Graphite
A portable graphing package

Michelle Mills Strout and Joseph J. Strout

November 15, 1999

Abstract

We discuss a Python graphing package called Graphite. Graphite enables the easy creation of 2D and 3D scientific, engineering, and business graphs. At the time of this writing Graphite handles line/point plots and bar plots. Some graph types to be included are function plots, polar plots, parametric plots, pie plots, density plots, image plots, and mesh plots. Graphite produces output for formats such as PostScript, PDF, GIF, QuickDraw, Tk, and Windows by using the PIDDLE backend. This paper gives an introduction to Graphite.

1 Introduction

A significant subset of Python users are scientists and engineers, who make use of Python's numeric processing capabilities. Python has built-in support for a variety of numeric types, including integer, real, and complex numbers of various sizes. There is also a contributed module called Numeric, developed originally at MIT and now supported by LLNL, which adds an efficient datatype for multidimensional matrices, and associated functions and operations for manipulating such data. Additional modules provide extra support for applications such as Fourier transforms and linear algebra. Together, these packages offer a powerful alternative to such commercial packages as Matlab. Python has won many converts in this arena because of its more sophisticated language, its large collection of pre-built modules, and its open-source nature.

However, Python's utility as a number-crunching environment comes with one serious drawback: there is no good, portable graphing module. The user community is fractionated by a large number of incompatible graphing utilities, none of them well-integrated with Python, none available on all platforms, and most lacking in functionality. This is a severe limitation when compared to a package such as Matlab, which has powerful graphing capabilities and is portable to all platforms.

A graphing package should satisfy the following goals:

Print-quality graphing The package should be able to produce graphs up to the standards of scientific journals, magazines, newspapers, and other professional media. Graph types should include all common 2D plots, and possibly 3D graphs as well.

Platform independence The package should run identically on any system which runs Python.

Output options The package should be able to generate graphs to the screen, for interactive data visualization, as well as to a variety of common graphics formats such as Postscript, GIF, and PICT.

Flexibility The user should ultimately have control over nearly every detail of a graph, such as line type, tick type and positioning, axis labelling, etc. In addition, it should be possible to define new element types by deriving from existing graph elements, to use one graph as an inset in another, and so on.

Easy to use A common problem with powerful graphing packages is their complexity, which can overwhelm new users. The package should minimize this difficulty as much as possible, through the use of GUI option-setting tools, preset but adjustable graph stationery, and clean design.

Good language integration The package should be integrated with Python in such a way that the user can take full advantage of the language features. That is, graph and graph elements should be Python objects, with properties available in the same way as other objects; and the package should accept common Python types (including functions) as parameters.

Open source Like all of Python, the graphing package should be open source and freely available. This allows end-users to readily inspect and extend the package, or to fix bugs if any are found. Extending the package by adding new graph types should be made especially feasible.

Graph Reusability Users should be able to save graph formats so they can apply the same format to different data. This ability will allow for sharing of common formats with others.

Graphite attempts to meet all of the above goals. *Print-quality graphing* is made possible by the fact that Graphite can generate Postscript or PDF output. These output formats are a result of using PIDDLE [3] which also gives Graphite *Platform independence.* The PIDDLE interface specifies a set of interactive functions which could at some point be used for interactive graph manipulation. Every property of the Graph object can be changed by the user to providing *Flexibility.* In order to make Graphite *Easy-to-use* there is a small set of utility functions which automate such things as reading data from a delimited file, selecting columns from that data, and generating output to a PIDDLE canvas. Also, reasonable defaults allow for quick graphing of data without setting any of the Graph properties. For *Good language integration* Graphite uses Python as its base language. Graphite is *Open source* and freely available at http://www.strout.net/python/graphite. *Graph reusability* is only somewhat possible at this point, but more support for this is planned for future releases.

First we discuss more concrete Graphite goals (such as plot format types) and their status. Next, we step through the creation of an example graph, show how a user might change various options. Then we describe the software architecture of Graphite, and various design issues. Finally, we cover related and future work.

