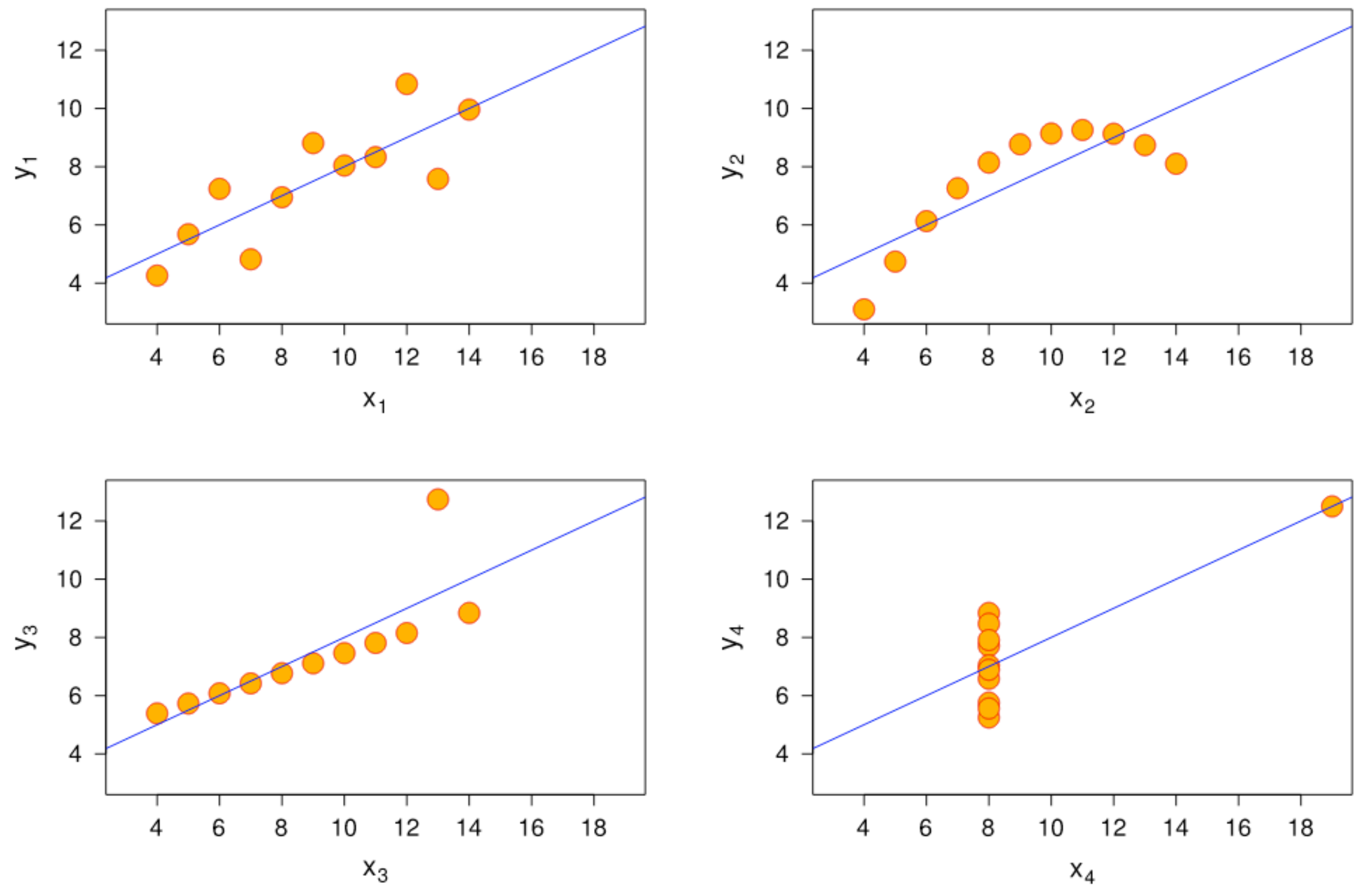
# Interactive Plotting with Chaco

Naveen Michaud-Agrawal
Enthought, Inc.

PyGotham 2012
June 8, 2012

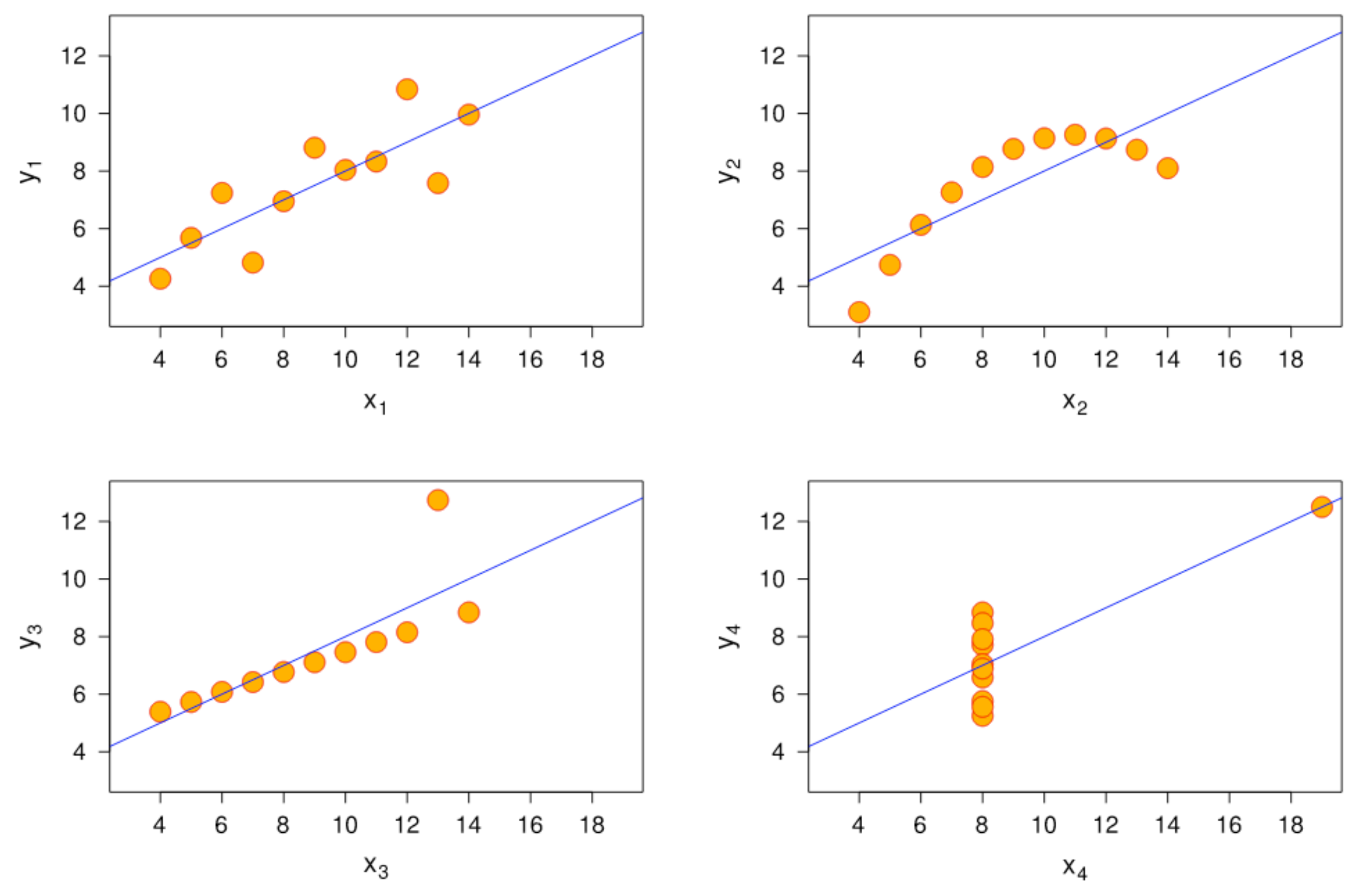
## Why Visualize?



http://upload.wikimedia.org/wikipedia/commons/b/b6/Anscombe.svg

Anscombe’s quartet, developed by Francis Anscombe to demonstrate the importance of graphing and the effect of outliers on statistical properties

# Why Visualize?



| Identical Statistics | |
|---|---|
| X mean | 9 |
| X variance | 11 |
| Y mean | 7.5 |
| Y variance | 4.122 |
| X/Y correlation | 0.816 |

http://upload.wikimedia.org/wikipedia/commons/b/b6/Anscombe.svg

# About this tutorial

## Recommended prerequisites:

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 Enaml. If you caught even just the basics of the previous tutorial on Enaml, you should be fine.

# About this tutorial

## Recommended prerequisites:

- Know a little numpy

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 Enaml. If you caught even just the basics of the previous tutorial on Enaml, you should be fine.

## About this tutorial

Recommended prerequisites:

- Know a little numpy
- Know a little about plotting data

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 Enaml. If you caught even just the basics of the previous tutorial on Enaml, you should be fine.

## About this tutorial

Recommended prerequisites:

- Know a little numpy
- Know a little about plotting data
- Know a little bit about GUI programming (in the context of Enaml)

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 Enaml. If you caught even just the basics of the previous tutorial on Enaml, you should be fine.

## Goals

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

- create Chaco plots of various types
- create a custom tool that interacts with the mouse
- understand the underlying architecture

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.

## Traits

```
from traits.api import HasTraits, Str, Float, List, Instance

class Department(HasTraits):
    name = Str

    def _name_default(self):
        return “Sales”

class Employee(HasTraits):
    first_name = Str
    last_name  = Str
    salary     = Float
    department = Instance(Department)

    def _department_default(self):
        return Department()

class Manager(Employee):
    employees = List(Employee)
```

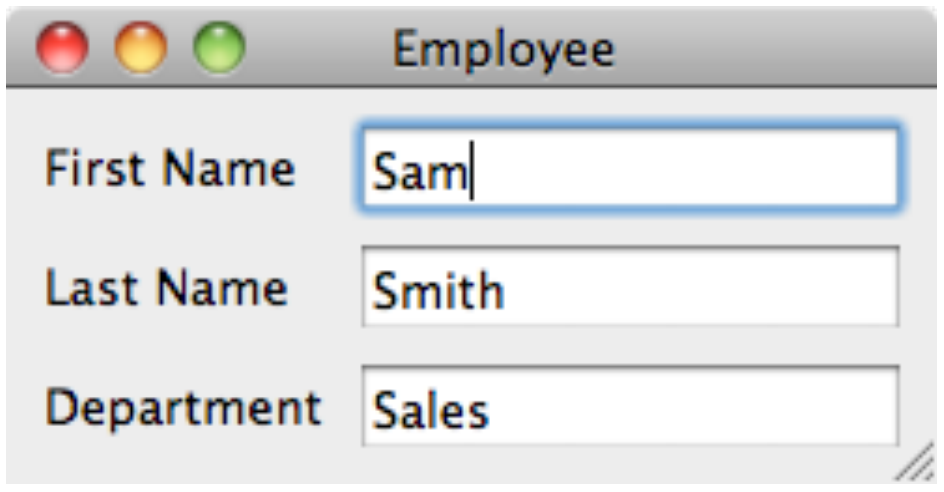
## Traits

```
from traits.api import HasTraits, Str, Float, List, Instance

class Department(HasTraits):
    name = Str

    def _name_default(self):
        return “Sales”

class Employee(HasTraits):
    first_name = Str
    last_name  = Str
    salary     = Float
    department = Instance(Department)

    def _department_default(self):
        return Department()

class Manager(Employee):
    employees = List(Employee)
```

# Enaml

- Build User Interfaces with a Declarative syntax
- Full bi-directional data binding

