Chaco is a Python package for building interactive and custom 2-D plots and visualizations. Chaco facilitates writing plotting applications at all levels of complexity, from simple scripts with hard-coded data to large plotting programs with complex data interrelationships and a multitude of interactive tools. While Chaco generates attractive static plots for publication and presentation, it also works well for interactive data visualization and exploration. Chaco is part of the Enthought Tool Suite.

Chaco includes renderers for many popular plot types, built-in implementations of common interactions with those plots, and a framework for extending and customizing plots and interactions. Chaco can also render graphics in a non-interactive fashion to images, in either raster or vector formats, and it has a subpackage for doing command-line plotting or simple scripting.

For a quick sample of Chaco’s features, see the gallery, the annotated examples page, the tutorial and examples and the resources page.
1.1 Quickstart

This section is meant to help users on well-supported platforms and common Python environments get started using Chaco as quickly as possible. Chaco users can subscribe to the enthought-dev mailing list to post questions, consult archives, and share tips.

1.1.1 Installation

There are several ways to get Chaco. The easiest way is through the Enthought Python Distribution (EPD), which is available for several platforms and also provides many other useful packages. Chaco may also be available through a package manager on your platform, such as apt on Ubuntu or MacPorts on OS X. You can also build Chaco yourself, but because of the number of packages required, we highly recommend you install EPD.

Dependencies

- Python 2.5 or later
- Traits, an event notification framework
- Kiva, part of the enable project, for rendering 2-D graphics to a variety of backends across platforms
- Enable, a framework for writing interactive visual components, and for abstracting away GUI-toolkit-specific details of mouse and keyboard handling
Chaco Documentation, Release 4.3.0

- NumPy, for dealing efficiently with large datasets
- Either wxPython or PyQt to display interactive plots. As an alternative to PyQt, Chaco is being tested more and more with the PySide toolkit (LGPL license).

Installing Chaco with EPD

Chaco, the rest of the Enthought Tool Suite, and a lot more are bundled with EPD. Getting EPD allows you to install Chaco and all its dependencies at once; however, these packages will be linked to a new instance of Python. The EPD Free distribution is free for all users and contains all that you need to use Chaco.

To get EPD, go to the EPD download page and get the appropriate version for your platform. After running the installer, you will have a working version of Chaco and several examples.

Building Chaco

Building Chaco on your machine requires you to build Chaco and each of its dependencies, but it has the advantage of installing Chaco on top of the Python instance you already have installed. The build process may be challenging and will require you to have SWIG, Cython and several development libraries installed.

To do this, you can either

1. Install Chaco and its Dependencies from PyPI using easy_install (part of setuptools) or using pip. For example
easy_install chaco
   or
   pip install chaco
2. Or, download the source from the Chaco GitHub repository or alternatively as a part of ETS.

1.1.2 Built-in Examples

Chaco ships with several examples for testing your installation and to show you what Chaco can do. Almost all of the examples are stand-alone files that you can run individually, from any location. Depending on how you installed Chaco, you may or may not have the examples already.

Location

1. If you installed Chaco as part of EPD, the location of the examples depends on your platform:
   - On Windows, they are in the Examples\ subdirectory of your installation location. This is typically C:\Python27\Examples\Chaco-<version>. On MS Windows these examples can be browsed from the start menu, by clicking Start → Applications → Enthought → Examples.
   - On Linux, they are in the Examples/Chaco-<version> subdirectory of your installation location.
   - On Mac OS X, they are in the /Applications/Enthought/Examples/chaco-<version> directory.

2. If you downloaded and installed Chaco from source (from GitHub or via the PyPI tar.gz file), the examples are located in the examples/ subdirectory inside the root of the Chaco source tree, next to docs/ and the enthought/ directories.

3. If you don’t know how Chaco was installed, you can download the latest versions of examples individually from github:
For ETS 3.0 or Chaco 3.0, you can check out the examples with Subversion:

    svn co https://svn.enthought.com/svn/enthought/Chaco/tags/3.0.0/examples

For ETS 2.8 or Chaco 2.0.x:

    svn co https://svn.enthought.com/svn/enthought/Chaco/tags/enthought.chaco2_2.0.5/examples

Chaco examples can be found in the examples/demo/ and examples/tutorials/ directories. Some are classified by themes and located in separate directories. Almost all of the Chaco examples are standalone files that can be run individually. We will first show how to execute them from the command line, and then we will show how to run Chaco in an interactive way from IPython. This “shell” mode will be more familiar to Matplotlib or Matlab users.

---

Note: Some of these examples can be visualized in our Chaco gallery.

---

First plots from the command line

From the examples/demo directory, run the simple_line example:

    python simple_line.py

This opens a plot of several Bessel functions with a legend.

![Bessel line plot](image)

You can interact with the plot in several ways: .. Ctrl-Left and Ctrl-Right don’t work in OS X?
• To pan the plot, hold down the left mouse button inside the plot area (but not on the legend) and drag the mouse.

• To zoom the plot:
  – Mouse wheel: scroll up to zoom in, and scroll down to zoom out (or the reverse you’re on a version of OS X with ‘natural scrolling’).
  – Zoom box: Press z, and then draw a box region to zoom in on. (There is no box-based zoom out.) Press Ctrl-Left and Ctrl-Right to go back and forward in your zoom box history.
  – Drag: hold down the right mouse button and drag the mouse up or down. Up zooms in, and down zooms out.
  – For any of the above, press Escape to reset the zoom to the original view.

• To move the legend, hold down the right mouse button inside the legend and drag it around. Note that you can move the legend outside of the plot area.

• To exit the plot, click the “close window” button on the window frame or (on Mac) choose the Quit option on the Python menu. Alternatively, can you press Ctrl-C in the terminal.

You can run most of the examples in the the examples/demo/basic/ directory and the examples/demo/shell/ directory. The examples/demo/advanced/ directory has some examples that require additional data or packages. In particular,

• spectrum.py requires that you have PyAudio installed and a working microphone.

• data_cube.py needs to download about 7.3mb of data from the Internet the first time it is executed, so you must have a working Internet connection. Once the data is downloaded, you can save it so you can run the example offline in the future.

For detailed information about each built-in example, see the Annotated Examples section.

First plots from IPython

While all of the Chaco examples can be launched from the command line using the standard Python interpreter, if you have IPython installed, you can poke around them in a more interactive fashion.

Chaco provides a subpackage, currently named the “Chaco Shell”, for doing command-line plotting like Matlab or Matplotlib. The examples in the examples/demo/shell/ directory use this subpackage, and they are particularly amenable to exploration with IPython.

The first example we’ll look at is the lines.py example. First, we’ll run it using the standard Python interpreter:

    python lines.py

This shows two overlapping line plots.
You can interact with this plot just as in the previous section.

Now exit the plot, and start IPython with the `--gui=wx` option:

```bash
ipython --gui=wx
```

This tells IPython to start a wxPython mainloop in a background thread. Now run the previous example again:

In [1]: run lines.py

This displays the plot window, but gives you another IPython prompt. You can now use various commands from the `chaco.shell` package to interact with the plot.

Import the shell commands:

In [2]: from chaco.shell import *

Set the X-axis title:

In [3]: xtitle("X data")

Toggle the legend:

1 Starting from IPython 0.12, it is possible to use the Qt backend with `--gui=qt`. Make sure that the environment variable `QT_API` is set correctly, as described here
In [4]: legend()

After running these commands, your plot looks like this:

The `chaco_commands()` function display a list of commands with brief descriptions.

You can explore the Chaco object hierarchy, as well. The `chaco.shell` commands are just convenience functions that wrap a rich object hierarchy that comprise the actual plot. See the `tutorial_ipython` section for information on all you can do with Chaco from within IPython.

### 1.1.3 Chaco plot embedded in a Traits application

Let’s create, from scratch, the simplest possible Chaco plot (embedded inside a Traits application).

First, some imports to bring in necessary components:

```python
from chaco.api import ArrayPlotData, Plot
from enable.component_editor import ComponentEditor
from traits.api import HasTraits, Instance
from traitsui.api import View, Item
```

```python
```

```python
```
The imports from `chaco` and `enable` support the creation of the plot. The imports from `traits` bring in components to embed the plot inside a Traits application. (Refer to the Traits documentation for more details about building an interactive application using Traits.) Now let’s create a Traits class with a view that contains only one element: a Chaco plot inside a slightly customized window:

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

A few options have been set to control the window containing the plot. Now, when the plot is created, we would like to pass in our data. Let’s assume the data is a set of points with coordinates contained in two NumPy arrays `x` and `y`. So, adding an `__init__` method to create the Plot object looks as follows:

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

    def __init__(self, x, y, *args, **kw):
        super(MyPlot, self).__init__(*args, **kw)
        plotdata = ArrayPlotData(x=x, y=y)
        plot = Plot(plotdata)
        plot.plot(('x','y'), type = "line", color = "blue")
        self.plot = plot
```

Since it inherits from HasTraits, the new class can use all the power of Traits, and the call to `super()` in its `__init__` method makes sure this object possesses the attributes and methods of its parent class. Now let’s use our Traits object. Below, we generate some data, pass it to an instance of MyPlot and call `configure_traits` to create the UI:

```python
import numpy as np
x = np.linspace(-14,14,100)
y = np.sin(x)*x**3
lineplot = MyPlot(x,y)
lineplot.configure_traits()
```

The result should look like
This might look like a lot of code to visualize a function, but this is a relatively simple basis on top of which we can build full-featured applications with custom UIs and custom tools. For example, the Traits object allows you to create controls for your plot at a very high level, add these controls to the UI with very little work, and add listeners to update the plot when the data changes. Chaco also allows you to create tools to interact with the plot and overlays that make these tools intuitive and visually appealing.

1.1.4 License

As part of the Enthought Tool Suite, Chaco is free and open source under the BSD license.
1.2 Tutorials, webinars, and examples

1.2.1 Tutorials

• Tutorial: Interactive plotting with Chaco
  This is the main Chaco tutorial and introduces the basic concepts of how to use Chaco and Traits UI to do basic plots, customize layout, and add interactivity.

• Tutorial: Using Chaco from IPython
  This tutorial explains how to use Chaco from IPython using the Chaco shell command-line plotting interface to build plots, in a Matlab or gnuplot-like style.

1.2.2 Webinars

• Step-by-step Chaco - 2D plotting with Python
  Webinar recorded on, July 17, 2009, available as Windows Media Player (.wmv) video, Matroska (.mkv) video, and as a slideshare slide show.

• EPDLab & Chaco
  Webinar recorded on, June 19, 2009, available as Windows Media Player (.wmv) video, Matroska (.mkv) video, and as a slideshare slide show.

1.2.3 Examples

• The annotated examples is a useful visual resource presenting a set of Chaco plots together with their source code.

• Modeling Van del Waal’s Equations is a complete example of creating a data model and then using Traits and Chaco to rapidly create interactive plot GUIs.

• Creating an interactive Hyetograph is an example of a hyetograph (a plot of rainfall intensity in relation to time) application. This example introduces the on_trait_listener decorator and uses Chaco, simple Traits views, and live GUI interaction.

1.3 User guide

1.3.1 Introduction

What is Chaco?

Chaco is a 2D plotting library that is part of and integrates with the Enthought Tools Suite.

The strong points of Chaco are

1. it can be embedded in any wx, Qt, or TraitsUI application
2. it is designed for building interactive plotting applications, rather than static 2D plots
3. Chaco classes can be easily extended to create new plot types, interactive tools, and plot containers
At the lowest level, Chaco is a hierarchy of classes that defines 2D plotting elements: plots, containers, interactive tools, color bars, etc. In principle, applications can create instances of these elements and lay them out in a container to define components that can be embedded in one of several of graphical back ends. Working at this level allows the maximum flexibility, but requires understanding Chaco’s basic elements.

Chaco defines two abstraction layers that allow a more high-level (albeit less flexible) plotting experience. First, Chaco contains a `Plot` class that defines several methods that create a complete plot given one or more data sets. In other words, `Plot` knows how to package data for the most common kinds of plots. Second, Chaco has a `shell` module that defines high-level plotting functions. This module allows using Chaco as an interactive plotting tool that will be familiar to users of matplotlib.

Basic elements

To venture deeper in Chaco’s architecture it is useful to understand a few basic ideas on which Chaco is based:

- **Plots are compositions of visual components**
  Each plot is composed by a number of graphical widgets: the plot graphics, axes, labels, legend, colorbar, etc. Everything you see in a plot is an individual component with position, shape, and appearance attributes, and with an opportunity to respond to events.

- **Data and screen space are separated**
  Although everything in a plot eventually ends up rendering into a common visual area, there are aspects of the plot which are intrinsically screen-space, and some which are fundamentally data-space. For example, data about the height of college students lives in data space (meters), but needs to be rendered in screen space (pixels). Chaco uses the concept of mapper to translate one into the other. Preserving the distinction between these two domains allows us to think about visualizations in a structured way.

- **Layers**
  Plot components are split into several layers, which are usually plotted in sequence. For example, axes and labels are usually plotted on the “underlay” layer, plot data on the “plot” layer, and legends and other plot annotations on the “overlay” layer. In this way one can define interactive tools that add graphical elements to a plot without having to modify the drawing logic.

These pages describe in detail the basic building blocks of Chaco plots, and the classes that implement them:

Data sources

A data source is a wrapper object for the actual data that the plot will be handling. For the most part, a data source looks like an array of values, with an optional mask and metadata.

The data source interface provides methods for retrieving data, estimating a size of the dataset, indications about the dimensionality of the data, a place for metadata (such as selections and annotations), and events that fire when the data gets changed.

There are two primary reasons for a data source class:

- It provides a way for different plotting objects to reference the same data.
- It defines the interface to expose data from existing applications to Chaco.

In most cases, the standard `ArrayDataSource` will suffice.

**Interface**  The basic interface for data sources is defined in `AbstractDataSource`. Here is a summary of the most important attributes and methods (see the docstrings of this class for more details):

value_dimension
The dimensionality of the data value at each point. It is defined as a `DimensionTrait`, i.e., one of “scalar”, “point”, “image”, or “cube”. For example, a `GridDataSource` represents data in a 2D array and thus its `value_dimension` is “scalar”.

`index_dimension`

The dimensionality of the data value at each point. It is defined as a `DimensionTrait`, i.e., one of “scalar”, “point”, “image”, or “cube”. For example, a `GridDataSource` represents data in a 2D array and thus its `index_dimension` is “image”.

`metadata`

A dictionary that maps strings to arbitrary data. Usually, the mapped data is a set of indices, as in the case of selections and annotations. By default, `metadata` contains the keys “selections” (representing indices that are currently selected by some tool) and “annotations”, both initialized to an empty list.

`persist_data`

If True (default), the data that this data source refers to is serialized when the data source is.

`get_data()`

Returns a data array containing the data referred to by the data source. Treat the returned array as read-only.

`is_masked()`

Returns True if this data source’s data uses a mask. In this case, to retrieve the data, call `get_data_mask()` instead of `get_data()`.

`get_data_mask()`

Returns the full, raw, source data array and a corresponding binary mask array. Treat both arrays as read-only.

`get_size()`

Returns the size of the data.

`get_bounds()`

Returns a tuple (min, max) of the bounding values for the data source. In the case of 2-D data, min and max are 2-D points that represent the bounding corners of a rectangle enclosing the data set. If data is the empty set, then the min and max vals are 0.0.

**Events**

`AbstractDataSource` defines three events that can be used in Traits applications to react to changes in the data source:

`data_changed`

Fired when the data values change.

---

**Note:** The majority of concrete data sources do not fire this event when the data values change. Rather, the event is usually fired when new data or a new mask is assigned through setter methods (see notes below).

`bounds_changed`

Fired when the data bounds change.

`metadata_changed`
Fired when the content of metadata changes (both the metadata dictionary object or any of its items).

List of Chaco data sources  This is a list of all concrete implementations of data sources in Chaco:

ArrayDataSource

A data source representing a single, continuous array of numerical data. This is the most common data source for Chaco plots.

This subclass adds the following attributes and methods to the basic interface:

sort_order

The sort order of the data, one of ‘ascending’, ‘descending’, or ‘none’. If the underlying data is sorted, and this attribute is set appropriately, Chaco is able to use shortcuts and optimizations in many places.

reverse_map(pt)

Returns the index of pt in the data source (optimized if sort_order is set).

Note:  This class does not listen to the array for changes in the data values. The data_changed event is fired only when the data or the mask are set with the methods set_data(), set_mask(), or remove_mask().

ImageData

Represents a 2D grid of image data.

The underlying data array is 3D, where the third dimension is either 1 (one scalar value at each point of the grid), 3 (one RGB vector at each point), or 4 (one RGBA vector at each point). The depth of the array is defined in the attribute value_depth.

Access to the image data is controlled by three properties: The boolean attribute transposed defines whether the data array stored by this class is to be interpreted as transposed; raw_value returns the underlying data array as-is, ignoring transposed; value returns the data array or its transposed depending on the value of transposed.

The correct usage pattern of these attributes is to give to the class contiguous image data, and assign transposed if the two axis should be swapped. Functions that would benefit from working on contiguous data can then use raw_value directly. (See the class docstrings for more details, and some caveats.)

Noteworthy methods of this class are:

fromfile(filename)

Factory method that creates an ImageData instance from an image file. filename can be either a file path or a file object.

get_width(), get_height()

Return the width or the height of the image (takes the value of transposed into account).

get_array_bounds()

Return ((0, width), (0, height)).

Note:  This class does not implement the methods related to masking, and it does not fire bounds_changed events.
Note: This class does not listen to the array for changes in the data values. The `data_changed` event is fired only when the data are set with the method `set_data()`.

**GridDataSource**

Data source representing the coordinates of a 2D grid. It is used, for example, as a source for the index data in an `ImagePlot`.

It defines these attributes:

`sort_order`  
Similar to the `sort_order` attribute for the `ArrayDataSource` class above, but this is a tuple with two elements, one per dimension.

**MultiArrayDataSource**

A data source representing a single, continuous array of multidimensional numerical data.

It is useful, for example, to define 2D vector data at each point of a scatter plot (as in `QuiverPlot`), or to represent multiple values for each index (as in `MultiLinePlot`).

As `ArrayDataSource`, this data source defines a `sort_order` attribute for its index dimension.

**Warning:** In `MultiArrayDataSource`, the `index_dimension` and `value_dimension` attributes are integers that define which dimension of the data array correspond to indices and which to values (default is 0 and 1, respectively). This is different from the same attributes in the interface, which are strings describing the dimensionality of index and value.

**PointDataSource**

A data source representing a set of (X,Y) points.

This is a subclass of `ArrayDataSource`, and inherits its methods and attributes. The attribute `sort_index` defines whether the data is sorted along the X’s or the Y’s (as specified in `sort_order`).

**FunctionDataSource**

A subclass of `ArrayDataSource` that sets the values of the underlying data array based on a function (defined in the callable attribute `func`) evaluated on a 1D data range (defined in `data_range`).
FunctionImageData

A subclass of ImageData that sets the values of the underlying data array based on a 2D function (defined in the callable attribute func) evaluated on a 2D data range (defined in data_range).

Data ranges

A data range expresses bounds on data space of some dimensionality. For example, the simplest data range is just a set of two scalars representing (low, high) bounds in 1-D. Data ranges are commonly used in plots to determine the range of plot axes.

Data ranges are typically associated to a data source, with their bounds set to auto, which means that they automatically scale to fit the data source bounds. Each data source can be associated with multiple ranges, and each data range can be associated with multiple data sources.

Interface  The basic interface for data sources is defined in AbstractDataRange and BaseDataRange. This is a summary of the most important attributes and methods (see the docstrings of this class for more details):

Attributes

sources

A list of data sources associated to the data range. Concrete implementations of data range listen to the event data_changed and refresh their bounds as appropriate (e.g., when the bounds are set to auto).

low

The actual value of the lower bounds of the range. The correct way to set it is to use the low_setting attribute.

high

The actual value of the upper bounds of the range. The correct way to set it is to use the high_setting attribute.

low_setting

Setting for the lower bound of the range. This can either be a valid lower bound value, or auto (default), in which case the lower bound is set automatically from the associated data sources.

high_setting

Setting for the upper bound of the range. This can either be a valid upper bound value, or auto (default), in which case the upper bound is set automatically from the associated data sources.

Methods

add(*datasources)

Convenience method to associate one or more data sources to the range. The method avoids adding the same data source twice.

remove(*datasources)

Convenience method to remove one or more data sources from the range. If one of the data sources is not associated with the range, it is ignored.

clip_data(data)

Given an array of data values of the same dimensionality as the range, return a list of data values that are inside the range.
Chaco Documentation, Release 4.3.0

mask_data(data)

Given an array of data values of the same dimensionality as the range, this method returns a mask array of the same length as data, filled with 1s and 0s corresponding to whether the data value at that index is inside or outside the range.

bound_data(data)

Given an array of monotonic data values of the same dimensionality as the range, returns a tuple of indices (start, end) corresponding to the first and last elements that fall within the range.

Events

The basic data range interface defines a single event, updated, which is fired when the bound values change. The value of the event is a tuple (low_bound, high_bound).

List of Chaco data ranges

There are two data range implementations in Chaco, one for 1D and one for 2D ranges:

DataRange1D

DataRange1D represents a 1D data range. This subclass adds several more ways to control the bound of the range given the associated data sources.

First of all, a new parameter, tight_bounds, controls whether the bounds should fit exactly the range of the associated data sources (the default is True). If it is False, the range adds some padding on either side of the data, controlled by margin, which is expressed as a percentage of the full data width.

Second, DataRange1D defines a new setting, track for low_setting and high_setting. When one of the bounds is set to track, it follows the other bound by the amount set in tracking_amount.

Third, bounds can be computed using a user-supplied function specified in bounds_func. The function takes the arguments (data_low, data_high, margin, tight_bounds), where data_low and data_high are the bounds computed after taking into account the auto or track settings, and margin and tight_bounds are defined as above.

The logic of computing the bounds is implemented in the function calc_bounds() in chaco.data_range_1d.

DataRange2D

DataRange2D represents a 2D data range. Under the hood, it is implemented using two DataRange1D objects, one for each dimension, which are stored in the x_range and y_range attributes. These can be accessed directly if one wants to use the full flexibility of the DataRange1D class.

The data range bounds, low and high, return 2-elements tuples containing the bounds for for the two dimensions.

Mappers

Mappers perform the job of mapping a data space region to screen space, and vice versa. They are used by plots to transform their data sources to pixel coordinates on the screen. While most of the time this is a relatively simple rescaling operation, mapper can also be used for non-linear transformations, most notably to map a data source to logarithmic coordinates on screen.

Interface

The general interface for mappers is defined in AbstractMapper and defines only a few methods:

map_screen(data_value), map_data(screen_value)

Maps a vector of data coordinates to screen coordinates, and vice versa.
map_data_array(screen_value_array)

Maps an array of points in data coordinates to screen coordinates. By default, this method just loops over the points, calling map_data() on each one. For vectorizable mapping functions, this implementation is overridden with a faster one.

Mappers for 1D data have a slightly larger interface, defined in Base1DMapper. These mappers rely on a DataRange1D object to find the bounds on the data domain.

range

A DataRange1D instance to define the data-space bounds of the mapper. The mapper listens to the updated event of the range and re-fires it as its own updated event (see below).

low_pos, high_pos, screen_bounds,

The screen space position of the lower/upper bound of the data space. screen_bounds is a convenience property to set/get the screen bounds with a single attribute.

stretch_data

When the screen bounds change (in response, for instance, to the window resizing) one could either fit more data space on the screen, or stretch the data space to the new bounds. If stretch_data is True (default), the data is stretched; if it is False, the mapper preserves the screen-to-data ratio.

Events

The AbstractMapper interface defines a single generic event, updated, which is fired when the bound values change.

For subclasses of Base1DMapper, the updated event is also fired in response to an updated event fired by the underlying data range. The value of the new event is the tuple (low_bound, high_bound) contained in the triggering event.

List of Chaco data mappers

LinearMapper (subclass of Base1DMapper)

This mapper transforms a 1D data space range linearly to a fixed 1D range in screen space.

LogMapper (subclass of Base1DMapper)

Maps a 1D data space range to a 1D range in screen space through a logarithmic transform. Data values smaller than or equal to 0.0 are substituted by fill_value (default is 1.0) before the logarithmic transformation.

GridMapper

Same as LinearMapper for 2D ranges. This class replaces the Base1DMapper attributes with analogous ones:

range

A DataRange2D instance to define the data-space bounds of the mapper.

x_low_pos, y_low_pos, x_high_pos, y_high_pos

Screen space positions for the lower and upper bounds of the x and y axes.

screen_bounds

Convenience property to set/get the screen bounds with a single attribute. The value of this attribute is a 4-element tuple (x_low_pos, x_high_pos, y_low_pos, y_high_pos).
GridMapper uses two Base1DMapper instances to define mappers for the two axes (accessible from the two private attributes _xmapper and _ymapper). It is possible to set them to be linear or logarithmic mappers. This is best made using the class constructor, which has this signature:

```python
GridMapper(x_type="linear", y_type="linear", range=None, **kwargs)
```

x_type and y_type can be either ‘linear’ or ‘log’, which will create a corresponding LinearMapper or LogMapper classes.

PolarMapper

This class should map data polar coordinates to screen cartesian coordinates, to use for example with a PolarLineRenderer, but at the moment it is a copy of LinearMapper.

**Warning:** The implementation of this mapper is under construction.

**Plot renderers**

Plot renderers are the classes that actually draw the different kinds of plots, or plot-like elements as for instance color bars.

This section describes the concepts that are common to all kind of plots. A separate page contains an exhaustive list of all plot types defined in Chaco.

**Common interface**

The base interface is defined in the abstract class AbstractPlotRenderer, and provides attributes and methods to set size, position, and aspect of the plotting area.

Two more specialized interface are used by most concrete implementations, namely BaseXYPlot, which is the interface for X-vs-Y plots, and Base2DPlot, which is the interface for 2D plots (e.g., image plots or contour plots).

The base interface inherits from a deep hierarchy of classes generating from the enable package, starting with enable.coordinate_box.CoordinateBox (representing a box in screen space) and enable.interactor.Interactor (which allows plot components to react to mouse and keyboard events), and down through enable.component.Component and chaco.plot_component.PlotComponent. The class where most of the functionality is defined is enable.component.Component.

Here we give a summary of all the important properties exposed in AbstractPlotRenderer, without worrying too much about their origin in the hierarchy. Also, to avoid unnecessary cluttering of the page, attributes and methods that are of secondary importance are not listed. Please refer to the API documentation for more details.

**Box properties**

All plot renderers are enable graphical components, and thus correspond to a rectangular area in screen space. The renderer keeps track of two areas: an inner box that only contains the plot, and an outer box that includes the padding and border area. The properties of the boxes are controlled by these attributes:

- **position**
  Position of the internal box relative to its container, given as a list [x, y]. If there is no container, this is set to [0, 0]. “Absolute” coordinates of point (i.e., relative to top-level parent Window object) can be obtained using get_absolute_coords(*coords).

- **x, y, x2, y2**
  Coordinates of the lower-left (x,y) and upper-right (x2,y2) pixel of the internal box, relative to its container.

- **bounds, width, height**
Bounds of the internal box, in pixels. `bounds` is a list `[width, height]`.

`outer_position, outer_x, outer_y, outer_x2, outer_y2, outer_bounds, outer_width, outer_height, set_outer_position(index, value), set_outer_bounds(index, value)`

Attributes for the outer box equivalent to those defined above for the inner box. Modifying the outer position attributes is the right way to move the plot without changing its padding or bounds. Similarly, modifying the outer bounds attributes leaves the lower-left position and the padding unchanged.

`resizable, fixed_preferred_size`

String that defines in which dimensions the component is resizable. One of ‘’ (not resizable), ‘v’ (resizable vertically), ‘h’ (resizable horizontally), ‘hv’ (resizable in both directions, default). If the component is resizable, `fixed_preferred_size` can be used to specify the amount of space that the component would like to get in each dimension, as a tuple (width, height). In this case, width and height have to be understood as relative sized: if one component in a container specifies, say, a fixed preferred width of 50 and another one specifies a fixed preferred width of 100, then the latter component will always be twice as wide as the former.

`aspect_ratio, auto_center`

Ratio of the component’s width to its height. This is used to maintain a fixed ratio between bounds when they are changed independently, for example when resizing the window. `auto_center` specifies if the component should center itself in any space that is left empty (default is True).

`padding_left, padding_right, padding_top, padding_bottom, padding, hpadding, vpadding`

Padding space (in pixels). `padding` is a convenience property that returns a tuple of (left, right, top, bottom) padding. It can also be set to a single integer, in which case all four padding attributes are set to the same value.

`hpadding` and `vpadding` are read-only properties that return the total amount of horizontal and vertical padding (including the border width if the border is visible).

`get_absolute_coords(*coords)`

Transform coordinates relative to this component’s origin to “absolute” coordinates, relative to top-level container.

**Aspect properties**  These attributes control the aspect (e.g. color) of padding, background, and borders:

`bgcolor`

The background color of this component (default is white). This can be set to “transparent” or “none” if the component should be see-through. The color can be specified as a string or as an RGB or RGBA tuple.

`fill_padding`

If True (default), fill the padding area with the background color.

`border_visible`

Determines if the border is visible (default is False).

`border_width`

Thickness of the border around the component in pixels (default is 1).
border_dash

Style of the lines tracing the border. One of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.

border_color

Color of the border. The color can be specified as a string or as an RGB or RGBA tuple.

Layers

Each plot is rendered in a sequence of layers so that different components can plot at different times. For example, a line plot is drawn before its legend, but after the axes and background grid.

The default drawing order is defined in draw_order as a list of the names of the layers. The definition of the layers is as follows:

1. ‘background’: Background image, shading, and borders
2. ‘image’: A special layer for plots that render as images. This is in a separate layer since these plots must all render before non-image plots
3. ‘underlay’: Axes and grids
4. ‘plot’: The main plot area itself
5. ‘annotation’: Lines and text that are conceptually part of the “plot” but need to be rendered on top of everything else in the plot.
6. ‘selection’: Selected content are rendered above normal plot elements to make them stand out. This can be disabled by setting use_selection to False (default).
7. ‘border’: Plot borders
8. ‘annotation’: Lines and text that are conceptually part of the “plot” but need to be rendered on top of everything else in the plot
9. ‘overlay’: Legends, selection regions, and other tool-drawn visual elements

Concrete plot renderers set their default draw layer in draw_layer (default is ‘plot’). Note that if this component is placed in a container, in most cases the container’s draw order is used, since the container calls each of its contained components for each rendering pass.

One can add new elements to a plot by appending them to the underlays or overlays lists. Components in these lists are drawn underneath/above the plots as part of the ‘underlay’/’overlay’ layers. They also receive mouse and keyboard events.

Interaction

Plot renderers also inherit from enable.interactor.Interactor, and as such are able to react to keyboard and mouse events. However, interactions are usually defined as tools and overlays. Therefore, this part of the interface is described at those pages.

TODO: add reference to interaction interface

Context

Since plot renderers take care of displaying graphics, they keep references to the larger graphical context:

container

Reference to a container object (None if no container is defined). The renderer defines its position relative to this.

window

Reference to the top-level enable Window.
viewports
List of viewport that are viewing this component

Others

use_backbuffer
If True, the plot renders itself to an offscreen buffer that is cached for later use. If False (default), then the component will never render itself back-buffered, even if asked to do so.

invalidate_and_redraw()
Convenience method to invalidate our contents and request redraw. This method is sometimes useful when modifying a Chaco plot in an ipython shell.

X-Y Plots interface
The class chaco.base_xy_plot.BaseXYPlot defines a more concrete interface for X-vs-Y plots. First of all, it handles data sources and data mappers to convert real data into screen coordinates. Second, it defines shortcuts for plot axes, labels and background grids.

Data-related traits
X-Y plots need two sources of data for the X and Y coordinates, and two mappers to map the data coordinates to screen space. The data sources are stored in the attributes index and value, and the corresponding mappers in index_mapper and value_mapper.

‘Index’ and ‘value’ correspond to either the horizontal ‘X’ coordinates or the vertical ‘Y’ coordinates depending on the orientation of the plot: for orientation equal to ‘h’ (for horizontal, default), indices are on the X-axis, and values on the Y-axis. The opposite is true when orientation is ‘v’. The convenience properties x_mapper and y_mapper allow accessing the mappers for the two axes in an orientation-independent way.

Finally, the properties index_range and value_range give direct access to the data ranges stored in the index and value mappers.

Axis, labels, and grids
BaseXYPlot defines a few properties that are shortcuts to find axis and grid objects in the underlays and overlays layers of the plot:

hgrid, vgrid
Look into the underlays and overlays layers (in this order) for a PlotGrid object of horizontal / vertical orientation and return it. Return None if none is found.

x_axis, y_axis
Look into the underlays and overlays layers (in this order) for a PlotAxis object positioned to the bottom or top, or to the left or right of plot, respectively. Return the axis, or None if none is found.

labels
Return a list of all PlotLabel objects in the overlays and underlays layers.

TODO: add links to axis and grid documentation

Hittest
BaseXYPlot also provides support for “hit tests”, i.e., for finding the data point or plot line closest to a given screen coordinate. This is typically used to implement interactive tools, for example to select a plot point with a mouse click.

The main functionality is implemented in the method hittest(screen_pt, threshold=7.0, return_distance=False), which accepts screen coordinates (x,y) as input argument screen_pt.
and returns either 1) screen coordinates of the closest point on the plot, or 2) the start and end coordinates of the closest plot line segment, as a tuple \(((x_1, y_1), (x_2, y_2))\). Which of the two behaviors is active is controlled by the attribute `hit_type`, which is one of ‘point’ (default), or ‘line’. If the closest point or line is further than `threshold` pixels away, the methods returns `None`.

Alternatively, users may call the methods `get_closest_point` and `get_closest_line`.

**Others**  Two more attributes are worth mentioning:

- `bgcolor`
  
  This is inherited from the AbstractPlotRenderer interface, but is now set to ‘transparent’ by default.

- `use_downsampling`
  
  If this attribute is `True`, the plot uses downsampling for faster display (default is `False`). In other words, the number of display points depends on the plot size and range, and not on the total number of data points available.

**Note:** At the moment, only `LinePlot` defines a downsampling function, while other plots raise a `NotImplementedError` when this feature is activated.

**2D Plots interface**  The class `chaco.base_2d_plot.Base2DPlot` is the interface for plots that display data defined on a 2D grid, like for example image and contour plots. Just like its companion interface, `BaseXYPlot`, it handles data sources and data mappers, along with convenient shortcuts to find axes, labels and grids.

Unlike other plot renderers, 2D plots draw on the ‘image’ layer, i.e., above any underlay element.

**Data-related traits**  2D plots need two sources of data: one for the coordinates of the 2D grid on which data is displayed, stored in the attribute `index` (a subclass of `GridDataSource`); and one for the values of the data at each point of the grid, `value` (a subclass of `ImageData`). The index data source also needs a 2D mapper, `index_mapper`, to map data coordinates to the screen.

The orientation on screen is set by `orientation` (either ‘h’ – the default – or ‘v’), which controls which of the two coordinates defined in `index` is mapped to the X axis. It is possible to access a mapper for the coordinates corresponding to the individual screen coordinates independently of orientation using the properties `x_mapper` and `y_mapper`.

Finally, `index_range` is a shortcut to the 2D range of the grid data.

**Others**  The attribute `alpha` defines the global transparency value for the whole plot. It ranges from 0.0 for transparent to 1.0 (default) for full intensity.

**Plot types**

This section gives an overview of individual plot classes in Chaco. It is divided in three parts: the first part lists all plot classes implementing the X-Y plots interface, the second all plot classes implementing the 2D plots interface, and finally a part collecting all plot types that do not fall in either category. See the section on plot renderers for a detailed description of the methods and attributes that are common to all plots.

The code to generate the figures in this section can be found in the path `tutorials/user_guide/plot_types/` in the examples directory.

For more complete examples, see also the annotated examples page.
X-Y Plot Types  These plots display information in a two-axis coordinate system and are subclasses of BaseXYPlot.

The common interface for X-Y plots is described in X-Y Plots interface.

Line Plot  Standard line plot implementation. The aspect of the line is controlled by the parameters

- **line_width** The width of the line (default is 1.0)
- **line_style** The style of the line, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
- **render_style** The rendering style of the line plot, one of ‘connectedpoints’ (default), ‘hold’, or ‘connectedhold’

These images illustrate the differences in rendering style:

- `renderstyle='connectedpoints'`

![Graph showing differences in rendering style](image)

- `renderstyle='hold'`
Scatter Plot  Standard scatter plot implementation. The aspect of the markers is controlled by the parameters

**marker** The marker type, one of 'square'(default), 'circle', 'triangle', 'inverted_triangle', 'plus',

- `renderstyle='connectedhold'`
‘cross’, ‘diamond’, ‘dot’, or ‘pixel’. One can also define a new marker shape by setting this parameter to ‘custom’, and set the custom_symbol parameter to a CompiledPath instance (see the file demo/basic/scatter_custom_marker.py in the Chaco examples directory).

**marker_size** Size of the marker in pixels, not including the outline. This can be either a scalar (default is 4.0), or an array with one size per data point.

**line_width** Width of the outline around the markers (default is 1.0). If this is 0.0, no outline is drawn.

**color** The fill color of the marker (default is black).

**outline_color** The color of the outline to draw around the marker (default is black).

This is an example with fixed point size:

![Graph example](image)

The same example, using marker size to map property-tax rate (larger is higher):
Colormapped Scatter Plot  Colormapped scatter plot. Additional information can be added to each point by setting a different color.

The color information is controlled by the `color_data` data source, and the `color_mapper` mapper. A large number of ready-to-use color maps are defined in the module `chaco.default_colormaps`.

In addition to the parameters supported by a scatter plot, a colormapped scatter plot defines these attributes:

- **fill_alpha** Set the alpha value of the points.

- **render_method** Set the sequence in which the points are drawn. It is one of
  - ‘banded’ draw points by color band; this is more efficient but some colors will appear more prominently if there are a lot of overlapping points
  - ‘bruteforce’ set the stroke color before drawing each marker
  - ‘auto’ (default) the approach is selected based on the number of points

In practice, there is not much performance difference between the two methods.

In this example plot, color represents nitric oxides concentration (green is low, red is high):
Using X,Y, color, and size we can display 4 variables at the time. In this example, color is again, and size is nitric oxides concentration:
Candle Plot  A candle plot represents summary statistics of distribution of values for a set of discrete items. Each distribution is characterized by a central line (usually representing the mean), a bar (usually representing one standard deviation around the mean or the 10th and 90th percentile), and two stems (usually indicating the maximum and minimum values).

The positions of the centers, and of the extrema of the bar and stems are set with the following data sources:

- **center_values** Value of the centers. It can be set to `None`, in which case the center is not plotted.
- **bar_min** and **bar_max** Lower and upper values of the bar.
- **min_values** and **max_values** Lower and upper values of the stem. They can be set to `None`, in which case the stems are not plotted.

It is possible to customize the appearance of the candle plot with these parameters:

- **bar_color** (alias of **color**) Fill color of the bar (default is black).
- **bar_line_color** (alias of **outline_color**) Color of the box forming the bar (default is black).
- **center_color** Color of the line indicating the center. If `None`, it defaults to **bar_line_color**.
- **stem_color** Color of the stems and endcaps. If `None`, it defaults to **bar_line_color**.
- **line_width**, **center_width**, and **stem_width** Thickness in pixels of the lines drawing the corresponding elements. If `None`, they default to **line_width**.
- **end_cap** If `False`, the end caps are not plotted (default is `True`).

At the moment, it is not possible to control the width of the central bar and end caps.
Errorbar Plot  A plot with error bars. Note that ErrorBarPlot only plots the error bars, and needs to be combined with a LinePlot if one would like to have a line connecting the central values.

The positions of the extrema of the bars are set by the data sources value_low and value_high.

In addition to the parameters supported by a line plot, an errorbar plot defines these attributes:

- **endcap_size** The width of the endcap bars in pixels.
- **endcap_style** Either ‘bar’ (default) or ‘none’, in which case no endcap bars are plotted.
**Filled Line Plot**  A line plot filled with color to the axis.

FilledLinePlot defines the following parameters:

- **fill_color** The color used to fill the plot.
- **fill_direction** Fill the plot toward the origin (‘down’, default) or towards the axis maximum (‘up’).
- **render_style** The rendering style of the line plot, one of ‘connectedpoints’ (default), ‘hold’, or ‘connectedhold’ (see line plot for a description of the different rendering styles).

FilledLinePlot is a subclass of PolygonPlot, so to set the thickness of the plot line one should use the parameter **edge_width** instead of **line_width**.
**Multi-line Plot**  A line plot showing multiple lines simultaneously.

The values of the lines are given by an instance of `MultiArrayDataSource`, but the lines are rescaled and displaced vertically so that they can be compared without crossing each other.

The relative displacement and rescaling of the lines is controlled by these attributes of `MultiLinePlot`:

- **index**
  The usual array data source for the index data.

- **yindex**
  Array data source for the starting point of each line. Typically, this is set to `numpy.arange(n_lines)`, so that each line is displaced by one unit from the others (the other default parameters are set to work well with this arrangement).

- **use_global_bounds**, **global_min**, **global_max**
  These attributes are used to compute an “amplitude scale” which that the largest trace deviation from its base y-coordinate will be equal to the y-coordinate spacing.

  If `use_global_bounds` is set to False, the maximum of the absolute value of the full data is used as the largest trace deviation. Otherwise, the largest between the absolute value of `global_min` and `global_max` is used instead.

  By default, `use_global_bounds` is set to False and `global_min` and `global_max` to 0.0, which means that one of these value has to be set to create a meaningful plot.

- **scale**, **offset**, **normalized_amplitude**
In addition to the rescaling done using the global bounds (see above), each line is individually scaled by normalized_amplitude (by default this is -0.5, but is normally it should be something like 1.0). Finally, all the lines are moved by offset and multiplied by scale (default are 0.0 and 1.0, respectively).

**MultiLinePlot** also defines the following parameters:

- **line_width**, **line_style**
  Control the thickness and style of the lines, as for line plots.

- **color**, **color_func**
  If color_func is None, all lines have the color defined in color. Otherwise, color_func is a function (or, more in general, a callable) that accept a single argument corresponding to the index of the line and returns a RGBA 4-tuple.

- **fast_clip**
  If True, traces whose base y-coordinate is outside the value axis range are not plotted, even if some of the data in the curve extends into the plot region. (Default is False)

---

**Image and 2D Plots** These plots display information as a two-dimensional image. Unless otherwise stated, they are subclasses of Base2DPlot.

The common interface for 2D plots is described in 2D Plots interface.

**Image Plots** Plot image data, provided as RGB or RGBA color information. If you need to plot a 2D array as an image, use a colormapped scalar plot.
In an ImagePlot, the **index** attribute corresponds to the data coordinates of the pixels (often a GridDataSource). The **index_mapper** maps the data coordinates to screen coordinates (typically using a GridMapper). The **value** is the image itself, wrapped into the data source class ImageData.

A typical use case is to display an image loaded from a file. The preferred way to do this is using the factory method `from_file()` of the class ImageData. For example:

```python
image_source = ImageData.fromfile('capitol.jpg')

w, h = image_source.get_width(), image_source.get_height()
index = GridDataSource(np.arange(w), np.arange(h))
index_mapper = GridMapper(range=DataRange2D(low=(0, 0), high=(w-1, h-1)))

image_plot = ImagePlot(
    index=index,
    value=image_source,
    index_mapper=index_mapper,
    origin='top left',
    **PLOT_DEFAULTS
)
```

The code above displays this plot:
Colormapped Scalar Plot  Plot a scalar field as an image. The image information is given as a 2D array; the scalar values in the 2D array are mapped to colors using a color map.

The basic class for colormapped scalar plots is `CMapImagePlot`. As in image plots, the `index` attribute corresponds to the data coordinates of the pixels (a `GridDataSource`), and the `index_mapper` maps the data coordinates to screen coordinates (a `GridMapper`). The scalar data is passed through the `value` attribute as an `ImageData` source. Finally, a color mapper maps the scalar data to colors. The module `chaco.default_colormaps` defines many ready-to-use colormaps.

For example:

```python
x = np.linspace(-2 * np.pi, +2 * np.pi, NPOINTS)
y = np.linspace(-1.5*np.pi, +1.5*np.pi, NPOINTS)
x, y = np.meshgrid(x, y)
z = scipy.special.jn(2, x) * y * x

index = GridDataSource(xdata=x, ydata=y)
index_mapper = GridMapper(range=DataRange2D(index))

color_source = ImageData(data=z, value_depth=1)
color_mapper = dc.Spectral(DataRange1D(color_source))

cmap_plot = CMapImagePlot(
    index=index,
    index_mapper=index_mapper,
    value=color_source,
    value_mapper=color_mapper,
    **PLOT_DEFAULTS
)
```
Contour Plots

Contour plots represent a scalar-valued 2D function, \( z = f(x, y) \), as a set of contours connecting points of equal value.

Contour plots in Chaco are derived from the base class `BaseContourPlot`, which defines these common attributes:

- **levels**
  
  The values for which to draw a contour. It can be either a list of values (floating point numbers); a positive integer, in which case the range of the value is divided in the given number of equally spaced levels; or “auto” (default), which divides the total range in 10 equally spaced levels.

- **colors**
  
  This attribute is used to define the color of the contours. Colors can be given as a color name, in which case all contours have the same color, as a list of colors, or as a colormap. If the list of colors is shorter than the number of levels, the values are repeated from the beginning of the list. If left unspecified, the contours are plot in black. Colors are associated with levels of increasing value.

- **color_mapper**
  
  If present, the color mapper for the colorbar. TODO: not sure how it works

- **alpha**
  
  Global alpha level for all contours.

Contour Line Plot

Draw a contour plots as a set of lines. In addition to the attributes in `BaseContourPlot`, `ContourLinePlot` defines the following parameters:

- **widths**
  
  The thickness of the contour lines. It can be either a scalar value, valid for all contour lines, or a list of widths. If the list is too short with respect to the number of contour lines, the values are repeated from the beginning of the list. Widths are associated with levels of increasing value.

- **styles**
  
  The style of the lines. It can either be a string that specifies the style for all lines (allowed styles are ‘solid’, ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’), or a list of styles, one for each line. If the list
is too short with respect to the number of contour lines, the values are repeated from the beginning of the list. The default, ‘signed’, sets all lines corresponding to positive values to the style given by the attribute \texttt{positive_style} (default is ‘solid’), and all lines corresponding to negative values to the style given by \texttt{negative_style} (default is ‘dash’).

\textbf{Filled contour Plot}  Draw a contour plot as a 2D image divided in regions of the same color. The class \texttt{ContourPolyPlot} inherits all attributes from \texttt{BaseContourPlot}.
**Polygon Plot**  Draws a polygon given the coordinates of its corners.

The x-coordinate of the corners is given as the *index* data source, and the y-coordinate as the *value* data source. As usual, their values are mapped to screen coordinates by *index_mapper* and *value_mapper*.

In addition, the class *PolygonPlot* defines these parameters:

- **edge_color** The color of the line on the edge of the polygon (default is black).
- **edge_width** The thickness of the edge of the polygon (default is 1.0).
- **edge_style** The line dash style for the edge of the polygon, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
- **face_color** The color of the face of the polygon (default is transparent).
Other Plot Types  This section collects all plots that do not fall in the previous two categories.

**Bar Plot**  Draws a set of rectangular bars, mostly used to plot histograms.

The class `BarPlot` defines the attributes of regular X-Y plots, plus the following parameters:

- **sorting_value** While `value` is a data source defining the upper limit of the bars, `sorting_value` can be used to define their bottom limit. Default is 0. (Note: “upper” and “bottom” assume a horizontal for the plot.)

- **bar_width_type** Determines how to interpret the `bar_width` parameter. If ‘data’ (default), the width is given in the units along the index dimension of the data space. If ‘screen’, the width is given in pixels.

- **bar_width** The width of the bars (see `bar_width_type`).

- **fill_color** The color of the bars.
Quiver Plot  This is a kind of scatter plot which draws an arrow at every point. It can be used to visualize 2D vector fields.

The information about the vector sizes is given through the data source vectors, which returns an Nx2 array. Usually, vectors is an instance of MultiArrayDataSource.

QuiverPlot defines these parameters:

- **line_width**  Width of the lines that trace the arrows (default is 1.0).
- **line_color**  The color of the arrows (default is black).
- **arrow_size**  The length of the arrowheads in pixels.
Polar Plot  Display a line plot in polar coordinates.

The implementation at the moment is at a proof-of-concept stage. The class `PolarLineRenderer` relies on `PolarMapper` to map polar to cartesian coordinates, and adds circular polar coordinate axes.

**Warning:** At the moment, `PolarMapper` does not do a polar to cartesian mapping, but just a linear mapping. One needs to do the transformation by hand.

The aspect of the polar plot can be controlled with these parameters:

- `line_width` Width of the polar plot line (default is 1.0).
- `line_style` The style of the line, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
- `color` The color of the line.
**grid_style**  The style of the lines composing the axis, one of ‘solid’, ‘dot dash’, ‘dash’, ‘dot’ (default), or ‘long dash’.

**grid_visible**  If True (default), the circular part of the axes is drawn.

**origin_axis_visible**  If True (default), the radial part of the axes is drawn.

**origin_axis_width**  Width of the radial axis in pixels (default is 2.0).

---

**Jitter Plot**  A plot showing 1D data by adding a random jitter around the main axis. It can be useful for visualizing dense collections of points. This plot has got a single mapper, called `mapper`.

Useful parameters are:

**jitter_width**  The size, in pixels, of the random jitter around the axis.

**marker**  The marker type, one of ‘square’ (default), ‘circle’, ‘triangle’, ‘inverted_triangle’, ‘plus’, ‘cross’, ‘diamond’, ‘dot’, or ‘pixel’. One can also define a new marker shape by setting this pa-
parameter to ‘custom’, and set the custom_symbol parameter to a CompiledPath instance (see the file demo/basic/scatter_custom_marker.py in the Chaco examples directory).

**marker_size** Size of the marker in pixels, not including the outline (default is 4.0).

**line_width** Width of the outline around the markers (default is 1.0). If this is 0.0, no outline is drawn.

**color** The fill color of the marker (default is black).

**outline_color** The color of the outline to draw around the marker (default is black).

---

**Overlays: axis, legend, grid, etc.**

Overlays are elements that decorate plots, like for example axes, legends, grids, etc.

Overlays are very similar to regular plot elements, and share most of their interface with *plot renderers* (both are subclasses of *chaco.plot_component.PlotComponent*).

In addition, they have a lightweight interface defined in *chaco.abstract_overlay.AbstractOverlay*: the additional features are that 1) they keep a reference to the plot they are decorating in *component*; 2) the background color *bgcolor* is ‘transparent’ by default; 3) they plot *on the ‘overlay’ layer* by default.

TODO: explain how to attach an overlay to an existing plot renderer

There are three important classes of overlays defined in Chaco: *axes, legends*, and *grids*.

**Axes** The Chaco overlay representing a plot axis is defined in the class *PlotAxis*. 
A new axis is created by passing a mapper, usually the mapper defined for the corresponding plot data coordinate. `PlotAxis` also defines a range of attributes to customize the appearance of labels, ticks, and other axis elements. For example, given an X-Y plot renderer, `plot`, we can define a new x-axis as:

```python
AXIS_DEFAULTS = {
    'axis_line_weight': 2,
    'tick_weight': 2,
    'tick_label_font': 'modern 16',
    'title_font': 'modern 20',
}

x_axis = PlotAxis(orientation='bottom',
                 title='My x axis',
                 mapper=plot.x_mapper,
                 component=plot,
                 **AXIS_DEFAULTS)
```

The newly created axis can then be attached to the plot renderer by appending it to its underlays layer:

```python
plot.underlays.append(x_axis)
```

**Attributes** These attributes control the appearance of the axis:

- **title, title_font, title_color, title_spacing**
  
  Define the axis label. `title` is a string or unicode object that is rendered using the given font and color. `title_font` is a string describing a font (e.g. ‘12 pt bold italic’, ‘swiss family Arial’ or ‘default 12’; see TraitKivaFont for details). Finally, `title_spacing` is the space between the axis line and the title (either the number of pixels or ‘auto’, default).

- **tick_weight, tick_color, tick_in, tick_out, tick_visible,**
  
  These attributes control the aspect of the ticks on the axis. If `tick_visible` is True, ticks are represented as lines of color `tick_color` (default is black) and thickness `tick_weight` (in pixels, default is 1). Each line extends into the plot area by `tick_in` pixels and into the label area by `tick_out` pixels (default is 5).

- **tick_label_font, tick_label_color, tick_label_rotate_angle, tick_label_alignment, tick_label_margin, tick_label_offset, tick_label_position,**
  
  These attributes allow to fine-tune the aspect of the tick labels: first of all, the font (e.g. ‘12 pt bold italic’) and color of the labels. The position and orientation of the label can be also be closely controlled: `tick_label_rotate_angle` give the rotation angle (only multiples of 90 degrees are supported); `tick_label_alignment` selects whether the corner (‘corner’) or center (‘edge’, default) of the label are aligned to the corresponding tick (‘corner’ is better for 45 degrees rotation); `tick_label_margin` and `tick_label_offset` control the margin around the tick labels, and their distance from the axis; finally, `tick_label_position` can be set to either ‘outside’ (default) or ‘inside’ depending on whether the labels should be displayed inside or outside the plot area.

- **tick_label_formatter**
  
  By default, tick labels are assumed to be floating point numbers, and are displayed as such after removing trailing zeros and the decimal dot if necessary (e.g., ‘10.000’ will be displayed as ‘10’, and ‘21.10’ as ‘21.1’). The default behavior can be changed by setting `tick_label_formatter` to a callable that takes the value of the tick label and returns a formatted string.

- **tick_interval, tick_generator,**
  
  Locations and distances of ticks are controlled by the attribute `tick_generator` Default is chaco.ticks.auto_ticks or chaco.ticks.log_auto_ticks
**Events**  updated

Fired when the axis’s range bounds change.

**Legend**

**Grid**  TODO: find out how the selection features are organized

TODO: to see how these elements collaborate to build an interactive plot, give complete low-level example of line plot with simple tool and describe the exchange of information

**Tools**

  before axes (axes are overlays) tools, overlays

**Plotting with Chaco**

**The Plot class**

Plot and PlotData

**chaco.shell**

**Low-level Chaco plotting**

1. create instances of PlotRenderer and add them to a Container. There are factory functions in plot_factory that make it simpler

2. Create a Plot instance, use methods to create new plots of different kinds. This automatizes 1) with an Overlay-PlotContainer, i.e., it plots multiple curves on the same element

Plots can be rendered in a traitsui, wx, or qt window

**Embedding Chaco plots**

**Traits UI**

**WxPython**

**Qt/PyQt**

**1.3.2 Plot types**

This section gives an overview of individual plot classes in Chaco. It is divided in three parts: the first part lists all plot classes implementing the $X-Y$ plots interface, the second all plot classes implementing the $2D$ plots interface, and finally a part collecting all plot types that do not fall in either category. See the section on plot renderers for a detailed description of the methods and attributes that are common to all plots.
The code to generate the figures in this section can be found in the path tutorials/user_guide/plot_types/ in the examples directory. For more complete examples, see also the annotated examples page.

**X-Y Plot Types**

These plots display information in a two-axis coordinate system and are subclasses of BaseXYPlot. The common interface for X-Y plots is described in X-Y Plots interface.

**Line Plot**

Standard line plot implementation. The aspect of the line is controlled by the parameters

- `line_width` The width of the line (default is 1.0)
- `line_style` The style of the line, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
- `render_style` The rendering style of the line plot, one of ‘connectedpoints’ (default), ‘hold’, or ‘connectedhold’

These images illustrate the differences in rendering style:

- `renderstyle=’connectedpoints’`

![Line Plot Example](image)

- `renderstyle=’hold’`

![Line Plot Example](image)
Scatter Plot

Standard scatter plot implementation. The aspect of the markers is controlled by the parameters

- `renderstyle='connectedhold'`
**marker** The marker type, one of ‘square’ (default), ‘circle’, ‘triangle’, ‘inverted_triangle’, ‘plus’, ‘cross’, ‘diamond’, ‘dot’, or ‘pixel’. One can also define a new marker shape by setting this parameter to ‘custom’, and set the custom_symbol parameter to a CompiledPath instance (see the file demo/basic/scatter_custom_marker.py in the Chaco examples directory).

**marker_size** Size of the marker in pixels, not including the outline. This can be either a scalar (default is 4.0), or an array with one size per data point.

**line_width** Width of the outline around the markers (default is 1.0). If this is 0.0, no outline is drawn.

**color** The fill color of the marker (default is black).

**outline_color** The color of the outline to draw around the marker (default is black).

This is an example with fixed point size:

![Graph](image)

The same example, using marker size to map property-tax rate (larger is higher):
Colormapped Scatter Plot

Colormapped scatter plot. Additional information can be added to each point by setting a different color.

The color information is controlled by the `color_data` data source, and the `color_mapper` mapper. A large number of ready-to-use color maps are defined in the module `chaco.default_colormaps`.

In addition to the parameters supported by a scatter plot, a colormapped scatter plot defines these attributes:

- **fill_alpha** Set the alpha value of the points.
- **render_method** Set the sequence in which the points are drawn. It is one of
  - ‘banded’ draw points by color band; this is more efficient but some colors will appear more prominently if there are a lot of overlapping points
  - ‘bruteforce’ set the stroke color before drawing each marker
  - ‘auto’ (default) the approach is selected based on the number of points

In practice, there is not much performance difference between the two methods.

In this example plot, color represents nitric oxides concentration (green is low, red is high):
Using X,Y, color, and size we can display 4 variables at the time. In this example, color is again, and size is nitric oxides concentration:
A candle plot represents summary statistics of distribution of values for a set of discrete items. Each distribution is characterized by a central line (usually representing the mean), a bar (usually representing one standard deviation around the mean or the 10th and 90th percentile), and two stems (usually indicating the maximum and minimum values).

The positions of the centers, and of the extrema of the bar and stems are set with the following data sources:

- `center_values` Value of the centers. It can be set to `None`, in which case the center is not plotted.
- `bar_min` and `bar_max` Lower and upper values of the bar.
- `min_values` and `max_values` Lower and upper values of the stem. They can be set to `None`, in which case the stems are not plotted.

It is possible to customize the appearance of the candle plot with these parameters:

- `bar_color` (alias of `color`) Fill color of the bar (default is black).
- `bar_line_color` (alias of `outline_color`) Color of the box forming the bar (default is black).
- `center_color` Color of the line indicating the center. If `None`, it defaults to `bar_line_color`.
- `stem_color` Color of the stems and endcaps. If `None`, it defaults to `bar_line_color`.
- `line_width`, `center_width`, and `stem_width` Thickness in pixels of the lines drawing the corresponding elements. If `None`, they default to `line_width`.
- `end_cap` If `False`, the end caps are not plotted (default is `True`).

Candle Plot

![Median house prices vs Percent lower status in the population](image)
At the moment, it is not possible to control the width of the central bar and end caps.

Errorbar Plot

A plot with error bars. Note that ErrorBarPlot only plots the error bars, and needs to be combined with a LinePlot if one would like to have a line connecting the central values.

The positions of the extrema of the bars are set by the data sources value_low and value_high.

In addition to the parameters supported by a line plot, an errorbar plot defines these attributes:

- **endcap_size** The width of the endcap bars in pixels.
- **endcap_style** Either ‘bar’ (default) or ‘none’, in which case no endcap bars are plotted.
Filled Line Plot

A line plot filled with color to the axis.

`FilledLinePlot` defines the following parameters:

- `fill_color` The color used to fill the plot.
- `fill_direction` Fill the plot toward the origin (‘down’, default) or towards the axis maximum (‘up’).
- `render_style` The rendering style of the line plot, one of ‘connectedpoints’ (default), ‘hold’, or ‘connectedhold’ (see `line plot` for a description of the different rendering styles).

`FilledLinePlot` is a subclass of `PolygonPlot`, so to set the thickness of the plot line one should use the parameter `edge_width` instead of `line_width`. 
Multi-line Plot

A line plot showing multiple lines simultaneously.

The values of the lines are given by an instance of `MultiArrayDataSource`, but the lines are rescaled and displaced vertically so that they can be compared without crossing each other.

The relative displacement and rescaling of the lines is controlled by these attributes of `MultiLinePlot`:

- `index`
  - The usual array data source for the index data.

- `yindex`
  - Array data source for the starting point of each line. Typically, this is set to `numpy.arange(n_lines)`, so that each line is displaced by one unit from the others (the other default parameters are set to work well with this arrangement).

- `use_global_bounds`, `global_min`, `global_max`,
  - These attributes are used to compute an “amplitude scale” which that the largest trace deviation from its base $y$-coordinate will be equal to the $y$-coordinate spacing.

  If `use_global_bounds` is set to False, the maximum of the absolute value of the full data is used as the largest trace deviation. Otherwise, the largest between the absolute value of `global_min` and `global_max` is used instead.

  By default, `use_global_bounds` is set to False and `global_min` and `global_max` to 0.0, which means that one of these value has to be set to create a meaningful plot.
In addition to the rescaling done using the global bounds (see above), each line is individually scaled by `normalized_amplitude` (by default this is -0.5, but is normally it should be something like 1.0). Finally, all the lines are moved by `offset` and multiplied by `scale` (default are 0.0 and 1.0, respectively).

`MultiLinePlot` also defines the following parameters:

- `line_width`, `line_style`
  Control the thickness and style of the lines, as for `line plots`.

- `color`, `color_func`
  If `color_func` is None, all lines have the color defined in `color`. Otherwise, `color_func` is a function (or, more in general, a callable) that accept a single argument corresponding to the index of the line and returns a RGBA 4-tuple.

- `fast_clip`
  If True, traces whose base y-coordinate is outside the value axis range are not plotted, even if some of the data in the curve extends into the plot region. (Default is False)

### Image and 2D Plots

These plots display information as a two-dimensional image. Unless otherwise stated, they are subclasses of `Base2DPlot`.

The common interface for 2D plots is described in `2D Plots interface`.
Image Plots

Plot image data, provided as RGB or RGBA color information. If you need to plot a 2D array as an image, use a colormapped scalar plot.

In an ImagePlot, the index attribute corresponds to the data coordinates of the pixels (often a GridDataSource). The index_mapper maps the data coordinates to screen coordinates (typically using a GridMapper). The value is the image itself, wrapped into the data source class ImageData.

A typical use case is to display an image loaded from a file. The preferred way to do this is using the factory method from_file() of the class ImageData. For example:

```python
image_source = ImageData.fromfile('capitol.jpg')

w, h = image_source.get_width(), image_source.get_height()
index = GridDataSource(np.arange(w), np.arange(h))
index_mapper = GridMapper(range=DataRange2D(low=(0, 0),
                                        high=(w-1, h-1)))

image_plot = ImagePlot(
    index=index,
    value=image_source,
    index_mapper=index_mapper,
    origin='top left',
    **PLOT_DEFAULTS
)
```

The code above displays this plot:
Colormapped Scalar Plot

Plot a scalar field as an image. The image information is given as a 2D array; the scalar values in the 2D array are mapped to colors using a color map.

The basic class for colormapped scalar plots is `CMapImagePlot`. As in `image plots`, the `index` attribute corresponds to the data coordinates of the pixels (a `GridDataSource`), and the `index_mapper` maps the data coordinates to screen coordinates (a `GridMapper`). The scalar data is passed through the `value` attribute as an `ImageData` source. Finally, a color mapper maps the scalar data to colors. The module `chaco.default_colormaps` defines many ready-to-use colormaps.

For example:

```python
xs = np.linspace(-2 * np.pi, +2 * np.pi, NPOINTS)
y = np.linspace(-1.5*np.pi, +1.5*np.pi, NPOINTS)
x, y = np.meshgrid(xs, ys)
z = scipy.special.jn(2, x)*y*x

index = GridDataSource(xdata=xs, ydata=ys)
index_mapper = GridMapper(range=DataRange2D(index))

color_source = ImageData(data=z, value_depth=1)
color_mapper = dc.Spectral(DataRange1D(color_source))

cmap_plot = CMapImagePlot(
    index=index,
    index_mapper=index_mapper,
    value=color_source,
)
```

```
Contour Plots

Contour plots represent a scalar-valued 2D function, \( z = f(x, y) \), as a set of contours connecting points of equal value. Contour plots in Chaco are derived from the base class `BaseContourPlot`, which defines these common attributes:

- **levels** levels is used to define the values for which to draw a contour. It can be either a list of values (floating point numbers); a positive integer, in which case the range of the value is divided in the given number of equally spaced levels; or “auto” (default), which divides the total range in 10 equally spaced levels

- **colors** This attribute is used to define the color of the contours. colors can be given as a color name, in which case all contours have the same color, as a list of colors, or as a colormap. If the list of colors is shorter than the number of levels, the values are repeated from the beginning of the list. If left unspecified, the contours are plot in black. Colors are associated with levels of increasing value.

- **color_mapper** If present, the color mapper for the colorbar. TODO: not sure how it works

- **alpha** Global alpha level for all contours.

Contour Line Plot  Draw a contour plots as a set of lines. In addition to the attributes in `BaseContourPlot`, `ContourLinePlot` defines the following parameters:
**widths** The thickness of the contour lines. It can be either a scalar value, valid for all contour lines, or a list of widths. If the list is too short with respect to the number of contour lines, the values are repeated from the beginning of the list. Widths are associated with levels of increasing value.

**styles** The style of the lines. It can either be a string that specifies the style for all lines (allowed styles are ‘solid’, ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’), or a list of styles, one for each line. If the list is too short with respect to the number of contour lines, the values are repeated from the beginning of the list. The default, ‘signed’, sets all lines corresponding to positive values to the style given by the attribute `positive_style` (default is ‘solid’), and all lines corresponding to negative values to the style given by `negative_style` (default is ‘dash”).

---

**Filled contour Plot** Draw a contour plot as a 2D image divided in regions of the same color. The class `ContourPolyPlot` inherits all attributes from `BaseContourPlot`.
Polygon Plot

Draws a polygon given the coordinates of its corners.

The x-coordinate of the corners is given as the index data source, and the y-coordinate as the value data source. As usual, their values are mapped to screen coordinates by index_mapper and value_mapper.

In addition, the class PolygonPlot defines these parameters:

- **edge_color** The color of the line on the edge of the polygon (default is black).
- **edge_width** The thickness of the edge of the polygon (default is 1.0).
- **edge_style** The line dash style for the edge of the polygon, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
- **face_color** The color of the face of the polygon (default is transparent).
Other Plot Types

This section collects all plots that do not fall in the previous two categories.

Bar Plot

Draws a set of rectangular bars, mostly used to plot histograms.

The class `BarPlot` defines the attributes of regular X-Y plots, plus the following parameters:

- `sorting_value` While `value` is a data source defining the upper limit of the bars, `sorting_value` can be used to define their bottom limit. Default is 0. (Note: “upper” and “bottom” assume a horizontal for the plot.)

- `bar_width_type` Determines how to interpret the `bar_width` parameter. If ‘data’ (default), the width is given in the units along the index dimension of the data space. If ‘screen’, the width is given in pixels.

- `bar_width` The width of the bars (see `bar_width_type`).

- `fill_color` The color of the bars.
Quiver Plot

This is a kind of scatter plot which draws an arrow at every point. It can be used to visualize 2D vector fields.

The information about the vector sizes is given through the data source vectors, which returns an Nx2 array. Usually, vectors is an instance of MultiArrayDataSource.

QuiverPlot defines these parameters:

- **line_width** Width of the lines that trace the arrows (default is 1.0).
- **line_color** The color of the arrows (default is black).
- **arrow_size** The length of the arrowheads in pixels.
Polar Plot

Display a line plot in polar coordinates.

The implementation at the moment is at a proof-of-concept stage. The class `PolarLineRenderer` relies on `PolarMapper` to map polar to cartesian coordinates, and adds circular polar coordinate axes.

**Warning:** At the moment, `PolarMapper` does not do a polar to cartesian mapping, but just a linear mapping. One needs to do the transformation by hand.

The aspect of the polar plot can be controlled with these parameters:

- **line_width** Width of the polar plot line (default is 1.0).
- **line_style** The style of the line, one of ‘solid’ (default), ‘dot dash’, ‘dash’, ‘dot’, or ‘long dash’.
**color**  The color of the line.

**grid_style**  The style of the lines composing the axis, one of ‘solid’, ‘dot dash’, ‘dash’, ‘dot’ (default), or ‘long dash’.

**grid_visible**  If True (default), the circular part of the axes is drawn.

**origin_axis_visible**  If True (default), the radial part of the axes is drawn.

**origin_axis_width**  Width of the radial axis in pixels (default is 2.0).

---

**Jitter Plot**

A plot showing 1D data by adding a random jitter around the main axis. It can be useful for visualizing dense collections of points. This plot has got a single mapper, called \texttt{mapper}.

Useful parameters are:
jitter_width  The size, in pixels, of the random jitter around the axis.

marker  The marker type, one of ‘square’ (default), ‘circle’, ‘triangle’, ‘inverted_triangle’, ‘plus’, ‘cross’, ‘diamond’, ‘dot’, or ‘pixel’. One can also define a new marker shape by setting this parameter to ‘custom’, and set the custom_symbol parameter to a CompiledPath instance (see the file demo/basic/scatter_custom_marker.py in the Chaco examples directory).

marker_size  Size of the marker in pixels, not including the outline (default is 4.0).

line_width  Width of the outline around the markers (default is 1.0). If this is 0.0, no outline is drawn.

color  The fill color of the marker (default is black).

outline_color  The color of the outline to draw around the marker (default is black).

TODO: add description of color bar class

1.3.3 Containers and Layout

Chaco containers

It is quite common to need to display multiple data side by side. In order to arrange plots and other components (e.g., colorbars, legends) in a single panel, Chaco uses containers to organize the layout.

Chaco implements 4 different containers: HPlotContainer and VPlotContainer, GridPlotContainer, and OverlayPlotContainer.

All containers are derived from the base class BasePlotContainer, and share a common interface:

• __init__(*components, **parameters) (constructor of the container object): The constructor of a plot container takes a sequence of components, which are added to the container itself, and a set of keyword
arguments, which are used to initialize the parameters of the container. For example:

```python
container = HPlotContainer(scatter_plot, line_plot, spacing=100)
```

creates a container with horizontal layout containing two plots (`scatter_plot` and `line_plot`), with a spacing of 100 pixels between them.

- **add(**components**)**: Append one or more plots to the ones already present in the container. For example, this is equivalent to the code above:

  ```python
container = HPlotContainer(spacing=100)
  container.add(line_plot, scatter_plot)
  ```

- **remove**(self, **components**): Remove a sequence of components from the container

- **insert**(index, component): Inserts a component at a specific position in the components list

**Note:** Each plot can have only one container, so adding the same plot to a second container will remove it from the first one. In the same way, adding the same plot multiple times will not have create multiple copies. Instead, one should create multiple plots objects.

E.g., this code:

```python
# Create a vertical container containing two horizontal containers
h_container1 = HPlotContainer()
h_container2 = HPlotContainer()
outer_container = VPlotContainer(h_container1, h_container2,
                                 stack_order="top_to_bottom")

# Add the three plots to the first container
h_container1.add(scatter_plot, line_plot1, line_plot2)

# Now add the first line plot to the second container => it is removed
# from the first, as each plot can only have one container
h_container2.add(line_plot1)
```

results in this layout:
HPlotContainer and VPlotContainer

HPlotContainer and VPlotContainer display a set of components in an horizontal and vertical stack, respectively, as shown in these simple examples:

In both cases, a series of line plots and scatter plots is added to an HPlotContainer or a VPlotContainer:

```
# Create the data and the PlotData object
x = linspace(-14, 14, 100)
y = sin(x) * x**3
plotdata = ArrayPlotData(x=x, y=y)

# Create a scatter plot
scatter_plot = Plot(plotdata)
scatter_plot.plot(("x", "y"), type="scatter", color="blue")

# Create a line plot
line_plot = Plot(plotdata)
line_plot.plot(("x", "y"), type="line", color="blue")

# Create a horizontal container and put the two plots inside it
```
container = HPlotContainer(line_plot, scatter_plot)
self.plot = container

HPlotContainer is also used often to display a colorbar or legend to the side of a plot. For example, this plot was created using a color-mapped scatter plot and a colorbar inside a horizontal container:

```python
# Create the plot
plot = Plot(data)
plot.plot(("index", "value", "color"), type="cmap_scatter",
    color_mapper=jet)

# Create the colorbar, handing in the appropriate range and colormap
colormap = plot.color_mapper
colorbar = ColorBar(index_mapper=LinearMapper(range=colormap.range),
    color_mapper=colormap,
    orientation='v',
    resizable='v',
    width=30,
    padding=20)

colorbar.padding_top = plot.padding_top
colorbar.padding_bottom = plot.padding_bottom

# Create a container to position the plot and the colorbar side-by-side
container = HPlotContainer(plot, colorbar)
```

**HPlotContainer parameters** This is a list of parameters that are specific to `HPlotContainer`:

- **stack_order**: The order in which components in the plot container are laid out. The default behavior is left-to-right.
  
  ```python
  stack_order = Enum("left_to_right", "right_to_left")
  ```

- **spacing**: The amount of space to put between components.
  
  ```python
  spacing = Float(0.0)
  ```

- **valign**: The vertical alignment of objects that don’t span the full height.
valign = Enum("bottom", "top", "center")

VPlotContainer parameters  This is a list of parameters that are specific to VPlotContainer

- stack_order: The order in which components in the plot container are laid out. The default behavior is bottom-to-top.
  
  stack_order = Enum("bottom_to_top", "top_to_bottom")

- spacing: The amount of space to put between components.
  
  spacing = Float(0.0)

- halign: The horizontal alignment of objects that don’t span the full width.
  
  halign = Enum("left", "right", "center")

See Also:

HPlotContainer and VPlotContainer in action. See demo/financial_plot.py, demo/two_plots.py, demo/advanced/scalar_image_function_inspector.py, and demo/basc/cmap_scatter.py in the Chaco examples directory.

GridPlotContainer

Just as the name suggests, a GridPlotContainer lays out plots in a regular grid.

Unlike the previous containers, one has to specify in advance the number of rows and columns in the plot. Plots with different sizes and/or aspect ratios are aligned according to the parameters halign and valign.

