

Chaco: A Plotting Package for Scientists and Engineers

**David C. Morrill
Enthought, Inc.**

Introduction

With packages such as Mathematica and MatLab, scientists and engineers already have a variety of high-quality plotting capabilities available to them. However, if you are a scientist or engineer who is also a Python user, or you are working with a limited budget, your choices are more limited. Chaco is an open source and freely available plotting package being developed to address many of the plotting needs of the scientific and engineering communities.

In many ways Python is a language well suited for scientific and engineering calculations, especially when used in conjunction with packages like Numeric. Its high-level syntax and interactive shell, combined with the high-performance and easy to use array manipulation capabilities provided by Numeric, make Python an attractive vehicle for scientific number crunching and analysis. Quite often the only thing missing from the "equation" is a high-quality, easy to use plotting package to express the results visually, either interactively on a computer screen, or as hard-copy via PDF or PostScript files.

Chaco is a package being developed to fill that void, and to do so in a way that scales well from simple, ad-hoc plots done from the Python command shell to elaborate plotting applications. And of course, Chaco is fully buzzword compliant:

- Open source (covered by a BSD style license)
- Works with both the wxPython and Tkinter GUI toolkits
- Cross-platform (courtesy of Python, Numeric, wxPython and Tkinter)
- Supports multiple output types (screen, PDF, PostScript, GIF, PNG, JPEG)

Chaco can be used interactively, either through the Python command shell or as part of a larger wxPython or Tkinter based application. It can also be used non-interactively, for example within a CGI request handler from a web server to dynamically generate plot images for use on web pages.

The Core Plotting Objects

Plotting is accomplished in Chaco using a small set of core plotting objects:

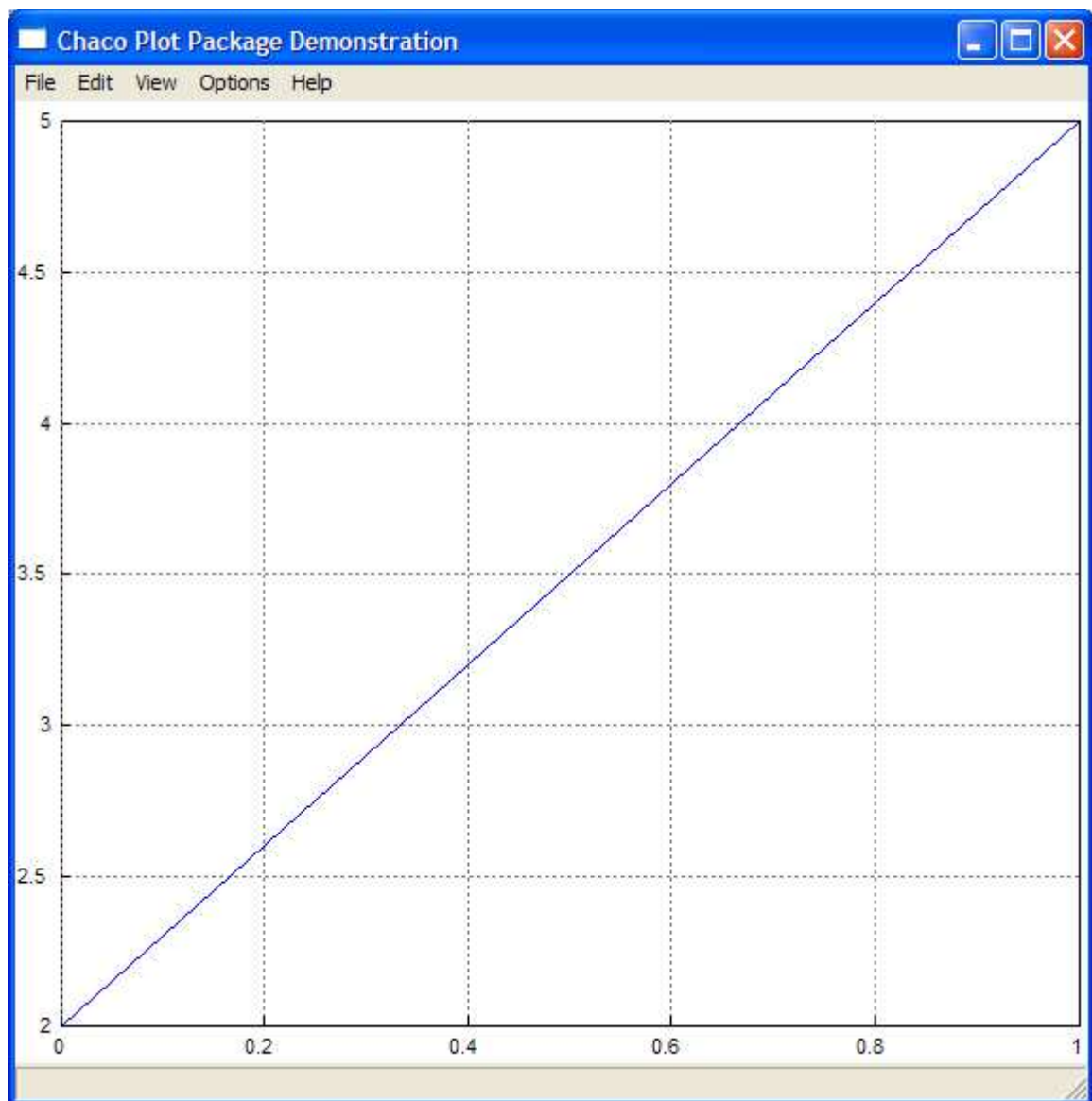
- **PlotValue**
- **PlotCanvas**
- **PlotIndexed**
- **PlotGroup**
- **PlotAxis**
- **PlotTitle**
- **PlotLabel**

For interactive use there is also an additional **PlotWindow** object that binds a collection of plotting objects to a particular GUI toolkit and display window.

Plots are created by composing plot objects in various ways. For example, perhaps the simplest possible plot is constructed using:

```
PlotValue( [ 2, 5 ] )
```

which creates a line plot of the two values (0,2) and (1,5) (the omitted x-values, 0 and 1, are automatically created by the **PlotValue** object). The following is an illustration of what this plot looks like using wxPython:



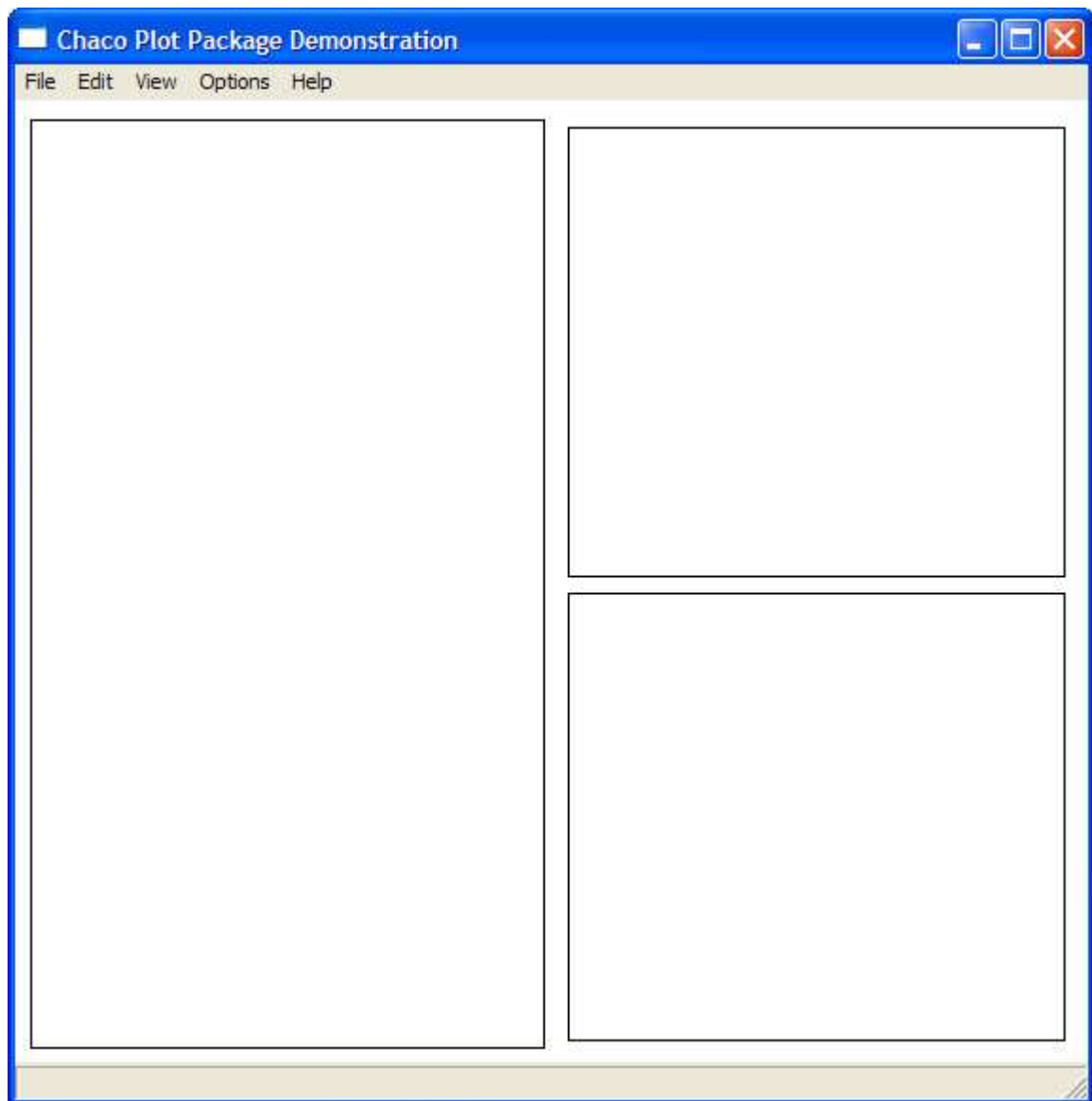
More complex plots are constructed by composing the core Chaco plotting objects in different ways. In the following sections we will give a quick tour of the high-level

features and characteristics of each of the core classes, accompanied by some visual illustrations of them in actual use.

