Interactive Plotting with Chaco

Peter Wang
Enthought, Inc.

Scipy 2008 Advanced Tutorial Track
August 20, 2008

About this tutorial

Recommended prerequisites:

- Know a little numpy
- Know a little bit about GUI programming
- Know a little bit about Traits and Traits UI (from previous tutorial)

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Today's tutorial is an introduction to Chaco. We're going to build several mini-applications of increasing capability and complexity. Chaco was designed to be primarily used by scientific programmers, and this tutorial only requires basic familiarity with Python.

Knowledge of numpy can be helpful for certain parts of the tutorial. Knowledge of GUI programming concepts is helpful (e.g. widgets, windows, events).

Also, today's tutorial will showcase using Chaco with Traits UI, so knowledge of Traits is helpful. If you caught even just the basics of the previous tutorial on Traits, you should be fine. It's worth pointing out that you don't *have* to use Traits UI in order to use Chaco - you can integrate Chaco directly with Qt or wxPython - but for this tutorial, we will be using Traits UI to make things easier.

Goals

By the end of this tutorial, you will have learned how to:

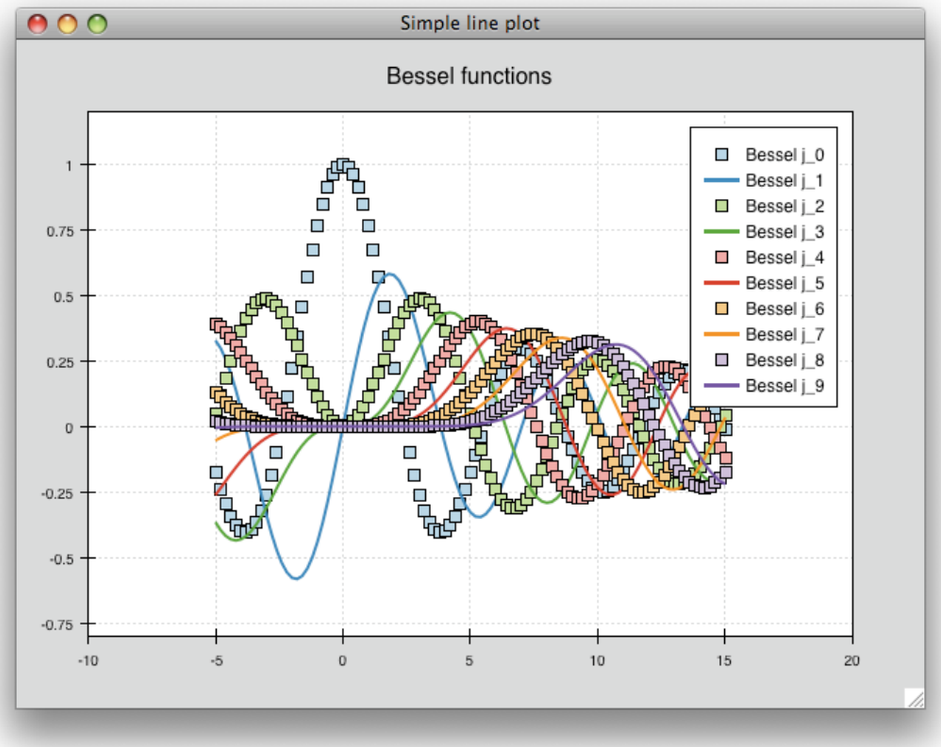
- create Chaco plots of various types
- arrange plots of data items in various layouts
- configure and interact with your plots using Traits UI
- create a custom plot renderer
- create a custom tool that interacts with the mouse

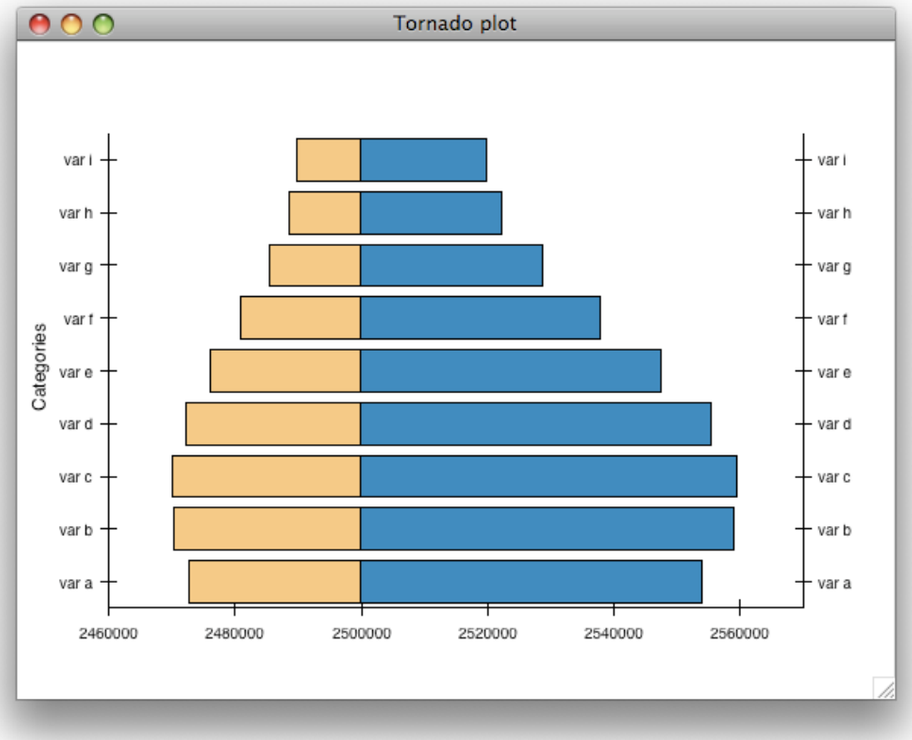
ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

What I'm hoping to achieve today is that by the end of this tutorial, everyone here will have some idea about how to do all of the things listed here.

Introduction

- Chaco is a *plotting application toolkit*.
- You can build simple, static plots





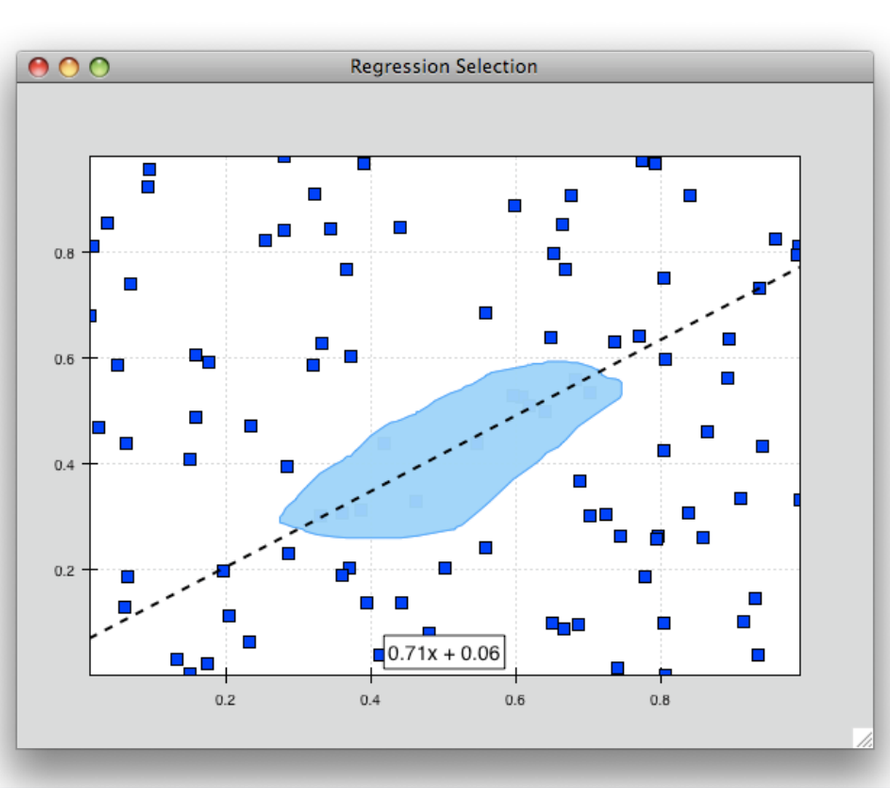
ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

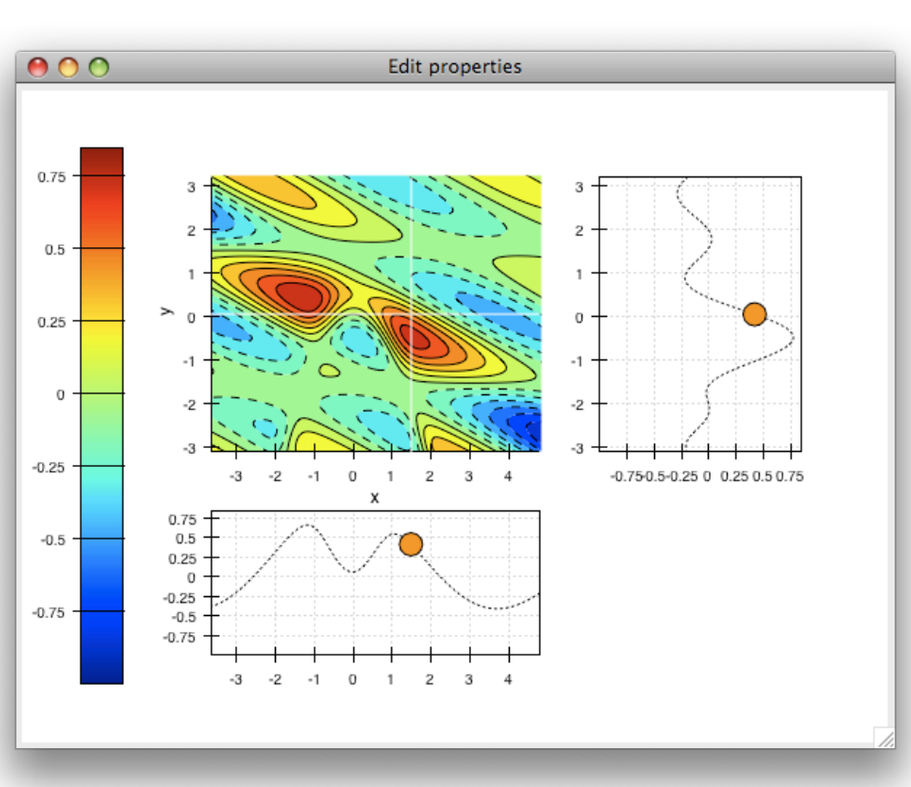
So, first off, what is Chaco and what makes it different?

Chaco is a toolkit for building plots and plotting applications. You can build simple, static plots like the ones shown here...

Introduction

- Chaco is a *plotting application toolkit*.
- You can build simple, static plots
- You can also build rich, interactive visualizations:





...but you can also build interactive visualization applications that let you explore your data.

I'm going to take a few minutes now to show you what I mean by "interactive visualization". These are all examples that ship with Chaco or that you can download off of the project website. I'll talk a little bit about what characteristics of Chaco each example highlights, but I'm not going to get into too much detail right now.

(The following examples are all standard Chaco examples)
- multiaxis.py
- two_plots.py
- data_labels.py: pan & zoom, data label movement
- regression.py
- line_plot_hold.py
- inset_plot.py: can move the smaller plot around, linked axes
- cmap_scatter.py
- zoom_plot.py
- traits_editor.py
- data_cube.py
- scalar_image_function_inspector.py
- spectrum.py

“Script-oriented” Plotting

```
from numpy import *
from enthought.chaco.shell import *

x = linspace(-2*pi, 2*pi, 100)
y = sin(x)

plot(x, y, “r-”)
title(“First plot”)
ytitle(“sin(x)”)
show()
```

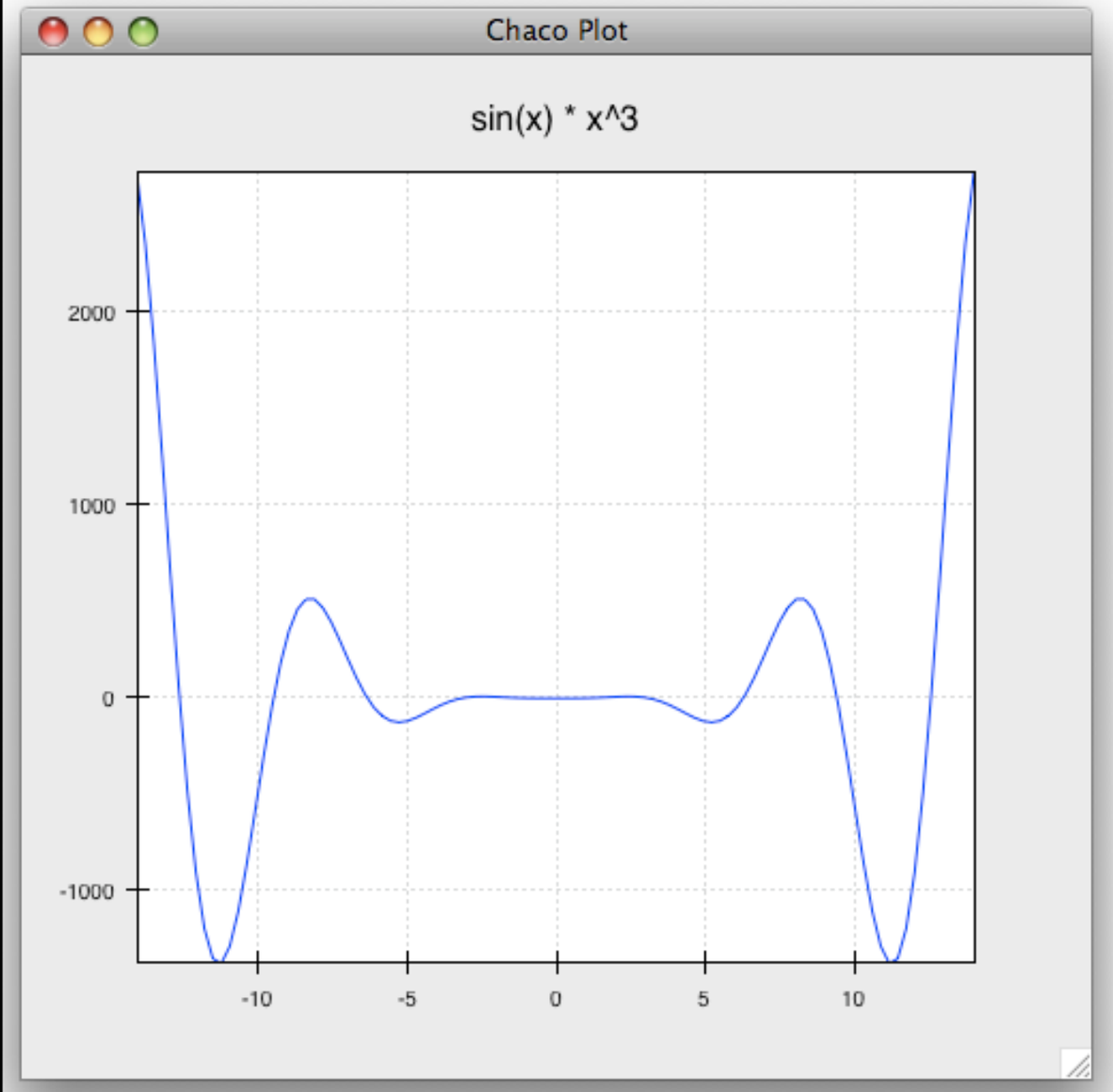
I distinguish between “static” plots and “interactive visualizations” because depending on which one you are trying to do, your code (and the code of the plotting framework you are using) will look very different.

