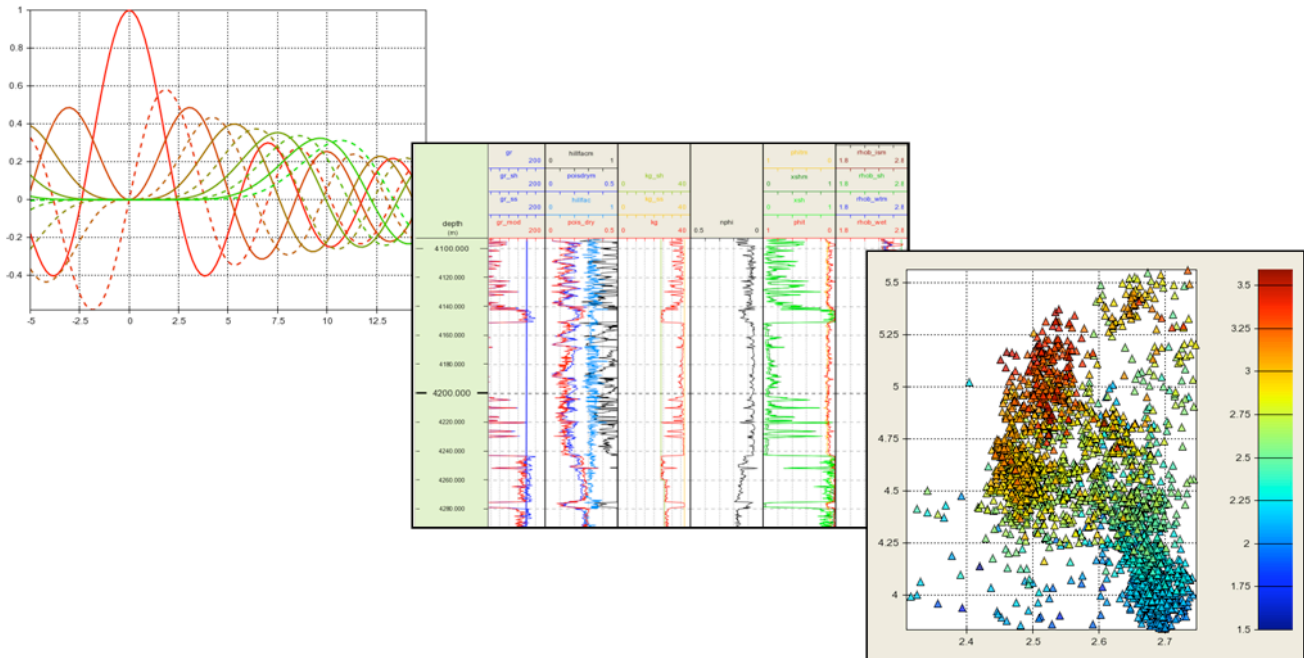


Data Exploration with Chaco



Structure of today's tutorial

- Overview of Chaco
- Basic plots and interactors
- Creating simple interactors
- Core concepts: data model, layout, interaction model
- More complex plots
- Walkthrough of some examples

Overview

Chaco is a *plot application toolkit* for Python. You use it to build stand-alone plotting applications, or embed it inside any application that needs to visualize numerical data.

Sample plotting applications:

- batch plotting of data (csv -> png)
- display for realtime data acquisition
- visual plot construction kit
- visual editor for tweaking input parameters to simulations
- mapping and GIS applications

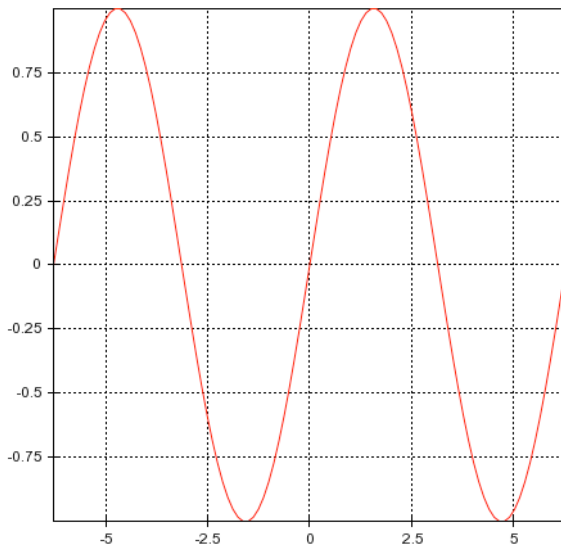
Chaco Features

- Different rendering layers, backbuffering
- Container model for layout and event dispatch
- Modular and extensible architecture
- Data model/filtering pipeline
- Vector drawing engine