2 Graphite Goals and Status

Graphite uses an object-oriented design which allows new plot types (i.e. PlotFormats) to be added easily. As a result, it is expected that the types of plots available will continue to grow. At a mininum, we expect to implement at least the following types:

Point Plot A combination of scatter and line plot; each point in the dataset is plotted with a certain symbol, and connected to its neighbor with lines. The line may be omitted to obtain a traditional scatter plot; the symbols may also be omitted if lines are drawn. Optional error bars are available on all three axes; these may be specified asymmetrically or symmetrically. Line style and symbol style are configurable. Points may also be individually labelled. Lines may optionally be given a fill style, used to fill below the line, and a Z-thickness used in 3D plots.

Bar Plot Each point is represented as a bar; the Y value indicates the height of the bar, and the X and Z specify bar position. Bar style (color, hashing, width, etc.) are configurable. Graphs containing multiple bar plots combine their displays in one of several ways: by displaying bars side-by-side, by stacking, or by overlaying. Bars may be individually labelled at the top. Labels below the bars may be specified as tick mark labels, as with any other plot type. Possibly, one may also give a set of labels as the X coordinate data (but this notion is not fully worked out yet).

Polar Plot Each Y value specifies the radius of the data point and each X value specifies the angle.

Pie Plot Each Y value specifies the angular extent of a slice of the pie; Z values, if given, are used to "explode" a slice out from the center. Each slice may be individually styled and labelled. The pie plot also has global attributes specifying its position and extent, making it easy to put several pies into one chart.

Density Plot Requires three-dimensional data (e.g., Y as a function of X and Z). This plot is given a plane (defaulting to Z=0) on which to draw, and a palette function (several standard palettes will be provided); each point on the plane will be colored according to the corresponding Y value in the data. Additionally, isocontours can be located in the data and added to the plot.

Surface Plot Similar to a Density plot, except that the surface drawn is extended into the third dimension rather than being confined to a plane. The surface is drawn with hidden line removal (unless the fill style is transparent), and with optional lighting. The user may also specify a color function to be applied to each point in the surface, e.g., color as a function of Y. One may also add either a mesh (periodic grid lines which follow the surface) to enhance perception of the surface, or contour lines. The camera position and projection time (perspective or orthographic) are adjustable – true with any plot, but especially relevant to the Surface plot. We hope to also add support for surfaces defined by irregularly sampled points.

Image Plot Also similar to a Density plot, but more efficient at displaying large matrices of color values.

At the time of this writing only the Point Plot and Bar Plot graph types are implemented.

3 Examples

The general schema for creating graphs is as follows:

1. create a Graph
2. attach one or more Datasets
3. attach or configure one or more PlotFormats
4. add any overlays (extra labels, arrows, etc.)
5. call the Graph's draw() method

By using utility functions it is possible to create a graph very easily. The below example (1) creates a graph, (2) attaches a dataset which was stored in a file, and (5) calls the Graph g's draw method by using the utility function genOutput. The resulting graph is shown in figure 1. Notice that steps (3) and (4) above are optional because a default PointPlot format is used automatically, and because overlays are not necessary.

```
from graphite import *
g = Graph()
data = loadTable('karin.dat',' ')
g.datasets.append(Dataset(data))
genOutput(g,'PS')
```

