The Molecular Modeling Toolkit: a case study of a large scientific application in Python

Konrad Hinsen
Laboratoire de Dynamique Moléculaire
Institut de Biologie Structurale – Jean-Pierre Ebel
41, av. des Martyrs
38027 Grenoble Cedex 1, France
hinsen@ibs.fr

Abstract

The Molecular Modeling Toolkit is a library of Python modules and C extension modules that provides an object-oriented description of molecular systems and an implementation of the most common modeling techniques. It is one of the first large scientific applications implemented in Python. This article presents some aspects of its design and implementation that are typical for scientific applications in general. Its aim is to demonstrate by example how scientific computing can benefit from Python, and to show how common problems can be solved.

1 Introduction

Scientific computing is inherently experimental and exploratory. Although a large number of established algorithms and techniques are used again and again, the specific combination of techniques needed to solve a particular problem is usually unique and not precisely known at the start of a project.

Typically, a scientist using computational techniques has a collection of various programs that deal with some part of the problem at hand. Some of these programs are widely used and come with documentation and perhaps even technical support. Others have been written by a student – who has graduated long ago – for a specific task, and their suitability and reliability may be questionable. All these programs have their own conventions, input and output file formats, and idiosyncrasies. In practice, computational scientists spend a significant amount of time with finding and testing the programs they need and dealing with file conversion and other interfacing problems.

The combination of the individual computational steps occurs manually (interactively) or with simple shell scripts. Some of the programs used may have a built-in scripting language, but these scripting languages are never sufficient to deal with all aspects of a problem, and they certainly do not allow the integration of other independently written programs. Sometimes there is no other way left than to write a specialized program in a "real" programming language, in practice Fortran or C.

Another problem with scientific programs is that they lack even the most elementary elements of user-friendly design. This is partly due to lack of awareness of the problem; many scientists have never seen a user-friendly program. Also, most programs are quite old, dating back to the mainframe or even the punched-card era. In addition, the predominant scientific programming language, Fortran-77, makes user-friendly programming difficult, if not impossible. For example, binary files produced by a Fortran program are inherently machine and compiler dependent, which makes data exchange difficult even if everyone uses the same programs.

This article will show how Python can help to improve this situation. The example presented, molecular modeling, is not an untypical scientific application. Many of the design decisions and techniques can be applied to other scientific domains. The library described here, the Molecular Modeling Toolkit (from now on MMTK), is freely available [MMTK].

2 Molecular Modeling

The term *molecular modeling* is used to describe the study of molecules and molecular systems by means of computer models. The most common model for a molecule consists of a set of atoms, considered to be a point mass having a position in space and possibly other properties. All the atoms in a molecular system interact with each other; these interactions are described by the *interaction energy*, which is a function of the positions and properties of the atoms. The interaction energy is the central quantity in molecular modeling, and constructing an appropriate function is one of the major problems. There are many "recipes" for calculating the interaction energy, ranging from very simple functions of interatomic distances to the solution of the Schrödinger equation (the fundamental equation of quantum mechanics) for all electrons of all atoms in the system. These recipes are called *force fields*.

Like many other scientific applications, molecular modeling relies heavily on a few computationally intensive algorithms. An example is *energy minimization*, i.e. finding a configuration of atoms that is a local or global minimum of the (high-dimensional) interaction energy function; this operation can take several days or even weeks of CPU time for large systems. Another example is *normal mode analysis*, which essentially involves the calculation of the matrix of second derivatives of the interaction energy function and the diagonalization of this matrix. Even calculating the interaction energy itself requires a substantial amount of CPU time for large systems.

However, modelers spend most of their time on other operations, which are to a high degree interactive. The first important step is the definition of the system to be studied, i.e. deciding which molecules should participate, what a reasonable configuration looks like, and which force field should be used. Then an appropriate technique for obtaining the required information must be selected or developed. All this requires a large set of fundamental operations: reading in external (e.g. experimental) data, calculating, comparing, and modifying geometrical quantities, visualization, modifying force fields, energy minimization, etc. The few time-consuming operations mentioned above are in fact nothing special, they only take longer to complete. Nevertheless, traditional modeling programs are designed around these few operations, ignoring the others altogether or implementing them as an afterthought.

It should also be mentioned that molecular modeling is far from a mature discipline. Although some generally accepted techniques exist, they are not sufficient for studying many of the large systems that are of practical relevance. New methods and techniques are being developed continuously. Therefore molecular modeling software ought to be designed for extensibility, but traditional packages have proved to be unsatisfactory in this respect.