Here is a simple example of what I call the “script-oriented” way of building up a static plot. The basic structure is that we generate some data, then we call functions to plot the data and configure the plot. There is a global concept of “the active plot”, and the functions do high-level manipulations on it.

Now, as it so happens, this particular example uses the chaco.shell script plotting package, so when you run this script, the plot that Chaco pops up does have some basic interactivity. You can pan and zoom, and even move forwards and backwards through your zoom history. But ultimately it’s a pretty static view into the data.

“Application-oriented” Plotting

```
class LinePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line",
                  color="blue")
        plot.title = "sin(x) * x^3"
        self.plot = plot

if __name__ == "__main__":
    LinePlot().configure_traits()
```



ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

The second approach to plotting is what I call “application-oriented”, for lack of a better term. There is definitely a bit more code, and the plot initially doesn’t look much different, but it sets us up to do more interesting things, as you’ll see later on.

So, this is our first “real” Chaco plot, and I’m going to walk through this code, and explain what each bit does. This example serves as the basis for many of the later examples.

(first_plot.py)

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


So we'll start off with the basics. You can see here that we are creating a class called "LinePlot", and it has a single Trait, which is an Instance of a Chaco "Plot" object.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


The LinePlot also has a Traits UI View declared on it. Inside the view, we are placing a reference to the "plot" trait, and telling Traits UI how we want that plot displayed. The attributes on the view are pretty self-explanatory, and the Traits UI manual documents all the various properties you can set here. But for our purposes, this Traits View is sort of boilerplate. It gets us a nice little window we can resize. We'll be using something like this View in most of the examples moving forward.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


Now, the real work happens in the constructor. The first thing we do is create some data just like in the script-oriented approach. But then, rather than calling a function to throw up a plot, we create this ArrayPlotData object and stick the data in there. The ArrayPlotData associates a name with a numpy array.

In a script-oriented approach to plotting, whenever you have to update the data or tweak any part of the plot, you basically re-run the entire script. Chaco's model is based on having objects representing each of the little pieces of a plot, and they all use Traits events to notify one another that some attribute has changed. So, the ArrayPlotData is an object that interfaces your data with the rest of the objects in the plot. In a later example we'll see how we can use the ArrayPlotData to quickly swap data items in and out, without affecting the rest of the plot.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


The next line creates an actual Plot object, and gives it the ArrayPlotData instance we created previously. The Plot object in Chaco serves two roles: it is both a container of renderers, which are what do the actual task of transforming data into lines and colors on the screen, as well a factory for instantiating renderers.

Once you get more familiar with Chaco, you can choose to not use the Plot object, and just directly create renderers and containers manually, but the Plot object does a lot of nice housekeeping for you.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


Next, we call the plot() method on the Plot object we just created. We want a blue line plot of the data items named “x” and “y”. Note that we are not passing in an actual array here. We are passing in names of arrays in the ArrayPlotData we created previously.

This method call actually creates a new object called a “renderer” - in this case, a LinePlot renderer - and adds it to the Plot.

Now, this may seem kind of redundant or roundabout to folks who are used to passing in a pile of numpy arrays to a plot function, but consider this: ArrayPlotData objects can be shared between multiple Plots. So, if you wanted several different plots of the same data, you don’t have to externally keep track of which plots are holding onto identical copies of what data, and then remember to shove in new data into every single one of those those plots. The ArrayPlotData acts almost like a symlink between consumers of data and the actual data itself.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


Next, we put a nice little title on the plot.

First Plot

```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.title = "sin(x) * x^3"

        self.plot = plot
```


And finally, we set the plot as an attribute.

First Plot

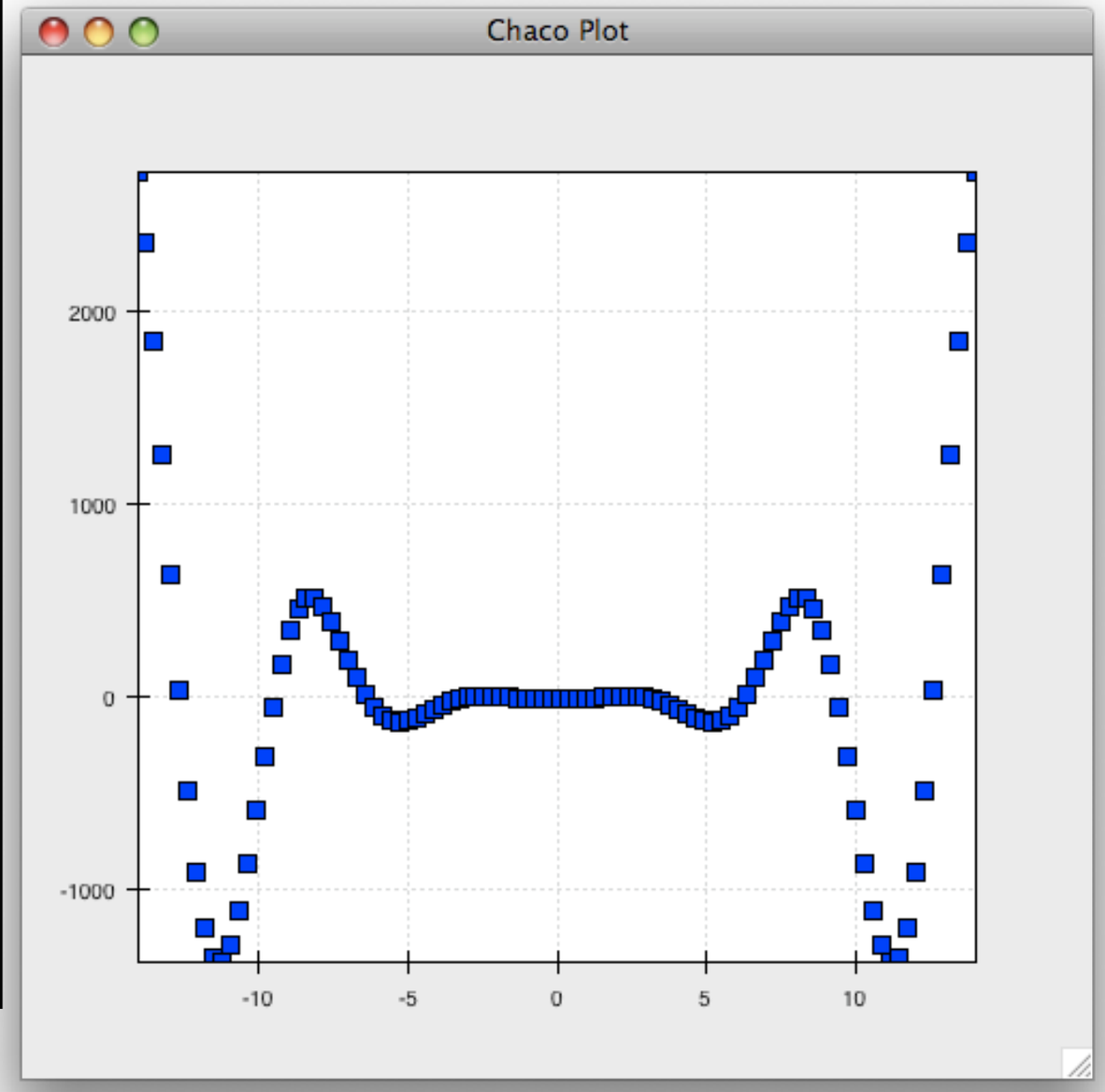
Finally, do something when this script is executed:

```
if __name__ == "__main__":

    LinePlot().configure_traits()
```


This one-liner instantiates a LinePlot() object, and calls configure_traits() on it, which, if you'll recall from the previous tutorial, brings up a dialog with a traits editor for the object. As it so happens, our traits editor just displays our "plot" attribute.

Scatter Plot

```
class ScatterPlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter",
                  color="blue")
        self.plot = plot

if __name__ == "__main__":
    ScatterPlot().configure_traits()
```

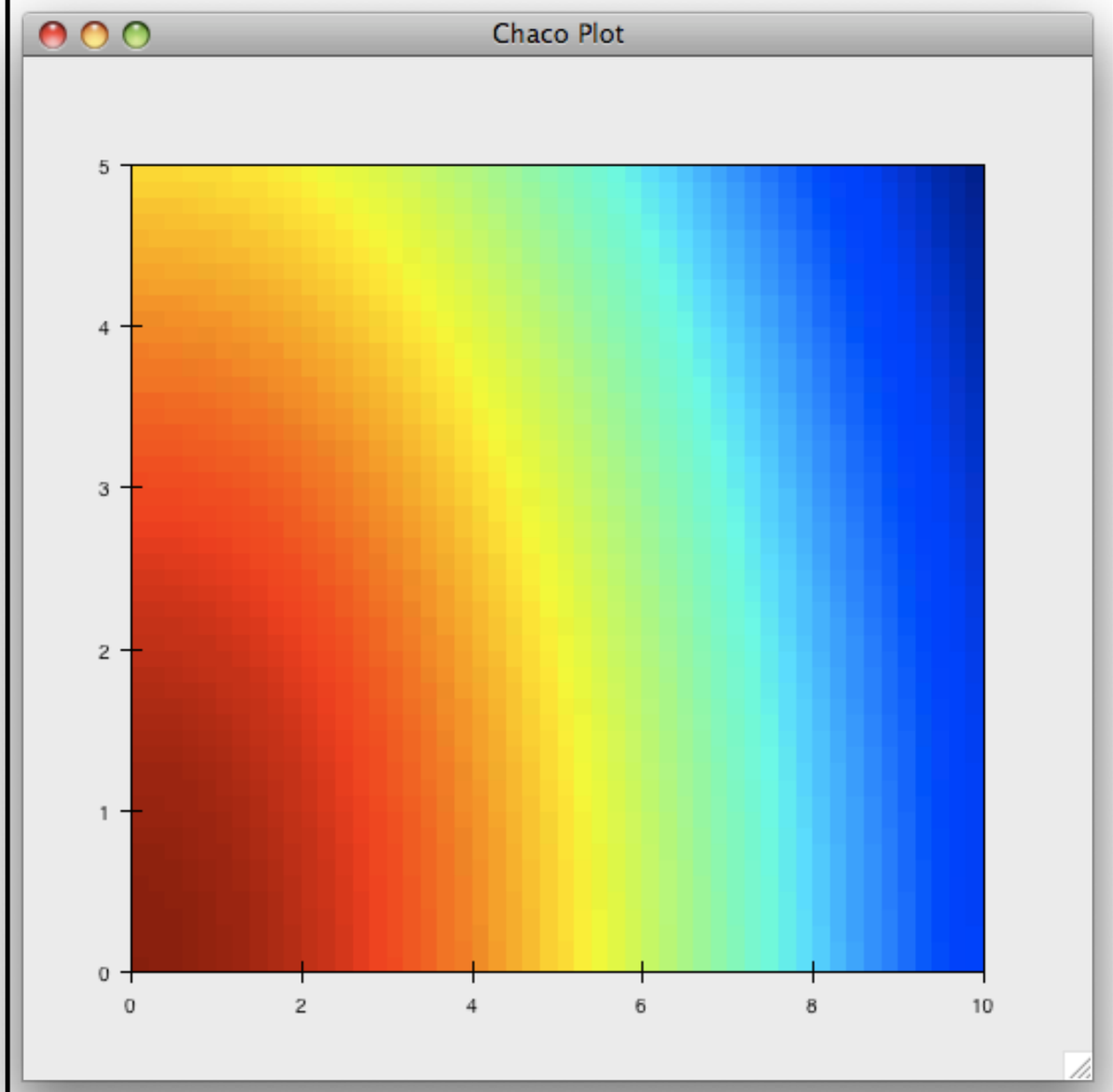


We can use the same basic structure as our first plot to do a scatter plot, as well.

