\endcsname
3911     [1]{\XINT_defufunc_tmpc}%
3912   \ifxintverbose\xintMessage {xintexpr}{Info}%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

```
3913     {Universal function \XINT_defufunc_tmpa\space for \string\xint #4 parser
3914     associated to \string\XINT_#2_userufunc_\XINT_defufunc_tmpa\space
3915     with \ifxintglobaldefs global \fi meaning \expandafter\xintmeaning
3916     \csname XINT_#2_userufunc_\XINT_defufunc_tmpa\endcsname}%
3917 \fi
3918 \xintexprRestoreCatcodes
3919 }% end of \xint_defufunc_c
3920 }%
3921 \def\xintdefufunc      {\xintexprSafeCatcodes\xintdefufunc_a}%
3922 \def\xintdefiifunc     {\xintexprSafeCatcodes\xintdefiifunc_a}%
3923 \def\xintdeffloatufunc {\xintexprSafeCatcodes\xintdeffloatufunc_a}%
3924 \XINT_tmpa\xintdefufunc_a {expr} \XINT_NewFunc {expr}%
3925     \xintdefufunc_b\xintdefufunc_c
3926 \XINT_tmpa\xintdefiifunc_a {iiexpr}\XINT_NewIIFunc {iiexpr}%
3927     \xintdefiifunc_b\xintdefiifunc_c
3928 \XINT_tmpa\xintdeffloatufunc_a{floatexpr}\XINT_NewFloatFunc{floatexpr}%
3929     \xintdeffloatufunc_b\xintdeffloatufunc_c
3930 \def\XINT_expr_defuserufunc #1#2#3#4%
3931 {%
3932     \XINT_global
3933     \def #1##1##2###3%
3934     {%
3935         \expandafter ##1\expandafter##2\expanded
3936         \XINT:NEhook:userufunc{XINT_#4_userufunc_#3}#2###3%
3937     }%
3938 }%
3939 \def\XINT:NEhook:userufunc #1{\XINT:expr:mapwithin}%
```

### 27.31.3. \xintunassignexprfunc, \xintunassigniiexprfunc, \xintunassignfloatexprfunc

See the `\xintunassignvar` for the embarrassing explanations why I had not done that earlier. A bit lazy here, no warning if undefining something not defined, and attention no precaution respective built-in functions.

```
3940 \def\XINT_tmpa #1{\expandafter\def\csname xintunassign#1func\endcsname ##1{%
3941     \edef\XINT_unfunc_tmpa{##1}%
3942     \edef\XINT_unfunc_tmpa {\xint_zapspaces_o\XINT_unfunc_tmpa}%
3943     \XINT_global\expandafter
3944     \let\csname XINT_#1_func_\XINT_unfunc_tmpa\endcsname\xint_undefined
3945     \XINT_global\expandafter
3946     \let\csname XINT_#1_userufunc_\XINT_unfunc_tmpa\endcsname\xint_undefined
3947     \XINT_global\expandafter
3948     \let\csname XINT_#1_userufunc_\XINT_unfunc_tmpa\endcsname\xint_undefined
3949     \ifxintverbose\xintMessage {\xintexpr}{Info}%
3950     {Function \XINT_unfunc_tmpa\space for \string\xint #1 parser now
3951     \ifxintglobaldefs globally \fi undefined.}%
3952     \fi}}%
3953 \XINT_tmpa{expr}\XINT_tmpa{iiexpr}\XINT_tmpa{floatexpr}%
```

### 27.31.4. \xintNewFunction

1.2h (2016/11/20). Syntax is `\xintNewFunction{<name>}[nb of arguments]{expression with #1, #2,... as in \xintNewExpr}`. This defines a function for all three parsers but the expression

parsing is delayed until function execution. Hence the expression admits all constructs, contrarily to `\xintNewExpr` or `\xintdeffunc`.

As the letters used for variables in `\xintdeffunc`, #1, #2, etc... can not stand for non numeric «oples», because at time of function call `f(a, b, c, ...)` how to decide if #1 stands for a or a, b etc... ? Or course «a» can be packed and thus the macro function can handle #1 as a «nutple» and for this be defined with the \* unpacking operator being applied to it.

```

3954 \def\xintNewFunction #1#2[#3]#4%
3955 {%
3956   \edef\XINT_newfunc_tmpa {#1}%
3957   \edef\XINT_newfunc_tmpa {\xint_zapspaces_o \XINT_newfunc_tmpa}%
3958   \def\XINT_newfunc_tmpb ##1##2##3##4##5##6##7##8##9{#4}%
3959   \begingroup
3960     \ifcase #3\relax
3961       \toks0{}%
3962     \or \toks0{##1}%
3963     \or \toks0{##1##2}%
3964     \or \toks0{##1##2##3}%
3965     \or \toks0{##1##2##3##4}%
3966     \or \toks0{##1##2##3##4##5}%
3967     \or \toks0{##1##2##3##4##5##6}%
3968     \or \toks0{##1##2##3##4##5##6##7}%
3969     \or \toks0{##1##2##3##4##5##6##7##8}%
3970     \else \toks0{##1##2##3##4##5##6##7##8##9}%
3971     \fi
3972     \expandafter
3973   \endgroup\expandafter
3974   \XINT_global\expandafter
3975   \def\csname XINT_expr_macrofunc_\XINT_newfunc_tmpa\expandafter\endcsname
3976   \the\toks0\expandafter{\XINT_newfunc_tmpb
3977     {\XINTfstop.{{##1}}}{\XINTfstop.{{##2}}}{\XINTfstop.{{##3}}}%
3978     {\XINTfstop.{{##4}}}{\XINTfstop.{{##5}}}{\XINTfstop.{{##6}}}%
3979     {\XINTfstop.{{##7}}}{\XINTfstop.{{##8}}}{\XINTfstop.{{##9}}}%
3980   \expandafter\XINT_expr_newfunction
3981     \csname XINT_expr_func_\XINT_newfunc_tmpa\expandafter\endcsname
3982     \expandafter{\XINT_newfunc_tmpa}\xintbareeval
3983   \expandafter\XINT_expr_newfunction
3984     \csname XINT_iiexpr_func_\XINT_newfunc_tmpa\expandafter\endcsname
3985     \expandafter{\XINT_newfunc_tmpa}\xintbareiieval
3986   \expandafter\XINT_expr_newfunction
3987     \csname XINT_flexpr_func_\XINT_newfunc_tmpa\expandafter\endcsname
3988     \expandafter{\XINT_newfunc_tmpa}\xintbarefloateval
3989   \ifxintverbose
3990     \xintMessage {xintexpr}{Info}%
3991     {Function \XINT_newfunc_tmpa\space for the expression parsers is
3992     associated to \string\XINT_expr_macrofunc_\XINT_newfunc_tmpa\space
3993     with \ifxintglobaldefs global \fi meaning \expandafter\xintmeaning
3994     \csname XINT_expr_macrofunc_\XINT_newfunc_tmpa\endcsname}%
3995   \fi
3996 }%
3997 \def\XINT_expr_newfunction #1#2#3%
3998 {%
3999   \XINT_global

```



```

4000 \def#1##1##2##3%
4001   {\expandafter ##1\expandafter ##2%
4002     \romannumeral0\XINT:NEhook:macrofunc
4003     #3{\csname XINT_expr_macrofunc_#2\endcsname##3}\relax
4004   }%
4005 }%
4006 \let\XINT:NEhook:macrofunc\empty

```

### 27.31.5. Mysterious stuff

There was an `\xintNewExpr` already in 1.07 from May 2013, which was modified in September 2013 to work with the `#` macro parameter character, and then refactored into a more powerful version in June 2014 for 1.1 release of 2014/10/28.

It is always too soon to try to comment and explain. In brief, this attempts to hack into the *purely numeric* `\xintexpr` parsers to transform them into *symbolic* parsers, allowing to do once and for all the parsing job and inherit a gigantic nested macro. Originally only f-expandable nesting. The initial motivation was that the `\csname` encapsulation impacted the string pool memory. Later this work proved to be the basis to provide support for implementing user-defined functions and it is now its main purpose.

Deep refactorings happened at 1.3 and 1.4.

At 1.3 the crucial idea of the «hook» macros was introduced, reducing considerably the preparatory work done by `\xintNewExpr`.

At 1.4 further considerable simplifications happened, and it is possible that the author currently does at long last understand the code!

The 1.3 code had serious complications with trying to identify would-be «list» arguments, distinguishing them from «single» arguments (things like parsing `#2+[[#1..[#3]..#4][#5:#6]]*#7` and convert it to a single nested f-expandable macro...)

The conversion at 1.4 is both more powerful and simpler, due in part to the new storage model which from `\csname` encapsulated comma separated values up to 1.3f became simply a braced list of braced values, and also crucially due to the possibilities opened up by usage of `\expanded` primitive.

```

4007 \catcode~ 12
4008 \def\XINT:NE:hasilde#1~#2#3\relax{\unless\if !#21\fi}%
4009 \def\XINT:NE:hashash#1{%
4010 \def\XINT:NE:hashash##1#1##2##3\relax{\unless\if !##21\fi}%
4011 }\expandafter\XINT:NE:hashash\string#%
4012 \def\XINT:NE:unpack #1{%
4013 \def\XINT:NE:unpack ##1%
4014 {%
4015   \if0\XINT:NE:hasilde ##1~!\relax
4016     \XINT:NE:hashash ##1#1!\relax 0\else
4017     \expandafter\XINT:NE:unpack:p\fi
4018   \xint_stop_atfirstofone{##1}%
4019 }\expandafter\XINT:NE:unpack\string#%
4020 \def\XINT:NE:unpack:p#1#2%
4021   {{~romannumeral0~expandafter~xint_stop_atfirstofone~expanded{#2}}}%
4022 \def\XINT:NE:f:one:from:one #1{%
4023 \def\XINT:NE:f:one:from:one ##1%
4024 {%
4025   \if0\XINT:NE:hasilde ##1~!\relax
4026     \XINT:NE:hashash ##1#1!\relax 0\else
4027     \xint_dothis\XINT:NE:f:one:from:one_a\fi
4028   \xint_orthat\XINT:NE:f:one:from:one_b

```

```

4029     ##1&&A%
4030 }}\expandafter\XINT:NE:f:one:from:one\string#%
4031 \def\XINT:NE:f:one:from:one_a\romannumeral`&&@#1#2&&A%
4032 {%
4033     \expandafter{\detokenize{\expandafter#1}#2}%
4034 }%
4035 \def\XINT:NE:f:one:from:one_b#1{%
4036 \def\XINT:NE:f:one:from:one_b\romannumeral`&&@##1##2&&A%
4037 {%
4038     \expandafter{\romannumeral`&&@%
4039         \if0\XINT:NE:hasilde ##2~!\relax
4040             \XINT:NE:hashash ##2#1!\relax 0\else
4041             \expandafter\string\fi
4042     ##1{##2}}%
4043 }}\expandafter\XINT:NE:f:one:from:one_b\string#%
4044 \def\XINT:NE:f:one:from:one:direct #1#2{\XINT:NE:f:one:from:one:direct_a #2&&A{#1}}%
4045 \def\XINT:NE:f:one:from:one:direct_a #1#2&&A#3%
4046 {%
4047     \if ##1\xint_dothis {\detokenize{#3}}\fi
4048     \if ~#1\xint_dothis {\detokenize{#3}}\fi
4049     \xint_orthat {#3}{#1#2}%
4050 }%
4051 \def\XINT:NE:f:one:from:two #1{%
4052 \def\XINT:NE:f:one:from:two ##1%
4053 {%
4054     \if0\XINT:NE:hasilde ##1~!\relax
4055         \XINT:NE:hashash ##1#1!\relax 0\else
4056         \xint_dothis\XINT:NE:f:one:from:two_a\fi
4057         \xint_orthat\XINT:NE:f:one:from:two_b ##1&&A%
4058 }}\expandafter\XINT:NE:f:one:from:two\string#%
4059 \def\XINT:NE:f:one:from:two_a\romannumeral`&&@#1#2&&A%
4060 {%
4061     \expandafter{\detokenize{\expandafter#1\expanded}{#2}}%
4062 }%
4063 \def\XINT:NE:f:one:from:two_b#1{%
4064 \def\XINT:NE:f:one:from:two_b\romannumeral`&&@##1##2##3&&A%
4065 {%
4066     \expandafter{\romannumeral`&&@%
4067         \if0\XINT:NE:hasilde ##2##3~!\relax
4068             \XINT:NE:hashash ##2##3#1!\relax 0\else
4069             \expandafter\string\fi
4070     ##1{##2}{##3}}%
4071 }}\expandafter\XINT:NE:f:one:from:two_b\string#%
4072 \def\XINT:NE:f:one:from:two:direct #1#2#3{\XINT:NE:two_fork #2&&A#3&&A#1{#2}{#3}}%
4073 \def\XINT:NE:two_fork #1#2&&A#3#4&&A{\XINT:NE:two_fork_nn#1#3}%
4074 \def\XINT:NE:two_fork_nn #1#2%
4075 {%
4076     \if #1##\xint_dothis\string\fi
4077     \if #1~\xint_dothis\string\fi
4078     \if #2##\xint_dothis\string\fi
4079     \if #2~\xint_dothis\string\fi
4080     \xint_orthat{}%

```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```
4081 }%
4082 \def\XINT:NE:f:one:and:opt:direct#1{%
4083 \def\XINT:NE:f:one:and:opt:direct##1!%
4084 {%
4085   \if0\XINT:NE:hasilde ##1~!\relax
4086     \XINT:NE:hashash ##1#1!\relax 0\else
4087     \xint_dothis\XINT:NE:f:one:and:opt_a\fi
4088     \xint_orthat\XINT:NE:f:one:and:opt_b ##1&&A%
4089 }}\expandafter\XINT:NE:f:one:and:opt:direct\string#%
4090 \def\XINT:NE:f:one:and:opt_a #1#2&&A#3#4%
4091 {%
4092   \detokenize{\romannumeral-`0\expandafter#1\expanded{#2}$XINT_expr_exclam#3#4}%$
4093 }%
4094 \def\XINT:NE:f:one:and:opt_b\XINT:expr:f:one:and:opt #1#2#3&&A#4#5%
4095 {%
4096   \if\relax#3\relax\expandafter\xint_firstoftwo\else
4097     \expandafter\xint_secondoftwo\fi
4098   {\XINT:NE:f:one:from:one:direct#4}%
4099   {\expandafter\XINT:NE:f:one:withopttoone\expandafter#5%
4100     \expanded{{\XINT:NE:f:one:from:one:direct\xintNum{#2}}}}%
4101   {#1}%
4102 }%
4103 \def\XINT:NE:f:one:withopttoone#1#2#3{\XINT:NE:two_fork #2&&A#3&&A#1[#2]{#3}}%
4104 \def\XINT:NE:f:tacitzeroifone:direct#1{%
4105 \def\XINT:NE:f:tacitzeroifone:direct##1!%
4106 {%
4107   \if0\XINT:NE:hasilde ##1~!\relax
4108     \XINT:NE:hashash ##1#1!\relax 0\else
4109     \xint_dothis\XINT:NE:f:one:and:opt_a\fi
4110     \xint_orthat\XINT:NE:f:tacitzeroifone_b ##1&&A%
4111 }}\expandafter\XINT:NE:f:tacitzeroifone:direct\string#%
4112 \def\XINT:NE:f:tacitzeroifone_b\XINT:expr:f:tacitzeroifone #1#2#3&&A#4#5%
4113 {%
4114   \if\relax#3\relax\expandafter\xint_firstoftwo\else
4115     \expandafter\xint_secondoftwo\fi
4116   {\XINT:NE:f:one:from:two:direct#4{0}}%
4117   {\expandafter\XINT:NE:f:one:from:two:direct\expandafter#5%
4118     \expanded{{\XINT:NE:f:one:from:one:direct\xintNum{#2}}}}%
4119   {#1}%
4120 }%
4121 \def\XINT:NE:f:iitacitzeroifone:direct#1{%
4122 \def\XINT:NE:f:iitacitzeroifone:direct##1!%
4123 {%
4124   \if0\XINT:NE:hasilde ##1~!\relax
4125     \XINT:NE:hashash ##1#1!\relax 0\else
4126     \xint_dothis\XINT:NE:f:iitacitzeroifone_a\fi
4127     \xint_orthat\XINT:NE:f:iitacitzeroifone_b ##1&&A%
4128 }}\expandafter\XINT:NE:f:iitacitzeroifone:direct\string#%
4129 \def\XINT:NE:f:iitacitzeroifone_a #1#2&&A#3%
4130 {%
4131   \detokenize
4132   {\romannumeral`$XINT_expr_null\expandafter#1\expanded{#2}$XINT_expr_exclam#3}%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
4133 }%
4134 \def\XINT:NE:f:iitacitzeroifone_b\XINT:expr:f:iitacitzeroifone #1#2#3&&A#4%
4135 {%
4136     \if\relax#3\relax\expandafter\xint_firstoftwo\else
4137         \expandafter\xint_secondoftwo\fi
4138     {\XINT:NE:f:one:from:two:direct#4{0}}}%
4139     {\XINT:NE:f:one:from:two:direct#4{#2}}}%
4140     {#1}%
4141 }%
4142 \def\XINT:NE:x:one:from:two #1#2#3{\XINT:NE:x:one:from:two_fork #2&&A#3&&A#1{#2}{#3}}%
4143 \def\XINT:NE:x:one:from:two_fork #1{%
4144 \def\XINT:NE:x:one:from:two_fork ##1##2&&A##3##4&&A%
4145 {%
4146     \if0\XINT:NE:hashtilde ##1##3~!\relax\XINT:NE:hashash ##1##3#1!\relax 0%
4147     \else
4148         \expandafter\XINT:NE:x:one:from:two:p
4149     \fi
4150 }}\expandafter\XINT:NE:x:one:from:two_fork\string#%
4151 \def\XINT:NE:x:one:from:two:p #1#2#3%
4152     {\~expanded{\detokenize{\expandafter#1}\~expanded{{#2}{#3}}}}}%
4153 \def\XINT:NE:x:listscl #1{%
4154 \def\XINT:NE:x:listscl ##1##2&%
4155 {%
4156     \if0\expandafter\XINT:NE:hashtilde\detokenize{##2}~!\relax
4157         \expandafter\XINT:NE:hashash\detokenize{##2}#1!\relax 0%
4158     \else
4159         \expandafter\XINT:NE:x:listscl:p
4160     \fi
4161     ##1##2&%
4162 }}\expandafter\XINT:NE:x:listscl\string#%
4163 \def\XINT:NE:x:listscl:p #1#2_#3&(#4%
4164 {%
4165     \detokenize{\expanded\XINT:expr:ListSel{{#3}{#4}}}}}%
4166 }%
4167 \def\XINT:expr:ListSel{\expandafter\XINT:expr:ListSel_i\expanded}%
4168 \def\XINT:expr:ListSel_i #1#2{\XINT_ListSel_top #2_#1&({#2}}}%
4169 \def\XINT:NE:f:reverse #1{%
4170 \def\XINT:NE:f:reverse ##1^%
4171 {%
4172     \if0\expandafter\XINT:NE:hashtilde\detokenize\expandafter{\xint_gobble_i##1}~!\relax
4173         \expandafter\XINT:NE:hashash\detokenize{##1}#1!\relax 0%
4174     \else
4175         \expandafter\XINT:NE:f:reverse:p
4176     \fi
4177     ##1^%
4178 }}\expandafter\XINT:NE:f:reverse\string#%
4179 \def\XINT:NE:f:reverse:p #1^#2\xint_bye
4180 {%
4181     \expandafter\XINT:NE:f:reverse:p_i\expandafter{\xint_gobble_i#1}%
4182 }%
4183 \def\XINT:NE:f:reverse:p_i #1%
4184 {%
```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

4185 \detokenize{\romannumeral0\XINT:expr:f:reverse{#{1}}}%
4186 }%
4187 \def\XINT:expr:f:reverse{\expandafter\XINT:expr:f:reverse_i\expanded}%
4188 \def\XINT:expr:f:reverse_i #1%
4189 {%
4190 \XINT_expr_reverse #1^^#1\xint:\xint:\xint:\xint:
4191 \xint:\xint:\xint:\xint:\xint_bye
4192 }%
4193 \def\XINT:NE:f:from:delim:u #1{%
4194 \def\XINT:NE:f:from:delim:u ##1##2^%
4195 {%
4196 \if0\expandafter\XINT:NE:hashtilde\detokenize{##2}~!\relax
4197 \expandafter\XINT:NE:hashash\detokenize{##2}#1!\relax 0%
4198 \xint_afterfi{\expandafter\XINT_foof:checkifnumber\expandafter##1\string}%
4199 \else
4200 \xint_afterfi{\XINT:NE:f:from:delim:u:p##1\empty}%
4201 \fi
4202 ##2^%
4203 }%\expandafter\XINT:NE:f:from:delim:u\string#%
4204 \def\XINT:NE:f:from:delim:u:p #1#2^%
4205 {%
4206 \detokenize
4207 {\expandafter\XINT:foof:checkifnumber\expandafter#1~\expanded{#2}$XINT_expr_caret%$
4208 }%
4209 \def\XINT:foof:checkifnumber#1{\expandafter\XINT_foof:checkifnumber\expandafter#1\string}%
4210 \def\XINT:NE:f:LFL#1#2{\expandafter\XINT:NE:f:LFL_a\expandafter#1#2\XINT:NE:f:LFL_a}%
4211 \def\XINT:NE:f:LFL_a#1#2%
4212 {%
4213 \if#2i\else\expandafter\XINT:NE:f:LFL_p
4214 \fi #1%
4215 }%
4216 \def\XINT:NE:r:check#1{%
4217 \def\XINT:NE:r:check##1\XINT:NE:f:LFL_a
4218 {%
4219 \if0\expandafter\XINT:NE:hashtilde\detokenize{##1}~!\relax%
4220 \expandafter\XINT:NE:hashash\detokenize{##1}#1!\relax 0%
4221 \else
4222 \expandafter\XINT:NE:r:check:p
4223 \fi
4224 1\expandafter{\romannumeral\XINT:NE:saved:r:check##1}%
4225 }%\expandafter\XINT:NE:r:check\string#%
4226 \def\XINT:NE:r:check:p 1\expandafter#1{\XINT:NE:r:check:p_i#1}%
4227 \def\XINT:NE:r:check:p_i\romannumeral\XINT:NE:saved:r:check{\XINT:NE:r:check:p_ii\empty}%
4228 \def\XINT:NE:r:check:p_ii#1^%
4229 {%
4230 5~\expanded{{~\romannumeral~\XINT:NE:saved:r:check#1$XINT_expr_caret}}}%$
4231 }%
4232 \def\XINT:NE:f:LFL_p#1%
4233 {%
4234 \detokenize{\romannumeral`$XINT_expr_null\expandafter#1}%$
4235 }%
4236 \catcode`- 11

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```
4237 \def\XINT:NE:exec_? #1#2%
4238 {%
4239   \XINT:NE:exec_?_b #2&&A#1{#2}%
4240 }%
4241 \def\XINT:NE:exec_?_b #1{%
4242 \def\XINT:NE:exec_?_b ##1&&A%
4243 {%
4244   \if0\XINT:NE:hasilde ##1~!\relax
4245     \XINT:NE:hashash ##1#1!\relax 0%
4246     \xint_dothis\XINT:NE:exec_?:x\fi
4247     \xint_orthat\XINT:NE:exec_?:p
4248 }%\expandafter\XINT:NE:exec_?_b\string#%
4249 \def\XINT:NE:exec_?:x #1#2#3%
4250 {%
4251   \expandafter\XINT_expr_check_-_after?\expandafter#1%
4252   \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifnotzero#3%
4253 }%
4254 \def\XINT:NE:exec_?:p #1#2#3#4#5%
4255 {%
4256   \csname XINT_expr_func_*If\expandafter\endcsname
4257   \romannumeral`&&@#2\XINTfstop.{#3},{#4},{#5})%
4258 }%
4259 \expandafter\def\csname XINT_expr_func_*If\endcsname #1#2#3%
4260 {%
4261   #1#2{~expanded{~xintiiifNotZero#3}}%
4262 }%
4263 \def\XINT:NE:exec_?? #1#2#3%
4264 {%
4265   \XINT:NE:exec_??_b #2&&A#1{#2}%
4266 }%
4267 \def\XINT:NE:exec_??_b #1{%
4268 \def\XINT:NE:exec_??_b ##1&&A%
4269 {%
4270   \if0\XINT:NE:hasilde ##1~!\relax
4271     \XINT:NE:hashash ##1#1!\relax 0%
4272     \xint_dothis\XINT:NE:exec_??:x\fi
4273     \xint_orthat\XINT:NE:exec_??:p
4274 }%\expandafter\XINT:NE:exec_??_b\string#%
4275 \def\XINT:NE:exec_??:x #1#2#3%
4276 {%
4277   \expandafter\XINT_expr_check_-_after?\expandafter#1%
4278   \romannumeral`&&@\expandafter\XINT_expr_getnext\romannumeral0\xintiiifsgn#3%
4279 }%
4280 \def\XINT:NE:exec_??:p #1#2#3#4#5#6%
4281 {%
4282   \csname XINT_expr_func_*IfSgn\expandafter\endcsname
4283   \romannumeral`&&@#2\XINTfstop.{#3},{#4},{#5},{#6})%
4284 }%
4285 \expandafter\def\csname XINT_expr_func_*IfSgn\endcsname #1#2#3%
4286 {%
4287   #1#2{~expanded{~xintiiifSgn#3}}%
4288 }%
```

## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrac*, xintexpr, *xinttrig*, *xintlog*

```
4289 \catcode`- 12
4290 \def\XINT:NE:branch #1%
4291 {%
4292   \if0\XINT:NE:hasilde #1~!\relax 0\else
4293     \xint_dothis\XINT:NE:branch_a\fi
4294     \xint_orthat\XINT:NE:branch_b #1&&A%
4295 }%
4296 \def\XINT:NE:branch_a\romannumeral`&&@#1#2&&A%
4297 {%
4298   \expandafter{\detokenize{\expandafter#1\expanded}{#2}}%
4299 }%
4300 \def\XINT:NE:branch_b#1{%
4301 \def\XINT:NE:branch_b\romannumeral`&&@##1##2##3&&A%
4302 {%
4303   \expandafter{\romannumeral`&&@%
4304     \if0\XINT:NE:hasilde ##2~!\relax
4305       \XINT:NE:hashash ##2#1!\relax 0\else
4306       \expandafter\string\fi
4307     ##1{##2}##3}%
4308 }}\expandafter\XINT:NE:branch_b\string#%
4309 \def\XINT:NE:seqx#1{%
4310 \def\XINT:NE:seqx\XINT_allexpr_seqx##1##2%
4311 {%
4312   \if 0\expandafter\XINT:NE:hasilde\detokenize{##2}~!\relax
4313     \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
4314   \else
4315     \expandafter\XINT:NE:seqx:p
4316   \fi \XINT_allexpr_seqx{##1}{##2}%
4317 }}\expandafter\XINT:NE:seqx\string#%
4318 \def\XINT:NE:seqx:p\XINT_allexpr_seqx #1#2#3#4%
4319 {%
4320   \expandafter\XINT_expr_put_op_first
4321   \expanded {%
4322     {%
4323       \detokenize
4324       {%
4325         \expanded\bgroup
4326         \expanded
4327         {\xint_noxpd{\XINT_expr_seq:_b{#1#4\relax $XINT_expr_exclam #3}}%
4328           #2$XINT_expr_caret}%
4329       }%
4330     }%
4331     \expandafter}\romannumeral`&&@\XINT_expr_getop
4332 }%
4333 \def\XINT:NE:opx#1{%
4334 \def\XINT:NE:opx\XINT_allexpr_opx ##1##2##3##4%##5##6##7##8%
4335 {%
4336   \if 0\expandafter\XINT:NE:hasilde\detokenize{##4}~!\relax
4337     \expandafter\XINT:NE:hashash \detokenize{##4}#1!\relax 0%
4338   \else
4339     \expandafter\XINT:NE:opx:p
4340   \fi \XINT_allexpr_opx ##1{##2}{##3}{##4}% en fait ##2 = \xint_c_, ##3 = \relax
```

```

4341 }}\expandafter\XINT:NE:opx\string#%
4342 \def\XINT:NE:opx:p\XINT_allexpr_opx #1#2#3#4#5#6#7#8%
4343 {%
4344     \expandafter\XINT_expr_put_op_first
4345     \expanded {%
4346         {%
4347             \detokenize
4348             {%
4349                 \expanded\bgroup
4350                 \expanded{\xint_noxp{\XINT_expr_iter:_b
4351                     {#1}\expandafter\XINT_allexpr_opx_ifnotomitted
4352                     \romannumeral0#1#6\relax#7@\relax $XINT_expr_exclam #5}}%
4353                     #4$XINT_expr_caret$XINT_expr_tilde{{#8}}}%$
4354             }%
4355         }%
4356     \expandafter}\romannumeral`&&@\XINT_expr_getop
4357 }%
4358 \def\XINT:NE:iter{\expandafter\XINT:NE:itery\expandafter}%
4359 \def\XINT:NE:itery#1{%
4360 \def\XINT:NE:itery\XINT_expr_itery##1##2%
4361 {%
4362     \if 0\expandafter\XINT:NE:hasilde\detokenize{##1##2}~!\relax
4363     \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
4364     \else
4365         \expandafter\XINT:NE:itery:p
4366     \fi \XINT_expr_itery{##1}{##2}%
4367 }}\expandafter\XINT:NE:itery\string#%
4368 \def\XINT:NE:itery:p\XINT_expr_itery #1#2#3#4#5%
4369 {%
4370     \expandafter\XINT_expr_put_op_first
4371     \expanded {%
4372         {%
4373             \detokenize
4374             {%
4375                 \expanded\bgroup
4376                 \expanded{\xint_noxp{\XINT_expr_iter:_b {#5#4\relax $XINT_expr_exclam #3}}%
4377                 #1$XINT_expr_caret$XINT_expr_tilde{#2}}}%$
4378             }%
4379         }%
4380     \expandafter}\romannumeral`&&@\XINT_expr_getop
4381 }%
4382 \def\XINT:NE:rseq{\expandafter\XINT:NE:rseqy\expandafter}%
4383 \def\XINT:NE:rseqy#1{%
4384 \def\XINT:NE:rseqy\XINT_expr_rseqy##1##2%
4385 {%
4386     \if 0\expandafter\XINT:NE:hasilde\detokenize{##1##2}~!\relax
4387     \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
4388     \else
4389         \expandafter\XINT:NE:rseqy:p
4390     \fi \XINT_expr_rseqy{##1}{##2}%
4391 }}\expandafter\XINT:NE:rseqy\string#%
4392 \def\XINT:NE:rseqy:p\XINT_expr_rseqy #1#2#3#4#5%