A first look

```
1. # tutorial1.py
2.
3. # Create some data
4. from scipy import arange, pi, sin
5. numpoints = 100
6. step = 4*pi / numpoints
7. x = arange(-2*pi, 2*pi+step/2, step)
8. y = sin(x)
9.
10. # Create the plot and set its size
11. from enthought import chaco2
12. myplot = chaco2.create_line_plot((x,y), bgcolor="white", add_grid=True, add_axis=True)
13. myplot.padding = 50
14. myplot.bounds = [400,400]
15.
16. # Create a graphics context for offline rendering
17. plot_gc = chaco2.PlotGraphicsContext(myplot.outer_bounds)
18. plot_gc.render_component(myplot)
19. plot_gc.save("tutorial1.png")
```

tutorial1.py



Creating a window

```

1. # tutorial2.py
2. import wx
3. from enthought import chaco2
4.
5. # Import the plot from Tutorial 1
6. from tutorial1 import myplot
7.
8. from enthought.enable2.wx_backend import Window
9. class PlotFrame(wx.Frame):
10.     def __init__(self, *args, **kw):
11.         kw["size"] = (600,600)
12.         wx.Frame.__init__( * (self,) + args, **kw )
13.
14.         # Use Enable.Window to bridge between the vector-drawing world of Chaco and WX
15.         self.plot_window = Window(parent=self, component=myplot)
16.
17.         sizer = wx.BoxSizer(wx.HORIZONTAL)
18.         sizer.Add(self.plot_window.control, 1, wx.EXPAND)
19.         self.SetSizer(sizer)
20.         self.SetAutoLayout(True)
21.         self.Show(True)
22.         return

```

Opening the window

Normal python (standalone app)

```

1. from tutorial2 import PlotFrame
2. import wx
3. app = wx.PySimpleApp()
4. frame = PlotFrame(None)
5. app.MainLoop()

```

From within IPython

```

1. # This requires invoking IPython as "ipython -pylab" or "ipython -wthread"
2.
3. from tutorial2 import PlotFrame
4. frame = PlotFrame(None)

```

Fun from within IPython

We can tweak plot attributes on the fly:

```

1. plot.vgrid.visible = False
2. plot.x_axis.visible = False
3. plot.request_redraw()

```

Fun from within IPython

We can even write functions to do the tweaking for us!

```
1. # tutorial2_ipython.py
2.
3. def ytitle(text):
4.     plot.y_axis.title = text
5.     plot.request_redraw()
6.
7. def xrange(low, high):
8.     plot.x_mapper.range.low = low
9.     plot.x_mapper.range.high = high
10.    plot.request_redraw()
11.
12. def yrange(low, high):
13.     plot.y_mapper.range.low = low
14.     plot.y_mapper.range.high = high
15.    plot.request_redraw()
```

The Joy of Traits

Since all Chaco primitives use traits, we can easily bring up property sheets.

Adding a basic interactor

All Chaco components can have tools on them, and any events they receive get forwarded on to their tools.

```
1. # tutorial3.py
2.
3. from tutorial2 import myplot, PlotFrame, main
4.
5. from enthought.chaco2.tools import PanTool
6.
7. myplot.tools.append(PanTool(myplot))
```

Adding zoom

Because the zoom tool draws a visible "zoom box", we need to add it to the list of overlays instead of the list of tools. (It will still get events.)

```
1. # tutorial4.py
2.
3. from tutorial2 import myplot, PlotFrame, main
4.
5. from enthought.chaco2.tools import SimpleZoom
6.
7. # Create an instance of the tool and add it as an overlay..
8. myplot.overlays.append(SimpleZoom(myplot, tool_mode="box", always_on=True))
```