(scatter.py)

Image Plot

```
class ImagePlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot', editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(0, 10, 50)
        y = linspace(0, 5, 50)
        xgrid, ygrid = meshgrid(x, y)
        z = exp(-(xgrid*xgrid+ygrid*ygrid)/100)
        plotdata = ArrayPlotData(imagedata = z)
        plot = Plot(plotdata)
        plot.img_plot("imagedata", xbounds=x,
                      ybounds=y, colormap=jet)
        self.plot = plot

if __name__ == "__main__":
    ImagePlot().configure_traits()
```

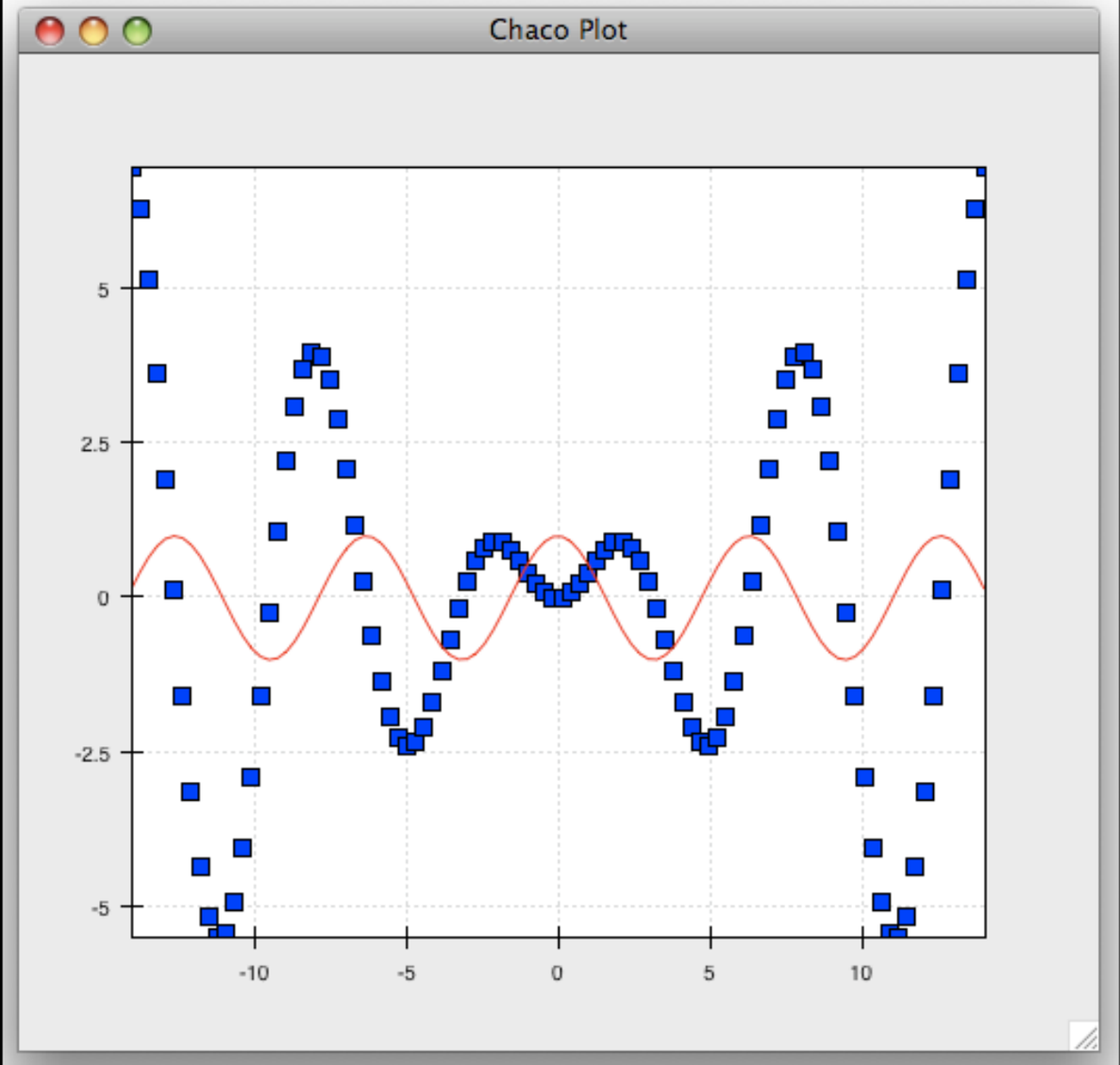


We can also do image plots in a similar fashion. Note that there are a few more steps to create the input Z data, and we also call a different method on the Plot – img_plot() instead of plot(). The details of the method parameters are not that important right now; this is just to show how you can apply the same basic pattern from the “first plot” example to do other kinds of plots.

(image.py)

A Slight Modification

```
class OverlappingPlot(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(),
             show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = x/2 * sin(x)
        y2 = cos(x)
        plotdata = ArrayPlotData(x=x,y=y,y2=y2)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter",
                  color="blue")
        plot.plot(("x", "y2"), type="line",
                  color="red")
        self.plot = plot

if __name__ == "__main__":
    OverlappingPlot().configure_traits()
```



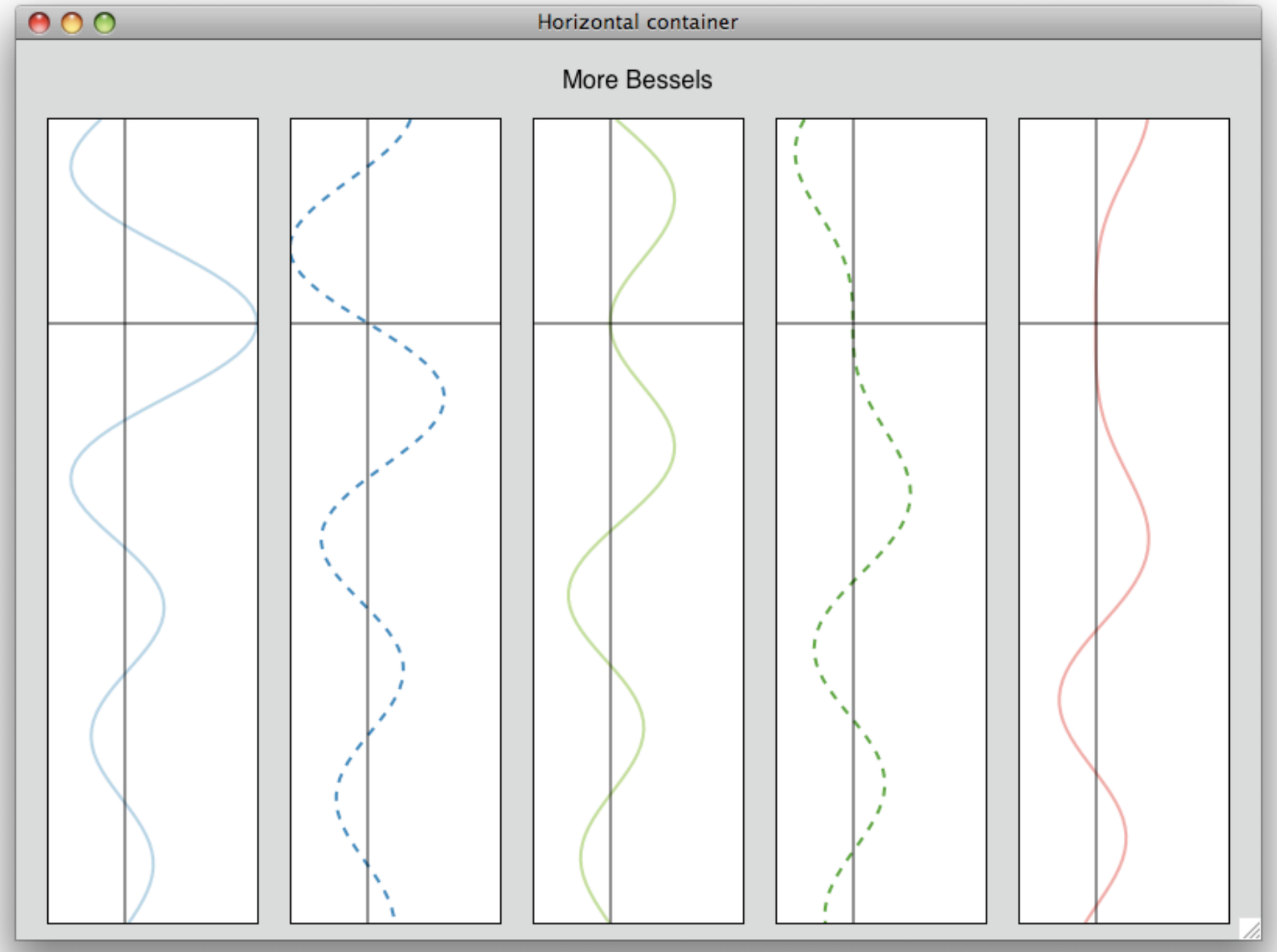
ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

I stated previously that the Plot object is both a container of renderers and a factory or generator of renderers. This modification of the previous example illustrates this point. We only create a single instance of Plot, but we call its plot() method twice. Each call creates a new renderer and adds it to the Plot's list of renderers.

Also notice that we are reusing the "x" array from the ArrayPlotData.

(overlapping.py)

HPlotContainer

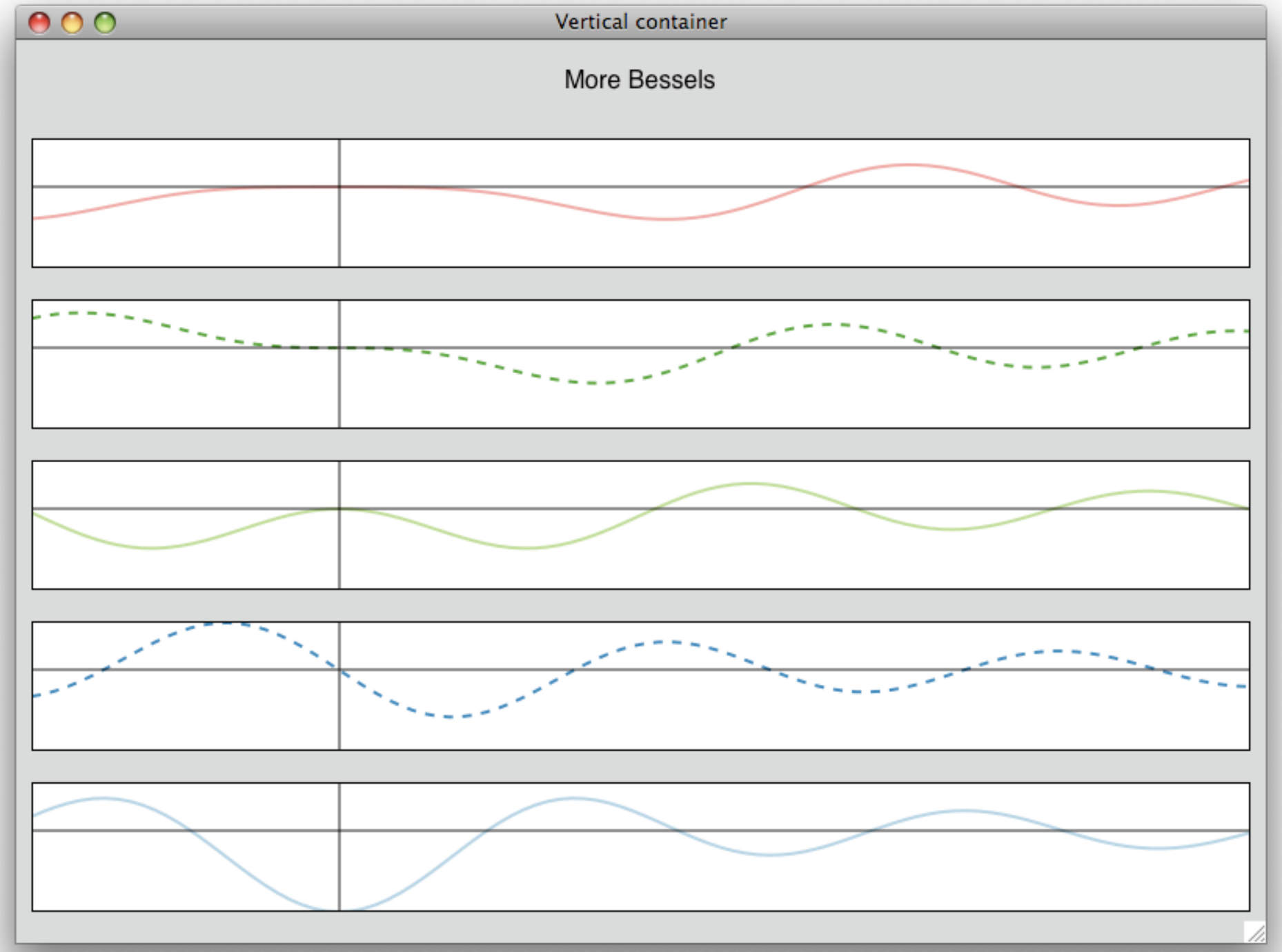


ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

So far we've only seen single plots, but frequently we need to plot data side-by-side. Chaco uses Containers to do layout.

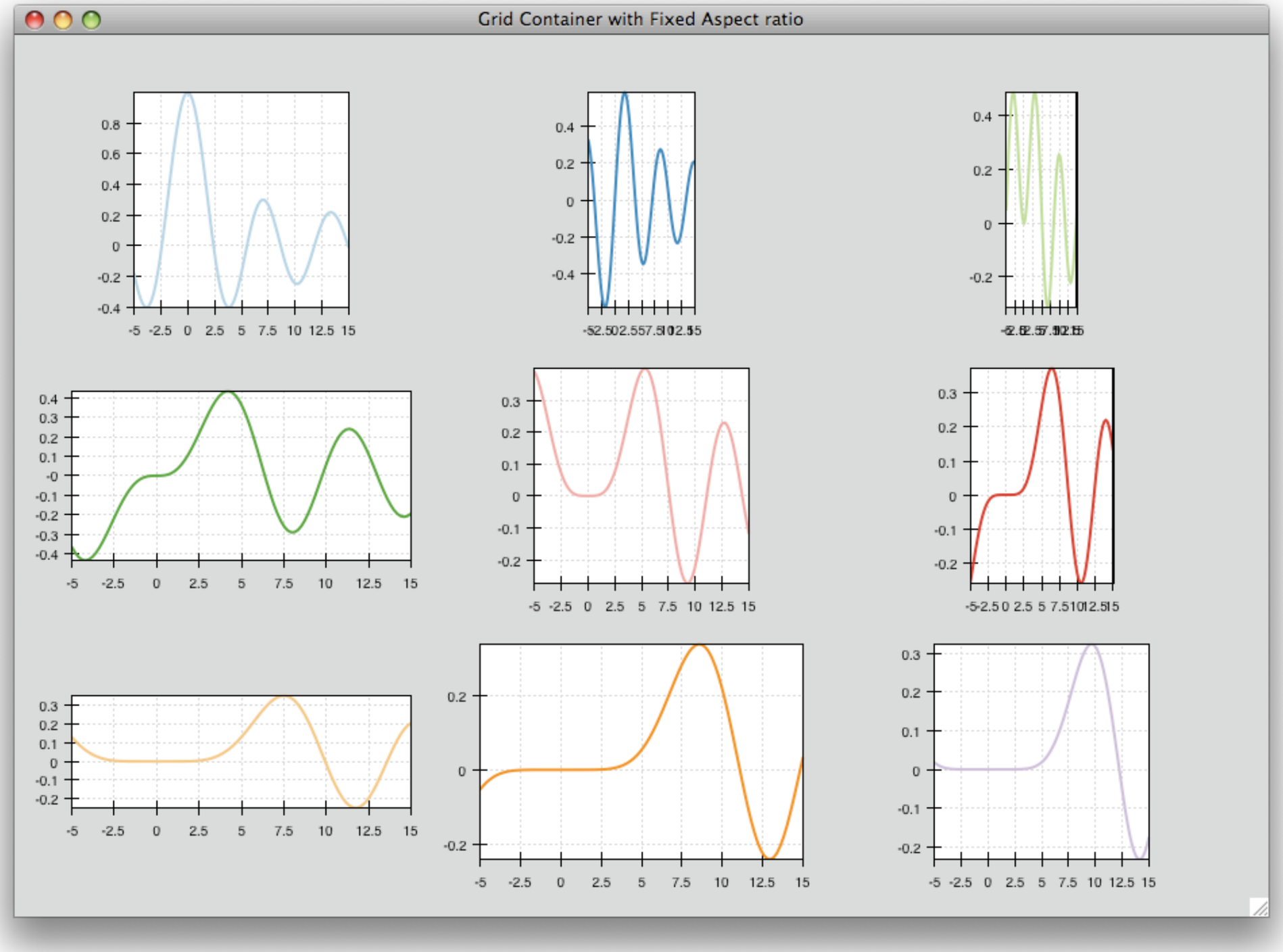
Horizontal containers place components horizontally.