```

```

4393 {%
4394   \expandafter\XINT_expr_put_op_first
4395   \expanded {%
4396     {%
4397       \detokenize
4398       {%
4399         \expanded\bgroup
4400         \expanded{#2\xint_noxp{\XINT_expr_rseq:_b {#5#4\relax $XINT_expr_exclam #3}}%
4401           #1$XINT_expr_caret$XINT_expr_tilde{#2}}%$
4402       }%
4403     }%
4404   \expandafter}\romannumeral`&&@\XINT_expr_getop
4405 }%
4406 \def\XINT:NE:iterr{\expandafter\XINT:NE:iterr\expandafter}%
4407 \def\XINT:NE:iterr#1{%
4408 \def\XINT:NE:iterr\XINT_expr_iterr##1##2%
4409 {%
4410   \if 0\expandafter\XINT:NE:hasilde\detokenize{##1##2}~!\relax
4411     \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
4412   \else
4413     \expandafter\XINT:NE:iterr:p
4414   \fi \XINT_expr_iterr{##1}{##2}%
4415 }%\expandafter\XINT:NE:iterr\string#%
4416 \def\XINT:NE:iterr:p\XINT_expr_iterr #1#2#3#4#5%
4417 {%
4418   \expandafter\XINT_expr_put_op_first
4419   \expanded {%
4420     {%
4421       \detokenize
4422       {%
4423         \expanded\bgroup
4424         \expanded{\xint_noxp{\XINT_expr_iterr:_b {#5#4\relax $XINT_expr_exclam #3}}%
4425           #1$XINT_expr_caret$XINT_expr_tilde #20$XINT_expr_qmark}%
4426       }%
4427     }%
4428   \expandafter}\romannumeral`&&@\XINT_expr_getop
4429 }%
4430 \def\XINT:NE:rrseq{\expandafter\XINT:NE:rrseq\expandafter}%
4431 \def\XINT:NE:rrseq#1{%
4432 \def\XINT:NE:rrseq\XINT_expr_rrseq##1##2%
4433 {%
4434   \if 0\expandafter\XINT:NE:hasilde\detokenize{##1##2}~!\relax
4435     \expandafter\XINT:NE:hashash \detokenize{##1##2}#1!\relax 0%
4436   \else
4437     \expandafter\XINT:NE:rrseq:p
4438   \fi \XINT_expr_rrseq{##1}{##2}%
4439 }%\expandafter\XINT:NE:rrseq\string#%
4440 \def\XINT:NE:rrseq:p\XINT_expr_rrseq #1#2#3#4#5#6%
4441 {%
4442   \expandafter\XINT_expr_put_op_first
4443   \expanded {%
4444     {%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrac, xintexpr, xinttrig, xintlog

```
4445 \detokenize
4446 {%
4447 \expanded\bgroup
4448 \expanded{#2\xint_noxpd{\XINT_expr_rrseq:_b {#6#5\relax $XINT_expr_exclam #4}}}%
4449 #1$XINT_expr_caret$XINT_expr_tilde #30$XINT_expr_qmark}%
4450 }%
4451 }%
4452 \expandafter}\romannumeral`&&@\XINT_expr_getop
4453 }%
4454 \def\XINT:NE:x:toblist#1{%
4455 \def\XINT:NE:x:toblist\XINT:expr:toblistwith##1##2%
4456 {%
4457 \if 0\expandafter\XINT:NE:hasilde\detokenize{##2}~!\relax
4458 \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
4459 \else
4460 \expandafter\XINT:NE:x:toblist:p
4461 \fi \XINT:expr:toblistwith{##1}{##2}%
4462 }}\expandafter\XINT:NE:x:toblist\string#%
4463 \def\XINT:NE:x:toblist:p\XINT:expr:toblistwith #1#2{\XINTfstop.{#2}}}%
4464 \def\XINT:NE:x:flatten#1{%
4465 \def\XINT:NE:x:flatten\XINT:expr:flatten##1%
4466 {%
4467 \if 0\expandafter\XINT:NE:hasilde\detokenize{##1}~!\relax
4468 \expandafter\XINT:NE:hashash \detokenize{##1}#1!\relax 0%
4469 \else
4470 \expandafter\XINT:NE:x:flatten:p
4471 \fi \XINT:expr:flatten{##1}%
4472 }}\expandafter\XINT:NE:x:flatten\string#%
4473 \def\XINT:NE:x:flatten:p\XINT:expr:flatten #1%
4474 {%
4475 {{%
4476 \detokenize
4477 {%
4478 \expandafter\XINT:expr:flatten_checkempty
4479 \detokenize\expandafter{\expanded{#1}}$XINT_expr_caret%$
4480 }%
4481 }}%
4482 }%
4483 \def\XINT:NE:x:zip#1{%
4484 \def\XINT:NE:x:zip\XINT:expr:zip##1%
4485 {%
4486 \if 0\expandafter\XINT:NE:hasilde\detokenize{##1}~!\relax
4487 \expandafter\XINT:NE:hashash \detokenize{##1}#1!\relax 0%
4488 \else
4489 \expandafter\XINT:NE:x:zip:p
4490 \fi \XINT:expr:zip{##1}%
4491 }}\expandafter\XINT:NE:x:zip\string#%
4492 \def\XINT:NE:x:zip:p\XINT:expr:zip #1%
4493 {%
4494 \expandafter{%
4495 \detokenize
4496 {%
```



## TOC

TOC, *xintkernel*, *xinttools*, *xintcore*, *xint*, *xintbinhex*, *xintgcd*, *xintfrac*, *xintseries*, *xintcfrc*, xintexpr, *xinttrig*, *xintlog*

```
4497 \expanded\expandafter\XINT_zip_A\expanded{#1}\xint_bye\xint_bye
4498 }%
4499 }%
4500 }%
4501 \def\XINT:NE:x:mapwithin#1{%
4502 \def\XINT:NE:x:mapwithin\XINT:expr:mapwithin ##1##2%
4503 {%
4504 \if 0\expandafter\XINT:NE:hasilde\detokenize{##2}~!\relax
4505 \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
4506 \else
4507 \expandafter\XINT:NE:x:mapwithin:p
4508 \fi \XINT:expr:mapwithin {##1}{##2}%
4509 }%\expandafter\XINT:NE:x:mapwithin\string#%
4510 \def\XINT:NE:x:mapwithin:p \XINT:expr:mapwithin #1#2%
4511 {%
4512 {{%
4513 \detokenize
4514 {%
```

Attention (2022/06/10) I do not remember why I left these two commented lines which docstrip will not remove, I hope this is not a forgotten left-over from some debugging session.

```
4515 %% \expanded
4516 %% {%
4517 \expandafter\XINT:expr:mapwithin_checkempty
4518 \expanded{noexpand#1$XINT_expr_exclam\expandafter}%$
4519 \detokenize\expandafter{\expanded{#2}}$XINT_expr_caret%$
```

This is is the matching one.

```
4520 %% }%
4521 }%
4522 }%
4523 }%
4524 \def\XINT:NE:x:ndmapx#1{%
4525 \def\XINT:NE:x:ndmapx\XINT_allexpr_ndmapx_a ##1##2^%
4526 {%
4527 \if 0\expandafter\XINT:NE:hasilde\detokenize{##2}~!\relax
4528 \expandafter\XINT:NE:hashash \detokenize{##2}#1!\relax 0%
4529 \else
4530 \expandafter\XINT:NE:x:ndmapx:p
4531 \fi \XINT_allexpr_ndmapx_a ##1##2^%
4532 }%\expandafter\XINT:NE:x:ndmapx\string#%
4533 \def\XINT:NE:x:ndmapx:p #1#2#3^ \relax
4534 {%
4535 \detokenize
4536 {%
4537 \expanded{%
4538 \expandafter#1\expandafter#2\expanded{#3}$XINT_expr_caret\relax %$
4539 }%
4540 }%
4541 }%
```

Attention here that user function names may contain digits, so we don't use a `\detokenize` or `~` approach.

This syntax means that a function defined by `\xintdeffunc` never expands when used in another definition, so it can implement recursive definitions.

`\XINT:NE:userefunc` et al. added at 1.3e.

I added at `\xintdefefunc`, `\xintdefiiefunc`, `\xintdeffloatefunc` at 1.3e to on the contrary expand if possible (i.e. if used only with numeric arguments) in another definition.

The `\XINTusefunc` uses `\expanded`. Its ancestor `\xintExpandArgs` (*xinttools* 1.3) had some more primitive f-expansion technique.

```

4542 \def\xINTusenoargfunc #1%
4543 {%
4544   0\csname #1\endcsname
4545 }%
4546 \def\xINT:NE:usernoargfunc\csname #1\endcsname
4547 {%
4548   ~romannumeral~XINTusenoargfunc{#1}%
4549 }%
4550 \def\xINTusefunc #1%
4551 {%
4552   0\csname #1\expandafter\endcsname\expanded
4553 }%
4554 \def\xINT:NE:usefunc #1#2#3%
4555 {%
4556   ~romannumeral~XINTusefunc{#1}{#3}\iffalse{\fi}%
4557 }%
4558 \def\xINTuseufunc #1%
4559 {%
4560   \expanded\expandafter\xINT:expr:mapwithin\csname #1\expandafter\endcsname\expanded
4561 }%
4562 \def\xINT:NE:useufunc #1#2#3%
4563 {%
4564   {\~expanded~XINTuseufunc{#1}{#3}}}%
4565 }%
4566 \def\xINT:NE:userfunc #1{%
4567 \def\xINT:NE:userfunc ##1##2##3%
4568 {%
4569   \if0\expandafter\xINT:NE:hashtilde\detokenize{##3}~!\relax
4570     \expandafter\xINT:NE:hashash\detokenize{##3}#1!\relax 0%
4571     \expandafter\xINT:NE:userfunc_x
4572   \else
4573     \expandafter\xINT:NE:usefunc
4574   \fi {##1}{##2}{##3}%
4575 }%\expandafter\xINT:NE:userfunc\string#%
4576 \def\xINT:NE:userfunc_x #1#2#3{#2#3\iffalse{\fi}}%
4577 \def\xINT:NE:userufunc #1{%
4578 \def\xINT:NE:userufunc ##1##2##3%
4579 {%
4580   \if0\expandafter\xINT:NE:hashtilde\detokenize{##3}~!\relax
4581     \expandafter\xINT:NE:hashash\detokenize{##3}#1!\relax 0%
4582     \expandafter\xINT:NE:userufunc_x
4583   \else
4584     \expandafter\xINT:NE:useufunc
4585   \fi {##1}{##2}{##3}%
4586 }%\expandafter\xINT:NE:userufunc\string#%
4587 \def\xINT:NE:userufunc_x #1{\xINT:expr:mapwithin}%
4588 \def\xINT:NE:macrofunc #1#2%

```

```

4589   {\expandafter\XINT:NE:macrofunc:a\string#1#2\empty&}%
4590 \def\XINT:NE:macrofunc:a#1\csname #2\endcsname#3&%
4591   {\~XINTusemacrofunc{#1}{#2}{#3}}}%
4592 \def\XINTusemacrofunc #1#2#3%
4593 {%
4594   \romannumeral0\expandafter\xint_stop_atfirstofone
4595   \romannumeral0#1\csname #2\endcsname#3\relax
4596 }%

```

### 27.31.6. \XINT\_expr\_redefinemacros

Completely refactored at 1.3.

Again refactored at 1.4. The availability of `\expanded` allows more powerful mechanisms and more importantly I better thought out the root problems caused by the handling of list operations in this context and this helped simplify considerably the code.

```

4597 \catcode`- 11
4598 \def\XINT_expr_redefinemacros {%
4599   \let\XINT:NEhook:unpack          \XINT:NE:unpack
4600   \let\XINT:NEhook:f:one:from:one  \XINT:NE:f:one:from:one
4601   \let\XINT:NEhook:f:one:from:one:direct \XINT:NE:f:one:from:one:direct
4602   \let\XINT:NEhook:f:one:from:two   \XINT:NE:f:one:from:two
4603   \let\XINT:NEhook:f:one:from:two:direct \XINT:NE:f:one:from:two:direct
4604   \let\XINT:NEhook:x:one:from:two   \XINT:NE:x:one:from:two
4605   \let\XINT:NEhook:f:one:and:opt:direct \XINT:NE:f:one:and:opt:direct
4606   \let\XINT:NEhook:f:tacitzeroifone:direct \XINT:NE:f:tacitzeroifone:direct
4607   \let\XINT:NEhook:f:iitacitzeroifone:direct \XINT:NE:f:iitacitzeroifone:direct
4608   \let\XINT:NEhook:x:listscl       \XINT:NE:x:listscl
4609   \let\XINT:NEhook:f:reverse        \XINT:NE:f:reverse
4610   \let\XINT:NEhook:f:from:delim:u   \XINT:NE:f:from:delim:u
4611   \let\XINT:NEhook:f:LFL            \XINT:NE:f:LFL
4612   \let\XINT:NEhook:r:check          \XINT:NE:r:check
4613   \let\XINT:NEhook:branch           \XINT:NE:branch
4614   \let\XINT:NEhook:seqx             \XINT:NE:seqx
4615   \let\XINT:NEhook:opx              \XINT:NE:opx
4616   \let\XINT:NEhook:rseq             \XINT:NE:rseq
4617   \let\XINT:NEhook:iter             \XINT:NE:iter
4618   \let\XINT:NEhook:rrseq            \XINT:NE:rrseq
4619   \let\XINT:NEhook:iterr            \XINT:NE:iterr
4620   \let\XINT:NEhook:x:toblist        \XINT:NE:x:toblist
4621   \let\XINT:NEhook:x:flatten        \XINT:NE:x:flatten
4622   \let\XINT:NEhook:x:zip            \XINT:NE:x:zip
4623   \let\XINT:NEhook:x:mapwithin      \XINT:NE:x:mapwithin
4624   \let\XINT:NEhook:x:ndmapx         \XINT:NE:x:ndmapx
4625   \let\XINT:NEhook:userfunc         \XINT:NE:userfunc
4626   \let\XINT:NEhook:userufunc        \XINT:NE:userufunc
4627   \let\XINT:NEhook:usernoargfunc    \XINT:NE:usernoargfunc
4628   \let\XINT:NEhook:macrofunc        \XINT:NE:macrofunc
4629   \def\XINTinRandomFloatSdigits{\~XINTinRandomFloatSdigits}%
4630   \def\XINTinRandomFloatSixteen{\~XINTinRandomFloatSixteen}%
4631   \def\xintiiRandRange{\~xintiiRandRange}%
4632   \def\xintiiRandRangeAtoB{\~xintiiRandRangeAtoB}%
4633   \def\xintRandBit{\~xintRandBit}%
4634   \let\XINT_expr_exec_? \XINT:NE:exec_?

```

```

4635 \let\XINT_expr_exec_?? \XINT:NE:exec_??
4636 \def\XINT_expr_op_? {\XINT_expr_op_?{\XINT_expr_op_-xii\XINT_expr_oparen}}%
4637 \def\XINT_flexpr_op_?{\XINT_expr_op_?{\XINT_flexpr_op_-xii\XINT_flexpr_oparen}}%
4638 \def\XINT_iexpr_op_?{\XINT_expr_op_?{\XINT_iexpr_op_-xii\XINT_iexpr_oparen}}%
4639 }%
4640 \catcode`- 12

```

### 27.31.7. \xintNewExpr, \xintNewIExpr, \xintNewFloatExpr, \xintNewIIExpr

1.2c modifications to accomodate \XINT\_expr\_deffunc\_newexpr etc..

1.2f adds token \XINT\_newexpr\_clean to be able to have a different \XINT\_newfunc\_clean.

As \XINT\_NewExpr always execute \XINT\_expr\_redefineprints since 1.3e whether with \xintNewExpr or \XINT\_NewFunc, it has been moved from argument to hardcoded in replacement text.

NO MORE \XINT\_expr\_redefineprints at 1.4 ! This allows better support for \xinteval, \xinttheexpr as sub-entities inside an \xintNewExpr. And the «cleaning» will remove the new \XINTfstop, to maintain backwards compatibility with former behaviour that created macros expand to explicit digits and not an encapsulated result.

(obsolete:) The #2#3 in clean stands for \noexpand\XINTfstop.

**Modified at 1.4n (2025/09/05).** The #2#3 in previous paragraph is obsolete, the pattern used for \XINT\_newexpr\_clean was formerly #1>#2#3 (not optimal, but legacy), but is now simply #1\XINTfstop. This is simpler and allows compatibility with LuaMetaTeX whose \meaning output differs from the one of other engines.

```

4641 \def\xintNewExpr {\XINT_NewExpr\xint_firstofone\xintexpr \XINT_newexpr_clean}%
4642 \def\xintNewFloatExpr{\XINT_NewExpr\xint_firstofone\xintfloatexpr\XINT_newexpr_clean}%
4643 \def\xintNewIExpr {\XINT_NewExpr\xint_firstofone\xintiexpr \XINT_newexpr_clean}%
4644 \def\xintNewIIExpr {\XINT_NewExpr\xint_firstofone\xintiiexpr \XINT_newexpr_clean}%
4645 \def\xintNewBoolExpr {\XINT_NewExpr\xint_firstofone\xintboolexpr \XINT_newexpr_clean}%
4646 \def\XINT_newexpr_clean #1\XINTfstop{\noexpand\expanded\noexpand\xintNEprinthead}%
4647 \def\xintNEprinthead#1.#2{\expanded{\xint_noexpd{#1.}{#2}}}%

```

1.2c for \xintdeffunc, \xintdefiifunc, \xintdeffloatfunc.

At 1.3, NewFunc does not use anymore a comma delimited pattern for the arguments to the macro being defined.

At 1.4 we use \xintthebareeval, whose meaning now does not mean unlock from csname but firstofone to remove a level of braces This is involved in functioning of expr:userfunc and expr:userefunc

```

4648 \def\XINT_NewFunc {\XINT_NewExpr\xint_gobble_i\xintthebareeval\XINT_newfunc_clean}%
4649 \def\XINT_NewFloatFunc{\XINT_NewExpr\xint_gobble_i\xintthebarefloateval\XINT_newfunc_clean}%
4650 \def\XINT_NewIIFunc {\XINT_NewExpr\xint_gobble_i\xintthebareiieval\XINT_newfunc_clean}%
4651 \def\XINT_newfunc_clean #1>{}%
4652 \ifdefined\notexpanded\let\XINT_newfunc_clean\xint_gobble_vi\fi%

```

1.2c adds optional logging. For this needed to pass to \_NewExpr\_a the macro name as parameter.

Up to and including 1.2c the definition was global. Starting with 1.2d it is done locally.

The \xintexprSafeCatcodes inserted here by \xintNewExpr is not paired with an \xintexprRestoreCatcodes, but this happens within a scope limiting group so does not matter. At 1.3c, \XINT\_NewFunc et al. do not even execute the \xintexprSafeCatcodes, as it gets already done by \xintdeffunc prior to arriving here.

```

4653 \def\XINT_NewExpr #1#2#3#4#5[#6]%
4654 {%
4655 \begingroup
4656 \ifcase #6\relax
4657 \toks0 {\endgroup\XINT_global\def#4}%
4658 \or \toks0 {\endgroup\XINT_global\def#4#1}%

```

```

4659 \or \toks0 {\endgroup\XINT_global\def#4##1##2}%
4660 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3}%
4661 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4}%
4662 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5}%
4663 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6}%
4664 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7}%
4665 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7##8}%
4666 \or \toks0 {\endgroup\XINT_global\def#4##1##2##3##4##5##6##7##8##9}%
4667 \fi
4668 #1\xintexprSafeCatcodes
4669 \XINT_expr_redefinemacros
4670 \XINT_NewExpr_a #1#2#3#4%
4671 }%

```

1.2d's `\xintNewExpr` makes a local definition. In earlier releases, the definition was global. `\the\toks0` inserts the `\endgroup`, but this will happen after `\XINT_tmpa` has already been expanded...

The %1 is `\xint_firstofone` for `\xintNewExpr`, `\xint_gobble_i` for `\xintdeffunc`.

Attention that at 1.4, there might be entire sub-xintexpressions embedded in detokenized form. They are re-tokenized and the main thing is that the parser should not mis-interpret catcode 11 characters as starting variable names. As some macros use `:` in their names, the retokenization must be done with `:` having catcode 11. To not break embedded non-evaluated sub-expressions, the `\XINT_expr_getop` was extended to intercept the `:` (alternative would have been to never inject any macro with `:` in its name... too late now). On the other hand the `!` is not used in the macro names potentially kept as is non expanded by the `\xintNewExpr/\xintdeffunc` process; it can thus be retokenized with catcode 12. But the «hooks» of `seq()`, `iter()`, etc... if deciding they can't evaluate immediately will inject a full sub-expression (possibly arbitrarily complicated) and append to it for its delayed expansion a catcode 11 `!` character (as well as possibly catcode 3 `~` and `?` and catcode 11 caret `^` and even catcode 7 `&`). The macros `\XINT_expr_tilde` etc... below serve for this injection (there are *two* successive `\scantokens` using different catcode regimes and these macros remain detokenized during the first pass!) and as consequence the final meaning may have characters such as `!` or `&` present with both standard and special catcodes depending on where they are located. It may thus not be possible to (easily) retokenize the meaning as printed in the log file if `\xintverbosetrue` was issued.

If a defined function is used in another expression it would thus break things if its meaning was included pre-expanded ; a mechanism exists which keeps only the name of the macro associated to the function (this name may contain digits by the way), when the macro can not be immediately fully expanded. Thus its meaning (with its possibly funny catcodes) is not exposed. And this gives opportunity to pre-expand its arguments before actually expanding the macro.

There is a problem with xetex -8bit which will convert `^^@` possibly present in the meaning written to log if `\xintverbosetrue` into a null byte. It surprises at first, but perhaps because I am too much used to pdfTeX and LaTeX converting the zero byte perhaps produced meaning internally to `^^@` notation at the "output" stage.

I observed that ConTeXt will spit out such null byte "as is", as does xetex -8bit. This is a bit annoying because contrarily to xetex no option is needed, and such output lets some software consider the output file is a binary one (for example git diff).

```

4672 \catcode`~ 3 \catcode`? 3
4673 \def\XINT_expr_tilde{~}\def\XINT_expr_qmark{?}% catcode 3
4674 \def\XINT_expr_caret{^}\def\XINT_expr_exclam{!}% catcode 11
4675 \def\XINT_expr_tab{&}% catcode 7
4676 \def\XINT_expr_null{&&@}%

```

**Added at 1.4n (2025/09/05).** Add `\xintmeaning` for matters of package test suite. At user level, a priori simply expands to `\meaning`. It gets redefined during execution of the test suite to keep

logged expectations the same also with LMTX engine.

```
4677 \ifdefined\xintmeaning\else\def\xintmeaning{\meaning}\fi
```

**Added at 1.4n (2025/09/05).** Add `\XINT_expr_set_tilde` to address LuaMetaTeX specifics. We won't need to restore because we will be in a group and `\toks0` will bring the `\endgroup`.

When I observed that the whole `\xintdeffunc` thing was broken with ConTeXt due to what appeared to be some weirdness during expansion, I realized much to my surprise that the active tilde was not expanding inside `\edef`! I got lucky I could quickly find a then recent discussion precisely about this on [tex.sx](#), from which I picked up the `\amcode` workaround. Such a trick is quite hidden in ConTeXt documentation, from which it is a long shot to deduce it behaves as it actually does.

Replacing about 130 occurrences of `~` in this file which are a core part of the mysterious dealings underpinning `\xintdeffunc` is not really an option. I don't have that many available ascii characters and I do not want to go into mass replacements and have to update unit tests sometimes checking internals. What a relief there is the `\amcode` way! (and seemingly no other way...).

```
4678 \catcode`~ 13 \catcode`$ 11 %$
4679 \def\xINT_NewExpr_set_tilde{\def~{\$noexpand$}}%
4680 \ifdefined\contextversion
4681   \ifdefined\amcode
4682     \def\xINT_NewExpr_set_tilde{\amcode`~\0\def~{\$noexpand$}}%
4683   \else
4684     \xintMessage{xintexpr}{Error}{This ConTeXt is incompatible.}%
4685     \errhelp{xintexpr requires ConTeXt-LMTX to have its \string\amcode.}%
4686     \errmessage{The \noexpand\amcode primitive does not exist.}%
4687   \fi
4688 \fi
4689 \catcode`@ 14 \catcode`\% 6 \catcode`# 12
4690 \def\xINT_NewExpr_a %1%2%3%4%5@
4691 {@
4692   \def\xINT_tmpa %1%2%3%4%5%6%7%8%9{%5}@
4693   \xINT_NewExpr_set_tilde
4694   \catcode`: 11 \catcode`_ 11 \catcode`@ 11
4695   \catcode`# 12 \catcode`~ 13 \escapechar 126
4696   \endlinechar -1 \everyeof {\noexpand }@
4697   \edef\xINT_tmpb
4698   {\scantokens\expandafter{\romannumeral`&&\expandafter
4699   %2\xINT_tmpa{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{#9}\relax}@
4700 }@
4701 \escapechar 92 \catcode`# 6 \catcode`$ 0 @ $
4702 \edef\xINT_tmpa %1%2%3%4%5%6%7%8%9@
4703 {\scantokens\expandafter{\expandafter%3\meaning\xINT_tmpb}}@
4704 \the\toks0\expandafter
4705 {\xINT_tmpa{%1}{%2}{%3}{%4}{%5}{%6}{%7}{%8}{%9}}@
4706 %1{\ifxintverbose
4707   \xintMessage{xintexpr}{Info}@
4708   {\string%4space now with @
4709   \ifxintglobaldefs global \fi meaning \xintmeaning%4}@
4710   \fi}@
4711 }@
4712 \catcode`\% 14
4713 \xINTsetcatcodes % clean up to avoid surprises if something changes
```

**27.31.8. `\xintexprSafeCatcodes`, `\xintexprRestoreCatcodes`**

**Modified at 1.3c (2018/06/17).** Added `\ifxintexprsafecatcodes` to allow nesting

**Modified at 1.4k (2022/05/18).** The "allow nesting" from the 2018 comment was strange, because the behaviour, as correctly documented in user manual, was that in case of a series of `\xintexprSafeCatcodes`, the `\xintexprRestoreCatcodes` would set catcodes to what they were before the \*first\* sanitization. But as `\xintdefvar` and `\xintdeffunc` used such a pair this meant that they would incomprehensibly for user reset catcodes to what they were before a possible user `\xintexprSafeCatcodes` located before... very lame situation. Anyway. I finally fix at 1.4k that by removing the silly `\ifxintexprsafecatcodes` thing and replace it by some stack-like method, avoiding extra macros thanks to the help of `\unexpanded`.

**Modified at 1.4m (2022/06/10).** Use `\protected` rather than `\unexpanded` mechanism, for lisibility.

```

4714 \protected\def\xintexprRestoreCatcodes{%
4715 \def\xintexprSafeCatcodes
4716 {%
4717   \protected\edef\xintexprRestoreCatcodes{%
4718     \endlinechar=\the\endlinechar
4719     \catcode59=\the\catcode59 % ;
4720     \catcode34=\the\catcode34 % "
4721     \catcode63=\the\catcode63 % ?
4722     \catcode124=\the\catcode124 % |
4723     \catcode38=\the\catcode38 % &
4724     \catcode33=\the\catcode33 % !
4725     \catcode93=\the\catcode93 % ]
4726     \catcode91=\the\catcode91 % [
4727     \catcode94=\the\catcode94 % ^
4728     \catcode95=\the\catcode95 % _
4729     \catcode47=\the\catcode47 % /
4730     \catcode41=\the\catcode41 % )
4731     \catcode40=\the\catcode40 % (
4732     \catcode42=\the\catcode42 % *
4733     \catcode43=\the\catcode43 % +
4734     \catcode62=\the\catcode62 % >
4735     \catcode60=\the\catcode60 % <
4736     \catcode58=\the\catcode58 % :
4737     \catcode46=\the\catcode46 % .
4738     \catcode45=\the\catcode45 % -
4739     \catcode44=\the\catcode44 % ,
4740     \catcode61=\the\catcode61 % =
4741     \catcode96=\the\catcode96 % `
4742     \catcode32=\the\catcode32\relax % space
4743   \protected\odef\xintexprRestoreCatcodes{\xintexprRestoreCatcodes}%
4744 }%
4745   \endlinechar=13 %
4746   \catcode59=12 % ;
4747   \catcode34=12 % "
4748   \catcode63=12 % ?
4749   \catcode124=12 % |
4750   \catcode38=4 % &
4751   \catcode33=12 % !
4752   \catcode93=12 % ]
4753   \catcode91=12 % [

```



```

4754      \catcode94=7   % ^
4755      \catcode95=8   % _
4756      \catcode47=12  % /
4757      \catcode41=12  % )
4758      \catcode40=12  % (
4759      \catcode42=12  % *
4760      \catcode43=12  % +
4761      \catcode62=12  % >
4762      \catcode60=12  % <
4763      \catcode58=12  % :
4764      \catcode46=12  % .
4765      \catcode45=12  % -
4766      \catcode44=12  % ,
4767      \catcode61=12  % =
4768      \catcode96=12  % `
4769      \catcode32=10   % space
4770 }%
4771 \let\XINT_tmpa\undefined \let\XINT_tmpb\undefined \let\XINT_tmpc\undefined
4772 \let\XINT_tmpd\undefined \let\XINT_tmpe\undefined

```

## 27.32. Matters related to loading log and trig libraries

1.41 makes a user level `\usepackage{xintlog}` (not the one done via `\usepackage{xintexpr}` attempt to do the right thing (in place of aborting). We have to work-around the fact that LaTeX will ignore a `\usepackage` here. Simpler for non-LaTeX.

In all cases, the input of `xintlog.sty` and `xinttrig.sty` is done with `xintexpr` catcodes in place. And `xintlog` will sanitize catcodes at time of loading poormanlog. Attention also to not mix-up things at time of restoring catcodes. This is reason why `xintlog.sty` and `xinttrig.sty` have their own endinput wrappers. And that the rescue attempt of loading `xintexpr` (which will load `xintlog`) from `xintlog` itself is done carefully.