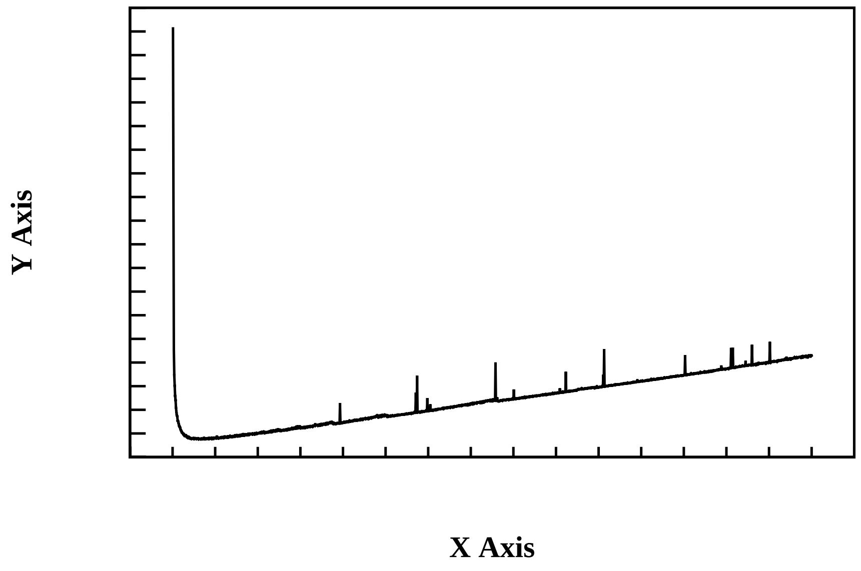

Figure 1: First attempt at graphing a Point Plot

After creating the very simple graph it is then possibe to export all of this graph's settings in a string. If this string is written to a file, we can then change all the properties of the graph. All of the graph properties can be printed to a file using the following code.

```
f = open('settings.py','w')
f.write(g.exportString('g'))
f.close()
```

That file can them be edited to make any changes to the graph which are desired. Below are some example properties as generated from exportString.

```
# list of PlotFormat objects which will be used in a round-robin fashion by the Datasets
g.formats = [PointPlot()]

# style used to draw the lines, or None if no lines are desired
g.formats[0].lineStyle = LineStyle(width=1,
color=Color(0.00,0.00,0.00), kind=SOLID)
```

By making the changes to various properties listed in the settings file and then executing the edited file, we were able to change figure 1 to figure 2.

```
f = open('settings.py')
```

```
commands = f.readlines()
f.close()
import string
commands = string.join(commands,'')
exec(commands)
genOutput(g,'PS',size=(400,250))
```

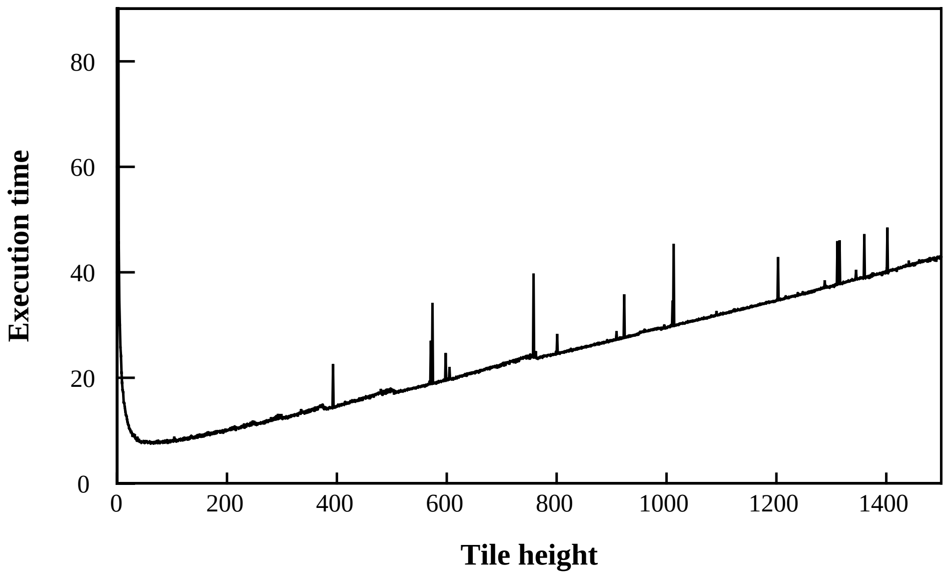

Figure 2: After editing the graph properties

Figure 3 and figure 4 show two more examples of graphs which can be generated with Graphite.