VPlotContainer



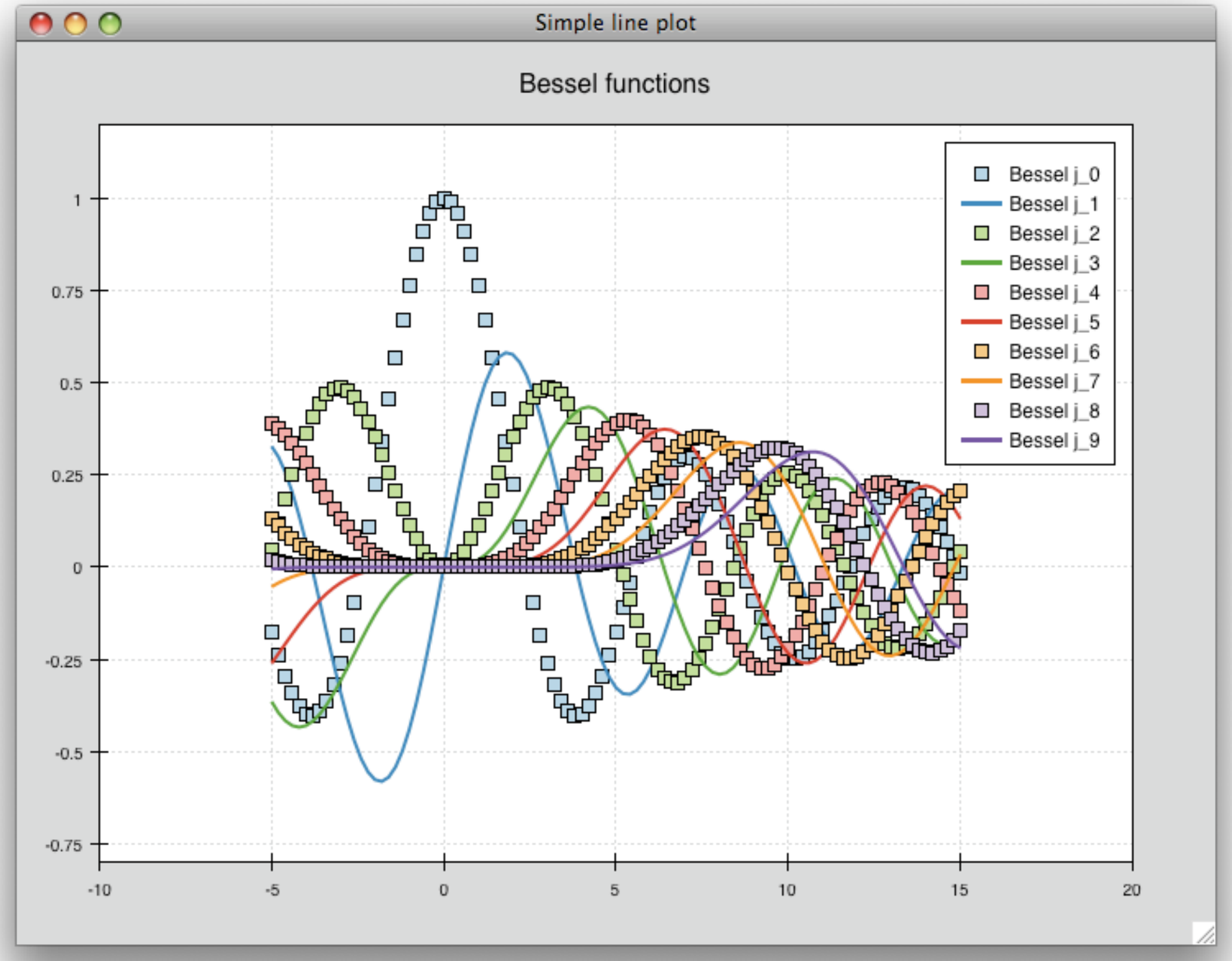
Vertical containers array components vertically.

GridContainer



Grid containers lay plots out in a grid.

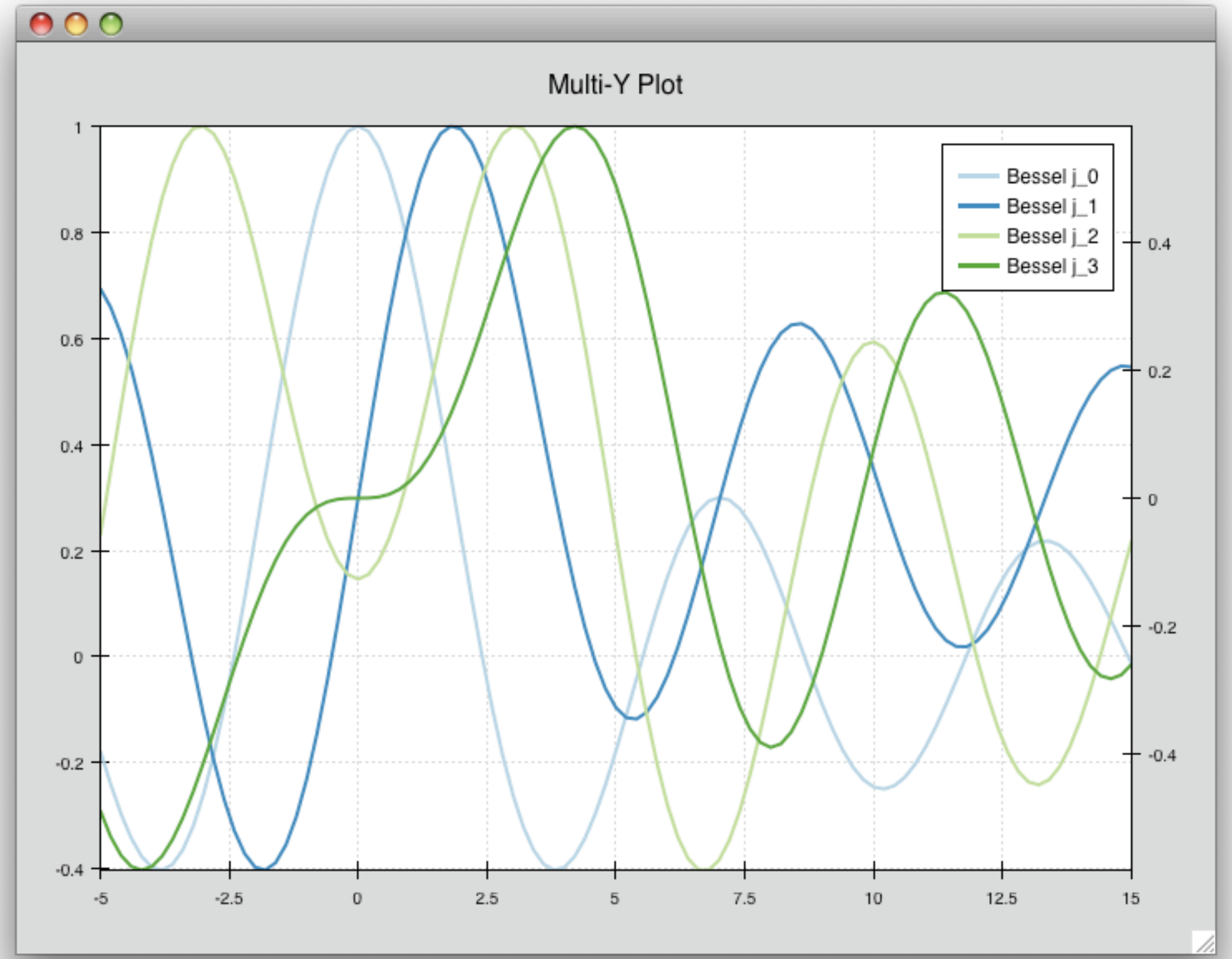
OverlayPlotContainer



OverlayPlotContainer just overlays plots on top of each other. You've actually already seen OverlayPlotContainers - the Plot object is actually a special subclass of OverlayPlotContainer.

All the plots inside this container share the same X and Y axis, but this is not a requirement of the container.

OverlayPlotContainer



ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

You might remember this example from earlier, which shows multiple line plots in an OverlayPlotContainer, and sharing only the X axis.

Using a Container

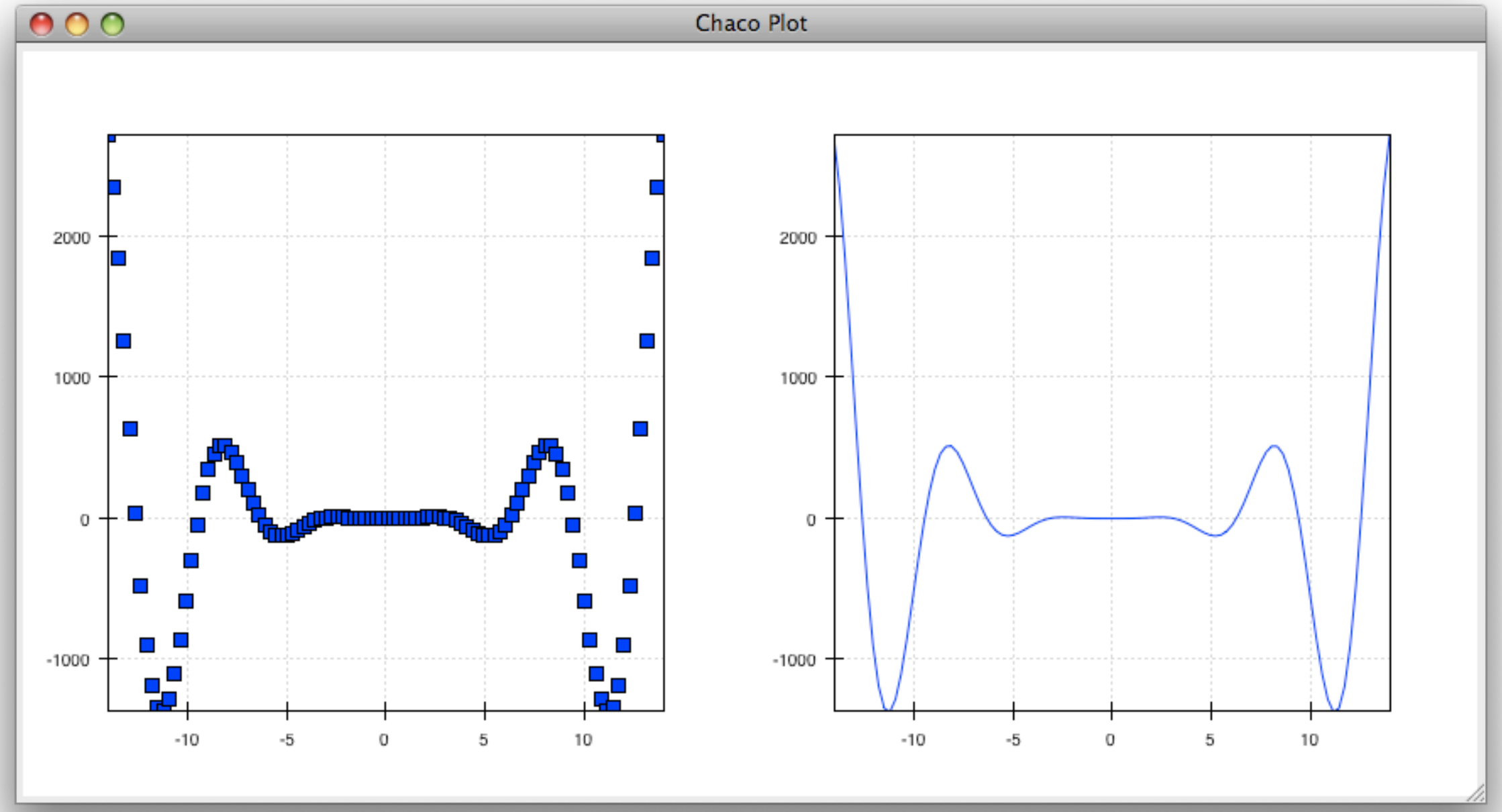
```
class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)
    traits_view = View(Item('plot', editor=ComponentEditor(), show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")

        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        container = HPlotContainer(scatter, line)
        self.plot = container
```


This code shows how we actually use a container.

(container.py)

Containers can have any Chaco component added to them. The above code creates a separate Plot instance for the scatter plot and the line plot, and adds them both to the HPlotContainer. This yields the following plot...