Coordinating different tools

PanTool and SimpleZoom both key off the left mouse click/drag. Fortunately, PanTool is not a "stateful" interaction, and there is a way to tell SimpleZoom to play nicely with others:

```
1. # tutorial5.py
2.
3. from tutorial2 import myplot, PlotFrame, main
4. from enthought.chaco2.tools import PanTool, SimpleZoom
5.
6. myplot.tools.append(PanTool(myplot))
7.
8. zoom = SimpleZoom(myplot, tool_mode="box", always_on=False)
9.
10. myplot.overlays.append(zoom)
```

Writing our first interactor

Diagnostic tool to dump out the events we are getting:

```
1. # tutorial6.py
2.
3. # AbstractController is the base class for non-rendering tools
4. from enthought.chaco2 import AbstractController
5.
6. class EventPrinter(AbstractController):
7.
8.     def dispatch(self, event, suffix):
9.         # event' is a MouseEvent or KeyEvent object from the enable2
10.        # package, and suffix is a text string
11.        print suffix, "event received at (%d,%d)" % (event.x, event.y)
12.
13.
14. from tutorial2 import myplot, PlotFrame, main
15.
16. myplot.tools.append(EventPrinter(myplot))
```

Looking at data

We don't really want to know the mouse position; we want to know about coordinates in data space:

```
1. # tutorial7.py
2.
3. from enthought.chaco2 import AbstractController
4.
5. class DataPrinter(AbstractController):
6.     def dispatch(self, event, suffix):
7.         # We are assuming that self.component is an X-Y plot
8.         x = self.component.x_mapper.map_data(event.x)
9.         y = self.component.y_mapper.map_data(event.y)
10.        print suffix, "event received at (%d,%d)" % (x, y)
11.
12.
13. from tutorial2 import myplot, PlotFrame, main
14.
15. myplot.tools.append(DataPrinter(myplot))
```

Digging deeper

```

1. # from tutorial11.py
2.
3. myplot = chaco2.create_line_plot((x,y))

```

What does the `create_line_plot()` actually do?

- creates datasources
- creates range objects and mappers
- creates the plot object

Datasources

```

1. # plot_factory.py:create_line_plot()
2.
3. index_data, value_data = transpose(data)
4. index = ArrayDataSource(index_data)
5. value = ArrayDataSource(value_data, sort_order="none")

```

`ArrayDataSource` is just a subclass of `AbstractDataSource` which works with Numeric/numpy arrays.

```

1. class AbstractDataSource(HasTraits):
2.     value_dimension = DimensionTrait
3.     index_dimension = DimensionTrait
4.     metadata = Dict
5.
6.     get_data() -> array
7.     get_data_mask() -> array, mask
8.     is_masked() -> bool
9.     get_size() -> int
10.    get_bounds() -> array

```

Ranges, Mappers, and the Plot

```

1. index_range = DataRange()
2. index_range.add(index)
3. index_mapper = LinearMapper(range=index_range)
4.
5. value_range = DataRange()
6. value_range.add(value)
7. value_mapper = value_mapper_class(range=value_range)
8.
9. plot = LinePlot(index=index, value=value,
10.                index_mapper = index_mapper,
11.                value_mapper = value_mapper,
12.                orientation = orientation,
13.                color = color,
14.                line_width = width,
15.                line_style = dash,
16.                border_visible = True)

```

Two plots

We can use a container to put two plots on the screen.

```

1. def _create_plot(self):
2.     x = arange(-5.0, 15.0, 20.0/100)
3.
4.     y = jn(0, x)
5.     left_plot = create_line_plot((x,y), bgcolor="white", add_grid=True, add_axis=True)
6.     left_plot.tools.append(PanTool(left_plot))
7.     self.left_plot = left_plot
8.
9.     y = jn(1, x)
10.    right_plot = create_line_plot((x,y), bgcolor="white", add_grid=True, add_axis=True)
11.    right_plot.tools.append(PanTool(right_plot))
12.    right_plot.y_axis.orientation = "right"
13.    self.right_plot = right_plot
14.
15.    container = HPlotContainer(spacing=20, padding=50, bgcolor="lightgray")
16.    container.add(left_plot)
17.    container.add(right_plot)
18.    return container