```

4773 \ifdefined\RequirePackage
4774 \ifcsname ver@xinttrig.sty\endcsname

```

We end up here since 1.41 with  $\TeX$  if the user has issued `\usepackage{xinttrig}` or `\RequirePackage{xinttrig}` with no prior loading of `xintexpr`. In such (not officially supported) case, the loading of `xintexpr` was launched from this first instance of `xinttrig`. This first `xinttrig` will abort itself, once this input concludes. But before that a second instance of `xinttrig` is `\input` and will do all its macro definitions. We can not do `\RequirePackage{xinttrig}` or `\usepackage{xinttrig}` as it has occurred already under the user responsibility, so we use `\@@input`.

```

4775     \@@input xinttrig.sty\relax
4776 \else

```

Here this is either the normal case with  $\TeX$  (or other formats providing `\RequirePackage`) and `xintexpr` requested by user directly, or some more exotic possibility such as  $\epsilon\text{-}\TeX$  with the `miniltx` loaded and then `\input xintexpr.sty\relax` was done. As `\RequirePackage` appears to be defined we use it.

```

4777     \RequirePackage{xinttrig}%
4778 \fi

```

Same situation with `xintlog`.

```

4779 \ifcsname ver@xintlog.sty\endcsname
4780     \@@input xintlog.sty\relax
4781 \else
4782     \RequirePackage{xintlog}%

```



```
4783 \fi
4784 \else
```

Here we are not with ~~TeX~~ and not with `miniltx` either. Let's just use `\input`. Perhaps there was an `\input xinttrig.sty` earlier which triggered `\input xintexpr.sty` after a warning to the user. The second `\input xinttrig.sty` issued here will execute the macro definitions, and the former one will abort its own input after that.

```
4785 \input xinttrig.sty
4786 \input xintlog.sty
4787 \fi
4788 \XINTrestorecatcodesendinginput%
```

## 28. Package [xinttrig](#) implementation

### Contents

28.1	Catcodes, $\varepsilon$ -TeX and reload detection . . . . .	690
28.2	Library identification . . . . .	691
28.3	Ensure used letters are dummy letters . . . . .	692
28.4	<code>\xintreloadxinttrig</code> . . . . .	692
28.5	Auxiliary variables . . . . .	692
28.5.1	<code>@twoPi</code> , <code>@threePiover2</code> , <code>@Pi</code> , <code>@Piover2</code> . . . . .	692
28.5.2	<code>@oneDegree</code> , <code>@oneRadian</code> . . . . .	692
28.6	Hack <code>\xintdeffloatfunc</code> for inserting usage of guard digits . . . . .	693
28.7	The sine and cosine series . . . . .	694
28.7.1	Support macros for the sine and cosine series . . . . .	694
28.7.2	The poor man approximate but speedier approach for Digits at most 8 . . . . .	697
28.7.3	Declarations of the <code>@sin_aux()</code> and <code>@cos_aux()</code> functions . . . . .	698
28.7.4	<code>@sin_series()</code> , <code>@cos_series()</code> . . . . .	698
28.8	Range reduction for sine and cosine using degrees . . . . .	698
28.8.1	Low level modulo 360 helper macro <code>\XINT_mod_ccclx_i</code> . . . . .	698
28.8.2	<code>@sind_rr()</code> function and its support macro <code>\xintSind</code> . . . . .	699
28.8.3	<code>@cosd_rr()</code> function and its support macro <code>\xintCosd</code> . . . . .	701
28.9	<code>@sind()</code> , <code>@cosd()</code> . . . . .	703
28.10	<code>@sin()</code> , <code>@cos()</code> . . . . .	703
28.11	<code>@sinc()</code> . . . . .	703
28.12	<code>@tan()</code> , <code>@tand()</code> , <code>@cot()</code> , <code>@cotd()</code> . . . . .	704
28.13	<code>@sec()</code> , <code>@secd()</code> , <code>@csc()</code> , <code>@cscd()</code> . . . . .	704
28.14	Core routine for inverse trigonometry . . . . .	704
28.15	<code>@asin()</code> , <code>@asind()</code> . . . . .	708
28.16	<code>@acos()</code> , <code>@acosd()</code> . . . . .	708
28.17	<code>@atan()</code> , <code>@atand()</code> . . . . .	708
28.18	<code>@Arg()</code> , <code>@atan2()</code> , <code>@Argd()</code> , <code>@atan2d()</code> , <code>@pArg()</code> , <code>@pArgd()</code> . . . . .	709
28.19	Restore <code>\xintdeffloatfunc</code> to its normal state, with no extra digits . . . . .	710
28.20	Let the functions be known to the <code>\xintexpr</code> parser . . . . .	710
28.21	Synonyms: <code>@tg()</code> , <code>@cotg()</code> . . . . .	711
28.22	Final clean-up . . . . .	711

A preliminary implementation was done only late in the development of [xintexpr](#), as an example of the high level user interface, in January 2019. In March and April 2019 I improved the algorithm for the inverse trigonometrical functions and included the whole as a new [\xintexpr](#) module. But, as the high level interface provided no way to have intermediate steps executed with guard digits, the whole scheme could only target say P-2 digits where P is the prevailing precision, and only with a moderate requirement on what it means to have P-2 digits about correct.

Finally in April 2021, after having at long last added exponential and logarithm up to 62 digits and at a rather strong precision requirement (something like, say with inputs in normal ranges: targeting at most 0.505ulp distance to exact result), I revisited the code here.

We keep most of the high level usage of `\xintdeffloatfunc`, but hack into its process in order to let it map the 4 operations and some functions such as square-root to macros using 4 extra digits. This hack is enough to support the used syntax here, but is not usable generally. All functions and their auxiliaries defined during the time the hack applies are named with `@` as first letter.

Later the public functions, without the `@`, are defined as wrappers of the `@`-named ones, which float-round to P digits on output.

Apart from that the sine and cosine series were implemented at macro level, bypassing the `\xintdeffloatfunc` interface. This is done mainly for handling Digits at high value (24 or more) as it

then becomes beneficial to float-round the variable to less and less digits, the deeper one goes into the series.

And regarding the arcsine I modified a bit my original idea in order to execute the first step in a single `\numexpr`. It turns out that that for 16 digits the algorithm then ``only'' needs one sine and one cosine evaluation (and a square-root), and there is no need for an arcsine series auxiliary then. I am aware this is by far not the ``best'' approach but the problem is that I am a bit enamored into the idea of the algorithm even though it is at least twice as costly than a sine evaluation! Actually, for many digits, it turns out the arcsine is less costly than two random sine evaluations, probably because the latter have the overhead of range reduction.

Speaking of this, the range reduction is rather naive and not extremely ambitious. I wrote it initially having only `sind()` and `cosd()` in mind, and in 2019 reduced degrees to radians in the most naive way possible. I have only slightly improved this for this 1.4e 2021 release, the announced precision for inputs less than say `1e6`, but at `1e8` and higher, one will start feeling the gradual loss of precision compared to the task of computing the exact mathematical result correctly rounded. Also, I do not worry here about what happens when the input is very near a big multiple of  $\pi$ , and one computes a sine for example. Maybe I will improve in future this aspect but I decided I was seriously running out of steam for the 1.4e release.

As commented in [xintlog](#) regarding exponential and logarithms, even though we have instilled here some dose of lower level coding, the whole suffers from [xintfrac](#) not yet having made floating point numbers a native type. Thus inefficiencies accumulate...

At 8 digits, the gain was only about 40% compared to 16 digits. So at the last minute, on the day I was going to do the release I decided to implement a poorman way for sine and cosine, for "speed". I transferred the idea from the arcsine `numexpr` to sine and cosine. Indeed there is an interesting speed again of about 4X compared to applying the same approach as for higher values of Digits. Correct rounding during random testing is still obtained reasonably often (at any rate more than 95% of cases near 45 degrees and always faithful rounding), although at less than the 99% reached for the main branch handling Digits up to 62. But the precision is more than enough for usage in plots for example. I am keeping the guard digits, as removing them would add a further speed gain of about 20% to 40% but the precision then would drop dramatically, and this is not acceptable at the time of our 2021 standards (not a period of enlightenment generally speaking, though).

## 28.1. Catcodes, $\varepsilon$ -TeX and reload detection

**Modified at 1.41 (2022/05/29).** Silly paranoid modification of `\z` in case `{` and `}` do not have their normal catcodes when `xinttrig.sty` is reloaded (initial loading via `xintexpr.sty` does not need this), to define `\XINTtrigendinput` there and not after the `\endgroup` from `\z` has already restored possibly bad catcodes.

1.41 handles much better the situation with `\usepackage{xinttrig}` without previous loading of `xintexpr` (or same with `\input` and `etex`). cf comments in `xintlog.sty`.

```

1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode35=6 % #
8 \catcode44=12 % ,
9 \catcode46=12 % .
10 \catcode58=12 % :
11 \catcode94=7 % ^
12 \def\empty{} \def\space{ } \newlinechar10

```

```

13 \def\z{\endgroup}%
14 \expandafter\let\expandafter\x\csname ver@xinttrig.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xintexpr.sty\endcsname
16 \expandafter
17 \ifx\csname PackageWarningNoLine\endcsname\relax
18 \def\y#1#2{\immediate\write128{^^JPackage #1 Warning:^^J%
19 \space\space\space\space#2.^^J}}%
20 \else
21 \def\y#1#2{\PackageWarningNoLine{#1}{#2}}%
22 \fi
23 \expandafter
24 \ifx\csname numexpr\endcsname\relax
25 \y{xinttrig}{\numexpr not available, aborting input}%
26 \def\z{\endgroup\endinput}%
27 \else
28 \ifx\w\relax % xintexpr.sty not yet loaded.
29 \edef\MsgBrk{^^J\space\space\space\space}%
30 \y{xinttrig}%
31 {\ifx\x\empty
32 xinttrig should not be loaded directly\MessageBreak
33 The correct way is \string\usepackage{xintexpr}.\MessageBreak
34 Will try that now%
35 \else
36 First loading of xinttrig.sty should be via
37 \string\input\space xintexpr.sty\relax\MsgBrk
38 Will try that now%
39 \fi
40 }%
41 \ifx\x\empty
42 \def\z{\endgroup\RequirePackage{xintexpr}\endinput}%
43 \else
44 \def\z{\endgroup\input xintexpr.sty\relax\endinput}%
45 \fi
46 \else
47 \def\z{\endgroup\edef\XINTtrigendinput{\XINTrestorecatcodes\noexpand\endinput}}%
48 \fi
49 \fi
50 \z%
51 \XINTsetcatcodes%
52 \catcode`? 12

```

## 28.2. Library identification

If the file has already been loaded, let's skip the `\ProvidesPackage`. Else let's do it and set a flag to indicate loading happened at least once already.

**Modified at 1.41 (2022/05/29).** Message also to Terminal not only log file.

```

53 \ifcsname xintlibver@trig\endcsname
54 \expandafter\xint_firstoftwo
55 \else
56 \expandafter\xint_secondoftwo
57 \fi
58 {\immediate\write128{Reloading xinttrig library using Digits=\xinttheDigits.}}%

```

```

59 {\expandafter\gdef\csname xintlibver@trig\endcsname{2025/09/06 v1.4o}%
60 \XINT_providespackage
61 \ProvidesPackage{xinttrig}%
62 [2025/09/06 v1.4o Trigonometrical functions for xintexpr (JFB)]%
63 }%

```

### 28.3. Ensure used letters are dummy letters

```

64 \xintFor* #1 in {iDTVtuwxyzX}\do{\xintensuredummy{#1}}%

```

### 28.4. \xintreloadxinttrig

Much simplified at 1.4e, from a modified catcode regime management.

```

65 \def\xintreloadxinttrig{\input xinttrig.sty }%

```

### 28.5. Auxiliary variables

The variables with private names have extra digits. Whether private or public, the variables can all be redefined without impacting the defined functions, whose meanings will contain already the variable values.

Formerly variables holding the  $1/n!$  were defined, but this got removed at 1.4e.

#### 28.5.1. @twoPi, @threePiover2, @Pi, @Piover2

At 1.4e we need more digits, also [\xintdeffloatvar](#) changed and always rounds to P=Digits precision so we use another path to store values with extra digits.

```

66 \xintdefvar @twoPi :=
67   float(
68 6.2831853071795864769252867665590057683943387987502116419498891846156328125724180
69   ,\XINTdigitsormax+4);%
70 \xintdefvar @threePiover2 :=
71   float(
72 4.7123889803846898576939650749192543262957540990626587314624168884617246094293135
73   ,\XINTdigitsormax+4);%
74 \xintdefvar @Pi :=
75   float(
76 3.1415926535897932384626433832795028841971693993751058209749445923078164062862090
77   ,\XINTdigitsormax+4);%
78 \xintdefvar @Piover2 :=
79   float(
80 1.5707963267948966192313216916397514420985846996875529104874722961539082031431045
81   ,\XINTdigitsormax+4);%

```

#### 28.5.2. @oneDegree, @oneRadian

Those are needed for range reduction, particularly @oneRadian. We define it with 12 extra digits. But the whole process of range reduction in radians is very naive one.

```

82 \xintdefvar @oneDegree :=
83   float(
84 0.017453292519943295769236907684886127134428718885417254560971914401710091146034494
85   ,\XINTdigitsormax+4);%
86 \xintdefvar @oneRadian :=
87   float(

```

```

88 57.295779513082320876798154814105170332405472466564321549160243861202847148321553
89 ,\XINTdigitsormax+12);%

```

## 28.6. Hack `\xintdeffloatfunc` for inserting usage of guard digits

1.4e. This is not a general approach, but it sufficient for the limited use case done here of `\xintdeffloatfunc`. What it does is to let `\xintdeffloatfunc` hardcode usage of macros which will execute computations with an elevated number of digits. But for example if  $5/3$  is encountered in a float expression it will remain unevaluated so one would have to use alternate input syntax for efficiency (`\xintexpr float(5/3,\xinttheDigits+4)\relax` as a subexpression, for example).

```

90 \catcode`~ 12
91 \def\XINT_tmpa#1#2#3.#4.%
92 {%
93   \let #1#2%
94   \def #2##1##2##3##4%
95     {##2##3{{~expanded{~xint_noxpd{#4[#3]}~expandafter}~expanded{##1##4}}}}%
96 }%
97 \expandafter\XINT_tmpa
98   \csname XINT_flexpr_exec_+~\expandafter\endcsname
99   \csname XINT_flexpr_exec_+~\expandafter\endcsname
100   \the\numexpr\XINTdigitsormax+4.~XINTinFloatAdd_wopt.%
101 \expandafter\XINT_tmpa
102   \csname XINT_flexpr_exec_-~\expandafter\endcsname
103   \csname XINT_flexpr_exec_-~\expandafter\endcsname
104   \the\numexpr\XINTdigitsormax+4.~XINTinFloatSub_wopt.%
105 \expandafter\XINT_tmpa
106   \csname XINT_flexpr_exec_*~\expandafter\endcsname
107   \csname XINT_flexpr_exec_*~\expandafter\endcsname
108   \the\numexpr\XINTdigitsormax+4.~XINTinFloatMul_wopt.%
109 \expandafter\XINT_tmpa
110   \csname XINT_flexpr_exec_/~\expandafter\endcsname
111   \csname XINT_flexpr_exec_/~\expandafter\endcsname
112   \the\numexpr\XINTdigitsormax+4.~XINTinFloatDiv_wopt.%
113 \def\XINT_tmpa#1#2#3.#4.%
114 {%
115   \let #1#2%
116   \def #2##1##2##3{{##1##2{{~expanded{~xint_noxpd{#4[#3]}~expandafter}##3}}}}%
117 }%
118 \expandafter\XINT_tmpa
119   \csname XINT_flexpr_sqrfunc~\expandafter\endcsname
120   \csname XINT_flexpr_func_sqr~\expandafter\endcsname
121   \the\numexpr\XINTdigitsormax+4.~XINTinFloatSqr_wopt.%
122 \expandafter\XINT_tmpa
123   \csname XINT_flexpr_sqrtfunc~\expandafter\endcsname
124   \csname XINT_flexpr_func_sqrt~\expandafter\endcsname
125   \the\numexpr\XINTdigitsormax+4.~XINTinFloatSqrt.%
126 \expandafter\XINT_tmpa
127   \csname XINT_flexpr_invfunc~\expandafter\endcsname
128   \csname XINT_flexpr_func_inv~\expandafter\endcsname
129   \the\numexpr\XINTdigitsormax+4.~XINTinFloatInv_wopt.%
130 \catcode`~ 3

```

## 28.7. The sine and cosine series

Old pending question: should I rather use successive divisions by  $(2n+1)(2n)$ , or rather multiplication by their precomputed inverses, in a modified Horner scheme ? The `\ifnum` tests are executed at time of definition.

Update at last minute: this is actually exactly what I do if Digits is at most 8.

Small values of the variable are very badly handled here because a much shorter truncation of the series should be used.

At 1.4e the original `\xintdeffloatfunc` was converted into macros, whose principle can be seen also at work in `xintlog.sty`. We prepare the input variables with shorter and shorter mantissas for usage deep in the series.

This divided by about 3 the execution cost of the series for P about 60.

Originally, the thresholds were computed a priori with 0.79 as upper bound of the variable, but then for 1.4e I developped enough test files to try to adjust heuristically with a target of say 99,5% of correct rounding, and always at most lulp error. The numerical analysis is not easy due to the complications of the implementation...

Also, random testing never explores the weak spots...

The 0.79 (a bit more than  $\pi/4$ ) upper bound induces a costly check of variable on input, if Digits is big. Much faster would be to check if input is less than 10 degrees or 1 radian as done in `xfp`. But using enough coefficients for allowing up to 1 radian, which is without pain for Digits=16 starts being annoying for higher values such as Digits=48.

But the main reason I don't do it now is that I spend too much time fine-tuning the table of thresholds... maybe in next release.

### 28.7.1. Support macros for the sine and cosine series

Computing the  $1/n!$  from  $n!$  then inverting would require costly divisions and significantly increase the loading time.

So a method is employed to simply divide by  $2k(2k-1)$  or  $(2k+1)(2k)$  step by step, with what we hope are enough 8 security digits, and reducing the sizes of the mantissas at each step.

This whole section is conditional on Digits being at least nine.

```

131 \ifnum\XINTdigits>8
132 \edef\XINT_tmpG % 1/3!
133 {1\xintReplicate{\XINTdigitsormax+2}{6}7[\the\numexpr-\XINTdigitsormax-4]}%
134 \edef\XINT_tmpH % 1/5!
135 {8\xintReplicate{\XINTdigitsormax+1}{3}3[\the\numexpr-\XINTdigitsormax-4]}%
136 \edef\XINT_tmpe % 1/5!
137 {8\xintReplicate{\XINTdigitsormax+9}{3}3[\the\numexpr-\XINTdigitsormax-12]}%
138 \def\XINT_tmpe#1.#2.#3.#4.#5#6#7%
139 {%
140 \def#5##1\xint:
141 {%
142 \expandafter#6\romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
143 }%
144 \def#6##1\xint:
145 {%
146 \expandafter#7\romannumeral0\xintsub{#4}{\XINTinFloat[#2]{\xintMul{#3}{##1}}}\xint:
147 }%
148 \def#7##1\xint:##2\xint:
149 {%
150 \xintSub{1/1[0]}{\XINTinFloat[#1]{\xintMul{##1}{##2}}}%
151 }%
152 }%
```

```

153 \expandafter\XINT_tmpe
154 \the\numexpr\XINTdigitsormax+4\expandafter.%
155 \the\numexpr\XINTdigitsormax+2\expandafter.\expanded{%
156 \XINT_tmpeH.% 1/5!
157 \XINT_tmpeG.% 1/3!
158 \expandafter}%
159 \csname XINT_SinAux_series_a_iii\expandafter\endcsname
160 \csname XINT_SinAux_series_b\expandafter\endcsname
161 \csname XINT_SinAux_series_c_i\endcsname
162 \def\XINT_tmpe #1 #2 #3 #4 #5 #6 #7 #8 %
163 {%
164 \def\XINT_tmpeb ##1##2##3##4##5%
165 {%
166 \def\XINT_tmpec####1.####2.####3.####4.####5.%
167 {%
168 \def##1#####1\xint:
169 {%
170 \expandafter##2%
171 \romannumeral0\XINTinfloatS[####1]{#####1}\xint:#####1\xint:
172 }%
173 \def##2#####1\xint:
174 {%
175 \expandafter##3%
176 \romannumeral0\XINTinfloatS[####2]{#####1}\xint:#####1\xint:
177 }%
178 \def##3#####1\xint:
179 {%
180 \expandafter##4%
181 \romannumeral0\xintsub{####4}{\XINTinfloat[####2]{\xintMul{####3}{#####1}}}\xint:
182 }%
183 \def##4#####1\xint:#####2\xint:
184 {%
185 \expandafter##5%
186 \romannumeral0\xintsub{####5}%
187 {\XINTinfloat[####1]{\xintMul{#####1}{#####2}}}\xint:
188 }%
189 }%
190 }%
191 \expandafter\XINT_tmpeb
192 \csname XINT_#8Aux_series_a\romannumeral\numexpr#1-1\expandafter\endcsname
193 \csname XINT_#8Aux_series_a\romannumeral\numexpr#1\expandafter\endcsname
194 \csname XINT_#8Aux_series_b\expandafter\endcsname
195 \csname XINT_#8Aux_series_c\romannumeral\numexpr#1-2\expandafter\endcsname
196 \csname XINT_#8Aux_series_c\romannumeral\numexpr#1-3\endcsname
197 \edef\XINT_tmpe
198 {\XINTinfloat[\XINTdigitsormax-#2+8]{\xintDiv{\XINT_tmpe}{\the\numexpr#5*(#5-1)\relax}}}%
199 \let\XINT_tmpeF\XINT_tmpeG
200 \let\XINT_tmpeH\XINT_tmpeH
201 \edef\XINT_tmpeH{\XINTinfloat[\XINTdigitsormax-#2]{\XINT_tmpe}}%
202 \expandafter\XINT_tmpec
203 \the\numexpr\XINTdigitsormax-#3\expandafter.%
204 \the\numexpr\XINTdigitsormax-#2\expandafter.\expanded{%

```



```

205 \XINT_tmpH.%
206 \XINT_tmpG.%
207 \XINT_tmpF.%
208 }%
209 }%
210 \XINT_tmpa 4 -1 -2 -4 7 5 3 Sin %
211 \ifnum\XINTdigits>3 \XINT_tmpa 5 1 -1 -2 9 7 5 Sin \fi
212 \ifnum\XINTdigits>5 \XINT_tmpa 6 3 1 -1 11 9 7 Sin \fi
213 \ifnum\XINTdigits>8 \XINT_tmpa 7 6 3 1 13 11 9 Sin \fi
214 \ifnum\XINTdigits>11 \XINT_tmpa 8 9 6 3 15 13 11 Sin \fi
215 \ifnum\XINTdigits>14 \XINT_tmpa 9 12 9 6 17 15 13 Sin \fi
216 \ifnum\XINTdigits>16 \XINT_tmpa 10 14 12 9 19 17 15 Sin \fi
217 \ifnum\XINTdigits>19 \XINT_tmpa 11 17 14 12 21 19 17 Sin \fi
218 \ifnum\XINTdigits>22 \XINT_tmpa 12 20 17 14 23 21 19 Sin \fi
219 \ifnum\XINTdigits>25 \XINT_tmpa 13 23 20 17 25 23 21 Sin \fi
220 \ifnum\XINTdigits>28 \XINT_tmpa 14 26 23 20 27 25 23 Sin \fi
221 \ifnum\XINTdigits>31 \XINT_tmpa 15 29 26 23 29 27 25 Sin \fi
222 \ifnum\XINTdigits>34 \XINT_tmpa 16 32 29 26 31 29 27 Sin \fi
223 \ifnum\XINTdigits>37 \XINT_tmpa 17 35 32 29 33 31 29 Sin \fi
224 \ifnum\XINTdigits>40 \XINT_tmpa 18 38 35 32 35 33 31 Sin \fi
225 \ifnum\XINTdigits>44 \XINT_tmpa 19 42 38 35 37 35 33 Sin \fi
226 \ifnum\XINTdigits>47 \XINT_tmpa 20 45 42 38 39 37 35 Sin \fi
227 \ifnum\XINTdigits>51 \XINT_tmpa 21 49 45 42 41 39 37 Sin \fi
228 \ifnum\XINTdigits>55 \XINT_tmpa 22 53 49 45 43 41 39 Sin \fi
229 \ifnum\XINTdigits>58 \XINT_tmpa 23 56 53 49 45 43 41 Sin \fi
230 \edef\XINT_tmpd % 1/4!
231 {41\xintReplicate{\XINTdigitsormax+8}{6}7[\the\numexpr-\XINTdigitsormax-12]}%
232 \edef\XINT_tmpH % 1/4!
233 {41\xintReplicate{\XINTdigitsormax}{6}7[\the\numexpr-\XINTdigitsormax-4]}%
234 \def\XINT_tmpG{5[-1]}% 1/2!
235 \expandafter\XINT_tmpe
236 \the\numexpr\XINTdigitsormax+4\expandafter.%
237 \the\numexpr\XINTdigitsormax+3\expandafter.\expanded{%
238 \XINT_tmpH.%
239 \XINT_tmpG.%
240 \expandafter}%
241 \csname XINT_CosAux_series_a_iii\expandafter\endcsname
242 \csname XINT_CosAux_series_b\expandafter\endcsname
243 \csname XINT_CosAux_series_c_i\endcsname
244 \XINT_tmpa 4 -2 -3 -4 6 4 2 Cos %
245 \ifnum\XINTdigits>2 \XINT_tmpa 5 0 -2 -3 8 6 4 Cos \fi
246 \ifnum\XINTdigits>4 \XINT_tmpa 6 2 0 -2 10 8 6 Cos \fi
247 \ifnum\XINTdigits>7 \XINT_tmpa 7 5 2 0 12 10 8 Cos \fi
248 \ifnum\XINTdigits>9 \XINT_tmpa 8 7 5 2 14 12 10 Cos \fi
249 \ifnum\XINTdigits>12 \XINT_tmpa 9 10 7 5 16 14 12 Cos \fi
250 \ifnum\XINTdigits>15 \XINT_tmpa 10 13 10 7 18 16 14 Cos \fi
251 \ifnum\XINTdigits>18 \XINT_tmpa 11 16 13 10 20 18 16 Cos \fi
252 \ifnum\XINTdigits>20 \XINT_tmpa 12 18 16 13 22 20 18 Cos \fi
253 \ifnum\XINTdigits>24 \XINT_tmpa 13 22 18 16 24 22 20 Cos \fi
254 \ifnum\XINTdigits>27 \XINT_tmpa 14 25 22 18 26 24 22 Cos \fi
255 \ifnum\XINTdigits>30 \XINT_tmpa 15 28 25 22 28 26 24 Cos \fi
256 \ifnum\XINTdigits>33 \XINT_tmpa 16 31 28 25 30 28 26 Cos \fi

```

```

257 \ifnum\XINTdigits>36 \XINT_tmpa 17 34 31 28 32 30 28 Cos \fi
258 \ifnum\XINTdigits>39 \XINT_tmpa 18 37 34 31 34 32 30 Cos \fi
259 \ifnum\XINTdigits>42 \XINT_tmpa 19 40 37 34 36 34 32 Cos \fi
260 \ifnum\XINTdigits>45 \XINT_tmpa 20 43 40 37 38 36 34 Cos \fi
261 \ifnum\XINTdigits>49 \XINT_tmpa 21 47 43 40 40 38 36 Cos \fi
262 \ifnum\XINTdigits>53 \XINT_tmpa 22 51 47 43 42 40 38 Cos \fi
263 \ifnum\XINTdigits>57 \XINT_tmpa 23 55 51 47 44 42 40 Cos \fi
264 \ifnum\XINTdigits>60 \XINT_tmpa 24 58 55 51 46 44 42 Cos \fi
265 \let\XINT_tmpH\xint_undefined\let\XINT_tmpG\xint_undefined\let\XINT_tmpF\xint_undefined
266 \let\XINT_tmpd\xint_undefined\let\XINT_tmpe\xint_undefined
267 \def\XINT_SinAux_series#1%
268 {%
269     \expandafter\XINT_SinAux_series_a_iii
270     \romannumeral0\XINTinfloatS[\XINTdigitsormax+4]{#1}\xint:
271 }%
272 \def\XINT_CosAux_series#1%
273 {%
274     \expandafter\XINT_CosAux_series_a_iii
275     \romannumeral0\XINTinfloatS[\XINTdigitsormax+4]{#1}\xint:
276 }%
277 \fi % end of \XINTdigits>8

```

### 28.7.2. The poor man approximate but speedier approach for Digits at most 8

```

278 \ifnum\XINTdigits<9
279 \def\XINT_SinAux_series#1%
280 {%
281     \the\numexpr\expandafter\XINT_SinAux_b\romannumeral0\xintiround9{#1}.[ -9]%
282 }%
283 \def\XINT_SinAux_b#1.%
284 {%
285     (((((((((((((%(\xint_c_x^ix/-210)
286     -4761905*#1/\xint_c_x^ix+\xint_c_x^ix)/%
287     -156)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
288     -110)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
289     -72)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
290     -42)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
291     -20)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
292     -6)*#1/\xint_c_x^ix+\xint_c_x^ix
293 }%
294 \def\XINT_CosAux_series#1%
295 {%
296     \the\numexpr\expandafter\XINT_CosAux_b\romannumeral0\xintiround9{#1}.[ -9]%
297 }%
298 \def\XINT_CosAux_b#1.%
299 {%
300     (((((((((((((((((%(\xint_c_x^ix/-240)
301     -4166667*#1/\xint_c_x^ix+\xint_c_x^ix)/%
302     -182)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
303     -132)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
304     -90)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
305     -56)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
306     -30)*#1/\xint_c_x^ix+\xint_c_x^ix)/%

```

```

307 -12)*#1/\xint_c_x^ix+\xint_c_x^ix)/%
308 -2)*#1/\xint_c_x^ix+\xint_c_x^ix
309 }%
310 \fi

```

### 28.7.3. Declarations of the @sin\_aux() and @cos\_aux() functions

```

311 \def\xINT_flexpr_func_@sin_aux#1#2#3%
312 {%
313   \expandafter #1\expandafter #2\expandafter{%
314     \romannumeral`&&\XINT:NEhook:f:one:from:one
315     {\romannumeral`&&\XINT_SinAux_series#3}}%
316 }%
317 \def\xINT_flexpr_func_@cos_aux#1#2#3%
318 {%
319   \expandafter #1\expandafter #2\expandafter{%
320     \romannumeral`&&\XINT:NEhook:f:one:from:one
321     {\romannumeral`&&\XINT_CosAux_series#3}}%
322 }%

```

### 28.7.4. @sin\_series(), @cos\_series()

```

323 \xintdeffloatfunc @sin_series(x) := x * @sin_aux(sqr(x));%
324 \xintdeffloatfunc @cos_series(x) := @cos_aux(sqr(x));%

```

## 28.8. Range reduction for sine and cosine using degrees

As commented in the package introduction, Range reduction is a demanding domain and we handle it semi-satisfactorily. The main problem is that in January 2019 I had done only support for degrees, and when I added radians I used the most naive approach. But one can find worse: in 2019 I was surprised to observe important divergences with Maple's results at 16 digits near  $-\pi$ . Turns out that Maple probably adds  $\pi$  in the floating point sense causing catastrophic loss of digits when one is near  $-\pi$ . On the other hand even though the approach here is still naive, it behaves much better.

The @sind\_rr() and @cosd\_rr() sine and cosine "doing range reduction" are coded directly at macro level via `\xintSind` and `\xintCosd` which will dispatch to usage of the sine or cosine series, depending on case.

Old note from 2019: attention that `\xintSind` and `\xintCosd` must be used with a positive argument.

We start with an auxiliary macro to reduce modulo 360 quickly.

### 28.8.1. Low level modulo 360 helper macro \XINT\_mod\_ccclx\_i

input: `\the\numexpr\xINT_mod_ccclx_i` k.N. (delimited by dots)

output: (N times  $10^k$ ) modulo 360. (with a final dot)

Attention that N must be non-negative (I could make it accept negative but the fact that numexpr / is not periodical in numerator adds overhead).

360 divides 9000 hence  $10^k$  is 280 for k at least 3 and the additive group generated by it modulo 360 is the set of multiples of 40.