Using a Container



Configuring the Container

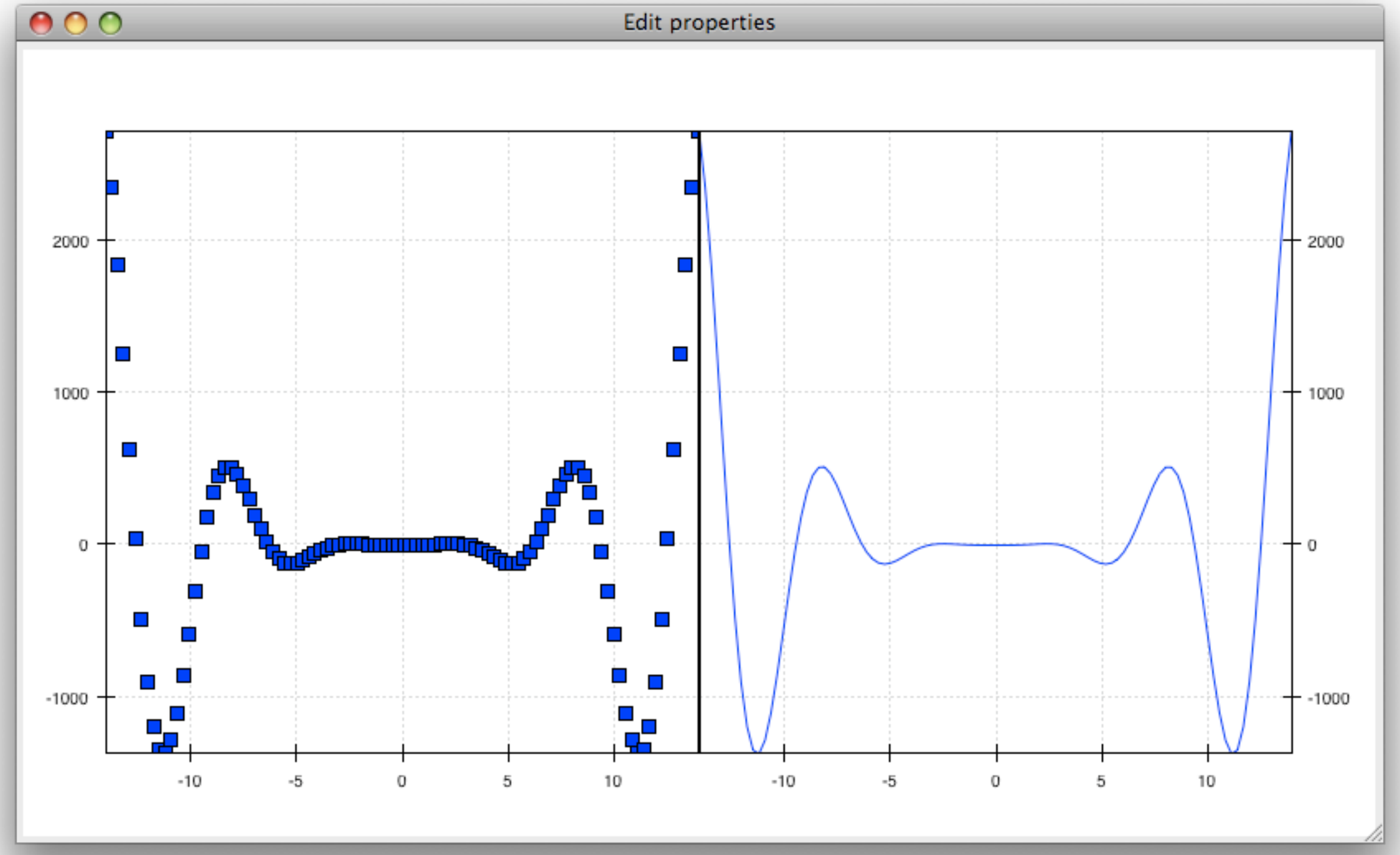
```
class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)
    traits_view = View(Item('plot', editor=ComponentEditor(), show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")

        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        container = HPlotContainer(scatter, line)
        container.spacing = 0
        scatter.padding_right = 0
        line.padding_left = 0
        line.y_axis.orientation = "right"
        self.plot = container
```


There are many parameters you can configure on a container, like background color, border, spacing, and padding. We're going to modify the previous example a little bit to make the two plots touch in the middle.

Something to note here is that all Chaco components have both bounds and padding or margin. In order to make our plots touch, we need to zero out the padding on the appropriate sides. We're also going to move the Y axis for the line plot (which is on the right hand side) to the right side.

(container_nospace.py)

Configuring the Container



...and this is the plot we get.

Editing Plot Traits

```
from enthought.traits.api import Int
from enthought.enable.api import ColorTraits
from enthought.chaco.api import marker_trait

class ScatterPlotTraits(HasTraits):
    plot = Instance(Plot)
    color = ColorTrait("blue")
    marker = marker_trait
    marker_size = Int(4)
```


So far, the stuff you've seen is pretty standard: building up a plot of some sort and doing some layout on them.

We're going to start taking advantage now of the underlying framework.

As I've mentioned before, Chaco is written using Traits. This means that all the graphical bits you see - and many of the bits you don't see - are all objects with various traits, generating events, and capable of responding to events.

We're going to modify our previous ScatterPlot example a little bit. We're going to add traits for color, marker type, and marker size. I've also added some lines here to show where the imports are coming from.

Editing Plot Traits

```
from enthought.traits.api import Int
from enthought.enable.api import ColorTraits
from enthought.chaco.api import marker_trait

class ScatterPlotTraits(HasTraits):
    plot = Instance(Plot)
    color = ColorTrait("blue")
    marker = marker_trait
    marker_size = Int(4)
    traits_view = View(
        Group(
            Item('color', label="Color", style="custom"),
            Item('marker', label="Marker"),
            Item('marker_size', label="Size"),
            Item('plot', editor=ComponentEditor(),
                 show_label=False),
            orientation = "vertical"
            ),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")
```


We're also going to change our Traits UI View to include references to these new Traits. We're also going to wrap them up in a Traits UI Group so that we can control the layout in the dialog a little better - here we're setting the layout orientation of the elements to "vertical".

Editing Plot Traits

```
def __init__(self):

    x = linspace(-14, 14, 100)
    y = sin(x) * x**3
    plotdata = ArrayPlotData(x = x, y = y)

    plot = Plot(plotdata)

    self.renderer = plot.plot(("x", "y"), type="scatter",
                              color="blue")[0]
    self.plot = plot
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Now we actually have to do something with those traits we just created.

This constructor is almost the same as the original LinePlot example, but we're going to grab a handle to the actual renderer returned by the plot() method. Recall that the Plot is a container for renderers and a factory for them. Well, the plot() method returns a list of the renderers it creates. In previous examples we've just been ignoring or discarding the return value, since we had no use for it. In this case, however, we're going to hold on to that renderer and assign it as an attribute.

The plot() method returns a list of renderers because for some values of the "type" argument, it will create multiple renderers. In our case, we are just doing a scatter plot, and this creates just a single renderer.

Editing Plot Traits

```
def __init__(self):

    x = linspace(-14, 14, 100)
    y = sin(x) * x**3
    plotdata = ArrayPlotData(x = x, y = y)

    plot = Plot(plotdata)

    self.renderer = plot.plot(("x", "y"), type="scatter",
                              color="blue")[0]
    self.plot = plot

def _color_changed(self):
    self.renderer.color = self.color
```


Then, we add Traits event handler for the “color” Trait. This takes the color trait that we declared on this class earlier, and copies the value to the ScatterPlot renderer. You can see why we needed to hold on to the renderer in the constructor: we need to modify attributes on it here in the event handler.

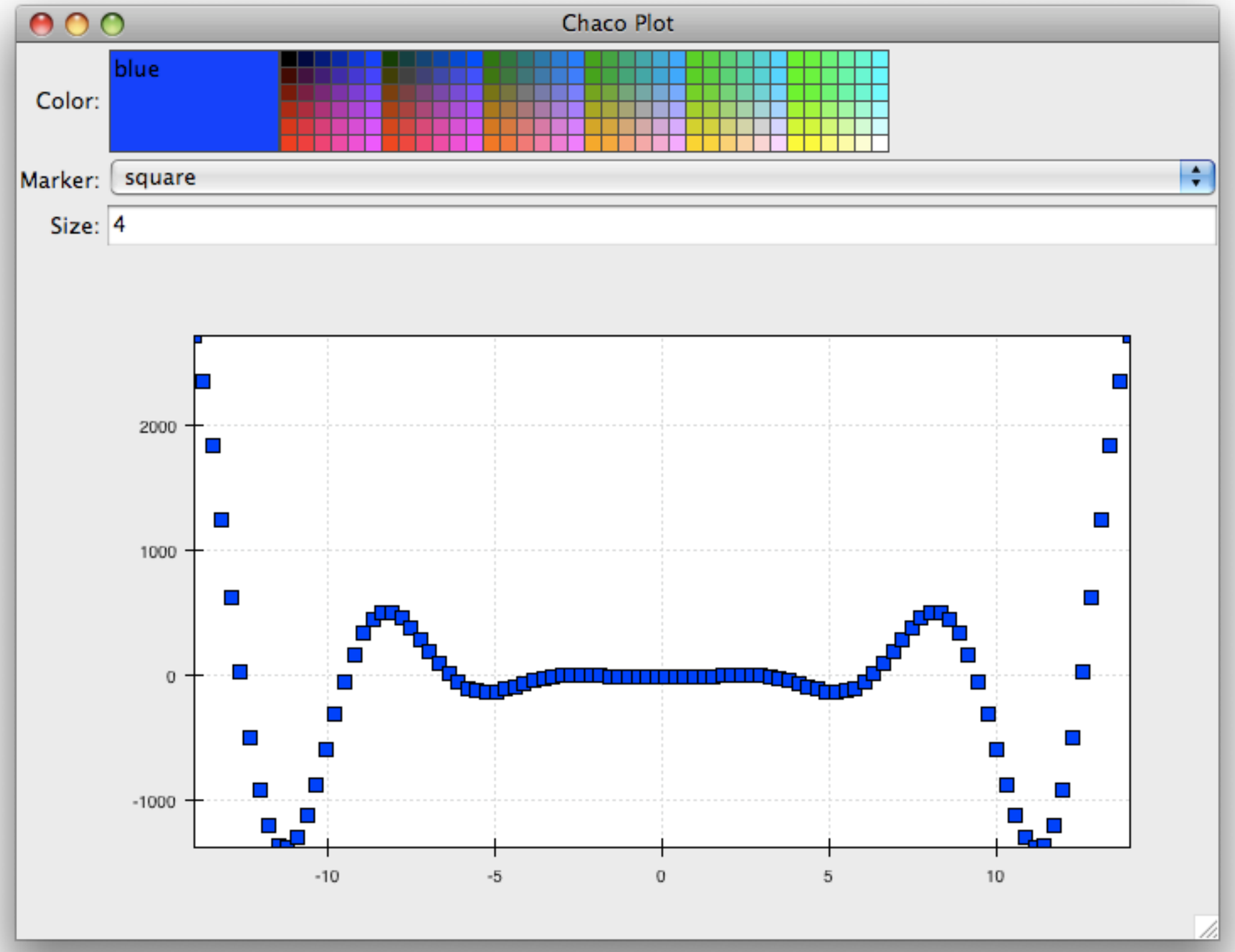
Editing Plot Traits

```
def __init__(self):

    x = linspace(-14, 14, 100)
    y = sin(x) * x**3
    plotdata = ArrayPlotData(x = x, y = y)

    plot = Plot(plotdata)

    self.renderer = plot.plot(("x", "y"), type="scatter",
                              color="blue")[0]
    self.plot = plot

def _color_changed(self):
    self.renderer.color = self.color

def _marker_changed(self):
    self.renderer.marker = self.marker

def _marker_size_changed(self):
    self.renderer.marker_size = self.marker_size
```


Now we do the same thing for the marker type and marker size traits.

Let's fire up this example and take a look.

Editing Plot Traits



Now, depending on your platform, the color editor may look different. This is how it looks on Mac OS X.

All of these are live. You can tweak them and the plot will update.

(traits.py)

Data Chooser

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")

    def __init__(self):
        x = linspace(-5, 10, 100)
        self.data = {"jn0": jn(0, x),
                     "jn1": jn(1, x),
                     "jn2": jn(2, x)}
        self.plotdata = ArrayPlotData(x = x, y = self.data["jn0"])
        plot = Plot(self.plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        self.plot = plot

    def _data_name_changed(self):
        self.plotdata.set_data("y", self.data[self.data_name])
```


Traits are not just useful for tweaking visual features. For instance, you can use them to select among several data items. This next example is based on the earlier LinePlot example, and I'll walk through the modifications.

Data Chooser

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")

    def __init__(self):
        x = linspace(-5, 10, 100)
        self.data = {"jn0": jn(0, x),
                     "jn1": jn(1, x),
                     "jn2": jn(2, x)}
        self.plotdata = ArrayPlotData(x = x, y = self.data["jn0"])
        plot = Plot(self.plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        self.plot = plot

    def _data_name_changed(self):
        self.plotdata.set_data("y", self.data[self.data_name])
```


First, we add an Enumeration Trait to select a particular data name, and we add an Item for this in the Traits UI View.

Data Chooser

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")

    def __init__(self):
        x = linspace(-5, 10, 100)
        self.data = {"jn0": jn(0, x),
                     "jn1": jn(1, x),
                     "jn2": jn(2, x)}
        self.plotdata = ArrayPlotData(x = x, y = self.data["jn0"])
        plot = Plot(self.plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        self.plot = plot

    def _data_name_changed(self):
        self.plotdata.set_data("y", self.data[self.data_name])
```


Next, we create a dictionary that maps those data names to actual numpy arrays.

Data Chooser

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")

    def __init__(self):
        x = linspace(-5, 10, 100)
        self.data = {"jn0": jn(0, x),
                     "jn1": jn(1, x),
                     "jn2": jn(2, x)}
        self.plotdata = ArrayPlotData(x = x, y = self.data["jn0"])
        plot = Plot(self.plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        self.plot = plot

    def _data_name_changed(self):
        self.plotdata.set_data("y", self.data[self.data_name])
```


When we initialize the ArrayPlotData, we'll set "y" to the "jn0" array. Also note that we are storing a reference to the "plotdata" object. In previous examples, we just passed it into the Plot and forgot about it.

Data Chooser