The PlotGroup Class

A **PlotGroup** object is a container object used to contain and visually layout and organize other Chaco plotting objects. Any number of the other Chaco plotting objects can be added to a **PlotGroup**, including other **PlotGroup** objects. A **PlotGroup** has an **orientation** attribute, which can be either horizontal or vertical. Its orientation specifies the direction in which its contained objects are laid out.

This is illustrated in the following figure:



which shows a collection of three **PlotCanvas** objects organized using two nested **PlotGroup** objects:

```
PlotGroup( orientation = 'horizontal' ).add(  
    PlotCanvas(),  
    PlotGroup( orientation = 'vertical' ).add(  
        PlotCanvas(),  
        PlotCanvas()  
    )  
)
```

The PlotValue Class

A **PlotValue** object describes a single collection of related data to be plotted. In many ways, it can be considered to be simply an array of data values. The following is a simple example of how a **PlotValue** might be created:

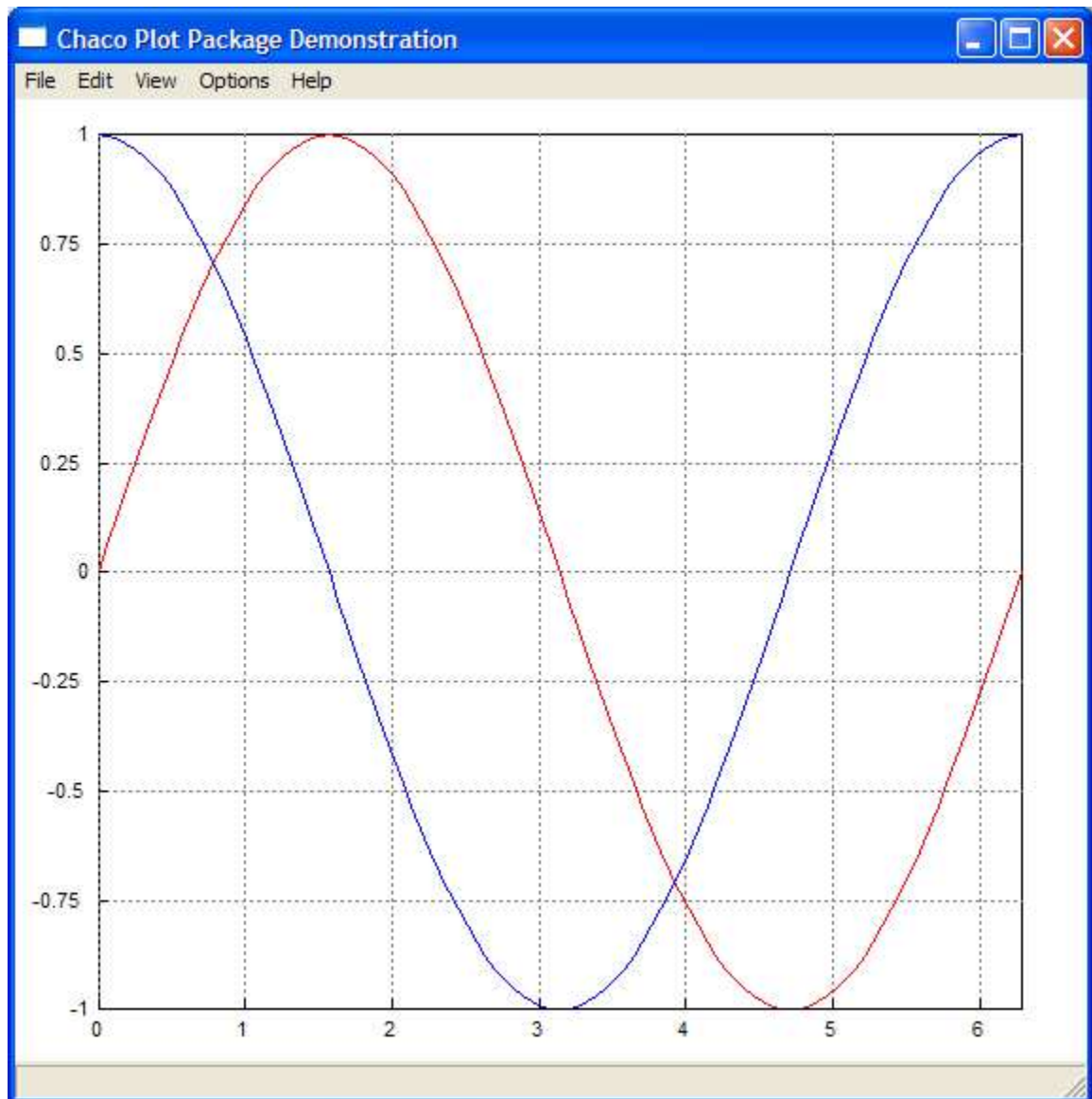
```
from Numeric import *  
x = arange( 0, 2 * pi, (2 * pi) / 100 )  
value = PlotValue( sin( x ) )
```

In this case, the **PlotValue** object contains the value of *sine* for all values between 0 and 2π over an evenly spaced group of 100 samples.

If used by itself, as in our earlier example, a **PlotValue** will render itself, along with its associated axes and grid. However, a **PlotValue** can also be imbedded in other objects, such as a **PlotCanvas**. In this case, a **PlotValue** object delegates many of its rendering functions up to its containing object. This will be illustrated in the next section.

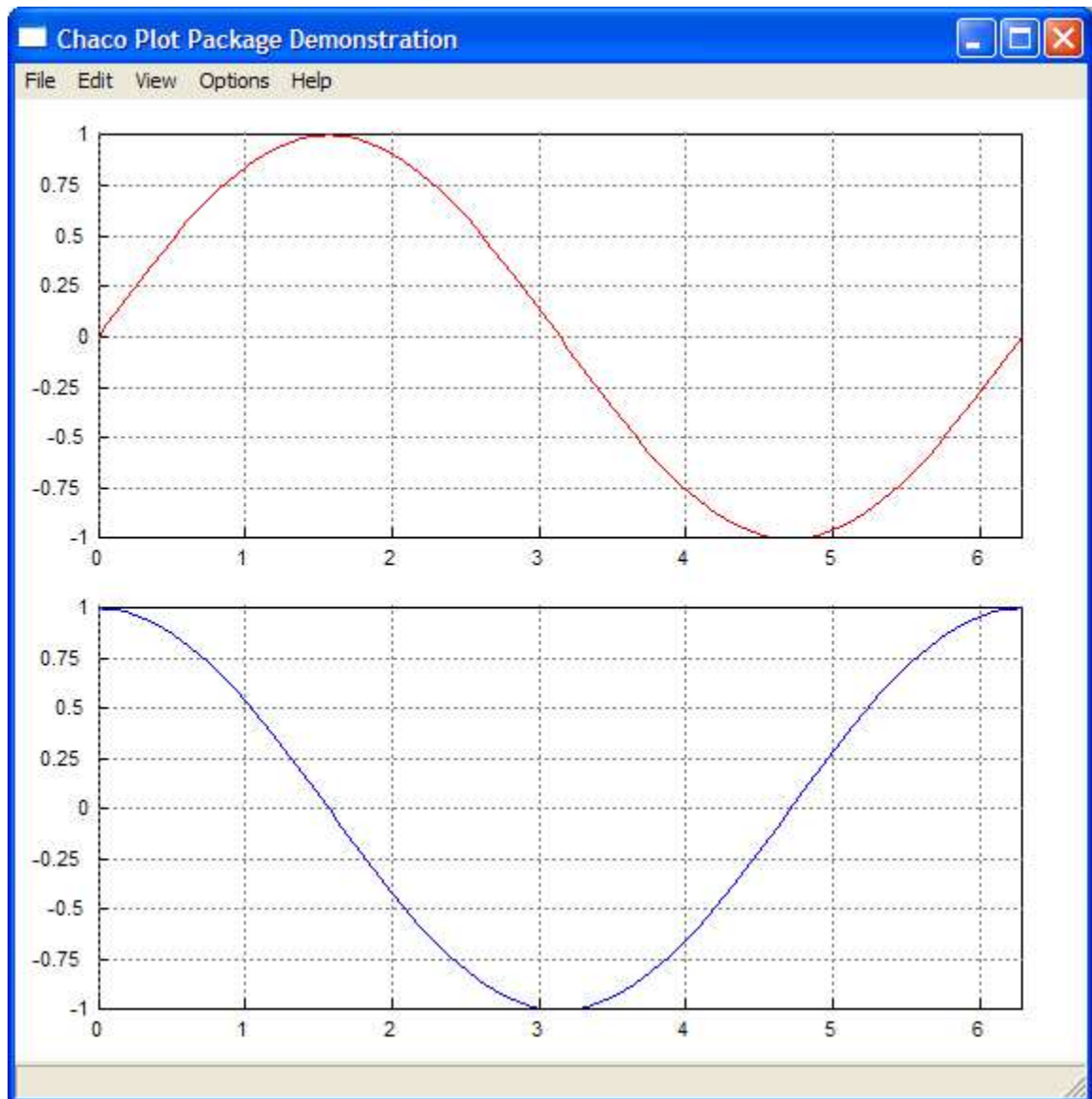
The PlotCanvas Class

A **PlotCanvas** is, as its names suggests, a canvas upon which data is plotted. Like a **PlotGroup**, it can contain other Chaco plot objects. Unlike a **PlotGroup** however, a **PlotCanvas** does not lay out its component parts in separate, non-overlapping areas, but instead manages and organizes them so that they overlap in a meaningful and useful manner, as shown in the following figure:



which shows a **PlotCanvas** object containing two separate **PlotValue** objects, one containing the values of *sine*, and the other containing the values of *cosine*, over the interval from 0 to 2π .