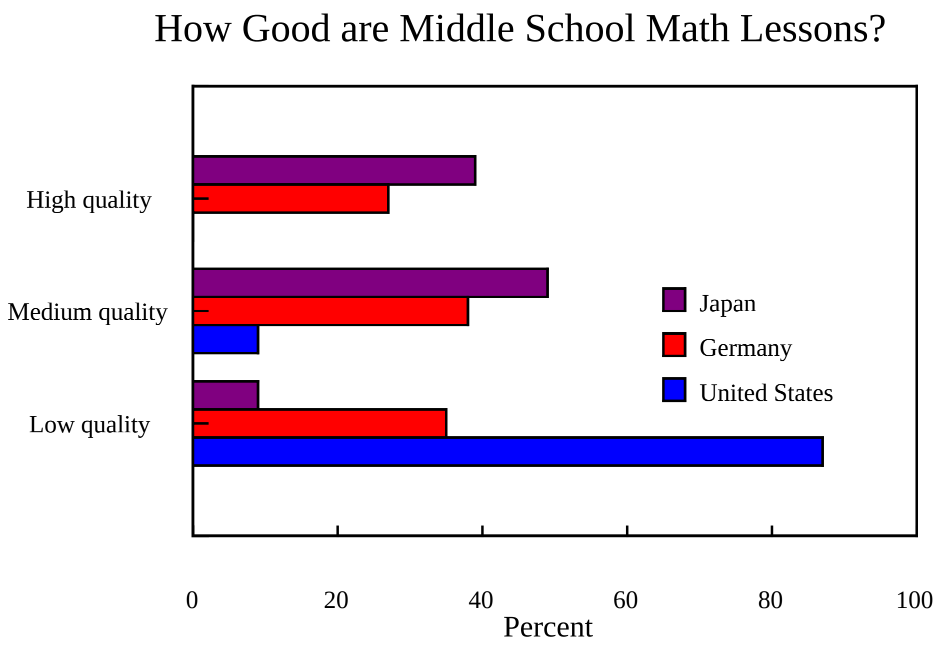

Figure 3: Bar chart example

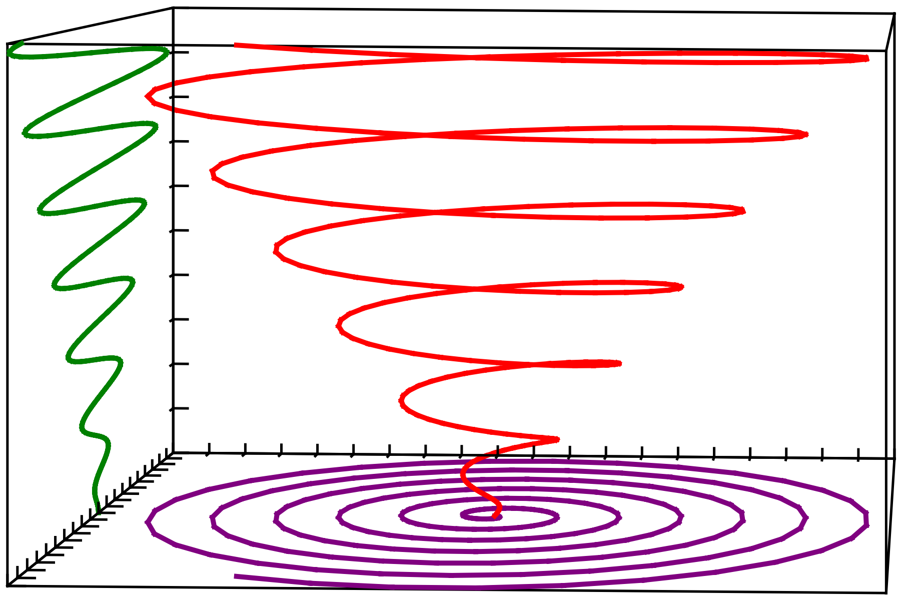

Figure 4: 3D Point plot example

4 Software Architecture

Graphite is divided into six Python modules whose uses-relationship is shown in the software architecture diagram in figure 5.