For example, to generate this plot

one needs to create six plots of fixed height and add them successively (left-to-right, top-to-bottom) to the GridPlotContainer. Plots are aligned to the top by setting valign = ‘top’.
The complete code looks like this:

class GridContainerExample(HasTraits):

    plot = Instance(GridPlotContainer)

    traits_view = View(
        Item('plot', editor=ComponentEditor(), show_label=False),
        width=1000, height=600, resizable=True
    )

def _plot_default(self):
    # Create a GridContainer to hold all of our plots: 2 rows, 3 columns
    container = GridPlotContainer(shape=(2,3),
        spacing=(10,5),
        valign='top',
        bgcolor='lightgray')

    # Create x data
    x = linspace(-5.0, 15.0, 100)
    pd = ArrayPlotData(index = x)

    # Plot some Bessel functions and add the plots to our container
    for i in range(6):
        data_name = 'y{}'.format(i)
        pd.set_data(data_name, jn(i,x))

        plot = Plot(pd)
        plot.plot(('index', data_name),
            color=COLOR_PALETTE[i],
            line_width=3.0)

        # Set each plot’s aspect based on its position in the grid
        plot.set(height=((i % 3) + 1)*50, resizable='h')

    # Add to the grid container
    container.add(plot)

    return container

GridPlotContainer parameters

This is a list of parameters that are specific to GridPlotContainer:

- **valign**: The vertical alignment of objects that don’t span the full height.:
  
  `valign = Enum("bottom", "top", "center")`

- **halign**: The horizontal alignment of objects that don’t span the full width.:
  
  `halign = Enum("left", "right", "center")`

- **spacing**: A tuple or list of (h_spacing, v_spacing), giving spacing values for the horizontal and vertical direction. Default is (0, 0).

See Also:

GridPlotContainer in action. See demo/basic/grid_container.py and demo/basic/grid_container_aspect_ratio.py in the Chaco examples directory.
OverlayPlotContainer

Overlay containers OverlayPlotContainers lay out plots on top of each other. The chaco.plot.Plot class in Chaco is a special subclass of OverlayPlotContainer.

Overlay containers can be used to create “inset” plots. In the following code, for instance, we create a zoomable plot with an fixed inset showing the full data:

```python
class OverlayContainerExample(HasTraits):

    plot = Instance(OverlayPlotContainer)

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

    def _plot_default(self):
        # Create data
        x = linspace(-5, 15.0, 100)
        y = jn(3, x)
        pd = ArrayPlotData(index=x, value=y)

        zoomable_plot = Plot(pd)
        zoomable_plot.plot(('index', 'value'),
            name='external', color='red', line_width=3)

        # Attach tools to the plot
        zoom = ZoomTool(component=zoomable_plot,
            tool_mode="box", always_on=False)
        zoomable_plot.overlays.append(zoom)
        zoomable_plot.tools.append(PanTool(zoomable_plot))

        # Create a second inset plot, not resizable, not zoom-able
        inset_plot = Plot(pd)
        inset_plot.plot(('index', 'value'), color='blue')
        inset_plot.set(resizable='',
            bounds = [250, 150],
            position = [450, 350],
            border_visible = True
        )

        # Create a container and add our plots
        container = OverlayPlotContainer()
        container.add(zoomable_plot)
        container.add(inset_plot)
        return container
```

The code above generates this plot:
See Also:

GridPlotContainer in action. See demo/basic/inset_plot.py and
demo/advanced/scalar_image_function_inspector.py in the Chaco examples direc-
tory. To learn more about sharing axes on overlapping plots, see demo/multiaxis.py and
demo/multiaxis_with_Plot.py.

Sizing, rendering, events

Containers are responsible for a handling communication with the components it contains, including defining the
rendering order, dispatching events, and determining sizes.

Sizing

Containers are the elements that set sizes and do layout. Components within containers declare their preferences,
which are taken into account by their container to set their final aspect.

The basic traits that control the layout preferences of a component are:

- **resizable**, a string indicating in which directions the component can be resized. Its value is one of "" (not
  resizable), 'h' (resizable in the horizontal direction), 'v' (resizable in the vertical direction), 'hv' (resizable
  in both, default).

- **aspect_ratio**, the ratio of the component’s width to its height. This is used by the component itself to
  maintain bounds when the bounds are changed independently. Default is **None**, meaning that the aspect ratio is
  not enforced.

- **padding_left, padding_right, padding_top, padding_bottom** set the amount of padding space to
  leave around the component (default is 0). The property **padding** allows to set all of them as a tuple (left,
  right, top, bottom).

- **auto_center**, controls the behavior when the component’s bounds are set to a value that does not conform
  its aspect ratio. If **True** (default), the component centers itself in the free space.
• fixed_preferred_size: If the component is resizable, this attribute specifies the amount of space that the component would like to get in each dimension, as a tuple (width, height). This attribute can be used to establish relative sizes between resizable components in a container: if one component specifies, say, a fixed preferred width of 50 and another one specifies a fixed preferred width of 100, then the latter component will always be twice as wide as the former.

You can get access to the actual bounds of the component, (including padding and border) using the outer properties:

• outer_position, the x,y point of the lower left corner of the padding outer box around the component. Use set_outer_position() to change these values.
• outer_bounds, the number of horizontal and vertical pixels in the padding outer box. Use set_outer_bounds() to change these values.
• outer_x, outer_y, outer_x2, outer_y2:, :attr:'outer_width, outer_height: coordinates of lower-left pixel of the box, coordinates of the upper-right pixel of the box, width and height of the outer box in pixels

See also the documentation of the class enable.component.Component for more details about the internal parameters of Chaco components.

The container can set the attribute fit_components to control if it should resize itself to fit its components. Allowed values are ” (do not resize, default), ’h’ (resize in the horizontal direction), ’v’ (resize in the vertical direction), ’hv’ (resize in both).

Rendering order

Every plot component has several layers:

1. background: Background image, shading, and borders
2. underlay: Axes and grids
3. image: A special layer for plots that render as images. This is in a separate layer since these plots must all render before non-image plots.
4. plot: The main plot area
5. annotation: Lines and text that are conceptually part of the “plot” but need to be rendered on top of everything else in the plot.
6. overlay: Legends, selection regions, and other tool-drawn visual elements

These are defined by DEFAULT_DRAWING_ORDER, and stored in the drawing_order trait.

Complexity arises when you have multiple components in a container: How do their layers affect each other? Do you want the “overlay” layer of a component to draw on top of all components? Do you want the “background” elements to be behind everything else?

This is resolved by the unified_draw trait. The container will draw all layers in succession. If a component sets unified_draw to False (default), the container will ask it to draw the corresponding layer as it is reached in the loop. If unified_draw is True, the whole component will draw in one go when the container reaches the layer specified in the attribute component.draw_layer, which by default is ’plot’.

For example, if you want a plot to act as an overlay, you could set unified_draw = True and draw_layer = ’overlay’. These values tell the container to render the component when it gets to the ‘overlay’ layer.

Set overlay_border to True if you want the border to draw as part of the overlay; otherwise it draws as part of the background. By default, the border is drawn just inside the plot area; set inset_border to False to draw it just outside the plot area.
**Backbuffer** A backbuffer provides the ability to render into an offscreen buffer, which is blitted on every draw, until it is invalidated. Various traits such as `use_backbuffer` and `backbuffer_padding` control the behavior of the backbuffer. A backbuffer is used for non-OpenGL backends, such as `agg` and on OS X. If `use_backbuffer` is False, a backbuffer is never used, even if a backbuffer is referenced by a component.

### Dispatching events

The logic of event dispatching is defined in the ‘enable’ library, which defines the superclasses for Chaco’s containers and components. In summary, when a component gets an event, it dispatches it to:

1. its overlays, in reverse order that they were added and are drawn
2. itself, so that any event handler methods on itself get called
3. its underlays, in reverse order that they were added and are drawn
4. its listener tools

On each of these elements, Chaco looks for a method of the form `{component_state}_{event_name}`. For example, in response to the user pressing the left mouse button on a tool in state `normal` (the default state, see `Tool states`), Chaco would look for a method called `normal_left_down`.

If this exists, the event is dispatched and the component decides whether to handle the element and set `event.handled = True`, in which case the dispatch chain is interrupted.

**Note:** If the attribute `auto_handle_event` of the component is set to `True`, calling the event method automatically sets `event.handled = True`.

Possible event names are:

- `left_down`
- `left_up`
- `left_dclick`
- `right_down`
- `right_up`
- `right_dclick`
- `middle_down`
- `middle_up`
- `middle_dclick`
- `mouse_move`
- `mouse_wheel`
- `mouse_enter`
- `mouse_leave`
- `key_pressed`
- `key_released`
- `character`
- `dropped_on`
- `drag_over`
- `drag_enter`
- `drag_leave`

Most objects default to having just a single event state, which is the “normal” event state. To make a component that handled a left-click, you could subclass `PlotComponent`, and implement `normal_left_down()` or `normal_left_up()`. The signature for handler methods is just one parameter, which is an event object that is an instance of (a subclass of) `BasicEvent`. Subclasses of `BasicEvent` are `MouseEvent`, `DragEvent`, `KeyEvent`,...
and BlobEvent and BlobFrameEvent (for multitouch). It’s fairly easy to extend this event system with new kinds of events and new suffixes (as was done for multitouch).

Events contain a reference to the GUI toolkit window that generated them as event.window. A common pattern is for component to call methods on the window to do things like set a tooltip or create a context menu. A draw or update of the window does not actually happen until the next paint(). By that time, the component no longer has a reference to the event or the event’s window, but uses instead its own reference to the window, self.window.

See also the documentation of the enable library, which gives more details about the event dispatching happening at that level.
1.3.4 Tools and Overlays

Overview

Chaco, Enable, and Event Dispatch

A Basic Tool

A Basic Overlay

Interaction Tools

PanTool

ZoomTool

RectZoom

DragZoom

LegendTool

DataLabelTool

MoveTool

Inspector-type Tools

DataPrinter

LineInspector

ScatterInspector

CursorTool

HighlightTool

ImageInspectorTool

TraitsTool

Selection Tools

RangeSelection

LassoSelection

SelectTool

DrawPointsTool
Basics

How do I...

• render data to an image file?

```python
def save_plot(plot, filename, width, height):
    plot.outer_bounds = [width, height]
    plot.do_layout(force=True)
    gc = PlotGraphicsContext((width, height), dpi=72)
    gc.render_component(plot)
    gc.save(filename)
```

• integrate a Chaco plot into my WX app?

```python
import wx
from scipy import arange
from scipy.special import jn
from chaco.api import HPlotContainer, create_line_plot
from enable.wx_backend.api import Window

class PlotFrame(wx.Frame):
    def __init__(self, *args, **kw):
        kw["size"] = (850, 550)
        wx.Frame.__init__( *(self,) + args, **kw )
        self.plot_window = Window(self, component=self._create_plot())
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(self.plot_window.control, 1, wx.EXPAND)
        self.SetSizer(sizer)
        self.SetAutoLayout(True)
        self.Show(True)
        return

    def _create_plot(self):
        x = arange(-5.0, 15.0, 20.0/100)
        y = jn(0, x)
        plot = create_line_plot((x,y), bgcolor="white",
                                 add_grid=True, add_axis=True)
        container = HPlotContainer(spacing=20, padding=50, bgcolor="lightgray")
        container.add(plot)
        return container

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = PlotFrame(None)
    app.MainLoop()
```

• integrate a Chaco plot into my QT app?

• integrate a Chaco plot into my Traits UI?
class MyPlot(HasTraits):
    plot = Instance(Plot)

    traits_view = View(Item('plot', editor=ComponentEditor()))

    def __init__(self, index, data_series, **kw):
        super(MyPlot, self).__init__(**kw)

        plot_data = ArrayPlotData(index=index)
        plot_data.set_data('data_series', data_series)
        self.plot = Plot(plot_data)
        self.plot.plot(('index', 'data_series'))

index = numpy.array([1, 2, 3, 4, 5])
data_series = index**2

my_plot = MyPlot(index, data_series)
my_plot.configure_traits()

• make an application to render many streams of data?

def plot_several_series(index, series_list):
    plot_data = ArrayPlotData(index=index)
    plot = Plot(plot_data)

    for i, data_series in enumerate(series_list):
        series_name = "series_%d" % i
        plot_data.set_data(series_name, data_series)
        plot.plot(('index', series_name))

• make a plot the right size?

def resize_plot(plot, width, height):
    plot.outer_bounds = [width, height]

• copy a plot the the clipboard?

def copy_to_clipboard(plot):
    # WX specific, though QT implementation is similar using
    # QImage and QClipboard
    import wx

    width, height = plot.outer_bounds

    gc = PlotGraphicsContext((width, height), dpi=72)
    gc.render_component(plot_component)

    # Create a bitmap the same size as the plot
    # and copy the plot data to it

    bitmap = wx.BitmapFromBufferRGBA(width+1, height+1, gc.bmp_array.flatten())
    data = wx.BitmapDataObject()
    data.SetBitmap(bitmap)

    if wx.TheClipboard.Open():
        wx.TheClipboard.SetData(data)
        wx.TheClipboard.Close()
    else:
Layout and Rendering

How do I...

• put multiple plots in a single window?
• change the background color?

```python
def make_black_plot(index, data_series):
    plot_data = ArrayPlotData(index=index)
    plot_data.set_data('data_series', data_series)
    plot = Plot(plot_data, bgcolor='black')
    plot.plot(('index', 'data_series'))
```

```python
def change_bgcolor(plot):
    plot.bgcolor = 'black'
```

• turn off borders?

```python
def make_borderless_plot(index, data_series):
    plot_data = ArrayPlotData(index=index)
    plot_data.set_data('data_series', data_series)
    plot = Plot(plot_data, border_visible=False)
    plot.plot(('index', 'data_series'))
```

```python
def change_to_borderless_plot(plot):
    plot.border_visible = False
```

Writing Components

How do I...

• compose multiple renderers?
• write a custom renderer?
• write a custom overlay/underlay?
• write a custom tool?
• write a new container?

Advanced

How do I...

• properly change/override draw dispatch?
• modify event dispatch?
• customize backbuffering?
• embed custom/native WX widgets on the plot?
1.3.7 Frequently Asked Questions

Where does the name “Chaco” come from?

It is named after Chaco Canyon, which had astronomical markings that served as an observatory for Native Americans. The original version of Chaco was built as part of a project for the Space Telescope Science Institute. This is also the origin of the name “Kiva” for our vector graphics layer that Chaco uses for rendering.

What are the pros and cons of Chaco vs. matplotlib?

This question comes up quite a bit. The bottom line is that the two projects initially set out to do different things, and although each project has grown a lot of overlapping features, the different original charters are reflected in the capabilities and feature sets of the two projects.

Here is an excerpt from a thread about this question on the enthought-dev mailing list.

Gael Varoquaux’s response:

On Fri, May 11, 2007 at 10:03:21PM +0900, Bill Baxter wrote:

> Just curious. What are the pros and cons of chaco vs matplotlib?

To me it seem the big pro of chaco is that it is much easier to use in a "programatic way" (I have no clue this means something in English). It is fully traited and rely quite a lot on inversion of control (sorry, I love this concept, so it has become my new buzz-word). You can make very nice object oriented interactive code.

Another nice aspect is that it is much faster than MPL.

The cons are that it is not as fully featured as MPL, that it does not has an as nice interactively useable functional interface (ie chaco.shell vs pylab) and that it is not as well documented and does not have the same huge community.

I would say that the codebase of chaco is nicer, but than if you are not developing interactive application, it is MPL is currently an option that is likely to get you where you want to go quicker. Not that I wouldn’t like to see chaco building up a bit more and becoming **the** reference.

Developers, if you want chaco to pick up momentum, give it a pylab-like interface (as close as you can to pylab) !