```

325 \def\xINT_mod_ccclx_i #1.%
326 {%
327   \expandafter\xINT_mod_ccclx_e\the\numexpr
328   \expandafter\xINT_mod_ccclx_j\the\numexpr1\ifcase#1 \or0\or00\else000\fi.%

```

```

329 }%
330 \def\XINT_mod_ccclx_j 1#1.#2.%
331 {%
332   (\XINT_mod_ccclx_ja {++}#2#1\XINT_mod_ccclx_jb 0000000\relax
333 }%
334 \def\XINT_mod_ccclx_ja #1#2#3#4#5#6#7#8#9%
335 {%
336   #9+#8+#7+#6+#5+#4+#3+#2\xint_firstoftwo{+\XINT_mod_ccclx_ja{+#9+#8+#7}}{#1}%
337 }%
338 \def\XINT_mod_ccclx_jb #1\xint_firstoftwo#2#3{#1+0)*280\XINT_mod_ccclx_jc #1#3}%
    Attention that \XINT_cclcx_e wants non negative input because \numexpr division is not periodical
    ...
339 \def\XINT_mod_ccclx_jc +#1+#2+#3#4\relax{+80*(#3+#2+#1)+#3#2#1.}%
340 \def\XINT_mod_ccclx_e#1.{\expandafter\XINT_mod_ccclx_z\the\numexpr(#1+180)/360-1.#1.}%
341 \def\XINT_mod_ccclx_z#1.#2.{#2-360*#1.}%

```

### 28.8.2. @sind\_rr() function and its support macro \xintSind

```

342 \def\XINT_flexpr_func@sind_rr #1#2#3%
343 {%
344   \expandafter #1\expandafter #2\expandafter{%
345   \romannumeral`&&\XINT:NEhook:f:one:from:one{\romannumeral`&&\xintSind#3}}%
346 }%

```

old comment: Must be f-expandable for nesting macros from [\xintNewExpr](#)

This is where the prize of using the same macros for two distinct use cases has serious disadvantages. The reason of Digits+12 is only to support an input which contains a multiplication by @oneRadian with its extended digits.

Then we do a somewhat strange truncation to a fixed point of fractional digits, which is ok in the "Degrees" case, but causes issues of its own in the "Radians" case. Please consider this whole thing as marked for future improvement, when times allows.

ATTENTION [\xintSind](#) ONLY FOR POSITIVE ARGUMENTS

```

347 \def\XINT_tmpa #1.{%
348 \def\xintSind##1%
349 {%
350   \romannumeral`&&\expandafter\xintsind\romannumeral0\XINTinfloatS[#1]{##1}}%
351 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+12.%
352 \def\xintsind #1[#2#3]%
353 {%
354   \xint_UDsignfork
355   #2\XINT_sind
356   -\XINT_sind_int
357   \krof#2#3.#1.%
358 }%
359 \def\XINT_tmpa #1.{%
360 \def\XINT_sind ##1.##2.%
361 {%
362   \expandafter\XINT_sind_a
363   \romannumeral0\xinttrunc{#1}{##2[##1]}%
364 }%
365 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+5.%
366 \def\XINT_sind_a{\expandafter\XINT_sind_i\the\numexpr\XINT_mod_ccclx_i0.}%
367 \def\XINT_sind_int

```

```

368 {%
369   \expandafter\XINT_sind_i\the\numexpr\expandafter\XINT_mod_ccclx_i
370 }%
371 \def\XINT_sind_i #1.%
372 {%
373   \ifcase\numexpr#1/90\relax
374     \expandafter\XINT_sind_A
375   \or\expandafter\XINT_sind_B\the\numexpr-90+%
376   \or\expandafter\XINT_sind_C\the\numexpr-180+%
377   \or\expandafter\XINT_sind_D\the\numexpr-270+%
378   \else\expandafter\XINT_sind_E\the\numexpr-360+%
379   \fi#1.%
380 }%
381 \def\XINT_tmpa #1.#2.{%
382 \def\XINT_sind_A##1.##2.%
383 {%
384   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
385     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
386 }%
387 \def\XINT_sind_B_n-##1.##2.%
388 {%
389   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
390     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
391 }%
392 \def\XINT_sind_B_p##1.##2.%
393 {%
394   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
395     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
396 }%
397 \def\XINT_sind_C_n-##1.##2.%
398 {%
399   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
400     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
401 }%
402 \def\XINT_sind_C_p##1.##2.%
403 {%
404   \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
405     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
406 }%
407 \def\XINT_sind_D_n-##1.##2.%
408 {%
409   \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
410     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
411 }%
412 \def\XINT_sind_D_p##1.##2.%
413 {%
414   \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
415     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
416 }%
417 \def\XINT_sind_E-##1.##2.%
418 {%
419   \xintiiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter

```

```

420         {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
421 }%
422 }\expandafter\XINT_tmpa
423   \the\numexpr\XINTdigitsormax+4\expandafter.%
424   \romannumeral`&&\xintbarefloateval @oneDegree\relax.%
425 \def\XINT_sind_B#1{\xint_UDsignfork#1\XINT_sind_B_n-\XINT_sind_B_p\krof #1}%
426 \def\XINT_sind_C#1{\xint_UDsignfork#1\XINT_sind_C_n-\XINT_sind_C_p\krof #1}%
427 \def\XINT_sind_D#1{\xint_UDsignfork#1\XINT_sind_D_n-\XINT_sind_D_p\krof #1}%

```

### 28.8.3. @cosd\_rr() function and its support macro \xintCosd

```

428 \def\XINT_flexpr_func_@cosd_rr #1#2#3%
429 {%
430   \expandafter #1\expandafter #2\expandafter{%
431     \romannumeral`&&\XINT:NEhook:f:one:from:one{\romannumeral`&&\xintCosd#3}}%
432 }%

```

ATTENTION ONLY FOR POSITIVE ARGUMENTS

```

433 \def\XINT_tmpa #1.{%
434 \def\xintCosd##1%
435 {%
436   \romannumeral`&&\expandafter\xintcosd\romannumeral0\XINTinfloatS[#1]{##1}}%
437 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+12.%
438 \def\xintcosd #1[#2#3]%
439 {%
440   \xint_UDsignfork
441     #2\XINT_cosd
442     -\XINT_cosd_int
443   \krof#2#3.#1..%
444 }%
445 \def\XINT_tmpa #1.{%
446 \def\XINT_cosd ##1.##2.%
447 {%
448   \expandafter\XINT_cosd_a
449   \romannumeral0\xinttrunc{#1}{##2[##1]}%
450 }%
451 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+5.%
452 \def\XINT_cosd_a{\expandafter\XINT_cosd_i\the\numexpr\XINT_mod_ccclx_i0.}%
453 \def\XINT_cosd_int
454 {%
455   \expandafter\XINT_cosd_i\the\numexpr\expandafter\XINT_mod_ccclx_i
456 }%
457 \def\XINT_cosd_i #1.%
458 {%
459   \ifcase\numexpr#1/90\relax
460     \expandafter\XINT_cosd_A
461   \or\expandafter\XINT_cosd_B\the\numexpr-90+%
462   \or\expandafter\XINT_cosd_C\the\numexpr-180+%
463   \or\expandafter\XINT_cosd_D\the\numexpr-270+%
464   \else\expandafter\XINT_cosd_E\the\numexpr-360+%
465   \fi#1.%
466 }%

```

#2 will be empty in the "integer" branch, but attention in general branch to handling of negative

integer part after the subtraction of 90, 180, 270, or 360.

```

467 \def\XINT_tmpa#1.#2.{%
468 \def\XINT_cosd_A##1.##2.%
469 {%
470   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
471     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
472 }%
473 \def\XINT_cosd_B_n-##1.##2.%
474 {%
475   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
476     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
477 }%
478 \def\XINT_cosd_B_p##1.##2.%
479 {%
480   \xintiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
481     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
482 }%
483 \def\XINT_cosd_C_n-##1.##2.%
484 {%
485   \xintiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
486     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
487 }%
488 \def\XINT_cosd_C_p##1.##2.%
489 {%
490   \xintiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
491     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
492 }%
493 \def\XINT_cosd_D_n-##1.##2.%
494 {%
495   \xintiopp\XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
496     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
497 }%
498 \def\XINT_cosd_D_p##1.##2.%
499 {%
500   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@sin_series\expandafter
501     {\romannumeral0\XINTinfloat[#1]{\xintMul{##1.##2}#2}}%
502 }%
503 \def\XINT_cosd_E-##1.##2.%
504 {%
505   \XINT_expr_unlock\expandafter\XINT_flexpr_userfunc_@cos_series\expandafter
506     {\romannumeral0\XINTinfloat[#1]{\xintMul{\xintSub{##1[0]}{.##2}}#2}}%
507 }%
508 }\expandafter\XINT_tmpa
509   \the\numexpr\XINTdigitsormax+4\expandafter.%
510   \romannumeral`&&@\xintbarefloateval @oneDegree\relax.%
511 \def\XINT_cosd_B#1{\xint_UDsignfork#1\XINT_cosd_B_n-\XINT_cosd_B_p\krof #1}%
512 \def\XINT_cosd_C#1{\xint_UDsignfork#1\XINT_cosd_C_n-\XINT_cosd_C_p\krof #1}%
513 \def\XINT_cosd_D#1{\xint_UDsignfork#1\XINT_cosd_D_n-\XINT_cosd_D_p\krof #1}%

```



## 28.9. @sind(), @cosd()

The -45 is stored internally as -45/1[0] from the action of the unary minus operator, which float macros then parse faster. The 45e0 is to let it become 45[0] and not simply 45.

Here and below the `\ifnum\XINTdigits>8 45\else60\fi` will all be resolved at time of definition. This is the charm and power of expandable parsers!

```

514 \xintdeffloatfunc @sind(x) := (x)??
515     {(x>=\ifnum\XINTdigits>8 45\else60\fi)?
516     {@sin_series(x*@oneDegree)}
517     {@sind_rr(-x)}
518     }
519     {0e0}
520     {(x<=\ifnum\XINTdigits>8 45\else60\fi e0)?
521     {@sin_series(x*@oneDegree)}
522     {@sind_rr(x)}
523     }
524     ;%
525 \xintdeffloatfunc @cosd(x) := (x)??
526     {(x>=\ifnum\XINTdigits>8 45\else60\fi)?
527     {@cos_series(x*@oneDegree)}
528     {@cosd_rr(-x)}
529     }
530     {1e0}
531     {(x<=\ifnum\XINTdigits>8 45\else60\fi e0)?
532     {@cos_series(x*@oneDegree)}
533     {@cosd_rr(x)}
534     }
535     ;%

```

## 28.10. @sin(), @cos()

For some reason I did not define sin() and cos() in January 2019 ??

The sub `\xintexpr x*@oneRadian\relax` means that the multiplication will be done exactly @oneRadian having its 12 extra digits (and x its 4 extra digits), before being rounded in entrance of `\xintSind`, respectively `\xintCosd`, to P+12 mantissa.

The strange 79e-2 could be 0.79 which would give 79[-2] internally too.

```

536 \xintdeffloatfunc @sin(x):= (abs(x)<\ifnum\XINTdigits>8 79e-2\else1e0\fi)?
537     {@sin_series(x)}
538     {(x)??
539     {@sind_rr(-\xintexpr x*@oneRadian\relax)}
540     {0}
541     {@sind_rr(\xintexpr x*@oneRadian\relax)}
542     }
543     ;%
544 \xintdeffloatfunc @cos(x):= (abs(x)<\ifnum\XINTdigits>8 79e-2\else1e0\fi)?
545     {@cos_series(x)}
546     {@cosd_rr(abs(\xintexpr x*@oneRadian\relax))}
547     ;%

```

## 28.11. @sinc()

Should I also consider adding  $(1-\cos(x))/(x^2/2)$  ? it is  $\text{sinc}^2(x/2)$  but avoids a square.

```

548 \xintdeffloatfunc @sinc(x):= (abs(x)<\ifnum\XINTdigits>8 79e-2\else1e0\fi) ?

```



```

549             {@sin_aux(sqr(x))}
550             {@sind_rr(\xintexpr abs(x)*@oneRadian\relax)/abs(x)}
551             ;%

```

## 28.12. @tan(), @tand(), @cot(), @cotd()

The 0 in cot(x) is a dummy place holder. We don't have a notion of Inf yet.

```

552 \xintdeffloatfunc @tand(x) := @sind(x)/@cosd(x);%
553 \xintdeffloatfunc @cotd(x) := @cosd(x)/@sind(x);%
554 \xintdeffloatfunc @tan(x) := (x)??
555             {(x>-\ifnum\XINTdigits>8 79e-2\else1e0\fi)?
556             {@sin(x)/@cos(x)}
557             {@cotd(\xintexpr9e1+x*@oneRadian\relax)}
558             }
559             }
560             {0e0}
561             {(x<\ifnum\XINTdigits>8 79e-2\else1e0\fi)?
562             {@sin(x)/@cos(x)}
563             {@cotd(\xintexpr9e1-x*@oneRadian\relax)}
564             }
565             ;%
566 \xintdeffloatfunc @cot(x) := (abs(x)<\ifnum\XINTdigits>8 79e-2\else1e0\fi)?
567             {@cos(x)/@sin(x)}
568             {(x)??
569             {@tand(\xintexpr9e1+x*@oneRadian\relax)}
570             {0}
571             {@tand(\xintexpr9e1-x*@oneRadian\relax)}
572             };%

```

## 28.13. @sec(), @secd(), @csc(), @cscd()

```

573 \xintdeffloatfunc @sec(x) := inv(@cos(x));%
574 \xintdeffloatfunc @csc(x) := inv(@sin(x));%
575 \xintdeffloatfunc @secd(x) := inv(@cosd(x));%
576 \xintdeffloatfunc @cscd(x) := inv(@sind(x));%

```

## 28.14. Core routine for inverse trigonometry

I always liked very much the general algorithm whose idea I found in 2019. But it costs a square root plus a sine plus a cosine all at target precision. For the arctangent the square root will be avoided by a trick. (memo: it is replaced by a division and I am not so sure now this is advantageous in fact)

And now I like it even more as I have re-done the first step entirely in a single `\numexpr...` Thus the inverse trigonometry got a serious improvement at 1.4e...

Here is the idea. We have  $0 < t < \sqrt{2}/2$  and we want  $a = \text{Arcsin } t$ .

Imagine we have some very good approximation  $b = a - h$ . We know  $b$ , and don't know yet  $h$ . No problem  $h$  is  $a - b$  so  $\sin(h) = \sin(a)\cos(b) - \cos(a)\sin(b)$ . And we know everything here:  $\sin(a)$  is  $t$ ,  $\cos(a)$  is  $u = \sqrt{1-t^2}$ , and we can compute  $\cos(b)$  and  $\sin(b)$ .

I said  $h$  was small so the computation of  $\sin(a)\cos(b) - \cos(a)\sin(b)$  will involve a lot of cancellation, no problem with `xint`, as it knows how to compute exactly... and if we wanted to go very low level we could do  $\cos(a)\sin(b)$  paying attention only on least significant digits.

Ok, so we have  $\sin(h)$ , but  $h$  is small, so the series of Arcsine can be used with few terms!

In fact  $h$  will be at most of the order of  $1e-9$ , so it is no problem to simply replace  $\sin(h)$  with  $h$  if the target precision is 16 !

Ok, so how do we obtain  $b$ , the good approximation to  $\text{Arcsin } t$  ? Simply by using its Taylor series, embedded in a single `\numexpr` working with nine digits numbers... I like this one! Notice that it reminisces with my questioning about how to best do Horner like for sine and cosine. Here in `\numexpr` we can only manipulate whole integers and simply can't do things such as  $\dots *x + 5/112) *x + 3/40) *x + 1/6) *x + 1 \dots$ . But I found another way, see the code, which uses extensively the "scaling" operations in `\numexpr`.

I have not proven rigorously that  $b-a$  is always less or equal in absolute value than  $1e-9$ , but it is possible for example in Python to program it and go through all possible (less than)  $1e9$  inputs and check what happens.

Very small inputs will give  $b=0$  (first step is a fixed point rounding of  $t$  to nine fractional digits, so this rounding gives zero for input  $<0.5e-9$ , others will give  $b=t$ , because the arcsine `numexpr` will end up with 1000000000 (last time I checked that was for  $t$  a bit less than  $5e-5$ , the latter gives 1000000001). All seems to work perfectly fine, in practice...

First we let the `@sin_aux()` and `@cos_aux()` functions be usable in exact `\xintexpr` context.

The `@asin_II()` function will be used only for `Digits>16`.

```

577 \expandafter\let\csname XINT_expr_func_@sin_aux\expandafter\endcsname
578         \csname XINT_flexpr_func_@sin_aux\endcsname
579 \expandafter\let\csname XINT_expr_func_@cos_aux\expandafter\endcsname
580         \csname XINT_flexpr_func_@cos_aux\endcsname
581 \ifnum\XINTdigits>16
582 \def\XINT_flexpr_func_@asin_II#1#2#3%
583 {%
584     \expandafter #1\expandafter #2\expandafter{%
585         \romannumeral`&&\XINT:NEhook:f:one:from:one
586         {\romannumeral`&&\XINT_Arcsin_II_a#3}}}%
587 }%
588 \def\XINT_tmpc#1.%
589 {%
590 \def\XINT_Arcsin_II_a##1%
591 {%
592     \expandafter\XINT_Arcsin_II_c_i\romannumeral0\XINTinfloatS[#1]{##1}%
593 }%
594 \def\XINT_Arcsin_II_c_i##1[##2]%
595 {%
596     \xintAdd{1/1[0]}{##1/6[##2]}%
597 }%
598 }%
599 \expandafter\XINT_tmpc\the\numexpr\XINTdigitsormax-14.%
600 \fi
601 \ifnum\XINTdigits>34
602 \def\XINT_tmpc#1.#2.#3.#4.%
603 {%
604 \def\XINT_Arcsin_II_a##1%
605 {%
606     \expandafter\XINT_Arcsin_II_a_iii\romannumeral0\XINTinfloatS[#1]{##1}\xint:
607 }%
608 \def\XINT_Arcsin_II_a_iii##1\xint:
609 {%
610     \expandafter\XINT_Arcsin_II_b\romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
611 }%

```

## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfac, xintexpr, xinttrig, xintlog

```

612 \def\XINT_Arcsin_II_b##1\xint:
613 {%
614     \expandafter\XINT_Arcsin_II_c_i
615     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{#3}{##1}}}\xint:
616 }%
617 \def\XINT_Arcsin_II_c_i##1\xint:##2\xint:
618 {%
619     \xintAdd{1/1[0]}{\XINTinFloat[#1]{\xintMul{##1}{##2}}}%
620 }%
621 }%
622 \expandafter\XINT_tmpc
623 \the\numexpr\XINTdigitsormax-14\expandafter.%
624 \the\numexpr\XINTdigitsormax-32\expandafter.\expanded{%
625     \XINTinFloat[\XINTdigitsormax-32]{3/40[0]}.%
626     \XINTinFloat[\XINTdigitsormax-14]{1/6[0]}.%
627 }%
628 \fi
629 \ifnum\XINTdigits>52
630 \def\XINT_tmpc#1.#2.#3.#4.#5.%
631 {%
632 \def\XINT_Arcsin_II_a_iii##1\xint:
633 {%
634     \expandafter\XINT_Arcsin_II_a_iv\romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
635 }%
636 \def\XINT_Arcsin_II_a_iv##1\xint:
637 {%
638     \expandafter\XINT_Arcsin_II_b\romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
639 }%
640 \def\XINT_Arcsin_II_b##1\xint:
641 {%
642     \expandafter\XINT_Arcsin_II_c_ii
643     \romannumeral0\xintadd{#4}{\XINTinfloat[#2]{\xintMul{#3}{##1}}}\xint:
644 }%
645 \def\XINT_Arcsin_II_c_ii##1\xint:##2\xint:
646 {%
647     \expandafter\XINT_Arcsin_II_c_i
648     \romannumeral0\xintadd{#5}{\XINTinFloat[#1]{\xintMul{##1}{##2}}}\xint:
649 }%
650 }%
651 \expandafter\XINT_tmpc
652 \the\numexpr\XINTdigitsormax-32\expandafter.%
653 \the\numexpr\XINTdigitsormax-50\expandafter.\expanded{%
654     \XINTinFloat[\XINTdigitsormax-50]{5/112[0]}.%
655     \XINTinFloat[\XINTdigitsormax-32]{3/40[0]}.%
656     \XINTinFloat[\XINTdigitsormax-14]{1/6[0]}.%
657 }%
658 \fi
659 \def\XINT_flexpr_func_@asin_I#1#2#3%
660 {%
661     \expandafter #1\expandafter #2\expandafter{%
662     \romannumeral`&&\XINT:NEhook:f:one:from:one
663     {\romannumeral`&&\XINT_Arcsin_I#3}}%

```

## TOC

TOC, [xintkernel](#), [xinttools](#), [xintcore](#), [xint](#), [xintbinhex](#), [xintgcd](#), [xintfrac](#), [xintseries](#), [xintcfac](#), [xintexpr](#), [xinttrig](#), [xintlog](#)

```

664 }%
665 \def\XINT_Arcsin_I#1%
666 {%
667     \the\numexpr\expandafter\XINT_Arcsin_Ia\romannumeral0\xintiround9{#1}.%
668 }%
669 \def\XINT_Arcsin_Ia#1.%
670 {%
671     (\expandafter\XINT_Arcsin_Ib\the\numexpr#1*#1/\xint_c_x^ix.)%
672     #1/\xint_c_x^ix[-9]%
673 }%
674 \def\XINT_Arcsin_Ib#1.%
675 {(
676     % 3481/3660)*#1/\xint_c_x^ix+\xint_c_x^ix)%
677     % 3249/3422)*#1/\xint_c_x^ix+\xint_c_x^ix)%
678     % 3025/3192)*#1/\xint_c_x^ix+\xint_c_x^ix)%
679     % 2809/2970)*#1/\xint_c_x^ix+\xint_c_x^ix)%
680     % 2601/2756)*#1/\xint_c_x^ix+\xint_c_x^ix)%
681     % 2401/2550)*#1/\xint_c_x^ix+\xint_c_x^ix)%
682     % 2209/2352)*#1/\xint_c_x^ix+\xint_c_x^ix)%
683     % 2025/2162)*#1/\xint_c_x^ix+\xint_c_x^ix)%
684     (
685         %(\xint_c_x^ix*1849/1980)*#1/\xint_c_x^ix+\xint_c_x^ix)%
686         933838384*#1/\xint_c_x^ix+\xint_c_x^ix)%
687         1681/1806)*#1/\xint_c_x^ix+\xint_c_x^ix)%
688         1521/1640)*#1/\xint_c_x^ix+\xint_c_x^ix)%
689         1369/1482)*#1/\xint_c_x^ix+\xint_c_x^ix)%
690         1225/1332)*#1/\xint_c_x^ix+\xint_c_x^ix)%
691         1089/1190)*#1/\xint_c_x^ix+\xint_c_x^ix)%
692         961/1056)*#1/\xint_c_x^ix+\xint_c_x^ix)%
693         841/930)*#1/\xint_c_x^ix+\xint_c_x^ix)%
694         729/812)*#1/\xint_c_x^ix+\xint_c_x^ix)%
695         625/702)*#1/\xint_c_x^ix+\xint_c_x^ix)%
696         529/600)*#1/\xint_c_x^ix+\xint_c_x^ix)%
697         441/506)*#1/\xint_c_x^ix+\xint_c_x^ix)%
698         361/420)*#1/\xint_c_x^ix+\xint_c_x^ix)%
699         289/342)*#1/\xint_c_x^ix+\xint_c_x^ix)%
700         225/272)*#1/\xint_c_x^ix+\xint_c_x^ix)%
701         169/210)*#1/\xint_c_x^ix+\xint_c_x^ix)%
702         121/156)*#1/\xint_c_x^ix+\xint_c_x^ix)%
703         81/110)*#1/\xint_c_x^ix+\xint_c_x^ix)%
704         49/72)*#1/\xint_c_x^ix+\xint_c_x^ix)%
705         25/42)*#1/\xint_c_x^ix+\xint_c_x^ix)%
706         9/20)*#1/\xint_c_x^ix+\xint_c_x^ix)%
707         1/6)*#1/\xint_c_x^ix+\xint_c_x^ix
708     )%
709 \ifnum\XINTdigits>16
710     \xintdeffloatfunc @asin_o(D, T) := T + D*@asin_II(sqr(D));%
711     \xintdeffloatfunc @asin_n(V, T, t, u) :=%
712         @asin_o(\xintexpr t*@cos_aux(V) - u*T*@sin_aux(V)\relax, T);%
713 \else
714     \xintdeffloatfunc @asin_n(V, T, t, u) :=%
715         \xintexpr t*@cos_aux(V) - u*T*@sin_aux(V)\relax + T;%

```

```

716 \fi
717 \xintdeffloatfunc @asin_m(T, t, u) := @asin_n(sqr(T), T, t, u);%
718 \xintdeffloatfunc @asin_l(t, u)    := @asin_m(@asin_I(t), t, u);%

```

### 28.15. @asin(), @asind()

Only non-negative arguments  $t$  and  $u$  for  $\text{asin}_a(t,u)$ , and  $\text{asind}_a(t,u)$ .