```

Connecting the two plots

We're going to link the X dimension of the two plots. Really, all we have to do to achieve this is to set the horizontal range on the two plots to be the same.

```

1. # tutorial9.py
2.
3. class PlotFrame2(PlotFrame):
4.
5.     def _create_plot(self):
6.
7.         container = super(PlotFrame2, self)._create_plot()
8.
9.         self.right_plot.index_mapper.range = self.left_plot.index_mapper.range
10.
11.         return container
12.

```

Connecting the two plots (cont.)

We can connect the Y dimension, too. And let's throw in a zoom tool.

```

1. # tutorial9b.py
2.
3. self.right_plot.value_mapper.range = self.left_plot.value_mapper.range
4.
5. self.left_plot.overlays.append(
6.     SimpleZoom(self.left_plot, tool_mode="box", always_on=False))
7.
8. self.right_plot.overlays.append(
9.     SimpleZoom(self.right_plot, tool_mode="box", always_on=False))
10.

```

Linked views, but not connected data

Adding a line inspector will show the problem.

```
1. # tutorial10.py
2.
3. from tutorial9b import PlotFrame2
4.
5. class PlotFrame3(PlotFrame2):
6.     def _create_plot(self):
7.         container = super(PlotFrame2, self)._create_plot()
8.
9.         self.left_plot.overlays.append(
10.             LineInspector(component=self.left_plot,
11.                            write_metadata=True,
12.                            is_listener=True))
13.
14.         self.right_plot.overlays.append(
15.             LineInspector(component=self.right_plot,
16.                            write_metadata=True,
17.                            is_listener=True))
18.         return container
19.
```

Connecting the data

```
1. # tutorial10b.py
2.
3. class PlotFrame3(PlotFrame2):
4.     def _create_plot(self):
5.         container = super(PlotFrame2, self)._create_plot()
6.
7.         self.left_plot.overlays.append(
8.             LineInspector(component=self.left_plot,
9.                            write_metadata=True,
10.                            is_listener=True))
11.
12.         self.right_plot.overlays.append(
13.             LineInspector(component=self.right_plot,
14.                            write_metadata=True,
15.                            is_listener=True))
16.
17.         self.right_plot.index = self.left_plot.index
18.
19.         return container
20.
```

Why Index-Value instead of X-Y?

Because you are then immune to a plot's physical layout.

```
1. # tutorial11.py
2.
3. class PlotFrame4(PlotFrame3):
4.     def _create_plot(self):
5.         container = super(PlotFrame4, self)._create_plot()
6.
7.         plot = self.right_plot
8.         plot.orientation = "v"
9.         plot.hgrid.mapper = plot.index_mapper
10.        plot.vgrid.mapper = plot.value_mapper
11.        plot.y_axis.mapper = plot.index_mapper
12.        plot.x_axis.mapper = plot.value_mapper
13.
14.        self.left_plot.overlays.append(
15.            LineInspector(component=self.left_plot,
16.                          write_metadata=True,
17.                          is_listener=True, color="blue",
18.                          axis="value"))
19.
20.        self.right_plot.overlays.append(
21.            LineInspector(component=self.right_plot,
22.                          write_metadata=True,
23.                          is_listener=True, color="blue",
24.                          axis="value"))
25.        return container
```

Examples

Visual Components

- Renderers
- Containers
- Tools and tool overlays
- Legends, annotations, etc.

Visual Components (cont.)

- have bounds, position, padding, border, bgcolor
- can be placed inside containers
- can be horizontally and vertically resized (or not)
- implement draw (and request draw on their underlays and overlays)
- implement dispatch (and dispatch to underlays and overlays)

More about Containers

- Layout
- Dispatch
- Draw order & backbuffering

Walkthrough of interesting examples

How to get it

<http://code.enthought.com/chaco>

Mailing lists: scipy-chaco@scipy.org, enthought-dev@enthought.com

Get Enthon! <http://code.enthought.com/enthon>