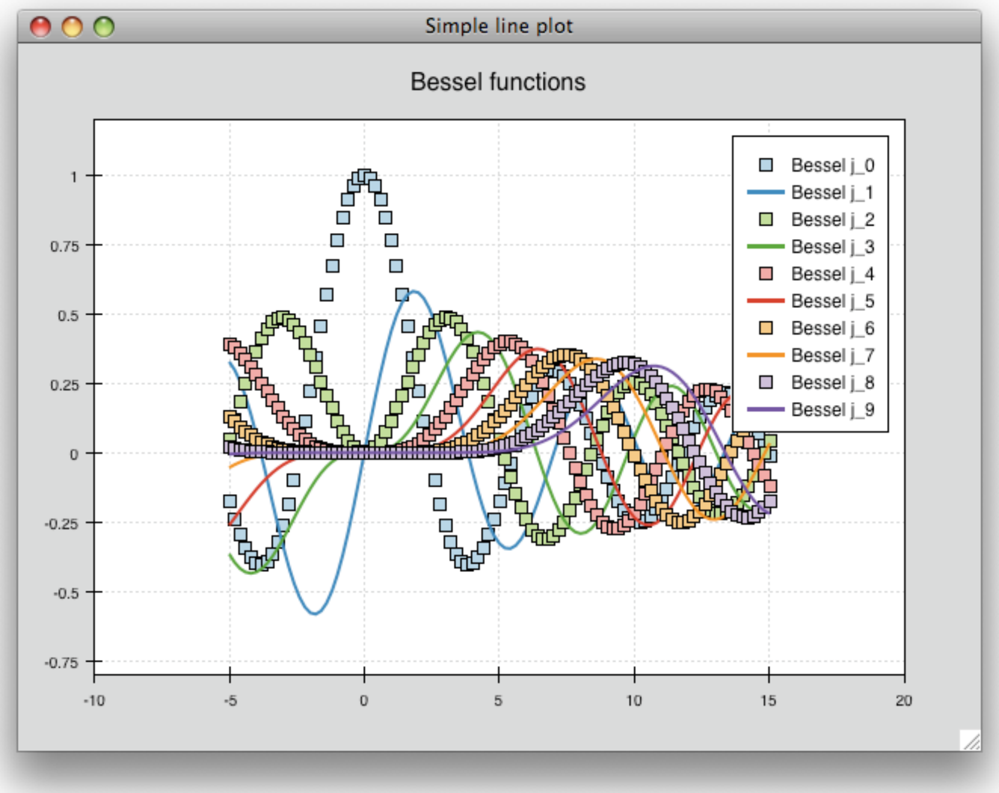
```
emp = Employee(first_name='Sam', last_name='Smith')

enamldef Main(MainWindow):
    title = "Employee"
    attr model = emp
    Form:
        Label:
            text = "First Name"
        Field:
            value := model.first_name
        Label:
            text = "Last Name"
        Field:
            value := model.last_name
        Label:
            text = "Department"
        Field:
            value := model.department.name
```

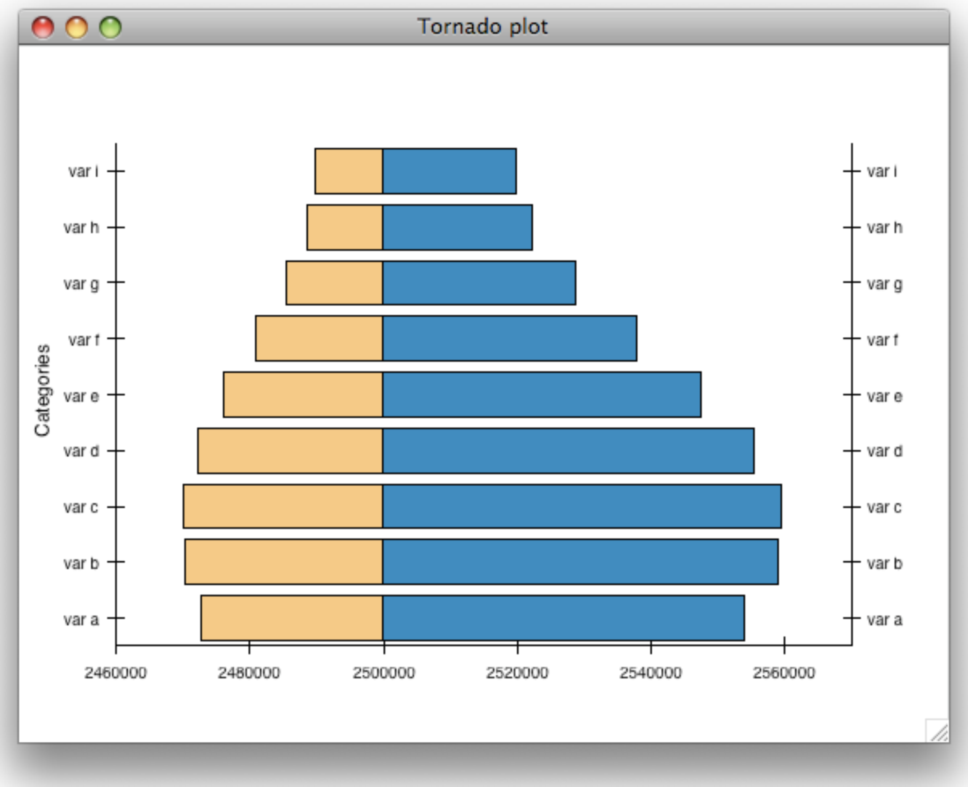


|  |  |
|---|---|
| First Name | Sam |
| Last Name | Smith |
| Department | Sales |

# What is Chaco?

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





ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Monday, June 18, 2012

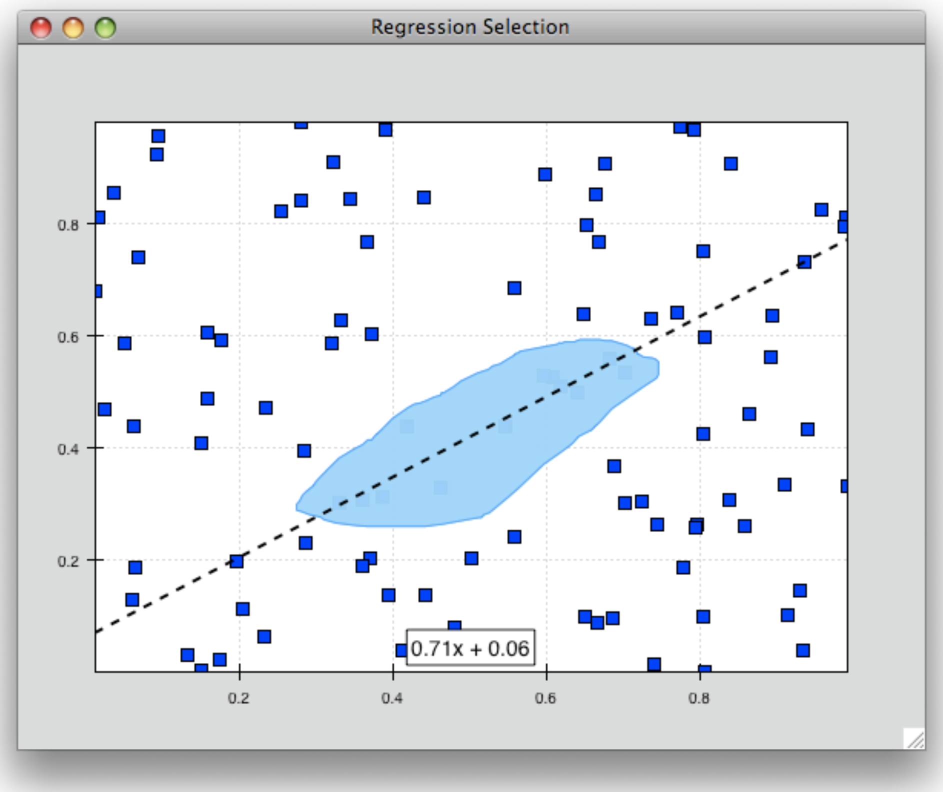
...but you can also build interactive visualization applications that let you explore your data.

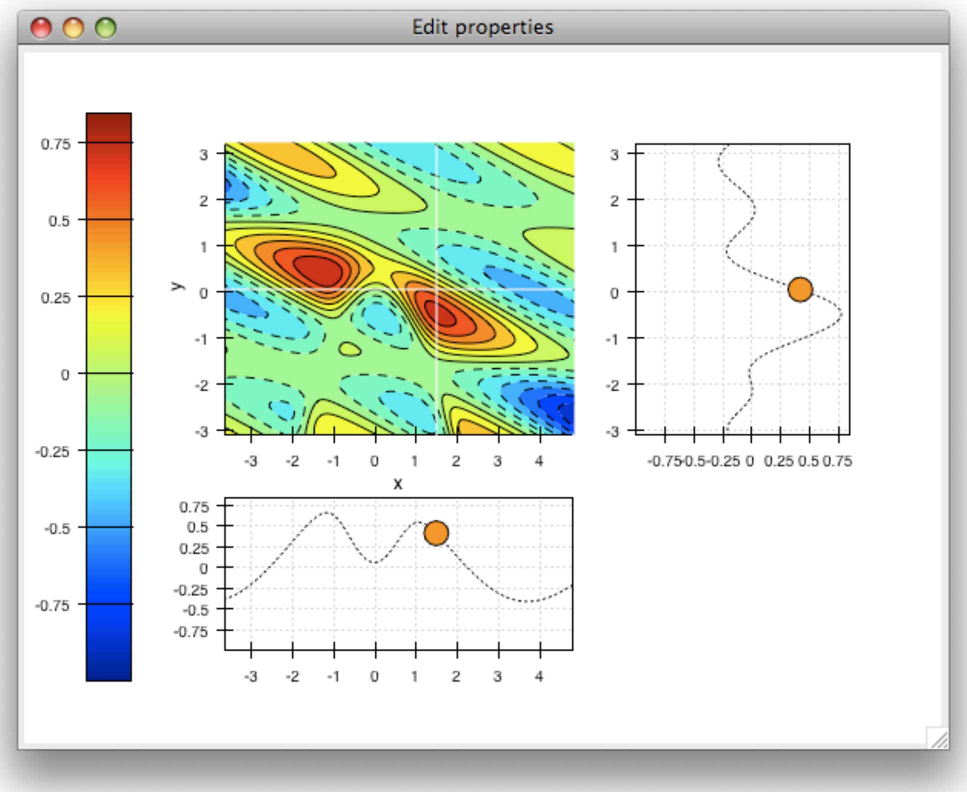
I'm going to take a few minutes now to show you what I mean by "interactive visualization". These are all examples that ship with Chaco or that you can download off of the project website. I'll talk a little bit about what characteristics of Chaco each example highlights, but I'm not going to get into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

(The following examples are all standard Chaco examples)
- data_labels.py: pan & zoom, data label movement
- two_plots.py: coordinated panning, zooming
- regression.py: selection of underlying datapoints
- line_plot_hold.py: different rendering styles
- inset_plot.py: can move the smaller plot around, linked axes
- cmap_scatter.py
- zoom_plot.py
- data_cube.py: magnetic resonance of human brain from UNC
- scalar_image_function_inspector.py
- spec_waterfall.py