3 Overview of MMTK

Most molecular modeling programs are written around the few time-consuming algorithms and adapt their data models to them. A typical representation of a molecular system is a list of atoms plus an implicit force field (few programs can handle more than one force field). The molecular structure disappears, although it would be very useful for the user. Instead of referring to, for example, "the side chain of histidine-34 in the regulatory chain" (a very meaningful description for a biochemist), you might have to say "the atoms in group 270 whose names are not C, N, O or H".

In contrast, MMTK is based on an object-oriented description of molecular systems, with classes matching the real objects that chemists talk about: atoms, functional groups, molecules, and complexes. In addition there are object types describing force fields, universes (the "environments" of molecular systems), "machines" acting on a universe (e.g. energy minimizers), and the results of some operations (dynamical trajectories, normal modes etc.). Most of these objects are familiar to chemists, and new users have no problems understanding them, although it takes a while to get rid of old habits and use the object-oriented description efficiently.

As the name implies, MMTK is not a program, but a library. It does not embed Python as a scripting language, but extends Python with new modules. MMTK users have the freedom to choose and combine what they need from MMTK and from other libraries. This permits explorative work in a way that with traditional tools would be prohibitively complicated. For example, non-standard visualization tasks can be handled by a short Python program using MMTK, a VRML module, and an external VRML viewer. With traditional tools, the effort to extract and convert the data would be prohibitive.

The idea of building scientific code in the form of libraries that are tied together by an interpreter is not new. The Basis Code Development System developed at LLNL [Basis], for example, provides a scripting language extendable in Fortran, and Tcl has been used in some scientific applications. However, such approaches never received the attention they deserved, and scripting languages are by design limited in scope. Python, in contrast, is powerful enough to be used for most of the library itself.

MMTK is a rather large library, consisting (in version 1.0b2) of 36 Python modules and four C modules. It would be very difficult for users to remember what is defined in which module, especially since the modules define many functions and classes for internal use that are not of direct utility to the MMTK user. Therefore MMTK contains one more module which does nothing but import all the user-relevant definitions from the other modules. To the user, MMTK appears thus as one module. The disadvantage of this approach is that all modules are loaded when the "master" module is imported, even if they are not needed. As MMTK grows, it might become necessary to introduce some more of these “meta-modules”.

In summary, the implementation of MMTK as a Python library offers two essential advantages over traditional modeling programs: an object-oriented description fully accessible to the end user, and the freedom to combine MMTK with other libraries. Other scientific applications would equally benefit from this approach.

4 The “chemical object” classes

The central part of MMTK – and the part in which it offers the most advantages compared to traditional modeling programs – is the representation of chemical objects (atoms, molecules, etc.) and complete systems. A common base class, `GroupOfAtoms`, defines the operations that are defined solely in terms of atomic properties and positions. This includes all geometrical operations (finding the center of mass and moment of inertia, moving the object in several ways, superposing two configurations, etc.), but also visualization.

A subclass is `ChemicalObject`, which in turn has the subclasses `Atom` and `CompositeChemicalObject`. The latter has the subclasses `Group`, `Molecule`, and `Complex`. Complexes can contain molecules and atoms, molecules can contains groups and atoms, and groups can also contain groups and atoms, but groups cannot be used outside molecule definitions.

Other subclasses of `GroupOfAtoms` are `Collection`, representing any collection of chemical objects, and `Universe`, representing a complete system with a force field and perhaps an outer environment.

A less obvious choice is how to represent different types of molecules, for example water molecules and chloroform molecules. Molecules differ only in their composition, not in the operations defined for them. One could define a subclass for each molecule, redefining only the initialization method. But molecules can change identity during chemical reactions, whereas class instances cannot change their class. Therefore all molecules must be instances of the same class.

MMTK implements a prototype-based molecule factory. It maintains a database of atom types, group types, molecule types, and complex types. Each database entry is an object with a structure almost identical to that of the ultimate chemical object instance. Creating a real instance involves a procedure similar to that used in Python’s copy module: the attributes of the prototype are examined and analogous attributes in the instance are created. Attributes that are identical for each molecule of a certain kind (e.g. force field parameters) are not copied, but referenced from the new instance.

The disadvantage of this system is that the instantiation procedure is slow, having to examine all attributes again each time an instance is created. It is not uncommon to create thousands of molecules of the same kind in a system, so efficiency matters. It is planned to replace the instantiation system in the future by a compiled-code version: when a prototype is instantiated for the first time, it creates a custom instantiation function as a text string, compiles it using the `exec` function, and stores the function object for later use. Such a technique, which relies on the dynamic nature of Python, may be applicable to prototype-based object factories in general.