My 2 cents,
Gael

Peter Wang’s response (excerpt):

On May 11, 2007, at 8:03 AM, Bill Baxter wrote:

> Just curious. What are the pros and cons of chaco vs matplotlib?

You had to go and ask, didn’t you? :) There are many more folks here who have used MPL more extensively than myself, so I’ll defer the comparisons to them. (Gael, as always, thanks for your comments and feedback!) I can comment, however, on the key goals of Chaco.

Chaco is a plotting toolkit targeted towards developers for building
interactive visualizations. You hook up pieces to build a plot that is then easy to inspect, interact with, add configuration UIs for (using Traits UI), etc. The layout of plot areas, the multiplicity and types of renderers within those windows, the appearance and locations of axes, etc. are all completely configurable since these are all first-class objects participating in a visual canvas. They can all receive mouse and keyboard events, and it’s easy to subclass them (or attach tools to them) to achieve new kinds of behavior. We’ve tried to make all the plot renderers adhere to a standard interface, so that tools and interactors can easily inspect data and map between screen space and data space. Once these are all hooked up, you can swap out or update the data independently of the plots.

One of the downsides we had a for a while was that this rich set of objects required the programmer to put several different classes together just to make a basic plot. To solve this problem, we’ve assembled some higher-level classes that have the most common behaviors built-in by default, but which can still be easily customized or extended. It’s clear to me that this is a good general approach to preserving flexibility while reducing verbosity.

At this point, Chaco is definitely capable of handling a large number of different plotting tasks, and a lot of them don’t require too much typing or hacking skills. (Folks will probably require more documentation, however, but I’m working on that. :) I linked to the source for all of the screenshots in the gallery to demonstrate that you can do a lot of things with Chaco in a few dozen lines of code. (For instance, the audio spectrogram at the bottom of the gallery is just a little over 100 lines.)

Fundamentally, I like the Chaco model of plots as compositions of interactive components. This really helps me think about visualization apps in a modular way, and it "fits my head". (Of course, the fact that I wrote much of it might have something to do with that as well. ;) The goal is to have data-related operations clearly happen in one set of objects, the view layout and configuration happen in another, and the interaction controls fit neatly into a third. IMHO a good toolkit should help me design/architect my application better, and we definitely aspire to make Chaco meet that criterion.

Finally, one major perk is that since Chaco is built completely on top of traits and its event-based component model, you can call edit_traits() on any visual component from within your app (or ipython) and get a live GUI that lets you tweak all of its various parameters in realtime. This applies to the axis, grid, renderers, etc. This seems so natural to me that I sometimes forget what an awesome feature it is. :)

1.3.8 Annotated Examples

This section describes each of the examples provided with Chaco. Each example is designed to be a stand-alone demonstration of some of Chaco’s features. Though they are simple, many of the examples have capabilities that are difficult to find in other plotting packages.

Extensibility is a core design goal of Chaco, and many people have used the examples as starting points for their own applications.
bar_plot_stacked.py

An example showing Chaco’s BarPlot class.

source: bar_plot_stacked.py

bigdata.py

Demonstrates chaco performance with large datasets.

There are 10 plots with 100,000 points each. Right-click and drag to create a range selection region. The region can be moved around and resized (drag the edges). These interactions are very fast because of the backbuffering built into chaco.

Zooming with the mousewheel and the zoombox (as described in simple_line.py) is also available, but panning is not.

source: bigdata.py
cursor_tool_demo.py

A Demonstration of the CursorTool functionality

Left-button drag to move the cursors round. Right-drag to pan the plots. ‘z’-key to Zoom

source: cursor_tool_demo.py
**data_labels.py**

Draws a line plot with several points labelled. Demonstrates how to annotate plots.

source: data_labels.py
**data_view.py**

Example of how to use a DataView and bare renderers to create plots.

source: data_view.py
**edit_line.py**

Allows editing of a line plot.

source: *edit_line.py*
financial_plot.py

Implementation of a standard financial plot visualization using Chaco renderers and scales. Right-clicking and selecting an area in the top window zooms in the corresponding area in the lower window.

source: financial_plot.py
financial_plot_dates.py

Implementation of a standard financial plot visualization using Chaco renderers and scales. Right-clicking and selecting an area in the top window zooms in the corresponding area in the lower window. This differs from the financial_plot.py example in that it uses a date-oriented axis.

source: financial_plot_dates.py
multiaxis.py

Draws several overlapping line plots like simple_line.py, but uses a separate Y range for each plot. Also has a second Y-axis on the right hand side. Demonstrates use of the BroadcasterTool.

source: multiaxis.py
multiaxis_using_Plot.py

Draws some x-y line and scatter plots. On the left hand plot:

- Left-drag pans the plot.
- Mousewheel up and down zooms the plot in and out.
- Pressing “z” opens the Zoom Box, and you can click-drag a rectangular region to zoom. If you use a sequence of zoom boxes, pressing alt-left-arrow and alt-right-arrow moves you forwards and backwards through the “zoom history”.

source: multiaxis_using_Plot.py
noninteractive.py

This demonstrates how to create a plot offscreen and save it to an image file on disk. The image is what is saved.

source: noninteractive.py
range_selection_demo.py

Demo of the RangeSelection on a line plot. Left-click and drag creates a horizontal range selection; this selection can then be dragged around, or resized by dragging its edges.

source: range_selection_demo.py
**scales_test.py**

Draws several overlapping line plots.

Double-clicking on line or scatter plots opens a Traits editor for the plot.

source: scales_test.py
**simple_line.py**

Draws several overlapping line plots.

Double-clicking on line or scatter plots opens a Traits editor for the plot.

*source: simple_line.py*
tornado.py

Tornado plot example from Brennan Williams.
source: tornado.py
two_plots.py

Demonstrates plots sharing datasources, ranges, etc...

source: two_plots.py
vertical_plot.py

Draws a static plot of bessel functions, oriented vertically, side-by-side.

You can experiment with using different containers (uncomment lines 32-33) or different orientations on the plots (comment out line 43 and uncomment 44).

source: vertical_plot.py
**data_cube.py**

Allows isometric viewing of a 3-D data cube (downloads the necessary data, about 7.8 MB)

source: data_cube.py
This demo shows how Chaco and Traits can be used to easily build a data acquisition and visualization system.

Two frames are opened: one has the plot and allows configuration of various plot properties, and one which simulates controls for the hardware device from which the data is being acquired; in this case, it is a mockup random number generator whose mean and standard deviation can be controlled by the user.

source: data_stream.py
**scalar_image_function_inspector.py**

Renders a colormapped image of a scalar value field, and a cross section chosen by a line interactor.

source: scalar_image_function_inspector.py
This plot displays the audio spectrum from the microphone.

source: spectrum.py
cmap_image_plot.py

Draws a colormapped image plot.

source: cmap_image_plot.py
cmap_image_select.py

Draws a colormapped image plot. Selecting colors in the spectrum on the right highlights the corresponding colors in the color map.

source: cmap_image_select.py
cmap_scatter.py

Draws a colormapped scatterplot of some random data. Selection works the same as in cmap_image_select.py.

source: cmap_scatter.py
contour_cmap_plot.py

Renders some contoured and colormapped images of a scalar value field.

source: contour_cmap_plot.py
contour_plot.py

Draws an contour polygon plot with a contour line plot on top.

source: contour_plot.py
grid_container.py

Draws several overlapping line plots.

source: grid_container.py
**grid_container_aspect_ratio**

Similar to grid_container.py, but demonstrates Chaco’s capability to used a fixed screen space aspect ratio for plot components.

source: grid_container_aspect_ratio.py
image_from_file.py

Loads and saves RGB images from disk.

source: image_from_file.py
**image_inspector.py**

Demonstrates the ImageInspectorTool and overlay on a colormapped image plot. The underlying plot is similar to the one in cmap_image_plot.py.

source: image_inspector.py
image_plot.py

Draws a simple RGB image

source: image_plot.py
inset_plot.py

A modification of line_plot1.py that shows the second plot as a subwindow of the first. You can pan and zoom the second plot just like the first, and you can move it around by right-clicking and dragging in the smaller plot.

source: inset_plot.py
line_drawing.py

Demonstrates using a line segment drawing tool on top of the scatter plot from simple_scatter.py.

source: line_drawing.py
line_plot1.py

Draws some x-y line and scatter plots.

source: line_plot1.py
line_plot_hold.py

Demonstrates the different ‘hold’ styles of LinePlot.

source: line_plot_hold.py
log_plot.py

Draws some x-y log plots. (No Tools).

source: log_plot.py

nans_plot.py

This plot displays chaco’s ability to handle data interlaced with NaNs.

source: nans_plot.py
**polygon_plot_demo.py**

Draws some different polygons.

source: polygon_plot_demo.py
**polygon_move.py**

Shares same basic interactions as polygon_plot.py, but adds a new one: right-click and drag to move a polygon around.

source: polygon_move.py
regression.py

Demonstrates the Regression Selection tool.

Hold down the left mouse button to use the mouse to draw a selection region around some points, and a line fit is drawn through the center of the points. The parameters of the line are displayed at the bottom of the plot region. You can do this repeatedly to draw different regions.

source: regression.py
**scatter.py**

Draws a simple scatterplot of a set of random points.

source: scatter.py
scatter_inspector.py

Example of using tooltips on Chaco plots.

source: scatter_inspector.py
scatter_select.py

Draws a simple scatterplot of random data. The only interaction available is the lasso selector, which allows you to circle a set of points. Upon completion of the lasso operation, the indices of the selected points are printed to the console.

source: scatter_select.py
console output:

New selection:
[789 799 819 830 835 836 851 867 892 901 902 909 913 924 929
 931 933 938 956 971 972 975 976 996 999 1011 1014 1016 1021 1030
 1045 1049 1058 1061 1073 1086 1087 1088]

scrollbar.py

Draws some x-y line and scatter plots.

source: scrollbar.py
**tabbed_plots.py**

Draws some x-y line and scatter plots.

source: tabbed_plots.py
traits_editor.py

This example creates a simple 1-D function examiner, illustrating the use of ChacoPlotEditors for displaying simple plot relations, as well as Traits UI integration. Any 1-D numpy/scipy.special function works in the function text box.

source: traits_editor.py
**zoomable_colorbar.py**

Draws a colormapped scatterplot of some random data. Interactions on the plot are the same as for simple_line.py, and additionally, pan and zoom are available on the colorbar. Left-click pans the colorbar’s data region. Right-click-drag selects a zoom range. Mousewheel up and down zoom in and out on the data bounds of the color bar.

*source: zoomable_colorbar.py*
The main executable file for the zoom_plot demo.

Right-click and drag on the upper plot to select a region to view in detail in the lower plot. The selected region can be moved around by dragging, or resized by clicking on one of its edges and dragging.

source: zoomed_plot
1.4 Other Chaco resources

1.4.1 Chaco Gallery

Examples of what can be done with Chaco is available in the Chaco gallery.

1.4.2 Further Reading and resources

You can also learn more about Chaco:

- running the tutorials included with the Chaco package,
- following demos of Chaco given during webinars Enthought to EPD subscribers,
- reading seminar slides posted on conference websites,
- reading about the API from the developer guide.
- following the Chaco mailing list: chaco-users@enthought.com
Built-in tutorials

For more details on how to use Chaco to embed powerful plotting functionality inside applications, refer to the Tutorials, webinars, and examples page. In particular, some tutorial examples were added into the examples/tutorials/scipy2008/ directory. These examples are numbered and introduce concepts one at a time, going from a simple line plot to building a custom overlay with its own trait editor and reusing an existing tool from the built-in set of tools. You can browse them on the GitHub repository at: https://github.com/enthought/chaco/tree/master/examples/tutorials . Finally, it is recommended to explore the examples (Annotated Examples section) as they are regularly updated to reflect the most recent changes and recommended ways to use Chaco.

Enthought webinars

The video webinars given in as part of the Enthought webinar series cover building interactive plotting using Chaco. If you are an EPD user, you can find the video, the slides, and the demo code for each webinar covering Chaco.

• The first one (April 2010) demoses how to use Chaco as your plotting tool (https://www.enthought.com/repo/epd/webinars/2010-04InteractiveChaco/).

• The seconds (October 2010) illustrates how to building interactive 2D visualization (see https://www.enthought.com/repo/epd/webinars/2010-10Building2DInteractiveVisualizations/).

Presentations

There have been several presentations on Chaco at previous PyCon and SciPy conferences:


• Follow the presentation of Chaco at the PyCon 2007 at http://code.enthought.com/projects/files/chaco_pycon07/.

• SciPy 2008 Tutorial slides (pdf)

Developers references and API Docs

For developers and architects,

• more details about the current architecture and API can be found in the Programmer’s Reference,

• the API for Chaco 3.0 (in ETS 3.0) can be found at http://code.enthought.com/projects/files/ETS3_API/enthought.chaco.html.

• the API for Chaco2 (in ETS 2.7.1) can be found at http://code.enthought.com/projects/files/ets_api/enthought.chaco2.html.

1.5 Programmer’s Reference

1.5.1 Architecture Overview

Note: This is an overview of not just Chaco, but also Kiva and Enable.
Core Ideas

The Chaco toolkit is defined by a few core architectural ideas:

- **Plots are compositions of visual components**
  Everything you see in a plot is some sort of graphical widget, with position, shape, and appearance attributes, and with an opportunity to respond to events.

- **Separation between data and screen space**
  Although everything in a plot eventually ends up rendering into a common visual area, there are aspects of the plot which are intrinsically screen-space, and some which are fundamentally data-space. Preserving the distinction between these two domains allows us to think about visualizations in a structured way.

- **Modular design and extensible classes**
  Chaco is meant to be used for writing tools and applications, and code reuse and good class design are important. We use the math behind the data and visualizations to give us architectural direction and conceptual modularity. The Traits framework allows us to use events to couple disjoint components at another level of modularity.
  Also, rather than building super-flexible core objects with myriad configuration attributes, Chaco’s classes are written with subclassing in mind. While they are certainly configurable, the classes themselves are written in a modular way so that subclasses can easily customize particular aspects of a visual component’s appearance or a tool’s behavior.

The Relationship Between Chaco, Enable, and Kiva

Chaco, Enable, and Kiva are three packages in the Enthought Tool Suite. They have been there for a long time now, since almost the beginning of Enthought as a company. Enthought has delivered many applications using these toolkits. The Kiva and Enable packages are bundled together in the “Enable” project.

**Kiva**

Kiva is a 2-D vector drawing library for Python. It serves a purpose similar to Cairo. It allows us to compose vector graphics for display on the screen or for saving to a variety of vector and image file formats. To use Kiva, a program instantiates a Kiva GraphicsContext object of an appropriate type, and then makes drawing calls on it like gc.draw_image(), gc.line_to(), and gc.show_text(). Kiva integrates with windowing toolkits like wxWindows and Qt, and it has an OpenGL backend as well. For wxPython and Qt, Kiva actually performs a high-quality, fast software rasterization using the Anti-Grain Geometry (AGG) library. For OpenGL, Kiva has a python extension that makes native OpenGL calls from C++.