# What is Chaco?

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





Monday, June 18, 2012

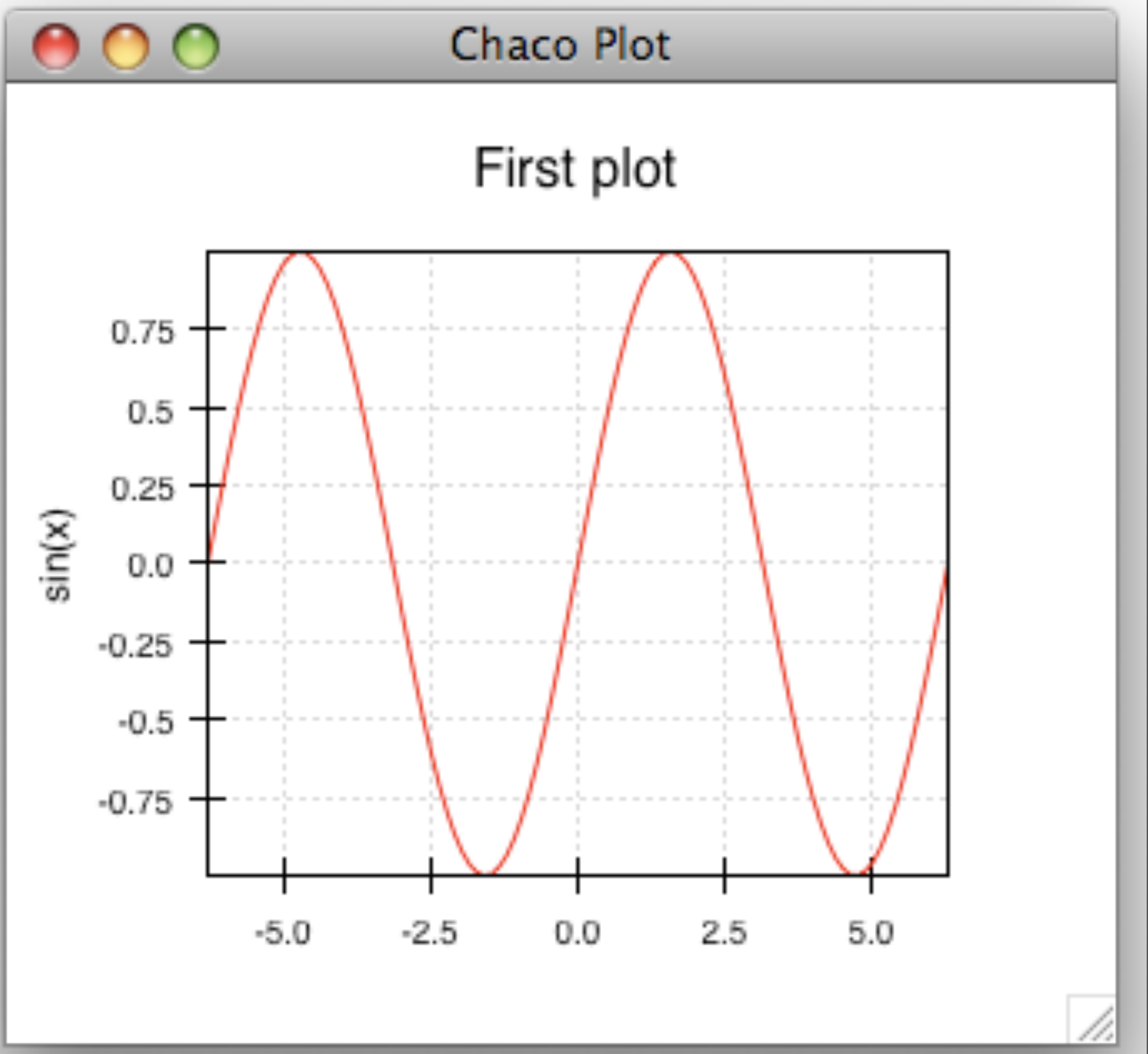
...but you can also build interactive visualization applications that let you explore your data.

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-based” 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()
```



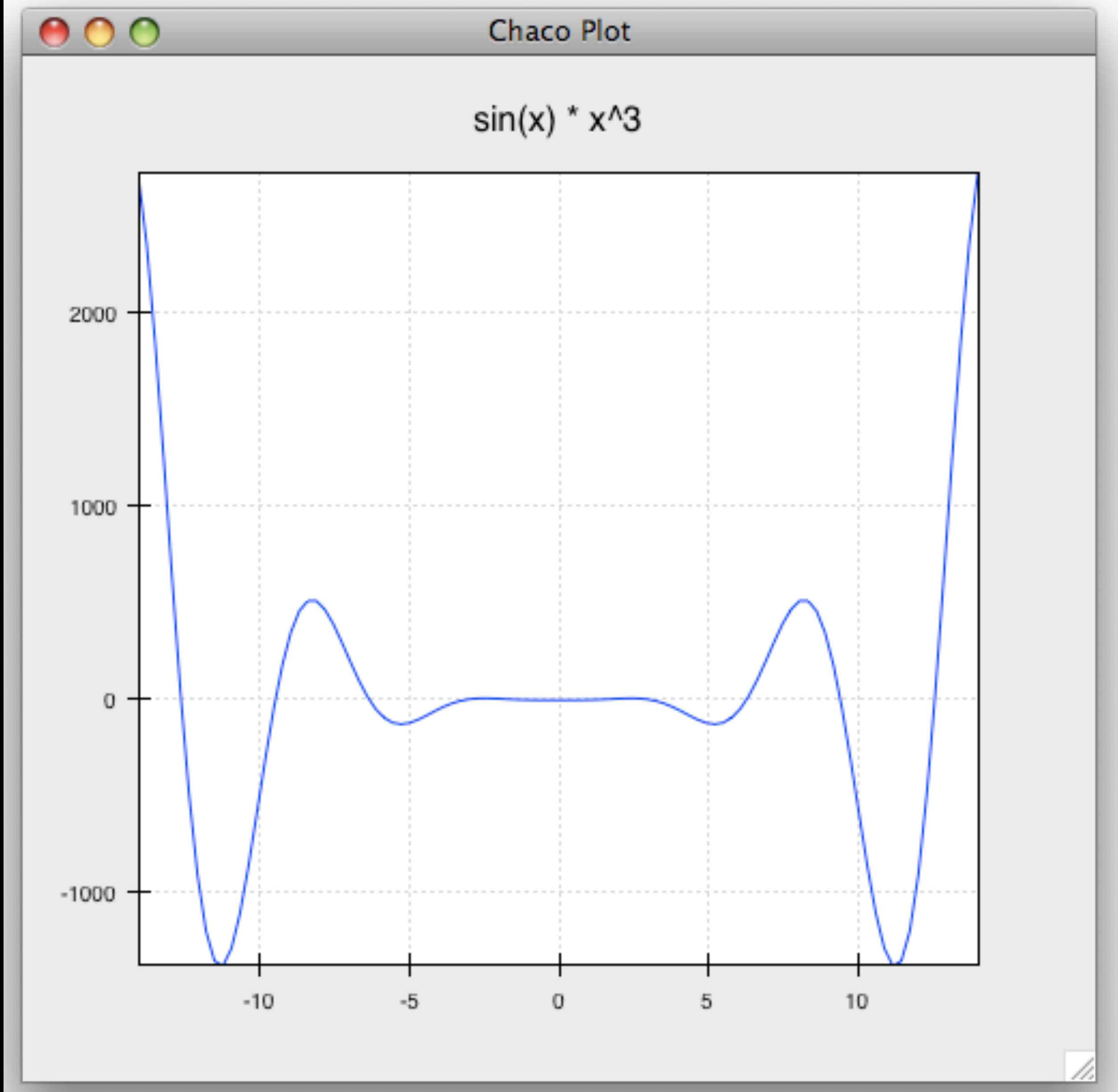
I distinguish between “static” plots and “interactive visualizations” because depending on which one you are trying to do, your code (and the code of the plotting framework you are using) will look very different.

Here is a simple example of what I call the “script-based” 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-based” Plotting

```
class LinePlot(HasTraits):
    plot = Instance(Plot)

    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="line",
                  color="blue")
        plot.title = "sin(x) * x^3"
        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```



Monday, June 18, 2012

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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```

Monday, June 18, 2012

This is the boilerplate needed to show the window containing the plot. 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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```

Monday, June 18, 2012

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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = "Chaco Plot"
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Monday, June 18, 2012

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)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```

Monday, June 18, 2012

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

# First Plot

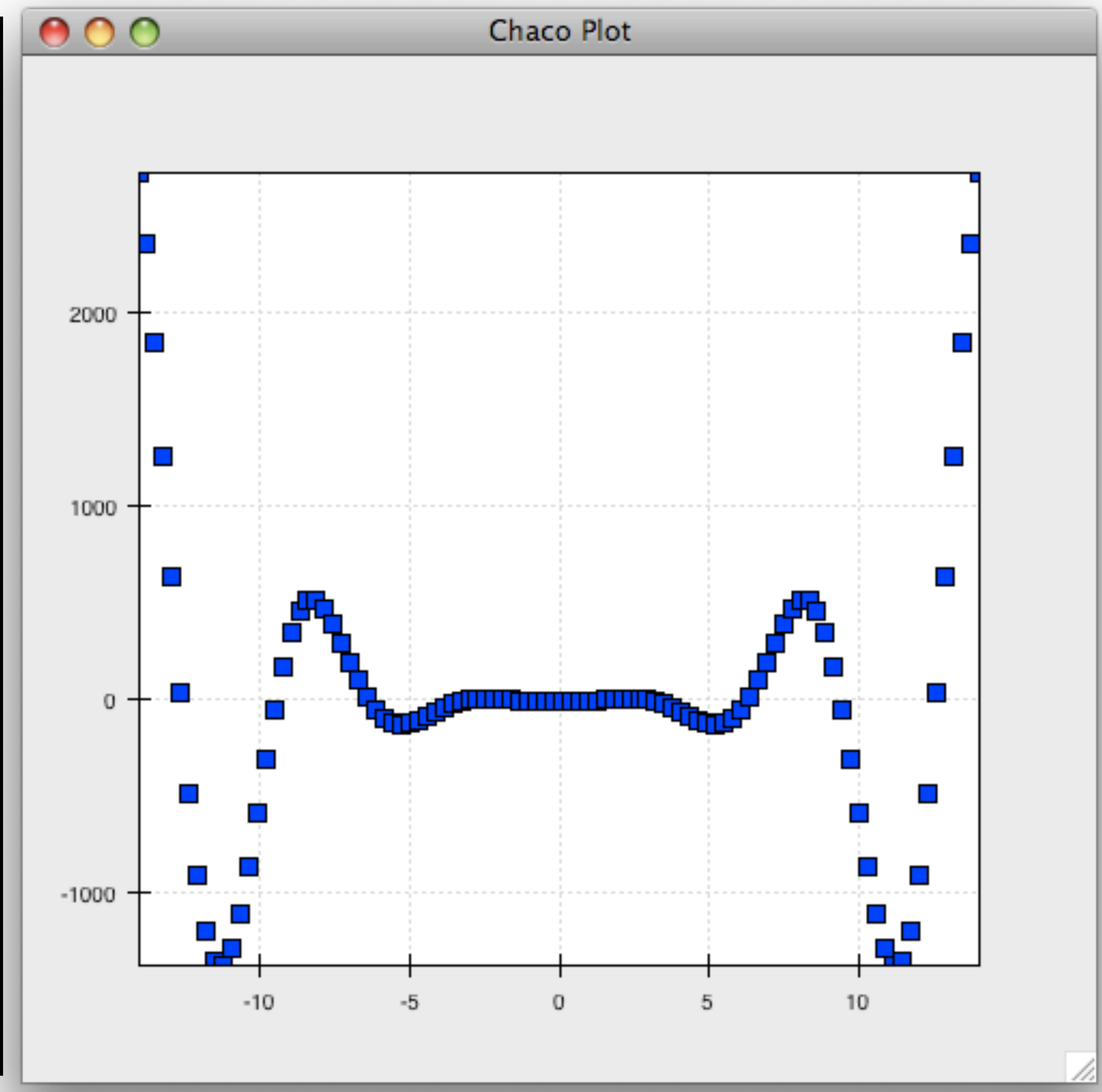
```
class LinePlot(HasTraits):

    plot = Instance(Plot)

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = LinePlot()
    EnableCanvas:
        component << model.plot
```

Monday, June 18, 2012

And finally, we return the constructed plot.