```

719 \xintdeffloatfunc @asin_a(t, u) := (t<u)?
720     { @asin_l(t, u) }
721     { @Piover2 - @asin_l(u, t) }
722     ;%
723 \xintdeffloatfunc @asind_a(t, u) := (t<u)?
724     { @asin_l(t, u) * @oneRadian }
725     { 9e1 - @asin_l(u, t) * @oneRadian }
726     ;%
727 \xintdeffloatfunc @asin(t) := (t)??
728     { -@asin_a(-t, sqrt(1e0-sqr(t))) }
729     { 0e0 }
730     { @asin_a(t, sqrt(1e0-sqr(t))) }
731     ;%
732 \xintdeffloatfunc @asind(t) := (t)??
733     { -@asind_a(-t, sqrt(1e0-sqr(t))) }
734     { 0e0 }
735     { @asind_a(t, sqrt(1e0-sqr(t))) }
736     ;%

```

### 28.16. @acos(), @acosd()

```

737 \xintdeffloatfunc @acos(t) := @Piover2 - @asin(t);%
738 \xintdeffloatfunc @acosd(t) := 9e1 - @asind(t);%

```

### 28.17. @atan(), @atand()

Uses same core routine  $\text{asin}_l()$  as for  $\text{asin}()$ , but avoiding a square-root extraction in preparing its arguments (to the cost of computing an inverse, rather).

radians

```

739 \xintdeffloatfunc @atan_b(t, w, z) := 5e-1 * (w< 0)?
740     { @Pi - @asin_a(2e0*z * t, -w*z) }
741     { @asin_a(2e0*z * t, w*z) }
742     ;%
743 \xintdeffloatfunc @atan_a(t, T) := @atan_b(t, 1e0-T, inv(1e0+T));%
744 \xintdeffloatfunc @atan(t) := (t)??
745     { -@atan_a(-t, sqr(t)) }
746     { 0 }
747     { @atan_a(t, sqr(t)) }
748     ;%

```

degrees

```

749 \xintdeffloatfunc @atand_b(t, w, z) := 5e-1 * (w< 0)?
750     { 18e1 - @asind_a(2e0*z * t, -w*z) }
751     { @asind_a(2e0*z * t, w*z) }
752     ;%
753 \xintdeffloatfunc @atand_a(t, T) := @atand_b(t, 1e0-T, inv(1e0+T));%

```

```

754 \xintdeffloatfunc @atand(t) := (t)??
755                               {-@atand_a(-t, sqr(t))}
756                               {0}
757                               {@atand_a(t, sqr(t))}
758                               ;%

```

### 28.18. @Arg(), @atan2(), @Argd(), @atan2d(), @pArg(), @pArgd()

Arg(x,y) function from  $-\pi$  (excluded) to  $+\pi$  (included)

```

759 \xintdeffloatfunc @Arg(x, y) := (y>x)?
760                               {(y>-x)?
761                               {@Piover2 - @atan(x/y)}
762                               {(y<0)?
763                               {-@Pi + @atan(y/x)}
764                               {@Pi + @atan(y/x)}
765                               }
766                               }
767                               {(y>-x)?
768                               {@atan(y/x)}
769                               {-@Piover2 + @atan(x/-y)}
770                               }
771                               ;%

```

atan2(y,x) = Arg(x,y) ... (some people have atan2 with arguments reversed but the convention here seems the most often encountered)

```

772 \xintdeffloatfunc @atan2(y,x) := @Arg(x, y);%

```

Argd(x,y) function from -180 (excluded) to +180 (included)

```

773 \xintdeffloatfunc @Argd(x, y) := (y>x)?
774                               {(y>-x)?
775                               {9e1 - @atand(x/y)}
776                               {(y<0)?
777                               {-18e1 + @atand(y/x)}
778                               {18e1 + @atand(y/x)}
779                               }
780                               }
781                               {(y>-x)?
782                               {@atand(y/x)}
783                               {-9e1 + @atand(x/-y)}
784                               }
785                               ;%

```

atan2d(y,x) = Argd(x,y)

```

786 \xintdeffloatfunc @atan2d(y,x) := @Argd(x, y);%

```

pArg(x,y) function from 0 (included) to  $2\pi$  (excluded) I hesitated between pArg, Argpos, and Arg-plus. Opting for pArg in the end.

```

787 \xintdeffloatfunc @pArg(x, y) := (y>x)?
788                               {(y>-x)?
789                               {@Piover2 - @atan(x/y)}
790                               {@Pi + @atan(y/x)}
791                               }
792                               {(y>-x)?
793                               {(y<0)?
794                               {@twoPi + @atan(y/x)}

```

```

795             {@atan(y/x)}
796         }
797         {@threePiover2 + @atan(x/-y)}
798     }
799     ;%

pArgd(x,y) function from 0 (included) to 360 (excluded)
800 \xintdeffloatfunc @pArgd(x, y):=(y>x)?
801     {(y>-x)?
802         {9e1 - @atan(x/y)*@oneRadian}
803         {18e1 + @atan(y/x)*@oneRadian}
804     }
805     {(y>-x)?
806         {(y<0e0)?
807             {36e1 + @atan(y/x)*@oneRadian}
808             {@atan(y/x)*@oneRadian}
809         }
810         {27e1 + @atan(x/-y)*@oneRadian}
811     }
812     ;%

```

## 28.19. Restore `\xintdeffloatfunc` to its normal state, with no extra digits

```

813 \expandafter\let
814     \csname XINT_flexpr_exec_+\expandafter\endcsname
815     \csname XINT_flexpr_exec_+\endcsname
816 \expandafter\let
817     \csname XINT_flexpr_exec_-\expandafter\endcsname
818     \csname XINT_flexpr_exec_-\endcsname
819 \expandafter\let
820     \csname XINT_flexpr_exec_*\expandafter\endcsname
821     \csname XINT_flexpr_exec_*\endcsname
822 \expandafter\let
823     \csname XINT_flexpr_exec_/\expandafter\endcsname
824     \csname XINT_flexpr_exec_/\endcsname
825 \expandafter\let
826     \csname XINT_flexpr_func_sqr\expandafter\endcsname
827     \csname XINT_flexpr_sqrfunc\endcsname
828 \expandafter\let
829     \csname XINT_flexpr_func_sqrt\expandafter\endcsname
830     \csname XINT_flexpr_sqrtfunc\endcsname
831 \expandafter\let
832     \csname XINT_flexpr_func_inv\expandafter\endcsname
833     \csname XINT_flexpr_invfunc\endcsname

```

## 28.20. Let the functions be known to the `\xintexpr` parser

We use here `float_dgtormax` which uses the smaller of `Digits` and 64.

```

834 \edef\xINTinFloatdigitsormax{\noexpand\xINTinFloat[\the\numexpr\xINTdigitsormax]}%
835 \edef\xINTinFloatSdigitsormax{\noexpand\xINTinFloatS[\the\numexpr\xINTdigitsormax]}%
836 \xintFor #1 in {sin, cos, tan, sec, csc, cot,
837               asin, acos, atan}\do
838 {%

```

```

839 \xintdeffloatfunc #1(x) := float_dgtormax(@#1(x));%
840 \xintdeffloatfunc #1d(x) := float_dgtormax(@#1d(x));%
841 \xintdeffunc #1(x) := float_dgtormax(\xintfloatexpr @#1(sfloat_dgtormax(x))\relax);%
842 \xintdeffunc #1d(x) := float_dgtormax(\xintfloatexpr @#1d(sfloat_dgtormax(x))\relax);%
843 }%
844 \xintFor #1 in {Arg, pArg, atan2}\do
845 {%
846 \xintdeffloatfunc #1(x, y) := float_dgtormax(@#1(x, y));%
847 \xintdeffloatfunc #1d(x, y) := float_dgtormax(@#1d(x, y));%
848 \xintdeffunc #1(x, y) :=
849 float_dgtormax(\xintfloatexpr @#1(sfloat_dgtormax(x), sfloat_dgtormax(y))\relax);%
850 \xintdeffunc #1d(x, y) :=
851 float_dgtormax(\xintfloatexpr @#1d(sfloat_dgtormax(x), sfloat_dgtormax(y))\relax);%
852 }%
853 \xintdeffloatfunc sinc(x) := float_dgtormax(@sinc(x));%
854 \xintdeffunc sinc(x) := float_dgtormax(\xintfloatexpr @sinc(sfloat_dgtormax(x))\relax);%

```

## 28.21. Synonyms: @tg(), @cotg()

These are my childhood notations and I am attached to them. In radians only, and for `\xintfloateval` only. We skip some overhead here by using a `\let` at core level.

```

855 \expandafter\let\csname XINT_flexpr_func_tg\expandafter\endcsname
856 \csname XINT_flexpr_func_tan\endcsname
857 \expandafter\let\csname XINT_flexpr_func_cotg\expandafter\endcsname
858 \csname XINT_flexpr_func_cot\endcsname

```

## 28.22. Final clean-up

Restore used dummy variables to their status prior to the package reloading. On first loading this is not needed, but I have not added a way to check here whether this a first loading or a re-loading.

```

859 \xintdefvar twoPi := float_dgtormax(@twoPi);%
860 \xintdefvar threePiover2 := float_dgtormax(@threePiover2);%
861 \xintdefvar Pi := float_dgtormax(@Pi);%
862 \xintdefvar Piover2 := float_dgtormax(@Piover2);%
863 \xintdefvar oneDegree := float_dgtormax(@oneDegree);%
864 \xintdefvar oneRadian := float_dgtormax(@oneRadian);%
865 \xintunassignvar{@twoPi}\xintunassignvar{@threePiover2}%
866 \xintunassignvar{@Pi}\xintunassignvar{@Piover2}%
867 \xintunassignvar{@oneRadian}\xintunassignvar{@oneDegree}%
868 \xintFor* #1 in {iDTVtuwxyzX}\do{\xintrestorevariable{#1}}%
869 \XINTtrigendinput%

```



## 29. Package `xintlog` implementation

### Contents

29.1	Catcodes, $\varepsilon$ -TeX and reload detection	713
29.2	Library identification	715
29.3	<code>\xintreloadxintlog</code>	715
29.4	Loading the <code>poormanlog</code> package	715
29.5	Macro layer on top of the <code>poormanlog</code> package	715
29.5.1	<code>\PoorManLogBaseTen</code> , <code>\PoorManLog</code>	716
29.5.2	<code>\PoorManPowerOfTen</code> , <code>\PoorManExp</code>	716
29.5.3	Removed: <code>\PoorManPower</code> , see <code>\XINTinFloatSciPow</code>	717
29.6	Macro support for powers	718
29.6.1	<code>\XINTinFloatSciPow</code>	718
29.6.2	<code>\xintPow</code>	720
29.7	Macro support for <code>\xintexpr</code> and <code>\xintfloatexpr</code> syntax	722
29.7.1	The <code>log10()</code> and <code>pow10()</code> functions	722
29.7.2	The <code>log()</code> , <code>exp()</code> functions	722
29.7.3	The <code>pow()</code> function	723
29.8	End of package loading for low Digits	723
29.9	Stored constants	724
29.10	April 2021: at last, <code>\XINTinFloatPowTen</code> , <code>\XINTinFloatExp</code>	727
29.10.1	Exponential series	730
29.11	April 2021: at last <code>\XINTinFloagLogTen</code> , <code>\XINTinFloatLog</code>	733
29.11.1	Log series, case II	738
29.11.2	Log series, case III	743

In 2019, at 1.3e release I almost included extended precision for `log()` and `exp()` but the time I could devote to `xint` expired. Finally, at long last, (and I had procrastinated far more than the two years since 2019) the 1.4e release in April 2021 brings `log10()`, `pow10()`, `log()`, `pow()` to `P=Digits` precision: up to 62 digits with at least (said roughly) 99% chances of correct rounding (the design is targeting less than about  $0.005\text{ulp}$  distance to mathematical value, before rounding).

Implementation is EXPERIMENTAL.

For up to `Digits=8`, it is simply based upon the `poormanlog` package. The probability of correct rounding will be less than for `Digits>8`, especially in the cases of `Digits=8` and to a lesser extent `Digits=7`. And, for all `Digits≤8`, there is a systematic loss of rounding precision in the floating point sense in the case of `log10(x)` for inputs close to 1:

Summary of limitations of `log10()` and `pow10()` in the case of `Digits≤8`:

- For `log10(x)` with `x` near 1, the precision of output as floating point will be mechanically reduced from the fact that this is based on a fixed point result, for example `log10(1.0011871)` is produced as `5.15245e-4`, which stands for `0.000515145` having indeed 9 correct fractional digits, but only 6 correct digits in the floating point sense.

This feature affects the entire range `Digits≤8`.

- Even if limiting to inputs `x` with  $1.26 < x < 10$  (`1.26` is a bit more than  $10^{0.1}$  hence its choice as lower bound), the `poormanlog` documentation mentions an absolute error possibly up to about  $1e-9$ . In practice a test of 10000 random inputs  $1.26 < x < 10$  revealed 9490 correctly rounded `log10(x)` at 8 digits (and the 510 non-correctly rounded ones with an error of 1 in last digit compared to correct rounding). So correct rounding achieved only in about 95% of cases here.

At 7 digits the same 10000 random inputs are correctly rounded in 99.4% of cases, and at 6 digits it is 99.94% of cases.

Again with `Digits=8`, the `log10(i)` for `i` in `1..1000` are all correctly rounded to 8 digits with two exceptions: `log10(3)` and `log10(297)` with a `ulp` error. And the `log(i)` in the same range are

correctly rounded to 8 digits with the 15 exceptions  $i = 99, 105, 130, 178, 224, 329, 446, 464, 564, 751, 772, 777, 886, 907, 962$ , whose natural logarithms are obtained with a *ulp* error.

- Regarding the computation of  $10^x$ , I obtained for  $-1 < x < 1$  the following with 10000 random inputs: 518/10000 errors at *ulp*, 60/10000, and 8/10000, at respectively Digits = 8, 7, 6 so chances of correct rounding are respectively about 95%, 99.4% and more than 99.9%.

Despite its limitations the *poormanlog* based approach used for Digits up to 8 has the advantage of speed (at least 8X compared to working with 16 digits) and is largely precise enough for plots.

For 9 digits or more, the observed precision in some random tests appears to be at least of 99.9% chances of correct rounding, and the  $\log_{10}(x)$  with  $x$  near 1 are correctly (if not really efficiently) handled in the floating point sense for the output. The *poormanlog* approximate  $\log_{10}()$  is still used to boot-strap the process, generally. The *pow10()* at Digits=9 or more is done independently of *poormanlog*.

All of this is done on top of my 2013 structures for floating point computations which have always been marked as provisory and rudimentary and instills intrinsic non-efficiency:

- no internal data format for a ``floating point number at P digits'',
- mantissa lengths are again and again computed,
- digits are not pre-organized say in blocks of 4 by 4 or 8 by 8,
- floating point multiplication is done via an *\*exact\** multiplication, then rounding to P digits!

This is legacy of the fact that the project was initially devoted to big integers only, but in the weeks that followed its inception in March 2013 I added more and more functionalities without a well laid out preliminary plan.

Anyway, for years I have felt a better foundation would help achieve at least something such as 2X gain (perhaps the last item by itself, if improved upon, would bring most of such 2X gain?)

I did not try to optimize for the default 16 digits, the goal being more of having a general scalable structure in place and there is no difficulty to go up to 100 digits precision if one stores extended pre-computed constants and increases the length of the ``series'' support.

Apart from  $\log(10)$  and its inverse, no other logarithms are stored or pre-computed: the rest of the stored data is the same for *pow10()* and  $\log_{10}()$  and consists of the fractional powers  $10^{\pm 0.i}$ ,  $10^{\pm 0.0i}$ , ...,  $10^{\pm 0.000000i}$  at P+5 and also at P+10 digits.

In order to reduce the loading time of the package the inverses are not computed internally (as this would require costly divisions) but simply hard-coded with enough digits to cover the allowed Digits range.

## 29.1. Catcodes, $\varepsilon$ -TeX and reload detection

**Modified at 1.41 (2022/05/29).** Silly paranoid modification of `\z` in case `{` and `}` do not have their normal catcodes when *xintlog.sty* is reloaded (initial loading via *xintexpr.sty* does not need this), to define `\XINTlogendinput` there and not after the `\endgroup` from `\z` has already restored possibly bad catcodes.

1.41 handles much better the situation with `\usepackage{xintlog}` without previous loading of *xintexpr* (or same with `\input` and *etex*). Instead of aborting with a message (which actually was wrong with LaTeX since 1.4e, mentioning `\input` in place of `\usepackage`), it will initiate loading *xintexpr* itself. This required an adaptation at end of *xintexpr* and some care to not leave bad catcodes.

```
1 \begingroup\catcode61\catcode48\catcode32=10\relax%
2 \catcode13=5 % ^^M
3 \endlinechar=13 %
4 \catcode123=1 % {
5 \catcode125=2 % }
6 \catcode64=11 % @
7 \catcode35=6 % #
```

```

8 \catcode44=12 % ,
9 \catcode46=12 % .
10 \catcode58=12 % :
11 \catcode94=7 % ^
12 \def\empty{}\def\space{ }\newlinechar10
13 \def\z{\endgroup}%
14 \expandafter\let\expandafter\x\csname ver@xintlog.sty\endcsname
15 \expandafter\let\expandafter\w\csname ver@xintexpr.sty\endcsname
16 \expandafter
17 \ifx\csname PackageWarningNoLine\endcsname\relax
18 \def\y#1#2{\immediate\write128{^^JPackage #1 Warning:^^J%
19 \space\space\space\space#2.^^J}}%
20 \else
21 \def\y#1#2{\PackageWarningNoLine{#1}{#2}}%
22 \fi
23 \expandafter
24 \ifx\csname numexpr\endcsname\relax
25 \y{xintlog}{\numexpr not available, aborting input}%
26 \def\z{\endgroup\endinput}%
27 \else
28 \ifx\w\relax % xintexpr.sty not yet loaded.
29 \edef\MsgBrk{^^J\space\space\space\space}%
30 \y{xintlog}%
31 {\ifx\x\empty
32 xintlog should not be loaded directly\MessageBreak
33 The correct way is \string\usepackage{xintexpr}.\MessageBreak
34 Will try that now%
35 \else
36 First loading of xintlog.sty should be via
37 \string\input\space xintexpr.sty\relax\MsgBrk
38 Will try that now%
39 \fi
40 }%
41 \ifx\x\empty
42 \def\z{\endgroup\RequirePackage{xintexpr}\endinput}%
43 \else
44 \def\z{\endgroup\input xintexpr.sty\relax\endinput}%
45 \fi
46 \else
47 \def\z{\endgroup\edef\XINTlogendinput{\XINTrestorecatcodes\noexpand\endinput}}%
48 \fi
49 \fi
50 \z%

```

Here we set catcodes to the package values, the current settings having been saved in the `\XINTlogendinput` macro. We arrive here only if xintlog is either loaded from xintexpr or is being reloaded via an `\input` from `\xintreloadxintlog`. Else we aborted right before via `\endinput` and do not modify catcodes. As xintexpr inputs xintlog.sty at a time the catcode configuration is already the package one we pay attention to not use `\XINTsetupcatcodes` which would badly redefine `\XINTrestorecatcodesendinput` as executed at end of xintexpr.sty. There is slight inefficiency here to execute `\XINTsetcatcodes` when xintexpr initiated the xintlog loading, but let's live with it.

```
51 \XINTsetcatcodes%
```

## 29.2. Library identification

If the file has already been loaded, let's skip the `\ProvidesPackage`. Else let's do it and set a flag to indicate loading happened at least once already.

Modified at 1.41 (2022/05/29). Message also to Terminal not only log file.

```

52 \ifcsname xintlibver@log\endcsname
53   \expandafter\xint_firstoftwo
54 \else
55   \expandafter\xint_secondoftwo
56 \fi
57 {\immediate\write128{Reloading xintlog library using Digits=\xinttheDigits.}}%
58 {\expandafter\gdef\csname xintlibver@log\endcsname{2025/09/06 v1.4o}}%
59   \XINT_providespackage
60   \ProvidesPackage{xintlog}%
61   [2025/09/06 v1.4o Logarithms and exponentials for xintexpr (JFB)]%
62 }%
```

## 29.3. `\xintreloadxintlog`

Now needed at 1.4e.

```

63 \def\xintreloadxintlog{\input xintlog.sty }%
```

## 29.4. Loading the poormanlog package

Attention to the catcode regime when loading poormanlog.

Also, for xintlog.sty to be multiple-times loadable, we need to avoid using LaTeX's `\RequirePackage` twice.

```

64 \xintexprSafeCatcodes
65 \unless\ifdefined\XINTinFloatPowTen
66 \ifdefined\RequirePackage
67   \RequirePackage{poormanlog}%
68 \else
69   \input poormanlog.tex
70 \fi\fi
71 \xintexprRestoreCatcodes
```

## 29.5. Macro layer on top of the poormanlog package

This was moved here with some macro renames from xintfrac on occasion of 1.4e release.

Breaking changes at 1.4e:

- `\poormanloghack` now a no-op (removed at 1.4m),
- `\xintLog` was used for `\xinteval` and differed slightly from its counterpart used for `\xintfloateval`, the latter float-rounded to  $P = \text{Digits}$ , the former did not and kept completely meaning-less digits in output. Both macros now replaced by a `\PoorManLog` which will always float round the output to  $P = \text{Digits}$ . Because xint does not really implement a fixed point interface anyhow.
- `\xintExp` (used in `\xinteval`) and another macro (used in `\xintfloateval`) did not use a sufficiently long approximation to  $1/\log(10)$  to support precisely enough  $\exp(x)$  if output of the order of  $10^{10000}$  for example, (last two digits wrong then) and situation became worse for very high values such as  $\exp(1e8)$  which had only 4 digits correct.

The new `\PoorManExp` which replaces them is more careful... and for example  $\exp(12345678)$  obtains correct rounding ( $\text{Digits}=8$ ).

- `\XINTinFloatxintLog` and `\XINTinFloatxintExp` were removed; they were used for `log()` and `exp()` in `\xintfloateval`, and differed from `\xintLog` and `\xintExp` a bit, now renamed to `\PoorManLog` and `\PoorManExp`.

- `\PoorManPower` has simply disappeared, see `\XINTinFloatSciPow` and `\xintPow`.

See the general `xintlog` introduction for some comments on the achieved precision and probabilities of correct rounding.

### 29.5.1. `\PoorManLogBaseTen`, `\PoorManLog`

1.3f. Code originally in `poormanlog v0.04` got transferred here. It produces the logarithm in base 10 with an error (believed to be at most) of the order of 1 unit in the 9th (i.e. last, fixed point) fractional digit. Testing seems to indicate the error is never exceeding 2 units in the 9th place, in worst cases.

These macros will still be the support macros for `\xintfloatexpr log10()`, `pow10()`, etc... up to `Digits=8` and the `poormanlog` logarithm is used as starting point for higher precision if `Digits` is at least 9.

Notice that `\PML@999999999.` expands (in `\numexpr`) to `1000000000` (ten digits), which is the only case with the output having ten digits. But there is no need here to treat this case especially, it works fine in `\PML@logbaseten`.

Breaking change at 1.4e: for consistency with various considerations on floats, the output will be float rounded to `P=Digits`.

One could envision the `\xinteval` variant to keep 9 fractional digits (it is known the last one may very well be off by 1 unit). But this creates complications of principles.

All of this is very strange because the logarithm clearly shows the deficiencies of the whole idea of floating point arithmetic, logarithm goes from floating point to fixed point, and coercing it into pure floating point has moral costs. Anyway, I shall obide.

```

72 \def\PoorManLogBaseTen{\romannumeral0\poormanlogbaseten}%
73 \def\poormanlogbaseten #1%
74 {%
75   \XINTinfloat[\XINTdigits]%
76   {\romannumeral0\expandafter\PML@logbaseten\romannumeral0\XINTinfloat[9]{#1}}%
77 }%
78 \def\PoorManLogBaseTen_raw%#1
79 {%
80   \romannumeral0\expandafter\PML@logbaseten\romannumeral0\XINTinfloat[9]{#1}%
81 }%
82 \def\PML@logbaseten#1[#2]%
83 {%
84   \xintiiadd{\xintDSx{-9}}{\the\numexpr#2+8\relax}}{\the\numexpr\PML@#1.}[-9]%
85 }%
86 \def\PoorManLog#1%
87 {%
88   \XINTinFloat[\XINTdigits]{\xintMul{\PoorManLogBaseTen_raw{#1}}{23025850923[-10]}}%
89 }%
```

### 29.5.2. `\PoorManPowerOfTen`, `\PoorManExp`

Originally in `poormanlog v0.04`, got transferred into `xintfrac.sty` at 1.3f, then here into `xintlog.sty` at 1.4e.

Produces  $10^x$  with 9 digits of float precision, with an error (believed to be) at most 2 units in the last place, when  $0 < x < 1$ . Of course for this the input must be precise enough to have 9 fractional digits of **fixed point** precision.

Breaking change at 1.4e: output always float-rounded at `P=Digits`.

The 1.3f definition for `\xintExp` (now `\PoorManExp`) was not careful enough (see comments above) for very large exponents. This has been corrected at 1.4e. Formerly `exp(12345678)` produced shameful 6.3095734e5361659 where only the first digit (and exponent...) is correct! Now, with `\xintDigits*:=8;`, `exp(12345678)` will produce 6.7725836e5361659 which is correct rounding to 8 digits. Sorry if your rover expedition to Mars ended in failure due to using my software. I was not expecting anyone to use it so I did back then in 2019 a bit too expeditively the `\xintExp` thing on top of  $10^x$ .

The 1.4e `\PoorManExp` replaces and amends deceased `\xintExp`.

Before using `\xintRound` we screen out the case of zero as `\xintRound` in this case outputs no fractional digits.

```

90 \def\PoorManPowerOfTen{\romannumeral0\poormanpoweroften}%
91 \def\poormanpoweroften #1%
92 {%
93   \expandafter\PML@powoften@out
94   \the\numexpr\expandafter\PML@powoften\romannumeral0\xintraw{#1}%
95 }%
96 \def\PML@powoften@out#1[#2]{\XINTinfloat[\XINTdigits]{#1[#2]}}%
97 \def\PML@powoften#1%
98 {%
99   \xint_UDzerominusfork
100   #1-\PML@powoften@zero
101   0#1\PML@powoften@neg
102   0-\PML@powoften@pos
103   \krof #1%
104 }%
105 \def\PML@powoften@zero 0/1[0]{1\relax/1[0]}%
106 \def\PML@powoften@pos#1[#2]%
107 {%
108   \expandafter\PML@powoften@pos@a\romannumeral0\xintround{9}{#1[#2]}.%
109 }%
110 \def\PML@powoften@pos@a#1.#2.{\PML@Pa#2.\expandafter[\the\numexpr-8+#1]}%
111 \def\PML@powoften@neg#1[#2]%
112 {%
113   \expandafter\PML@powoften@neg@a\romannumeral0\xintround{9}{#1[#2]}.%
114 }%
115 \def\PML@powoften@neg@a#1.#2.%
116 {%
117   \ifnum#2=\xint_c_ \xint_afterfi{1\relax/1[#1]}\else
118   \expandafter\expandafter\expandafter
119   \PML@Pa\expandafter\xint_gobble_i\the\numexpr2000000000-#2.%
120   \expandafter[\the\numexpr-9+#1\expandafter]\fi
121 }%
122 \def\PoorManExp#1{\PoorManPowerOfTen{\xintMul{#1}{43429448190325182765[-20]}}}%

```

### 29.5.3. Removed: `\PoorManPower`, see `\XINTinFloatSciPow`

Removed at 1.4e. See `\XINTinFloatSciPow`.

## 29.6. Macro support for powers

### 29.6.1. `\XINTinFloatSciPow`

This is the new name and extension of `\XINTinFloatPowerH` which was a non user-documented macro used for  $a^b$  previously, and previously was located in `xintfrac`.

A check is done whether the exponent is integer or half-integer, and if positive, the legacy `\xintFloatPower`/`\xintFloatSqrt` macros are used. The rationale is that:

- they give faster evaluations for integer exponent  $b < 10000$  (and beyond)
- they operate at any value of Digits
- they keep accuracy even with gigantic exponents, whereas the `pow10()/log10()` path starts losing accuracy for  $b$  about  $1e8$ . In fact at 1.4e it was even for  $b$  about 1000, as `log10(A)` was not computed with enough fractional digits, except for  $0.8 < A < 1.26$  (roughly), for this usage. At the 1.4f bugfix we compute `log10(A)` with enough accuracy for  $A^b$  to be safe with  $b$  as large as  $1e7$ , and show visible degradation only for  $b$  about  $1e9$ .

The user documentation of `\xintFloatPower` mentions a 0.52 ulp(Z) error where Z is the computed result, which seems not as good as the kind of accuracy we target for `pow10()` (for  $-1 < x < 1$ ) and `log10()` (for  $1 < x < 10$ ) which is more like about 0.505ulp. Perhaps in future I will examine if I need to increase a bit the theoretical accuracy of `\xintFloatPower` but at time of 1.4e/1.4f release I have left it standing as is.

The check whether exponent is integer or half-integer is not on the value but on the representation. Even in `\xintfloatexpr`, input such  $10^{\frac{4}{2}} \text{relax}$  is possible, and  $\frac{4}{2}$  will not be recognized as integer to avoid costly overhead.  $\frac{3}{2}$  will not be recognized as half-integer. But 2.0 will be recognized as integer,  $25e^{-1}$  as half-integer.

In the computation of  $A^b$ , A will be float-rounded to Digits, but the exponent b will be handled "as is" until last minute. Recall that the `\xintfloatexpr` parser does not automatically float round isolated inputs, this happens only once involved in computations.

In the `Digits ≤ 8` branch we do the same as for `Digits > 8` since 1.4f. At 1.4e I had strangely chosen (for "speed", but that was anyhow questionable for integer exponents less than 10 for example) to always use `log10()/pow10()`... But with only 9 fractional digits for the logarithms, exponents such as 1000 naturally led to last 2 or 3 digits being wrong and let's not even mention when the exponent was of the order or  $1e6$ ... now  $A^{1000}$  and  $A^{1000.5}$  are accurately computed and one can handle  $a^{1000.1}$  as  $a^{1000} * a^{0.1}$ .

I wrote the code during 1.4e to 1.4f transition for doing this split of exponent automatically, but it induced a very significant time penalty down the line for fractional exponents, whereas currently  $a^b$  is computed at `Digits=8` with perfectly acceptable accuracy for fractional  $abs(b) < 10$ , and at high speed, and accuracy for big exponents can be obtained by manually splitting as above (although the above has no user interface for keeping each contribution with its extra digits; a single one for  $a^h$ ,  $-1 < h < 1$ ).

```

123 \def\XINTinFloatSciPow{\romannumeral0\XINTinfloatscipow}%
124 \def\XINTinfloatscipow#1#2%
125 {%
126   \expandafter\XINT_scipow_a\romannumeral0\xintrez{#2}\XINT_scipow_int{#1}%
127 }%
128 \def\XINT_scipow_a #1%
129 {%
130   \xint_gob_til_zero#1\XINT_scipow_Biszero0\XINT_scipow_b#1%
131 }%
132 \def\XINT_scipow_Biszero#1#2#3{ 1[0]}%
133 \def\XINT_scipow_b #1#2/#3[#4]#5%
134 {%
135   \unless\if1\XINT_is_One#3XY\xint_dothis\XINT_scipow_c\fi
136   \ifnum#4<\xint_c_mone\xint_dothis\XINT_scipow_c\fi

```



```

137 \ifnum#4=\xint_c_mone
138 \if5\xintLDg{#1#2} %
139 \xint_afterfi{\xint_dothis\xINT_scipow_halfint}\else
140 \xint_afterfi{\xint_dothis\xINT_scipow_c}%
141 \fi
142 \fi
143 \xint_orthat#5#1#2/#3[#4]%
144 }%
145 \def\xINT_scipow_int #1/1[#2]#3%
146 {%
147 \expandafter\xINT_flpower_checkB_a
148 \romannumeral0\xINT_dsx_addzeros{#2}#1;.\XINTdigits.{#3}{\XINTinfloatS[\XINTdigits]}%
149 }%

```

The `\XINT_flpowerh_finish` is the sole remnant of `\XINTinFloatPowerH` which was formerly stitched to `\xintFloatPower` and checked for half-integer exponent.