Any number of **PlotValues** can be added to a single **PlotCanvas** object. Notice how the behavior of the **PlotCanvas** is different from that of a **PlotGroup**. If the same two **PlotValue** objects had been added to a **PlotGroup**, instead of a **PlotCanvas**, the result would have looked like:



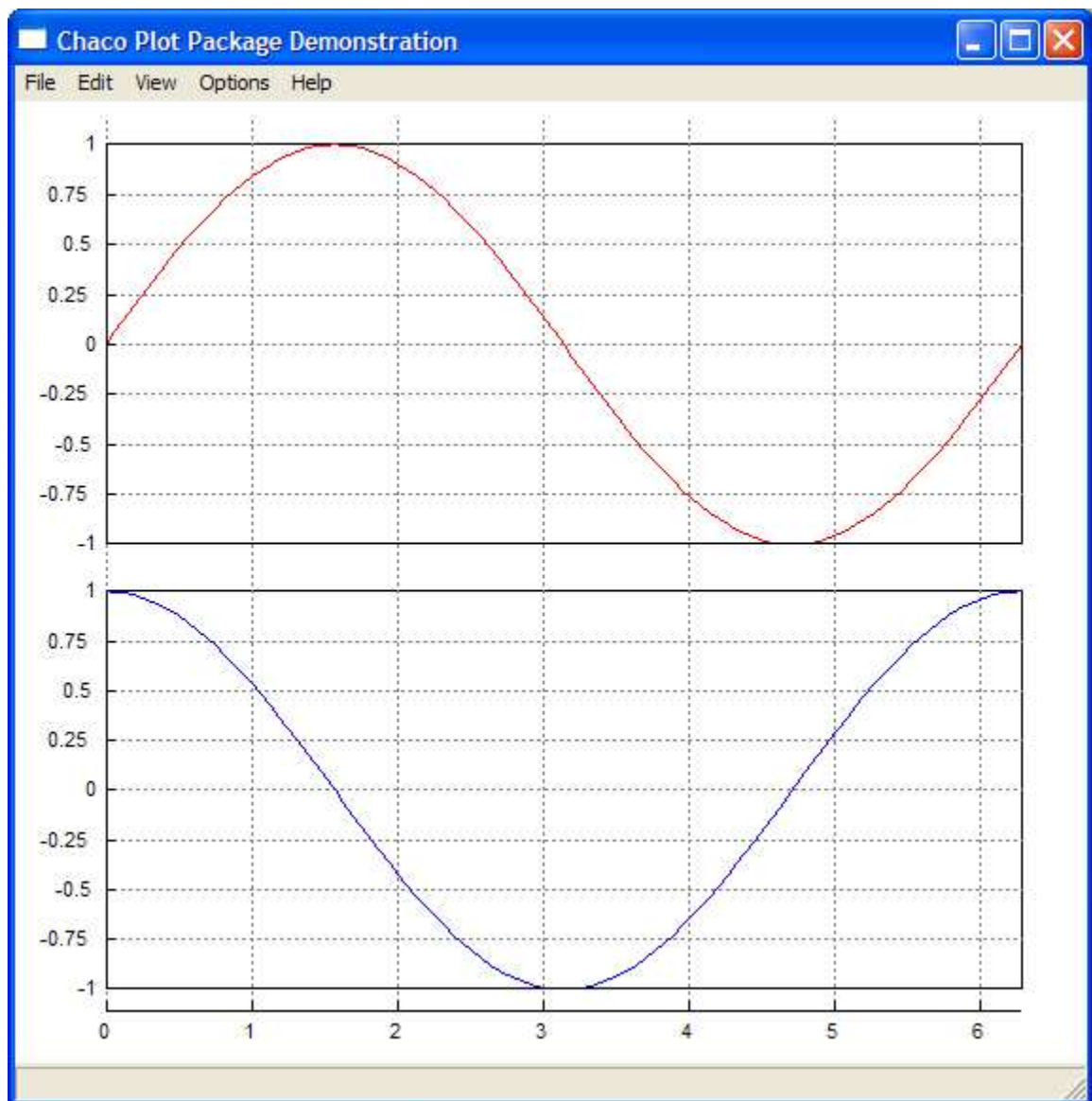
The PlotIndexed Class

Another useful class that sits somewhere between the **PlotGroup** and **PlotCanvas** classes in terms of its layout and drawing capabilities is the **PlotIndexed** class. For the most part, a **PlotIndexed** object behaves very much like a **PlotGroup** object, laying out its contained objects in either a horizontal or vertical orientation. But unlike a **PlotGroup** object, which does not know anything about plotting, a **PlotIndexed** object understands plotting, and in fact factors out and coordinates the drawing of common index axes across all of its contained plottable objects.

What is an index axis? Because Chaco can plot data in any orientation, it would be awkward to talk in terms of the more standard mathematical *x* and *y* axes, since with a simple property change plots can easily be rotated 90 degrees or have the direction in which data is plotted be reversed. Instead, Chaco refers to a plot's *index* and *value* axes,

where *value* describes the set of data being plotted, and corresponds in the normal case to the y axis, and *index* describes the set of points at which the data was measured, and corresponds in the normal case to the x-axis.

So a **PlotIndexed** object factors out the common set of points at which various sets of data were measured and coordinates the drawing of all of the plottable objects so that they share a common index axis, as illustrated below:



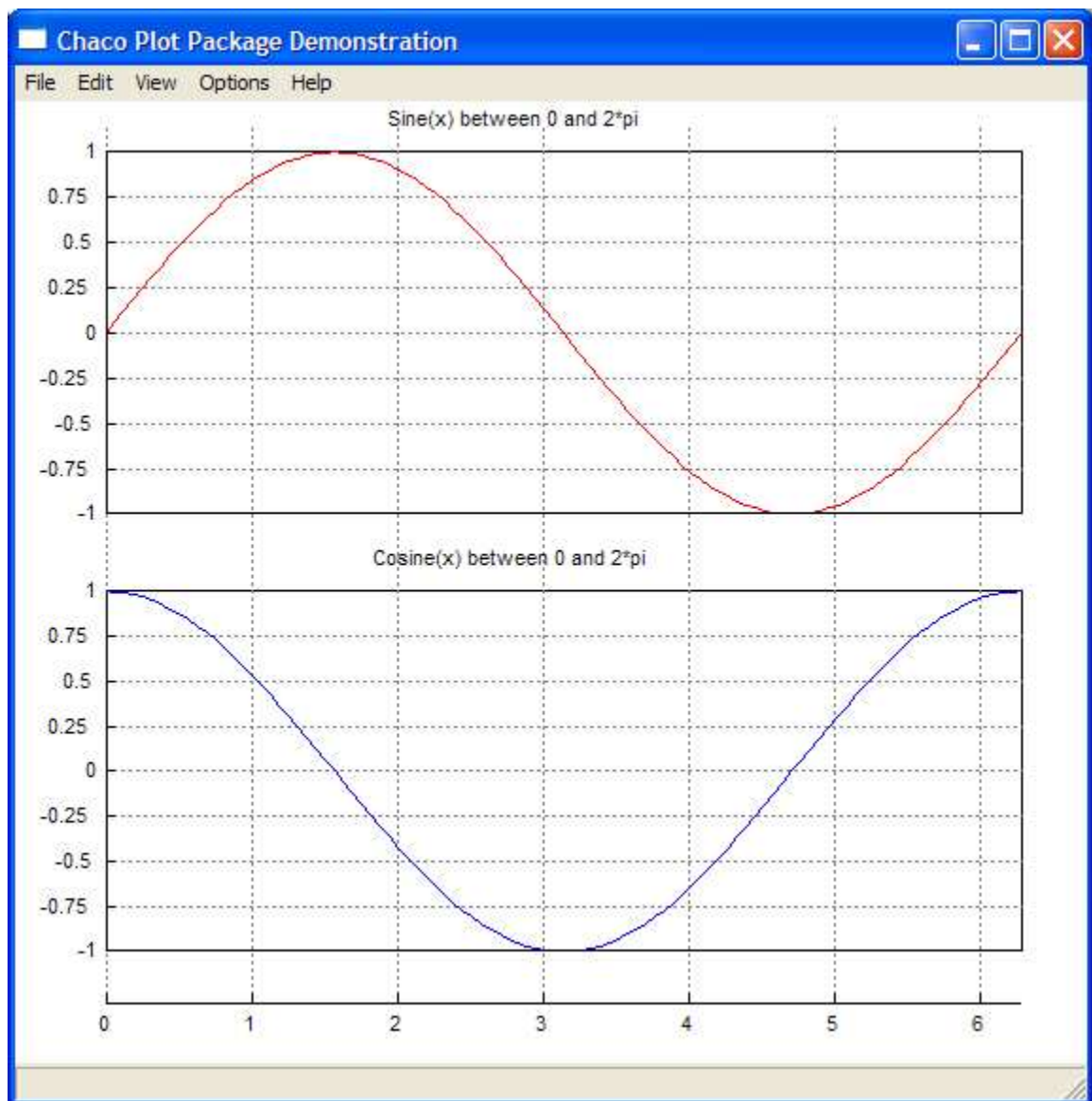
Notice how in this case there is a single index (i.e. x) axis drawn along the bottom of the plot, and that the grid lines span the entire distance from top to bottom of the plotting area. **PlotIndexed** objects are very useful in cases where there are several sets of data, derived from a common set of measurement points, that need to be plotted together, but which may be too visually cluttered when drawn on a common **PlotCanvas** object.