# Scatter Plot

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

    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")
        return plot

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = ScatterPlot()
    EnableCanvas:
        component << model.plot
```



ENTHOUGHT SCIENTIFIC COMPUTING SOLUTIONS

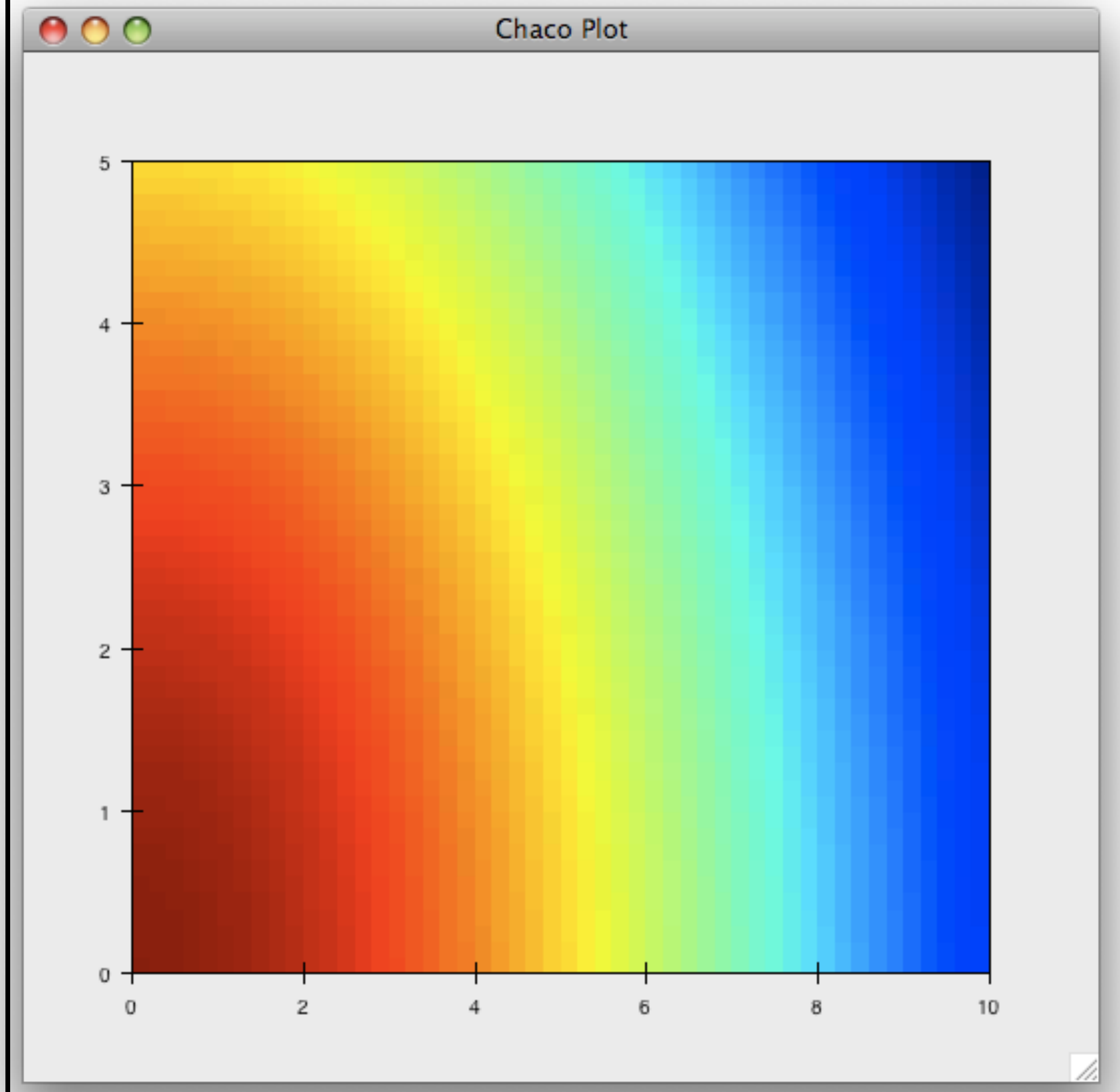
Monday, June 18, 2012

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)

    def _plot_default(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)
        return plot

enamldef Main(MainWindow):
    title = "Chaco Plot"
    attr model = ImagePlot()
    EnableCanvas:
        component << model.plot
```



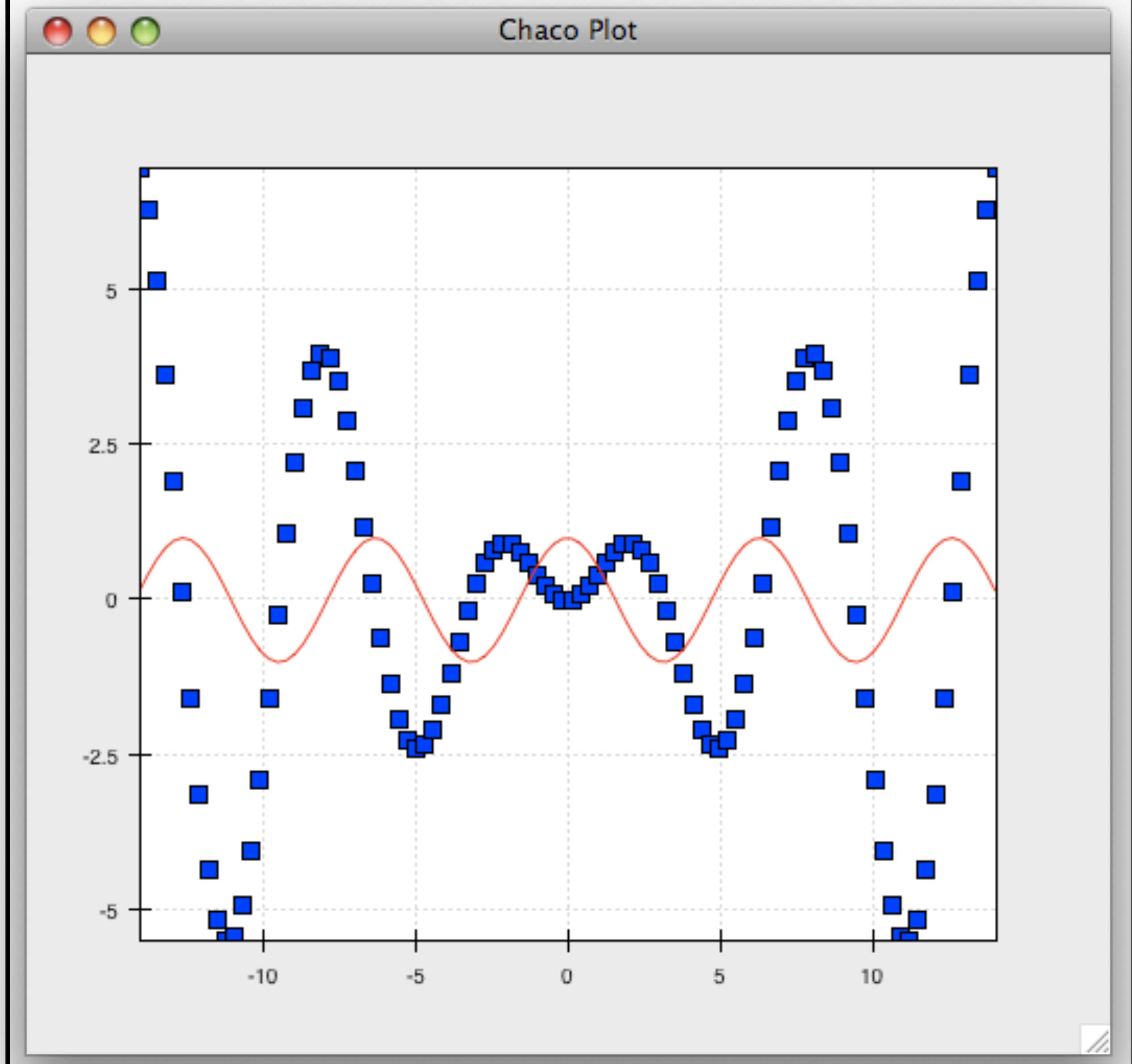
Monday, June 18, 2012

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)

    def _plot_default(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")
        return plot