Kiva provides a GraphicsContext for drawing onto the screen or saving out to disk, but it provides no mechanism for user input and control. For this “control” layer, it would be convenient to have to write only one set of event callbacks
Chaco Documentation, Release 4.3.0

or handlers for all the platforms we support, and all the toolkits on each platform. The Enable package provides this layer. It insulates all the rendering and event handling code in Chaco from the minutiae of each GUI toolkit. Additionally, and to some extent more importantly, Enable defines the concept of “components” and “containers” that form the foundation of Chaco’s architecture. In the Enable model, the top-most Window object is responsible for dispatching events and drawing a single component. Usually, this component is a container with other containers and components inside it. The container can perform layout on its internal components, and it controls how events are subsequently dispatched to its set of components.

Enable

Almost every graphical component in Chaco is an instance of an Enable component or container. We’re currently trying to push more of the layout system (implemented as the various different kinds of Chaco plot containers) down into Enable, but as things currently stand, you have to use Chaco containers if you want to get layout. The general trend has been that we implement some nifty new thing in Chaco, and then realize that it is a more general tool or overlay that will be useful for other non-plotting visual applications. We then move it into Enable, and if there are plotting-specific aspects of it, we will create an appropriate subclass in Chaco to encapsulate that behavior.

The sorts of applications that can and should be done at the Enable level include things like a visual programming canvas or a vector drawing tool. There is nothing at the Enable level that understands the concept of mapping between a data space to screen space and vice versa. Although there has been some debate about the incorporating rudimentary mapping into Enable, for the time being, if you want some kind of canvas-like thing to model more than just pixel space on the screen, implement it using the mechanisms in Chaco.

The way that Enable hooks up to the underlying GUI toolkit system is via an enable.Window object. Each toolkit has its own implementation of this object, and they all subclass from enable.AbstractWindow. They usually contain an instance of the GUI toolkit’s specific window object, whether it’s a wx.Window or Qt.QWidget or pyglet.window.Window. This instance is created upon initialization of the enable.Window and stored as the control attribute on the Enable window. From the perspective of the GUI toolkit, an opaque widget gets created and stuck inside a parent control (or dialog or frame or window). This instance serves as a proxy between the GUI toolkit and the world of Enable. When the user clicks inside the widget area, the control widget calls a method on the enable.Window object, which then in turn can dispatch the event down the stack of Enable containers and components. When the system tells the widget to draw itself (e.g., as the result of a PAINT or EXPOSE event from the OS), the enable.Window is responsible for creating an appropriate Kiva GraphicsContext (GC), then passing it down through the object hierarchy so that everyone gets a chance to draw. After all the components have drawn onto the GC, for the AGG-based bitmap backends, the enable.Window object is responsible for blitting the rastered off-screen buffer of the GC into the actual widget’s space on the screen. (For Kiva’s OpenGL backend, there is no final blit, since calls to the GC render in immediate mode in the window’s active OpenGL context, but the idea is the same, and the enable.Window object does perform initialization on the GL GraphicsContext.)

Some of the advantages to using Enable are that it makes mouse and key events from disparate windowing systems all share the same kind of signature, and be accessible via the same name. So, if you write bare wxPython and handle a key_pressed event in wx, this might generate a value of wx.WXK_BACK. Using Enable, you would just get a “key” back and its value would be the string “Backspace”, and this would hold true on Qt4 and Pyglet. Almost all of the event handling and rendering code in Chaco is identical under all of the backends; there are very few backend-specific changes that need to be handled at the Chaco level.

The enable.Window object has a reference to a single top-level graphical component (which includes containers, since they are subclasses of component). Whenever it gets user input events, it recursively dispatches all the way down the potentially-nested stack of components. Whenever a components wants to signal that it needs to be redrawn, it calls self.request_redraw(), which ultimately reaches the enable.Window, which can then make sure it schedules a PAINT event with the OS. The nice thing about having the enable.Window object between the GUI toolkits and our apps, and sitting at the very top of event dispatch, is that we can easily interject new kinds of events; this is precisely what we did to enable all of our tools to work with Multitouch.

The basic things to remember about Enable are that:
• Any place that your GUI toolkit allows you stick a generic widget, you can stick an Enable component (and this extends to Chaco components, as well). Dave Morrill had a neat demonstration of this by embedding small Chaco plots as cells in a wx Table control.

• If you have some new GUI toolkit, and you want to provide an Enable backend for it, all you have to do is implement a new Window class for that backend. You also need to make sure that Kiva can actually create a GraphicsContext for that toolkit. Once the kiva.gl branch is committed to the trunk, Kiva will be able to render into any GL context. So if your newfangled unsupported GUI toolkit has a GLWindow type of thing, then you will be able to use Kiva, Enable, and Chaco inside it. This is a pretty major improvement to interoperability, if only because users now don’t have to download and install wxPython just to play with Chaco.

Chaco

Note: This section provides an overview of the relationships between these classes, and illustrates some sample usages. For a more detailed list of the class hierarchy, please see Commonly Used Modules and Classes.

At the highest level, Chaco consists of:

• Visual components that render to screen or an output device (e.g., LinePlot, ScatterPlot, PlotGrid, PlotAxis, Legend)

• Data handling classes that wrap input data, interface with application-specific data sources, and transform coordinates between data and screen space (e.g., ArrayDataSource, GridDataSource, LinearMapper)

• Tools that handle keyboard or mouse events and modify other components (e.g., PanTool, ZoomTool, ScatterInspector)

Every Chaco plot is composed of these elements. One can think of them as comprising a “display pipeline”, although the components form more of a graph.

For example, a simple scatter plot will have:

• Two ArrayDataSource objects, one for the array of X data and one for the Y data

• Two DataRange1D ranges, one for the X axis and one for the Y axis. If we want the ranges to automatically compute the bounds of the dataset, then they need a reference to the an ArrayDataSource.

• Two independent LinearMapper mappers, one for X axis and one for the Y axis. The mappers convert from screen space to data space and vice versa, so they need a reference to the DataRange1D objects so they know the data space extents.

• A ScatterPlot renderer, that has a reference to two mappers, as well as an index and a value ArrayDataSource.

This creates only the renderer that draws scatter markers in some region of screen space. This does not create an X-axis, a Y-axis, or horizontal and vertical grids. These other visuals are embodied as separate, distinct components: axes are drawn by the PlotAxis component, and grids are drawn by the PlotGrid component. Both of these overlays require a mapper in order to know where on the screen they should draw.
1.5.2 Commonly Used Modules and Classes

Base Classes

Plot Component

All visual components in Chaco subclass from PlotComponent. It defines all of the common visual attributes like background color, border styles and color, and whether the component is visible. (Actually, most of these visual attributes are inherited from the Enable drawing framework.) More importantly, it provides the base behaviors for participating in layout, handling event dispatch to tools and overlays, and drawing various layers in the correct order. Subclasses almost never need to override or customize these base behaviors, but if they do, there are several easy extension points.

PlotComponent is a subclass of Enable Component. It has its own default drawing order. It redefines the inherited traits draw_order and draw_layer, but it doesn’t define any new traits. Therefore, you may need to refer to the API documentation for Enable Component, even when you have subclassed Chaco PlotComponent.

If you subclass PlotComponent, you need to implement do_layout(), if you want to size the component correctly.

Data Objects

Data Source

A data source is a wrapper object for the actual data that it will be handling. It provides methods for retrieving data, estimating a size of the dataset, indications about the dimensionality of the data, a place for metadata (such as selections and annotations), and events that fire when the data gets changed. There are two primary reasons for a data source class:

- It provides a way for different plotting objects to reference the same data.
- It defines the interface for embedding Chaco into an existing application. In most cases, the standard ArrayDataSource will suffice.

   Interface: AbstractDataSource

   Subclasses: ArrayDataSource, MultiArrayDataSource, PointDataSource, GridDataSource, ImageData

Data Range

A data range expresses bounds on data space of some dimensionality. The simplest data range is just a set of two scalars representing (low, high) bounds in 1-D. One of the important aspects of data ranges is that their bounds can be set to auto, which means that they automatically scale to fit their associated datasources. (Each data source can be associated with multiple ranges, and each data range can be associated with multiple data sources.)

   Interface: AbstractDataRange

   Subclasses: BaseDataRange, DataRange1D, DataRange2D

Data Source

A data source is an object that supplies data to Chaco. For the most part, a data source looks like an array of values, with an optional mask and metadata.
The metadata trait attribute is a dictionary where you can stick stuff for other tools to find, without inserting it in the actual data.

Events that are fired on data sources are:

- data_changed
- bounds_changed
- metadata_changed

Mapper

Mappers perform the job of mapping a data space region to screen space, and vice versa. Bounds on mappers are set by data range objects.

Containers

PlotContainer

PlotContainer is Chaco’s way of handling layout. Because it logically partitions the screen space, it also serves as a way for efficient event dispatch. It is very similar to sizers or layout grids in GUI toolkits like WX. Containers are subclasses of PlotComponent, thus allowing them to be nested. BasePlotContainer implements the logic to correctly render and dispatch events to sub-components, while its subclasses implement the different layout calculations.

A container gets the preferred size from its components, and tries to allocate space for them. Non-resizeable components get their required size; whatever is left over is divided among the resizeable components.

Chaco currently has three types of containers, described in the following sections.

Renderers

Plot renderers are the classes that actually draw a type of plot.

- BarPlot
- Base2DPlot
  - ContourLinePlot
  - ContourPolyPlot
  - ImagePlot: displays an image file, or color-maps scalar data to make an image
- CMapImagePlot

- BaseXYPlot: This class is often emulated by writers of other plot renderers, but renderers don’t need to be structured this way. By convention, many have a hittest() method. They do need to implement map_screen(), map_data(), and map_index() from AbstractPlotRenderer.

  - LinePlot
    * ErrorBarPlot
  - PolygonPlot
    * FilledLinePlot
  - ScatterPlot
    * ColormappedScatterPlot
  - ColorBar

- PolarLineRenderer: NOTE: doesn’t play well with others

You can use these classes to compose more interesting plots.

The module chaco.plot_factory contains various convenience functions for creating plots, which simplify the set-up.

The chaco.plot.Plot class (called “capital P Plot” when speaking) represents what the user usually thinks of as a “plot”: a set of data, renderers, and axes in a single screen region. It is a subclass of DataView.

**Tools**

Tools attach to a component, which gives events to the tool.

All tools subclass from Enable’s BaseTool, which is in turn an Enable Interactor. Do not try to make tools that draw: use an overlay for that.

Some tool subclasses exist in both Enable and Chaco, because they were created first in Chaco, and then moved into Enable.

**Interface: BaseTool Subclasses:**

- BroadcasterTool: Keeps a list of other tools, and broadcasts events it receives to all those tools.
- DataPrinter: Prints the data-space position of the point under the cursor.
- enable.tools.api.DragTool: Enable base class for tools that do dragging.
  - MoveTool
  - ResizeTool
  - ViewportPanTool
- chaco.tools.api.DragTool: Chaco base class for tools that do dragging.
  - BaseCursorTool
    * CursorTool1D
    * CursorTool2D
  - DataLabelTool
  - DragZoom
- LegendTool
- MoveTool
- DrawPointsTool
- HighlightTool
- HoverTool
- ImageInspectorTool
- LineInspector
- PanTool
  - TrackingPanTool
- PointMarker
- SaveTool
- SelectTool
  - ScatterInspector
  - SelectableLegend
- enable.tools.api.TraitsTool
- chaco.tools.api.TraitsTool

DragTool is a base class for tools that do dragging.

Other tools do things like panning, moving, highlighting, line segments, range selection, drag zoom, move data labels, scatter inspection, Traits UI.
Overlays

Miscellaneous

1.5.3 Data Sources

AbstractDataSource
ArrayDataSource
MultiArrayDataSource
PointDataSource
GridDataSource
ImageData

1.5.4 Data Ranges

AbstractDataRange
BaseDataRange
DataRange1D
DataRange2D

1.5.5 Mappers

AbstractMapper
Base1DMapper
LinearMapper
LogMapper
GridMapper
ColorMapper
ColorMapTemplate
TransformColorMapper

1.5.6 Containers

BasePlotContainer
OverlayPlotContainer
HPlotContainer
VPlotContainer
GridPlotContainer

1.6 Tech Notes

This section compiles some of the more detailed, architecture-level notes and discussions from the mailing list. Much of the information here will eventually find its way into the User Manual or the Reference Manual.
1.6.1 About the Chaco Scales package

In the summer of 2007, I spent a few weeks working through the axis ticking and labelling problem. The basic goal was that I wanted to create a flexible ticking system that would produce nicely-spaced axis labels for arbitrary sets of labels and arbitrary intervals. The chaco2.scales package is the result of this effort. It is an entirely standalone package that does not import from any other Enthought package (not even traits!), and the idea was that it could be used in other plotting packages as well.

The overall idea is that you create a ScaleSystem consisting of various Scales. When the ScaleSystem is presented with a data range (low, high) and a screen space amount, it searches through its list of scales for the scale that produces the “nicest” set of labels. It takes into account whitespace, the formatted size of labels produced by each scale in the ScaleSystem, etc. So, the basic numerical Scales defined in scales.py are:

- **FixedScale**: Simple scale with a fixed interval; places ticks at multiples of the resolution.
- **DefaultScale**: Scale that tries to place ticks at 1, 2, 5, and 10 so that ticks don’t “pop” or suddenly jump when the resolution changes (when zooming).
- **LogScale**: Dynamic scale that only produces ticks and labels that work well when doing logarithmic plots.

By comparison, the default ticking logic in DefaultTickGenerator (in ticks.py) is basically just the DefaultScale. (This is currently the default tick generator used by PlotAxis.)

In time_scale.py, I define an additional scale, the TimeScale. TimeScale not only handles time-oriented data using units of uniform interval (microseconds up to days and weeks), it also handles non-uniform calendar units like “day of the month” and “month of the year”. So, you can tell Chaco to generate ticks on the 1st of every month, and it will give you non-uniformly spaced tick and grid lines.

The scale system mechanism is configurable, so although all of the examples use the CalendarScaleSystem, you don’t have to use it. In fact, if you look at CalendarScaleSystem.__init__, it just initializes its list of scales with **HMSScales** + **MDYScales**:

```
HMSScales = [TimeScale(microseconds=1), TimeScale(milliseconds=1)] + \
[TimeScale(seconds=dt) for dt in (1, 5, 15, 30)] + \
[TimeScale(minutes=dt) for dt in (1, 5, 15, 30)] + \
[TimeScale(hours=dt) for dt in (1, 2, 3, 4, 6, 12, 24)]
```

```
MDYScales = [TimeScale(day_of_month=range(1,31,3)),
             TimeScale(day_of_month=(1,8,15,22)),
             TimeScale(day_of_month=(1,15)),
             TimeScale(month_of_year=range(1,13)),
             TimeScale(month_of_year=range(1,13,3)),
             TimeScale(month_of_year=(1,7)),
             TimeScale(month_of_year=(1,))]
```

So, if you wanted to create your own ScaleSystem with days, weeks, and whatnot, you could do:

```
ExtendedScales = HMSScales + [TimeScale(days=n) for n in (1,7,14,28)]
MyScaleSystem = CalendarScaleSystem(*ExtendedScales)
```

To use the Scales package in your Chaco plots, just import `PlotAxis` from `chaco2.scales_axis` instead of `chaco2.axis`. You will still need to create a `ScalesTickGenerator` and pass it in. The financial_plot_dates.py demo is a good example of how to do this.
• search