The PlotAxis Class

A **PlotAxis** object describes the visual and layout characteristics of a plot axis. It controls such characteristics as the minimum and maximum plot bounds, the color and line styles used for the axis tick marks and grid lines, label formatting and spacing, and so on.

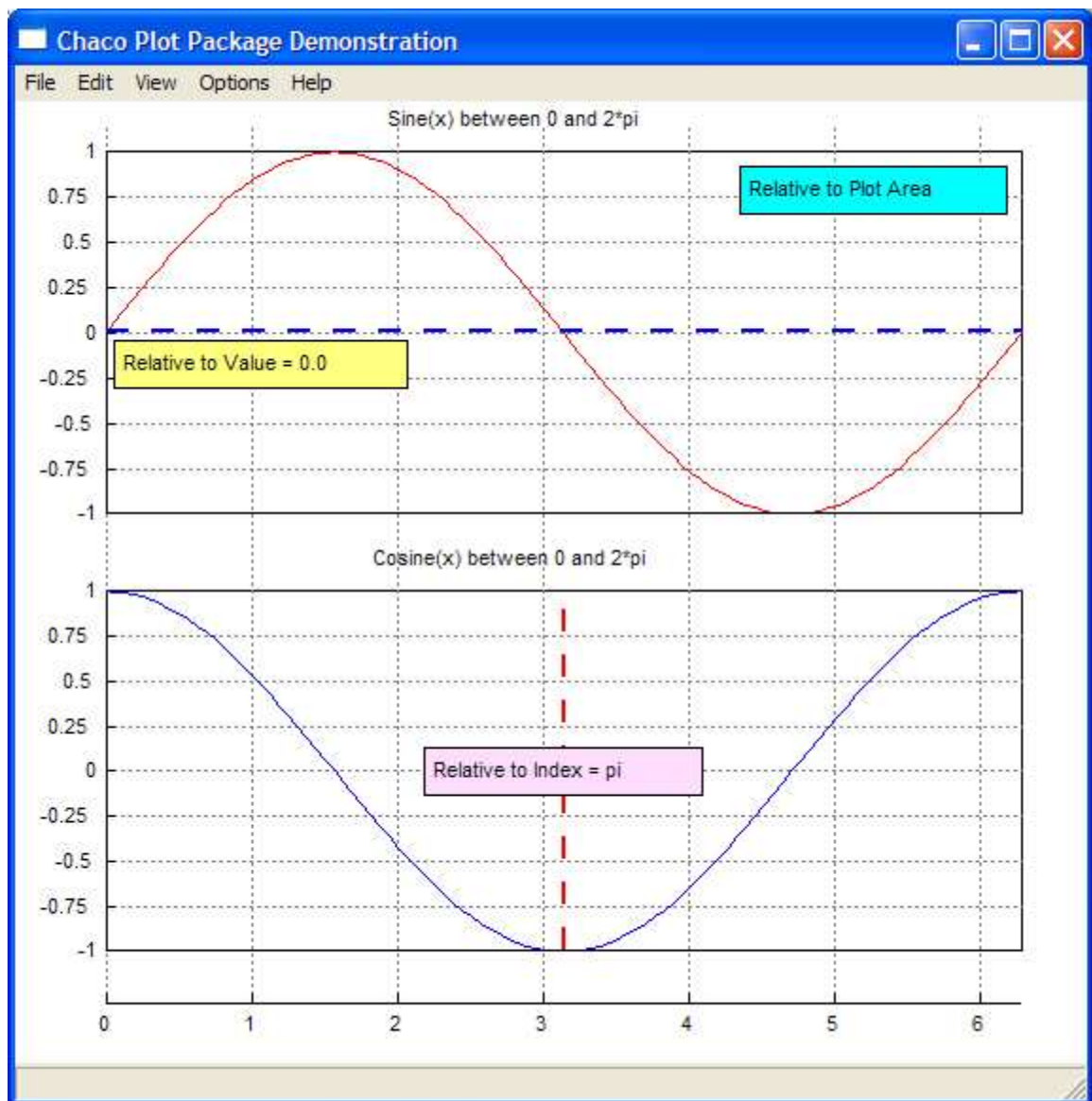
The PlotTitle Class

A **PlotTitle** object provides a way to add titles to any of the top-level Chaco plotting objects (i.e. **PlotGroup**, **PlotIndexed**, **PlotCanvas** and **PlotValue**). Any number of titles can be added to any of the four sides of any of these objects. The following shows the result of adding a **PlotTitle** object to the top of each of the **PlotValue** objects from the preceding figure:



The PlotOverlay Class

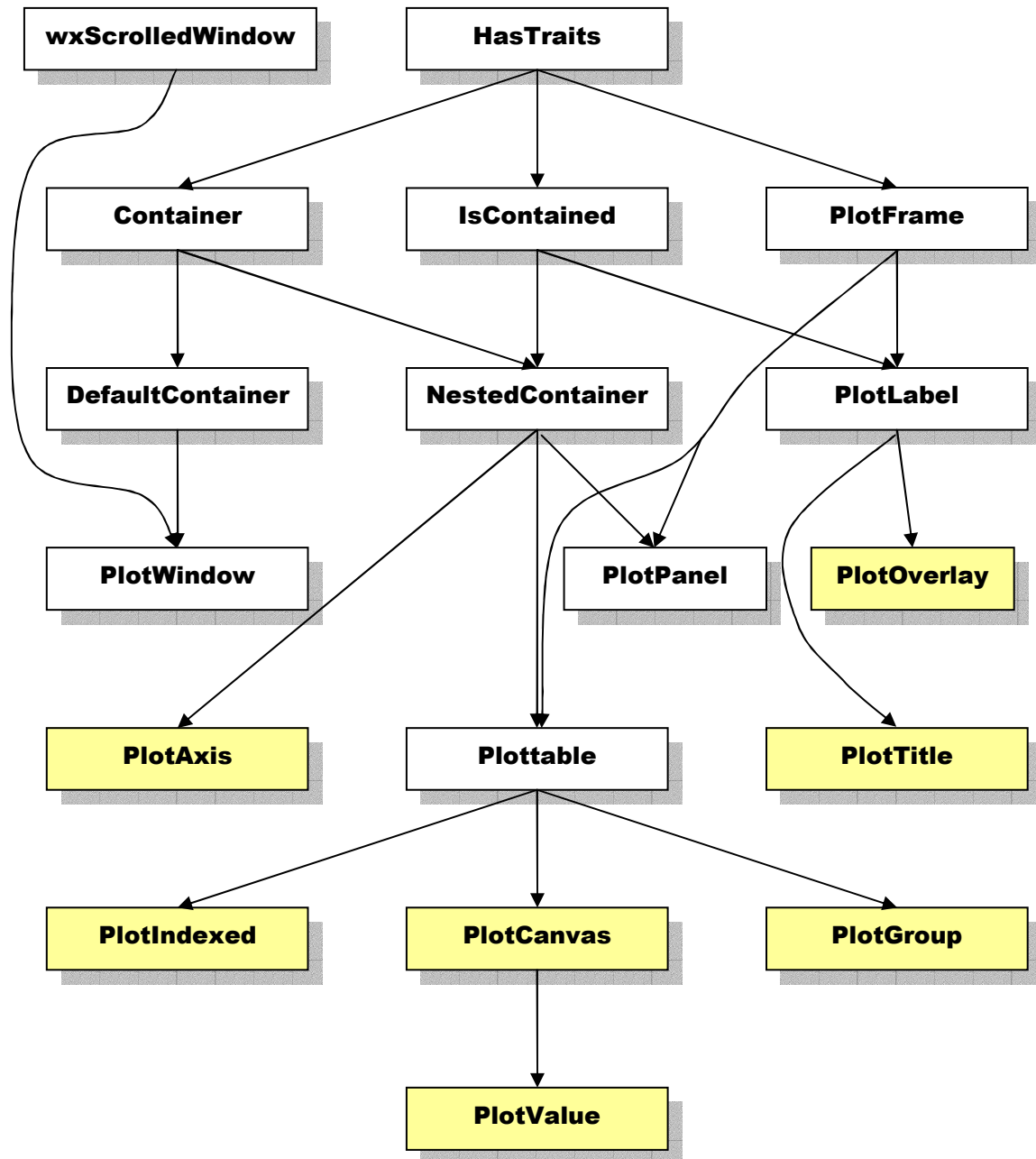
A **PlotOverlay** object is similar to a **PlotTitle** object, but instead of being drawn outside the plot area, it overlays a section of the plotting area. A **PlotOverlay** object can either be positioned relative to the plotting area (e.g. 'top left') or relative to a particular **PlotValue** index or data value (or both). This is illustrated in the following figure, which shows examples of the different types of **PlotOverlay** objects:



The Chaco Class Hierarchy

So far we have provided a brief overview of most of the high-level Chaco plotting objects. The following figure illustrates these same classes in terms of their underlying Python class hierarchy:

Chaco Class Hierarchy



Note that there are a number of classes that have not been mentioned so far. Some, like **Plottable** and **PlotFrame**, are base classes which define many of the common visual characteristics of the Chaco plotting objects we have seen in the preceding figures.

Others classes, like **IsContained**, **NestedContainer**, **Container** and **DefaultContainer** help define one of the other key architectural characteristics of Chaco: its container/delegation model.