enamldef Main(MainWindow):
    title = "Chaco Plot"
    attr model = OverlappingPlot()
    EnableCanvas:
        component << model.plot
```



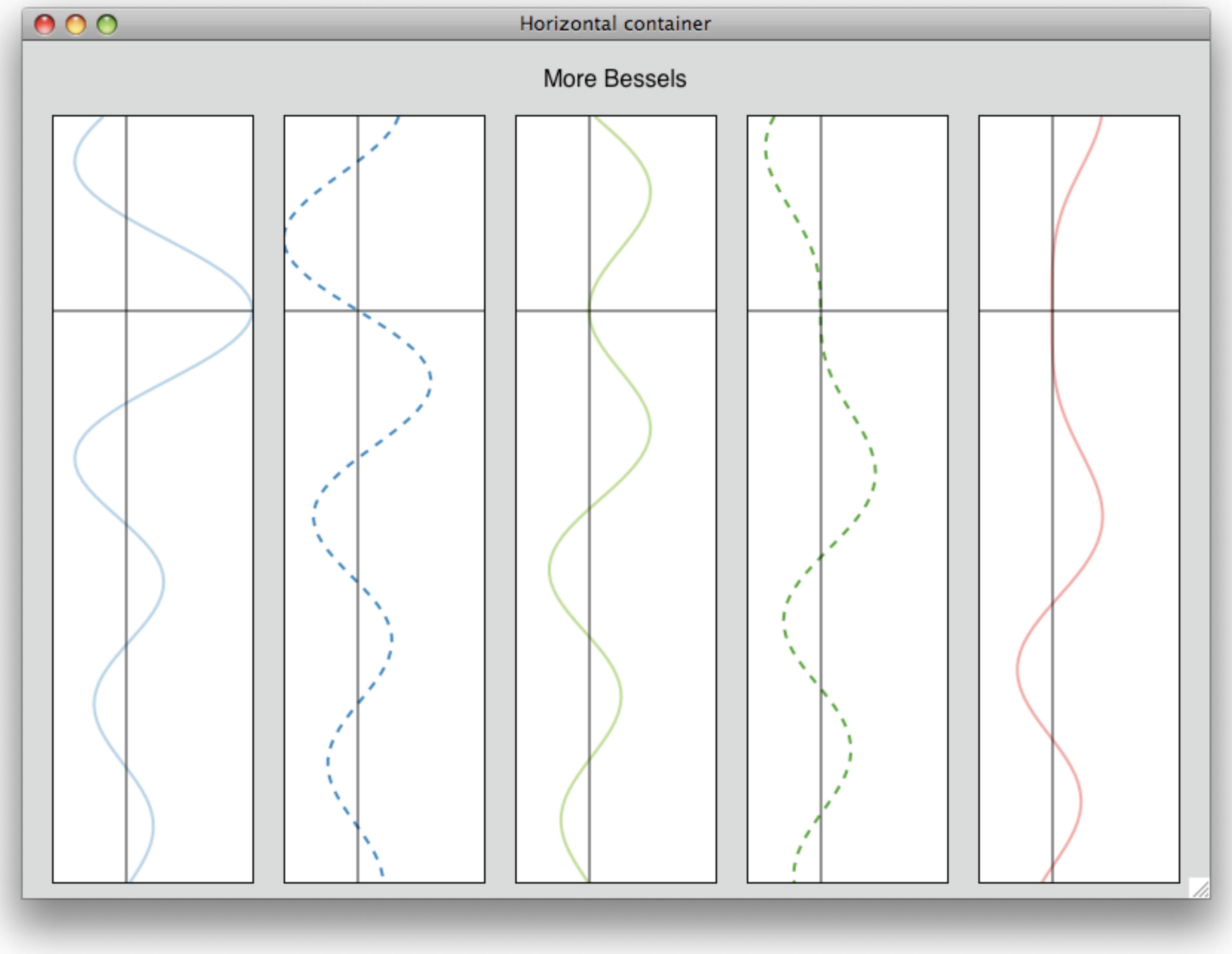
Monday, June 18, 2012

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

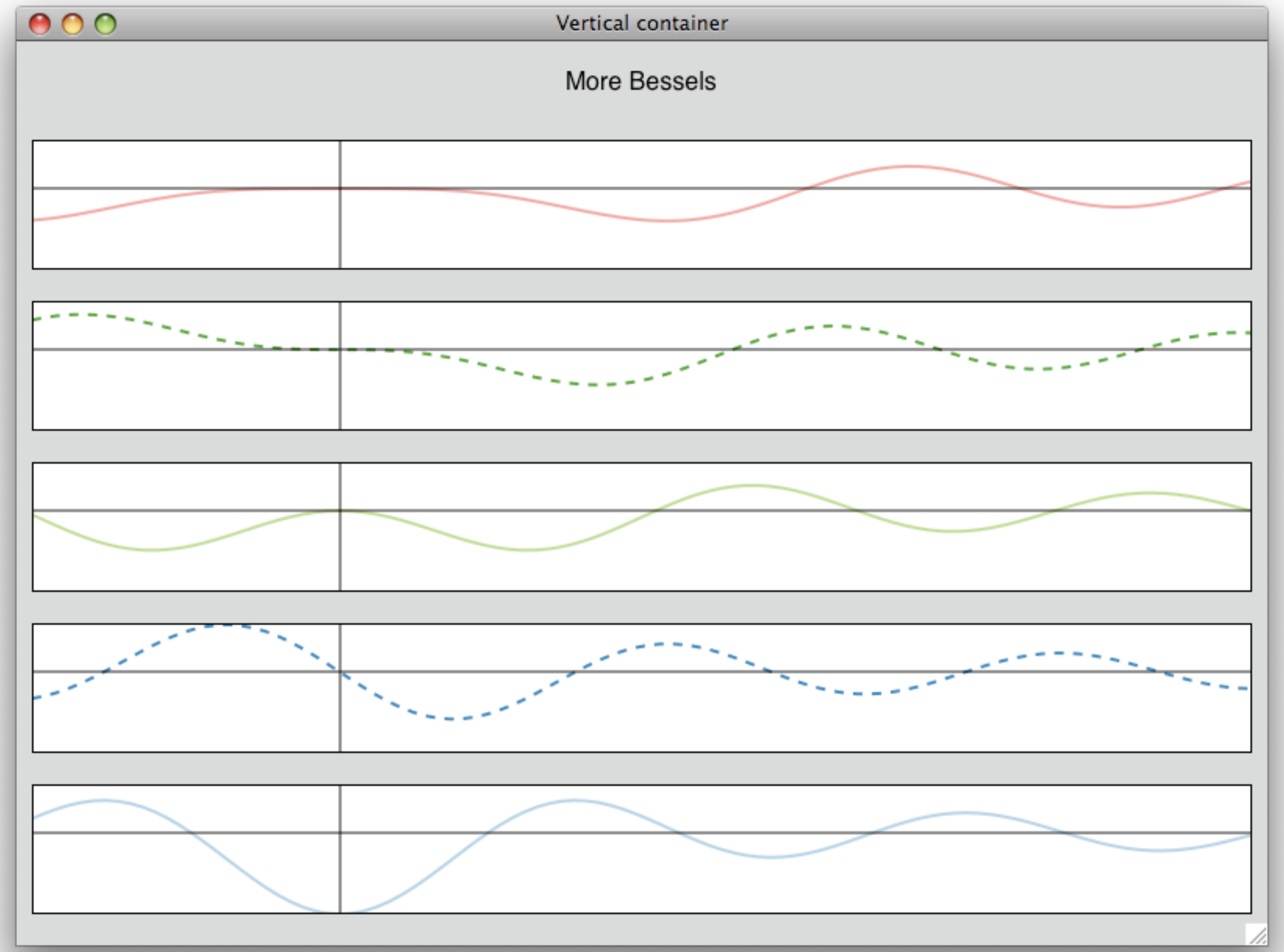


Monday, June 18, 2012

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

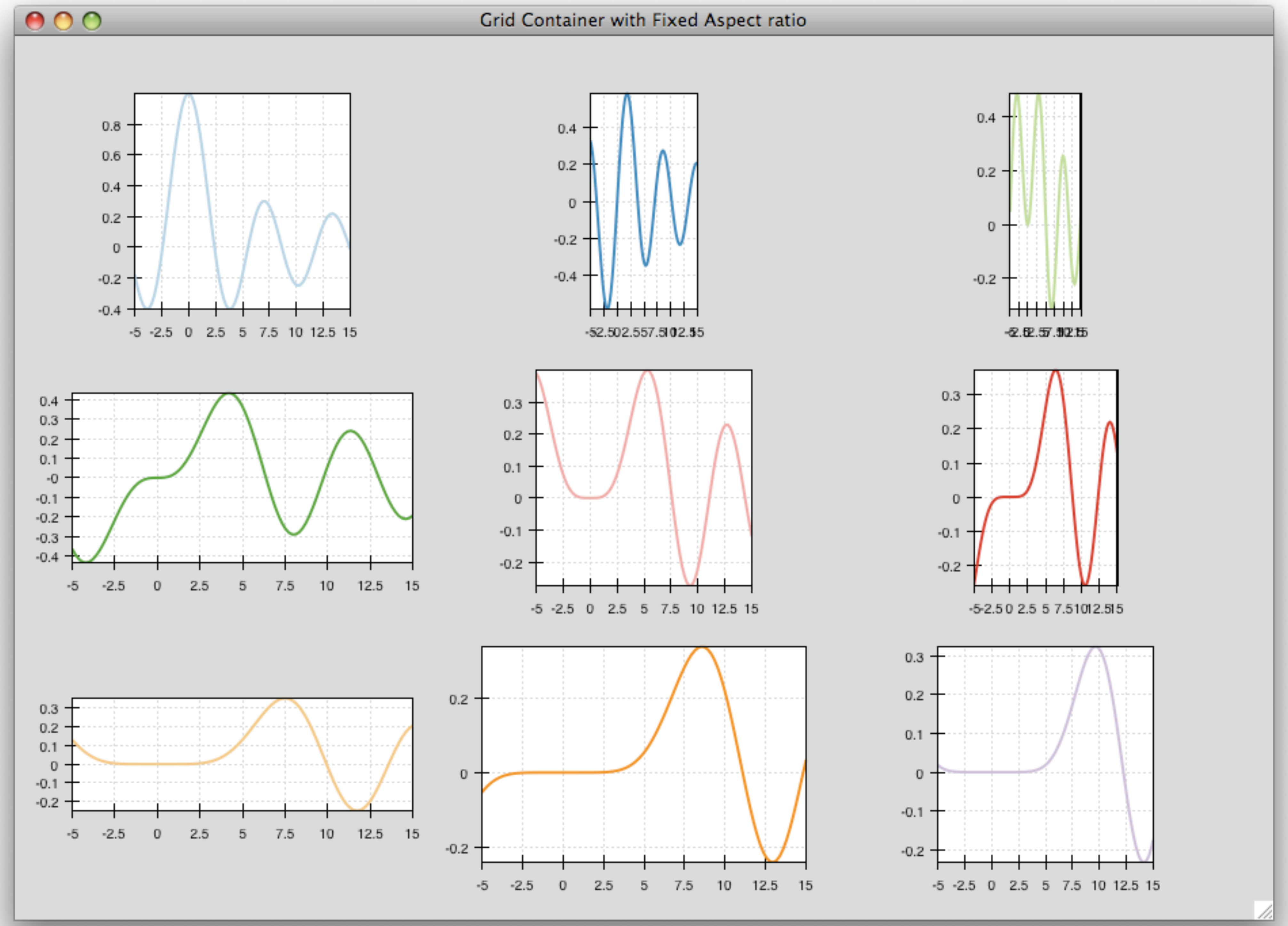
Horizontal containers place components horizontally.

# VPlotContainer



Vertical containers array components vertically.

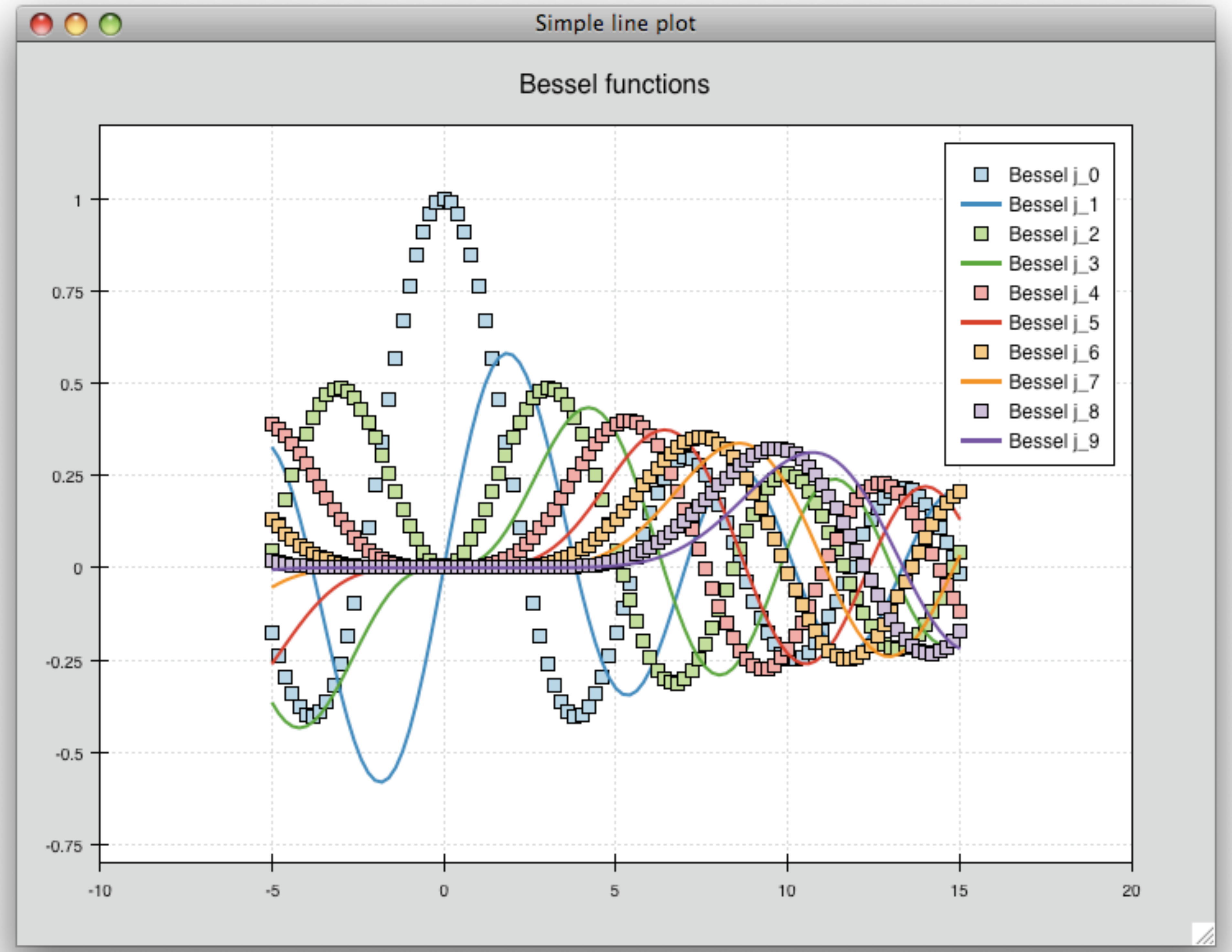
# GridContainer



Monday, June 18, 2012

Grid containers lay plots out in a grid.

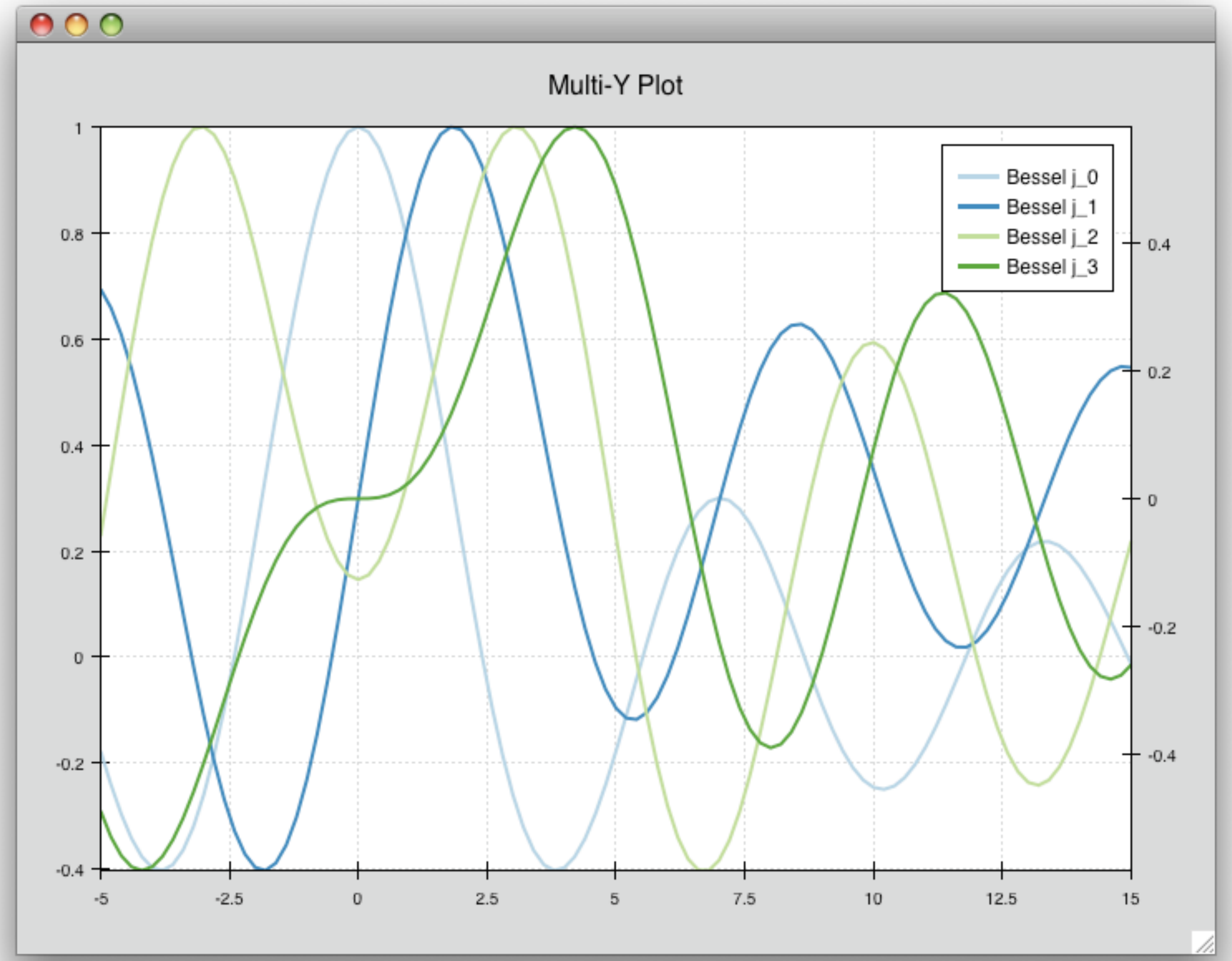
# OverlayPlotContainer



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



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)

    def _plot_default(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)
        return container

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = ContainerExample()
    EnableCanvas:
        component << model.plot
```