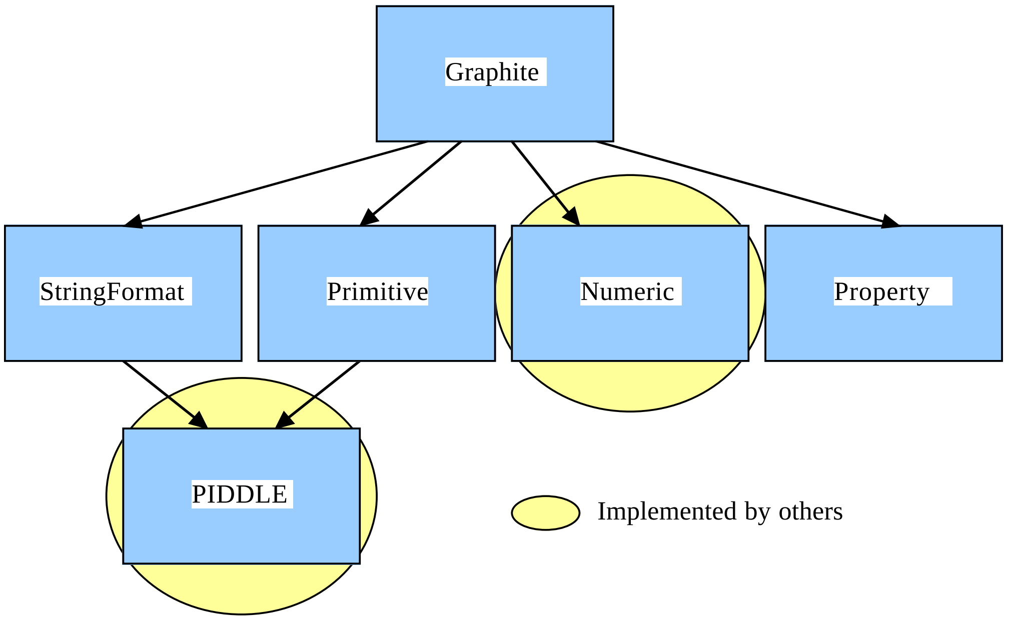

Figure 5: Graphite Software Architecture

PIDDLE [3] is a Python module for generating cross-platform two-dimensional graphics. It provides Graphite with a large number of backend formats such as PostScript, PDF, PNG, TK, QuickDraw (Mac), Windows, etc. All graph objects are described in terms of 3D primitives which are then translated into PIDDLE canvas drawing functions. Each Primitive subclass abstracts a 3D drawing element. Currently Graphite has Box, Line, Symbol and Text Primitive subclasses.

The *Property* module describes a base class which contains user-configurable properties. A property looks and acts just like an ordinary Python member variable, with two exceptions: a property may have a restricted type and range of acceptable values, and every property has an associated help string.

Numeric is an established Python module for doing numerical mathematics. Graphite uses the array manipulation provided by Numeric. The *StringFormat* module allows for character-by-character formatting of strings for things such as superscripting, subscripting, and greek letters.

The main classes of Graphite objects are shown in the class diagram (figure 6), using the notation of Design Patterns [2]. Detailed understanding of the diagram is not necessary for this discussion, but for the sake of completeness, a brief description of the notation is as follows. Classes are shown as boxes, with the name of the class in bold type. Important data members and functions are shown in the lower part of a box. Arrows connecting boxes indicate relationships between the classes. A filled circle at the head of an arrow indicates a one-to-many relationship. A diamond at the start of an arrow indicates a part-of (aggregate) relationship.