The Chaco Container Model

All of the Chaco objects we have talked about so far can be characterized as objects that can be contained in other objects, or that can contain other objects, or both. The **IsContained** class is the base class for any Chaco object that can be contained in another object. The **NestedContainer** class is the base class for objects that can contain other objects. Since **NestedContainer** is a subclass of **IsContained**, objects derived from **NestedContainer** can both contain, and be contained in, other objects. Any object that can be contained in another object has an attribute called **container** that holds the reference to its container object.

The container model is used for two related, yet distinct, purposes in Chaco:

1. Containment is used to provide a visual hierarchy for Chaco objects. This is evident in the way that objects contained in **PlotGroup**, **PlotIndexed** and **PlotCanvas** objects are organized and drawn on the screen for example.
2. An object's container also provides a source of default values, such as line color, font, or axis orientation. If a particular object property is not explicitly set for an object, its value is automatically retrieved from its containing object. If the containing object does not have an explicit value set, the value is retrieved from its container, and so on, until a containing object which has an explicit value for the property is reached.

This second use of containment is referred to as delegation, and is based entirely on the **traits** package (refer to the “Traits: A New Way of Adding Properties To Python Classes” paper for more information on the **traits** package).

Nearly every Chaco object property (i.e. *trait*) is defined using the delegation model. That is, every trait has a value defined either in the object itself, or in one of the container objects in its containment hierarchy. Note that the delegation chain is always guaranteed to terminate (that is, reach an object which has an explicit value set for the trait). This is true because of the **DefaultContainer** class, which defines explicit default values for every Chaco object trait.

The default value of each Chaco object's **container** trait is an instance of a **DefaultContainer** object. When a Chaco object is added to another Chaco container object, its **container** trait is modified to refer to its new container object. The topmost object in a Chaco object hierarchy is normally associated with a **PlotWindow** object. Referring to the previous class hierarchy figure, we see that **PlotWindow** is a subclass of **DefaultContainer**. Thus we are guaranteed that the delegation process always terminates in an instance of **DefaultContainer**, either the default one, or an instance of **PlotWindow**.

The delegation model is very useful because it allows global object defaults to be set on a top-level Chaco object and be automatically applied to all contained Chaco objects. Yet it still allows individual traits to be overridden as needed on any particular object. And because it is a hierarchy, defaults can be established at any level that apply to all subordinate objects.

The Trait Model

Nearly every characteristic of each Chaco object is defined using the **traits** package. This provides two major benefits:

1. It allows object properties to be defined in terms of the powerful containment and delegation model just described.
2. It provides a flexible, user friendly, yet type safe, method of setting object properties.

As an example of the flexibility provided by using traits to define object properties, consider the following example, which shows three different ways that a user can set the color of a **PlotTitle** object's text:

```
title = PlotTitle( 'My Plot' )
title.color = 'blue'           # Standard color name
title.color = 0x0000FF        # Hexadecimal RGB value
title.color = (0.,0.,1.,1.)   # Floating point (R,G,B,Alpha)
```

Any of these are valid methods of setting the text color to blue. Note however that the following statement would result in an exception being generated:

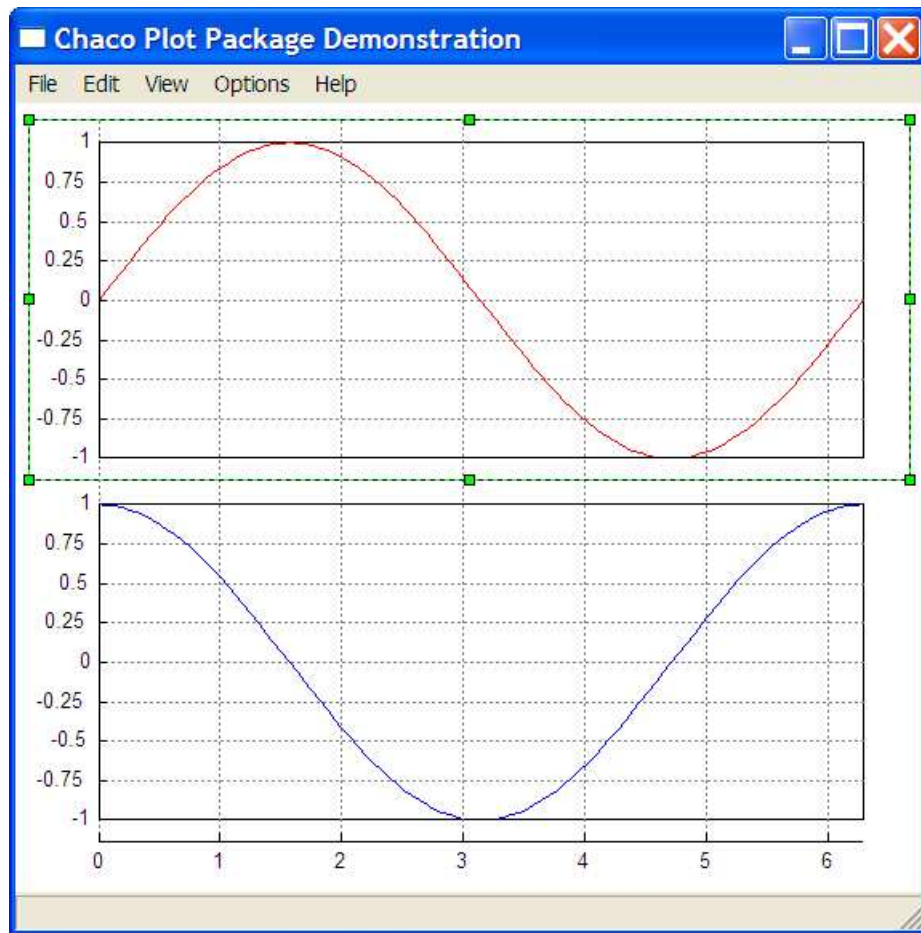
```
title.color = 'pretty in pink'
```

because the string 'pretty in pink' is not recognized as one of the valid values for a Chaco color.

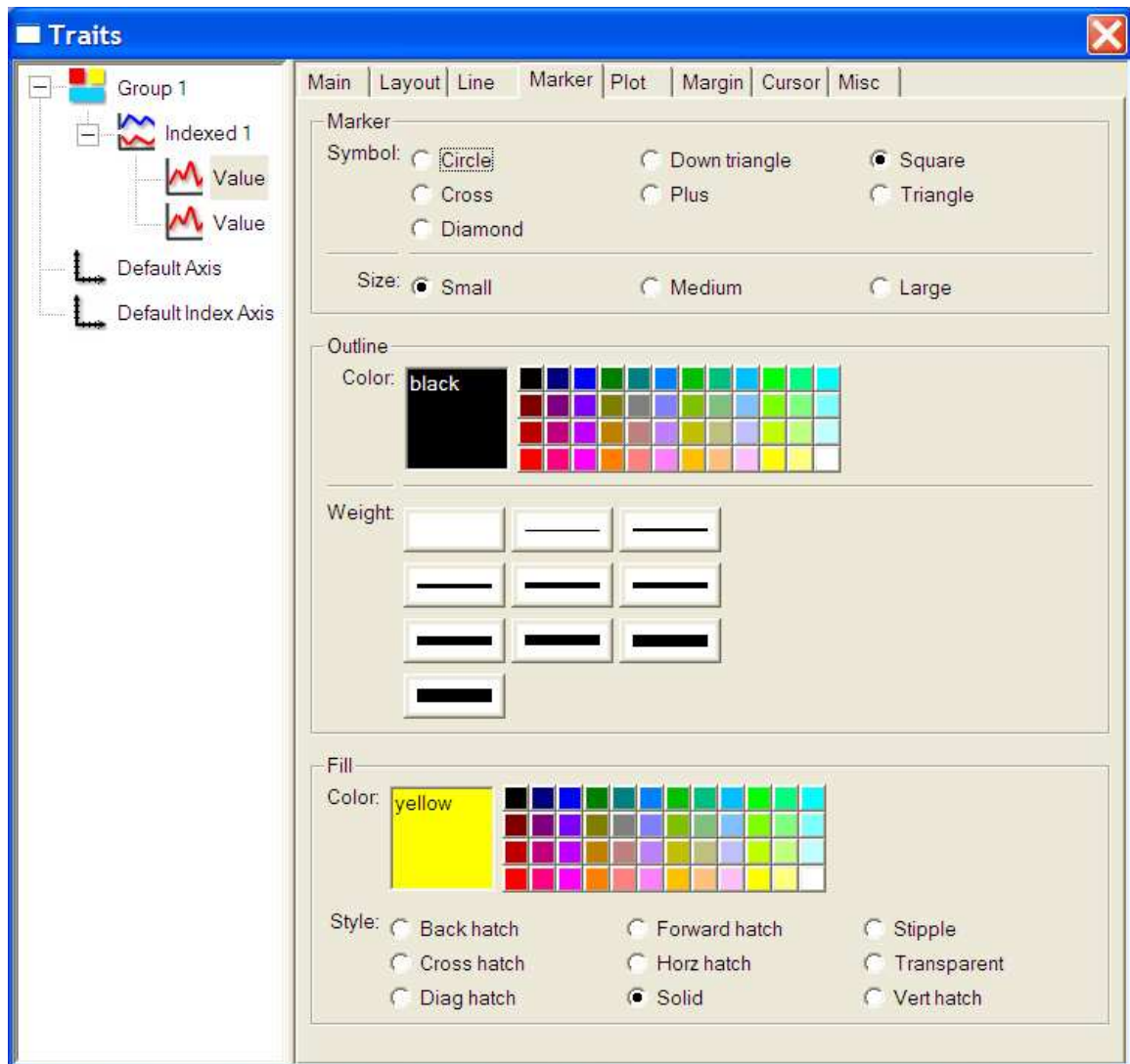
Trait Editors

Another benefit of using the **traits** package is the ability to easily generate high-quality, multi-GUI (wxPython and Tkinter currently) trait editors with very little, or no, effort.