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

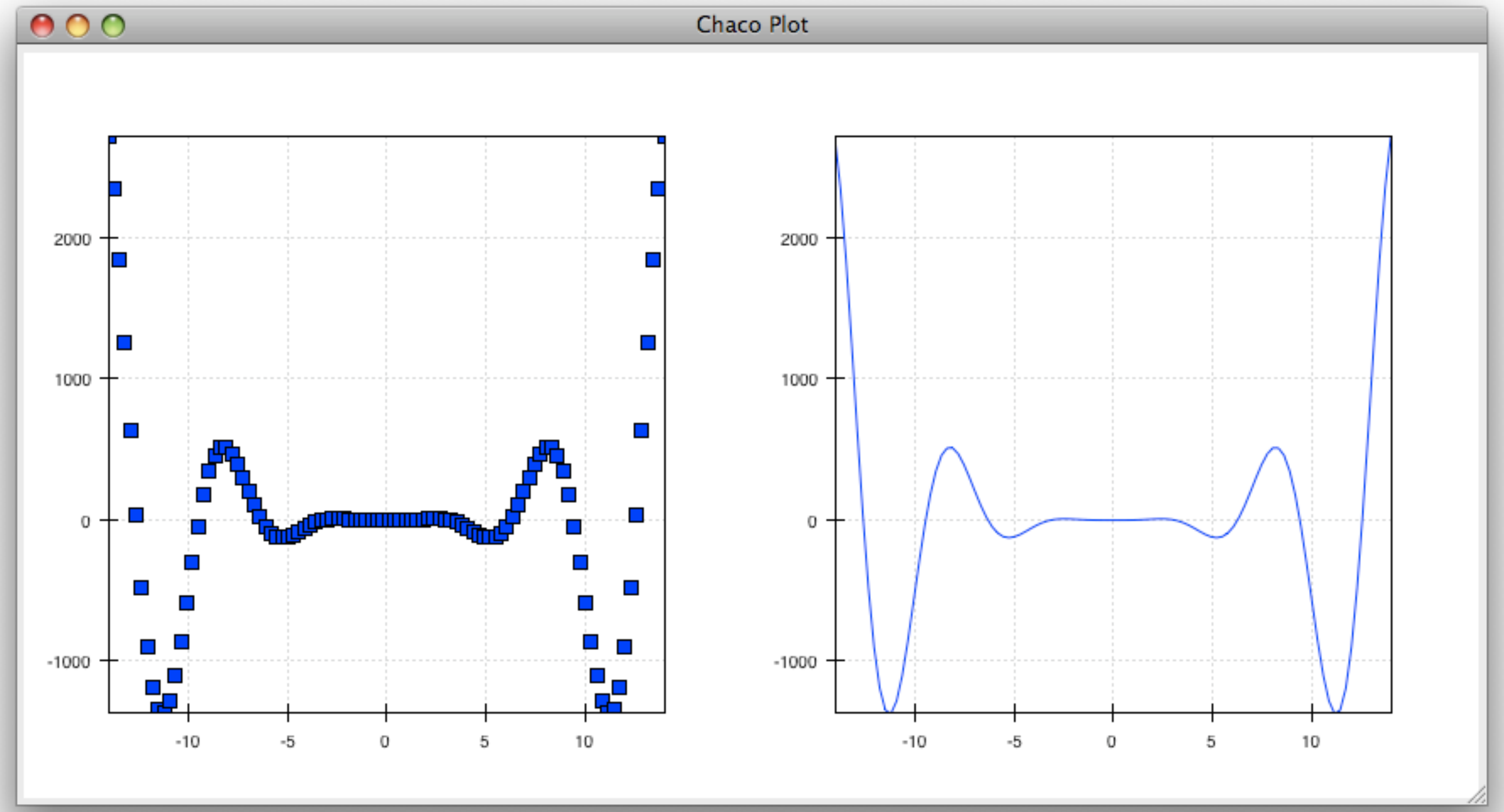
```
class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)

    def _plot_default(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)
        return container

enamldef Main(MainWindow):
    title = “Chaco Plot”
    attr model = ContainerExample()
    EnableCanvas:
        component << model.plot
```

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

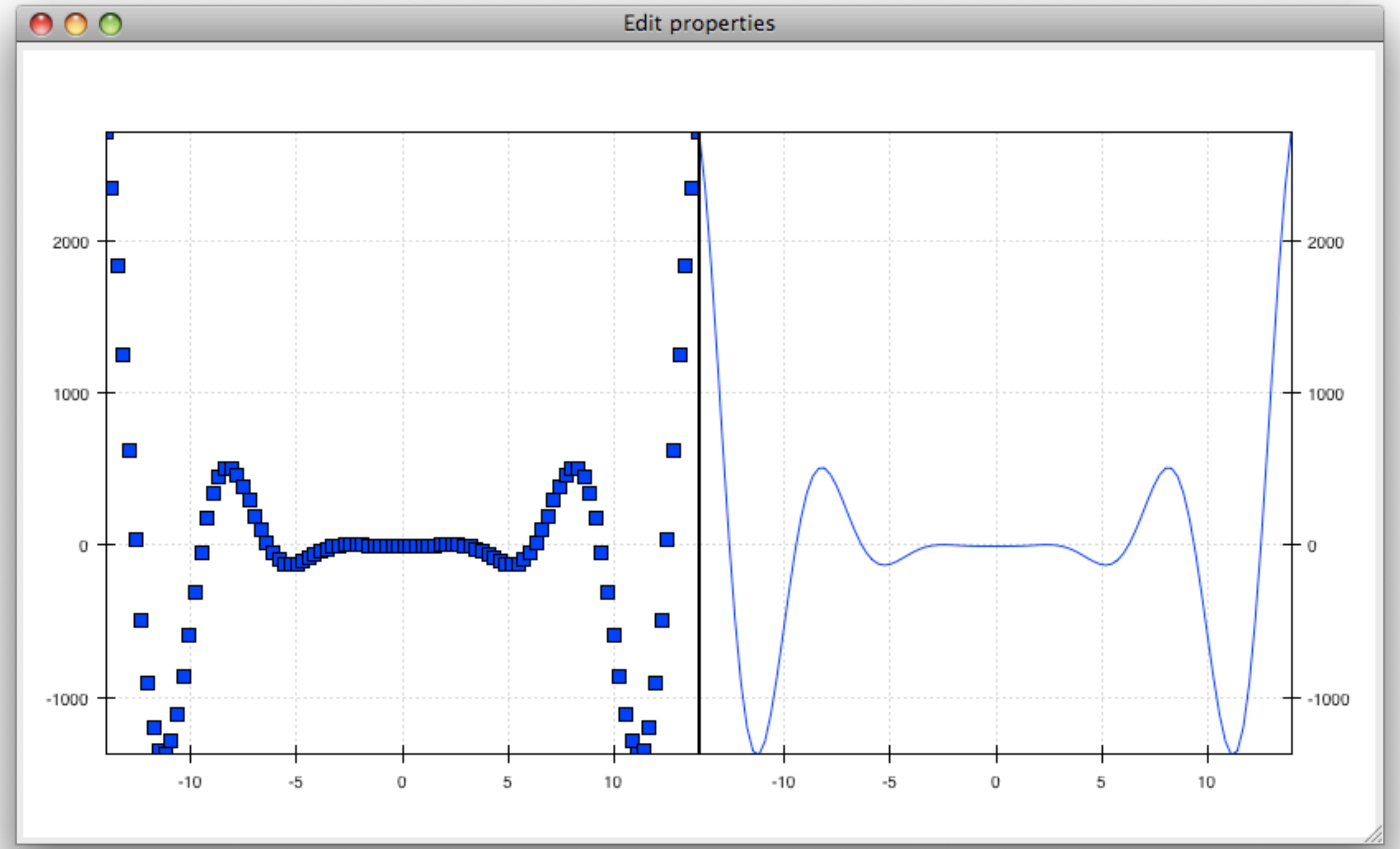
```
class ContainerExample(HasTraits):
    plot = Instance(HPlotContainer)

    def _plot_default(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"

        return 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.

# Tools

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

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

    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="line", color="blue")

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

        return plot

enamldef Main(MainWindow):
     ...
```

Monday, June 18, 2012

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.