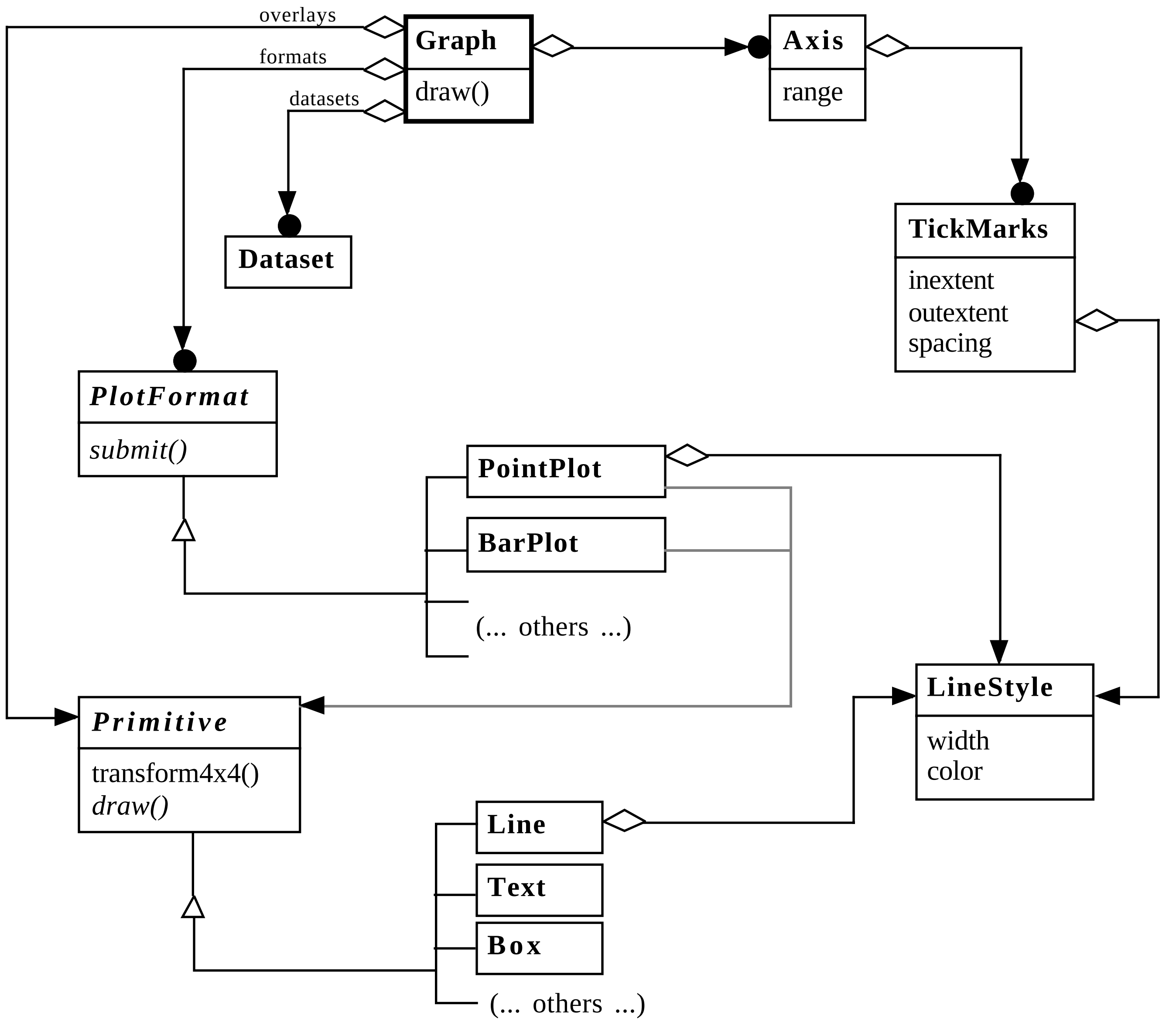

Figure 6: Class diagram for the Graph class

4.1 Coordinate Systems

One should note three separate coordinate systems used by Graphite. There are data coordinates, i.e., the space occupied by the raw data. These are transformed by the axes into view/frame coordinates, which range from 0 to 1 in X, Y, and Z. Finally, view coordinates get converted to device/screen coordinates. These distinctions are important because some positions must be specified in data coordinates, while others are specified in view or device coordinates.

4.2 Plot Format

PlotFormat is a general category which will include many subtypes, such as PointPlot, BarPlot, MeshPlot, etc. PlotFormats are attached to the graph in an order that corresponds to the order of the Datasets applied to the same graph. If there are fewer PlotFormats than Datasets, the PlotFormats will be cycled through as needed. Because PlotFormats and Datasets are attached separately, it is easy to change the format of a graph without changing the data, or vice versa.