One of the optional features of the Chaco package is the ability to allow the user to visually select and edit the properties of any Chaco object interactively. The following figure shows one of the previous examples with one of the **PlotValue** objects selected:



The next figure shows one page from the trait editor displayed by double-clicking on the selected object:



Changing any of the trait values in the editor is immediately reflected in the displayed **PlotValue**'s on-screen representation.

Note also that the trait editor displays the complete hierarchy of all Chaco objects contained in the window. New objects can be selected either by clicking on their label in the hierarchy view or by clicking on the object itself in the plot window. In either case, the editor notebook on the right side of the dialog immediately reconfigures itself to display the newly selected object's traits.

Plot Renderers

Up until now, each of the plots displayed has been a simple line plot of the associated data. While this is probably the most common type of plot for many scientific and engineering applications, a plot package that supported only one type of plot would not be very useful. To that end, Chaco supports an open-ended plotting architecture based on the notion of *plot renderers*.

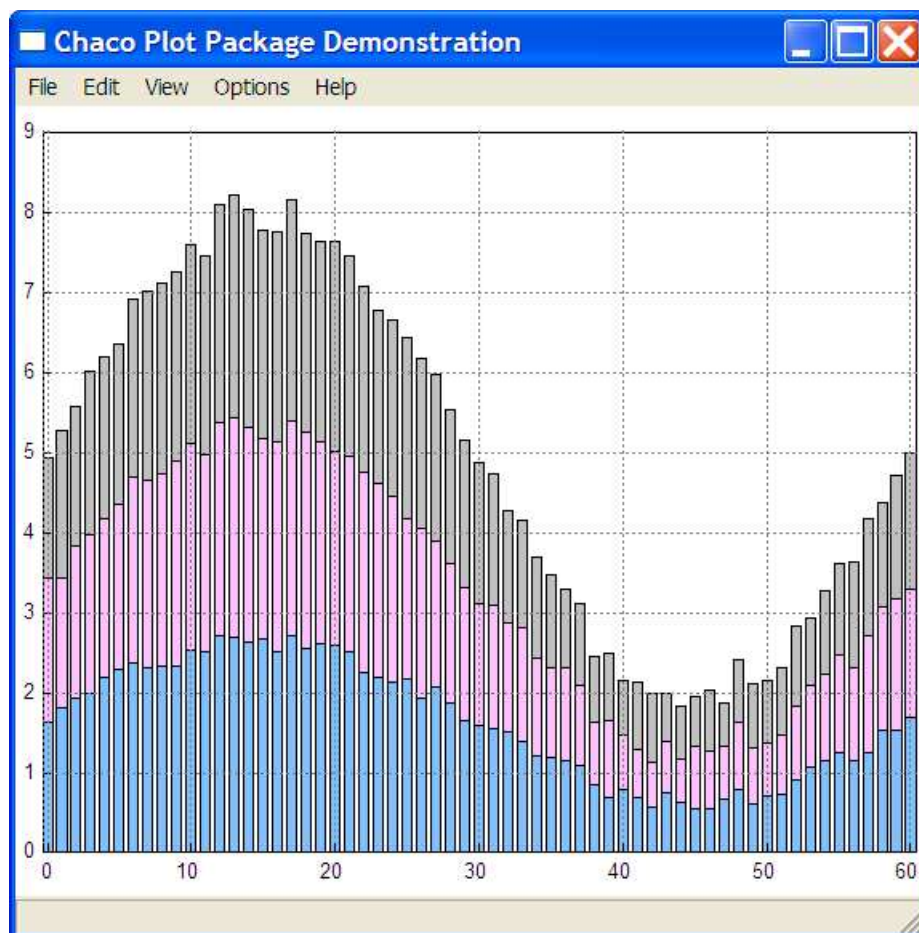
Each **PlotValue** has a **plot_type** trait that defines a set of one or more plot renderers that will be used to render (i.e. draw) the associated **PlotValue**'s data. The default value of this trait is 'line', which specifies the line rendering style we have seen so far in all of the examples.

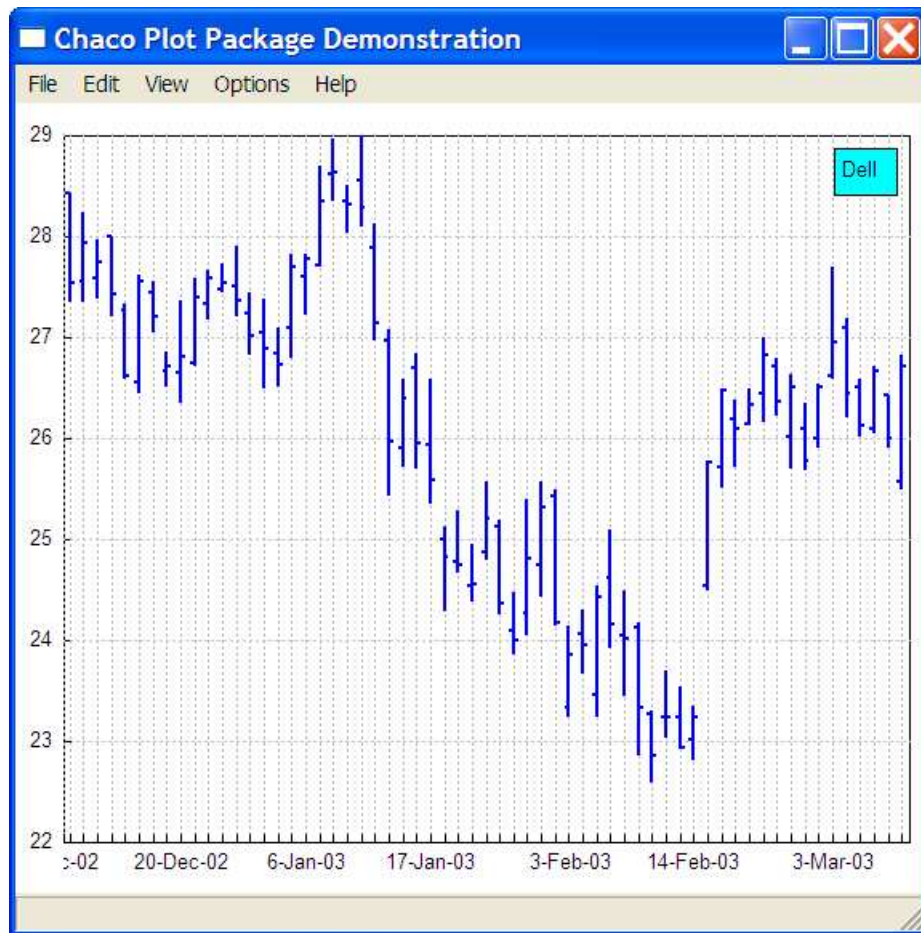
However, the following are some examples of other values that could be set:

```
value = PlotValue( ... )
value.plot_type = 'scatter'          # Scatter plot
value.plot_type = 'scatter,line'    # Both scatter and line
value.plot_type = [ 'scatter', 'line' ] # Same thing
```

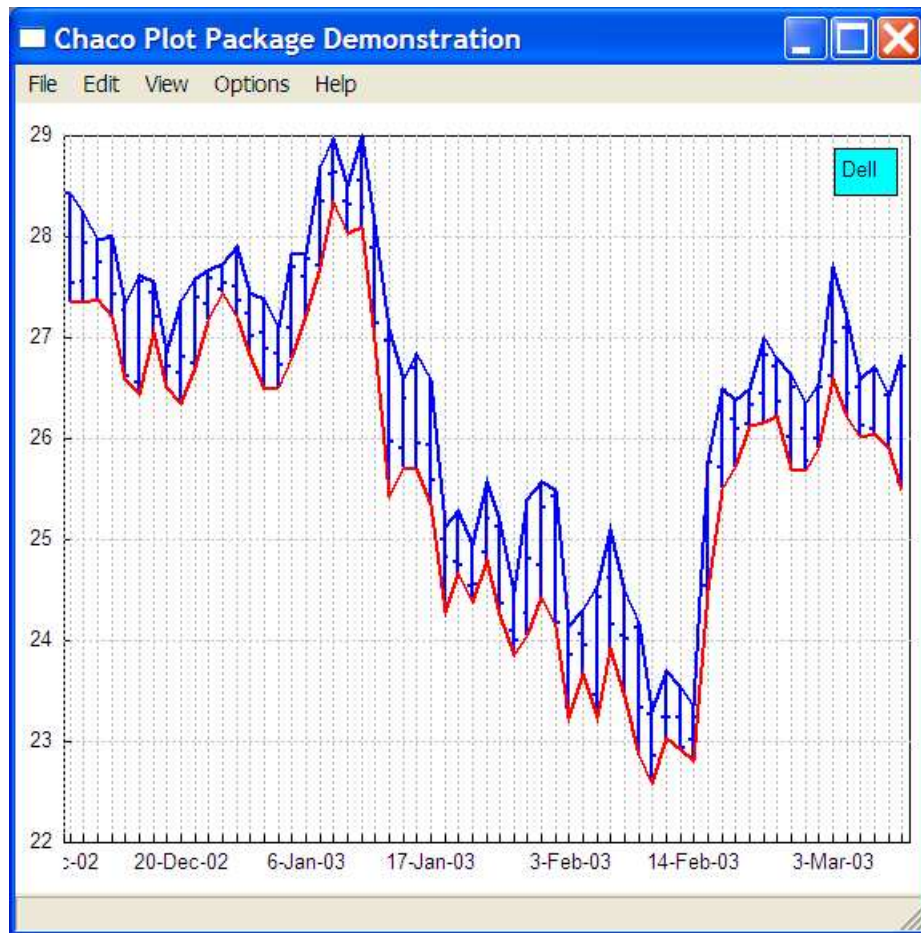
Note that more than one type of plot renderer can be specified. Also, new types of plot renderers can be added to Chaco simply by subclassing the **PlotRenderer** base class.