```
class DataChooser(HasTraits):
    plot = Instance(Plot)
    data_name = Enum("jn0", "jn1", "jn2")
    traits_view = View(
        Item('data_name', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=800, height=600, resizable=True,
        title="Data Chooser")

    def __init__(self):
        x = linspace(-5, 10, 100)
        self.data = {"jn0": jn(0, x),
                     "jn1": jn(1, x),
                     "jn2": jn(2, x)}
        self.plotdata = ArrayPlotData(x = x, y = self.data["jn0"])
        plot = Plot(self.plotdata)
        plot.plot(("x", "y"), type="line", color="blue")
        self.plot = plot

    def _data_name_changed(self):
        self.plotdata.set_data("y", self.data[self.data_name])
```

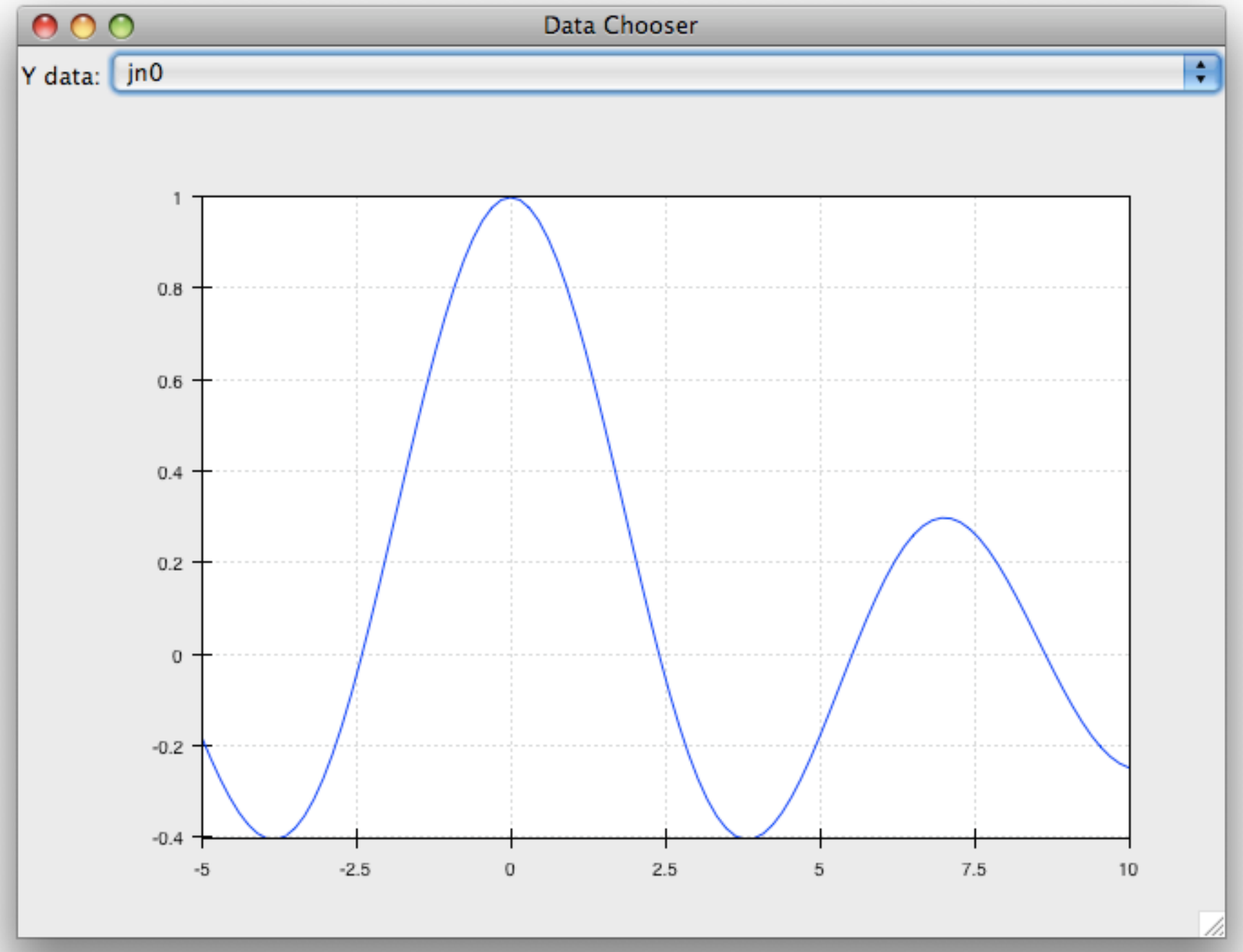

Finally, we create an event handler for the "data_name" Trait. Any time the data_name changes, we're going to look it up in the self.data dictionary, and push that value into the "y" data item in ArrayPlotData.

Note that there is no actual copying of data here, we're just passing around numpy references.

Let's take a look at the plot...

(data_chooser.py)

Data Chooser



Tools

```
from enthought.chaco.tools.api import PanTool, ZoomTool, DragZoom

class ToolsExample(HasTraits):
    plot = Instance(Plot)
    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500,
        resizable=True,
        title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line", color="blue")

        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        plot.tools.append(DragZoom(plot, drag_button="right"))

        self.plot = plot
```


Now we're going to move on to interacting with plots directly.

Chaco takes a modular approach to interactivity. We try to avoid hardcoding functionality into specific plot types or plot renderers. Instead, we factor out the interaction logic into classes we call "tools". An advantage of this approach is that we can then add new plot types and container types and we still get all of our old interactions "for free", as long as we adhere to certain basic interfaces.

Thus far, none of the example plots we've built are interactive. You can't pan or zoom them. We'll now modify the LinePlot so we can pan and zoom.

The pattern is that we create a new instance of a Tool, giving it a reference to the Plot, and then we append that tool to a list of tools on the plot. This looks a little redundant, and we can probably make this a little nicer so that if you hand in a Tool class instead of an instance, the plot itself automatically instantiates the class, but there is a reason why the tools need a reference back to the plot.

The tools use facilities on the plot to transform and interpret the events that it receives, as well as act on those events. Most tools will attributes on the plot. The pan and zoom tools, for instance, modify the data ranges on the component handed in to it.

(tools.py)

Tool Chooser

```
from enthought.traits.ui.api import CheckListEditor

class ToolsExample(HasTraits):
    plot = Instance(Plot)
    tools = List(editor=CheckListEditor(values = ["PanTool",
                                        "SimpleZoom", "DragZoom"]))
```


One of the nice things about having interactivity bundled up into modular tools is that in your application, you can dynamically adjust when a user can do something or when they can't.

We'll modify the previous example a little bit so that we can externally control what interactions are available on a plot.

First, we add a new trait to hold a list of names of the tools. This is just like when we added a list of data items in the Data Chooser example. However, instead of a drop-down (which is the default editor for an Enumeration trait), we tell Traits that we would like a CheckListEditor. This will allow us to mix and match interactions. We give the CheckListEditor a list of possible values, which are just the names of the tools. Notice that these are strings, and not the tool classes themselves.

Tool Chooser

```
from enthought.traits.ui.api import CheckListEditor

class ToolsExample(HasTraits):
    plot = Instance(Plot)
    tools = List(editor=CheckListEditor(values = ["PanTool",
                                    "SimpleZoom", "DragZoom"]))

    def __init__(self):
        x = linspace(-14, 14, 500)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)

        plot.plot(("x", "y"), type="line", color="blue")

        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        plot.tools.append(DragZoom(plot, drag_button="right"))

        self.plot = plot
```


Next, we're going to remove the lines that adds the tools to the plot.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


Instead, we're going to replace them with a trait event handler for the "tools" trait. This is just a little more sophisticated, so I'll step through it and explain each bit.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


Recall that “self.tools” is actually a list of strings. Those strings are not just any strings; they are exactly the names of the tool classes. Every time the user interacts with a CheckListEditor, it updates the self.tools trait to be a list of just the items that are selected.

So, this list comprehension turns the list of selected strings into a list of Classes by calling “eval” on each string.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


Next, we're going to iterate through all of the existing tools on the plot...

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


...and we're going to check to see if the tool is already on the plot. The way we do this is by looking at the __class__ attribute of the tool, and seeing if it is in our list of selected classes.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


If it is not, then we remove the tool from the plot.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```

If the tool is in the list of selected classes, then we leave the plot alone, but we remove the tool's class from the list of selected classes.

Tool Chooser

```
def _tools_changed(self):
    classes = [eval(class_name) for class_name in self.tools]

    for tool in self.plot.tools:
        if tool.__class__ not in classes:
            self.plot.tools.remove(tool)
        else:
            classes.remove(tool.__class__)

    for cls in classes:
        self.plot.tools.append(cls(self.plot))
    return
```


The reason we do this is because once we are done checking all of the existing tools on the plot, we're going to go through the remaining classes, and create new instances of them and stick them on the plot.

Tool Chooser

```
class ToolChooserExample(HasTraits):

    plot = Instance(Plot)
    tools = List(editor=CheckListEditor(values = ["PanTool", "ZoomTool", "DragZoom"]))
    traits_view = View(Item("tools", label="Tools", style="custom"),
                       Item('plot', editor=ComponentEditor(), show_label=False),
                       width=800, height=600, resizable=True,
                       title="Tool Chooser")

    def __init__(self):
        ...

    def _tools_changed(self):
        classes = [eval(class_name) for class_name in self.tools]
        # Remove all tools that are not in the enabled list in self.tools
        for tool in self.plot.tools:
            if tool.__class__ not in classes:
                self.plot.tools.remove(tool)
            else:
                classes.remove(tool.__class__)
        # Create new instances of tools for the remaining tool classes
        for cls in classes:
            self.plot.tools.append(cls(self.plot))
        return
```

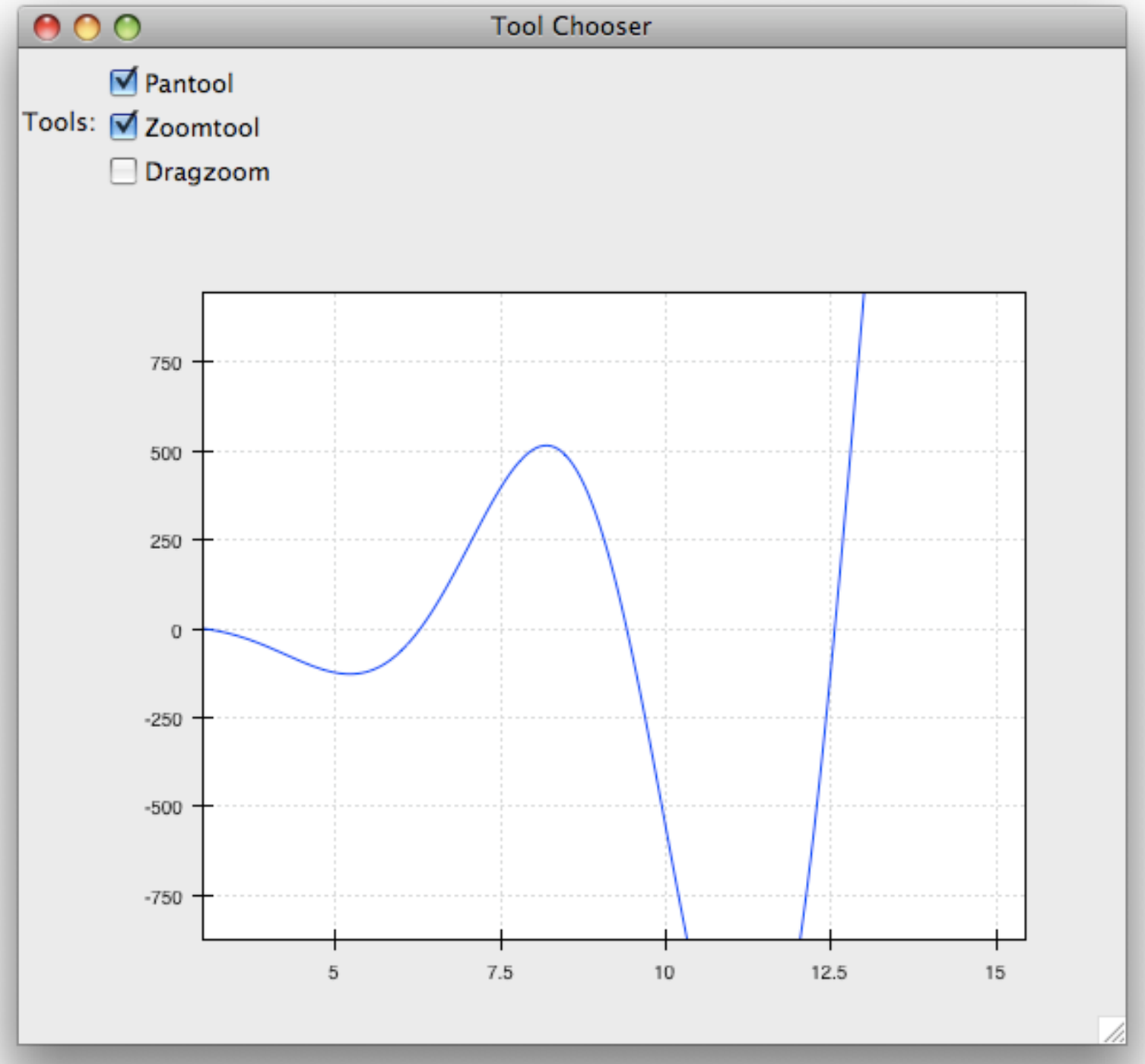
ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

So here's the full code listing for this example. I have omitted the constructor because it's the same as all the previous examples.

Now let's run this example and see what it does.

(tool_chooser.py)

Tool Chooser



Quick Recap

- Plot creation
  - Using ArrayPlotData to link data to plots
  - Using Plot to create different types of plots and multiple overlapping plots
- Plot layout using containers
- Plot configuration
  - Using Traits UI to modify plot attributes
  - Using Traits UI to modify plot data
- Tools

Here's a quick recap of the things we've looked at so far.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


So far, we've looked at how to manipulate individual plots.

But one of the features of Chaco's architecture is that all the underlying components of a plot are live objects, connected via events. In the next set of examples, we'll look at how to hook some of those up.

First, we're going to make two separate plots look at the same data space region. Here's the full code listing, and I'll walk through it.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


The first thing to mention is that since we are going to create two plots side by side, we're going to use an HPlotContainer at the top level. So, we're going to change the trait type, and we'll rename the trait to be something more appropriate.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


Next, we'll create some data just like we've done in the past.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


Then we'll create our plots. We'll create one scatter and one line plot, but both are looking at the same ArrayPlotData object. It's worth pointing out that the data space linking we do in this example does not require the plots to share data; I'm just doing this for the sake of convenience.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


Next, we create a horizontal container and put the two plots inside it.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


We're going to add pan and zoom to both plots.

Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```

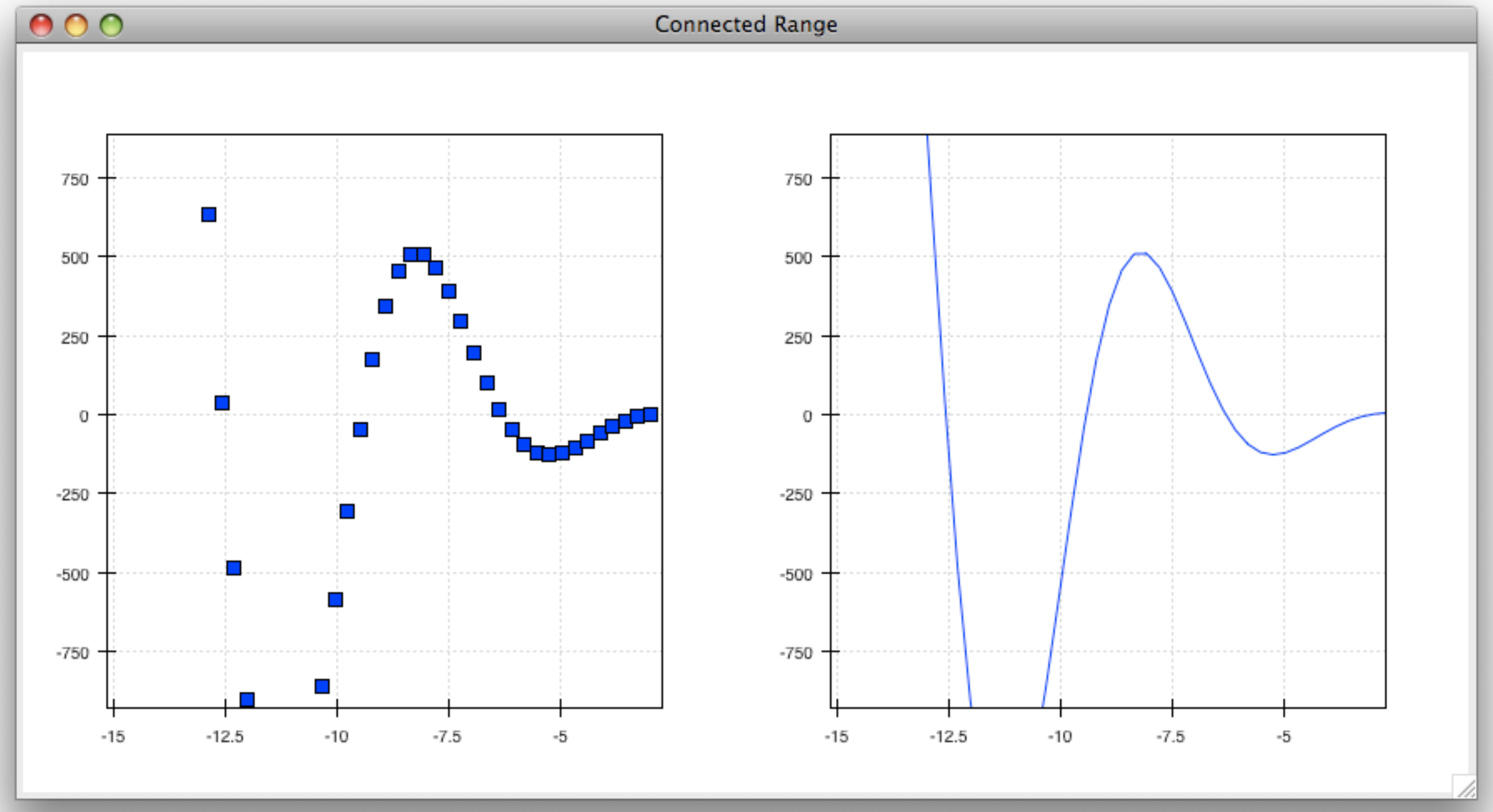

This final line is the magic one.

Chaco has a concept of a “data range” to express bounds in data space. there are a series of objects representing this concept. The standard, 2D, rectilinear plots that we’ve been looking at all have a two-dimensional range on them. Here, we are replacing the range on the scatter plot with the range from the line plot. We can also have reversed this list, and set the line plot’s range to be the scatter plot’s range. Either way, the underlying thing that’s happened is that now, both plots are looking at the same data space bounds.

So now let’s look at the plot.

(connected_range.py)

Connected Plots



Partially Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata)
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.index_range = line.index_range
```

ENTHOUGHT SCIENTIFIC COMPUTING SOLUTIONS

We can modify this slightly to make it more interesting. The 2D data range is actually composed one two 1D data ranges, and you can get at those independently. So, you can link up just the X or the Y axes.

The term “index” and “value” show up a lot in Chaco, and I want to talk about that real quick. Since you can easily change the orientation of most Chaco plots, we want some way to differentiate between the abscissa and the ordinate axes. If we just stuck with “x” and “y”, things would get pretty confusing – for instance, you would be setting the Y axis labels with the name of the X data set. I thought that “index” and “value” were reasonable terms, and they are less obscure than “abscissa” and “ordinate”.

Flipped Connected Plots

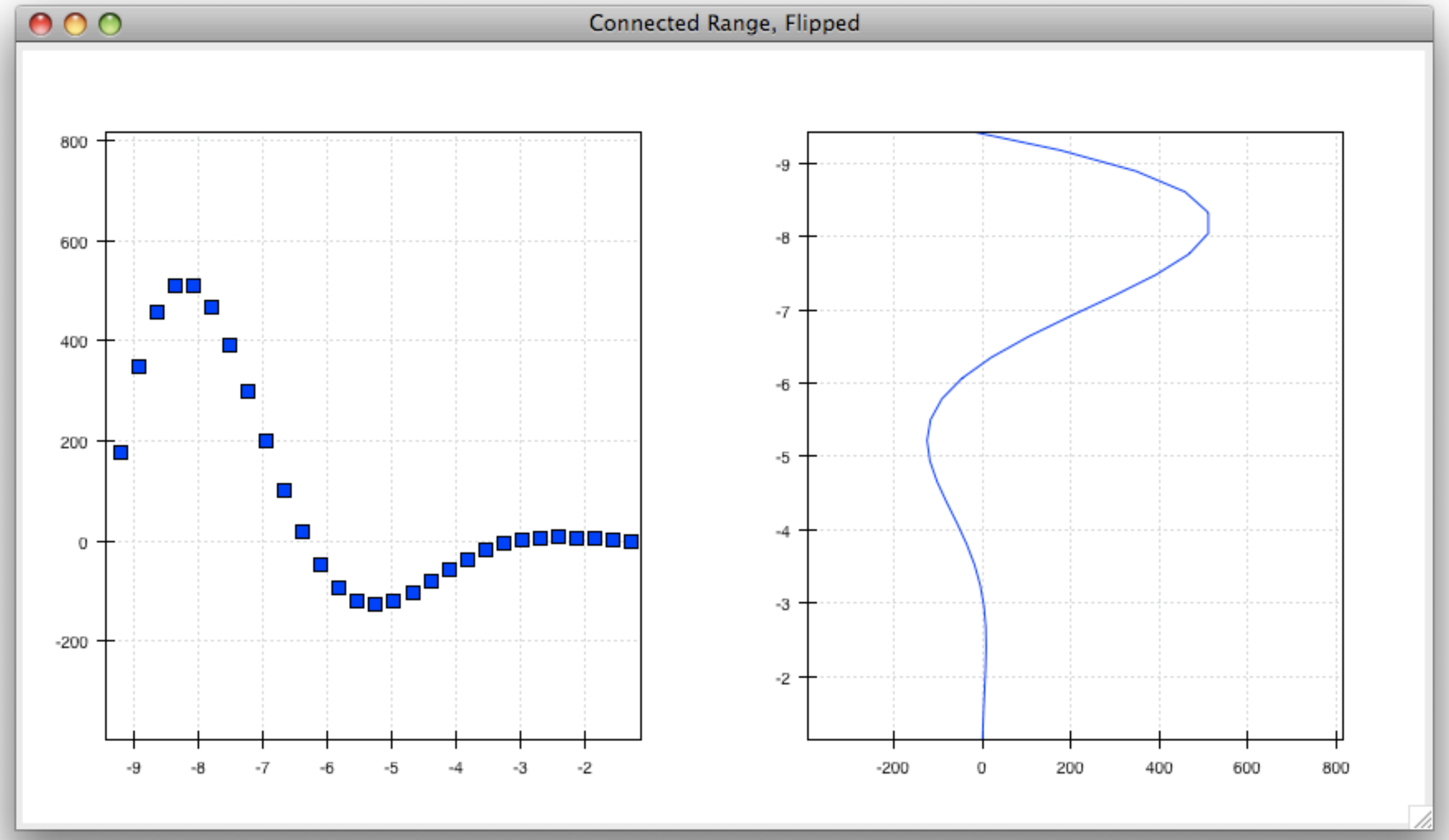
```
class FlippedExample(HasTraits):
    container = Instance(HPlotContainer)
    traits_view = View(Item('container', editor=ComponentEditor(),
                            show_label=False),
                       width=1000, height=600, resizable=True,
                       title="Connected Range, Flipped")
    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        scatter = Plot(plotdata)
        scatter.plot(("x", "y"), type="scatter", color="blue")
        line = Plot(plotdata, orientation="v", default_origin="top left")
        line.plot(("x", "y"), type="line", color="blue")

        self.container = HPlotContainer(scatter, line)

        scatter.tools.append(PanTool(scatter))
        scatter.tools.append(ZoomTool(scatter))
        line.tools.append(PanTool(line))
        line.tools.append(ZoomTool(line))

        scatter.range2d = line.range2d
```


To show you how easy it is to change the orientation of Chaco plots, we'll do it real quick here. I just have to add one line.

This sets the line plot to be vertically oriented, and the "default_origin" parameter sets the index axis to be increasing downwards.

(connected_orientation.py)

Flipped Connected Plots



Two Windows

```
class PlotEditor(HasTraits):

    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def __init__(self, *args, **kw):
        HasTraits.__init__(self, *args, **kw)

        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        self.plot = plot
```


So far all the examples we've seen have lived inside a single window. This next example demonstrates that you can hook up Chaco components beyond the borders of a single window.

We'll take our trusty LinePlot class from the very first example, and add some things.

(connected_widgets.py)

Two Windows

```
class PlotEditor(HasTraits):

    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def __init__(self, *args, **kw):
        HasTraits.__init__(self, *args, **kw)

        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        self.plot = plot
```


First, we'll expose the plot type and the orientation as traits.

Two Windows

```
class PlotEditor(HasTraits):

    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def __init__(self, *args, **kw):
        HasTraits.__init__(self, *args, **kw)

        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        self.plot = plot
```


We'll also add a Traits UI item for the orientation. Since the orientation Trait is an Enum, this will appear as a drop-down box in the window.

Two Windows

```
class PlotEditor(HasTraits):

    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def __init__(self, *args, **kw):
        HasTraits.__init__(self, *args, **kw)

        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        self.plot = plot
```


The constructor has a few changes, too. First of all, we're going to be a good Python class and properly call up to our base class constructor. I didn't do this in any of the previous examples because it wasn't needed, and it would have added clutter. But we'll be needing it in this class.

Two Windows

```
class PlotEditor(HasTraits):

    plot = Instance(Plot)
    plot_type = Enum("scatter", "line")
    orientation = Enum("horizontal", "vertical")
    traits_view = View(Item('orientation', label="Orientation"),
                       Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=500, height=500, resizable=True,
                       title="Chaco Plot")

    def __init__(self, *args, **kw):
        HasTraits.__init__(self, *args, **kw)

        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)

        plot = Plot(plotdata)
        plot.plot(("x", "y"), type=self.plot_type, color="blue")
        plot.tools.append(PanTool(plot))
        plot.tools.append(ZoomTool(plot))
        self.plot = plot
```


Next, when we create the plot, we'll use the plot_type trait instead of hardcoding "scatter" or "line".

Two Windows

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"
```


We're also going to declare a Trait event handler for the "orientation" trait...

Two Windows

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"


if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```


... and we're actually going to do something interesting in the __main__ section.

Two Windows

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"


if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```


Rather than creating two individual Chaco plots within the same window, we're going to create two instances of this PlotEditor object. One will be configured to be a scatter plot, and the other will be configured to be a line plot.

Two Windows

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"


if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```

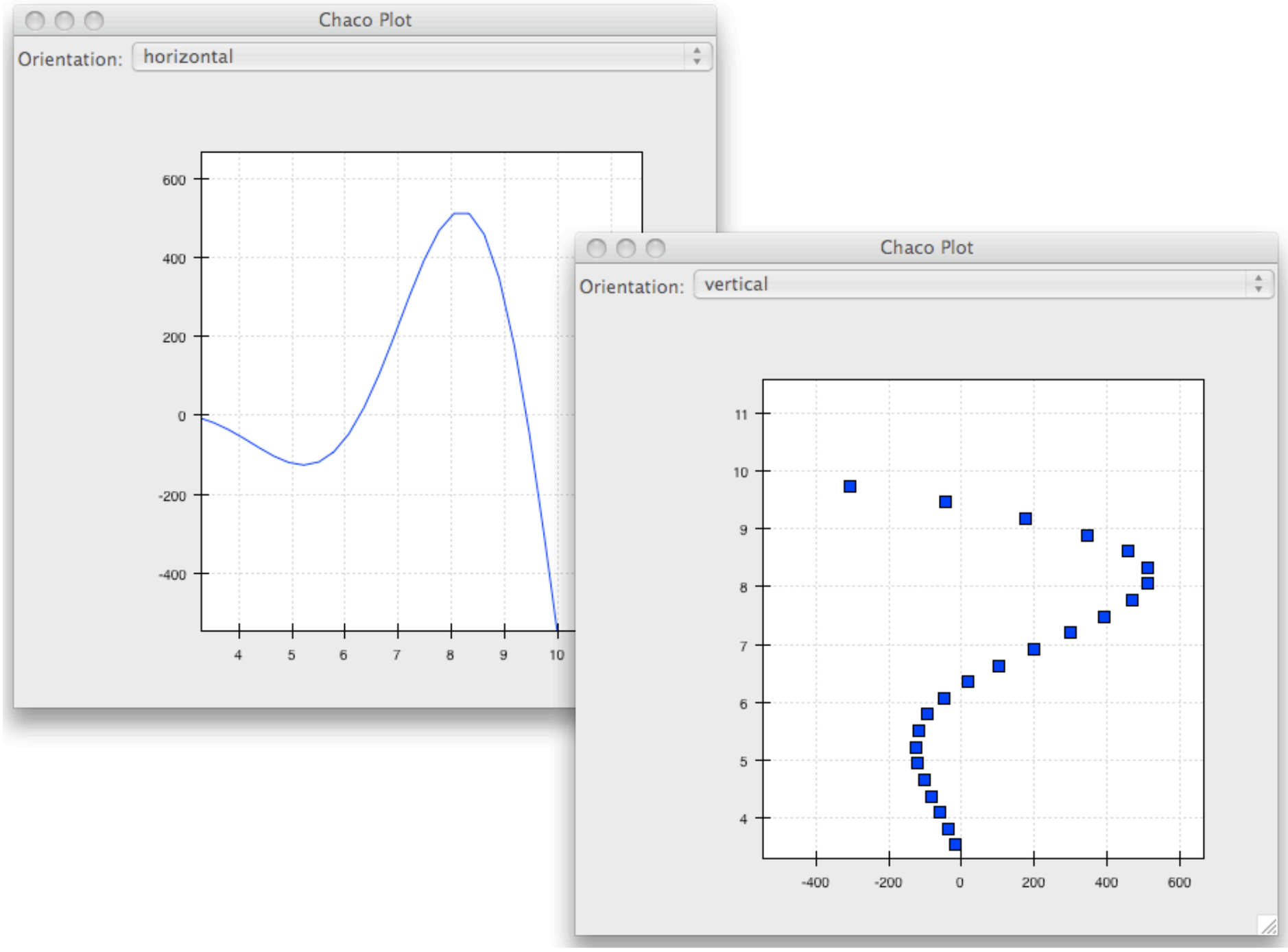

We then reach down into the Plot objects inside each editor and hook up their ranges.

Two Windows

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"


if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```


And finally, we open traits UI editors on both objects. Note that we call edit_traits() on the first object, but configure_traits() on the second object. The technical reason for this is that configure_traits() will start the wxPython main loop, whereas edit_traits() won't. So, if we had a whole bunch of these editors, more than just two, we could call edit_traits() on all of them, except for the last one - it would have to be configure_traits(). If you were to call edit_traits() on the last one, then you would see a whole bunch of windows show up and then disappear.

Two Windows


Chaco Plot
Orientation:
horizontal
600
400
200
0
-200
-400
4
5
6
7
8
9
10
Chaco Plot
Orientation:
vertical
11
10
9
8
7
6
5
4
-400
-200
0
200
400
600

Interacting with IPython

```
    def _orientation_changed(self):
        if self.orientation == "vertical":
            self.plot.orientation = "v"
        else:
            self.plot.orientation = "h"


if __name__ == "__main__":

    scatter = PlotEditor(plot_type = "scatter")
    line = PlotEditor(plot_type = "line")

    scatter.plot.range2d = line.plot.range2d

    line.edit_traits()
    scatter.configure_traits()
```


The HasTraits objects we've been building are actually pretty nifty. Let's look again at what we did with the PlotEditor in the previous example.

We created two instances of the PlotEditor, and then we called edit_traits() or configure_traits() (which is basically the same thing) on both of them.

(ploteditor.py)

Interacting with IPython

```
$ ipython -wthread
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
Type "copyright", "credits" or "license" for more information.