```

150 \def\xINT_scipow_halfint#1/1[#2]#3%
151 {%
152 \expandafter\xINT_flpower_checkB_a
153 \romannumeral0\xintdsr{\xintDouble{#1}}.\XINTdigits.{#3}\XINT_flpowerh_finish
154 }%
155 \def\xINT_flpowerh_finish #1%
156 {%
157 \XINTinfloatS[\XINTdigits]{\XINTinFloatSqrt[\XINTdigits+\xint_c_iii]{#1}}%
158 }%
159 \def\xINT_tmpa#1.{%
160 \def\xINT_scipow_c ##1[##2]##3%
161 {%
162 \expandafter\xINT_scipow_d\romannumeral0\xINTinfloatS[#1]{##3}\xint:##1[##2]\xint:
163 }%
164 }\expandafter\xINT_tmpa\the\numexpr\xINTdigits.%
165 \def\xINT_scipow_d #1%
166 {%
167 \xint_UDzerominusfork
168 #1-\XINT_scipow_Aiszero
169 0#1\xINT_scipow_Aisneg
170 0-\XINT_scipow_Aispos
171 \krof #1%
172 }%
173 \def\xINT_scipow_Aiszero #1\xint:#2#3\xint:
174 {%

```

Missing NaN and Infinity causes problems. Inserting something like `1["7FFF8000]` is risky as certain macros convert `[N]` into `N` zeros... so the run can appear to stall and will crash possibly badly if we do that. There is some usage in relation to `ilog10` in `xint.sty` and `xintfrac.sty` of `"7FFF8000` but here I will stay prudent and insert the usual `0` value (changed at 1.4g)

```

175 \if-#2\xint_dothis
176 {\XINT_signalcondition{InvalidOperation}{0 raised to power #2#3.}{\ 0[0]}}\fi
177 \xint_orthat{ 0[0]}%
178 }%
179 \def\xINT_scipow_Aispos #1\xint:#2\xint:
180 {%
181 \XINTinfloatpowten{\xintMul{#2}{\XINTinFloatLogTen_xdgout#1}}%
182 }%

```



If  $a^b$  with  $a < 0$ , we arrive here only if  $b$  was not considered to be an integer exponent. So let's raise an error.

```

183 \def\XINT_scipow_Aisneg #1#2\Xint:#3\Xint:
184 {%
185   \XINT_signalcondition{InvalidOperation}%
186   {Fractional power #3 of negative #1#2.}{0[0]}%
187 }%
188 \ifnum\XINTdigits<9

```

At 1.4f we only need for Digits up to 8 to insert usage of `poormanlog` for non integer, non half-integer exponents. At 1.4e the code was more complicated because I had strangely opted for using always the `log10()` path. However we have to be careful to use `\PML@logbaseten` with 9 digits always.

As the legacy macros used for integer and half-integer exponents float-round the input to Digits digits, we must do the same here for coherence. Which induces some small complications here.

```

189 \def\XINT_tmpa#1.#2.#3.{%
190 \def\XINT_scipow_c ##1[##2]##3%
191 {%
192   \expandafter\XINT_scipow_d
193   \romannumeral0\expandafter\XINT_scipow_c_i
194   \romannumeral0\XINTinfloat[#1]{##3}\Xint:##1[##2]\Xint:
195 }%
196 \def\XINT_scipow_c_i##1[##2]{ ##1#3[##2-#2]}%
197 }\expandafter\XINT_tmpa\the\numexpr\XINTdigits\expandafter.%
198 \the\numexpr9-\XINTdigits\expandafter.%
199 \romannumeral\Xintreplicate{9-\XINTdigits}0.%
200 \def\XINT_scipow_Aispos #1\Xint:#2\Xint:
201 {%
202   \poormanpoweroften{\XintMul{#2}{\romannumeral0\expandafter\PML@logbaseten#1}}%
203 }%
204 \fi

```

### 29.6.2. \xintPow

Support macro for  $a^b$  in `\xinteval`. This overloads the original `xintfrac` macro, keeping its original meaning only for integer exponents, which are not too big: for exact evaluation of  $A^b$ , we want the output to not have more than about 10000 digits (separately for numerator and denominator). For this we limit  $b$  depending on the length of  $A$ , simply we want  $b$  to be smaller than the rounded value of 10000 divided by the length of  $A$ . For one-digit  $A$ , this would give 10000 as maximal exponent but due to organization of code related to avoid arithmetic overflow (we can't immediately operate in `\numexpr` with  $b$  as it is authorized to be beyond TeX bound), the maximal exponent is 9999.

The criterion, which guarantees output (numerator and denominator separately) does not exceed by much 10000 digits if at all is that the exponent should be less than the (rounded in the sense of `\numexpr`) quotient of 10000 by the number of digits of  $a$  (considering separately numerator and denominator).

The decision whether to compute  $A^b$  exactly depends on the length of internal representation of  $A$ . So  $9^{9999}$  is evaluated exactly (in `\xinteval`) but for  $9.0$  it is  $9.0^{5000}$  the maximal power. This may change in future.

1.4e had the following bug (for  $\text{Digits} > 8$ ): big integer exponents used the `log10()/pow10()` based approach rather than the legacy macro path which goes via `\xintFloatPower`, as done by `\xintfloateval`! As a result powers with very large integer exponents were more precise in `\xintfloateval` than in `\xinteval`!

1.4f fixes this. Also, it handles  $\text{Digits} \leq 8$  as  $\text{Digits} > 8$ , bringing much simplification here.

```

205 \def\xintPow{\romannumeral0\xintpow}%
206 \def\xintpow#1#2%
207 {%
208   \expandafter\XINT_scipow_a\romannumeral0\xintrez{#2}\XINT_pow_int{#1}%
209 }%

```

In case of half-integer exponent the `\XINT_scipow_a` will have triggered usage of the (new incarnation) of `\XINTinFloatPowerH` which combines `\xintFloatPower` and square root extraction. So we only have to handle here the case of integer exponents which will trigger execution of this `\XINT_pow_int` macro passed as parameter to `\xintpow`.

```

210 \def\XINT_pow_int #1/1[#2]%
211 {%
212   \expandafter\XINT_pow_int_a\romannumeral0\XINT_dsx_addzeros{#2}#1;.%
213 }%

```

1.4e had a bug here for integer exponents  $\geq 10000$ : they triggered going back to the floating point routine but at a late location where the `log10()/pow10()` approach is used.

```

214 \def\XINT_pow_int_a #1#2.%
215 {%
216   \ifnum\if-#1\xintLength{#2}\else\xintLength{#1#2}\fi>\xint_c_iv
217     \expandafter\XINT_pow_bigint
218   \else\expandafter\XINT_pow_int_b
219   \fi #1#2.%
220 }%

```

At 1.4f we correctly jump to the appropriate entry point into the `\xintFloatPower` routine of `xintfrac`, in case of a big integer exponent.

```

221 \def\XINT_pow_bigint #1.#2%
222 {%
223   \XINT_flpower_checkB_a#1.\XINTdigits.{#2}\XINTinfloatS[\XINTdigits]]%
224 }%
225 \def\XINT_pow_int_b #1.#2%
226 {%

```

We now check if the output will not be too bulky. We use here (on the a of  $a^b$ ) `\xintraw`, not `\xintrez`, on purpose so that for example  $9.0^{9999}$  is computed in floating point sense but  $9^{9999}$  is computed exactly. However  $9.0^{5000}$  will be computed exactly. And if I used `\xintrez` here `\xinteval{100^2}` would print 10000.0 and `\xinteval{100^3}` would print 1.0e6. Thus situation is complex.

By the way I am happy to see that  $9.0 \cdot 9.0$  in `\xinteval` does print 81.0 but the truth is that internally it does have the more bulky  $8100/1[-2]$  maybe I should make some revision of this, i.e. use rather systematically `\xintREZ` on input rather than `\xintRaw` (note taken on 2021/05/08 at time of doing 1.4f bugfix release).

```

227   \expandafter\XINT_pow_int_c\romannumeral0\xintraw{#2}\xint:#1\xint:
228 }%

```

The `\XINT_fpow_fork` is (quasi top level) entry point we have found into the legacy `\xintPow` routine of `xintfrac`. Its interface is a bit weird, but let's not worry about this now.

```

229 \def\XINT_pow_int_c#1#2/#3[#4]\xint:#5\xint:
230 {%
231   \if0\ifnum\numexpr\xint_c_x^iv/%
232     (\xintLength{#1#2}\if-#1-\xint_c_i\fi)<\XINT_Abs#5 %
233     1\else
234     \ifnum\numexpr\xint_c_x^iv/\xintLength{#3}<\XINT_Abs#5 %
235     1\else

```

```

236 0\fi\fi
237 \expandafter\XINT_fpow_fork\else\expandafter\XINT_pow_bigint_i
238 \fi
239 #5\Z{#4}{#1#2}{#3}%
240 }%

\XINT_pow_bigint_i is like \XINT_pow_bigint but has its parameters organized differently.
241 \def\XINT_pow_bigint_i#1\Z#2#3#4%
242 {%
243 \XINT_flpower_checkB_a#1.\XINTdigits.{#3/#4[#2]}\XINTinfloatS[\XINTdigits]}%
244 }%

```

## 29.7. Macro support for \xintexpr and \xintfloatexpr syntax

### 29.7.1. The log10() and pow10() functions

Up to 8 digits included we use the poormanlog based ones.

```

245 \ifnum\XINTdigits<9
246 \expandafter\def\csname XINT_expr_func_log10\endcsname#1#2#3%
247 {%
248 \expandafter #1\expandafter #2\expandafter{%
249 \romannumeral`&&\XINT:NEhook:f:one:from:one
250 {\romannumeral`&&\PoorManLogBaseTen#3}}%
251 }%
252 \expandafter\def\csname XINT_expr_func_pow10\endcsname#1#2#3%
253 {%
254 \expandafter #1\expandafter #2\expandafter{%
255 \romannumeral`&&\XINT:NEhook:f:one:from:one
256 {\romannumeral`&&\PoorManPowerOfTen#3}}%
257 }%
258 \else
259 \expandafter\def\csname XINT_expr_func_log10\endcsname#1#2#3%
260 {%
261 \expandafter #1\expandafter #2\expandafter{%
262 \romannumeral`&&\XINT:NEhook:f:one:from:one
263 {\romannumeral`&&\XINTinFloatLogTen#3}}%
264 }%
265 \expandafter\def\csname XINT_expr_func_pow10\endcsname#1#2#3%
266 {%
267 \expandafter #1\expandafter #2\expandafter{%
268 \romannumeral`&&\XINT:NEhook:f:one:from:one
269 {\romannumeral`&&\XINTinFloatPowTen#3}}%
270 }%
271 \fi
272 \expandafter\let\csname XINT_flexpr_func_log10\endcsname
273 \csname XINT_expr_func_log10\endcsname
274 \expandafter\let\csname XINT_flexpr_func_pow10\endcsname
275 \csname XINT_expr_func_pow10\endcsname

```

### 29.7.2. The log(), exp() functions

```

276 \ifnum\XINTdigits<9
277 \def\XINT_expr_func_log #1#2#3%
278 {%

```

```

279 \expandafter #1\expandafter #2\expandafter{%
280 \romannumeral`&&\XINT:NEhook:f:one:from:one
281 {\romannumeral`&&\PoorManLog#3}}%
282 }%
283 \def\XINT_expr_func_exp #1#2#3%
284 {%
285 \expandafter #1\expandafter #2\expandafter{%
286 \romannumeral`&&\XINT:NEhook:f:one:from:one
287 {\romannumeral`&&\PoorManExp#3}}%
288 }%
289 \let\XINT_flexpr_func_log\XINT_expr_func_log
290 \let\XINT_flexpr_func_exp\XINT_expr_func_exp
291 \else
292 \def\XINT_expr_func_log #1#2#3%
293 {%
294 \expandafter #1\expandafter #2\expandafter{%
295 \romannumeral`&&\XINT:NEhook:f:one:from:one
296 {\romannumeral`&&\XINTinFloatLog#3}}%
297 }%
298 \def\XINT_expr_func_exp #1#2#3%
299 {%
300 \expandafter #1\expandafter #2\expandafter{%
301 \romannumeral`&&\XINT:NEhook:f:one:from:one
302 {\romannumeral`&&\XINTinFloatExp#3}}%
303 }%
304 \let\XINT_flexpr_func_log\XINT_expr_func_log
305 \let\XINT_flexpr_func_exp\XINT_expr_func_exp
306 \fi

```

### 29.7.3. The pow() function

The mapping of `**` and `^` to `\XINTinFloatSciPow` (in `\xintfloatexpr` context) and `\xintPow` (in `\xintexpr` context), is done in `xintexpr`.

```

307 \def\XINT_expr_func_pow #1#2#3%
308 {%
309 \expandafter #1\expandafter #2\expandafter{%
310 \romannumeral`&&\XINT:NEhook:f:one:from:two
311 {\romannumeral`&&\xintPow#3}}%
312 }%
313 \def\XINT_flexpr_func_pow #1#2#3%
314 {%
315 \expandafter #1\expandafter #2\expandafter{%
316 \romannumeral`&&\XINT:NEhook:f:one:from:two
317 {\romannumeral`&&\XINTinFloatSciPow#3}}%
318 }%

```

## 29.8. End of package loading for low Digits

```

319 \ifnum\XINTdigits<9 \expandafter\XINTlogendinput\fi%

```

## 29.9. Stored constants

The constants were obtained from Maple at 80 digits: fractional power of 10, but only one logarithm  $\log(10)$ .

Currently the code whether for exponential or logarithm will not screen out 0 digits and even will do silly multiplication by  $10^0 = 1$  in that case, and we need to store such silly values.

We add the data for the  $10^{-0.i}$  etc... because pre-computing them on the fly significantly adds overhead to the package loading.

The fractional powers of ten with D+5 digits are used to compute `pow10()` function, those with D+10 digits are used to compute `log10()` function. This is done with an elevated precision for two reasons:

- handling of inputs near 1,
- in order for  $a^b = \text{pow10}(b \cdot \log10(a))$  to keep accuracy even with large exponents, say in absolute value up to  $1e7$ , degradation beginning to show-up at  $1e8$ .

```

320 \def\XINT_tmpa{1[0]}%
321 \expandafter\let\csname XINT_c_1_0\endcsname\XINT_tmpa
322 \expandafter\let\csname XINT_c_2_0\endcsname\XINT_tmpa
323 \expandafter\let\csname XINT_c_3_0\endcsname\XINT_tmpa
324 \expandafter\let\csname XINT_c_4_0\endcsname\XINT_tmpa
325 \expandafter\let\csname XINT_c_5_0\endcsname\XINT_tmpa
326 \expandafter\let\csname XINT_c_6_0\endcsname\XINT_tmpa
327 \expandafter\let\csname XINT_c_1_0_x\endcsname\XINT_tmpa
328 \expandafter\let\csname XINT_c_2_0_x\endcsname\XINT_tmpa
329 \expandafter\let\csname XINT_c_3_0_x\endcsname\XINT_tmpa
330 \expandafter\let\csname XINT_c_4_0_x\endcsname\XINT_tmpa
331 \expandafter\let\csname XINT_c_5_0_x\endcsname\XINT_tmpa
332 \expandafter\let\csname XINT_c_6_0_x\endcsname\XINT_tmpa
333 \expandafter\let\csname XINT_c_1_0_inv\endcsname\XINT_tmpa
334 \expandafter\let\csname XINT_c_2_0_inv\endcsname\XINT_tmpa
335 \expandafter\let\csname XINT_c_3_0_inv\endcsname\XINT_tmpa
336 \expandafter\let\csname XINT_c_4_0_inv\endcsname\XINT_tmpa
337 \expandafter\let\csname XINT_c_5_0_inv\endcsname\XINT_tmpa
338 \expandafter\let\csname XINT_c_6_0_inv\endcsname\XINT_tmpa
339 \expandafter\let\csname XINT_c_1_0_inv_x\endcsname\XINT_tmpa
340 \expandafter\let\csname XINT_c_2_0_inv_x\endcsname\XINT_tmpa
341 \expandafter\let\csname XINT_c_3_0_inv_x\endcsname\XINT_tmpa
342 \expandafter\let\csname XINT_c_4_0_inv_x\endcsname\XINT_tmpa
343 \expandafter\let\csname XINT_c_5_0_inv_x\endcsname\XINT_tmpa
344 \expandafter\let\csname XINT_c_6_0_inv_x\endcsname\XINT_tmpa
345 \def\XINT_tmpa#1#2#3#4;%
346   {\expandafter\edef\csname XINT_c_#1_#2\endcsname
347     {\XINTinFloat[\XINTdigitsormax+5]{#3#4[-79]}}}%
348   \expandafter\edef\csname XINT_c_#1_#2_x\endcsname
349     {\XINTinFloat[\XINTdigitsormax+10]{#3#4[-79]}}}%
350   }%
351 % 10^0.i
352 \XINT_tmpa 1 1 12589254117941672104239541063958006060936174094669310691079230195266476157825020;%
353 \XINT_tmpa 1 2 15848931924611134852021013733915070132694421338250390683162968123166568636684540;%
354 \XINT_tmpa 1 3 19952623149688796013524553967395355579862743154053460992299136670049309106980490;%
355 \XINT_tmpa 1 4 25118864315095801110850320677993273941585181007824754286798884209082432477235613;%
356 \XINT_tmpa 1 5 31622776601683793319988935444327185337195551393252168268575048527925944386392382;%
357 \XINT_tmpa 1 6 39810717055349725077025230508775204348767703729738044686528414806022485386945804;%
358 \XINT_tmpa 1 7 50118723362727228500155418688494576806047198983281926392969745588901125568883069;%

```

```

359 \XINT_tmpa 1 8 63095734448019324943436013662234386467294525718822872452772952883349494329768681;%
360 \XINT_tmpa 1 9 79432823472428150206591828283638793258896063175548433209232392931695569719148754;%
361 % 10^0.0i
362 \XINT_tmpa 2 1 10232929922807541309662751748198778273411640572379813085994255856738296458625172;%
363 \XINT_tmpa 2 2 10471285480508995334645020315281400790567914715039292120056525299012577641023719;%
364 \XINT_tmpa 2 3 10715193052376064174083022246945087339158659633422172707894501914136771607653870;%
365 \XINT_tmpa 2 4 10964781961431850131437136061411270464271158762483023169080841607885740984711300;%
366 \XINT_tmpa 2 5 11220184543019634355910389464779057367223085073605529624450744481701033026862244;%
367 \XINT_tmpa 2 6 11481536214968827515462246116628360182562102373996119340874991068894793593040890;%
368 \XINT_tmpa 2 7 1174897554939529541722067765126844227813431797179312479195387580500791285226246;%
369 \XINT_tmpa 2 8 12022644346174129058326127151935204486942664354881189151104892745683155052368222;%
370 \XINT_tmpa 2 9 12302687708123815342415404364750907389955639574572144413097319170011637639124482;%
371 % 10^0.00i
372 \XINT_tmpa 3 1 10023052380778996719154048893281105540536684535421606464116348523047431367720401;%
373 \XINT_tmpa 3 2 10046157902783951424046519858132787392010166060319618489538315083825599423438638;%
374 \XINT_tmpa 3 3 10069316688518041699296607872661381368099438247964820601930206419324524707606686;%
375 \XINT_tmpa 3 4 10092528860766844119155277641202580844111492027373621434478800545314309618714957;%
376 \XINT_tmpa 3 5 10115794542598985244409323144543146957419235215102899054703546688078254946034250;%
377 \XINT_tmpa 3 6 10139113857366794119988279023017296985954042032867436525450889437280417044987125;%
378 \XINT_tmpa 3 7 10162486928706956276733661150135543062420167220622552197768982666050994284378619;%
379 \XINT_tmpa 3 8 10185913880541169240797988673338257820431768224957171297560936579346433061037662;%
380 \XINT_tmpa 3 9 10209394837076799554149033101487543990018213667630072574873723356334069913329713;%
381 % 10^0.000i
382 \XINT_tmpa 4 1 10002302850208247526835942556719413318678216124626534526963475845228205382579041;%
383 \XINT_tmpa 4 2 10004606230728403216239656646745503559081482371024284871882409614422496765669196;%
384 \XINT_tmpa 4 3 10006910141682589957025973521996241909035914023642264228577379693841345823180462;%
385 \XINT_tmpa 4 4 10009214583192958761081718336761022426385537997384755843291864010938378093197023;%
386 \XINT_tmpa 4 5 10011519555381688769842032367472488618040778885656970999331288116685029387850446;%
387 \XINT_tmpa 4 6 10013825058370987260768186632475607982636715641432550952229573271596547716373358;%
388 \XINT_tmpa 4 7 10016131092283089653826887255241073941084503769368844606021481400409002185558343;%
389 \XINT_tmpa 4 8 10018437657240259517971072914549205297136779497498835020699531587537662833033174;%
390 \XINT_tmpa 4 9 10020744753364788577622204725249622301332888222801030351604197113557132455165040;%
391 % 10^0.0000i
392 \XINT_tmpa 5 1 10000230261160268806710649793464495797824846841503180050673957122443571394978721;%
393 \XINT_tmpa 5 2 10000460527622557806255008596155855743730116854295068547616656160734125748005947;%
394 \XINT_tmpa 5 3 10000690799386989083565213461287219981856579552059660369243804541364501659468630;%
395 \XINT_tmpa 5 4 10000921076453684726384543254593368743049141124080210677706489564626675960578367;%
396 \XINT_tmpa 5 5 10001151358822766825267483384008265483772370538793312970508590203623535763866465;%
397 \XINT_tmpa 5 6 10001381646494357473579790530833073090516914490540536234536867917078761046656260;%
398 \XINT_tmpa 5 7 10001611939468578767498557382394677469502542123237272447312733350028467607076918;%
399 \XINT_tmpa 5 8 10001842237745552806012277366194752842273812293689190856411757410911882303011468;%
400 \XINT_tmpa 5 9 10002072541325401690920909385549403068574626162727745910217443397959031898734024;%
401 % 10^0.00000i
402 \XINT_tmpa 6 1 10000023025877439451356029805459000097926504781151663770980171880313737943886754;%
403 \XINT_tmpa 6 2 10000046051807898005897723104514851394069452605882077809669546315010724085277647;%
404 \XINT_tmpa 6 3 10000069077791375785706217087438809625967243923218032821061587553353589726808164;%
405 \XINT_tmpa 6 4 10000092103827872912862930047032391734439796534302560512742030066798473305401477;%
406 \XINT_tmpa 6 5 10000115129917389509449561379274639104559958866285946533811801963402821672829477;%
407 \XINT_tmpa 6 6 10000138156059925697548091583969382297005329013199894805417325991907389143667949;%
408 \XINT_tmpa 6 7 10000161182255481599240782265392507269793911275470978276390154932321984777772469;%
409 \XINT_tmpa 6 8 10000184208504057336610176132939223090407041937631374389422968832433217547184883;%
410 \XINT_tmpa 6 9 10000207234805653031739097001771331138303016031686764989867510425362339583809842;%

```



```

411 \def\XINT_tmpa#1#2#3#4;%
412   {\expandafter\edef
413     \csname XINT_c_#1_#2_inv\endcsname{\XINTinFloat[\XINTdigitsormax+5]{#3#4[-80]}}}%
414   \expandafter\edef
415     \csname XINT_c_#1_#2_inv_x\endcsname{\XINTinFloat[\XINTdigitsormax+10]{#3#4[-80]}}}%
416   }%
417 % 10^-0.i
418 \XINT_tmpa 1 1 79432823472428150206591828283638793258896063175548433209232392931695569719148754;%
419 \XINT_tmpa 1 2 63095734448019324943436013662234386467294525718822872452772952883349494329768681;%
420 \XINT_tmpa 1 3 50118723362727228500155418688494576806047198983281926392969745588901125568883069;%
421 \XINT_tmpa 1 4 39810717055349725077025230508775204348767703729738044686528414806022485386945804;%
422 \XINT_tmpa 1 5 31622776601683793319988935444327185337195551393252168268575048527925944386392382;%
423 \XINT_tmpa 1 6 25118864315095801110850320677993273941585181007824754286798884209082432477235613;%
424 \XINT_tmpa 1 7 1995262314968879601352455396739535579862743154053460992299136670049309106980490;%
425 \XINT_tmpa 1 8 15848931924611134852021013733915070132694421338250390683162968123166568636684540;%
426 \XINT_tmpa 1 9 12589254117941672104239541063958006060936174094669310691079230195266476157825020;%
427 % 10^-0.0i
428 \XINT_tmpa 2 1 97723722095581068269707600696156123863427170069897801526639004097175507042084888;%
429 \XINT_tmpa 2 2 95499258602143594972395937950148401513087269708053320302465127242741421479104601;%
430 \XINT_tmpa 2 3 93325430079699104353209661168364840720225485199736026149257155811788093771138272;%
431 \XINT_tmpa 2 4 91201083935590974212095940791872333509323858755696109214760361851771695487999100;%
432 \XINT_tmpa 2 5 89125093813374552995310868107829696398587478293004836994794349506746891059190135;%
433 \XINT_tmpa 2 6 87096358995608063751082742520877054774747128501284704090761796673224328569285177;%
434 \XINT_tmpa 2 7 85113803820237646781712631859248682794521725442067093899553745086385146367436049;%
435 \XINT_tmpa 2 8 83176377110267100616669140273840405263880767161887438462740286611379995442629360;%
436 \XINT_tmpa 2 9 81283051616409924654127879773132980187568851100062454636602325121954484722491710;%
437 % 10^-0.00i
438 \XINT_tmpa 3 1 99770006382255331719442194285376231055211861394573154624878230890945476532432225;%
439 \XINT_tmpa 3 2 99540541735152696244806147089510943107144177264574823668081299845609359857038344;%
440 \XINT_tmpa 3 3 99311604842093377157642607688515474663519162181123336122073822476734517364853150;%
441 \XINT_tmpa 3 4 99083194489276757440828314388392035249938006860819409201135652190410238171119287;%
442 \XINT_tmpa 3 5 98855309465693884028524792978202683686410726723055209558576898759166522286083202;%
443 \XINT_tmpa 3 6 98627948563121047157261523093421290951784086730437722805070296627452491731402556;%
444 \XINT_tmpa 3 7 98401110576113374484101831088824192144756194053451911515003663381199842081528019;%
445 \XINT_tmpa 3 8 98174794301998439937928161622872240632362817134775142288598128693131032909278350;%
446 \XINT_tmpa 3 9 97948998540869887269961493687844910565420716785032030061251916654655049965062649;%
447 % 10^-0.000i
448 \XINT_tmpa 4 1 99976976799815658635141604638981297541396466984477711459083930684685186989697929;%
449 \XINT_tmpa 4 2 99953958900308784552845777251512089759003230012954649234748668826546533498169555;%
450 \XINT_tmpa 4 3 9993094630025899216869377702512591351888960684418033717545524043693899420866954;%
451 \XINT_tmpa 4 4 99907938998446176870082987427724649318531547584410414997787083472394558389284098;%
452 \XINT_tmpa 4 5 99884936993650514951538205746462968844845952521633937925370747725933629958238429;%
453 \XINT_tmpa 4 6 99861940284652463550037839584112909891259691850983307437097305856727153967481065;%
454 \XINT_tmpa 4 7 99838948870232760580354983175435314251655958968480344701699631967048474751069525;%
455 \XINT_tmpa 4 8 99815962749172424670413384320528274471550942114263604264788586703624513163664479;%
456 \XINT_tmpa 4 9 99792981920252755096658293766085025870392854106037465990011216356523334125368417;%
457 % 10^-0.0000i
458 \XINT_tmpa 5 1 99997697441416293040019992468837639003787989306240470048763511538639048400765328;%
459 \XINT_tmpa 5 2 99995394935850346394065999228750187791584034668237852053859761641089829514536011;%
460 \XINT_tmpa 5 3 99993092483300939297147020491645017932348508508297743745039515152378182676736684;%
461 \XINT_tmpa 5 4 99990790083766851012380885556584619169980753943113396677545915245611923361705686;%
462 \XINT_tmpa 5 5 99988487737246860830993605587529673614422529030613405900998412734419982883669223;%

```