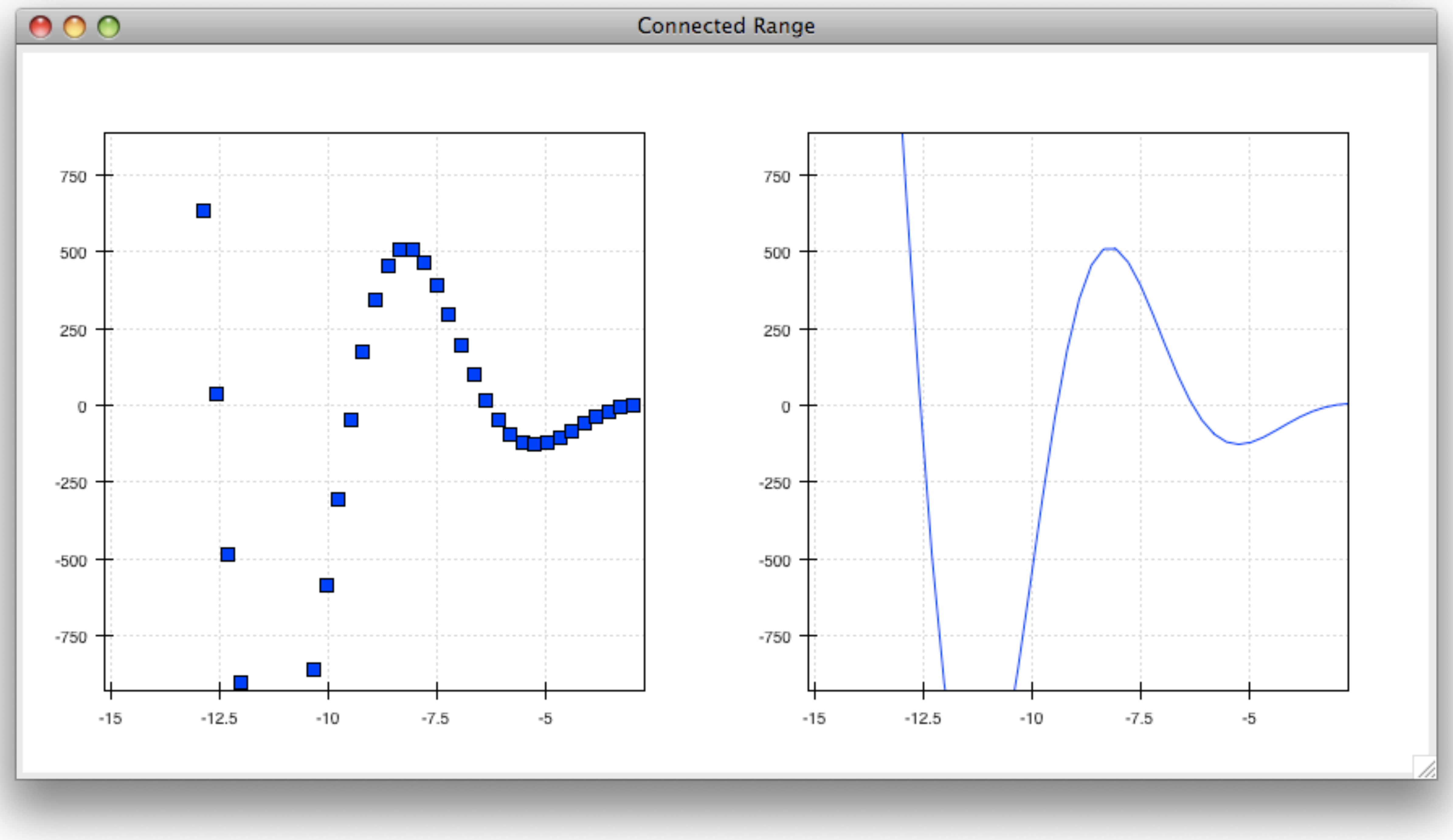
Thus far, none of the example plots we've built are interactive. You can't pan or zoom them. We'll now modify the LinePlot so we can pan and zoom.

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)

# Connected Plots



ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

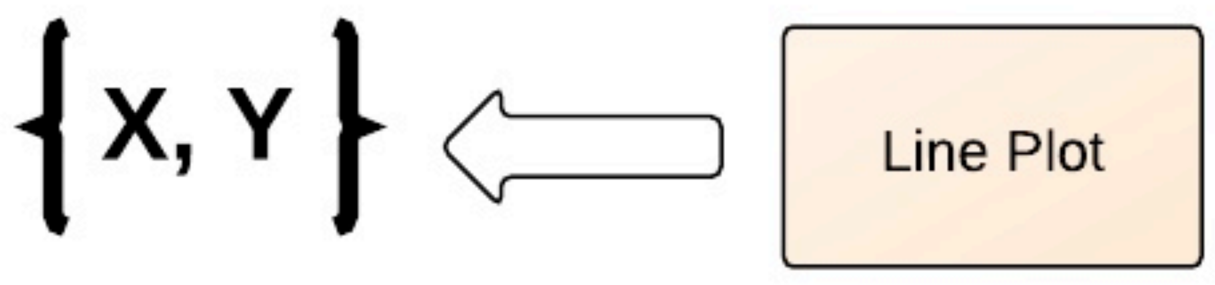
Monday, June 18, 2012

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

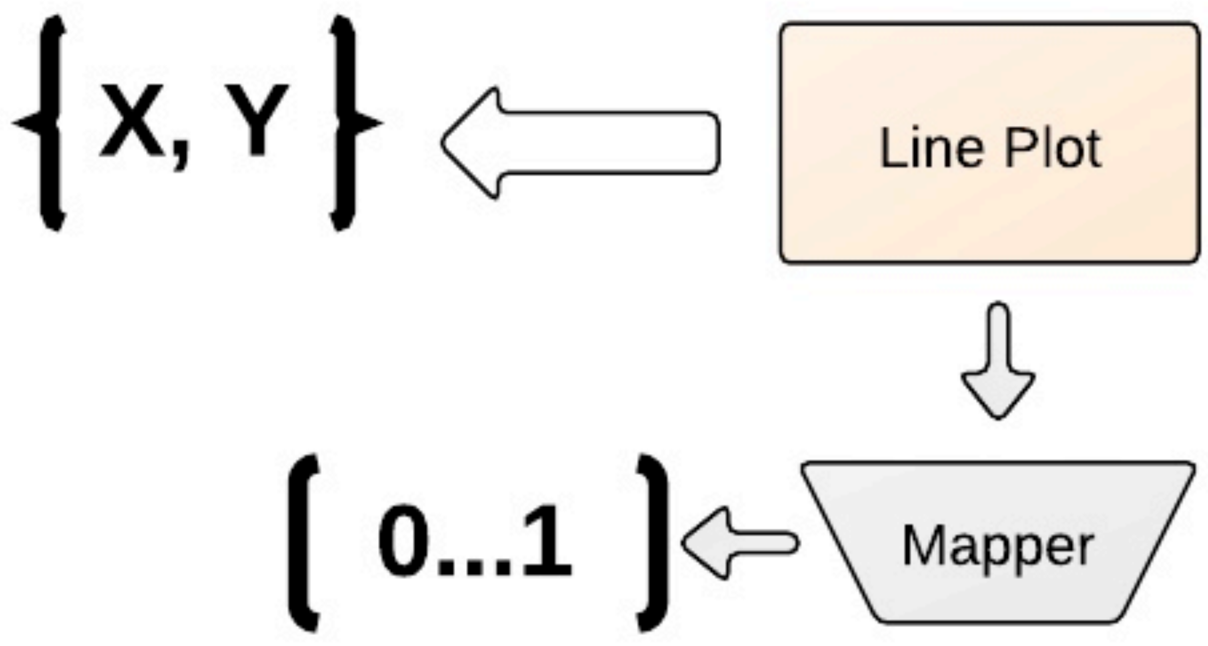
But one of the features of Chaco's architecture is that all the underlying components of a plot are live objects, connected via events. In the next set of examples, we'll look at how to hook some of those up.

First, we're going to make two separate plots look at the same data space region.