IPython 0.9.0.bzr.r1016 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints
more.

In [1]: from ploteditor import PlotEditor

In [2]: p1 = PlotEditor()

In [3]: p2 = PlotEditor()

In [4]: p3 = PlotEditor()

In [5]: p1.edit_traits(); p2.edit_traits(); p3.edit_traits()
Out[5]: <enthought.traits.ui.ui.UI object at 0x195ce210>
Out[5]: <enthought.traits.ui.ui.UI object at 0x199d2660>
Out[5]: <enthought.traits.ui.ui.UI object at 0x199fb0c0>
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Well, we don't have to just create them inside the __main__ of a script. We can import them into an IPython session.

This pops up more editors than you can shake a stick at. You can now do things like:
- change the orientation of a plot
- change the bgcolor of a plot (and then call request_redraw)
- call p1.x_grid.edit_traits()
- get the data from p1.plot.data.get_data("x") and then set_data() with a new function
- manually set axis bounds via p1.index_range.set_bounds()
- link up ranges
- toggle the legend, or configure its traits

Writing a custom Tool

```
from enthought.enable.api import BaseTool

class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

class ScatterPlot(HasTraits):
    plot = Instance(Plot)

    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=800, height=600, resizable=True,
                       title="Custom Tool")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter", color="blue")
        plot.tools.append(CustomTool(plot))
        return plot
```


So far we've been using things that are built in to Chaco. One of Chaco's strengths as a framework is that it is easy to extend and customize. There are nice interfaces between various parts of the framework, so you can write a custom plot or tool, plug it in, and it will play well with the existing pieces.

Our next step is to write a simple, custom tool that will print out the position on the plot under the mouse cursor.

Writing a custom Tool

```
from enthought.enable.api import BaseTool

class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

class ScatterPlot(HasTraits):
    plot = Instance(Plot)

    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=800, height=600, resizable=True,
                       title="Custom Tool")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter", color="blue")
        plot.tools.append(CustomTool(plot))
        return plot
```


To write a tool, we subclass from BaseTool.

Writing a custom Tool

```
from enthought.enable.api import BaseTool

class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

class ScatterPlot(HasTraits):
    plot = Instance(Plot)

    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=800, height=600, resizable=True,
                       title="Custom Tool")

    def _plot_default(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="scatter", color="blue")
        plot.tools.append(CustomTool(plot))
        return plot
```


Then, we define a callback method with a particular name. This name is broken down into two parts: "normal", and "mouse_move". The first part indicates the "event state" of the tool, which is something that we'll cover next. The second part is the interesting bit for now: it is the name of the event that triggers this callback.

All events have an X and a Y position, and we're just going to print it out. Let's see what this looks like.

(custom_tool_screen.py)

Writing a custom Tool

```
class CustomTool(BaseTool):
    def normal_mouse_move(self, event):
        print "Screen point:", event.x, event.y

    def normal_left_down(self, event):
        print "Mouse went down at", event.x, event.y

    def normal_left_up(self, event):
        print "Mouse went up at:", event.x, event.y
```


Some other event names are “mouse_enter”, “mouse_leave”, “mouse_wheel”, “left_down”, “left_up”, “right_down”, “right_up”, “key_pressed”. So, we can modify the tool to do things on mouse clicks as well.

(custom_tool_click.py)

Writing a custom Tool

```
class CustomTool(BaseTool):

    event_state = Enum("normal", "mousedown")

    def normal_mouse_move(self, event):
        print "Screen:", event.x, event.y

    def normal_left_down(self, event):
        self.event_state = "mousedown"
        event.handled = True

    def mousedown_left_up(self, event):
        self.event_state = "normal"
        event.handled = True
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Chaco tools are stateful – you can think of them as state machines that toggle states based on the events they receive. All tools have at least one state, and that state is named “normal”. That’s why the callbacks we’ve been implementing are all named “normal_” this and “normal_” that.

Our next tool is going to have two states, “normal” and “mousedown”. We’re going to enter the “mousedown” state when we detect a “left down”, and we’ll exit that state when we detect a “left up”.

Writing a custom Tool

```
class CustomTool(BaseTool):

    event_state = Enum("normal", "mousedown")

    def normal_mouse_move(self, event):
        print "Screen:", event.x, event.y

    def normal_left_down(self, event):
        self.event_state = "mousedown"
        event.handled = True

    def mousedown_mouse_move(self, event):
        print "Data:", self.component.map_data((event.x, event.y))

    def mousedown_left_up(self, event):
        self.event_state = "normal"
        event.handled = True
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

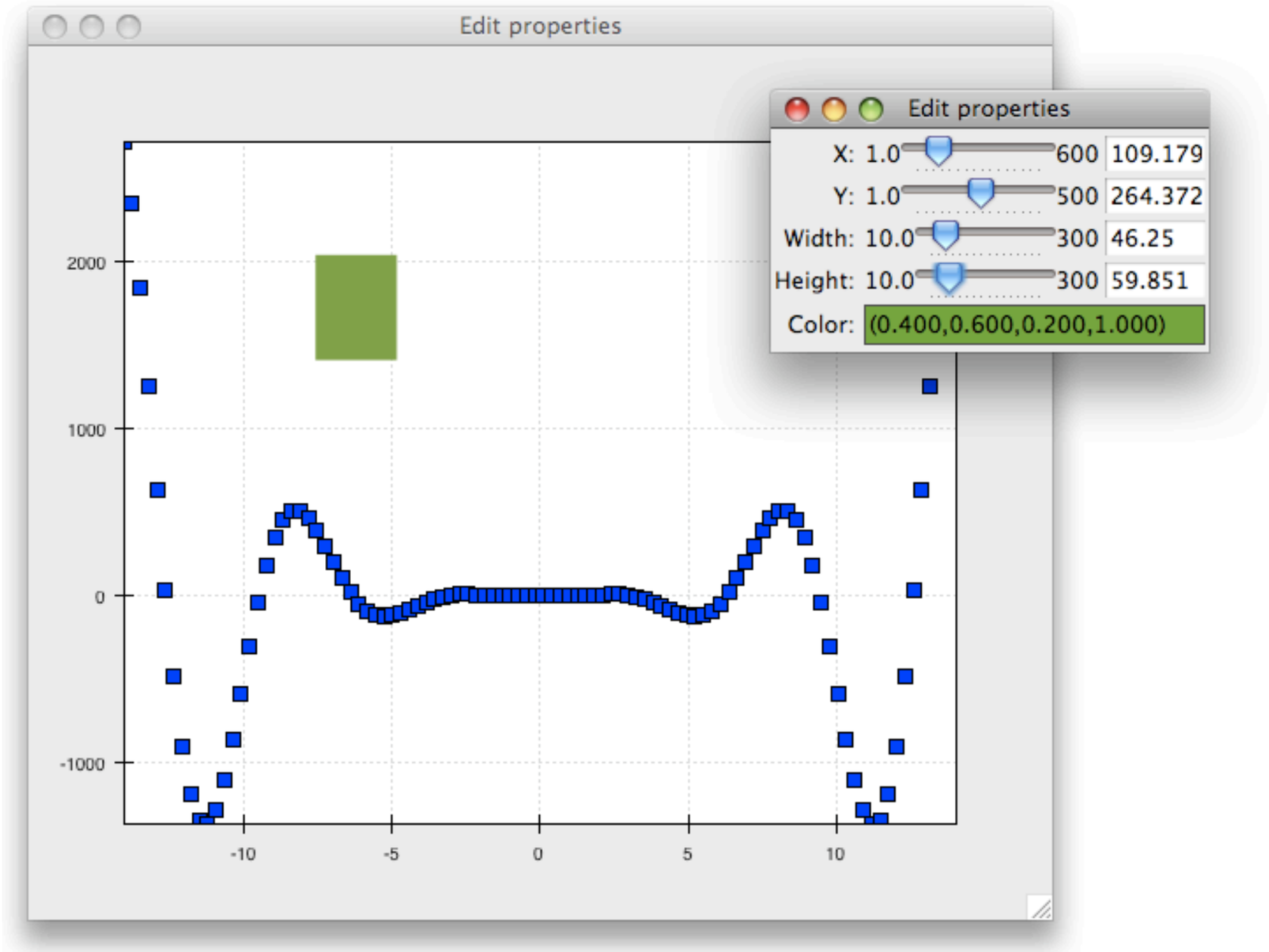
We're also going to do something a little different when the mouse moves while it's down. Rather than printing out the screen coordinates, we're going to ask our "component" to map the screen coordinates into data space. This is why tools need to be handed in a reference to a plot when they get constructed, because almost all tools need to use some capabilities (like map_data) on the components for which they are receiving events.

So let's see what this example looks like.

(custom_tool.py)

Additional Tutorial Examples

Custom Overlay

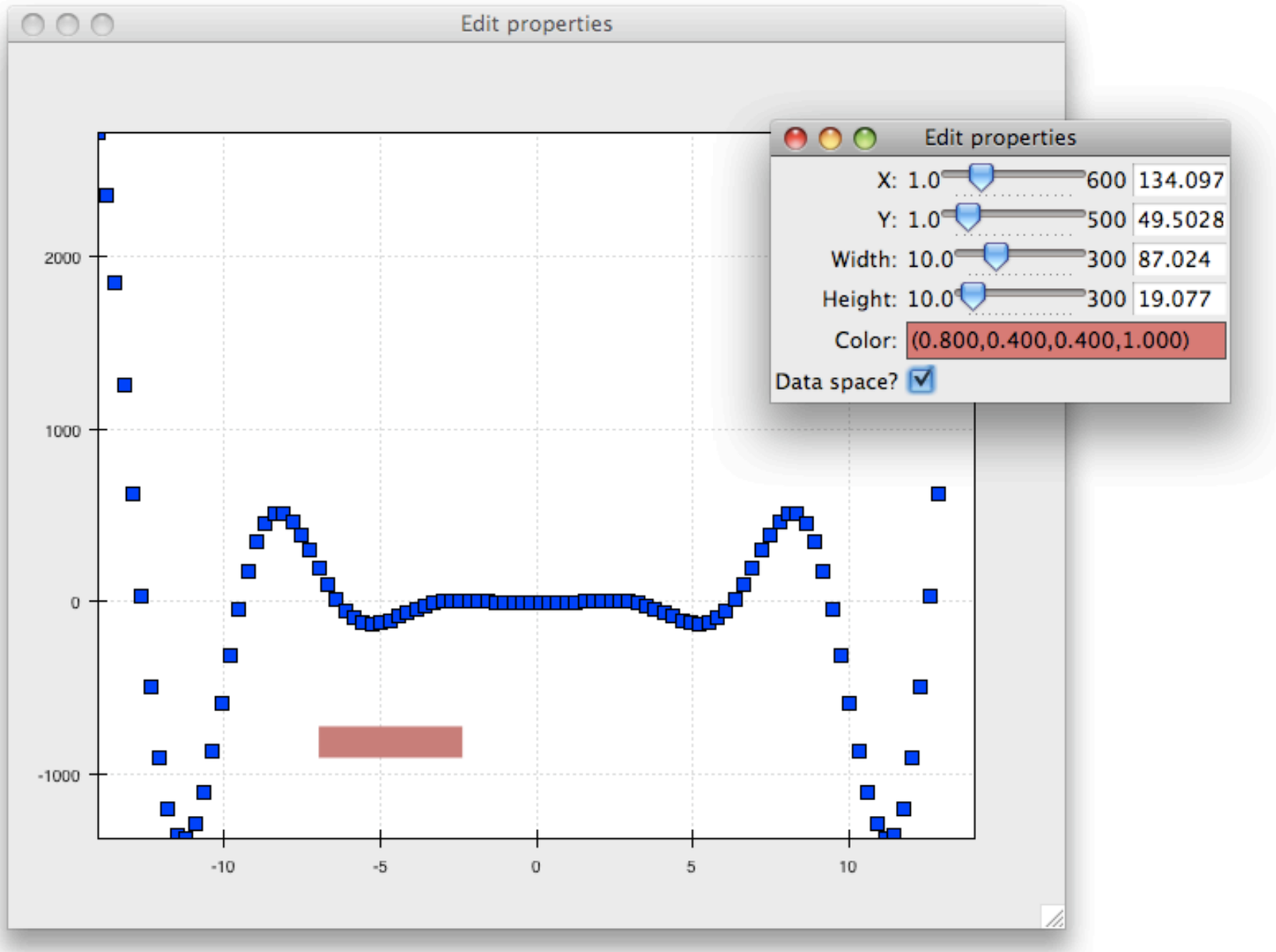


ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

(custom_overlay.py)

Additional Tutorial Examples

Custom Overlay with Dataspace Option



ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

(custom_overlay_dataspace.py, custom_overlay_movetool.py)

More information

- Web page:
  http://code.enthought.com/chaco

- Wiki:
  https://svn.enthought.com/enthought/wiki/ChacoProject

- Gallery:
  http://code.enthought.com/projects/chaco/gallery.php

- Mailing lists:
  enthought-dev@enthought.com,
  chaco-users@enthought.com