The database is not part of the MMTK source code, since it must frequently be extended by end users, who should not have to understand the instantiation system. Instead, users write each definition as a separate text file and place it in a directory which is searched by MMTK. The definition files are tiny Python programs that are executed with the `execfile` command in a context defined by a special Python module. In addition to the obvious advantage of not requiring a special parser, the use of tiny Python programs for definition and configuration files makes life easier for the user, who does not have to remember another syntax.

Finally it should be mentioned that the prototype database is not the only way to create chemical objects in MMTK. For chain molecules like proteins, whose building blocks are a small number of functional groups, it would be too much of an effort to write a database entry whose structure would be the same for all cases. Such molecules are described by specialized subclasses that define appropriate constructors and implement additional operations that would not make sense for general molecules.

5 Numerically intensive algorithms

It was mentioned earlier that molecular modeling requires a few computationally expensive operations. Not all of these operations are of the “simple-operation-on-large-dataset” type, so they cannot be implemented efficiently in terms of the array operations from the numerics extension (aka NumPy) [NumPy]. They must therefore be implemented in C modules.

However, a C module dealing with instances of Python classes is hardly faster than Python code itself. A low-level representation of the data (chemical objects etc.) is therefore necessary. This low-level representation is subject to two conditions:

- The efficiency of the C code must have top priority. Some of the data (e.g. the atomic configuration) is accessed very frequently. Requiring even as little as a C function call for each data access would be disastrous. Therefore the data must be stored in typical C data structures.

- The information in the low-level data structures should not be repeated in the high-level structures. That would be a waste of memory and an important source of errors. The high-level data structures should instead use references to the low-level ones.

One possibility would be to define several specialized C extension types. But writing an extension type involves a substantial effort, and unless the data is to be used directly in application programs, this effort is not justified. However, the array type provided by the numerics extension fits that description nicely. Its internal data structure is a C array, so C code can access it efficiently. On the other hand, Python code can easily create and store references to any part of the data. This approach keeps the C modules simple at the cost of some added complexity in the Python code, where it is easier to handle.

MMTK uses almost only arrays in the interface between Python and C code. An example is the storage of atomic configurations. At the C level, all positions of all atoms in a universe are stored in a single array for efficiency. At the Python level, each atom has an attribute `position` whose value is an instance of the class `Vector`. Class `Vector` stores its internal data (three coordinates) in an array – and this array is created by indexing from the full configuration array. This ensures consistency between the two system descriptions.

Three of the four C modules in MMTK cover force field evaluation, energy minimization, and computation of dynamical trajectories. The fourth one provides base services for algorithms that create trajectories, i.e. it provides specialized I/O operations. One other computationally expensive operation, normal mode analysis, is implemented in Python using the `LinearAlgebra` module of NumPy, which ultimately calls LAPACK, a highly optimized linear algebra library written in Fortran.

An important design goal in MMTK was the separation of force field evaluation and the algorithms that call it (such as energy minimization). Since these operations are closely intertwined at execution time, it is tempting to implement them together in one module. In fact, this is what most modeling programs do. However, this creates problems for later extensions: new force fields might not work correctly with all force field users, and new algorithms using force field information might not obtain the correct force field.

A consequence of the decision to separate these tasks into different modules is that MMTK's C modules must communicate among each other in an efficient way. This is different from the majority of C modules, which provide functions and object types to be used from Python code only. It is, however, typical for many large scientific applications.

If the modules were statically linked to the Python interpreter, they could simply provide a C API in the form of a header file to be included by client modules, much like the C API for built-in Python types. However, the possibility of compiling C modules into shared library files is a very convenient one, especially for code development. But shared libraries can't simply call each other's functions, at least not under all operating systems. A solution is provided by the CObject data type, introduced with Python 1.4. CObjects simply wrap a C pointer into a Python object. The C modules in MMTK create an array of `void *` in which they store the addresses of the type objects and C functions they want to export. The address of this array is put into a CObject, which is assigned to the name `_C_API` in the module's name space. Another C module wishing to use this C API imports the module and takes the address of the pointer array from the CObject. A header file that comes with the imported module defines the C API functions as macros in terms of the array entries.

It must be stressed that once the initial setup phase is finished, the time-critical algorithms in MMTK execute as pure C code on typical C data structures, with absolutely no overhead caused by the Python interpreter or the use of Python data types. The overhead caused by the use of pointer indirection for function calls across modules is negligible, and not uncommon even in pure C programs (not to mention C++). The only disadvantage compared to a straightforward C implementation is a more complex design.

6 Storing and retrieving data

The two main reasons to store data in permanent files, other than interchanging data with other people or programs, are storing the state of a system after interactive manipulations for later reuse and storing the result of a time-consuming operation that would be too costly to repeat. In molecular modeling, the objects that must be stored frequently are

- complete universes
- atomic configurations for a universe
- trajectories (i.e. sequences of configurations and other data)
- normal modes

In MMTK, everything except trajectories is saved and retrieved using the standard Python pickle module. This is easy and sufficiently efficient, and generates portable files. However, the resulting files are unnecessarily large, because they contain much detailed information that could easily be regenerated instead of loaded. This is a special problem for configurations and normal modes. Since these objects contain a reference to the universe for which they were generated, the whole universe is saved too. It would be nice to store a universe only once, and then be able to store configurations and normal modes in separate files containing just a reference to the universe file. This is in principle doable with the pickle module, but many details are not yet resolved. It may be preferable to switch to a different storage method.

Trajectories are different in that they can become much larger than computer memory. They have to be created directly in disk files, and read in piecewise, which would be difficult to arrange with the pickle module. In addition, the creation of trajectories must be efficient, both in CPU time and in disk space usage. Therefore MMTK uses the netCDF file format [netCDF] for its trajectories. The netCDF format is portable between machines and operating systems, and an efficient free library is available to read and write the files from C and Fortran programs. A Python interface module is available which lets Python programs access netCDF variables much like NumPy arrays [Python/netCDF].

Since trajectories are not written in pickle format, they do not automatically contain a description of the universe they belong to. Such a description is desirable both to allow verification when a trajectory is opened again for reading, and to make the trajectory fully self-describing, i.e. to allow analysis of a trajectory without needing additional files. One possibility would be to store a pickle string in the file, but CPU time, memory, and disk space use would be substantial. Instead a minimal description of the universe is stored in a special string format. This string can be executed in the context of a special module to construct a universe which matches the original one in structure (i.e. contains the same molecules), but not necessarily in other features that are irrelevant for the interpretation of the trajectory. Construction and reinterpretation of this string format are very fast, and it is much smaller than a full pickle representation. It might be possible to implement a general library module for storing and retrieving objects in such a format. It would require more support from the classes that use it, but offer efficiency in return.

7 MMTK in practice

Although MMTK is still young and does not have a large user base, it has been used successfully for real research problems, mostly for analyzing data from experiments or simulations performed with other programs. Some of these analyses would have been prohibitively complicated using traditional tools.

The most important advantage is the easy integration with other Python modules and external programs for which Python interface modules exist (e.g. visualization and plotting programs). Examples of non-standard visualization tasks that were made easy by MMTK (plus a VRML viewer) are a charge visualization realized by color-coding atoms according to their charge, and "static" trajectory visualization by lines indicating the paths of selected atoms.

Another advantage is the ease of using MMTK interactively. Since the bare Python interpreter is not very convenient for longer interactive sessions, it is recommended to use either Emacs with Python mode or the Tk-based GUI package PTUI [PTUI]. The combination with PTUI again demonstrates the advantage of MMTK's library approach.

A not yet satisfactorily resolved problem with MMTK is documentation. There is a user manual which explains the objects and functions that are of interest to scientists who want to apply the techniques provided by MMTK. However, scientists involved in the development and implementation of techniques need much more detailed information. A developer manual describing the design concepts will have to be provided, and documentation of individual modules with comments and documentation strings will have to be improved. In addition, a tutorial for beginners and people coming from traditional programs would be highly desirable.

A frequent question is whether MMTK is sufficiently fast for practical work. The parts written in Python were certainly fast enough to deal with the analysis of large-scale domain motion in a 26000-atom protein interactively on a 200 MHz PentiumPro machine running Linux. The time-critical routines are written entirely in C and don't use any Python features once they are started, so they are just as efficient as any other C implementation.

However, there is some overhead caused by the design decision to give safety, ease of use, and flexibility priority over speed. An example: For larger systems, a significant amount of CPU time is spent calculating the distance vectors between atoms. This is a trivial operation, taking negligible CPU time, but the code to be used depends on the geometry of the simulated system (e.g. periodic boundary conditions). Most modeling programs implement a fixed set of allowed geometries and hard-code the corresponding formulas. MMTK allows new geometries to be defined without any change to its code, at the cost of an indirect function call for distance vector calculation, which is used in the inner loop of the most time-critical calculation. Eventually a compromise will have to be found, e.g. hard-coding the most common geometries while still allowing external definitions. It is difficult to compare the computational efficiency of different modeling packages, because there are so many variations of the basic algorithms that affect speed as well as suitability for various applications. Nevertheless, to give a rough idea of the performance to be expected, a small real-life application (equilibration of a system of 300 water molecules with periodic boundary conditions) was done with both MMTK and CHARMM [CHARMM]. CHARMM is a widely used modeling program written in Fortran-77 with a reputation for both efficiency and user frustration. On a PentiumPro computer under Linux, using gcc and g77 for compilation, CHARMM was 2.5 times faster. On the other hand, the MMTK application consists of seven Python statements and worked correctly the first time, whereas the CHARMM input file contains 20 statements (in the CHARMM scripting language, which is more program-oriented than problem-oriented), worked only after an hour of trial-and-error modification, and required manual editing of the input geometry file, although it was in a widely used standard format. The total time required for this mini project was therefore significantly smaller with MMTK.

8 Future developments

Compared to the popular traditional modeling packages, MMTK is still lacking many features. This is to be expected, given that MMTK was written by a single person during about one year as a part-time project, whereas other programs represent the work of large research groups during several years. More modeling techniques will be implemented as the need arises and as time permits. Many established techniques that are not yet implemented have nevertheless been considered in the design phase and should therefore not cause difficulties.
Some possible future developments are particularly interesting and/or important:

- Parallelization of the time-critical computational techniques. The bookkeeping tasks could be handled in Python, leaving only a small part of hand-parallelized C code that could be hard to write and test.
- Development of GUI-based tools (using Tk) for some standard tasks on top of MMTK.
- Distributed computing: number crunching on a supercomputer, setup and analysis on a workstation. This could be realized via ILU.
- Interface modules for dealing with foreign file formats, to allow a progressive transition from traditional tools.

9 Conclusion

The MMTK project has shown that Python is not only a viable, but a very appropriate language for scientific computing. It allows to provide a problem-oriented object-oriented view for the user and an algorithm-oriented low-level one for routines in C or Fortran that deal with the numerically intensive parts of the application.
Based on the experience with MMTK, the following recommendations seem appropriate for scientific Python applications in general:

- Don't write programs, write libraries.
- Design the user-visible objects in a strictly problem-oriented way. Don't force the user to deal with low-level implementation details.
- Use arrays at the Python-C interface for data that has to be accessed from both sides.
- Provide a C API via CObjects for C modules that might be used directly from other C modules.
- As always: don't forget documentation.

A potential problem is that following these recommendations requires extensive experience not only with Python, but also with C. Most scientists do not have this experience and might not be willing or able to acquire it. A much more popular language in science, in spite of its many shortcomings, is Fortran. Mixed Fortran-C programming is possible on most machines and would allow the use of Fortran routines in Python extension modules. To be attractive to scientists, however, such an interfacing would have to be doable with no knowledge of C. Automatic interface generators, similar to SWIG for C and C++ code [SWIG], are possible, and indeed have been discussed [FIDL], but no such tool is actually available today. It would be an important step to make Python more attractive to the scientific community.

10 Acknowledgements

The author would like to thank the early MMTK users for their feedback, the entire Python community for help with many problems, and the Human Frontier Science Project Organization for financial support.

References

[MMTK] The Molecular Modelling Toolkit is available at
`http://starship.skyport.net/crew/hinsen/mmtk.html`.

[Basis] The Basis Code Development System, written at LLNL, is available at
`http://www-phys.llnl.gov/X_Div/htdocs/basis.html`.

[NumPy] The numerics extension, written by Jim Hugunin, is available at
`http://www.sls.lcs.mit.edu/jjh/numpy/`.

[netCDF] The Unidata netCDF format and library is available at
`http://www.unidata.ucar.edu/packages/netcdf/`.

[Python/netCDF] The netCDF interface module, written by Konrad Hinsen, is available at
`http://starship.skyport.net/crew/hinsen/netcdf.html`.

[PTUI] The user interface package PTUI, written by Zachary Roadhouse, is available at
`http://starship.skyport.net/crew/zack/ptui/`.

[CHARMM] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, *CHARMM, A Program for Macromolecular Energy, Minimization, and Dynamics Calculations*, J. Comp. Chem. 4 (1983) p. 187-217.

[SWIG] The Simplified Wrapper and Interface Generator (SWIG), written by David Beazly, is available at
`http://www.cs.utah.edu/~beazley/SWIG/swig.html`.

[FIDL] The FIDL project has been presented by Jim Fulton at a previous Python workshop (see `http://www.python.org/workshops/1995-05/extModulesNotes.html`), but never been completed.