```

463 \XINT_tmpa 5 6 99986185443739748072318726405984801565268578044798475766025647187221659622450651;%
464 \XINT_tmpa 5 7 99983883203244292083796681298546635825139453823571398432959235283529730820181019;%
465 \XINT_tmpa 5 8 99981581015759272240974143839353881367972777961073357987943600347058023396510672;%
466 \XINT_tmpa 5 9 99979278881283467947503380727439017235290006415950636109257677645557027950744160;%
467 % 10^-0.00000i
468 \XINT_tmpa 6 1 99999769741755795297487775997495948154386159348543852707438213487494386559762090;%
469 \XINT_tmpa 6 2 99999539484041779185217876175552674518572114763104546143049036309870762496098218;%
470 \XINT_tmpa 6 3 99999309226857950442387361668529812394860404492721699528707852590634886516924591;%
471 \XINT_tmpa 6 4 99999078970204307848196104610199226516866442484686906173860803560254163287393673;%
472 \XINT_tmpa 6 5 99998848714080850181846788127272455158309917012010320554498356105168896062430977;%
473 \XINT_tmpa 6 6 99998618458487576222544906332928167145404344730731751204389698696345970645201375;%
474 \XINT_tmpa 6 7 99998388203424484749498764320339633772810463403640242228131015918494067456365331;%
475 \XINT_tmpa 6 8 99998157948891574541919478156202215623119146605983303201215215949834619332550929;%
476 \XINT_tmpa 6 9 9999792769488844379020974874260864289829523807763942234420930258187873904191138;%
477 % log(10)
478 \edef\XINT_c_logten
479 {\XINTinFloat[\XINTdigitsormax+4]
480 {23025850929940456840179914546843642076011014886287729760333279009675726096773525[-79]}}%
481 \edef\XINT_c_oneoverlogten
482 {\XINTinFloat[\XINTdigitsormax+4]
483 {43429448190325182765112891891660508229439700580366656611445378316586464920887077[-80]}}%
484 \edef\XINT_c_oneoverlogten_xx
485 {\XINTinFloat[\XINTdigitsormax+14]
486 {43429448190325182765112891891660508229439700580366656611445378316586464920887077[-80]}}%

```

## 29.10. April 2021: at last, \XINTinFloatPowTen, \XINTinFloatExp

Done April 2021. I have procrastinated (or did not have time to devote to this) at least 5 years, even more.

Speed improvements will have to wait to long delayed refactoring of core floating point support which is still in the 2013 primitive state !

I did not try to optimize for say 16 digits, as I was more focused on reaching 60 digits in a reasonably efficient manner (trigonometric functions achieved this since 2019) in the same coding framework. Finally, up to 62 digits.

The stored constants are  $\log(10)$  at P+4 digits and the powers  $10^{0.d}$ ,  $10^{0.0d}$ , etc, up to  $10^{0.00000d}$  for  $d=1..9$ , as well as their inverses, at P+5 and P+10 digits. The constants were obtained from Maple at 80 digits.

Initially I constructed the exponential series  $\exp(h)$  as one big unique nested macro. It contained pre-rounded values of the  $1/i!$  but would float-round  $h$  to various numbers of digits, with always the full initial  $h$  as input.

After having experimented with the logarithm, I redid  $\exp(h) = 1 + h(1 + h(1/2 + \dots))$  with many macros in order to have more readable code, and to dynamically cut-off more and more digits from  $h$  the deeper it is used. See the logarithm code for (perhaps) more comments.

The thresholds have been obtained from considerations including an  $h_{\max}$  (a bit more than  $0.5 \log(10) 10^{-6}$ ). Here is the table:

- maximal value of P: 8, 15, 21, 28, 35, 42, 48, 55, 62
- last included term: /1, /2, /6, /4!, /5!, /6!, /7!, /8!, /9!

Computations are done morally targeting P+4 fractional fixed point digits, with a stopping criteria at say about  $5e(-P-4)$ , which was used for the table above using only the worst case. As the used macros are a mix of exact operations and floating point reductions this is in practice a bit different. The  $h$  will be initially float rounded to P-1 digits. It is cut-off more and more, the deeper nested it is used.



The code for this evaluation of  $10^x$  is very poor with  $x$  very near zero: it does silly multiplication by 1, and uses more terms of exponential series than would then be necessary.

For the computation of  $\exp(x)$  as  $10^{(c*x)}$  with  $c=\log(10)^{-1}$ , we need more precise  $c$  the larger  $\text{abs}(x)$  is. For  $\text{abs}(x)<1$  (or 2), the  $c$  with  $P+4$  fractional digits is sufficient. But decimal exponents are more or less allowed to be near the TeX maximum  $2^{31}-1$ , which means that  $\text{abs}(x)$  could be as big as  $0.5e10$ , and we then need  $c$  with  $P+14$  digits to cover that range.

I am hesitating whether to first examine integral part of  $\text{abs}(x)$  and for example to use  $c$  with either  $P+4$ ,  $P+9$  or  $P+14$  digits, and also take this opportunity to inject an error message if  $x$  is too big before TeX arithmetic overflow happens later on. For time being I will use overhead of `oneoverlogten` having ample enough digits...

The exponent received as input is float rounded to  $P + 14$  digits. In practice the input will be already a  $P$ -digits float. The motivation here is for low Digits situation: but this done so that for example with `Digits=4`, we want  $\exp(12345)$  not to be evaluated as  $\exp(12350)$  which would have no meaning at all. The  $+14$  is because we have prepared `1/log(10)` with that many significant digits. This conundrum is due to the inadequation of the world of floating point numbers with `exp()` and `log()`: clearly `exp()` goes from fixed point to floating point and `log()` goes from floating point to fixed point, and coercing them to work inside the sole floating point domain is not mathematically natural. Although admittedly it does create interesting mathematical questions! A similar situation applies to functions such as `cos()` and `sin()`, what sense is there in the expression `cos(exp(50))` for example with 16 digits precision? My opinion is that it does not make ANY sense. Anyway, I shall obide.

As `\XINTinFloatS` will not add unnecessarily trailing zeros, the `\XINTdigits+14` is not really an enormous overhead for integer exponents, such as in the example above the 12345, or more realistically small integer exponents, and if the input is already float rounded to  $P$  digits, the overhead is also not enormous (float-rounding is costly when the input is a fraction).

`\XINTinfloatpowten` will receive an input with at least  $P+14$  and up to  $2P+28$  digits... fortunately with no fraction part and will start rounding it in the fixed point sense of its input to  $P+4$  digits after decimal point, which is not enormously costly.

Of course all these things pile up...

```

487 \def\XINTinFloatExp{\romannumeral0\XINTinfloatexp}%
488 \def\XINT_tmpa#1.{%
489 \def\XINTinfloatexp##1%
490 {%
491   \XINTinfloatpowten
492   {\xintMul{\XINT_c_oneoverlogten_xx}{\XINTinFloatS[#1]{##1}}}%
493 }%
494 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+14.%

```

Here is how the reduction to computations of an  $\exp(h)$  via series is done.

Starting from  $x$ , after initial argument normalization, it is fixed-point rounded to 6 fractional digits giving  $x' = \text{sn.d}_1\dots\text{d}_6$  (which may be 0).

I have to resist temptation using very low level routines here and wisely will employ the available user-level stuff. One computes then the difference  $x-x'$  which gives some  $\text{eta}$ , and the  $h$  will be  $\log(10).\text{eta}$ . The subtraction and multiplication are done exactly then float rounded to  $P-1$  digits to obtain the  $h$ .

Then  $\exp(h)$  is computed. And to finish it is multiplied with the stored  $10^{\text{sn.d}_1}$ ,  $10^{\text{sn.d}_2}$ , etc..., constants and its decimal exponent is increased by  $\text{sn}$ . These operations are done at  $P+5$  floating point digits. The final result is then float-rounded to the target  $P$  digits.

Currently I may use nested macros for some operations but will perhaps revise in future (it makes tracing very complicated if one does not have intermediate macros). The exponential series itself was initially only one single macro, but as commented above I have now modified it.

```

495 \def\XINTinFloatPowTen{\romannumeral0\XINTinfloatpowten}%
496 \def\XINT_tmpa#1.{%

```

```

497 \def\XINTinfloatpowten##1%
498 {%
499     \expandafter\XINT_powten_fork
500     \romannumeral0\xintiround{#1}{##1}[-#1]%
501 }%
502 \expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+4.%
503 \def\XINT_powten_fork#1%
504 {%
505     \xint_UDzerominusfork
506     #1-\XINT_powten_zero
507     0#1\XINT_powten_neg
508     0-\XINT_powten_pos
509     \krof #1%
510 }%
511 \def\XINT_powten_zero #1[#2]{ 1[0]}%
    This rounding may produce 0.000000 but will always have 6 exactly fractional digits, because the
    special case of a zero input was filtered out preventively.
512 \def\XINT_powten_pos#1[#2]%
513 {%
514     \expandafter\XINT_powten_pos_a\romannumeral0\xintround{6}{#1[#2]}#1[#2]%
515 }%
516 \def\XINT_tmpa #1.#2.#3.%
517 \def\XINT_powten_pos_a ##1.##2##3##4##5##6##7##8[##9]%
518 {%
519     \expandafter\XINT_infloate
520     \romannumeral0\XINTinfloat[#3]{%
521         \xintMul{\csname XINT_c_1_##2\endcsname}{%
522             \XINTinFloat[#1]{%
523                 \xintMul{\csname XINT_c_2_##3\endcsname}{%
524                     \XINTinFloat[#1]{%
525                         \xintMul{\csname XINT_c_3_##4\endcsname}{%
526                             \XINTinFloat[#1]{%
527                                 \xintMul{\csname XINT_c_4_##5\endcsname}{%
528                                     \XINTinFloat[#1]{%
529                                         \xintMul{\csname XINT_c_5_##6\endcsname}{%
530                                             \XINTinFloat[#1]{%
531                                                 \xintMul{\csname XINT_c_6_##7\endcsname}{%
532                                                     \xintAdd{1[0]}{%
533                 \expandafter\XINT_Exp_series_a_ii
534                 \romannumeral0\XINTinfloat[#2]{%
535                     \xintMul{\XINT_c_logten}%
536                         {\xintAdd{-##1.##2##3##4##5##6##7}{##8[##9]}}%
537                     }%
538                 \xint:
539                 }%
540                 }}}}}}}}}}}{##1}%
541 }}\expandafter\XINT_tmpa
542     \the\numexpr\XINTdigitsormax+5\expandafter.%
543     \the\numexpr\XINTdigitsormax-1\expandafter.%
544     \the\numexpr\XINTdigitsormax.%

```

This rounding may produce -0.000000 but will always have 6 exactly fractional digits and a leading minus sign.

```

545 \def\XINT_powten_neg#1[#2]%
546 {%
547   \expandafter\XINT_powten_neg_a\romannumeral0\xintround{6}{#1[#2]}#1[#2]%
548 }%
549 \def\XINT_tmpa #1.#2.#3.{%
550 \def\XINT_powten_neg_a -##1.##2##3##4##5##6##7##8[##9]%
551 {%
552   \expandafter\XINT_infloate
553   \romannumeral0\XINTinfloat[#3]{%
554     \xintMul{\csname XINT_c_1_##2_inv\endcsname}{%
555       \XINTinFloat[#1]{%
556         \xintMul{\csname XINT_c_2_##3_inv\endcsname}{%
557           \XINTinFloat[#1]{%
558             \xintMul{\csname XINT_c_3_##4_inv\endcsname}{%
559               \XINTinFloat[#1]{%
560                 \xintMul{\csname XINT_c_4_##5_inv\endcsname}{%
561                   \XINTinFloat[#1]{%
562                     \xintMul{\csname XINT_c_5_##6_inv\endcsname}{%
563                       \XINTinFloat[#1]{%
564                         \xintMul{\csname XINT_c_6_##7_inv\endcsname}{%
565                           \xintAdd{1[0]}{%
566                             \expandafter\XINT_Exp_series_a_ii
567                             \romannumeral0\XINTinfloat[#2]{%
568                               \xintMul{\XINT_c_logten}%
569                               {\xintAdd{##1.##2##3##4##5##6##7}{##8[##9]}}%
570                             }%
571                             \xint:
572                           }%
573                       }}}}]]}}}}}}{-##1}%
574 }}\expandafter\XINT_tmpa
575 \the\numexpr\XINTdigitsormax+5\expandafter.%
576 \the\numexpr\XINTdigitsormax-1\expandafter.%
577 \the\numexpr\XINTdigitsormax.%

```

### 29.10.1. Exponential series

Or rather here  $h(1 + h(1/2 + h(1/6 + \dots)))$ . Upto at most  $h^9/9!$  term.

The used initial  $h$  has been float rounded to  $P-1$  digits.

```

578 \def\XINT_tmpa#1.#2.{%
579 \def\XINT_Exp_series_a_ii##1\xint:
580 {%
581   \expandafter\XINT_Exp_series_b
582   \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
583 }%
584 \def\XINT_Exp_series_b##1[##2]\xint:
585 {%
586   \expandafter\XINT_Exp_series_c_
587   \romannumeral0\xintadd{1}{\xintHalf{##10}[##2-1]}\xint:
588 }%
589 \def\XINT_Exp_series_c_##1\xint:##2\xint:
590 {%
591   \XINTinFloat[#2]{\xintMul{##1}{##2}}%
592 }%

```

```

593 }%
594 \expandafter\XINT_tmpa
595         \the\numexpr\XINTdigitsormax-6\expandafter.%
596         \the\numexpr\XINTdigitsormax-1.%
597 \ifnum\XINTdigits>15
598 \def\XINT_tmpa#1.#2.#3.#4.{%
599 \def\XINT_Exp_series_a_ii##1\xint:
600 {%
601     \expandafter\XINT_Exp_series_a_iii
602     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
603 }%
604 \def\XINT_Exp_series_a_iii##1\xint:
605 {%
606     \expandafter\XINT_Exp_series_b
607     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
608 }%
609 \def\XINT_Exp_series_b##1[##2]\xint:
610 {%
611     \expandafter\XINT_Exp_series_c_i
612     \romannumeral0\xintadd{#3}{##1/6[##2]}\xint:
613 }%
614 \def\XINT_Exp_series_c_i##1\xint:##2\xint:
615 {%
616     \expandafter\XINT_Exp_series_c_
617     \romannumeral0\xintadd{#4}{\XINTinfloatS[#2]{\xintMul{##1}{##2}}}\xint:
618 }%
619 }\expandafter\XINT_tmpa
620 \the\numexpr\XINTdigitsormax-13\expandafter.%
621 \the\numexpr\XINTdigitsormax-6.%
622 {5[-1]}.%
623 {1[0]}.%
624 \fi
625 \ifnum\XINTdigits>21
626 \def\XINT_tmpa#1.#2.#3.#4.{%
627 \def\XINT_Exp_series_a_iii##1\xint:
628 {%
629     \expandafter\XINT_Exp_series_a_iv
630     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
631 }%
632 \def\XINT_Exp_series_a_iv##1\xint:
633 {%
634     \expandafter\XINT_Exp_series_b
635     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
636 }%
637 \def\XINT_Exp_series_b##1[##2]\xint:
638 {%
639     \expandafter\XINT_Exp_series_c_ii
640     \romannumeral0\xintadd{#3}{##1/24[##2]}\xint:
641 }%
642 \def\XINT_Exp_series_c_ii##1\xint:##2\xint:
643 {%
644     \expandafter\XINT_Exp_series_c_i

```

```

645 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
646 }%
647 }\expandafter\XINT_tmpa
648 \the\numexpr\XINTdigitsormax-19\expandafter.%
649 \the\numexpr\XINTdigitsormax-13\expandafter.%
650 \romannumeral0\XINTinfloat[\XINTdigitsormax-13]{1/6[0]}.%
651 {5[-1]}.%
652 \fi
653 \ifnum\XINTdigits>28
654 \def\XINT_tmpa #1 #2 #3 #4 #5 #6 #7 %
655 {%
656 \def\XINT_tmppb ##1##2##3##4%
657 {%
658 \def\XINT_tmppc####1.####2.####3.####4.%
659 {%
660 \def##2#####1\xint:
661 {%
662 \expandafter##1%
663 \romannumeral0\XINTinfloatS[####2]{#####1}\xint:#####1\xint:
664 }%
665 \def##1#####1\xint:
666 {%
667 \expandafter\XINT_Exp_series_b
668 \romannumeral0\XINTinfloatS[####1]{#####1}\xint:#####1\xint:
669 }%
670 \def\XINT_Exp_series_b#####1[#####2]\xint:
671 {%
672 \expandafter##3%
673 \romannumeral0\xintadd{####3}{#####1/#5[#####2]}\xint:
674 }%
675 \def##3#####1\xint:#####2\xint:
676 {%
677 \expandafter##4%
678 \romannumeral0\xintadd{####4}%
679 {\XINTinFloat[####2]{\xintMul{#####1}{#####2}}}\xint:
680 }%
681 }%
682 }%
683 \expandafter\XINT_tmppb
684 \csname XINT_Exp_series_a_\romannumeral\numexpr#1\expandafter\endcsname
685 \csname XINT_Exp_series_a_\romannumeral\numexpr#1-1\expandafter\endcsname
686 \csname XINT_Exp_series_c_\romannumeral\numexpr#1-2\expandafter\endcsname
687 \csname XINT_Exp_series_c_\romannumeral\numexpr#1-3\endcsname
688 \expandafter\XINT_tmppc
689 \the\numexpr\XINTdigitsormax-#2\expandafter.%
690 \the\numexpr\XINTdigitsormax-#3\expandafter.\expanded{%
691 \XINTinFloat[\XINTdigitsormax-#3]{1/#6[0]}.%
692 \XINTinFloat[\XINTdigitsormax-#4]{1/#7[0]}.%
693 }%
694 }%
695 \XINT_tmpa 5 26 19 13 120 24 6 %<-- keep space
696 \ifnum\XINTdigits>35 \XINT_tmpa 6 33 26 19 720 120 24 \fi

```

```

697 \ifnum\XINTdigits>42 \XINT_tmpa 7 40 33 26 5040 720 120 \fi
698 \ifnum\XINTdigits>48 \XINT_tmpa 8 46 40 33 40320 5040 720 \fi
699 \ifnum\XINTdigits>55 \XINT_tmpa 9 53 46 40 362880 40320 5040 \fi
700 \fi

```

## 29.11. April 2021: at last \XINTinFloagLogTen, \XINTinFloatLog

Attention that this is not supposed to be used with `\XINTdigits` at 8 or less, it will crash if that is the case. The `log10()` and `log()` functions in case `\XINTdigits` is at most 8 are mapped to `\PoormanLogBaseTen` respectively `\PoormanLog` macros.

In the explications here I use the function names rather than the macro names.

Both `log(x)` and `log10(x)` are on top of an underlying macro which will produce  $z$  and  $h$  such that  $x$  is about  $10^z e^h$  (with  $h$  being small is obtained via a log series). Then `log(x)` computes `log(10)z+h` whereas `log10(x)` computes as  $z+h/\log(10)$ .

There will be three branches [NO FINALLY ONLY TWO BRANCHES SINCE 1.4f] according to situation of  $x$  relative to 1. Let  $y$  be the math value `log10(x)` that we want to approximate to target precision  $P$  digits.  $P$  is assumed at least 9.

I will describe the algorithm roughly, but skip its underlying support analysis; at some point I mention "fixed point calculations", but in practice it is not done exactly that way, but describing it would be complicated so look at the code which is very readable (by the author, at the present time).

First we compute  $z = \text{\$n.d\_1d\_2...d\_6}$  as the rounded to 6 fractional digits approximation of  $y = \log_{10}(x)$  obtained by first using the `poormanlog` macros on  $x$  (float rounded to 9 digits) then rounding as above.

Warning: this description is not in sync with the code, now the case where `d\_1d\_2...d\_6` is `000000` is filtered out and one jumps directly either to case I if  $n \neq 0$  or to case III if  $n = 0$ . The case when rounding produces a  $z$  equal to zero is also handled especially.

WARNING: at 1.4f, the CASE I was REMOVED. Everything is handled as CASE II or exceptionally case III. Indeed this removal was observed to simply cost about 10% extra time at  $D=16$  digits, which was deemed an acceptable cost. The cost is certainly higher at  $D=9$  but also relatively lower at high  $D$ 's. It means that logarithms are always computed with 9, not 4, safety *\*\*fractional\*\** digits, and this allows to compute powers accurately with exponents say up to  $1e7$ , degradation starting to show at  $1e8$  and for sure at  $1e9$ . However for integer and half-integer exponents the old routine `\xintFloatPower` will still be used, and perhaps it will need some increased precision update as the documented  $0.52\text{ulp}$  error bound is higher than our more stringent standards of 2021.

CASE I: [removed at 1.4f!] either  $n$  is NOT zero or `d\_1d\_2...d\_6` is at least `100001`. Then we compute  $X = 10^{(-z)}x$  which is near 1, by using the table of powers of 10, using  $P+5$  digits significands. Then we compute (exactly)  $\eta = X-1$ , (which is in absolute value less than  $0.0000012$ ) and obtain  $y$  as  $z + \log(10)^{-1}$  times  $\log(1+\eta)$  where  $\log(1+\eta) = \eta - \eta^2/2 + \eta^3/3 - \dots$  is "computed with  $P+4$  fractional fixed point digits" [1]\_ according to the following table:

- maximal value of  $P$ : 9, 15, 21, 27, 33, 39, 45, 51, 57, 63
- last included term:  $/1, /2, /3, /4, /5, /6, /7, /8, /9, /10$

.. [1] this " $P+4$ " includes leading fractional zeroes so in practice it will rather be done as  $\eta(1 - \eta(1/2 + \eta(1/3 - \dots)))$ , and the inner sums will be done in various precisions, the top level (external)  $\eta$  probably at  $P-1$  digits, the first inner  $\eta$  at  $P-7$  digits, the next at  $P-13$ , something in this style. The heuristics is simple: at  $P=9$  we don't need the first inner  $\eta$ , so let's use there  $P-9$  or rather  $P-7$  digits by security. Similarly at  $P=3$  we would not need at all the  $\eta$ , so let's use the top level one rounded at  $P-3+2 = P-1$  digits. And there is a shift by 6 less digits at each inner level. RÉFLÉCHIR SI C'EST PAS PLUTÔT  $P-2$  ICI, suffisant au regard de la précision par ailleurs pour la réduction près de 1.

The sequence of maximal  $P$ 's is simply an arithmetic progression.

The addition of  $z$  will trigger the final rounding to  $P$  digits. The inverse of  $\log(10)$  is pre-computed with  $P+4$  digits.

This case I essentially handles  $x$  such as  $\max(x, 1/x) > 10^{0.1} = 1.2589\dots$

CASE II:  $n$  is zero and  $d_{1d_2} \dots d_6$  is not zero. We operate as in CASE I, up to the following differences:

- the table of fractional powers of 10 is used with  $P+10$  significands.
- the  $X$  is also computed with  $P+10$  digits, i.e.  $\eta = X-1$  (which obeys the given estimate) is estimated with  $P+9$  [2]\_ fractional fixed points digits and the log series will be evaluated in this sense.
- the constant  $\log(10)^{-1}$  is still used with only  $P+4$  digits

The log series is terminated according to the following table:

- maximal value of  $P$ : 4, 10, 16, 22, 28, 34, 40, 46, 52, 58, 64
- last included term:  $/1, /2, /3, /4, /5, /6, /7, /8, /9, /10$

Again the  $P$ 's are in arithmetic progression, the same as before shifted by 5.

.. [2] same remark as above. The top level  $\eta$  in  $\eta(1 - \eta(1/2 - \eta(\dots)))$  will use  $P+4$  significant digits, but the first inner  $\eta$  will be used with only  $P-2$  digits, the next inner one with  $P-8$  digits etc...

This case II handles the  $x$  which are near 1, but not as close as  $10^{\pm 0.000001}$ .

CASE III:  $z=0$ . In this case  $X = x = 1+\eta$  and we use the log series in this sense :  $\log(10)^{-1} * \eta * (1 - \eta/2 + \eta^2/3 - \dots)$  where again  $\log(10)^{-1}$  has been precomputed with  $P+4$  digits and morally the series uses  $P+4$  fractional digits ( $P+3$  would probably be enough for the precision I want, need to check my notes) and the thresholds table is:

- maximal value of  $P$ : 3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63
- last included term:  $/1, /2, /3, /4, /5, /6, /7, /8, /9, /10, /11$

This is same progression but shifted by one.

To summarize some relevant aspects:

- this algorithm uses only  $\log(10)^{-1}$  as precomputed logarithm
- in particular the logarithms of small integers 2, 3, 5, ... are not pre-computed. Added note: I have now tested at 16, 32, 48 and 62 digits that all of the  $\log_{10}(n)$ , for  $n = 1..1000$ , are computed with correct rounding. In fact, generally speaking, random testing of a about 20000 inputs has failed to reveal a single non-correct rounding. Naturally, randomly testing is not the way to corner the software into its weak points...

- it uses two tables of fractional powers of ten: one with  $P+5$  digits and another one with extended precision at  $P+10$  digits.

- it needs three distinct implementations of the log series.

- it does not use the well-known trick reducing to using only odd powers in the log series (somehow I have come to dread divisions, even though here as is well-known it could be replaced with some product, my impression was that what is gained on one side is lost on the other, for the range of  $P$  I am targeting, i.e.  $P$  up to about 60.)

- all of this is experimental (in particular the previous item was not done perhaps out of sheer laziness)

Absolutely no error check is done whether the input  $x$  is really positive. As seen above the maximal target precision is 63 (not 64).

Update for 1.4f: when the logarithm is computed via case I, i.e. basically always except roughly for  $0.8 < a < 1.26$ , its fractional part has only about 4 safety digits. This is barely enough for  $a^b$  with  $b$  near 1000 and certainly not enough for  $a^b$  with  $b$  of the order 10000.

I hesitated with the option to always handle  $b$  as  $N+h$  with  $N$  integer for which we can use old `xintFloatPower` (which perhaps I will have to update to ensure better than the 0.52ulp it mentions in its documentation). But in the end, I decided to simply add a variant where case I is handled as case II, i.e. with 9 not 4 safety fractional digits for the logarithm. This variant will be the one used by the power function for fractional exponents (non integer, non half-integer).

```
701 \def\XINT_tmpa#1.{%
702 \def\XINTinFloatLog{\romannumeral0\XINTinfloatlog}%
703 \def\XINTinfloatlog
```



## TOC

TOC, xintkernel, xinttools, xintcore, xint, xintbinhex, xintgcd, xintfrac, xintseries, xintcfrc, xintexpr, xinttrig, xintlog

```

704 {%
705     \expandafter\XINT_log_out
706     \romannumeral0\expandafter\XINT_logtenxdg_a
707     \romannumeral0\XINTinfloat[#1]{##1}
708 }%
709 \def\XINT_log_out ##1\xint:##2\xint:
710 {%
711     \XINTinfloat[#1]%
712     {\xintAdd{\xintMul{\XINT_c_logten}{##1}}{##2}}%
713 }%
714 \def\XINTinFloatLogTen{\romannumeral0\XINTinfloatlogten}%
715 \def\XINTinfloatlogten
716 {%
717     \expandafter\XINT_logten_out
718     \romannumeral0\expandafter\XINT_logtenxdg_a
719     \romannumeral0\XINTinfloat[#1]{##1}
720 }%
721 \def\XINT_logten_out ##1\xint:##2\xint:
722 {%
723     \XINTinfloat[#1]%
724     {\xintAdd{##1}{\xintMul{\XINT_c_oneoverlogten}{##2}}}%
725 }%
726 }\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax.%
727 \def\XINTinFloatLogTen_xdgout%#1[#2]
728 {%
729     \romannumeral0\expandafter\XINT_logten_xdgout\romannumeral0\XINT_logtenxdg_a
730 }%
731 \def\XINT_logten_xdgout #1\xint:#2\xint:
732 {%
733     \xintadd{#1}{\xintMul{\XINT_c_oneoverlogten_xx}{#2}}%
734 }%

```

No check is done whether input is negative or vanishes. We apply `\XINTinfloat[9]` which if input is not zero always produces 9 digits (and perhaps a minus sign) the first digit is non-zero. This is the expected input to `\numexpr\PML@<digits><dot>.\relax`

The variants `xdg_a`, `xdg_b`, `xdg_c`, `xdg_d` were added at 1.4f to always go via II or III, ensuring more fractional digits to the logarithm for accuracy of fractional powers with big exponents. "Old" 1.4e routines were removed.

```

735 \def\XINT_logtenxdg_a#1[#2]%
736 {%
737     \expandafter\XINT_logtenxdg_b
738     \romannumeral0\XINTinfloat[9]{#1[#2]}#1[#2]%
739 }%
740 \def\XINT_logtenxdg_b#1[#2]%
741 {%
742     \expandafter\XINT_logtenxdg_c
743     \romannumeral0\xintround{6}%
744     {\xintiiAdd{\xintDSx{-9}{\the\numexpr#2+8\relax}}%
745     {\the\numexpr\PML@#1.\relax}%
746     [-9]}%
747     \xint:
748 }%

```

If we were either in `100000000[0]` or `999999999[-1]` for the `#1[#2]` `\XINT_logten_b` input, and only



in those cases, the `\xintRound{6}` produced "0". We are very near 1 and will treat this as case III, but this is sub-optimal.

```

749 \def\xINT_logtenxdg_c #1#2%
750 {%
751   \xint_gob_til_xint:#2\xINT_logten_IV\xint:
752   \XINT_logtenxdg_d #1#2%
753 }%
754 \def\xINT_logten_IV\xint:\XINT_logtenxdg_d0{\XINT_logten_f_III}%

```

Here we are certain that `\xintRound{6}` produced a decimal point and 6 fractional digit tokens #2, but they can be zeros and also -0.000000 is possible.

If #1 vanishes and #2>100000 we are in case I.

If #1 vanishes and 100000>=#2>0 we are in case II.

If #1 and #2 vanish we are in case III.

If #1 does not vanish we are in case I with a direct quicker access if #2 vanishes.

Attention to the sign of #1, it is checked later on.

At 1.4f, we handle the case I with as many digits as case II (and exceptionnally case III).

```

755 \def\xINT_logtenxdg_d #1.#2\xint:
756 {%
757   \ifcase
758     \ifnum#1=\xint_c_
759       \ifnum #2=\xint_c_ \xint_c_iii\else \xint_c_ii\fi
760     \else
761       \ifnum#2>\xint_c_ \xint_c_ii\else \xint_c_\fi
762     \fi
763     \expandafter\xINT_logten_f_Isp
764   \or% never
765   \or\expandafter\xINT_logten_f_IorII
766   \else\expandafter\xINT_logten_f_III
767   \fi
768   #1.#2\xint:
769 }%
770 \def\xINT_logten_f_IorII#1%
771 {%
772   \xint_UDsignfork
773   #1\xINT_logten_f_IorII_neg
774   -\XINT_logten_f_IorII_pos
775   \krof #1%
776 }%

```

We are here only with a non-zero ##1, so no risk of a -0[0] which would be illegal usage of A[N] raw format. A negative ##1 is no trouble in ##3-##1.