Another interesting feature of the plot renderer architecture is that a given **PlotRenderer** subclass can plot several related sets of data at the same time. Examples of this type of plot are *stacked bar charts* and stock *HLOC* (high/low/open/close) diagrams, as illustrated in the following figures:





As mentioned previously, more than one plot renderer can be associated with a **PlotValue**, as illustrated in the following figure, which combines the HLOC renderer shown above with the standard line renderer:



PlotData and PlotSource Objects

As mentioned previously, a **PlotValue** object can be logically viewed as simply an array of values to be plotted. While this is a very convenient model in many cases, especially when working interactively in a Python command shell, it is not always the best model when working with more structured data, such as the stock high, low, open, close data we used in the previous example.

Accordingly, Chaco also provides a means of dealing with more structured forms of data through the use of **PlotData** and **PlotSource** objects. A **PlotData** object is simply a wrapper around one or more sets of related data. The data contained in the **PlotData** object can be connected to a plottable Chaco object by means of a **PlotSource** object, which acts as a conduit between a **PlotValue** and a particular set of data contained within a **PlotData** object.

A good example of a **PlotData** object is the **StockQuote** class we used in the previous example. A **StockQuote** object contains a particular stock's open, close, high, low and volume data over a specified range of dates. The data is obtained dynamically over the Web whenever the **StockQuote** object's **symbol** trait is modified:

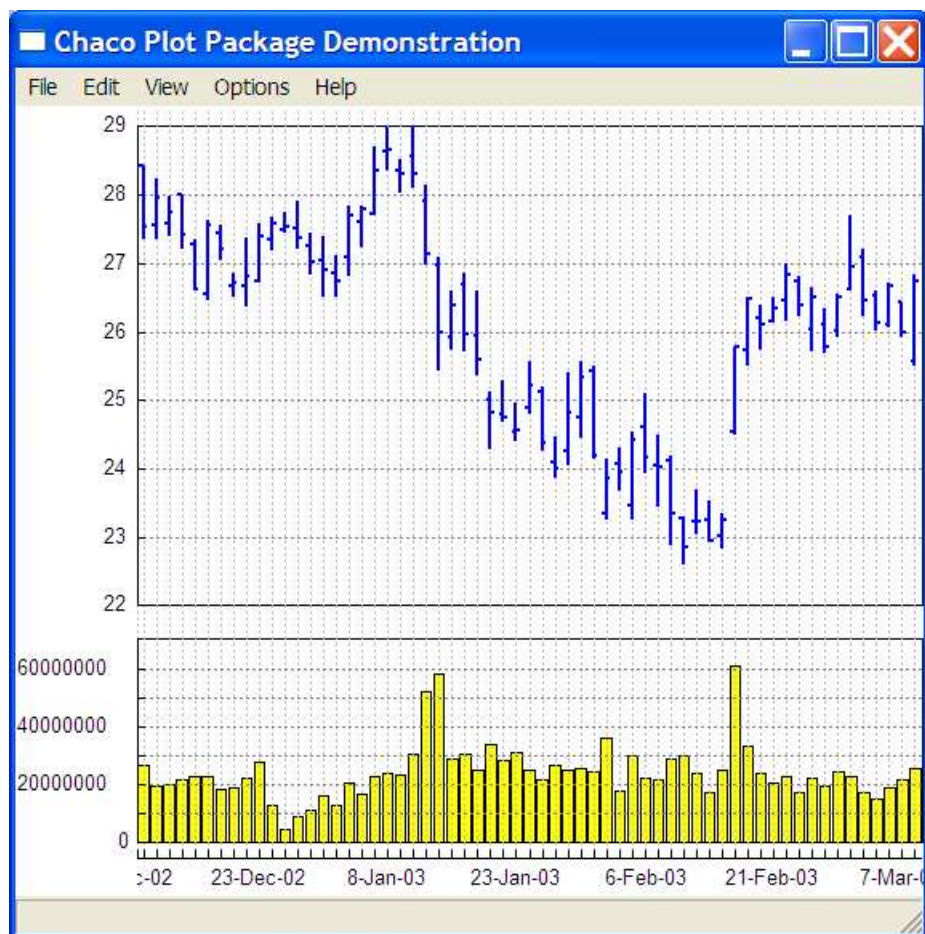
```
sq = StockQuote()  
sq.symbol = 'intc' # Fetches new values for symbol 'intc'
```

A **StockQuote** object also exposes five sources of data: **high**, **low**, **open**, **close**, and **volume**, any of which can be associated with one or more **PlotValue** objects. **PlotData** objects can also be *smart* about how they add themselves to a plottable object. For example, the preceding stock quote examples showing the HLOC plots were constructed using a statement similar to the following:

```
canvas.add( StockQuote( 'dell' ) )
```

In this case, the **StockQuote** object automatically connected its high, low, open, close data sources to the **PlotCanvas** object and set the **plot_type** trait of the canvas to 'HLOC' to select the correct default plot renderer for its associated data.

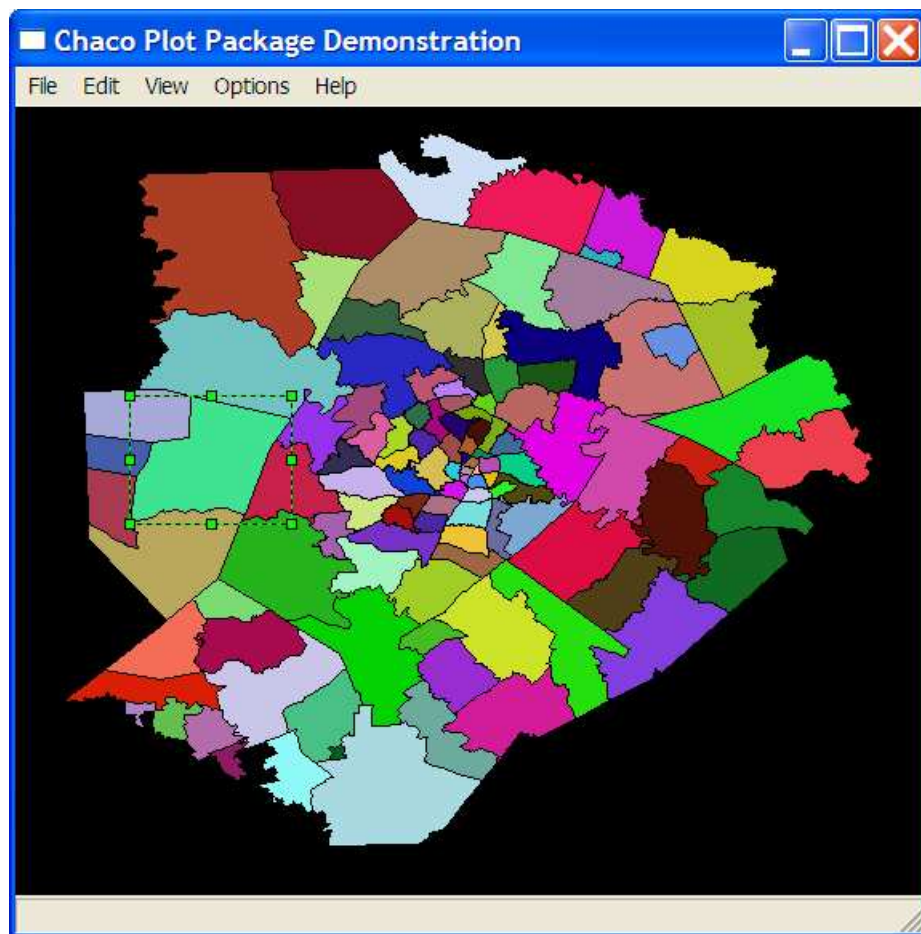
We can also explicitly connect a **PlotData** object source to a particular plottable object directly, as illustrated in the following figure:



where we have explicitly connected the ‘volume’ source of the **StockQuote** object to the bottom **PlotCanvas** using a statement like:

```
sq.add( canvas, 'volume' )
```

Another example of the usefulness of the **PlotData** capability of Chaco is shown in the following figure, which was constructed using an instance of the **ArcView PlotData** subclass:



An **ArcView** object loads GIS data in **ArcView** format from a data file and provides each of the polygons it contains as a separate data source. When we execute a statement like the following:

```
canvas.add( arcview_object )
```

the **ArcView** object (i.e. **arcview_object**) adds each of its polygons to the canvas as a separate **PlotValue** object and sets the **PlotCanvas** object’s **plot_type** trait to ‘polygon’, which is the appropriate plot renderer for this type of data.