A PlotFormat works by taking a Dataset given to it by the Graph, and returning a set of 3D drawing primitives. These are then combined with all the other primitives and rendered. To extend Graphite's capabilities with a new plot type, it should suffice to create a new subclass of PlotFormat; all details of data handling, rendering, and so on will be handled by other classes.

4.3 Dataset

A Dataset is an object which contains all the information needed to specify a single set of data points. It is a very simple object which is really little more than a container for numeric sequences. It has fields for the common uses of a sequence, e.g.: x, y, z, yerr, etc. At its purest level, a Dataset is prepared by assigning a sequence of numbers to each relevant field (e.g., for a scatter plot, you'd assign one sequence to x and a corresponding sequence to y). However, methods such as setXY make typical cases easier.

Each Dataset is associated (by position in the Graph) with one PlotFormat object. So to make a graph with two different data formats (e.g., two lines, one in thick red and the other in dashed green), there would be two Datasets and two PlotFormats. A Dataset is sometimes called a "data series" in other graphing packages.

In addition to storing numeric data, a Dataset may also store a function which will be sampled at graph time to obtain the numbers to plot. This takes advantage of Python's functional programming capabilities, and is often useful for generating a quick graph of a function without manual sampling.

4.4 Axis

An Axis defines the numeric range covered by one axis of the plot, as well as that axis' appearance: line style, TickMarks, and label. Note that one Axis can have many sets of TickMarks, each with a different size, style, and label; this is how Graphite implements major and minor tickmarks and grid lines. Each Axis can specify a list of drawing locations. There are constants for putting axes along the graph frame. Datasets are mapped to a set of 3 axes through an axis mapping. The list of axes mappings are associated with Datasets with the same mechanism used in assigning PlotFormats to Datasets (round-robin).

4.5 TickMarks

A TickMarks object defines a set of tick marks for one axis. Tickmarks may extend an adjustable distance (given in view coordinates) inside and outside the graph frame (including all the way across the graph, for making grid lines), and may have labels. By default, numeric labels will be generated based on the axis range.

In a 3D graph, each tick mark is actually two lines, extending in the two directions orthogonal to the axis. For example, tick marks on the X axis extend both in Y and in Z.

5 Related Work

There are many existing graphing solutions. We have found that these packages do not meet all of the goals set forth at the start of this paper. Some examples of similar graphing packages are Snow, GnuPlot, Mathematica and Matlab, SigmaPlot, existing python plot packages, etc.

There are also many other approaches to graphing in Python; many of these serve specialized purposes or platforms. A few general-purpose packages will be mentioned here.

Snow [1] is a 2D graphics API based on a custom C module; its author (David Ascher) has been assisting with the design of Graphite. GnuPlot is an open-source plotting package, written in C and ported to many platforms. The Python interface, however, depends on pipes and so is not fully portable; it also requires the use of a special command language. Mathematica, Matlab, and SigmaPlot are commercial packages which do extensive 2D and 3D graphics, but are not well integrated with Python.

Each of these packages has a sizable user base, and may be perfectly suitable for some applications. Graphite differs from them primarily in two ways. First, by generating output with PIDDLE, it gains both true platform independence and a wide variety of output types, from GIF to PDF to interactive windowing systems. Second, because it is written from scratch entirely in Python, there is no second language to learn; all graph elements are ordinary Python constructs and manipulated in standard Python. This may decrease learning time, and increase comfort, for users who are already using Python for other purposes.

6 Future Work

As indicated in the status section there is much work to be done. Future work falls under three main groupings. One, new plot formats need to be implemented. Two, features need to be added to the existing BarPlot and PointPlot formats. And finally, we would like to take advantage of the interactive aspects of PIDDLE to allow for editing graph properties.

7 Acknowledgements

We would like to thank David Ascher for his contributions to the initial design of Graphite.

References

[1] David Ascher. Snow. http://starship.python.net:9673/crew/da/Code/Snow.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[3] Joseph J. Strout. Piddle: A cross-platform drawing library. http://www.strout.net/python/piddle.