```

777 \def\xINT_tmpa#1.{%
778 \def\xINT_logten_f_Isp##1.000000\xint:##2[##3]%
779 {%
780   {##1[0]}\xint:
781   {\expandafter\xINT_LogTen_serII_a_ii
782     \romannumeral0\xINTinfloatS[#1]{\xintAdd{##2[##3-##1]}\{-1[0]}}}%
783   \xint:
784   }\xint:
785 }%
786 }\expandafter\xINT_tmpa\the\numexpr\xINTdigitsormax.%
787 \def\xINT_tmpa#1.{%

```

```

788 \def\XINT_logten_f_III##1\xint:##2[##3]%
789 {%
790     {0[0]}\xint:
791     {\expandafter\XINT_LogTen_serIII_a_ii
792         \romannumeral0\XINTinfloatS[#1]{\xintAdd{##2[##3]}\{-1[0]}}}%
793     \xint:
794 } \xint:
795 }}\expandafter\XINT_tmpa\the\numexpr\XINTdigitsormax+4.%
796 \def\XINT_tmpa#1.#2.{%
797 \def\XINT_logten_f_IorII_pos##1.##2##3##4##5##6##7\xint:##8[##9]%
798 {%
799     {\the\numexpr##1##2##3##4##5##6##7[-6]}\xint:
800     {\expandafter\XINT_LogTen_serII_a_ii
801         \romannumeral0\XINTinfloat[#2]%
802         {\xintAdd{-1[0]}}%
803         {\xintMul{\csname XINT_c_1_##2_inv_x\endcsname}{%
804             \XINTinFloat[#1]}%
805             \xintMul{\csname XINT_c_2_##3_inv_x\endcsname}{%
806                 \XINTinFloat[#1]}%
807             \xintMul{\csname XINT_c_3_##4_inv_x\endcsname}{%
808                 \XINTinFloat[#1]}%
809             \xintMul{\csname XINT_c_4_##5_inv_x\endcsname}{%
810                 \XINTinFloat[#1]}%
811             \xintMul{\csname XINT_c_5_##6_inv_x\endcsname}{%
812                 \XINTinFloat[#1]}%
813             \xintMul{\csname XINT_c_6_##7_inv_x\endcsname}
814                 {##8[##9-##1]}}}%
815         {}{}{}{}{}{}{}%
816     }%
817 } \xint:
818 } \xint:
819 }%
820 \def\XINT_logten_f_IorII_neg##1.##2##3##4##5##6##7\xint:##8[##9]%
821 {%
822     {\the\numexpr##1##2##3##4##5##6##7[-6]}\xint:
823     {\expandafter\XINT_LogTen_serII_a_ii
824         \romannumeral0\XINTinfloat[#2]%
825         {\xintAdd{-1[0]}}%
826         {\xintMul{\csname XINT_c_1_##2_x\endcsname}{%
827             \XINTinFloat[#1]}%
828             \xintMul{\csname XINT_c_2_##3_x\endcsname}{%
829                 \XINTinFloat[#1]}%
830             \xintMul{\csname XINT_c_3_##4_x\endcsname}{%
831                 \XINTinFloat[#1]}%
832             \xintMul{\csname XINT_c_4_##5_x\endcsname}{%
833                 \XINTinFloat[#1]}%
834             \xintMul{\csname XINT_c_5_##6_x\endcsname}{%
835                 \XINTinFloat[#1]}%
836             \xintMul{\csname XINT_c_6_##7_x\endcsname}
837                 {##8[##9-##1]}}}%
838         {}{}{}{}{}{}{}%
839     }%

```

```

840     }\xint:
841     }\xint:
842 }%
843 }\expandafter\XINT_tmpa
844 \the\numexpr\XINTdigitsormax+10\expandafter.\the\numexpr\XINTdigitsormax+4.%

```

Initially all of this was done in a single big nested macro but the float-rounding of argument to less digits worked again each time from initial long input; the advantage on the other hand was that the  $1/i$  constants were all pre-computed and rounded.

Pre-coding the successive rounding to six digits less at each stage could be done via a single loop which would then walk back up inserting coeffs like  $1/\#1$  having no special optimizing tricks. Pre-computing the  $1/\#1$  too is possible but then one would have to copy the full set of such constants (which would be pre-computed depending on  $P$ ), and this will add grabbing overhead in the loop expansion. Or one defines macros to hold the pre-rounded constants.

Finally I do define macros, not only to hold the constants but to hold the whole build-up. Sacrificing brevity of code to benefit of expansion "speed".

First one prepares  $\eta$ , with  $P+4$  digits for mantissa, and then hands it over to the log series. This will proceed via first preparing  $\eta\backslash xint: \eta\backslash xint: \dots \eta\backslash xint:$ , the leftmost ones being more and more reduced in number of digits. Finally one goes back up to the right, the hard-coded number of steps depending on value of  $P=\backslash XINTdigits$  at time of reloading of package. This number of steps is hard-coded in the number of macros which get defined.

Descending (leftwards) chain:  $\_a$ , Turning point:  $\_b$ , Ascending:  $\_c$ .

As it is very easy to make silly typing mistakes in the numerous macros I have refactored a number of times the set-up to make manual verification straightforward. Automatization is possible but the  $\_b$  macros complicate things, each one is its own special case. In the end the set-up will define then redefine some  $\_a$  and the (finally unique)  $\_b$  macro, this allows easier to read code, with no nesting of conditionals or else branches.

Actually series III and series II differ by only a shift by and we could use always the slightly more costly series III in place of series II. But that would add one un-needed term and a bit overhead to the default  $P$  which is 16...

(1.4f: hesitation on 2021/05/09 after removal or case I log series should I not follow the simplifying logic and use always the slightly more costly III?)

### 29.11.1. Log series, case II

```

845 \def\XINT_tmpa#1.#2.{%
846 \def\XINT_LogTen_serII_a_ii##1\xint:
847 {%
848     \expandafter\XINT_LogTen_serII_b
849     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
850 }%
851 \def\XINT_LogTen_serII_b#1[#2]\xint:
852 {%
853     \expandafter\XINT_LogTen_serII_c_
854     \romannumeral0\xintadd{1}{\xintiiOpp\xintHalf{#10}{#2-1}}\xint:
855 }%
856 \def\XINT_LogTen_serII_c_##1\xint:##2\xint:
857 {%
858     \XINTinFloat[#2]{\xintMul{##1}{##2}}%
859 }%
860 }%
861 \expandafter\XINT_tmpa
862     \the\numexpr\XINTdigitsormax-2\expandafter.%
863     \the\numexpr\XINTdigitsormax+4.%

```

```

864 \ifnum\XINTdigits>10
865 \def\XINT_tmpa#1.#2.#3.#4.{%
866 \def\XINT_LogTen_serII_a_ii##1\xint:
867 {%
868     \expandafter\XINT_LogTen_serII_a_iii
869     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
870 }%
871 \def\XINT_LogTen_serII_a_iii##1\xint:
872 {%
873     \expandafter\XINT_LogTen_serII_b
874     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
875 }%
876 \def\XINT_LogTen_serII_b##1[##2]\xint:
877 {%
878     \expandafter\XINT_LogTen_serII_c_i
879     \romannumeral0\xintadd{#3}{##1/3[##2]}\xint:
880 }%
881 \def\XINT_LogTen_serII_c_i##1\xint:##2\xint:
882 {%
883     \expandafter\XINT_LogTen_serII_c_
884     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
885 }%
886 }\expandafter\XINT_tmpa
887 \the\numexpr\XINTdigitsormax-8\expandafter.%
888 \the\numexpr\XINTdigitsormax-2.%
889 {-5[-1]}.%
890 {1[0]}.%
891 \fi
892 \ifnum\XINTdigits>16
893 \def\XINT_tmpa#1.#2.#3.#4.{%
894 \def\XINT_LogTen_serII_a_iii##1\xint:
895 {%
896     \expandafter\XINT_LogTen_serII_a_iv
897     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
898 }%
899 \def\XINT_LogTen_serII_a_iv##1\xint:
900 {%
901     \expandafter\XINT_LogTen_serII_b
902     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
903 }%
904 \def\XINT_LogTen_serII_b##1[##2]\xint:
905 {%
906     \expandafter\XINT_LogTen_serII_c_ii
907     \romannumeral0\xintadd{#3}{\xintiiMul{-25}{##1}[##2-2]}\xint:
908 }%
909 \def\XINT_LogTen_serII_c_ii##1\xint:##2\xint:
910 {%
911     \expandafter\XINT_LogTen_serII_c_i
912     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
913 }%
914 }\expandafter\XINT_tmpa
915 \the\numexpr\XINTdigitsormax-14\expandafter.%

```

```

916 \the\numexpr\XINTdigitsormax-8\expandafter.%
917 \romannumeral0\XINTinfloat[\XINTdigitsormax-8]{1/3[0]}.%
918 {-5[-1]}.%
919 \fi
920 \ifnum\XINTdigits>22
921 \def\XINT_tmpa#1.#2.#3.#4.{%
922 \def\XINT_LogTen_serII_a_iv##1\xint:
923 {%
924 \expandafter\XINT_LogTen_serII_a_v
925 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
926 }%
927 \def\XINT_LogTen_serII_a_v##1\xint:
928 {%
929 \expandafter\XINT_LogTen_serII_b
930 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
931 }%
932 \def\XINT_LogTen_serII_b##1[##2]\xint:
933 {%
934 \expandafter\XINT_LogTen_serII_c_iii
935 \romannumeral0\xintadd{#3}{\xintDouble{##1}[##2-1]}\xint:
936 }%
937 \def\XINT_LogTen_serII_c_iii##1\xint:##2\xint:
938 {%
939 \expandafter\XINT_LogTen_serII_c_ii
940 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
941 }%
942 }\expandafter\XINT_tmpa
943 \the\numexpr\XINTdigitsormax-20\expandafter.%
944 \the\numexpr\XINTdigitsormax-14\expandafter.\expanded{%
945 {-25[-2]}.%
946 \XINTinFloat[\XINTdigitsormax-8]{1/3[0]}.%
947 }%
948 \fi
949 \ifnum\XINTdigits>28
950 \def\XINT_tmpa#1.#2.#3.#4.{%
951 \def\XINT_LogTen_serII_a_v##1\xint:
952 {%
953 \expandafter\XINT_LogTen_serII_a_vi
954 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
955 }%
956 \def\XINT_LogTen_serII_a_vi##1\xint:
957 {%
958 \expandafter\XINT_LogTen_serII_b
959 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
960 }%
961 \def\XINT_LogTen_serII_b##1[##2]\xint:
962 {%
963 \expandafter\XINT_LogTen_serII_c_iv
964 \romannumeral0\xintadd{#3}{\xintiiOpp##1/6[##2]}\xint:
965 }%
966 \def\XINT_LogTen_serII_c_iv##1\xint:##2\xint:
967 {%

```

```

968 \expandafter\XINT_LogTen_serII_c_iii
969 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
970 }%
971 }\expandafter\XINT_tmpa
972 \the\numexpr\XINTdigitsormax-26\expandafter.%
973 \the\numexpr\XINTdigitsormax-20.%
974 {2[-1]}.%
975 {-25[-2]}.%
976 \fi
977 \ifnum\XINTdigits>34
978 \def\XINT_tmpa#1.#2.#3.#4.{%
979 \def\XINT_LogTen_serII_a_vii##1\xint:
980 {%
981 \expandafter\XINT_LogTen_serII_a_vii
982 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
983 }%
984 \def\XINT_LogTen_serII_a_vii##1\xint:
985 {%
986 \expandafter\XINT_LogTen_serII_b
987 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
988 }%
989 \def\XINT_LogTen_serII_b##1[##2]\xint:
990 {%
991 \expandafter\XINT_LogTen_serII_c_v
992 \romannumeral0\xintadd{#3}{##1/7[##2]}\xint:
993 }%
994 \def\XINT_LogTen_serII_c_v##1\xint:##2\xint:
995 {%
996 \expandafter\XINT_LogTen_serII_c_iv
997 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
998 }%
999 }\expandafter\XINT_tmpa
1000 \the\numexpr\XINTdigitsormax-32\expandafter.%
1001 \the\numexpr\XINTdigitsormax-26\expandafter.%
1002 \romannumeral0\XINTinfloatS[\XINTdigitsormax-26]{-1/6[0]}.%
1003 {2[-1]}.%
1004 \fi
1005 \ifnum\XINTdigits>40
1006 \def\XINT_tmpa#1.#2.#3.#4.{%
1007 \def\XINT_LogTen_serII_a_vii##1\xint:
1008 {%
1009 \expandafter\XINT_LogTen_serII_a_viii
1010 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1011 }%
1012 \def\XINT_LogTen_serII_a_viii##1\xint:
1013 {%
1014 \expandafter\XINT_LogTen_serII_b
1015 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1016 }%
1017 \def\XINT_LogTen_serII_b##1[##2]\xint:
1018 {%
1019 \expandafter\XINT_LogTen_serII_c_vi

```

```

1020 \romannumeral0\xintadd{#3}{\xintiiMul{-125}{##1}{##2-3}}\xint:
1021 }%
1022 \def\XINT_LogTen_serII_c_vi##1\xint:##2\xint:
1023 {%
1024 \expandafter\XINT_LogTen_serII_c_v
1025 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1026 }%
1027 }\expandafter\XINT_tmpa
1028 \the\numexpr\XINTdigitsormax-38\expandafter.%
1029 \the\numexpr\XINTdigitsormax-32\expandafter.\expanded{%
1030 \XINTinFloat[\XINTdigitsormax-32]{1/7[0]}.%
1031 \XINTinFloat[\XINTdigitsormax-26]{-1/6[0]}.%
1032 }%
1033 \fi
1034 \ifnum\XINTdigits>46
1035 \def\XINT_tmpa#1.#2.#3.#4.{%
1036 \def\XINT_LogTen_serII_a_viii##1\xint:
1037 {%
1038 \expandafter\XINT_LogTen_serII_a_ix
1039 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1040 }%
1041 \def\XINT_LogTen_serII_a_ix##1\xint:
1042 {%
1043 \expandafter\XINT_LogTen_serII_b
1044 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1045 }%
1046 \def\XINT_LogTen_serII_b##1[##2]\xint:
1047 {%
1048 \expandafter\XINT_LogTen_serII_c_vii
1049 \romannumeral0\xintadd{#3}{##1/9[##2]}\xint:
1050 }%
1051 \def\XINT_LogTen_serII_c_vii##1\xint:##2\xint:
1052 {%
1053 \expandafter\XINT_LogTen_serII_c_vi
1054 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1055 }%
1056 }\expandafter\XINT_tmpa
1057 \the\numexpr\XINTdigitsormax-44\expandafter.%
1058 \the\numexpr\XINTdigitsormax-38\expandafter.\expanded{%
1059 {-125[-3]}.%
1060 \XINTinFloat[\XINTdigitsormax-32]{1/7[0]}.%
1061 }%
1062 \fi
1063 \ifnum\XINTdigits>52
1064 \def\XINT_tmpa#1.#2.#3.#4.{%
1065 \def\XINT_LogTen_serII_a_ix##1\xint:
1066 {%
1067 \expandafter\XINT_LogTen_serII_a_x
1068 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1069 }%
1070 \def\XINT_LogTen_serII_a_x##1\xint:
1071 {%

```

```

1072 \expandafter\XINT_LogTen_serII_b
1073 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1074 }%
1075 \def\XINT_LogTen_serII_b##1[##2]\xint:
1076 {%
1077 \expandafter\XINT_LogTen_serII_c_viii
1078 \romannumeral0\xintadd{#3}{\xintiiOpp##1[##2-1]}\xint:
1079 }%
1080 \def\XINT_LogTen_serII_c_viii##1\xint:##2\xint:
1081 {%
1082 \expandafter\XINT_LogTen_serII_c_vii
1083 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1084 }%
1085 }\expandafter\XINT_tmpa
1086 \the\numexpr\XINTdigitsormax-50\expandafter.%
1087 \the\numexpr\XINTdigitsormax-44\expandafter.%
1088 \romannumeral0\XINTinfloat[\XINTdigitsormax-44]{1/9[0]}.%
1089 {-125[-3]}.%
1090 \fi
1091 \ifnum\XINTdigits>58
1092 \def\XINT_tmpa#1.#2.#3.#4.{%
1093 \def\XINT_LogTen_serII_a_x##1\xint:
1094 {%
1095 \expandafter\XINT_LogTen_serII_a_xi
1096 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1097 }%
1098 \def\XINT_LogTen_serII_a_xi##1\xint:
1099 {%
1100 \expandafter\XINT_LogTen_serII_b
1101 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1102 }%
1103 \def\XINT_LogTen_serII_b##1[##2]\xint:
1104 {%
1105 \expandafter\XINT_LogTen_serII_c_ix
1106 \romannumeral0\xintadd{#3}{##1/11[##2]}\xint:
1107 }%
1108 \def\XINT_LogTen_serII_c_ix##1\xint:##2\xint:
1109 {%
1110 \expandafter\XINT_LogTen_serII_c_viii
1111 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1112 }%
1113 }\expandafter\XINT_tmpa
1114 \the\numexpr\XINTdigitsormax-56\expandafter.%
1115 \the\numexpr\XINTdigitsormax-50\expandafter.\expanded{%
1116 {-1[-1]}.%
1117 \XINTinFloat[\XINTdigitsormax-44]{1/9[0]}.%
1118 }%
1119 \fi

```

### 29.11.2. Log series, case III

```

1120 \def\XINT_tmpa#1.#2.{%
1121 \def\XINT_LogTen_serIII_a_ii##1\xint:

```



```

1122 {%
1123     \expandafter\XINT_LogTen_serIII_b
1124     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1125 }%
1126 \def\XINT_LogTen_serIII_b#1[#2]\xint:
1127 {%
1128     \expandafter\XINT_LogTen_serIII_c_
1129     \romannumeral0\xintadd{1}{\xintiOpp\xintHalf{#10}[#2-1]}\xint:
1130 }%
1131 \def\XINT_LogTen_serIII_c_##1\xint:##2\xint:
1132 {%
1133     \XINTinFloat[#2]{\xintMul{##1}{##2}}%
1134 }%
1135 }%
1136 \expandafter\XINT_tmpa
1137     \the\numexpr\XINTdigitsormax-1\expandafter.%
1138     \the\numexpr\XINTdigitsormax+4.%
1139 \ifnum\XINTdigits>9
1140 \def\XINT_tmpa#1.#2.#3.#4.{%
1141 \def\XINT_LogTen_serIII_a_ii##1\xint:
1142 {%
1143     \expandafter\XINT_LogTen_serIII_a_iii
1144     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1145 }%
1146 \def\XINT_LogTen_serIII_a_iii##1\xint:
1147 {%
1148     \expandafter\XINT_LogTen_serIII_b
1149     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1150 }%
1151 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1152 {%
1153     \expandafter\XINT_LogTen_serIII_c_i
1154     \romannumeral0\xintadd{#3}{##1/3[##2]}\xint:
1155 }%
1156 \def\XINT_LogTen_serIII_c_i##1\xint:##2\xint:
1157 {%
1158     \expandafter\XINT_LogTen_serIII_c_
1159     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1160 }%
1161 }\expandafter\XINT_tmpa
1162     \the\numexpr\XINTdigitsormax-7\expandafter.%
1163     \the\numexpr\XINTdigitsormax-1.%
1164     {-5[-1]}.%
1165     {1[0]}.%
1166 \fi
1167 \ifnum\XINTdigits>15
1168 \def\XINT_tmpa#1.#2.#3.#4.{%
1169 \def\XINT_LogTen_serIII_a_iii##1\xint:
1170 {%
1171     \expandafter\XINT_LogTen_serIII_a_iv
1172     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1173 }%

```

```

1174 \def\XINT_LogTen_serIII_a_iv##1\xint:
1175 {%
1176     \expandafter\XINT_LogTen_serIII_b
1177     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1178 }%
1179 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1180 {%
1181     \expandafter\XINT_LogTen_serIII_c_ii
1182     \romannumeral0\xintadd{#3}{\xintiMul{-25}{##1}[##2-2]}\xint:
1183 }%
1184 \def\XINT_LogTen_serIII_c_ii##1\xint:##2\xint:
1185 {%
1186     \expandafter\XINT_LogTen_serIII_c_i
1187     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1188 }%
1189 }\expandafter\XINT_tmpa
1190 \the\numexpr\XINTdigitsormax-13\expandafter.%
1191 \the\numexpr\XINTdigitsormax-7\expandafter.%
1192 \romannumeral0\XINTinfloat[\XINTdigitsormax-7]{1/3[0]}.%
1193 {-5[-1]}.%
1194 \fi
1195 \ifnum\XINTdigits>21
1196 \def\XINT_tmpa#1.#2.#3.#4.{%
1197 \def\XINT_LogTen_serIII_a_iv##1\xint:
1198 {%
1199     \expandafter\XINT_LogTen_serIII_a_v
1200     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1201 }%
1202 \def\XINT_LogTen_serIII_a_v##1\xint:
1203 {%
1204     \expandafter\XINT_LogTen_serIII_b
1205     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1206 }%
1207 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1208 {%
1209     \expandafter\XINT_LogTen_serIII_c_iii
1210     \romannumeral0\xintadd{#3}{\xintDouble{##1}[##2-1]}\xint:
1211 }%
1212 \def\XINT_LogTen_serIII_c_iii##1\xint:##2\xint:
1213 {%
1214     \expandafter\XINT_LogTen_serIII_c_ii
1215     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1216 }%
1217 }\expandafter\XINT_tmpa
1218 \the\numexpr\XINTdigitsormax-19\expandafter.%
1219 \the\numexpr\XINTdigitsormax-13\expandafter.\expanded{%
1220 {-25[-2]}.%
1221 \XINTinFloat[\XINTdigitsormax-7]{1/3[0]}.%
1222 }%
1223 \fi
1224 \ifnum\XINTdigits>27
1225 \def\XINT_tmpa#1.#2.#3.#4.{%

```

```

1226 \def\XINT_LogTen_serIII_a_v##1\xint:
1227 {%
1228     \expandafter\XINT_LogTen_serIII_a_vi
1229     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1230 }%
1231 \def\XINT_LogTen_serIII_a_vi##1\xint:
1232 {%
1233     \expandafter\XINT_LogTen_serIII_b
1234     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1235 }%
1236 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1237 {%
1238     \expandafter\XINT_LogTen_serIII_c_iv
1239     \romannumeral0\xintadd{#3}{\xintiiOpp##1/6[##2]}\xint:
1240 }%
1241 \def\XINT_LogTen_serIII_c_iv##1\xint:##2\xint:
1242 {%
1243     \expandafter\XINT_LogTen_serIII_c_iii
1244     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1245 }%
1246 }\expandafter\XINT_tmpa
1247 \the\numexpr\XINTdigitsormax-25\expandafter.%
1248 \the\numexpr\XINTdigitsormax-19.%
1249 {2[-1]}.%
1250 {-25[-2]}.%
1251 \fi
1252 \ifnum\XINTdigits>33
1253 \def\XINT_tmpa#1.#2.#3.#4.{%
1254 \def\XINT_LogTen_serIII_a_vi##1\xint:
1255 {%
1256     \expandafter\XINT_LogTen_serIII_a_vii
1257     \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1258 }%
1259 \def\XINT_LogTen_serIII_a_vii##1\xint:
1260 {%
1261     \expandafter\XINT_LogTen_serIII_b
1262     \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1263 }%
1264 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1265 {%
1266     \expandafter\XINT_LogTen_serIII_c_v
1267     \romannumeral0\xintadd{#3}{##1/7[##2]}\xint:
1268 }%
1269 \def\XINT_LogTen_serIII_c_v##1\xint:##2\xint:
1270 {%
1271     \expandafter\XINT_LogTen_serIII_c_iv
1272     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1273 }%
1274 }\expandafter\XINT_tmpa
1275 \the\numexpr\XINTdigitsormax-31\expandafter.%
1276 \the\numexpr\XINTdigitsormax-25\expandafter.%
1277 \romannumeral0\XINTinfloatS[\XINTdigitsormax-25]{-1/6[0]}.%

```

```

1278 {2[-1]}.%
1279 \fi
1280 \ifnum\XINTdigits>39
1281 \def\XINT_tmpa#1.#2.#3.#4.{%
1282 \def\XINT_LogTen_serIII_a_vii##1\xint:
1283 {%
1284 \expandafter\XINT_LogTen_serIII_a_viii
1285 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1286 }%
1287 \def\XINT_LogTen_serIII_a_viii##1\xint:
1288 {%
1289 \expandafter\XINT_LogTen_serIII_b
1290 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1291 }%
1292 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1293 {%
1294 \expandafter\XINT_LogTen_serIII_c_vi
1295 \romannumeral0\xintadd{#3}{\xintiiMul{-125}{##1}[##2-3]}\xint:
1296 }%
1297 \def\XINT_LogTen_serIII_c_vi##1\xint:##2\xint:
1298 {%
1299 \expandafter\XINT_LogTen_serIII_c_v
1300 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1301 }%
1302 }\expandafter\XINT_tmpa
1303 \the\numexpr\XINTdigitsormax-37\expandafter.%
1304 \the\numexpr\XINTdigitsormax-31\expandafter.\expanded{%
1305 \XINTinFloat[\XINTdigitsormax-31]{1/7[0]}.%
1306 \XINTinFloat[\XINTdigitsormax-25]{-1/6[0]}.%
1307 }%
1308 \fi
1309 \ifnum\XINTdigits>45
1310 \def\XINT_tmpa#1.#2.#3.#4.{%
1311 \def\XINT_LogTen_serIII_a_viii##1\xint:
1312 {%
1313 \expandafter\XINT_LogTen_serIII_a_ix
1314 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1315 }%
1316 \def\XINT_LogTen_serIII_a_ix##1\xint:
1317 {%
1318 \expandafter\XINT_LogTen_serIII_b
1319 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1320 }%
1321 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1322 {%
1323 \expandafter\XINT_LogTen_serIII_c_vii
1324 \romannumeral0\xintadd{#3}{##1/9[##2]}\xint:
1325 }%
1326 \def\XINT_LogTen_serIII_c_vii##1\xint:##2\xint:
1327 {%
1328 \expandafter\XINT_LogTen_serIII_c_vi
1329 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:

```

```

1330 }%
1331 }\expandafter\XINT_tmpa
1332 \the\numexpr\XINTdigitsormax-43\expandafter.%
1333 \the\numexpr\XINTdigitsormax-37\expandafter.\expanded{%
1334 {-125[-3]}.%
1335 \XINTinFloat[\XINTdigitsormax-31]{1/7[0]}.%
1336 }%
1337 \fi
1338 \ifnum\XINTdigits>51
1339 \def\XINT_tmpa#1.#2.#3.#4.{%
1340 \def\XINT_LogTen_serIII_a_ix##1\xint:
1341 {%
1342 \expandafter\XINT_LogTen_serIII_a_x
1343 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1344 }%
1345 \def\XINT_LogTen_serIII_a_x##1\xint:
1346 {%
1347 \expandafter\XINT_LogTen_serIII_b
1348 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1349 }%
1350 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1351 {%
1352 \expandafter\XINT_LogTen_serIII_c_viii
1353 \romannumeral0\xintadd{#3}{\xintiiOpp##1[##2-1]}\xint:
1354 }%
1355 \def\XINT_LogTen_serIII_c_viii##1\xint:##2\xint:
1356 {%
1357 \expandafter\XINT_LogTen_serIII_c_vii
1358 \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1359 }%
1360 }\expandafter\XINT_tmpa
1361 \the\numexpr\XINTdigitsormax-49\expandafter.%
1362 \the\numexpr\XINTdigitsormax-43\expandafter.%
1363 \romannumeral0\XINTinfloat[\XINTdigitsormax-43]{1/9[0]}.%
1364 {-125[-3]}.%
1365 \fi
1366 \ifnum\XINTdigits>57
1367 \def\XINT_tmpa#1.#2.#3.#4.{%
1368 \def\XINT_LogTen_serIII_a_x##1\xint:
1369 {%
1370 \expandafter\XINT_LogTen_serIII_a_xi
1371 \romannumeral0\XINTinfloatS[#2]{##1}\xint:##1\xint:
1372 }%
1373 \def\XINT_LogTen_serIII_a_xi##1\xint:
1374 {%
1375 \expandafter\XINT_LogTen_serIII_b
1376 \romannumeral0\XINTinfloatS[#1]{##1}\xint:##1\xint:
1377 }%
1378 \def\XINT_LogTen_serIII_b##1[##2]\xint:
1379 {%
1380 \expandafter\XINT_LogTen_serIII_c_ix
1381 \romannumeral0\xintadd{#3}{##1/11[##2]}\xint:

```

```

1382 }%
1383 \def\XINT_LogTen_serIII_c_ix##1\xint:##2\xint:
1384 {%
1385     \expandafter\XINT_LogTen_serIII_c_viii
1386     \romannumeral0\xintadd{#4}{\XINTinFloat[#2]{\xintMul{##1}{##2}}}\xint:
1387 }%
1388 }\expandafter\XINT_tmpa
1389 \the\numexpr\XINTdigitsormax-55\expandafter.%
1390 \the\numexpr\XINTdigitsormax-49\expandafter.\expanded{%
1391 {-1[-1]}.%
1392 \XINTinFloat[\XINTdigitsormax-43]{1/9[0]}.%
1393 }%
1394 \fi
1395 \XINTlogendinput%

```

## 30. Cumulative line and macro count

module	lines	macros	
<a href="#">xintkernel</a>	701	(160)	Total number of code lines: 19347. (but 4473 lines among them start either with {% or with }%). Each package starts with circa 50 lines dealing with catcodes, package identification and reloading management, also for Plain $\TeX$ .
<a href="#">xinttools</a>	1625	(376)	
<a href="#">xintcore</a>	2104	(525)	
<a href="#">xint</a>	1611	(405)	
<a href="#">xintbinhex</a>	783	(157)	Total number of def'ed (or let'ed) macros: 4595. This is an approximation as some macros are def'ed in a way escaping the automated detection, in particular this applies to <a href="#">xintexpr</a> macros associated to infix operators and syntax elements, whose construction uses <code>\csname</code> -based definitions with a template and auxiliary macros. Their number has been evaluated manually at being at least about 452 (this is incorporated into the <a href="#">xintexpr</a> count shown left, and the total above.)
<a href="#">xintgcd</a>	366	(63)	
<a href="#">xintfrac</a>	3683	(988)	
<a href="#">xintseries</a>	384	(66)	
<a href="#">xintcfrac</a>	1038	(257)	
<a href="#">xintexpr</a>	4788	(1407)	
<a href="#">xinttrig</a>	869	(68)	
<a href="#">xintlog</a>	1395	(123)	

Version 1.4o of 2025/09/06.