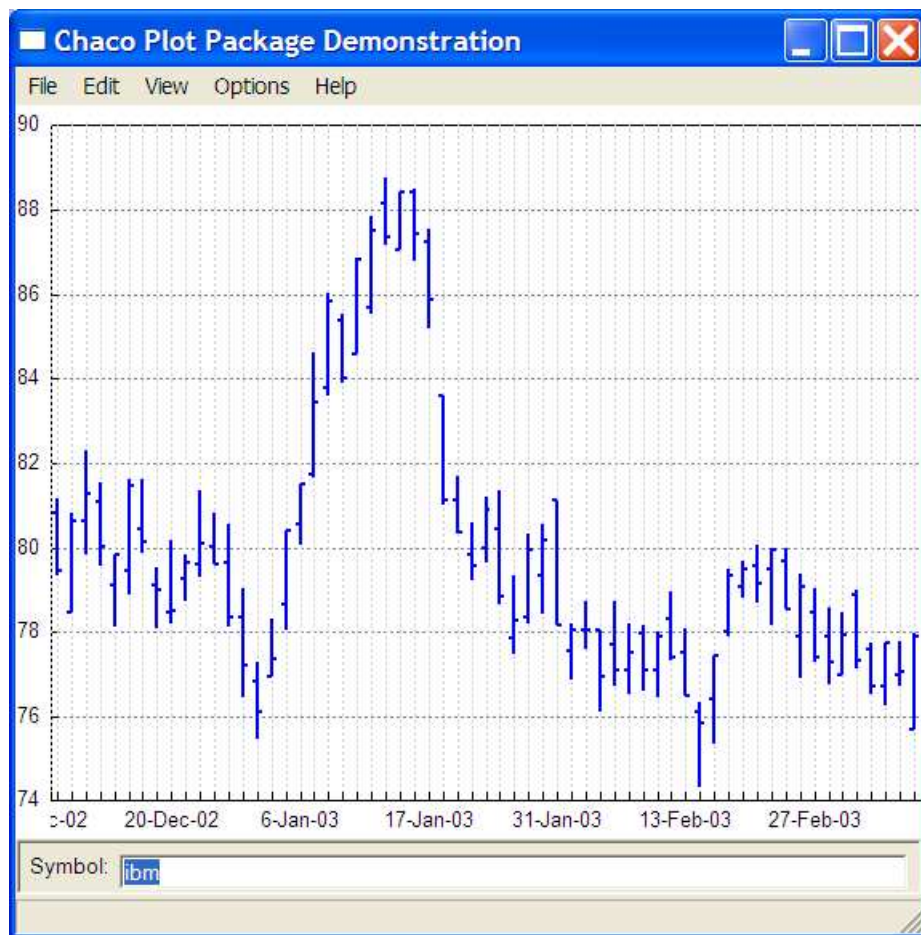
The fill colors for this example were set randomly as follows:

```
for value in canvas[0]:  
    value.fill_color = (random(), random(), random(), 1.0)
```

Note also that one of the polygons (i.e. **PlotValues**) on the canvas has been selected in the figure. Double-clicking the polygon would display its trait editor, allowing the user to interactively change any of the object's properties, such as selecting a new fill color.

PlotPanel Objects

Another useful feature of Chaco is its ability to imbed panel's containing various useful widgets within a **PlotWindow**. An example of this is shown in the following figure:



This view was constructed using the following statements:

```
sq = StockQuote( symbol = 'ibm' )  
group.add( PlotCanvas( sq ) )  
group.add( PlotPanel( sq ) )
```

When added to a **PlotCanvas**, the **StockQuote** object binds its **open**, **close**, **high** and **low** data sources to **PlotValue** objects added to the canvas. When added to a **PlotPanel**, a trait editor for the **StockQuote** object's traits is automatically constructed and imbedded within the panel.

Furthermore, the resulting Symbol field is live. Typing a new stock symbol into the text entry field results in a new set of stock values being displayed. It should be further noted that there is no explicit code in the implementation of the **StockQuote** class to do this.

This latter point is illustrated in the following example, which shows the code for a simple **Polynomial** class for displaying fifth degree polynomials:

```
class Polynomial ( PlotData ) :

    coefficient = Trait( 0.0, TraitRange( -10.0, 10.0 ) )

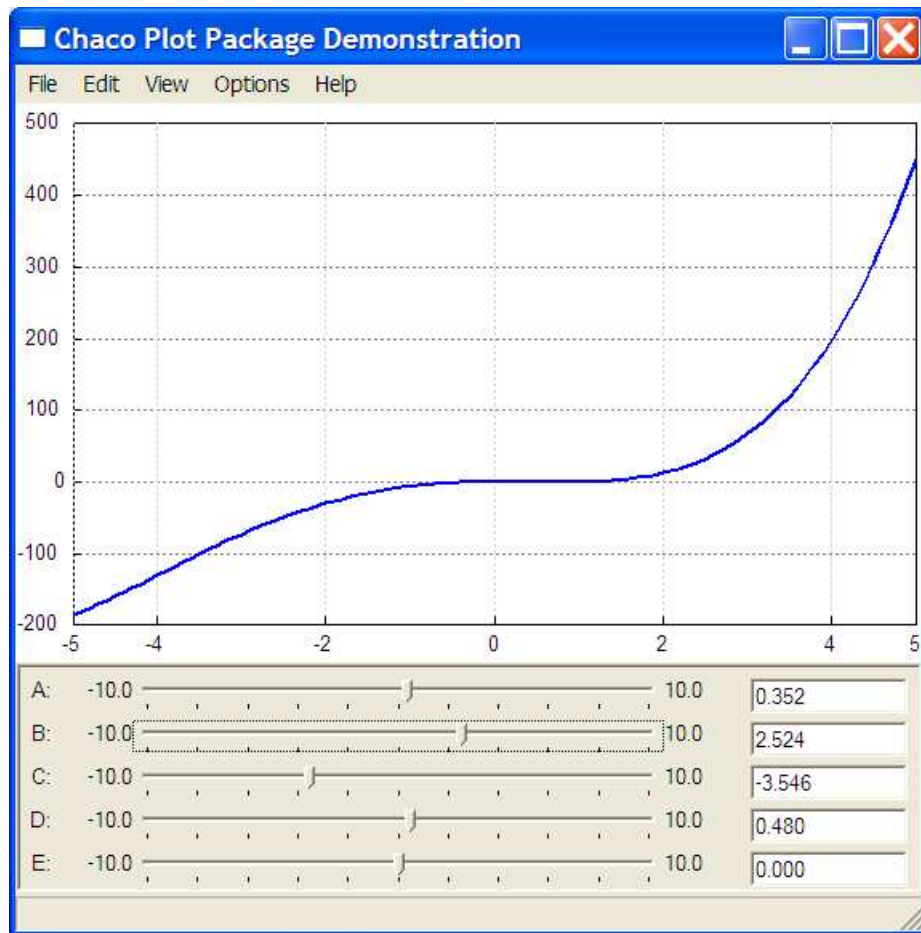
    __traits__ = {
        'a': coefficient,
        'b': coefficient,
        'c': coefficient,
        'd': coefficient,
        'e': coefficient
    }

    def data ( self, source_name ) :
        x = arange( -5.0, 5.01, 0.1 )
        y = ((self.a * (x ** 4)) +
              (self.b * (x ** 3)) +
              (self.c * (x ** 2)) +
              (self.d * x) + self.e)
        return transpose( reshape( concatenate( ( x, y ) ),
                                   ( 2, len( x ) ) ) ) )
```

With the exception of a few **import** statements, this is the entire definition of the **Polynomial** class. We can now construct a view containing an instance of this class using code very similar to the preceding example:

```
poly = Polynomial()
group.add( poly )
group.add( PlotPanel( poly ) )
```

This results in a window looking something like the following:



This figure shows the results achieved after adjusting several of the panel sliders, each of which corresponds to one of the polynomial coefficients. Again, it is interesting to note that there is no code in the **Polynomial** class that deals explicitly with changes to the polynomial coefficients. All of the “magic” occurs under the covers, courtesy of the **traits** package.

Kiva

Chaco implements all of its plotting capabilities using **kiva**, a Display PostScript based drawing package that supports multiple output types. As a result of this, any composition of Chaco objects can be rendered to any of the following output formats:

- wxPython or Tkinter based display screen windows
- PDF
- PostScript
- Raster images such as JPEG, PNG or GIF

This flexibility in output formats makes Chaco potentially useful in a wide variety of applications, including:

- Ad-hoc plots performed from a Python interpreter shell.
- Interactive wxPython or Tkinter based applications requiring plotting capabilities.
- On-demand production of plot related images by a web server CGI script.
- Compositing of plots with other forms of text and graphics within a PDF or PostScript based report generator.

Conclusion

Chaco is a fairly young package with a lot of work still remaining to be done. Hopefully however, this paper has demonstrated that even at this early stage in its development, Chaco has several architectural and implementation features that will in the future make it a versatile and useful plotting package for scientists and engineers.

In particular, some of the key features we have demonstrated are:

- A visual containment model that allows composing plot objects in a wide variety of ways.
- A flexible and user friendly set of object properties based on the traits package.
- A powerful property delegation model based on the visual containment hierarchy, also implemented using the traits package.
- An open-ended plot rendering architecture that allows new types of plots to be easily added to the package.
- A useful abstraction model that allows external data sources to be easily defined and plotted, as demonstrated by the **StockQuote** and **ArcView** examples.
- A simple to implement and use architecture for adding interactive control widgets to a plot, based on the traits package, and illustrated by the **StockQuote** and **Polynomial** examples.

Where To Get The Code

The **Chaco** package is available as part of the **SciPy** (*Scientific Python*) open source project at www.scipy.org and is covered by a BSD-style license. All of the code described in this paper is contained in the **chaco** project in the www.scipy.org CVS repository.

The CVS repository can be accessed from a command shell using the commands:

```
> cvs -d:pserver:anonymous@scipy.org:/home/cvsroot login
> cvs -d:pserver:anonymous@scipy.org:/home/cvsroot co
chaco_all
```

Note the use of **chaco_all** as the CVS module name. Checking out this module ensures that all of the Chaco code and its supporting packages, such as **kiva** and **traits**, are copied from the CVS repository.

The Chaco source code can also be browsed on-line using the **ViewCVS** facility. There is also a Chaco page at **http://www.scipy.org/site_content/chaco** that contains recent Chaco source tarballs and a Windows installer program.