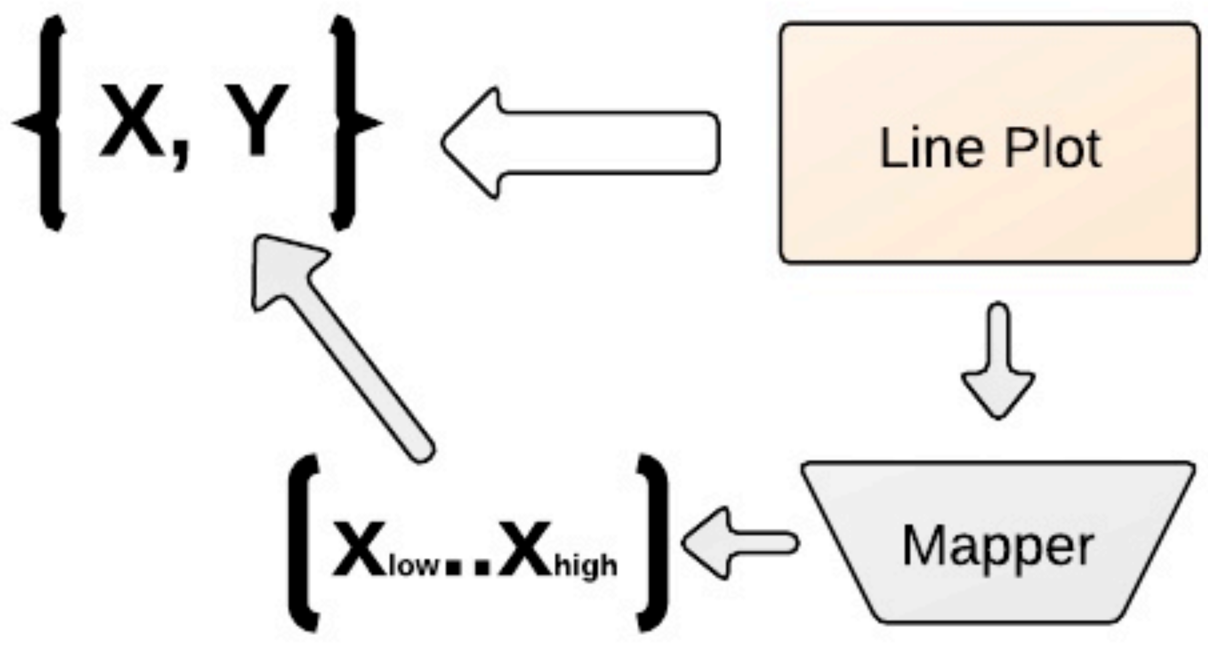
## Chaco architecture



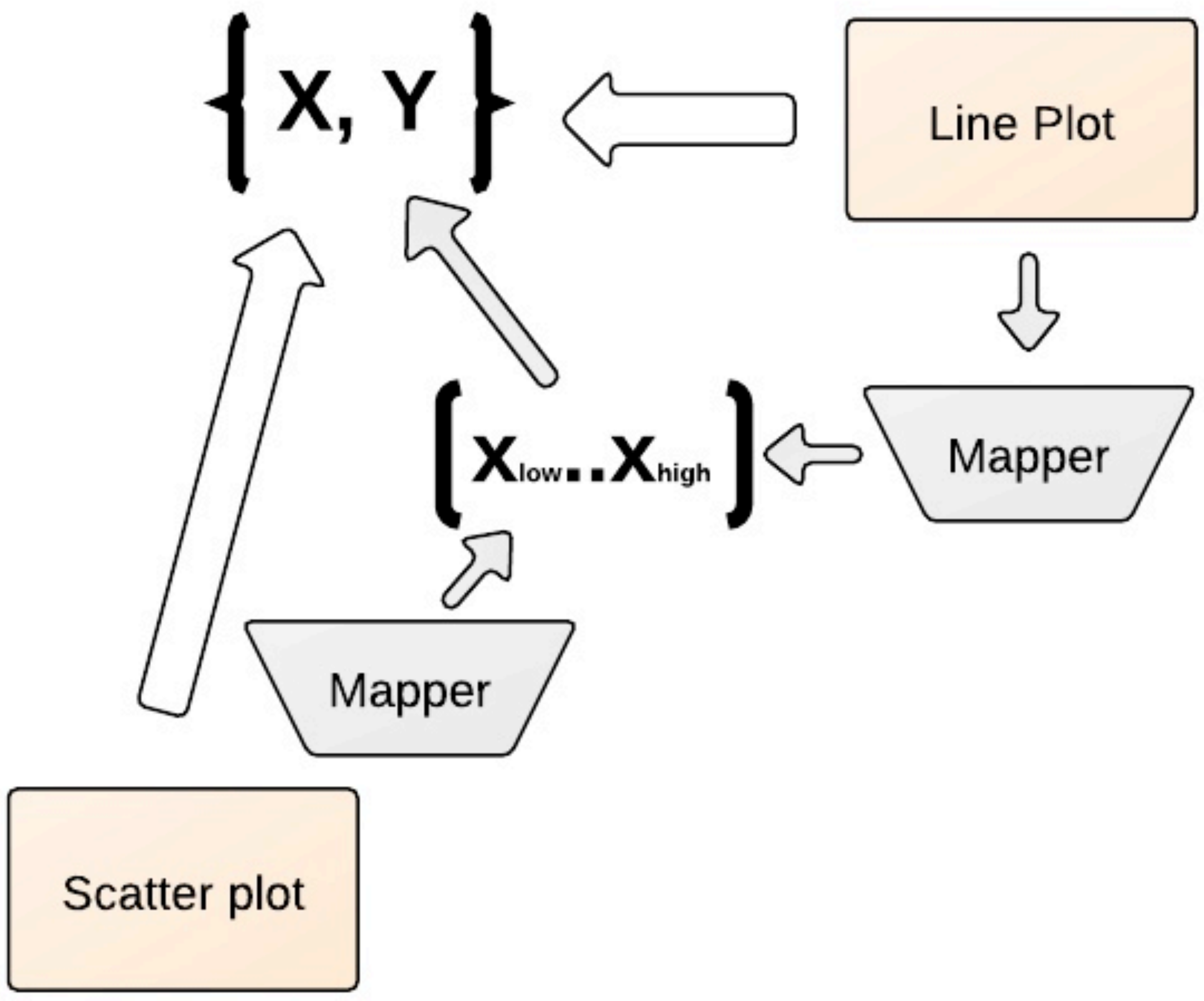
# Chaco architecture



# Chaco architecture



# Chaco architecture



## Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

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)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

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

## Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

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)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

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

# Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

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

## Connected Plots

```
class ConnectedRange(HasTraits):
    container = Instance(HPlotContainer)

    def _container_default(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)

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

        scatter.range2d = line.range2d

        return container
```

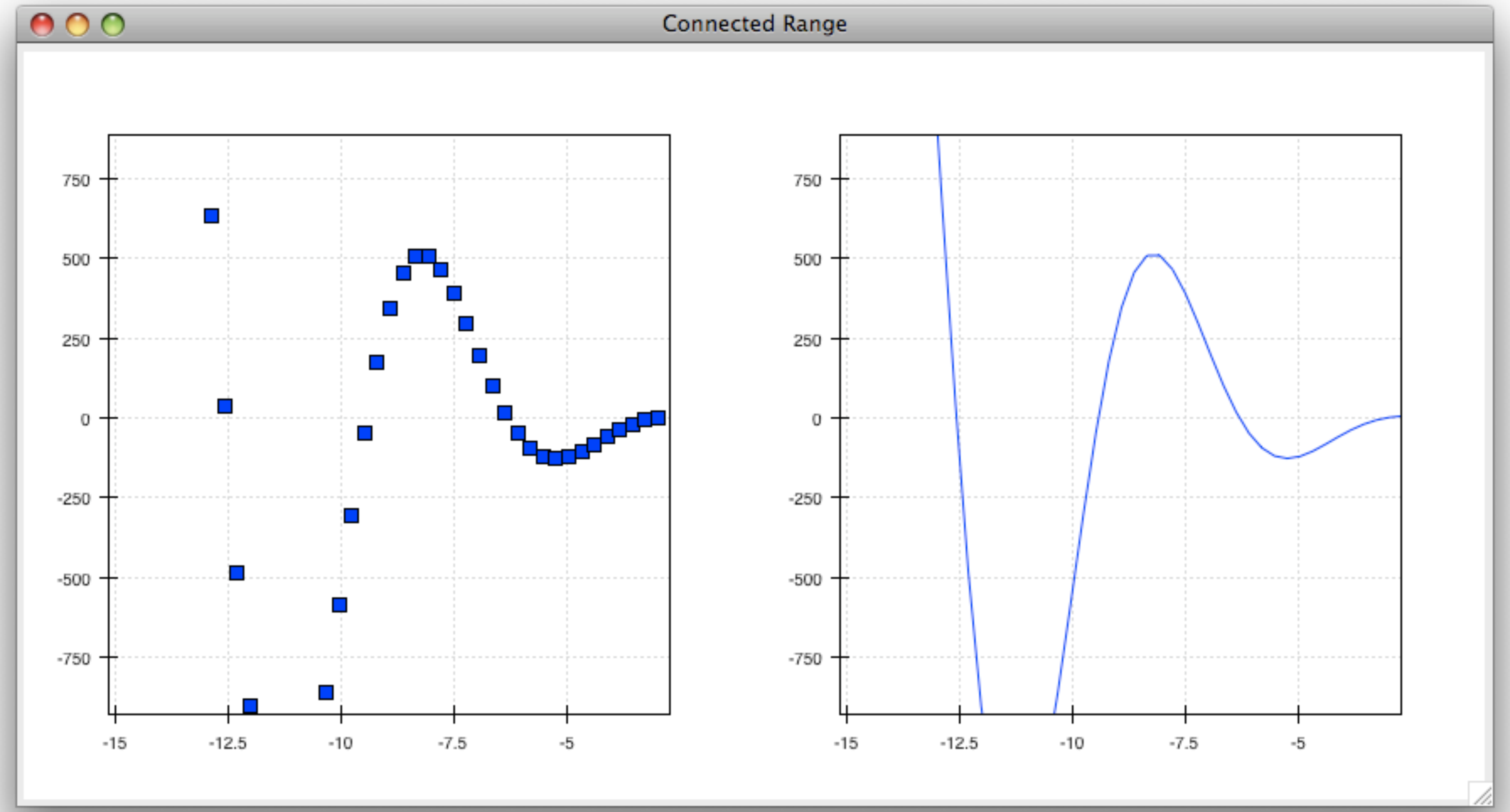
This final line is the magic one.

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)

    def _container_default(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)

        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

        return container
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Monday, June 18, 2012

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

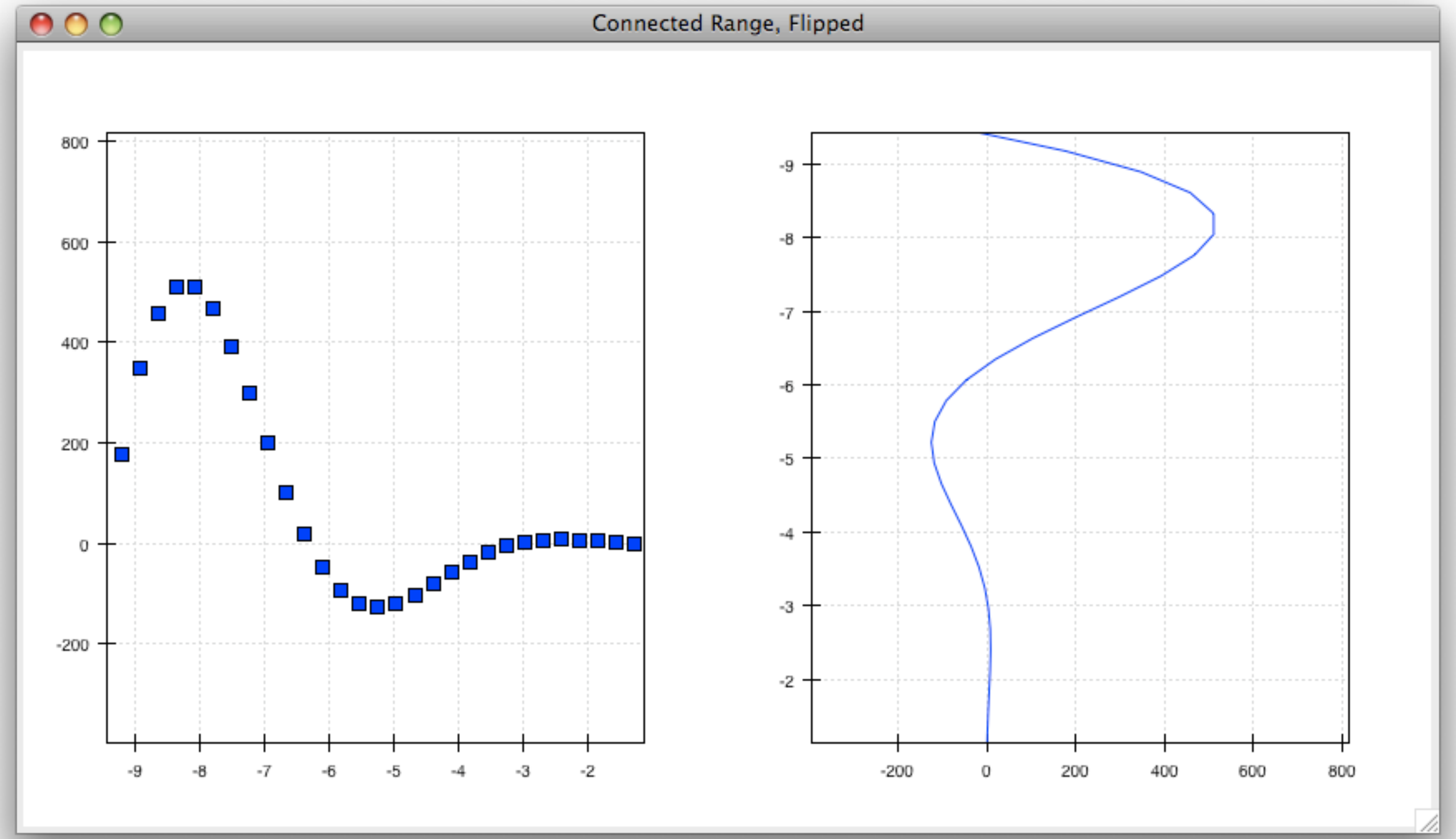
```
class FlippedExample(HasTraits):
    container = Instance(HPlotContainer)

    def _container_default(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")

        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

        return container
```

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



## Writing a custom Tool

```
from 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)

    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 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)

    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
```

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

To write a tool, we subclass from BaseTool.

# Writing a custom Tool

```
from 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)

    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

```
from enable.api import BaseTool

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


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

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Monday, June 18, 2012

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

```
from enable.api import BaseTool

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


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

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

```
from enable.api import BaseTool

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

Monday, June 18, 2012

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)

## Questions?

- Web page:
  https://github.com/enthought/chaco
- Gallery:
  http://code.enthought.com/projects/chaco/gallery.php
- Mailing lists:
  enthought-dev@enthought.com,
  chaco-users@enthought.com

ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS