The Matplotlib User's Guide

John Hunter
and
Darren Dale

May 27, 2007

Contents

1 Introduction **5**
1.1 Migrating from MATLAB™ . . . 6

2 Installation and Setup **9**
2.1 Installing . . . 9
2.1.1 Quickstart on windows . . . 9
2.1.2 Package managers: (rpms, apt, fink) . . . 9
2.1.3 Compiling matplotlib . . . 10
2.1.4 Trial Run . . . 11
2.2 Backends . . . 11
2.3 Integrated development environments . . . 12
2.4 Interactive . . . 12
2.5 Numerix . . . 14
2.5.1 Choosing Numeric, numarray, or NumPy . . . 14
2.6 Customization using `matplotlibrc` . . . 15
2.6.1 RC file format . . . 15
2.6.2 Which rc file is used? . . . 15
2.6.3 Getting feedback from matplotlib . . . 16

3 The pylab interface **17**
3.1 Simple plots . . . 17
3.2 More on plot . . . 19
3.2.1 Multiple lines . . . 19
3.2.2 Controlling line properties . . . 20
3.3 Color arguments . . . 22
3.4 Loading and saving data . . . 22
3.4.1 Loading and saving ASCII data . . . 22
3.4.2 Loading and saving binary data . . . 23
3.4.3 Processing several data files . . . 24
3.5 axes and figures . . . 24
3.5.1 `figure` . . . 24
3.5.2 `subplot` . . . 25
3.5.3 `axes` . . . 27
3.6 Text . . . 29
3.6.1 Basic text commands . . . 29
3.6.2 Text properties . . . 29
3.6.3 Text layout . . . 30
3.6.4 mathtext . . . 32
3.6.5 Annotations . . . 33
3.7 Images . . . 36

3.7.1 Axes images . . . 36
3.7.2 Figure images . . . 38
3.7.3 Scaling and color mapping . . . 38
3.7.4 Image origin . . . 39
3.8 Bar charts, histograms and errorbar plots . . . 40
3.8.1 Broken bar charts . . . 40
3.9 Polar charts . . . 41
3.10 Pseudocolor and scatter plots . . . 43
3.11 Spectral analysis . . . 43
3.12 Axes properties . . . 45
3.13 Legends and tables . . . 45
3.14 Navigation . . . 45
3.14.1 Classic toolbar . . . 45
3.14.2 toolbar2 . . . 45
3.15 Event handling . . . 46
3.16 Customizing plot defaults . . . 48

4 Font finding and properties **49**

5 Collections **53**

6 Tick locators and formatters **55**
6.1 Tick locating . . . 55
6.2 Tick formatting . . . 56
6.3 Example 1: major and minor ticks . . . 56
6.4 Example 2: date ticking . . . 58

7 Interactive object picking **61**
7.1 Picking with a lasso tool . . . 63

8 Custom objects and units **67**

9 Cookbook **73**
9.1 Plot elements . . . 73
9.1.1 Horizontal or vertical lines/spans . . . 73
9.1.2 Fill the area between two curves . . . 73
9.1.3 Drawing true ellipses and circles . . . 73
9.2 Text . . . 74
9.2.1 Adding a ylabel on the right of the axes . . . 74
9.3 Data analysis . . . 75
9.3.1 Linear regression . . . 75
9.3.2 Polynomial regression . . . 76
9.4 Working with images . . . 77
9.4.1 Loading existing images into matplotlib . . . 77
9.4.2 Blending several axes images using alpha . . . 78
9.4.3 Creating a mosaic of images . . . 79
9.4.4 Defining your own colormap . . . 80
9.5 Output . . . 80
9.5.1 Printing to standard output . . . 80

10 Matplotlib API **81**
10.1 The matplotlib backends . . . 81
10.1.1 The renderer and graphics context . . . 82
10.1.2 The figure canvases . . . 83
10.2 The matplotlib Artists . . . 83
10.3 pylab interface internals . . . 83

A A sample `matplotlibrc` **89**

B mathtext symbols **95**

C matplotlib source code license **97**

Chapter 1

Introduction

matplotlib is a library for making 2D plots of arrays in python. Although it has its origins in emulating the MATLAB™ graphics commands, it does not require MATLAB™, and can be used in a pythonic, object oriented way. Although matplotlib is written primarily in pure python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so it; it should just work.

For years, I used to use MATLAB™ exclusively for data analysis and visualization. MATLAB™ excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed and EEG analysis application in MATLAB™. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB™ as a programming language, and decided to start over in python. python more than makes up for all of matlab's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc)
- Postscript output for inclusion with TEX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it.
- Making plots should be easy.

Finding no package that suited me just right, I did what any self-respecting python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB™'s plotting capabilities because that is something MATLAB™ does very well. This had the added advantage that many people have a lot of MATLAB™ experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by `matplotlib.pylab` which allow the user to create plots with code quite similar to MATLAB™ figure generating code. The *matplotlib frontend* or *matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on. This is an abstract interface that knows nothing about output. The *backends* are device dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device. Example backends: PS creates postscript hardcopy, SVG creates scalar vector graphics hardcopy, Agg

creates PNG output using the high quality antigrain library that ships with matplotlib - http://antigrain.com, GTK embeds matplotlib in a GTK application, GTKAgg uses the antigrain renderer to create a figure and embed it a GTK application, and so on for WX, Tkinter, FLTK. . . .

matplotlib is used by many people in many different contexts. Some people want to automatically generate postscript files to send to a printer or publishers. Others deploy matplotlib on a web application server to generate PNG output for inclusion in dynamically generated web pages. Some use matplotlib interactively from the python shell in Tkinter on windows. My primary use is to embed matplotlib in a GTK EEG application that runs on windows, linux and OS X.

Because there are so many ways people want to use a plotting library, there is a certain amount of complexity inherent in configuring the library so that it will work naturally the way you want it to. Before diving into these details, let's first explore matplotlib's simplicity by comparing a typical matplotlib script with its analog in MATLAB^{TM}.

— JDH

1.1 Migrating from MATLAB^{TM}

Using matplotlib should come naturally if you have ever plotted with MATLAB^{TM} and should be fairly straightforward if you haven't. Like all interpreted languages used for serious number crunching, python has an extension module for processing numeric arrays. NumPy comes with many MATLAB^{TM}-compatible analysis functions, which matplotlib extends. The example code below shows two complete scripts: on the left hand side is python with matplotlib, and on the right is MATLAB^{TM}.

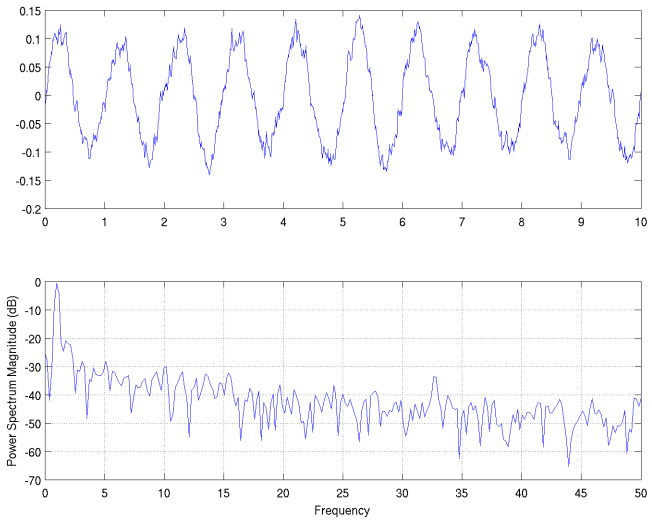

Figure 1.1: Colored noise signal and power spectrum generated with MATLAB^{TM} as shown in Listing 1.1. Compare with matplotlib in Figure 1.2.

Both scripts do the same thing: generate a white noise vector, convolve it with an exponential function, add it to a sine wave, plot the signal in one subplot and plot the power spectrum in another.

Listing 1.1: matplotlib and MATLAB^{TM}

```
# python                                  % matlab
from pylab import *                       % no import necessary

dt = 0.01                                 dt = 0.01;
```

```
t = arange(0,10,dt)
nse = randn(len(t))
r = exp(-t/0.05)

cnse = conv(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*sin(2*pi*t) + cnse

subplot(211)
plot(t,s)
subplot(212)
psd(s, 512, 1/dt)
```

```
t = [0:dt:10];
nse = randn(size(t));
r = exp(-t/0.05);

cnse = conv(nse, r)*dt;
cnse = cnse(1:length(t));
s = 0.1*sin(2*pi*t) + cnse;

subplot(211)
plot(t,s)
subplot(212)
psd(s, 512, 1/dt)
```

The major differences are 1) NumPy has a function for creating arrays (`arange` above) whereas MATLAB™ has the handy notation `[0:dt:10]`, 2) Python uses square brackets rather than parentheses for array indexing, and there are some small differences in how to do array lengths, sizes, and indexing. But the differences are minute compared to the similarities: 1) MATLAB™ and NumPy both do array processing and have a variety of functions that efficiently operate on arrays and scalars, 2) moderately sophisticated signal processing (white noise, convolution, power spectra) is achieved in only a few lines of clear code and 3) plots are simple, intuitive and attractive (compare Figures 1.1 and Figures 1.2).

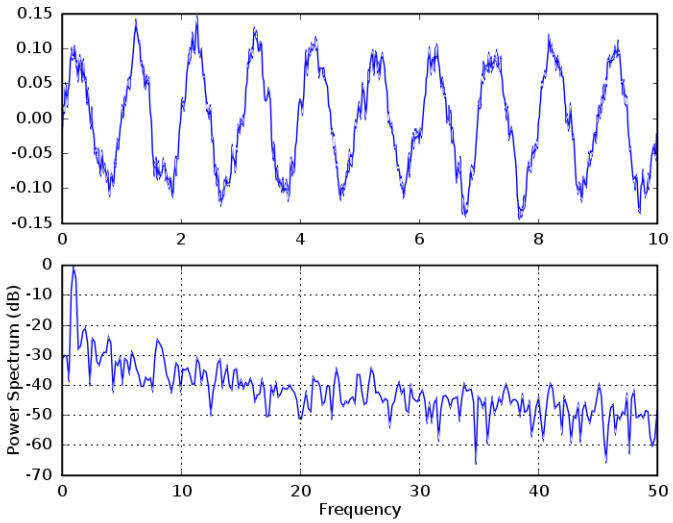

Figure 1.2: Colored noise signal and power spectrum generated with python matplotlib as shown in Listing 1.1. Compare with MATLAB™ in Figure 1.1. Note that the waveforms are not identical because they were generated from random signals!

Hopefully, this example will have instilled some confidence in those who have previously worked with MATLAB™ that migrating to Python is not too daunting a task. However, this guide will not attempt to serve as an introduction to Python itself, and therefore assumes you already have a rudimentary knowledge of the language. For users who are new to Python, we recommend getting accustomed to the language by experimenting with some of the tutorials at http://wiki.python.org/moin/BeginnersGuide/Programmers.

Finally, matplotlib does not intend to meet the needs of MATLAB™ users alone. Many matplotlib users previously worked with gnuplot, for example, and have influenced matplotlib's features based on their previous experience. Our goal is to provide a flexible, powerful library that is capable of easily producing beautiful plots for scientists and engineers who work with Python.

Chapter 2

Installation and Setup

2.1 Installing

Matplotlib is known to work on linux, unix, win32 and OS X platforms. This chapter will begin with basic installation instructions to help new users get going quickly. The suggested setup for matplotlib version 0.87.7 and later requires python 2.3 or later, NumPy 1.0 or later and freetype. To get the most out of matplotlib, you should also install IPython and at least one of the GUI toolkits. We suggest using the Tk GUI toolkit if you are just getting started.

2.1.1 Quickstart on windows

If you don't already have python installed, you may want to consider using the enthought edition of python, which includes everything you need to start plotting with matplotlib. Enthought's Python distribution also includes a lot of other goodies, like the wxPython GUI toolkit and SciPy - see http://www.enthought.com/python .

For standard Python installations, you should install NumPy before running the matplotlib installer. The windows installer (*.exe) on the download page contains everything else you need to get up and running. We highly recommend installing PyReadline and IPython as well, (see http://ipython.scipy.org).

There are many examples that are not included in the windows installer. They can be found at http://matplotlib.sourceforge.net/matplotlib_examples_0.87.7.zip .

2.1.2 Package managers: (rpms, apt, fink)

RPMS

To build all the backends on a binary linux distro such as redhat, you need to install a number of the devel libs (and whatever dependencies they require), I suggest

- matplotlib core: zlib, zlib-devel, libpng, libpng-devel, freetype, freetype-devel, freetype-utils
- gtk backend: gtk2-devel, gtk+-devel, pygtk2, glib-devel, pygtk2-devel, gnome-libs-devel, pygtk2-libglade
- tk backend: tcl, tk, tkinter
- wx, wxagg backend. The wxpython rpms.

Debian and Ubuntu

Vittorio Palmisano <redclay@email.it> maintails the debian packages at http://mentors.debian.net. He provides the following instructions

- add these lines to your /etc/apt/sources.list:

```
deb http://anakonda.altervista.org/debian packages/
deb-src http://anakonda.altervista.org/debian sources/
```

- then run

```
> apt-get update
> apt-get install python-matplotlib python-matplotlib-doc
```

Alternatively, Andrew Straw maintains an Apt Repository of scientific Python packages:

- add these lines to your /etc/apt/sources.list:

```
deb http://debs.astraw.com/ dapper/
deb-src http://debs.astraw.com/ dapper/
```

fink

fink users should use Jeffrey Whitaker's matplotlib fink package, which includes support for the GTK, Tk, and WX GUI toolkits (see http://fink.sourceforge.net/pdb/package.php/matplotlib-py23 or http://fink.sourceforge.net/pdb/package.php/matplotlib-py24, or http://fink.sourceforge.net/pdb/package.php/matplotlib-py25).

2.1.3 Compiling matplotlib

You will need to have recent versions of freetype (>= 2.1.7), libpng and zlib installed on your system. If you are using a package manager, also make sure the devel versions of these packages are also installed (eg freetype-devel).

If you want to use a GUI backend, you will need either Tkinter, pygtk, PyQt, PyQt4 or wxpython installed on your system, from src or from a package manager, including the devel packages. You can choose which backends to enable by setting the flags in setup.py, but the 'auto' flags will work in most cases, as matplotlib tries to find a GUI and build the backend accordingly. If you know you don't want a particular backend or extension, you can set that flag to *False*.

Most users will want to keep the setup.py default `BUILD_AGG = 1`. Exceptions to this are if you know you don't need a GUI (eg a web server) or you only want to produce vector graphics like postscript, svg or pdf.

If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit setupext.py an add the base dirs to the 'basedir' dictionary entry for your sys.platform. Eg, if the header to some required library is in `/some/path/include/somheader.h`, put `/some/path` in the basedir list for your platform.

Note that if you install matplotlib anywhere other than the default location, you will need to set the `MATPLOTLIBDATA` environment variable to point to the install base dir. Eg, if you install matplotlib with `python setup.py build -prefix=/home/jdhunter` then set `MATPLOTLIBDATA` to `/home/jdhunter/share/matplotlib`.

OS X

All of the backends run on OS X. fink users consult the fink discussion in section 2.1.2. Another option is http://www.stecf.org/macosxscisoft which packages many scientific packages for python on OS X, including matplotlib, although it is designed for astronomical analysis.

If you want to compile matplotlib yourself on OS X, make sure you read the compiling instructions above. You will need to install freetype2, libpng and zlib via fink or from src. You will also need the base libraries for a given backend. Eg, if you want to run TkAgg, you will need a python with Tkinter; if you want to use WxAgg, install wxpython. See Section 2.2 for a more comprehensive discussion of the various backend requirements. Edit `setup.py` to configure the backends you want to build as described above.

Note when running a GUI backend in OS X, you should launch your programs with pythonw rather than python, or you may get nonresponsive GUIs.

2.1.4 Trial Run

To test your matplotlib installation, run IPython in pylab mode, which includes special support for interactive use of matplotlib. Linux and Mac users, run the following in a shell:

```
$ ipython -pylab
```

Windows users can edit the ipython launch icon properties to include the `-pylab` flag.

IPython's pylab mode automatically imports from matplotlib, and prepares the session for interactive plotting. At the command prompt (`In [1]:`), run the following:

```
plot([1,2,3])
```

A window should appear, which looks like figure 2.1. (If you get errors instead of a plot window, you probably were missing one of the packages required by matplotlib during installation.) Now that we have demonstrated how easy it can be to get started, perhaps it is now safe to explore the various options associated with installing and configuring matplotlib.

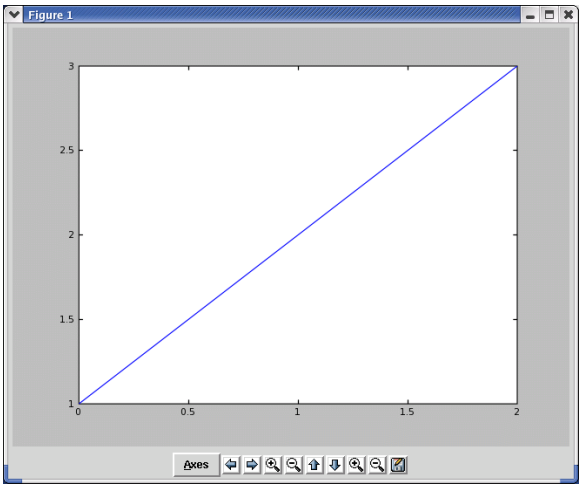

Figure 2.1: A simple plot shown in the TkAgg graphical user interface. Navigation controls shown below the figure provide an easy way to pan and zoom around your figures, and a save dialog allows you to save your figure after you have set the pan and zoom.

2.2 Backends

The matplotlib backends are responsible for taking the figure representation and transferring this to a display device, either a hardcopy image (`*.jpg`, `*.png`, `*.ps`, `*.svg`, etc) or a GUI window that you can interact with. There are many GUIs for python: pygtk, wxpython, Tkinter, PyQT, pyfltk, and more, and matplotlib supports most of them.

In choosing your backend, the following considerations are important:

- What kind of output do you require? Any matplotlib installation can generate PS and SVG. For other hardcopy formats, different backends have different capabilities. Agg can only generate png but produces the highest quality output (antialiased, alpha). The native GTK and WX backends support many more image formats (JPG, TIFF, ...) but can only be used in GUI mode and produce lower quality images. The GUI hybrid backends (WXAgg, GTKAgg, Tkagg, FLTKAgg, QtAgg, Qt4Agg) have the same limitations and capabilities as Agg.

- Do you want to produce plots interactively from the python shell? Because most GUIs have a mainloop, they become unresponsive to input outside of their mainloop once they are launched. Thus you often need to use a custom shell to work interactively with a GUI application from the shell (pycrust for wx, PyShell for gtk). A notable exception is Tkinter, which can be controlled from a standard python shell or ipython. Fernando Perez, the author of ipython, has written a pylab mode for ipython that lets you use WX, GTK, Qt, Qt4 or Tk interactively from the python shell. If you want to work interactively with matplotlib, this is the recommended approach.

- What platform do you work most on? Do you want to embed matplotlib in an application that you distribute across platforms? Do you need a GUI interface? Each of the python GUIs work on all major platforms, but some are easier than others to install. Each have different advantages: GTK is natural for linux and has excellent looking widgets but is a tough install on OS X. Tkinter is deployed with most python installations but has primitive looking widgets. wxpython has native widgets but can be difficult to install. *Windows users note: the enthought edition of python from* `http://www.enthought.com/python` *comes with Tkinter and wxpython included*. Now that Qt-4 has been released under the GPL for windows, the Qt backend is a new alternative with excellent looking widgets.

- What features do you need? Some of the matplotlib features including alpha blending, antialiasing, images and mathtext are not ported to all backends. Agg and the *Agg hybrids support all matplotlib features (agg is a core matplotlib backend). postscript, native gtk and native wx do not support alpha or antialiasing. svg supports everything except mathtext (which will hopefully be supported soon).

- Do you need dynamic images such as animation? The GUI backends vary in their ability to support rapid updating of the image canvas. GTKAgg is currently the fastest backend for animation, with FLTKAgg a close second.

Once you have decided on which backends you want to use, make sure you have installed the required GUI toolkits (and devel versions if you are using a package manager). Then install matplotlib and edit your `matplotlibrc` files to reflect these choices as described in section 2.6.

2.3 Integrated development environments

If you work primarily in an integrated development environment such as idle, pycrust, SciTE, Pythonwin, you will probably want to set your default backend to be compatible with the GUI your IDE uses. See Table 2.1 for a summary of the various python IDEs and their matplotlib compatibility.[1]

IDE	GUI	Backends and Options
idle	Tkinter	Works best with TkAgg if idle is launched with -n
pycrust	WX	Works best with WX/WXAgg
Scintilla and SciTE	GTK	Should work with GTK/GTKAgg backends but untested
Eric3, Eric4	Qt, Qt4	works with QtAgg, Qt4Agg
pythonwin	MFC	Unknown

Table 2.1: python IDEs and matplotlib compatibility.

2.4 Interactive

The recommended way to use matplotlib interactively from a shell is with IPython. IPython has a `-pylab` mode that detects your matplotlibrc file and makes the right settings to run matplotlib with your GUI of choice in interactive

[1] If you have experience with these or other IDEs and matplotlib backends to help me finish this table, please contact me or the matplotlib-devel mailing list.

mode using threading. Ipython's pylab mode is compatible with the Tk, GTK, WX and Qt GUI toolkits. GTK users will need to make sure that they have compiled GTK with threading for this to work. Using ipython in pylab mode is basically a nobrainer because it knows enough about matplotlib internals to make all the right settings for you internally.

```
peds-pc311:~> ipython -pylab
Python 2.3.3 (#2, Apr 13 2004, 17:41:29)
Type "copyright", "credits" or "license" for more information.

IPython 0.6.5 -- An enhanced Interactive Python.
?       -> Introduction to IPython's features.
%magic  -> Information about IPython's 'magic' % functions.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
  help(matplotlib) -> generic matplotlib information.
  help(matlab)     -> matlab-compatible commands from matplotlib.
  help(plotting)   -> plotting commands.

In[1]: plot( rand(20), rand(20), 'go' )
```

Note that you did not need to import any matplotlib names because in pylab mode ipython will import them for you. ipython turns on interactive mode for you, and also provides a `run` command so you can run matplotlib scripts from the matplotlib shell and then interactively update your figure. ipython will turn off interactive mode during a run command for efficiency, and then restore the interactive state at the end of the run.

```
>>> cd python/projects/matplotlib/examples/
/home/jdhunter/python/projects/matplotlib/examples
>>> run simple_plot.py
>>> title('a new title', color='r')
```

The pylab interface provides 4 commands that are useful for interactive control. Note again that the interactive setting primarily controls whether the figure is redrawn with each plotting command. `isinteractive` returns the interactive setting, `ion` turns interactive on, `ioff` turns it off, and `draw` forces a redraw of the entire figure. Thus when working with a big figure in which drawing is expensive, you may want to turn matplotlib's interactive setting off temporarily to avoid the performance hit

```
>>> run mybigfatfigure.py
>>> ioff()      # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()      # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
>>> ion()      # turn updates back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
```

If you are not using `ipython -pylab`, then by default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation. Often you don't want to update the plot every time a single property is changed, only once after all the properties have changed. But in interactive mode, eg from the standard python shell, you usually do want to update the plot with every command, eg, after changing the xlabel or the marker style of a line. To do this, you need to set `interactive : True` in your configuration file; see Section 2.6.

There are many python shells out there: the standard python shell, ipython, PyShell, pysh, pycrust. Some of these are GUI dependent (PyShell/pycrust) and some are not (ipython, pysh). As discussed in backends Section 2.3, not all shells are compatible with all matplotlib backends because of GUI mainloop issues. With a non-GUI python shell

such as the standard python shell or pysh, the TkAgg backend is the best choice for interactive use. Just set `backend : TkAgg` and `interactive : True` in your `matplotlibrc` file and fire up python. Then

```
# using matplotlib interactively from the python shell
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. Note, in batch mode, ie when making figures from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want to think carefully before making this the default behavior.

2.5 Numerix

Numeric is the original python module for efficiently processing arrays of numeric data. While highly optimized for performance and very stable, some limitations in the design made it inefficient for very large arrays, and developers decided it was better to start with a new array package to solve some of these design problems and numarray was born. In a sense, the numerical python community split into Numeric and numarray user groups. Travis Oliphant, one of the maintainers of Numeric, began work on a third package, based on the Numeric code base, which incorporated the advances made in numarray, and was called NumPy. NumPy is intended to be the successor to both Numeric and numarray, and to reunite the numerical python community. An array interface was developed in order to allow the three array packages to play well together and to easy migration to NumPy. Numeric is no longer undergoing active development, and the numarray release notes suggest users to switch to Numpy.

Matplotlib requires one of Numeric, numarray, or NumPy to operate. If you have no experience with any of them, you are strongly advised to install Numpy and read through some of the documentation before continuing. The `matplotlib.numerix` module, written by Todd Miller, allows you to choose between Numeric, numarray and NumPy at the prompt or in a config file. Thus when you do

```
# import matplotlib and all the numerix functions
from pylab import *
```

you'll not only get all the matplotlib pylab interface commands, but most of the Numeric, numarray or NumPy package as well (depending on your numerix setting). All of the array creation and manipulation functions are imported, such as `array`, `arange`, `take`, `where`, etc. The other modules, such as mlab, fft and linear_algebra, are available under the numarray package structure. To make your matplotlib scripts as portable as possible with respect to your choice of array packages, it is advised not to explicitly import Numeric, numarray or NumPy. Rather, you should use matplotlib.numerix where possible, either by using the functions imported by `pylab`, or by explicitly importing the numerix module, as in

```
# create a numerix namespace
import matplotlib.numerix as n
x = n.arange(100)
y = n.take(x, range(10,20))
```

For the remainder of this manual, the term *numerix* is used to mean either the Numeric, numarray or NumPy package.

2.5.1 Choosing Numeric, numarray, or NumPy

To select Numeric, numarray, or NumPy from the prompt, run your matplotlib script with

```
> python myscript.py --numarray  # use numarray
> python myscript.py --Numeric   # use Numeric
```

Typically, however, users will choose one or the other and make this setting in their rc file using either `numerix : Numeric`, `numerix : numarray`, or `numerix : numpy`; see Section 2.6.

Since the array packages all play well together, we expect that in the near future, matplotlib will depend on NumPy alone.

2.6 Customization using `matplotlibrc`

Almost all of the matplotlib settings and figure properties can be customized with a plain text file `matplotlibrc`. This file is installed with the rest of the matplotlib data (fonts, icons, etc) into a directory determined by distutils. Before compiling matplotlib, it resides in the same dir as `setup.py` and will be copied into your install path. Typical locations for this file are

```
C:\Python24\Lib\site-packages\matplotlib\mpl-data\matplotlibrc  # windows
/usr/lib/python2.4/site-packages/matplotlib/mpl-data/matplotlibrc # linux and friends
```

By default, the installer will overwrite the existing file in the install path, so if you want to preserve your changes, please move it to the .matplotlib directory in your HOME directory (and set the HOME environment variable if necessary).

In the rc file, you can set your backend (Section 2.2), your numerix setting (Section 2.5), whether you'll be working interactively (Section 2.4) and default values for most of the figure properties.

2.6.1 RC file format

Blank lines, or lines starting with a comment symbol, are ignored, as are trailing comments. Other lines must have the format

```
key : val   # optional comment
```

where `key` is some property like `backend`, `lines.linewidth`, or `figure.figsize` and `val` is the value of that property. Example entries for these properties are

```
# this is a comment and is ignored
backend           : GTKAgg     # the default backend
lines.linewidth   : 0.5        # line width in points
figure.figsize    : 8, 6       # figure size in inches
```

A complete sample rc file is shown in Appendix A.

The matplotlib rc values are read into a dictionary `rcParams` which contains the key/value pairs. You can changes these values within a script by importing this dictionary. For example, to require that a given script uses numarray, you could do

```
from matplotlib import rcParams
rcParams['numerix'] = 'numarray'
from pylab import *
```

Additionally, the commands `matplotlib.rc` and `matplotlib.rcdefaults` can be used to dynamically customize the defaults during a script execution (discussed below).

2.6.2 Which rc file is used?

matplotlib will search for an rc file in the following locations

- The current directory - this allows you to have a project specific configuration that differs from your default configuration

- Your HOME dir. On linux and other UNIX operating systems, this environment variable is set by default. Windows users can set in the My Computer properties
- PATH/matplotlibrc where PATH is the return value of `matplotlib.get_data_path()`. This function looks where distutils would have installed the file - if it doesn't find it there, it checks for the environment variable MATPLOTLIBDATA and uses that if found. The latter should be set if you are installing matplotlib to a non-standard location. Eg, if you install matplotlib with `python setup.py build -prefix=/home/jdhunter` then set matplotlib data to `/home/jdhunter/share/matplotlib`.
- After all that, if it cannot find your rc file, it will issue a warning and use defaults. This is not recommended!

2.6.3 Getting feedback from matplotlib

matplotlib uses a verbose setting, defined in the matplotlibrc file to determine how much information to report.

```
verbose.level  : error       # one of silent, error, helpful, debug, debug-annoying
verbose.fileo  : sys.stdout  # a log filename, sys.stdout or sys.stderr
verbose.erro   : sys.stderr  # a log filename, sys.stdout or sys.stderr
```

These settings control how much information matplotlib gives you at runtime and where it goes. The verbosity levels are: `silent, error, helpful, debug, debug-annoying`. At the `error` level, you will only get error messages. Any level is inclusive of all the levels below it. Ie, if your setting is `helpful`, you'll also get all the `error` messages. If you setting is `debug`, you'll get all the `error` and `helpful` messages. It is not recommended to make your setting `silent` because you will not even get `error` messages. When submitting problems to the mailing-list, please set verbose to `helpful` or `debug` and paste the output into your report.

The `verbose.fileo` setting gives the destination for any calls to the verbose report function. The `verbose.erro` setting gives the destination for any calls to verbose error reporting function. These objects can a filename or a full path to a filename, `sys.stderr`, or `sys.stdout`. You can override the rc default verbosity from the command line by giving the flags `-verbose-LEVEL` where `LEVEL` is one of the legal levels, eg `-verbose-error -verbose-helpful`.

You can access the verbose instance in your code `from matplotlib import verbose`.

Chapter 3

The pylab interface

Although matplotlib has a full object oriented API (see Chapter 10), the primary way people create plots is via the pylab interface, which can be imported with

```
from pylab import *
```

This import command brings in all of the matplotlib code needed to produce plots, the extra MATLAB^{TM} compatible, non-plotting functions found in `matplotlib.mlab` and all of the matplotlib.numerix code needed to create and manipulate arrays. When you import `pylab`, you will get all of NumPy (or Numeric or numarray depending on your `numerix` setting).

matplotlib is organized around figures and axes. The figure contains an arbitrary number of axes, which can be placed anywhere in the figure you want, including over other axes. You can directly create and manage your own figures and axes, but if you don't, matplotlib will try and do the right thing by automatically creating default figures and axes for you.

There are two ways of working in the pylab interface: interactively or in script mode. When working interactively, you want every plotting command to update the figure. Under the hood, this means that the canvas is redrawn after every command that affects the figure. When working in script mode, this is inefficient. In this case, you only want the figure to be drawn once, either to the GUI window or saved to a file. To handle these two cases, matplotlib has an `interactive` setting in `matplotlibrc`. When `interactive : True`, the figure will be redrawn with each command. When `interactive : False`, the figure will be drawn only when there is a call to `show` or `savefig`. In the examples that follow, I'll assume you have set `interactive : True` in your `matplotlibrc` file and are working from an interactive python shell using a compatible backend. Please make sure you have read and understood Sections 2.2, 2.3, 2.4 and 2.6, before trying these examples.

3.1 Simple plots

Just about the simplest plot you can create is

```
>>> from pylab import *
>>> plot([1,2,3])
```

I have set my backend to `backend : TkAgg`, which causes the plot in Figure 2.1 to appear, with navigation controls for interactive panning and zooming.

I can continue to decorate the plot with labels and titles

```
>>> xlabel('time (s)')
>>> ylabel('volts')
>>> title('A really simple plot')
>>> grid(True)
```

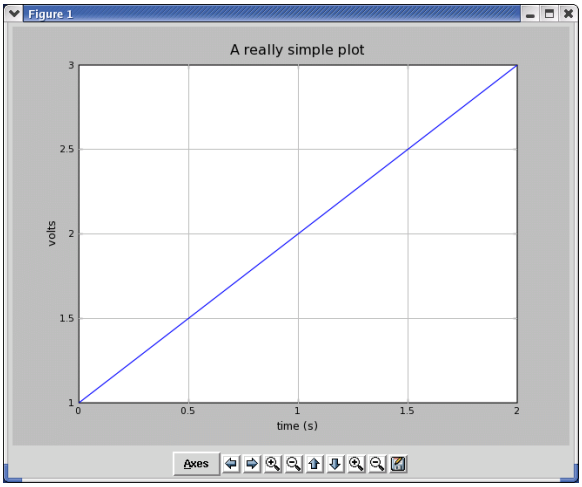

Figure 3.1: A simple plot decorated with some text labels and an axes grid

and the updated figure is shown in Figure 3.1.

At this point we're getting a little bored plotting `[1,2,3]`. matplotlib is designed around plotting numerix arrays, and can handle large arrays efficiently. To create a regularly sampled 1 Hz sine wave use the arange and sin methods methods provided by numerix which produces the plot shown in Figure 3.2.

```
>>> t = arange(0.0, 3.0, 0.05)  # in matlab t = [0.0: 0.05: 3.0];
>>> s = sin(2*pi*t)
>>> plot(t,s)
```

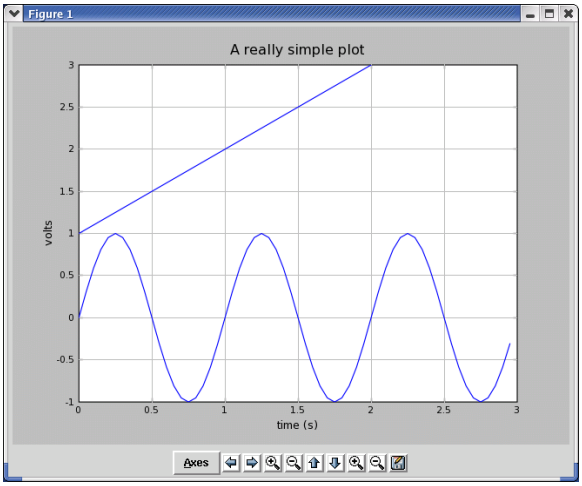

Figure 3.2: A sine wave added to the simple plot. This may not be what we wanted. Because the `hold` state was on, the two plots were superimposed.

Note that the two plots are superimposed. matplotlib (and MATLAB^{TM}) have a `hold` state. When hold is on, subsequent plotting commands are superimposed over previous commands. When hold is off, the plot is cleared with every plotting command. This is controlled by the `hold` command, which can be called like `hold(True)` or `hold(False)`. The default setting is in `matplotlibrc` as `axes.hold : True`, which you can change according to your preferences.

To clear the previous plot and reissue the plot command for just the sine wave, you can use `cla` to clear the current axes and `clf` to clear the current figure, or simply turn the hold state off.

```
>>> hold(False)
>>> plot(t, s)
```

3.2 More on plot

3.2.1 Multiple lines

`plot` is a versatile command, and will create an arbitrary number of lines with different line styles and markers. This example plots a sine wave and a damped exponential using the default line styles

```
>>> clf()   # clear the figure
>>> t = arange(0.0, 5.0, 0.05)
>>> s1 = sin(2*pi*t)
>>> s2 = s1 * exp(-t)
>>> plot(t, s1, t, s2)
```

If you plot multiple lines in a single plot command, the line color will cycle through a list of predefined colors. The default line color and line style are determined by the rc parameters `lines.style` and `lines.color`. You can include an optional third string argument to each line in the plot command, which specifies any of the line style, marker style and line color. To plot the above using a green dashed line with circle markers, and a red dotted line with circle markers, as shown in Figure 3.3,

```
>>> clf()
>>> plot(t, s1, 'g--o', t, s2, 'r:s')
>>> legend(('sine wave', 'damped exponential'))
```

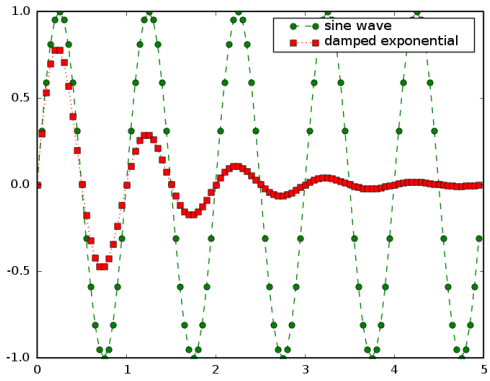

Figure 3.3: All line plots take an optional third string argument, which is composed of (optionally) a line color (eg, 'r', 'g', 'k'), a line style (eg, '-', '–', ':') and a line marker ('o', 's', 'd'). The sine wave line (green dashed line with circle markers) is created with 'g–o'. The `legend` command will automatically create a legend for all the lines in the plot.

The color part of the format string applies only to the facecolor of 2D plot markers like circles, triangles, and squares. The edgecolor of these markers will be determined by the default rc parameter `lines.markeredgecolor` and can be defined for individual lines using the methods discussed below.

3.2.2 Controlling line properties

In the last section, we showed how to choose the default line properties using plot format strings. For finer grained control, you can set any of the attributes of a `matplotlib.lines.Line2D` instance. There are three ways to do this: using keyword arguments, calling the line methods directly, or using the `set` command. The line properties are shown in Table 3.1.

Property	*Value*
alpha	The alpha transparency on 0-1 scale
antialiased	*True* or *False* - use antialised rendering
color	A matplotlib color arg
data_clipping	Whether to use numeric to clip data
label	A string optionally used for legend
linestyle	One of `- : -. -`
linewidth	A float, the line width in points
marker	One of `+ , o . s v x > <, etc`
markeredgewidth	The line width around the marker symbol
markeredgecolor	The edge color if a marker is used
markerfacecolor	The face color if a marker is used
markersize	The size of the marker in points

Table 3.1: Line properties; see `pylab.plot` for more marker styles

Using keyword arguments to control line properties

You can set any of the line properties listed in Table 3.1 using keyword arguments to the plot command. The following command plots large green diamonds with a red border

```
>>> plot(t, s1, markersize=15, marker='d', \
...       markerfacecolor='g', markeredgecolor='r')
```

Using `set` to control line properties

You can set any of the line properties listed in Table 3.1 using the `set` command. Set operates on the return value of the plot command (a list of lines), so you need to save the lines. You can use an arbitrary number of key/value pairs

```
>>> lines = plot(t, s1)
>>> set(lines, markersize=15, marker='d', \
...      markerfacecolor='g', markeredgecolor='r')
```

`set` can either operate on a single instance or a sequence of instances (in the example code above, `lines` is a length one sequence of lines). Under the hood, if you pass a keyword arg named *something*, `set` looks for a method of the object called `set_something` and will call it with the value you pass. If `set_something` does not exist, then an exception will be raised.

Using `matplotlib.lines.Line2D` methods

You can also call Line2D methods directly. The return value of plot is a sequence of `matplotlib.lines.Line2D` instances. Note in the example below, I use tuple unpacking with the "," to extract the first element of the sequence as *line*: `line, = plot(t, s1)`

```
>>> line, = plot(t, s1)
>>> line.set_markersize(15)
>>> line.set_marker('d')
```

```
>>> line.set_markerfacecolor('g')
>>> line.set_markeredgecolor('r')
```

Note, however, that we haven't issued any `pylab` commands after the initial `plot` command so the figure will not be redrawn even though interactive mode is set. To trigger a redraw, you can simply resize the figure window a little or call the `draw` method. The fruits of your labors are shown in Figure 3.4.

```
>>> draw()
```

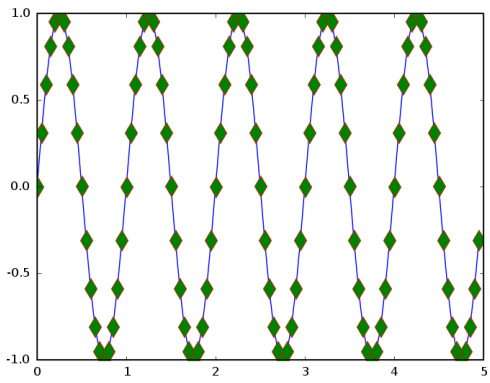

Figure 3.4: Large green diamonds with red borders, created with three different recipes.

Abbreviated method names

Abbreviation	*Fullname*
aa	antialiased
c	color
ls	linestyle
lw	linewidth
mec	markeredgecolor
mew	markeredgewidth
mfc	markerfacecolor
ms	markersize

Table 3.2: Abbreviated names for line properties. You can use any of the line customization methods above with abbreviated names.

When working from an interactive python shell, typing 'markerfacecolor' can be a pain – too many keystrokes. The `matplotlib.lines.Line2D` class provides a number of abbreviated method names, listed in Table 3.2. Thus you can, for example, call

```
# no antialiasing, thick green markeredge lines
>>> plot(range(10), 'ro', aa=False, mew=2, mec='g')
```

3.3 Color arguments

matplotlib is fairly tolerant of a number of formats for passing color information. As discussed above, you can use and of the single character color strings listed in Table 3.3. Additionally, anywhere a color character string is accepted, you can also use a grayscale, hex, RGB color argument, or any legal hml color name, ebg “red” or “darkslategray”.

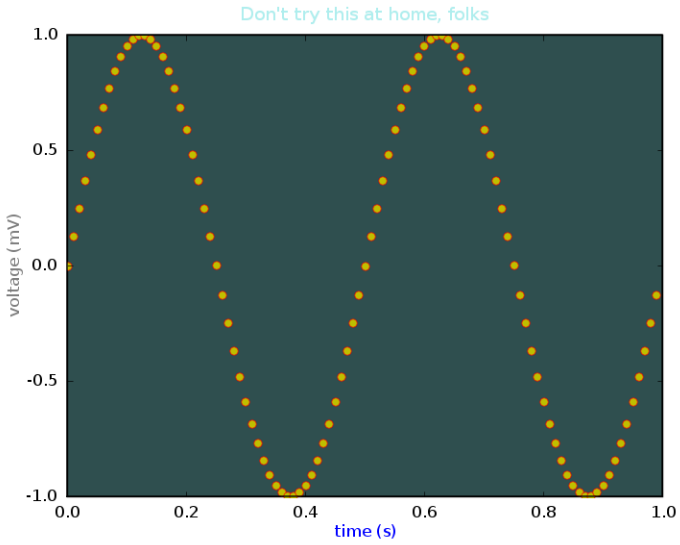

Figure 3.5: Lots of different ways to specify colors generated from Listing 3.1– not necessarily recommended for aesthetic quality!

Listing 3.1: Wild and wonderful ways to specify colors; see Figure 3.5

```
from pylab import *

# axis background in dark slate gray
subplot(111, axisbg=(0.1843, 0.3098, 0.3098))
t = arange(0.0, 1.0, 0.01)
s = sin(2*2*pi*t)

# yellow circles with red edge color
plot(t, s, 'yo', markeredgecolor='r')
xlabel('time (s)', color='b')       # xlabel is blue
ylabel('voltage (mV)', color='0.5')   # ylabel is light gray
title("Don't try this at home, folks", color='#afeeee')
```

3.4 Loading and saving data

`pylab` provides support for loading and saving ASCII arrays or vectors with the `load` and `save` command. `matplotlib.numerix` provides support for loading and saving binary arrays with the `fromstring` and `tostring` methods.

3.4.1 Loading and saving ASCII data

Suppose you have an ASCII file of measured times and voltages like so

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white
0.75	a grayscale intensity (any float in [0,1]
#2F4F4F	an RGB hex color string, eg, this example is dark slate gray
(0.18, 0.31, 0.31)	an RGB tuple; this is also dark slate gray
red	any legal html color name

Table 3.3: Color format strings, which can be used to set the line or text properties, eg the line, the marker edgecolor or marker facecolor.

```
0.0000 0.4911
0.0500 0.5012
0.1000 0.7236
0.1500 1.1756
... and so on
```

You can load that data into an array `X` with the `load` command. The shape of X is numSamples rows by 2 columns, with the first column containing the time points and the second column containing the measured voltages. You can use numerix array indexing to extract the two columns into the 1D arrays `t` and `s`

```
X = load('../data/ascii_data.dat')
t = X[:,0]  # the first column
s = X[:,1]  # the second row
plot(t, s, 'o')
```

Likewise, you can save array or vector data in an ASCII file with the `save` command. The following script was used to create the sample data above

```
from pylab import *
t = arange(0.0, 1.0, 0.05)
s = sin(2*pi*t) + 0.5*rand(len(t))
X = zeros((len(t),2), Float)
X[:,0] = t
X[:,1] = s
save('../data/ascii_data.dat', X)
```

3.4.2 Loading and saving binary data

ASCII is bloated and slow for working with large arrays, and so binary data should be used if performance is a consideration. To save the array `X` in binary form, use the numerix `tostring` method

```
# open the file for writing binary and write the binary string
file('../data/binary_data.dat', 'wb').write(X.tostring())
```

This data can later be loaded into a numerix array using `fromstring`. This method takes two arguments, a string and a data type (note that numarray users can use `fromfile` which is more efficient for importing data directly from a file).

```
# load the data as a string
s = file('../data/binary_data.dat', 'rb').read()
```

```
# convert to 1D numerix array of type Float
X = fromstring(s, Float)

# reshape to numSamples rows by 2 columns
X.shape = len(X)/2, 2
t = X[:,0]  # the first column
s = X[:,1]  # the second row
plot(t, s, 'o')
```

Note that although Numerix and numarray use different typecode arguments (Numeric uses strings whereas numarray uses type objects), the matplotlib.numerix compatibility layer provides symbols which will work with either `numerix` rc setting.

3.4.3 Processing several data files

Since python is a programming language *par excellence*, it is easy to process data in batch. When I started the gradual transition from a full time MATLAB^{TM} user to a full time python user, I began processing my data in python and saving the results to data files for plotting in MATLAB^{TM}. When that became too cumbersome, I decided to write matplotlib so I could have all the functionality I needed in one environment. Here is a brief example showing how to iterate over several data files, named `basename001.dat, basename002.dat, basename003.dat, ... basename100.dat` and plot all of the traces to the same axes. I'll assume for this example that each file is a 1D ASCII array, which I can load with the `load` command.

```
hold(True)  # set the hold state to be on
for i in range(1,101):  #start at 1, end at 100
    fname = 'basename%03d.dat'%i  # %03d pads the integers with zeros
    x = load(fname)
    plot(x)
```

3.5 axes and figures

All the examples thus far used *implicit* figure and axes creation. You can use the functions `figure`, `subplot`, and `axes` to explicitly control this process. Let's take a look at what happens under the hood when you issue the commands

```
>>> from pylab import *
>>> plot([1,2,3])
```

When plot is called, the pylab interface makes a call to `gca()` ("get current axes") to get a reference to the current axes. `gca` in turn, makes a call to `gcf` to get a reference to the current figure. `gcf`, finding that no figure has been created, creates the default figure `figure()` and returns it. `gca` will then return the current axes of that figure if it exists, or create the default axes `subplot(111)` if it does not. Thus the code above is equivalent to

```
>>> from pylab import *
>>> figure()
>>> subplot(111)
>>> plot([1,2,3])
```

3.5.1 `figure`

You can create and manage an arbitrary number of figures using the `figure` command. The standard way to create a figure is to number them from $1 \ldots N$. A call to `figure(1)` creates figure 1 if it does not exist, makes figure 1 active (`gcf` will return a reference to it), and returns the `matplotlib.figure.Figure` instance. The syntax of the `figure` command is

```
def figure(num=1,
           figsize   = None, # defaults to rc figure.figsize
           dpi       = None, # defaults to rc figure.dpi
           facecolor = None, # defaults to rc figure.facecolor
           edgecolor = None, # defaults to rc figure.edgecolor
           frameon = True,   # whether to draw the figure frame
           ):
```

figsize gives the figure size in inches and is width by height. Eg, to create a figure 12 inches wide and 2 inches high, you can call `figure(figsize=(12,2))`. *dpi* gives the dots per inch of your display device. Increasing this number effectively creates a higher resolution figure. *facecolor* and *edgecolor* determine the face and edge color of the figure rectangular background. This is what gives the figure a gray background in the GUI figures such as Figure 2.1. You can turn this background completely off by setting `frameon=False`. The default for saving figures is to have a white face and edge color, and all of these properties can be customized using the rc parameters `figure.*` and `savefig.*`.

In typical usage, you will only provide the figure number, and let your rc parameters govern the other figure attributes

```
>>> figure(1)
>>> plot([1,2,3])
>>> figure(2)
>>> plot([4,5,6])
>>> title('big numbers')    # figure 2 title
>>> figure(1)
>>> title('small numbers')  # figure 1 title
```

You can close a figure simply by clicking on the close "x" in the GUI window, or by issuing the `close` command. `close` can be used to close the current figure, a figure referenced by number, a given figure instance, or all figures

- `close()` by itself closes the current figure
- `close(num)` closes figure number *num*
- `close(fig)` where *fig* is a figure instance closes that figure
- `close('all')` closes all the figure windows

If you close a figure directly, eg `close(2)` the previous current figure is restored to the current figure. `clf` is used to clear the current figure without closing it.

If you save the return value of the figure command, you can call any of the methods provided by `matplotlib.figure.Figure`, for example, you can set the figure facecolor

```
>>> fig = figure(1)
>>> fig.set_facecolor('g')
```

or use `set` for the same purpose

```
>>> set(fig, facecolor='g')
```

3.5.2 subplot

`axes` and `subplot` are both used to create axes in a figure. `subplot` is used more commonly, and creates axes assuming a regular grid of axes *numRows* by *numCols*. For example, to create two rows and one column of axes, you would use `subplot(211)` to create the upper axes and `subplot(212)` to create the lower axes. The last digit counts across the rows.

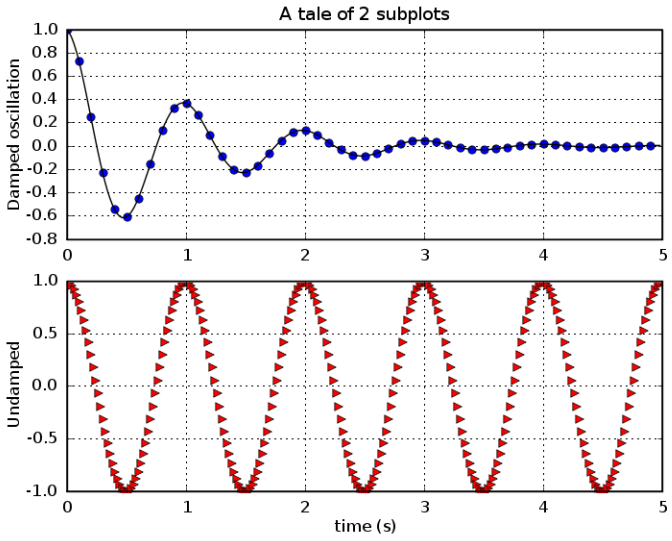

Figure 3.6: Multiple rows of axes created with the subplot command, as shown in Listing 3.2

Listing 3.2: Generating multiple axes with subplot; see Figure 3.6

```
from pylab import *

def f(t):
    'a damped oscillation'
    return cos(2*pi*t) * exp(-t)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)

# the upper subplot; 2 rows, 1 column, subplot #1
subplot(211)
l = plot(t1, f(t1), 'bo', t2, f(t2), 'k')
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

# the lower subplot; 2 rows, 1 column, subplot #2
subplot(212)
plot(t2, cos(2*pi*t2), 'r>')
grid(True)
xlabel('time (s)')
ylabel('Undamped')
```

Likewise, to create two columns and one row of axes, you would use `subplot(121)` to create the left axes and `subplot(122)` to create the right axes. If the total number of axes exceeds single digits, use comma separated arguments to subplot. For example, the lower right panel of a 3 x 4 grid of axes is created with `subplot(3,4,12)`. matplotlib uses MATLAB™ style indexing in creating figures and axes, so `subplot(3,4,1)` is the first subplot, not `subplot(3,4,0)`.

The subplot command returns a `matplotlib.axes.Subplot` instance, which is derived from `matplotlib.axes.Axes`. Thus you can call and Axes or Subplot method on it. When creating multiple subplots with the same axes, for example the same time axes, sometimes it helps to turn off the x tick labeling for all but the lowest plot. Here is some example code

```
subplot(211)
plot([1,2,3], [1,2,3])
set(gca(), xticklabels=[])

subplot(212)
plot([1,2,3], [1,4,9])
```

Likewise, with multiple columns and shared y axes, you may want turn off the ytick labels for all but the first row. The subplot command returns a `matplotlib.axes.Subplot` instance, which is derived from `matplotlib.axes.Axes`. Thus you can call and Axes or Subplot method on it. Subplot defines some helper methods (`is_first_row`, `is_first_col`, `is_last_row`, `is_last_col`, to help you conditionally set subplot properties, eg

```
cnt = 0
for i in range(numRows):
    for j in range(numCols):
        cnt += 1
        ax = subplot(numRows, numCols, cnt)
        plot(blah, blah)
        if ax.is_last_row()  : xlabel('time (s)')
        if ax.is_first_col() : ylabel('volts')
```

Here is some example code to create multiple figures and axes, using the `figure` and `subplot` command to control the current figure and axes.

```
from pylab import *

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = sin(4*pi*t)

figure(1)
subplot(211)
plot(t,s1)
subplot(212)
plot(t,2*s1)

figure(2)
plot(t,s2)

# now switch back to figure 1 and make some changes to the upper
# subplot
figure(1)
subplot(211)
plot(t,s2, 'gs')
set(gca(), 'xticklabels', [])

show()
```

3.5.3 `axes`

When you need a finer grained control over axes placement than afforded by `subplot`, use the `axes` command. The `axes` command in initialized with a rectangle `[left, bottom, width, height]` in relative figure coordinates.

`left, bottom = (0, 0)` is the bottom left of the of the figure canvas, and a `width`/`height` of 1 spans the figure width/height. This to create an axes that entirely fills the figure canvas, you would do `axes([0, 1, 0, 1])`. This may not be a good idea, because it leaves no room for text labels. `axes([0.25, 0.25, 0.5, 0.5])` creates an axes offset by one quarter of the figure width and height on all sides.

There are several ways to use the `axes` command; in all cases, a `matplotlib.axes.Axes` instance is returned

- `axes()` by itself creates a default full `subplot(111)` window axis
- `axes(rect, axisbg='w')` where `rect=[left, bottom, width, height]` in normalized (0,1) units. *axisbg* is the background color for the axis, default white.
- `axes(ax)` where *ax* is an axes instance makes *ax* current.

`gca` returns the current axes instance and `cla` clears the current axes. You can use the `axes` command lay the axes exactly where you want them, including to overlaying one axes on top of another, as in this example

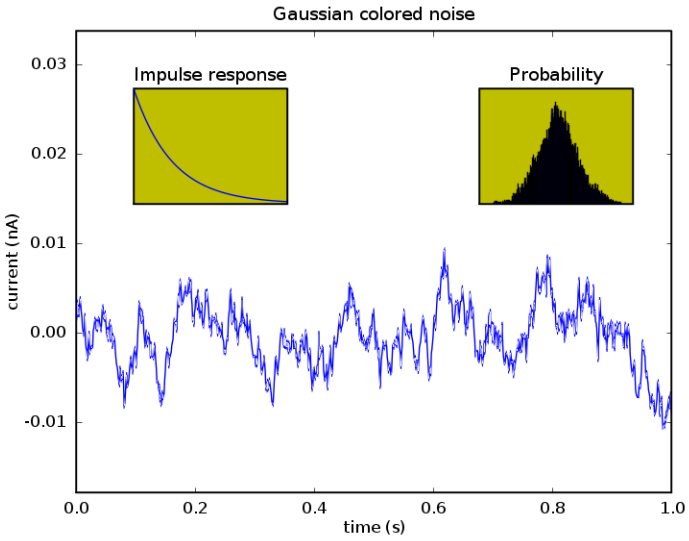

Figure 3.7: Using the `axes` command to create inset axes over another axes; see Listing 3.3

Listing 3.3: Custom axes; see Figure 3.7

```
from pylab import *

# create some data to use for the plot
dt = 0.001
t = arange(0.0, 10.0, dt)
r = exp(-t[:1000]/0.05)               # impulse response
x = randn(len(t))
s = convolve(x,r,mode=2)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plot(t, s)
axis([0, 1, 1.1*min(s), 2*max(s) ])
xlabel('time (s)')
ylabel('current (nA)')
```

```
title('Gaussian colored noise')

# this is an inset axes over the main axes
a = axes([.65, .6, .2, .2], axisbg='y')
n, bins, patches = hist(s, 400, normed=1)
title('Probability')
setp(a, xticks=[], yticks=[])

# this is another inset axes over the main axes
a = axes([0.2, 0.6, .2, .2], axisbg='y')
plot(t[:len(r)], r)
title('Impulse response')
setp(a, xlim=(0,.2), xticks=[], yticks=[])
```

3.6 Text

matplotlib has excellent text support, including newline separated text with arbitrary rotations and mathematical expressions. freetype2 support produces very nice, antialiased fonts, that look good even at small raster sizes. It includes its own font_manager, thanks to Paul Barrett, which implements a cross platform, W3C compliant font finding algorithm. You have total control over every text property (font size, font weight, text location and color, etc) with sensible defaults set in the rc file. And significantly for those interested in mathematical or scientific figures, matplotlib implements a large number of TEX math symbols and commands, to support mathematical expressions anywhere in your figure. To get the most out of text in matplotlib, you should use a backend that supports freetype2 and mathtext, notably all the *Agg backends (see Section 2.2), or the postscript backend, which embeds the freetype fonts directly into the PS/EPS output file.

3.6.1 Basic text commands

The following commands are used to create text in the pylab interface

- `xlabel(s)` - add a label *s* to the x axis
- `ylabel(s)` - add a label *s* to the y axis
- `title(s)` - add a title *s* to the axes
- `text(x, y, s)` - add text *s* to the axes at *x*, *y* in data coords
- `figtext(x, y, s)` - add text to the figure at *x*, *y* in relative 0-1 figure coords

3.6.2 Text properties

The text properties are listed in Table 3.4. As with lines, there are three ways to set text properties: using keyword arguments to a text command, calling `set` on a text instance or a sequence of text instances, or calling an instance method on a text instance. These three are illustrated below

```
# keyword args
>>> xlabel('time (s)', color='r', size=16)
>>> title('Fun with text', horizontalalignment='left')

# use set
>>> locs, labels = xticks()
>>> set(labels, color=g', rotation=45)
```

```
# instance methods
>>> l = ylabel('volts')
>>> l.set_weight('bold')
```

Property	Value
alpha	The alpha transparency on 0-1 scale
color	A matplotlib color arg
family	set the font family, eg 'sans-serif', 'cursive', 'fantasy'
fontangle	the font slant, one of 'normal', 'italic', 'oblique'
horizontalalignment	'left', 'right' or 'center'
multialignment	'left', 'right' or 'center' only for multiline strings
name	the font name, eg, 'Sans', 'Courier', 'Helvetica'
position	the x,y location
variant	the font variant, eg 'normal', 'small-caps'
rotation	the angle in degrees for rotated text
size	the fontsize in points, eg, 8, 10, 12
style	the font style, one of 'normal', 'italic', 'oblique'
text	set the text string itself
verticalalignment	'top', 'bottom' or 'center'
weight	the font weight, eg 'normal', 'bold', 'heavy', 'light'

Table 3.4: Properties of `matplotlib.text.Text`

See the example http://matplotlib.sourceforge.net/examples/fonts_demo_kw.py which makes extensive use of font properties for more information. See also Chapter 4 for more discussion of the font finder algorithm and the meaning of these properties.

3.6.3 Text layout

You can layout text with the alignment arguments *horizontalalignment*, *verticalalignment*, and *multialignment*. *horizontalalignment* controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. *verticalalignment* controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. *multialignment*, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the `text` command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the axes bounding box, with 0,0 being the lower left of the axes and 1,1 the upper right.

Listing 3.4: Aligning text; see Figure 3.8

```
from pylab import *
from matplotlib.patches import Rectangle

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
ax = gca()
p = Rectangle((left, bottom), width, height,
              fill=False,
              )
# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p.set_transform(ax.transAxes)
p.set_clip_on(False)
```

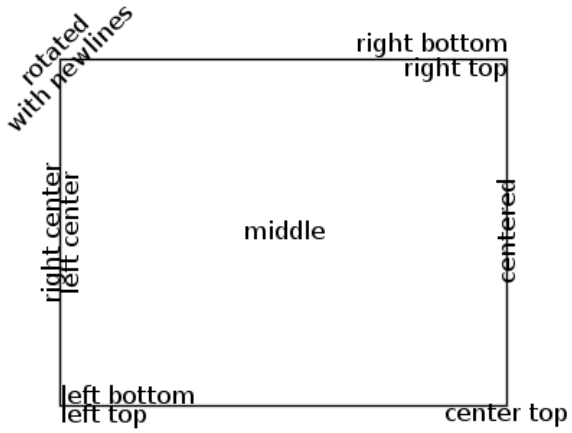

Figure 3.8: Aligning text with *horizontalalignment*, *verticalalignment*, and *multialignment* options to the `text` command; see Listing 3.4

```
ax.add_patch(p)


ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
```

```
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

axis('off')
```

3.6.4 mathtext

matplotlib supports TEX mathematical expressions anywhere a text string can be used, as long as the string is delimited by "$" on both sides, as in r'5λ'; embedded mathtext strings, such as in r'The answer is 5λ' are not currently supported. A large set of the TEX symbols from the computer modern fonts are provided. Subscripting and superscripting are supported, as well as the over/under style of subscripting with \sum, \int etc.

Note that matplotlib does not use or require that TEX be installed on your system, as it does not use it. Rather, it uses the parsing module pyparsing to parse the TEX expression, and does the layout manually in the matplotlib.mathtext module using the font information provided by matplotlib.ft2font.
The spacing elements \/ and \hspace{num} are provided. \/ inserts a small space, and \hspace{num} inserts a fraction of the current fontsize. Eg, if num=0.5 and the fontsize is 12.0, \hspace{0.5} inserts 6 points of space.

The following accents are provided: \hat, \breve, \grave, \bar, \acute, \tilde, \vec, \dot, \ddot. All of them have the same syntax, eg to make an $\bar{o}$ you do \bar{o} or to make an $\ddot{o}$ you do \ddot{o}. The shortcuts are also provided, eg:

```
\"o \'e \`e \~n \.x \^y
```

3.6.5 Annotations

The uses of the basic `text` command above place text at an arbitrary position on the Axes. A common use case of text is to *annotate* some feature of the plot, and the `Axes.annotate` method (pylab method `annotate`) provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument *xy* and the location of the text *xytext*. Both of these arguments are `(x,y)` tuples. Optionally, you can specify the coordinate system of xy and xytext with one of the following strings for *xycoords* and *textcoords* (default is 'data')

argument	coordinate system
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0,0 is lower left of figure and 1,1 is upper, right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the axes data coordinate system

Table 3.5: Coordinate systems for the text point *xy* and annotation text point *xytext* are supplied by optional keyword arguments *xycoords* and *textcoords* .

For physical coordinate systems (points or pixels) the origin is the (bottom, left) of the figure or axes. If the value is negative, however, the origin is from the (right, top) of the figure or axes, analogous to negative indexing of sequences.

Optionally, you can specify arrow properties which draws and arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument *arrowprops*.

arrowprops key	description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annotated point and text
*kwargs	any key for `matplotlib.patches.polygon` (eg facecolor)

Table 3.6: keys controlling the arrow properties in the *arrowprops* dictionary.

Listing 3.5: Annotations with different coordinate systems for the annotated point and text see Figure 3.9

```
from pylab import figure, nx, show
# you can specify the xypoint and the xytext in different
# positions and coordinate systems, and optionally turn on a
# connecting line and mark the point with a marker.  Annotations
# work on polar axes too.  In the example below, the xy point is
# in native coordinates (xycoords defaults to 'data').  For a
# polar axes, this is in (theta, radius) space.  The text in this
# example is placed in the fractional figure coordinate system.
# Text keyword args like horizontal and vertical alignment are
# respected
fig = figure()
ax = fig.add_subplot(111, polar=True)
r = nx.arange(0,1,0.001)
theta = 2*2*nx.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)
```

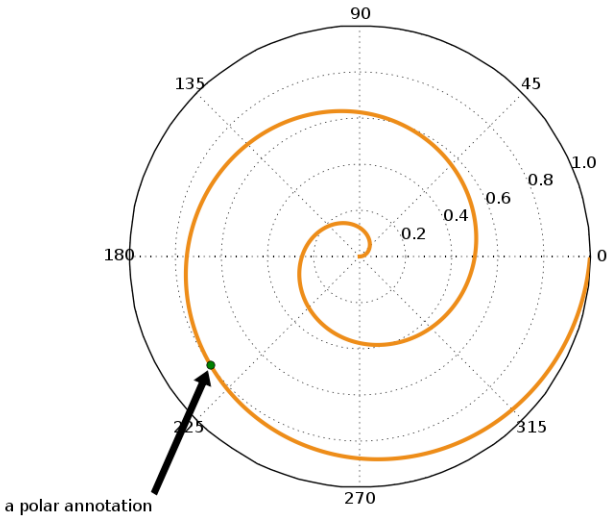

Figure 3.9: Sample annotation output generated from Listing 3.5

```
ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            )
```

Licensing

The computer modern fonts this package uses are part of the BaKoMa fonts, which are (in my understanding) free for noncommercial use. For commercial use, please consult the licenses in fonts/ttf and the author Basil K. Malyshev - see also http://www.mozilla.org/projects/mathml/fonts/encoding/license-bakoma.txt and the file BaKoMa-CM.Fonts in the matplotlib fonts dir.

Note that all the code in this module is distributed under the matplotlib license, and a truly free implementation of mathtext for either freetype or ps would simply require deriving another concrete implementation from the Fonts class defined in this module which used free fonts.

Using mathtext

Any text element can use math text. You need to use raw strings (preceed the quotes with an `r`), and surround the string text with dollar signs, as in TEX.

```
# plain text
title('alpha > beta')

# math text
```

```
title(r'$\alpha > \beta$')
```

To make subscripts and superscripts use the underscore and caret symbols, as in

```
title(r'$\alpha_i > \beta^i$')
```

You can also use a large number of the TEX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`; see Appendix B for a complete list. The over/under subscript/superscript style is also supported. To write the sum of x_i from 0 to ∞ ($\sum_{i=0}^{\infty} x_i$), you could do

```
text(1, -0.6, r'$\sum_{i=0}^\infty x_i$')
```

The default font is *italics* for mathematical symbols. To change fonts, eg, to write 'sin' in a roman font, enclose the text in a font command, as in

```
text(1,2, r's(t) = $\cal{A}\rm{sin}(2 \omega t)$')
```

Here 's' and 't' are variable in italics font (default), 'sin' is in roman font, and the amplitude '$\mathcal{A}$' is in caligraphy font. The fonts `\cal`, `\rm`, `\it` and `\tt` are allowed.

Fairly complex TEX expressions render correctly; you can compare the expression

```
s = r'$\cal{R}\prod_{i=\alpha}^\infty a_i\rm{sin}(2 \pi f x_i)$'
```

rendered by TEX below and by matplotlib in Figure 3.10.

$$\mathcal{R}\prod_{i=\alpha}^{\infty} a_i \sin(2\pi f x_i) \tag{3.1}$$

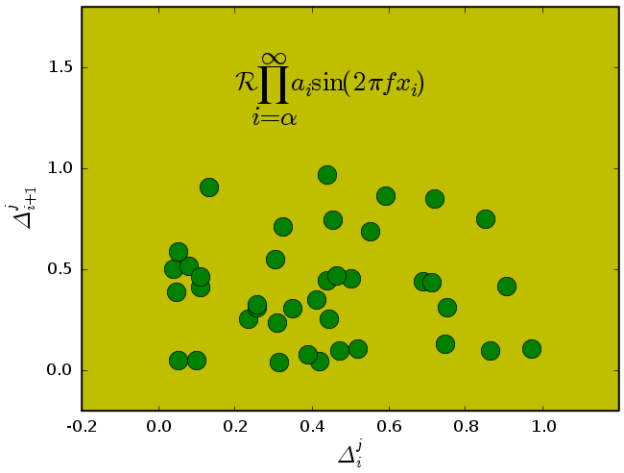

Figure 3.10: Incorpating TEX expressions into your figure; see Listing 3.6

Listing 3.6: Using mathtext; see Figure 3.10

```
from matplotlib import rcParams
rcParams['ps.useafm']=False
from pylab import *
```

```
# use a custom axes to provide room for the large labels used below
ax = axes([.2, .2, .7, .7], axisbg='y')

# generate some random symbols to plot
x = rand(40)
plot(x[:-1], x[1:], 'go', markeredgecolor='k', markersize=14)

# this is just a made up equation that has nothing to do with the
# plot!
s = r'$\cal{R}\prod_{i=\alpha}^\infty a_i\rm{sin}(2 \pi f x_i)$'
text(0.2, 1.2, s, fontsize=20)
axis([-0.2, 1.2, -0.2, 1.8])

# subscripts, superscripts and groups with {} are supported
```

usetex

If you have LaTeX, ghostscript, and dvipng installed on your computer, matplotlib can use LaTeXto perform all of the text layout in your figures. To enable this option, set `text.usetex : True` in your rc settings. For more information and examples, see http://www.scipy.org/Cookbook/Matplotlib/UsingTex .

3.7 Images

matplotlib provides support for working with raw image data in numerix arrays. Currently, there is no support for loading image data from image files such as PNG, TIFF or JPEG, though this is on the TODO list. If you need to load data from existing image files, one good solution is to use The Python Imaging Library to load the data and convert this to a numerix array - see Recipe 9.4.1. The following examples will assume you have your image data loaded into a numerix array, either luminance (MxN), RGB (MxNx3) or RGBA (MxNx4).

3.7.1 Axes images

An axes image is created with `im = imshow(X)` where *X* is a numerix array an *im* is a `matplotlib.image.AxesImage` instance. The image is rescaled to fit into the current axes box. Here is some example code to display an image

```
# create a random MxN numerix array and plot it as an axes image
from pylab import *
X = rand(20,20)
im = imshow(X)
```

`imshow` a command in the `pylab` interface. This is a thin wrapper of the `matplotlib.Axes.imshow` method, which can be called from any Axes instance, eg `ax.imshow(X)`.

There are two parameters that determine how the image is resampled into the axes bounding box: *interpolation* and *aspect*. The following *interpolation* schemes are available: *bicubic, bilinear, blackman100, blackman256, blackman64, nearest, sinc144, sinc256, sinc64, spline16, and spline36*. The default interpolation method is given by the value of `image.interpolation` in your `matplotlibrc` file. *aspect* can be either *equal*, *auto*, or some number, which will constrain the aspect ratio of the image. The default aspect setting is given by the value of the rc parameter `image.aspect`.

The full syntax of the `imshow` command is

```
imshow(X,                    # the numerix array
       cmap = None,          # the matplotlib.colors.Colormap instance
       norm = None,          # the normalization instance
       aspect=None,          # the aspect setting
       interpolation=None,   # the interpolation method
```

```
        alpha=1.0,            # the alpha transparency value
        vmin = None,          # the min for image scaling
        vmax = None,          # the max for image scaling
        origin=None):         # the image origin
```

When *None*, these parameters will assume a default value, in many cases determined by the rc setting. The meaning of *cmap, norm, vmin, vmax, and origin* will be explained in sections below.

The following shows a simple command which creates an image using bilinear interpolation, shown in Figure 3.11.

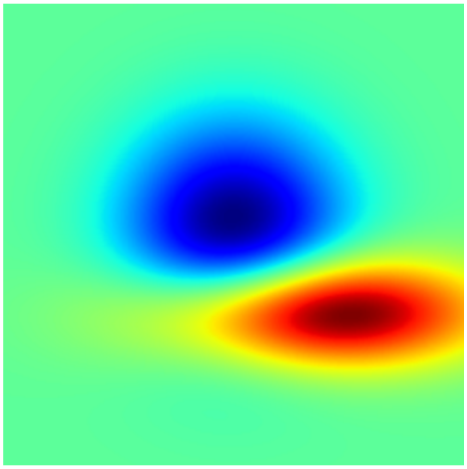

Figure 3.11: Simple axes image; code in Listing 3.7

Listing 3.7: Axes images; see Figure 3.11

```
from pylab import *

delta = 0.025
# generate a mesh of x and y vectors
x = y = arange(-3.0, 3.0, delta)
X, Y = meshgrid(x, y)
# create 2D gaussian distributions
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)

# plot the difference of Gaussians with blinear interpolation
im = imshow(Z2-Z1, interpolation='bilinear')
axis('off')
```

You can create an arbitrary number of axes images inside a single axes, and these will be composed via alpha blending. However, if you want to blend several images, you must make sure that the `hold` state is *True* and that the alpha of the layered images is less than 1.0; if alpha=1.0 then the image on top will totally obscure the images below. Because the image blending is done using antigrain (regardless of your backend choice), you can blend images even on backends which don't support alpha (eg, postscript). This is because the alpha blending is done in the frontend and the blended image is transferred directly to the backend as an RGB pixel array. See Recipe 9.4.2 for an example of how to layer images.

3.7.2 Figure images

Often times you want to be able to look at your raw image data directly, without interpolation. This is the function of figure images, which do a pixel-by-pixel transfer of your image data to the figure canvas [1]. Figure images are drawn first, and thus can become the background of other matplotlib drawing commands.

In the pylab interface, figure images are created with the `figimage` command, which unlike `imshow`, does not accept an *interpolation* or *aspect* keyword argument because no image resampling is used. If the pixel extent of the figure image extends beyond the figure canvas, the image will simply be truncated. The basic syntax is `figimage(X, xo=0, yo=0)` where X is luminance (MxN), RGB (MxNx3) or RGBA (MxNx4) numerix array and *xo*, *yo* are pixel offsets from the origin (see Section 3.7.4). You can use `figimage` to create a figure image that fills the entire canvas with no x or y offsets, or you can make multiple calls to `figimage` with different x and y offsets to create a mosaic of images, as shown in Recipe 9.4.3.

The full syntax of the `figimage` command is

```
figimage(X,              # the numerix array
         xo=0,           # the x offset
         yo=0,           # the y offset
         alpha=1.0,      # the alpha transparency
         norm=None,      # the matplotlib.colors.normalization instance
         cmap=None,      # the matplotlib.colors.Colormap instance
         vmin=None,      # the min for image scaling
         vmax=None,      # the max for image scaling
         origin=None)    # the image origin
```

The *cmap, norm, vmin, vmax* and *origin* arguments are explained in the sections below.

`pylab.figimage` is a thin wrapper of `matplotlib.figure.figimage` and you can generate figure images directly with the pythonic API using `fig.figimage(X)` where fig is a Figure instance.

3.7.3 Scaling and color mapping

In addition to supporting raw image RGB and RGBA formats, matplotlib will scale and map luminance data for MxN float (luminance) arrays. The conversion from luminance data to RGBA occurs in two steps: scaling and color mapping.

Scaling is the process of normalizing an MxN floating point array to the 0,1 interval, by mapping *vmin* to 0.0 and *vmax* to 1.0, where *vmin* and *vmax* are user defined parameters. If either are *None*, the min and max of the image data will be used, respectively. Scaling is handled by a `matplotlib.colors.normalization` instance, which defaults to `normalization(vmin=None, vmax=None)` - ie, the default is to scale the image so that the minimum of the luminance array is zero and the maximum of the luminance array is one.

Typically, you will not create a normalization instance yourself, but may set *vmin* or *vmax* in the keyword arguments of the image creation function. In this case, a normalization instance is created for you, and your vmin, vmax settings are applied. If you do supply a normalization instance for the *norm* argument, *vmin* and *vmax* will be ignored. See Table 3.7 for some examples of image normalization commands and their interpretation.

command	interpretation
`>> imshow(X)`	$X \leq \min(X) \rightarrow 0$ and $X \geq \max(X) \rightarrow 1$
`>> imshow(X, vmax=10)`	$X \leq \min(X) \rightarrow 0$ and $X \geq 10 \rightarrow 1$
`>> imshow(X, vmin=0, vmax=10)`	$X \leq 0 \rightarrow 0$ and $X \geq 10 \rightarrow 1$
`>> anorm=normalize(2,8)`	
`>> imshow(X, norm=anorm)`	$X \leq 2 \rightarrow 0$ and $X \geq 8 \rightarrow 1$

Table 3.7: Example image normalization commands and their interpretation

[1] If you want a resampled image to occupy the full space of the figure canvas, you can achieve this by specifying a custom axes that fills the figure canvas `axes([0, 1, 0, 1])` and using `imshow`.

Once the luminance data are normalized, they color mapper transforms the normalized data to RGBA using a `matplotlib.colors.Colormap` instance. Common colormaps are defined in `matplotlib.cm`, including `cm.jet` and `cm.gray`. If the cmap argument to an image command is *None*, the default is given by he rc parameter `image.cmap`.

The keyword arguments *cmap, norm, vmin, vmax* control color mapping and scaling in the image construction commands. Once the images have been created, several commands exist to interactively control the color map of the current image. Like the current figure (`gcf`) and the current axes (`gca`), matplotlib keeps track of the current image (`gci`) to determine which image to apply the commands which affect image properties. To interactively set the image normalization limits, use `clim(vmin=None, vmax=None)`, where *vmin* and *vmax* have the same meaning as above. To interactively change the colormap, use `jet` or `gray` (More colormaps and colormap commands are planned).. These latter commands not only change the colormap of the current image, they also set the default for future images.

For quantitative plotting of pseduocolor images, use the `colorbar` function to provide a colorbar associated with the image, Here is an example interactive session controlling image scaling and color mapping with a colorbar

```
>>> imshow(X)    # plot the luminance image X
>>> clim(-1,2)   # scale the image
>>> jet()        # use colormap jet
>>> colorbar()   # add a colorbar to the current axes
>>> gray()       # use grayscale; image and colorbar are updated
```

The image scaling and color mapping are handled by the mixin base class `matplotlib.colors.ScalarMappable`.

3.7.4 Image origin

Depending on your data, it may be more natural to plot your data with the image origin up (`X[0,0]` is upper left) or down (`X[0,0]` is lower left). matplotlib supports these two modes with the *origin* parameter, which can be supplied as an optional keyword argument to the image commands `imshow` and `figimage` with the default set by the rc parameter `image.origin`. To plot an image with the origin in the upper left, pass `origin='upper'` and with the image in the lower left, pass `origin='lower'`, as shown in Figure 3.12.

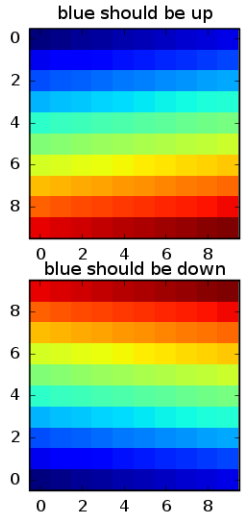

Figure 3.12: Controlling the image origin with the *origin* keyword argument to `imshow` and `figimage`; see Listing 3.8.

Listing 3.8: Setting the image origin; see Figure 3.12

```
from pylab import *

x = arange(100.0); x.shape = 10,10

subplot(211)
title('blue should be up')
imshow(x, origin='upper', interpolation='nearest')

subplot(212)
title('blue should be down')
imshow(x, origin='lower', interpolation='nearest')
```

3.8 Bar charts, histograms and errorbar plots

Use the `bar` function to create simple bar plots. The simplest form of this function is simply `bar(x,y)` which creates bars with their left edge at *x* and height *y*. There are a number of options to support more sophisticated bar plots, including stacked bar plots and bar plots with errorbars. The signature of the bar method is

```
    def bar(left, height, width=0.8, bottom=0,
            color='b', yerr=None, xerr=None,
            ecolor='k', capsize=3
            ):
```

3.8.1 Broken bar charts

matplotlib provides a collection `matplotlib.collections.BrokenBarHCollection`, and a wrapper function `broken_barh` to facilitate plotting a series of `(xmin, xwidth)` ranges with horizontal bars. This functionality is typically used in plotting temporal data, when you want to visually represent the windows of time where a certain feature is available, etc... The signature of the class and the wrapper function are

```
    def broken_barh(self, xranges, yrange, **kwargs):
        """
        A collection of horizontal bars spanning yrange with a sequence of
        xranges

        xranges : sequence of (xmin, xwidth)
        yrange  : (ymin, ywidth)

        kwargs are PatchCollection properties
        """
```

The collection properties are controlled with standard `matplotlib.collections.PatchColllection` kwargs

Property	*Value*
edgecolors	a single edge colors or a sequence
facecolors	a single facecolor or a sequence
linewidths	a single linewidth or a sequence

Table 3.8: The broken bar collection properties

Any of these arguments can be a single object, like `facecolors='blue'` in which case all bars will be homogeneous on that property, or a sequence, eg `facecolors=('blue', 'red', 'green')` which will capply different colors to the bars.

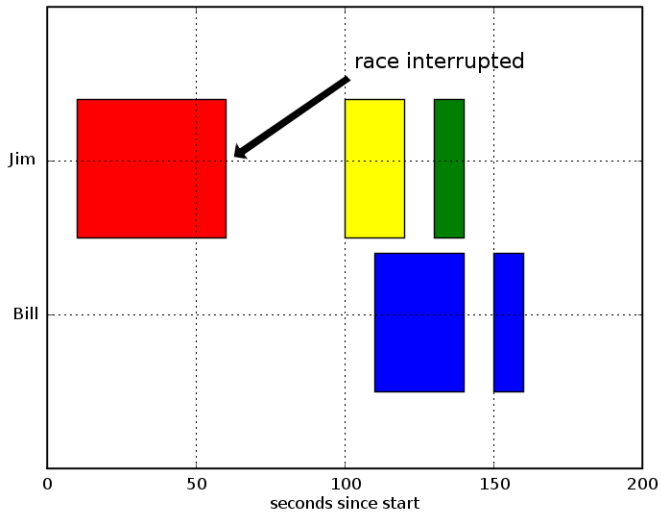

Figure 3.13: This graph simulates data in which a race was interrupted and the bars show the time periods when the race was on and when it was interrupted; code is in Listing 3.9. The annotation command used here is discussed in Section 3.6.5

Listing 3.9: Use the broken_barh to make horizontal broken bar plots, eg to represent intervals where something is turned on or off; see Figure 3.13.

```
from pylab import figure, show, nx

fig = figure()
ax = fig.add_subplot(111)
ax.broken_barh([ (110, 30), (150, 10) ] , (10, 9), facecolors='blue')
ax.broken_barh([ (10, 50), (100, 20),  (130, 10)] , (20, 9),
               facecolors=('red', 'yellow', 'green'))
ax.set_ylim(5,35)
ax.set_xlim(0,200)
ax.set_xlabel('seconds since start')
ax.set_yticks([15,25])
ax.set_yticklabels(['Bill', 'Jim'])
ax.grid(True)
ax.annotate('race interrupted', (61, 25),
            xytext=(0.8, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16,
            horizontalalignment='right', verticalalignment='top')
```

3.9 Polar charts

matplotlib has a `PolarAxes` class and a `polar` function in the pylab interface. Not every matplotlib polar function is supported on polar axes, but most are: you can do polar line plots, bar charts, scatter plots, area fills, text and legends.

To create a `PolarSubplot`, pass the `polar=True` keyword argument to the axes or subplot creation fuinction you are using, eg to create a

```
subplot(211, polar=True)
```

or

```
axes([left, bottom, width, height], polar=True)
```

The view limits (eg `xlim` and `ylim`) apply to the lower left and upper right of the rectangular box that surrounds to polar axes. Eg if you have

```
r = arange(0,1,0.01)
theta = 2*pi*r
```

the lower left corner is $5/4\pi, \sqrt{2}$ and the upper right corner is $1/4\pi, \sqrt{2}$.

To customize the radial and angular gridding, ticking and formatting, a few helper methods of the polar axes (with corresponding pylab interface wrappers) are provided.

To customize the radial grids, the PolarAxes has a method `set_rgrids` (the pylab method is simply `rgrids`).

```
def set_rgrids(self, radii, labels=None, angle=22.5, rpad=0.05, **kwargs):
```

This method will set the radial locations and labels of the radial grids. The `labels` will appear at radial distances `radii` at the specified `angle`. `labels`, if not `None`, is a `len(radii)` list of strings of the labels to use at each angle. If `labels` is `None`, the radial tick formatter `ax.rformatter` will be used. `rpad` is a fraction of the max of `radii` which will pad each of the radial labels in the radial direction. You can use the additional keyword arguments in `kwargs` to pass in `matplotlib.text.Text` properties to customize the text labels.

Similarly, to set and format the angular gridlines and labels, use the `set_thetagrids` function. See the class documentation – the signature is similar to the rgrids method discussed above.

The `PolarAxes` will clip all objects to the circular polygon which comprises the Axes bounding polygon, and you can use the `set_rmax` method to set the maximum radius and data that will be visible.

Listing 3.10: A simple polar line plot; see Figure 3.14. The matplotlib source distribution has addition examples `polar_scattter.py`, `polar_bar.py` and `polar_legend.py` which show how to make additional polar plot types.

```
#!/usr/bin/env python
import matplotlib.numerix as nx
from pylab import figure, show, rc

# radar green, solid grid lines
rc('grid', color='#316931', linewidth=1, linestyle='-')
rc('xtick', labelsize=15)
rc('ytick', labelsize=15)

# force square figure and square axes looks better for polar, IMO
fig = figure(figsize=(8,8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True, axisbg='#d5de9c')

r = nx.arange(0, 3.0, 0.01)
theta = 2*nx.pi*r
ax.plot(theta, r, color='#ee8d18', lw=3)
ax.set_rmax(2.0)

ax.set_title("And there was much rejoicing!", fontsize=20)
```

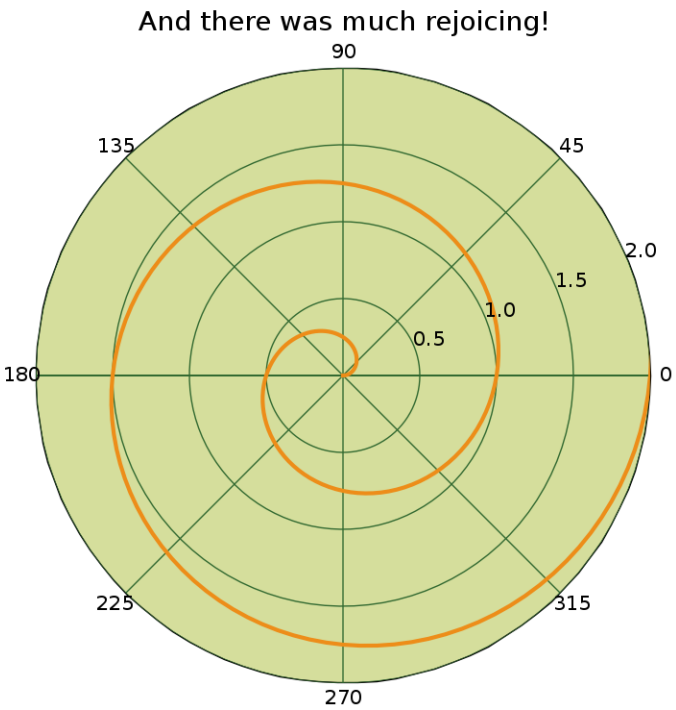

Figure 3.14: A polar line chart; code is in Listing 3.10.

3.10 Pseudocolor and scatter plots

3.11 Spectral analysis

matplotlib provides a number of MATLAB™ compatible functions for computing and plotting spectral analysis results. All of them are based on Welch's Averaged Periodogram Method [Bendat and Piersol, 1986] using the numerix `fft` method for the fast fourier transforms. The spectral plotting functions are `psd` for the power spectral density, `csd` for the cross spectral density, and `cohere` for the coherence (normalized cross spectral density).

```
# signature and defaults for arguments to a typical
# spectral analysis function
def psd(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
        window=mlab.window_hanning, noverlap=0):
```

In addition to the time series arguments x/y, these functions take a number of optional parameters. The averaged periodogram method chops the time series into *NFFT* length segments which overlap by *noverlap* samples. The default values are `NFFT=256` and `noverlap=0`. Each of the functions will compute the spectral analysis and then generate a plot window with frequency on the x-axis - if you want the frequency axis to be properly scaled, you should provide the sampling frequency *Fs*.

Each of the segments will be detrended and windowed before the fft, according to the values of *detrend* and *window*. Unlike MATLAB™, in which these arguments are strings, in matplotlib they are functions. Several helper functions are provided in matplotlib.mlab for detrending and windowing:

- `mlab.detrend_none` - no detrending

- `mlab.detrend_mean` - remove the mean of each segment before fft
- `mlab.detrend_linear` - remove the best fit line of each segment before fft
- `mlab.window_none` - no windowing
- `mlab.window_hanning` - multiply each segment by a Hanning window

An example power spectra calculation is shown in Listing 1.1 and the output in Figure 1.2.

You can create a spectrogram with the `specgram` function. `specgram` splits the data into NFFT length segments and plots the instantaneous power in each segment along the y axis using a pseudocolor plot, unlike `psd` which averages the power across each segment.

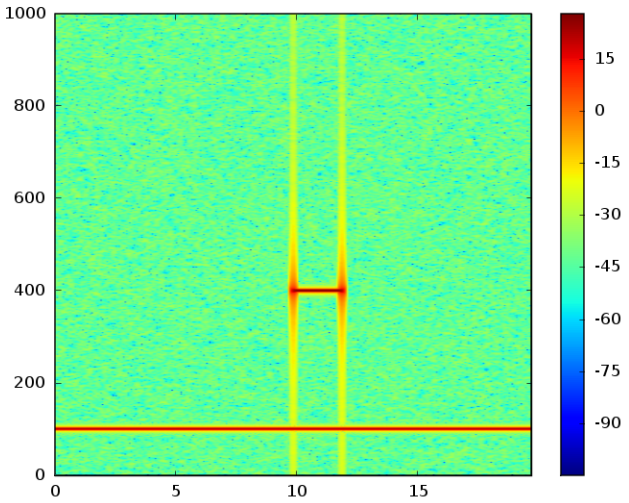

Figure 3.15: A spectrogram generated by Listing 3.11

Listing 3.11: Instantaneous power spectra with `specgram`; see Figure 3.15

```
from pylab import *

dt = 0.0005
t = arange(0.0, 20.0, dt)
# a 100 Hz signal
s1 = sin(2*pi*100*t)

# create a transient "chirp" at 400 Hz
s2 = 2*sin(2*pi*400*t)
mask = where(logical_and(t>10, t<12), 1.0, 0.0)
s2 = s2 * mask

# add some noise into the mix
nse = 0.01*randn(len(t))

x = s1 + s2 + nse # the signal
NFFT = 1024       # the length of the windowing segments
Fs = int(1.0/dt)  # the sampling frequency
```

```
# Pxx is the segments x freqs array of instantaneous power, freqs is
# the frequency vector, bins are the centers of the time bins in which
# the power is computed, and im is the matplotlib.image.AxesImage
# instance
Pxx, freqs, bins, im = specgram(x, NFFT=NFFT, Fs=Fs, noverlap=900)
colorbar()
```

3.12 Axes properties

3.13 Legends and tables

3.14 Navigation

matplotlib comes with two navigation toolbars for the graphical user interfaces: classic and toolbar2. You can use these to change the view limits of the axes in the figure. toolbar2 superceeds classic and was designed to overcome shortcomings of the classic toolbar. The default toolbar is determined by the `toolbar` parameter in `matplotlibrc`.

3.14.1 Classic toolbar

You can pan and zoom on the X and Y axis for any combination of the axes that are plotted. If you have a wheel mouse, you can move bidirectionally by scrolling the wheel over the controls. For examples, the wheel mouse can be used to pan left or right by scrolling over either of the left arrow or right arrow buttons, so you never have to move the mouse to pan the x-axis left and right. If you don't have a wheel mouse, buy one!

The left widget that says 'All' on the controls on the bottom of Figure 3.16 is a drop down menu used to select which axes the controls affect. You can select all, none, single, or combinations of axes. The first set of 4 controls are used to pan left, pan right, zoom in and zoom out on the x axes. The second set are used to pan up, pan down, zoom in and zoom out on the y axes. The remaining buttons are used to redraw the figure, save (PNG or JPEG) the figure, or to close the figure window.

3.14.2 toolbar2

The toolbar2 buttons (see Figure 3.17 behave very differently from the classic the classic matplotlib toolbar (else why introduce a new one!) despite the visual similarity of the forward and back buttons.

The `Forward` and `Back` buttons are akin to the web browser forward and back buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click 'back' on your web browser before visiting a new page. Nothing happens. `Home` always takes you to the first view. For `Home`, `Forward` and `Back`, think web browser where data views are web pages. Use the `Pan/Zoom` and `Zoom to rectangle` buttons, discussed below, to define new views.

The `Pan/Zoom` button has two modes: pan and zoom. Click this toolbar button to activate this mode. Then put your mouse somewhere over an axes.

- Mode 1: Press the left mouse button and hold it, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press 'x' or 'y' while panning, the motion will be contrained to the x or y axis, respectively

- Mode 2: Press the right mouse button, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the yaxis and up/down motions. The point under your mouse when you begin the zoom should remain in place, allowing you

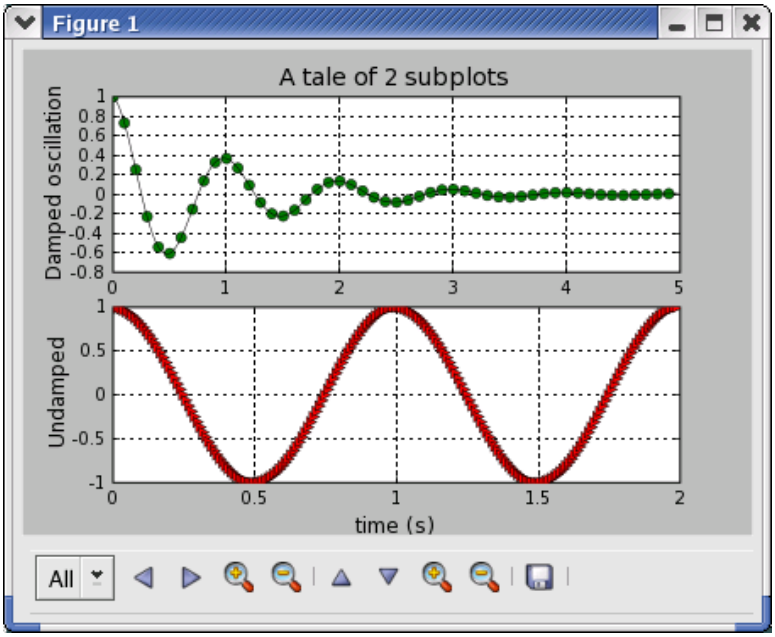

Figure 3.16: The classic toolbar, discussed in Section 3.14.1

to zoom to an arbitrary point in the figure. You can use the modifier keys 'x', 'y' or 'CONTROL' to constrain the zoom to the x axes, the y axes, or aspect ratio preserve, respectively.

The `Zoom to rectangle` button: Click this toolbar button to activate this mode. Put your mouse somewhere over and axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined. There is also an experimental 'zoom out to rectangle' in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.

The `Save` button: click this button to launch a file save dialog. All the *Agg backends know how to save the following image types: PNG, PS, EPS, SVG. There is no support currently in Agg for writing to JPEG, TIFF (the regular wx and gtk backends handle these types). It is possible to use matplotlib/agg + PIL to convert agg images to one of these other formats if required. I can provide a recipe for you. I prefer PNG over JPG and TIFF, which is why I haven't worked too hard to include these other image formats in agg.

3.15 Event handling

When visualizing data, it's often helpful to get some interactive input from the user. All graphical user interfaces (GUIs) provide event handling to determine things like key presses, mouse position, and button clicks. matplotlib supports a number of GUIs, and provides an interface to the GUI event handling via the `mpl_connect` and `mpl_disconnect` methods of the pylab interface (API users will probably want to use their GUIs event handling directly, but do have the option of using their `FigureCanvas.mpl_connect` method).

matplotlib uses a callback event handling mechanism. The basic idea is that you register an event that you want to listen for, and the figure canvas, will call a user defined function when that event occurs. For example, if you want to know where the user clicks a mouse on your figure, you could define a function

```
# this function will be called with every click
def click(event):
    print 'you clicked', event.x, event.y
```

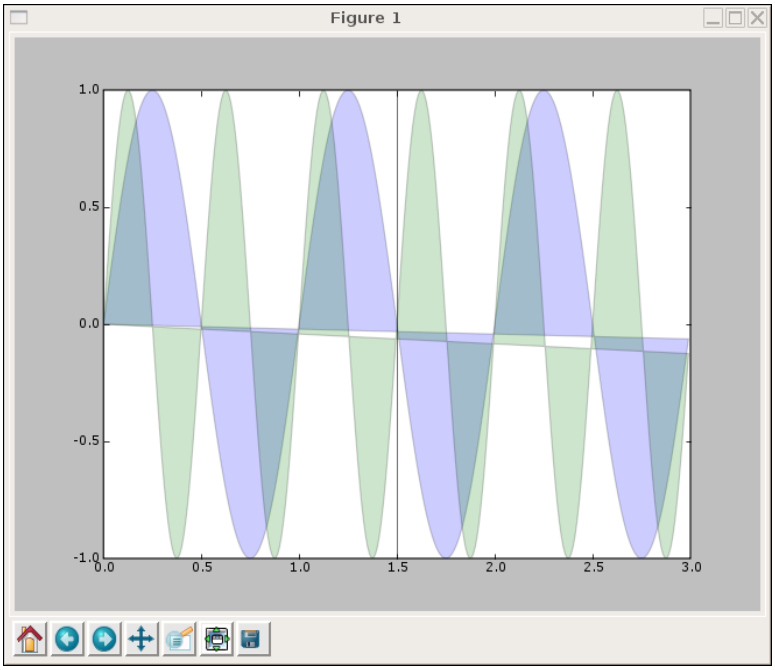

Figure 3.17: The newfangled toolbar2, discussed in Section 3.14.2

```
# register this function with the event handler
cid = connect('button_press_event', click)}.
```

Then whenever the user clicks anywhere on the figure canvas, your function will be called and passed a `matplotlib.backend_bases.Mp` instance. The event instance will have the following attributes defined.

Property	*Meaning*
x	x position - pixels from left of canvas
y	y position - pixels from bottom of canvas
button	button pressed None, 1, 2, 3
inaxes	the Axes instance if mouse is over axes (or None)
xdata	x coord of mouse in data coords (None if mouse isn't over axes)
ydata	y coord of mouse in data coords (None if mouse isn't over axes)
name	The string name of the event
canvas	The FigureCanvas instance the event occured in
key	The key press if any, eg 'a', 'b', '1'. Also records 'control and 'shift'

Table 3.9: The event attributes

You can connect to the following events: 'button_press_event', 'button_release_event', 'motion_notify_event', 'key_press_event', and 'key_release_event'.

You can connect multiple event handlers, and later disconnect them if you want with the disconnect function

```
# register this function with the event handler
def click1(event): pass
def click2(event): pass
cid1 = connect('key_press_event', click1)}.
```

```
cid2 = connect('key_press_event', click2)}.

...later on....
disconnect(cid1)  # now only click2 is connected
```

Here's an example to get the mouse location in data coordinates as the mouse moves

```
# Connect to the mouse move event and print the location of the mouse
# in data coordinates if the mouse is over an axes
from pylab import *

plot(arange(10))

def on_move(event):
    # get the x and y pixel coords
    x, y = event.x, event.y

    if event.inaxes:
        print 'data coords', event.xdata, event.ydata

connect('motion_notify_event', on_move)


show()
```

3.16 Customizing plot defaults

Chapter 4

Font finding and properties

`matplotlib.fonts.font_manager` is module for finding, managing, and using fonts across-platforms. This module provides a single FontManager that can be shared across backends and platforms. The findfont() method returns the best TrueType (TTF) font file in the local or system font path that matches the specified FontProperties. The FontManager also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification (http://www.w3.org/TR/1998/REC-CSS2-19980512). Future versions may implement the Level 2 or 2.1 specifications.

The `font.family` property has five values: 'serif' (e.g. Times), 'sans-serif' (e.g. Helvetica), 'cursive' (e.g. Zapf-Chancery), 'fantasy' (e.g. Western), and 'monospace' (e.g. Courier). Each of these font families has a default list of font names in decreasing order of priority associated with them. You describe which family you want by choosing, eg, `family='serif'`, and the font manager will search the font.serif list looking for one of the named fonts on your system. The lists are user configurable, and reside in your `matplotlibrc`.

This allows you to choose your family in your matplotlib script and the font manager will try and find the best font no matter which platform you run on.

- The `font.style` property has three values: normal (or roman), italic or oblique. The oblique style will be used for italic, if it is not present.
- The `font.variant` property has two values: normal or small-caps. For TrueType fonts, which are scalable fonts, small-caps is equivalent to using a font size of 'smaller', or about 83
- The `font.weight` property has effectively 13 values: normal, bold, bolder, lighter, 100, 200, 300, ..., 900. Normal is the same as 400, and bold is 700. bolder and lighter are relative values with respect to the current weight.
- The `font.stretch` property has 11 values: ultra-condensed, extra-condensed, condensed, semi-condensed, normal, semi-expanded, expanded, extra-expanded, ultra-expanded, wider, and narrower. This property is not currently implemented.
- The `font.size` property is the default font size for text, given in pts. 12pt is the standard value. Special text sizes for tick labels, axes, labels, title, etc. can be defined relative to `font.size` using the following values: xx-small, x-small, small, medium, large, x-large, xx-large, larger, or smaller. Special text sizes can also be an absolute size, given in pts.

Here is an example using the font properties to illustrate the different fonts

```
from matplotlib import rcParams
rcParams['ps.useafm']=False
from pylab import *

subplot(111, axisbg='w')
```

```
alignment = {'horizontalalignment':'center', 'verticalalignment':'center'}

###  Show family options

family = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']
t = text(-0.8, 0.9, 'family', size='large', **alignment)
yp = [0.7, 0.5, 0.3, 0.1, -0.1, -0.3, -0.5]
for k in range(5):
    if k == 2:
        t = text(-0.8, yp[k], family[k], family=family[k],
                 name='Script MT', **alignment)
    else:
        t = text(-0.8, yp[k], family[k], family=family[k], **alignment)

###  Show style options
style  = ['normal', 'italic', 'oblique']
t = text(-0.4, 0.9, 'style', **alignment)
for k in range(3):
    t = text(-0.4, yp[k], style[k], family='sans-serif', style=style[k],
             **alignment)

###  Show variant options
variant= ['normal', 'small-caps']
t = text(0.0, 0.9, 'variant', **alignment)
for k in range(1):
    t = text( 0.0, yp[k], variant[k], family='serif', variant=variant[k],
              **alignment)

###  Show weight options
weight = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
t = text( 0.4, 0.9, 'weight',  **alignment)
for k in range(7):
    t = text( 0.4, yp[k], weight[k], weight=weight[k],
              **alignment)

###  Show size options
size  = ['xx-small', 'x-small', 'small', 'medium', 'large',
         'x-large', 'xx-large']
t = text( 0.8, 0.9, 'size', **alignment)
for k in range(7):
    t = text( 0.8, yp[k], size[k], size=size[k],
              **alignment)

x = 0
###  Show bold italic
t = text(x, 0.1, 'bold italic', style='italic',
         weight='bold', size='x-small',
         **alignment)
t = text(x, 0.2, 'bold italic',
         style = 'italic', weight='bold', size='medium',
         **alignment)
t = text(x, 0.3, 'bold italic',
         style='italic', weight='bold', size='x-large',
         **alignment)
axis([-1, 1, 0, 1])
savefig('../figures/fonts_demo_kw.png')
```

```
savefig('../figures/fonts_demo_kw.eps')
show()
```

Chapter 5

Collections

Chapter 6

Tick locators and formatters

The `matplotlib.ticker` module contains classes to support completely configurable tick locating and formatting. Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom locators an formatters.

6.1 Tick locating

Choosing tick locations and formats is a difficult and essential part of making nice looking graphs. The `matplotlib.ticker` module divides the workload between two bases classes: the locators and the formatters. Each axis (eg the xaxis and yaxis) has a major and minor tick locator and a major and minor tick formatter. The default minor tick locators always return the empty list, ie, there are no minor ticks. Each of these can be set independently, and it is easy for the user to create and plug-in a custom tick locator or formatter.

The `matplotlib.ticker.Locator` class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and choosing the tick locations. The most generally useful tick locator is `MultipleLocator`. You initialize this with a base, eg 10, and it picks axis limits and ticks that are multiples of your base. The class `AutoLocator` contains a `MultipleLocator` instance, and dynamically updates it based upon the data and zoom limits. This should provide much more intelligent automatic tick locations both in figure creation and in navigation than in prior versions of matplotlib. See Tables 6.1 and 6.2 for a summary of the basic and date tick locators.

Class	*Summary*
NullLocator	No ticks
IndexLocator	locator for index plots (eg where `x = range(len(y))`
LinearLocator	evenly spaced ticks from min to max
LogLocator	logarithmically ticks from min to max
MultipleLocator	ticks and range are a multiple of base; either integer or float
AutoLocator	choose a MultipleLocator and dynamically reassign

Table 6.1: The basic tick locators

You can define your own locator by deriving from Locator. You must override the `__call__` method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits. If you want to override the default locator, use one of the above or a custom locator and pass it to the x or y axis instance. The relevant methods are

```
ax.xaxis.set_major_locator( xmajorLocator )
ax.xaxis.set_minor_locator( xminorLocator )
```

Class	*Summary*
MinuteLocator	locate minutes
HourLocator	locate hours
DayLocator	locate specifed days of the month
WeekdayLocator	Locate days of the week, eg MO, TU
MonthLocator	locate months, eg 7 for july
YearLocator	locate years that are multiples of base
RRuleLocator	locate using a matplotlib.dates.rrulewrapper. The rrulewrapper is a simple wrapper around a dateutils.rrule https:

Table 6.2: The tick locators specialized for date plots; these reside in the `matplotlib.dates` module

```
ax.yaxis.set_major_locator( ymajorLocator )
ax.yaxis.set_minor_locator( yminorLocator )
```

The default minor locator is the NullLocator, eg no minor ticks on by default.

6.2 Tick formatting

Tick formatting is the process of converting the numeric tick location into a suitable string, and is controlled by classes derived from `matplotlib.ticker.Formatter`. The formatter operates on a single tick value (and its tick position) and returns a string to the axis. The tick formatters are summarized in Table 6.3.

Class	*Summary*
NullFormatter	no labels on the ticks
FixedFormatter	set the strings manually for the labels
FuncFormatter	user defined function sets the labels
FormatStrFormatter	use a sprintf format string
IndexFormatter	cycle through fixed strings by tick position
ScalarFormatter	default formatter for scalars; autopick the fmt string
LogFormatter	formatter for log axes
DateFormatter	use an strftime string to format the date

Table 6.3: The tick formatting classes

You can derive your own formatter from the Formatter base class by simply overriding the __call__ method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods::

```
ax.xaxis.set_major_formatter( xmajorFormatter )
ax.xaxis.set_minor_formatter( xminorFormatter )
ax.yaxis.set_major_formatter( ymajorFormatter )
ax.yaxis.set_minor_formatter( yminorFormatter )
```

6.3 Example 1: major and minor ticks

In this example, the xaxis has major ticks that are multiples of 20 and minor ticks that are multiples of 5. The ticks are formatted with an integer format string formatter '%d'. The minor ticks are unlabelled (`NullFormatter`).

The `MultipleLocator` ticker class is used to place ticks on multiples of some base. The `FormatStrFormatter` uses a string format string (eg'%d' or '%1.2f' or '%1.1f cm') to format the tick.

Note that the pylab interface grid command changes the grid settings of the major ticks of the y and y axis together. If you want to control the grid of the minor ticks for a given axis, use for example `ax.xaxis.grid(True, which='minor')`. See Figure 6.1.

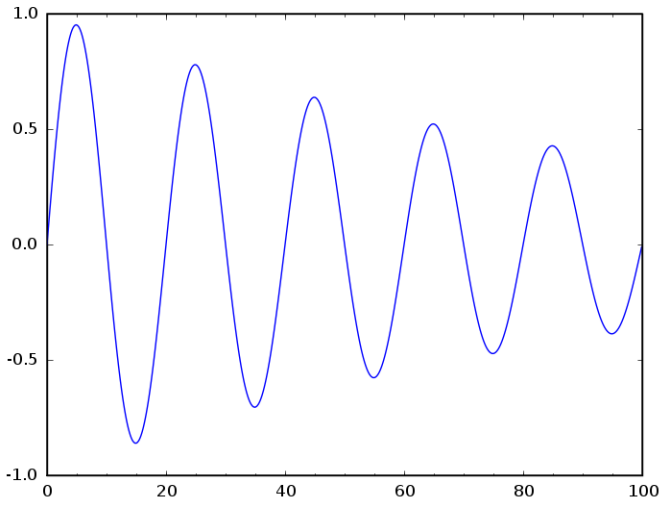

Figure 6.1: Creating custom major and minor tick locators and formatters; see Listing 6.1

Listing 6.1: Custom tickers and formatters; see Figure 6.1

```
from pylab import *
# import the tick locator and formatter classes
from matplotlib.ticker import MultipleLocator, FormatStrFormatter

majorLocator   = MultipleLocator(20)       # multiples of 20
majorFormatter = FormatStrFormatter('%d')  # integer format string
minorLocator   = MultipleLocator(5)        # multiples of 5

# my favorite plot!
t = arange(0.0, 100.0, 0.1)
s = sin(0.1*pi*t)*exp(-t*0.01)

ax = subplot(111)
plot(t,s)

# now just set the major and minor locators and formatters
ax.xaxis.set_major_locator(majorLocator)
ax.xaxis.set_major_formatter(majorFormatter)

#for the minor ticks, use no labels; default NullFormatter
ax.xaxis.set_minor_locator(minorLocator)
```

6.4 Example 2: date ticking

Making nice date/time plots requires custom tick locating and formatting. matplotlib converts all datetimes to days since 0001-01-01, and uses a floating point number to represent fractions of days. The functions `date2num` and `num2date` are used to convert back and forth between python datetimes and these floating point numbers.

The example below uses the `matplotlib.finance` module to get some historical quotes from yahoo's historical quotes server. The datetime start and end points are specified using a python's datetime module. Major ticks are on the months (`MonthLocator`) and minor ticks are on Mondays (`WeekdayLocator`). Only the major ticks are labelled, using a strftime format string (`DateFormatter`). Finally since the y axis is a stock price, a string formatter (`FormatStrFormatter`) is used to place dollar signs on the y tick labels.

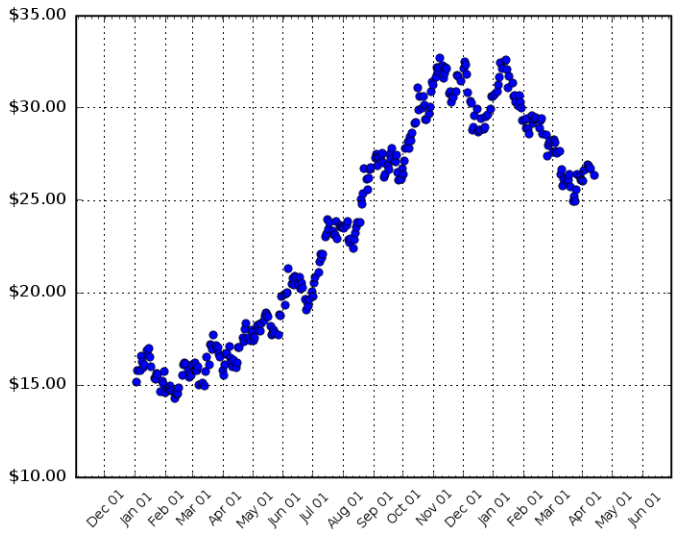

Figure 6.2: Providing custom tick locators and formatters for financial/date plots; see Listing 6.2.

Listing 6.2: Custom date tick locators and formatters; see Figure 6.2

```
import datetime
from pylab import *
from matplotlib.dates import MONDAY, SATURDAY
from matplotlib.finance import quotes_historical_yahoo
from matplotlib.dates import MonthLocator, WeekdayLocator, DateFormatter
from matplotlib.ticker import FormatStrFormatter

# the start and end date range for the financial plots
date1 = datetime.date( 2003, 1, 1 )
date2 = datetime.date( 2004, 4, 12 )

# the tick locators and formatters
mondays     = WeekdayLocator(MONDAY)        # every monday
months      = MonthLocator()                # every month
monthsFmt   = DateFormatter('%b %d')        # looks like May 01
dollarFmt   = FormatStrFormatter('$%0.2f') # dollars!

# get some financial data from the finance module
```

```
quotes = quotes_historical_yahoo(
    'INTC', date1, date2)
if not quotes:  raise SystemExit  # failsafe

# extract the date and opening prices from the quote tuples
dates = [q[0] for q in quotes]
opens = [q[1] for q in quotes]

# plot_date will choose a default date ticker and formatter
ax = subplot(111)
plot_date(dates, opens, markeredgecolor='k')

# but we'll override the default with our custom locators and
# formatters
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(monthsFmt)
ax.xaxis.set_minor_locator(mondays)

# format the y axis in dollars
ax.yaxis.set_major_formatter(dollarFmt)

# call autoscale to pick intelligent view limits based on our major
# tick locator
ax.autoscale_view()

# rotate the x labels for nicer viewing
labels = ax.get_xticklabels()
setp(labels, 'rotation', 45, fontsize=10)

grid(True)
```

Chapter 7

Interactive object picking

When interacting with a plot in a user interface window, it is often helpful to be able to select, or "pick" graphical elements with a mouse click or a lasso select. matplotlib provides a pick interface to select and inspect objects and their data though a pick event handler. Everything in the graph, from basic elements like Text, Line2D, Polygon to complex objects like Figure, Axes and Axis, are derived from the matplotlib Artist base class, and each artist defines a `pick` method that responds to mouse click events and fires a `pick_event` when the mouse click is over the artist. When selected, the codepick_event is fired off and clients who are registered to get callbacks on `pick_event` will be notified with a callback with the following signature

```
def onpick(event):
    pass

fig.canvas.mpl_connect('pick_event', onpick)
```

`event` is a `matplotlib.backend_bases.PickEvent` instance which has attributes

attribute	Description
mouseevent	the MouseEvent that generated the pick
artist	the artist picked

Table 7.1: The `PickEvent` attributes

In addition, certain matplotlib Artists, such as `Line2D` may attach additional type dependent attributes. For example, Line2D attaches `ind` which is a sequeunce of integers giving the indices into the data of the points which meet the pick criterion tolerance. These extra attributes are documented in the `Artist.pick` method, eg, from `matplotlib.lines.Line2D.pick`

```
    def pick(self, mouseevent):
        """
        If mouseevent is over data that satisifies the picker, fire
        off a backend_bases.PickEvent with the additional attribute "ind"
        which is a sequence of indices into the data that meet the criteria
        """
```

In order to enable picking on a given artist, you need to set the *pickeps* property, which take several forms. At the most basic level, it is simply a boolean `True|False` which will enable/disable picking for that artist and the artist defines a default rule for hit testing (eg for Text, if the mouse click occurs in the rectangular bounding box surrounding the text it will fire off a `pick_event`. Alternatively, one can specify a scalar value which is interpreted as a epsilon tolerance distance in printer's points; Line2D for example will determine the indicies of all the data points which fall within this epsilon tolerance distance. The list of valid picker values and types is given in the table below.

picker value	description
None	picking is disabled for this artist (default)
boolean	if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
float	if picker is a number it is interpreted as an epsilon tolerance in points and the the Artist will fire off an event if it's da
function	if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. hit,

Table 7.2: The valid types and their interpretation for the *picker* Artist property.

The example below shows how to enable picking on basic matplotlib Artists (Line2D, Rectangle and Text). Note that the picker property must be explicitly set either as a keyword argument to the plotting function that creates the Artist (eg `plot` or `bar`) or by setting the property of the Artist directly (eg `label.set_picker(True)`)

Listing 7.1: Basic picker event handling

```
from pylab import figure, show, nx
from matplotlib.lines import Line2D
from matplotlib.patches import Patch, Rectangle
from matplotlib.text import Text

fig = figure()
ax1 = fig.add_subplot(211)
ax1.set_title('click on points, rectangles or text', picker=True)
ax1.set_ylabel('ylabel', picker=True, bbox=dict(facecolor='red'))
line, = ax1.plot(nx.mlab.rand(100), 'o', picker=5)  # 5 points tolerance


ax2 = fig.add_subplot(212)

# pick the bars
bars = ax2.bar(range(10), nx.mlab.rand(10), picker=True)
for label in ax2.get_xticklabels():
    label.set_picker(True)   # make the xtick labels pickable

# this function will be called when one of the picker Artists is
# clicked on
def onpick1(event):
    if isinstance(event.artist, Line2D):
        thisline = event.artist
        xdata = thisline.get_xdata()
        ydata = thisline.get_ydata()
        ind = event.ind
        print 'onpick1 line:', zip(nx.take(xdata, ind), nx.take(ydata, ind))
    elif isinstance(event.artist, Rectangle):
        patch = event.artist
        print 'onpick1 patch:', patch.get_verts()
    elif isinstance(event.artist, Text):
        text = event.artist
        print 'onpick1 text:', text.get_text()

# now register your function to get a callback on a pick event
fig.canvas.mpl_connect('pick_event', onpick1)
show()
```

You can also define your own custom hit testing function and return custom data to the callback. This is very useful if you want to define your own distance metric or to decorate your plot objects with custom data and return it to the user through the pick infrastructure, eg to query plot objects for associated data. In the example below we define the

pick distance function to operate in data coordinate space rather than printer's points, and return additional metadata to the user, specifically the x and y coordinates of the pick point in data coordinates

Listing 7.2: Defining custom hit testing functions

```
from pylab import figure, show, nx

def line_picker(line, mouseevent):
    """
    find the points within a certain distance from the mouseclick in
    data coords and attach some extra attributes, pickx and picky
    which are the data points that were picked
    """
    if mouseevent.xdata is None: return False, dict()
    xdata = line.get_xdata()
    ydata = line.get_ydata()
    maxd = 0.05
    d = nx.sqrt((xdata-mouseevent.xdata)**2. + (ydata-mouseevent.ydata)**2.)

    ind = nx.nonzero(nx.less_equal(d, maxd))
    if len(ind):
        pickx = nx.take(xdata, ind)
        picky = nx.take(ydata, ind)
        props = dict(ind=ind, pickx=pickx, picky=picky)
        return True, props
    else:
        return False, dict()

def onpick2(event):
    print 'onpick2 line:', event.pickx, event.picky

fig = figure()
ax1 = fig.add_subplot(111)
ax1.set_title('custom picker for line data')
line, = ax1.plot(nx.mlab.rand(100), nx.mlab.rand(100), 'o', picker=line_picker)
fig.canvas.mpl_connect('pick_event', onpick2)

show()
```

7.1 Picking with a lasso tool

matplotlib provides a lasso tool to select a region of the plot, and some high performance hit testing functions to query whether a point or group of points is withing the polygon created by the lasso.

The basic lasso widget is defined in `matplotlib.widgets` and is initialized with an Axes instance, an `x,y` starting location, and a callback function. The widget will handle the mouse movement events and trigger a callback with the vertices of the lasso when the mouse button is released. A standard use case is to create the lasso on a button press event with the Axes in which the mouse press occurred and the x,y locations of the mouse press event.

```
    def onpress(self, event):
        if self.canvas.widgetlock.locked(): return
        if event.inaxes is None: return
        self.lasso = Lasso(event.inaxes, (event.xdata, event.ydata), self.callback)
        # acquire a lock on the widget drawing
        self.canvas.widgetlock(self.lasso)
```

The callback function will get the vertices of the lasso, and can use the `matplotlib.nxutils.points_inside_poly` function to test whether a list of x,y points are inside the lasso region or not.

```
    def callback(self, verts):
        # self.xys is a sequene of (x,y) data points and verts are
        # the x,y vertices of the lasso polygon
        ind = nx.nonzero(points_inside_poly(self.xys, verts))
        self.canvas.draw_idle()
        self.canvas.widgetlock.release(self.lasso)
        del self.lasso
```

In the example code, we allso manage the widget locking of the canvas so different widgets will not compete to draw on the canvas at the same time, eg a rectangular select widget and a lasso widget may both need to get a lock on the canvas at different times. A complete example is below

```
from matplotlib.widgets import Lasso
import matplotlib.mlab
from matplotlib.nxutils import points_inside_poly
from matplotlib.colors import colorConverter
from matplotlib.collections import RegularPolyCollection

from pylab import figure, show, nx

class Datum:
    colorin = colorConverter.to_rgba('red')
    colorout = colorConverter.to_rgba('green')
    def __init__(self, x, y, include=False):
        self.x = x
        self.y = y
        if include: self.color = self.colorin
        else: self.color = self.colorout


class LassoManager:
    def __init__(self, ax, data):
        self.axes = ax
        self.canvas = ax.figure.canvas
        self.data = data

        self.Nxy = len(data)

        self.facecolors = [d.color for d in data]
        self.xys = [(d.x, d.y) for d in data]

        self.collection = RegularPolyCollection(
            fig.dpi, 6, sizes=(100,),
            facecolors=self.facecolors,
            offsets = self.xys,
            transOffset = ax.transData)

        ax.add_collection(self.collection)

        self.cid = self.canvas.mpl_connect('button_press_event', self.onpress)

    def callback(self, verts):
        #ind = matplotlib.mlab._inside_poly_deprecated(self.xys, verts)
        ind = nx.nonzero(points_inside_poly(self.xys, verts))
```

```
        for i in range(self.Nxy):
            if i in ind:
                self.facecolors[i] = Datum.colorin
            else:
                self.facecolors[i] = Datum.colorout

        self.canvas.draw_idle()
        self.canvas.widgetlock.release(self.lasso)
        del self.lasso

    def onpress(self, event):
        if self.canvas.widgetlock.locked(): return
        if event.inaxes is None: return
        self.lasso = Lasso(event.inaxes, (event.xdata, event.ydata), self.callback)
        # acquire a lock on the widget drawing
        self.canvas.widgetlock(self.lasso)

data = [Datum(*xy) for xy in nx.mlab.rand(100, 2)]

fig = figure()
ax = fig.add_subplot(111, xlim=(0,1), ylim=(0,1), autoscale_on=False)
lman = LassoManager(ax, data)

show()
```

Chapter 8

Custom objects and units

matplotlib provides support for working with custom objects that know how to convert themselves to numeric-like objects. Suppose you want to plot a certain type of object (eg a python `datetime.date`) and you do not have access to the code to provide an array interface to it. This may be because the object is coming from a 3rd party or from some part of your API that you do not have control over. You could wrap it to put an array interface around it, but then it may not work properly with other parts of your code, again parts you may not have control over. To support this, matplotlib allows you to register object types with conversion interfaces.

As a simple example, matplotlib assumes dates are passed as floating point days since '0000-00-00'. To work with native python date objects, we need to provide a conversion interface that converts `datetime.date` objects to the floats matplotlib is expecting. The basic conversion interface is in `matplotlib.units`. Additionally, we can provide axis info to enable default axis labeling, tick locating and tick formatting. Here is the basic conversion interface required to support dates

Listing 8.1: Conversion interface for working with python dates

```
import matplotlib
from matplotlib.cbook import iterable, is_numlike
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):

    def axisinfo(unit):
        'return the unit AxisInfo'
        if unit=='date':
            majloc = dates.AutoDateLocator()
            majfmt = dates.AutoDateFormatter(majloc)
            return units.AxisInfo(
                majloc = majloc,
                majfmt = majfmt,
                label='date',
                )
        else: return None
    axisinfo = staticmethod(axisinfo)

    def convert(value, unit):
        if units.ConversionInterface.is_numlike(value): return value
        return dates.date2num(value)
    convert = staticmethod(convert)
```

```
    def default_units(x):
        'return the default unit for x or None'
        return 'date'
    default_units = staticmethod(default_units)


# Once we have provided the basic required interface, we need to
# register it with matplotlib so the matplotlib code knows how to
# convert date and datetime objects to matplotlib floats.

units.registry[datetime.date] = DateConverter()
units.registry[datetime.datetime] = DateConverter()
```

Now we are ready to use native python date objects with matplotlib. If we put the above in a modulke called `datesupport`, we can use datetime object in matplotlib code as follows

```
import datesupport # set up the date converters
import datetime
from matplotlib.dates import drange
from pylab import figure, show, nx


xmin = datetime.date(2007,1,1)
xmax = datetime.date.today()
delta = datetime.timedelta(days=1)
xdates = drange(xmin, xmax, delta)

fig = figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
ax.plot(xdates, nx.mlab.rand(len(xdates)), 'o')
ax.set_xlim(datetime.date(2007,2,1), datetime.date(2007,3,1))

fig.autofmt_xdate()
show()
```

In addition to custom types, in the same units infrastructure matplotlib supports unit types; that is, objects which know how to convert themselves to scalars given a unit argument. Because there is no standard python units package, we do not assume any particular form of the units. Rather, `matplotlib.axis.Axis` stores a `units` property (which can be an arbitrary type) and will pass this value of the units property to objects which are registered with the units registry. The units conversion interface provides the default do-nothing method

```
    def convert(obj, unit):
        """
        convert obj using unit.  If obj is a sequence, return the
        converted sequence.  The ouput must be a sequence of scalars
        that can be used by the numerix array layer
        """
        return obj
    convert = staticmethod(convert)
```

to support unit conversions. The implementor is free to ignore the `unit` argument which defaults to None. But if you set the `unit` property (eg to `inches`), this will be passed to your unit enabled objects registered with the units registry. Below is a toy example in which a unit enabled class `Foo` is plotted with different units (here the "units" are simple scalars 1.0, 2.0, etc... but in real code would likely be strings like 'inches', 'cm' or object instances like `inches`, `cm`....

```
from matplotlib.cbook import iterable
import matplotlib.units as units
import matplotlib.ticker as ticker
from pylab import figure, show, nx

class Foo:
    def __init__( self, val, unit=1.0 ):
        self.unit = unit
        self._val = val * unit

    def value( self, unit ):
        if unit is None: unit = self.unit
        return self._val / unit

class FooConverter:

    def axisinfo(unit):
        'return the Foo AxisInfo'
        if unit==1.0 or unit==2.0:
            return units.AxisInfo(
                majloc = ticker.IndexLocator( 4, 0 ),
                majfmt = ticker.FormatStrFormatter("VAL: %s"),
                label='foo',
                )

        else:
            return None
    axisinfo = staticmethod(axisinfo)

    def convert(obj, unit):
        """
        convert obj using unit.  If obj is a sequence, return the
        converted sequence
        """
        if units.ConversionInterface.is_numlike(obj):
            return obj

        if iterable(obj):
            return [o.value(unit) for o in obj]
        else:
            return obj.value(unit)
    convert = staticmethod(convert)

    def default_units(x):
        'return the default unit for x or None'
        if iterable(x):
            for thisx in x:
                return thisx.unit
        else:
            return x.unit
    default_units = staticmethod(default_units)

units.registry[Foo] = FooConverter()

# create some Foos
```

```
x = []
for val in range( 0, 50, 2 ):
    x.append( Foo( val, 1.0 ) )

# and some arbitrary y data
y = [i for i in range( len(x) ) ]


# plot specifying units
fig = figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
ax.plot( x, y, 'o', xunits=2.0 )
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')


# plot without specifying units; will use the None branch for axisinfo
fig2 = figure()
ax = fig2.add_subplot(111)
ax.plot( x, y ) # uses default units

fig3 = figure()
ax = fig3.add_subplot(111)
ax.plot( x, y, xunits=0.5)

show()
```

In the matplotlib examples directory, there is a mockup units package `basic_units`. While this is not designed for production use and is not meant to be expanded upon, it does illustrate how one can use unit enabled arrays with matplotlib with default axis labeling and ticking. Here is an example using the `basic_units` code for plotting with radians and degrees; not the axis labels are set "automagically"

Listing 8.2: Working with units and conversions Figure 8.2

```
from basic_units import radians, degrees, cos
from pylab import figure, show, nx

x = nx.arange(0, 15, 0.01) * radians

fig = figure()
fig.subplots_adjust(hspace=0.3)

ax = fig.add_subplot(211)
ax.plot(x, cos(x), xunits=radians)

ax = fig.add_subplot(212)
```

Unit information is stored by the XAxis and YAxis instance using the `units` property. This property can be set implicitly by passing the *xunits* or *yunits* keyword arguments to the plotting functions, eg

```
ax.plot(cms, cms, xunits=cm, yunits=inch)
```

or set explicitly using the axis `set_units` method, eg

```
ax.xaxis.set_units(cm)
```

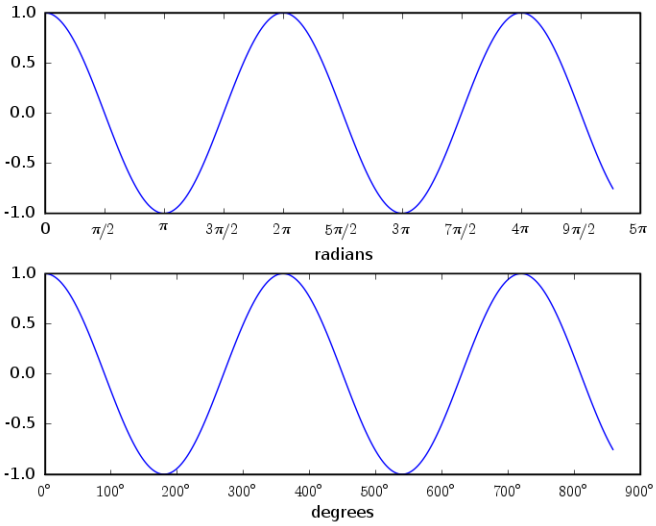

Figure 8.1: Basic units: radians and degrees; see Listing 8.2.

When units are reset, the matplotlib artists will attempt to convert themselves to the new unit, eg `cm` to `inch` conversions shouldbe fine, and `cm` to `Hz` conversions should raise an exception. Whether an exception is actually raise and what that exception is is the province of the underlying units package, since, as indicated above, matplotlib makes no assumption about the underlying units functionality other than it's interface as specified by `units.ConversionInterface`.

Chapter 9

Cookbook

9.1 Plot elements

9.1.1 Horizontal or vertical lines/spans

It is often useful to draw a line that stretches from the left to the right side of the axes at a given height, eg to represent a y-axis threshold. In this case, the left and right are plotted in axes coordinates (0 and 1 respectively), and the y coordinate is in data coordinates. Plotted this way, the horizontal extent of the line will not change if you interactively change the xlimits, eg by using the pan navigation tool. Although you can create these lines yourself using `matplotlib.lines.Line2D` instances and setting the appropriate transforms, several helper functions are provided to make this easier.

9.1.2 Fill the area between two curves

The fill command takes a list of vertices and draws a polygon. A filled area between two curves is simply a large polygon. All you need to do is get the vertices in the correct order, which basically means reversing the order of the x,y pairs in one of the lines, so that path across the vertices of the polygon is continuous. Here is a simple example

Listing 9.1: Fill the area between two curves; see Figure 9.1

```
from pylab import *
x1 = arange(0, 2, 0.01)
y1 = sin(2*pi*x1)
y2 = sin(4*pi*x1) + 2

# reverse x and y2 so the polygon fills in order
x = concatenate( (x1,x1[::-1]) )
y = concatenate( (y1,y2[::-1]) )

p = fill(x, y, facecolor='g')
```

9.1.3 Drawing true ellipses and circles

The `matplotlib.patches.Ellipse` class, and the special case derived class `matplotlib.patches.Circle` use 4 cubic splines to approximate a true ellipse, which gives a very close approximation to ellipses and is scale free. With a polygonal representation, you can see the discreteness of the approximation as you zoom into the vertices of the polygon.

Listing 9.2: The Ellipse patch approximates true ellipses using 4 cubic splines, which is much more accurate than a high resolution polygon representation. SeeFigure 9.2

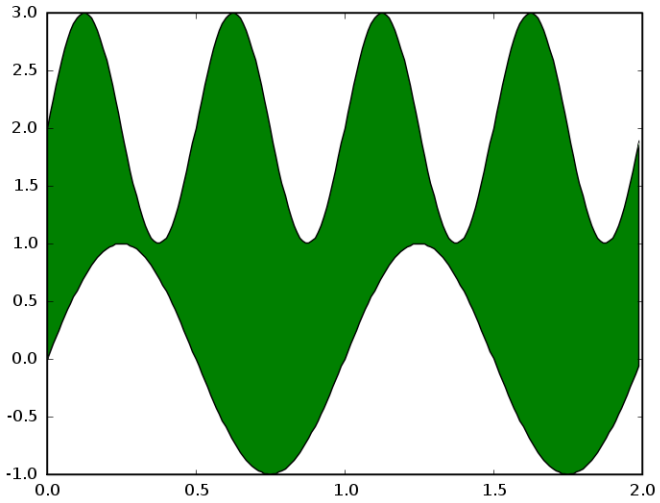

Figure 9.1: Fill the area between two curves; see Listing 9.1.

```
from pylab import figure, show, nx
from matplotlib.patches import Ellipse
rand = nx.mlab.rand

NUM = 250

ells = [Ellipse(xy=rand(2)*10, width=rand(), height=rand(), angle=rand()*360)
        for i in xrange(NUM)]

fig = figure()
ax = fig.add_subplot(111, aspect='equal')
for e in ells:
    ax.add_artist(e)
    e.set_clip_box(ax.bbox)
    e.set_alpha(rand())
    e.set_facecolor(rand(3))

ax.set_xlim(0, 10)
ax.set_ylim(0, 10)
```

9.2 Text

9.2.1 Adding a ylabel on the right of the axes

To make a ylabel on the right, use the `text` command. You need to set the transform to use axes coordinates (`ax.transAxes`), rotate the text vertically, make the horizontal alignment left, the vertical alignment centered. Note that x, y = 1, 0.5 is the right, middle of the axes in axes coordinates. You also need to turn off clipping, so the text can appear outside the axes w/o being clipped by the axes bounding box, which is the default behavior.

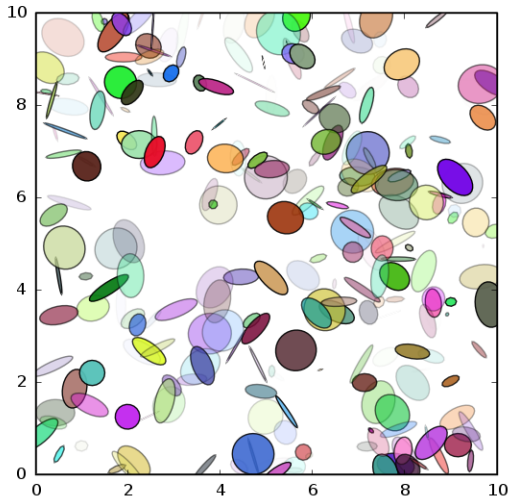

Figure 9.2: Drawing true ellipses using spline, rather than polygonal, representations; see Listing 9.2.

```
text(1.02, 0.5, 'volts',
     horizontalalignment='left',
     verticalalignment='center',
     rotation='vertical',
     transform=gca().transAxes,
     clip_on=False)
```

9.3 Data analysis

9.3.1 Linear regression

One of the most common tasks in analyzing data is a linear regression of one variable on another. matplotlib provides `polyfit` in the `matplotlib.mlab` module for general polynomial regression.

Listing 9.3: Best fit line; see Figure 9.3

```
from pylab import *

# Generate some test data; y is a linear function of x + nse
x = arange(0.0, 2.0, 0.05)
nse = 0.3*randn(len(x))
y = 2+ 3*x + nse

# the bestfit line from polyfit; you can do arbitrary order
# polynomials but here we take advantage of a line being a first order
# polynomial
m,b = polyfit(x,y,1)

# plot the data with blue circles and the best fit with a thick
# solid black line
```

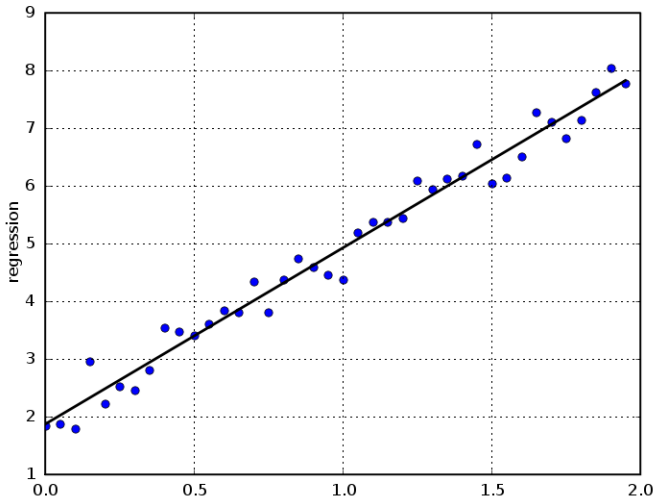

Figure 9.3: Estimating a best fit line for some random data; see Listing 9.3.

```
plot(x, y, 'bo', x, m*x+b, '-k', linewidth=2)
ylabel('regression')
grid(True)
```

9.3.2 Polynomial regression

polyfit can also be used for general polynomial fitting. The signature of polyfit is `coeffs = polyfit(x, y, N)` where N is the order of the polynomial. The best fit can be obtained from the coefficients and the x data using `best = polyval(x, coeefs)`. `coeffs` are the coefficients of the polynomial $coeffs = p_k, \ldots, p_1, p_0$.

The algorithm for polyfit is taken from Mathworld's *Least Squares Fitting Polynomial* and *Vandermonde Matrix* entries [Weisstein, 2002]. To do a best fit polynomial regression of order N of y onto x. We must solve an N-dimensional system of equations; eg, for $N = 2$

$$\begin{aligned} p_2 * x_0^2 + p_1 * x_0 + p_0 &= y_1 \\ p_2 * x_1^2 + p_1 * x_1 + p_0 &= y_1 \\ p_2 * x_2^2 + p_1 * x_2 + p_0 &= y_2 \\ &\ldots \\ p_2 * x_k^2 + p_1 * x_k + p_0 &= y_k \end{aligned}$$

If X is a the Vandermonde Matrix computed from x, then the polynomial least squares solution is given by the p in $X * p = y$ where X is a x by $N+1$ matrix, p is a $N+1$ length vector, and y is a len(x) by 1 vector. This equation can be solved as

$$p = (\bar{X}X)^{-1} * \bar{X} * y \tag{9.1}$$

where $\bar{X}$ is the transpose of X and the -1 superscript denotes the inverse. For more info, see Mathworld[1], but note that the k's and n's in the superscripts and subscripts on that page are problematic. The linear algebra is correct, however.

[1] http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html

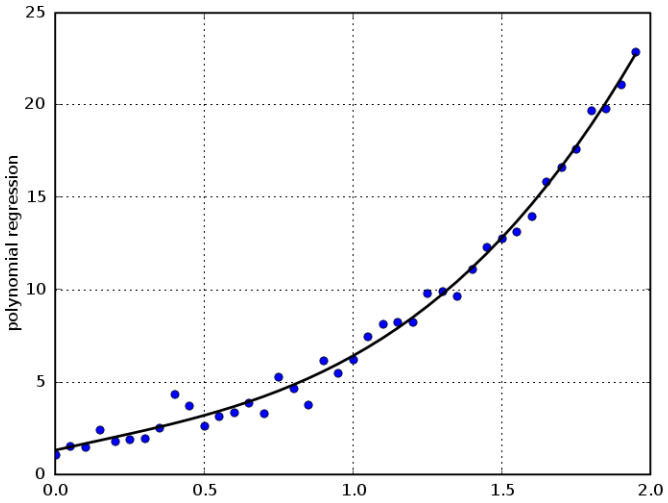

Figure 9.4: Estimating a best fit cubic for some random data; see Listing 9.4

Listing 9.4: est fit polynomial; see Figure 9.4

```
from pylab import *

# Generate some test data; y is a poly function of x + nse
x = arange(0.0, 2.0, 0.05)
nse = 0.6*randn(len(x))
y = 1.1 + 3.2*x + 0.1*x**2 + 2*x**3 + nse

# the bestfit line from polyfit
coeffs = polyfit(x,y,3)

# plot the data with blue circles and the best fit with a thick
# solid black line
besty = polyval(coeffs, x)
plot(x, y, 'bo', x, besty, '-k', linewidth=2)
ylabel('polynomial regression')
grid(True)
```

9.4 Working with images

9.4.1 Loading existing images into matplotlib

Currently matplotlib only supports plotting images from numerix arrays, either luminance, RGB or RGBA. If you have some existing data in an image file format such as PNG, JPEG or TIFF, you can load this into matplotlib by first loading the file into PIL - http://www.pythonware.com/products/pil and then converting this to a numerix array using fromstring / tostring methods

```
import Image
from pylab import *
```

```
im = Image.open('../data/leo_ratner.jpg')
s = im.tostring()     # convert PIL image -> string

# convert string -> numerix array of floats
rgb = fromstring(s, UInt8).astype(Float)/255.0

# resize to RGB array
rgb = resize(rgb, (im.size[1], im.size[0], 3))

imshow(rgb, interpolation='nearest')
axis('off')  # don't display the image axis
show()
```

9.4.2 Blending several axes images using alpha

You can compose several axes images on top of one another using alpha blending, as described in Section 3.7.1. If your `hold` state is *True*, multiple calls to `imshow` will cause the image arrays to be resampled to the axes bounding box and blended. Of course, the uppermost images must have alpha less than one, or else they will be fully opaque and hence the lower images will be invisible. Note that you can blend images from arrays of different shapes as well as blending images with different colormaps and interpolation schemes. The example below creates a black and white checkboard using a grayscale colormap and then blends a color image over it, as shown in Figure 9.5.

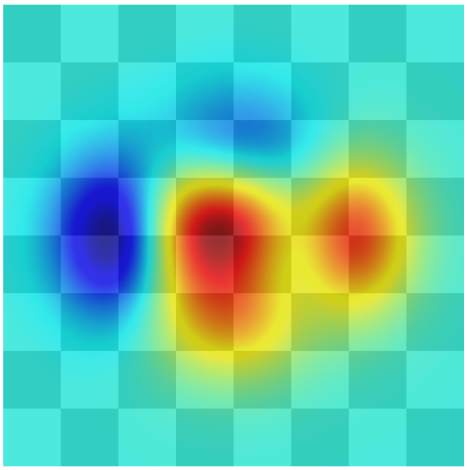

Figure 9.5: Layering axes images using alpha blending; see Listing 9.5.

Listing 9.5: Alpha-blending multiple images; see Figure 9.5

```
from pylab import *

def func3(x,y):
    return (1- x/2.0 + x**5 + y**3)*exp(-x**2-y**2)

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05
```

```
x = arange(-3.0, 3.0, dx)
y = arange(-3.0, 3.0, dy)
X,Y = meshgrid(x, y)
extent = min(x), max(x), min(y), max(y)


# make an 8 by 8 chessboard
Z1 = array(([0,1]*4 + [1,0]*4)*4); Z1.shape = 8,8
im1 = imshow(Z1, cmap=cm.gray,
             interpolation='nearest', extent=extent)

# prevents the axes from clearing on next command
hold(True)

Z2 = func3(X, Y)
im2 = imshow(Z2, cmap=cm.jet, alpha=.9,
             interpolation='bilinear',  extent=extent)
axis('off')
```

9.4.3 Creating a mosaic of images

You can compose several figure images into a mosaic using the `figimage` command, as discussed Section 3.7.2. If your `hold` state is *True*, multiple calls to `figimage(X, xo, yo)`. xo and yo are the pixel offsets from the origin (the origin can be either upper left or lower left, as discussed in Section 3.7.4). The code below using color mapping to place two images on the diagonal. Note that you can use different kinds of arrays (luminance, RGB, RGBA) and different colormaps when creating figure mosaics. See Figure 9.6.

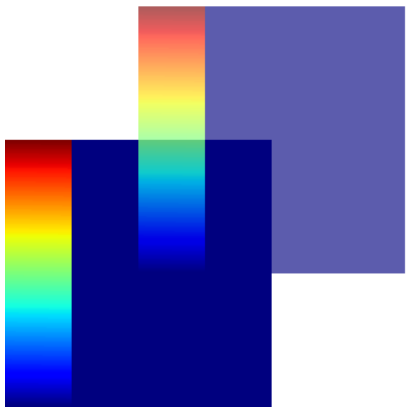

Figure 9.6: Creating a mosaic using `figimage`; see Listing 9.6.

Listing 9.6: Creating figure mosaics; see Figure 9.6

```
from pylab import *
rc('axes', hold=True)
```

```
rc('image', origin='upper')

Z = arange(40000.0); Z.shape = 200,200
Z[:,50:] = 1
im1 = figimage(Z, xo=0,  yo=0)
im2 = figimage(Z, xo=100, yo=100, alpha=.8)
```

9.4.4 Defining your own colormap

Perry Greenfield has provided a nice framework with `matplotlib.colors.LinearSegmentedColormap` to define new colormaps. You can create new colormaps fairly easy by following the example of jet in `matplotlib.cm`. Here are the steps

- define your rgb linear segments in `matplotlib.cm`, following the lead of the *_jet_data* dictionary in that module
- add an entry to the datad dictionary in that module which maps rc string names for your color map to the dictionary you just defined.
- instantiate a single instance of your colormap in cm, following the example

  ```
  jet = colors.LinearSegmentedColormap('jet', _jet_data, LUTSIZE)
  ```

- add a pylab function which has the same name as your colormap, following the example of `pylab.jet`.

Now anyone can use the colormap interactively from the shell, by setting it as the default image.cmap in rc, etc. Please submit your changes to the matplotlib-devel mailing list.

9.5 Output

9.5.1 Printing to standard output

In some instances, it is nice to be able to print to a file object, eg `sys.stdot`, for example in a web application server where the creation of a temporary file storing the images is a wasted step. The antigrain backend accepts a file object to the `savefig` command and will print a PNG to it. Thus to print to standard output, you could do

```
import sys
import matplotlib
# this is not supported across all backends, as of matplotlib-0.63
matplotlib.use('Agg')
from pylab import *
plot([1,2,3])
savefig(sys.stdout)
```

Chapter 10

Matplotlib API

The pylab interface does a lot of work for you under the hood, creating and managing multiple figure windows, directing your plotting commands to the current axes and figure, managing the interactive state, and so on. But that is all it does: all of the plotting is handled by a set of classes that the user can instantiate directly. If you are developing a GUI application, or simply don't want any hidden magic in your plots, you can create any plots using pure OO code that you could create with the pylab interface.

From a developer standpoint, the pylab interface has been a blessing in disguise. Because the interface was fixed by the Mathworks before the start of matplotlib provided considerable freedom to redesign the guts of the plotting library – the object model can be totally revamped and the user interface remains fixed.

The matplotlib code is divided conceptually into 3 parts: the MATLAB^{TM} interface, the matplotlib Artists, and the backend renderers. The *pylab interface* was covered in Chapter 3. This module `pylab` is comprised mainly of code to manage the multiple figure windows across backends, and provide a thin, procedural wrapper around the object oriented plotting code. The *matplotlib Artists* are a series of classes that derive from `matplotlib.artist.Artist`, so named because these are the objects that actually draw into the figure; see Figure 10.3. The *backend renderers* each implement a common drawing interface that actually puts the ink on the paper, eg creating a postscript document, filling an antigrain pixel buffer, or calling the gtk drawing code.

This chapter delves into matplotlib internals to give a clearer picture of how things work and how they are organized. Some of this material may be of interest only to developers, but most of it should shed light for anyone who wants to be able to exploit the full capabilities of matplotlib. The normal path of figure creation in matplotlib is pylab interface $\rightarrow$ creates artists $\rightarrow$ calls to the backend renderer. This section will invert that process, starting with the backend, which is where the drawing actually takes place. This is the natural order of presentation because the backend knows nothing about Artists, which in turn known nothing about the pylab interface. After the overview of the backend API, there is a discussion of the matplotlib artists; this is the section that is most useful to users, particularly those who want to embed matplotlib in an application. The final section shows how the pylab interface controls the backends and artists – this section is probably of interest to developers and the terminally curious.

10.1 The matplotlib backends

The backend consists of a number of related base classes that together define a drawing API. The original backend was GTK, and the drawing API is heavily based on the GTK drawing model, which is very simple. There are three essential classes defined in matplotlib.backend_bases: `RendererBase`, `GraphicsContextBase` and `FigureCanvasBase`. In addition, there are some classes for use with the GUI backends to define the interface to the toolbars and event handling. The `RendererBase`, aka *renderer* handles all the drawing primitives in display coordinates, typical renderer methods are `draw_text` and `draw_lines`. The `GraphicsContextBase`, aka textitgraphics context stores information about the graphics properties, such as linewidth, cap or join style, color, alpha translucency. The `FigureCanvasBase`, aka *figure canvas*, is primarily a container class to hold the `Figure` instance; this facilitates separation of the Figure from the backend dependent code. For GUI backends, the figure canvas should be a GUI widget embeddable in a GUI

window.

10.1.1 The renderer and graphics context

The renderer defines the low-level matplotlib drawing API; all of the drawing commands are done in display coordinates. The matplotlib Artist classes handle all of the layout and transformation issue, and pass the primitive drawing commands on to the renderer. The renderers know nothing about the matplotlib Artists, and nothing about the pylab interface. Their one and only job is to get ink onto the canvas. The graphics context stores information about the objects to be drawn, their color, linewidths, cap and join styles, alpha transparency, etc. Taken together, you can use the backend renderer and graphics context directly to make drawings. This may not be advisable, since the whole purpose of the matplotlib Artists and pylab interface is to simplify the process of getting ink onto the canvas, but it is possible. However, it is potentially useful to developers who may want to extend the capabilities of matplotlib, eg, to implement block diagram drawing.

Every backend in `matplotlib/backends` defines a `Renderer` that inherits from `RendererBase`; some also define a backend dependent `GraphicsContext`, while other simply use the `GraphicsContextBase` for storing the information and do all the work of translating these values in the `R`enderer. This is the approach the Agg backend uses, shown in the listing below.

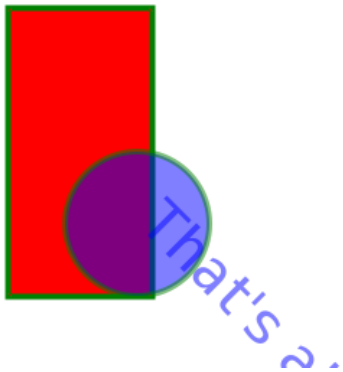

Figure 10.1: Drawing directly with the backend renderer and graphics context; see Listing 10.1

Listing 10.1: Drawing with the agg renderer; see Figure 10.1

```
# working directly with renderer and graphics contexts primitives
from matplotlib.font_manager import FontProperties
from matplotlib.backends.backend_agg import RendererAgg
from matplotlib.transforms import Value

# a 400x400 canvas at 72dpi canvas
dpi = Value(72.0)
o = RendererAgg(400,400, dpi)

# the graphics context
```

```
gc = o.new_gc()

# draw the background white
gc.set_foreground('w')
face = (1,1,1)  # white
o.draw_rectangle(gc, face, 0, 0, 400, 400)

# the gc's know about color strings, and can handle any matplotlib
# color arguments (hex strings, rgb, format strings, etc)
gc.set_foreground('g')
gc.set_linewidth(4)
face = (1,0,0)  # must be rgb
o.draw_rectangle(gc, face, 10, 50, 100, 200)

# draw a translucent ellipse
rgb = (0,0,1)
gc.set_alpha(0.5)
o.draw_arc(gc, rgb,  100, 100, 100, 100, 360, 360, 0)

# draw a dashed line
gc.set_dashes(0, [5, 10])
gc.set_joinstyle('miter')
gc.set_capstyle('butt')
gc.set_linewidth(3.0)
#broken with new API
#o.draw_lines( gc, (50, 100, 150, 200, 250), (400, 100, 300, 200, 250))

# draw some text using the matplotlib font manager
prop = FontProperties(size=40)
gc.set_foreground('b')
o.draw_text( gc, 100, 300, "That's all folks!", prop, -45, 0)

# there is no standard renderer interface to save the input to a file,
# as this is the job of the figure canvas.  Here I make the call that
# the figure canvas would make for the antigrain render
o._renderer.write_png('../figures/renderer_agg.png')
```

10.1.2 The figure canvases

10.2 The matplotlib Artists

10.3 pylab interface internals

Let's look at the simplest matplotlib script and walk through what happens under the hood. This section will be of interest mainly to developers or those curious about matplotlib internals - it can be safely skipped by others. We'll assume you are using one of the GUI backends, eg GTKAgg and have are running this as a script (`interactive : False`)

```
from pylab import *
plot([1,2,3])
show()
```

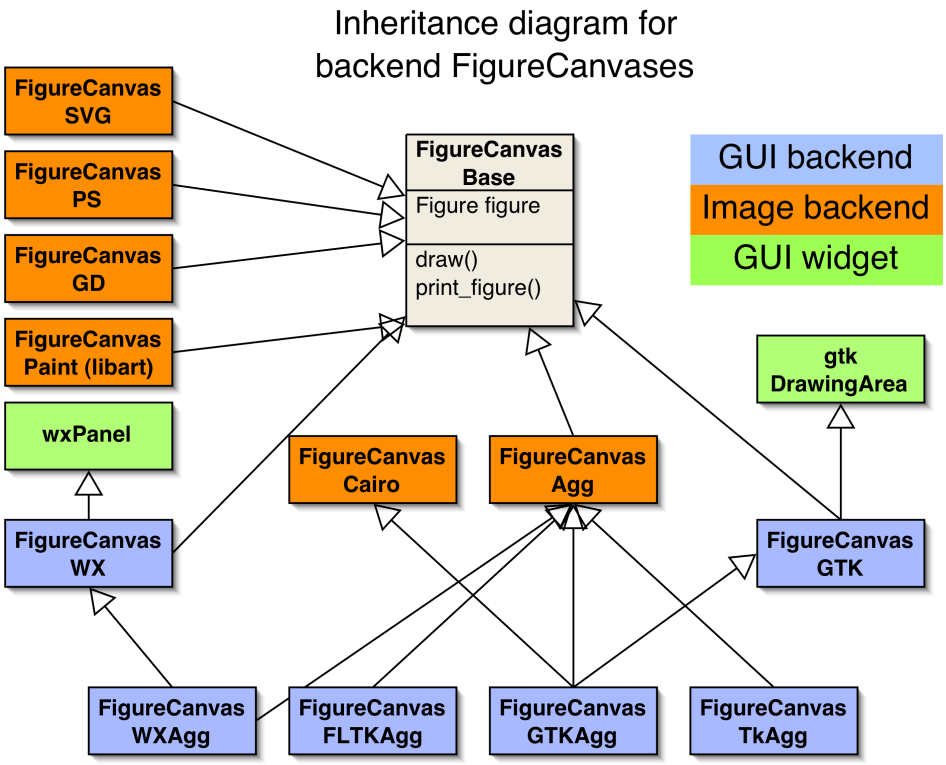

Figure 10.2: The inheritance diagram for The FigureCanvas hierarchy. The FigureCanvas is a backend dependent class which contains a figure instance. For GUI backends, the canvas should be a GUI widget embeddable in a GUI window. Some of the GUIs have backends with both native drawing and antigrain drawing (GTK/GTKAgg, WX/WXAgg), which is readily achieved with multiple inheritance.

Line 1: `from pylab import *`

When any matplotlib code is imported the first time, the `matplotlib.__init__.py` code is called. The primary thing the init code does is find and parse your rc file, or if it fails fall back on a set of default parameters. Once this is done, `pylab` proceeds to import all of the `matplotlib.numerix` and `matplotlib.mlab` symbols into the namespace, and loads the backend from `matplotlib.backends`, which use the rc information to load four functions from the backend module specified by the rc `backend` parameter. The pylab interface requires only four functions from the backend: `new_figure_manager`, `error_msg`, `draw_if_interactive`, and `show`

- `new_figure_manager` is responsible for creating a new instance from a backend dependent class derived from `matplotlib.backend_bases.FigureManager`; this class wraps GUI window creation and management.
- `error_msg` displays an error message; for image backends this message is printed to the file object determined by the rc parameter `verbose.erro` and for GUI backends it is typically displayed in a GUI dialog box.
- `draw_if_interactive` is called after every `pylab` drawing command (`plot`, `set`, `xlim`, ...) and updates the figure window with the new information only if `interactive` is *True*.
- `show` raises all the GUI figure windows and triggers a command to draw the figure.

The pylab interface also imports a `Gcf` instance from the `matplotlib._matlab_helpers` module, which manages the current figure and current axes. The pylab interface defines `gcf` and `gca` to get a reference to the current figure and

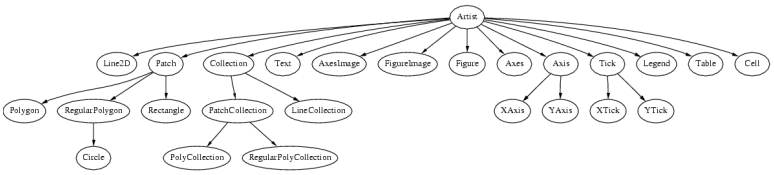

Figure 10.3: The matplotlib Artist hierarchy. The primitive Artists are the Patches, Line2D, Text, AxesImage, FigureImage and Collection classes. All other artists are composites of these. For example a Tick is comprised of a Line2D instance and a Text instance, which make up the tick line and tick label; the Axis is comprised of a list of Ticks and a Text axis label; see Figure 10.4.

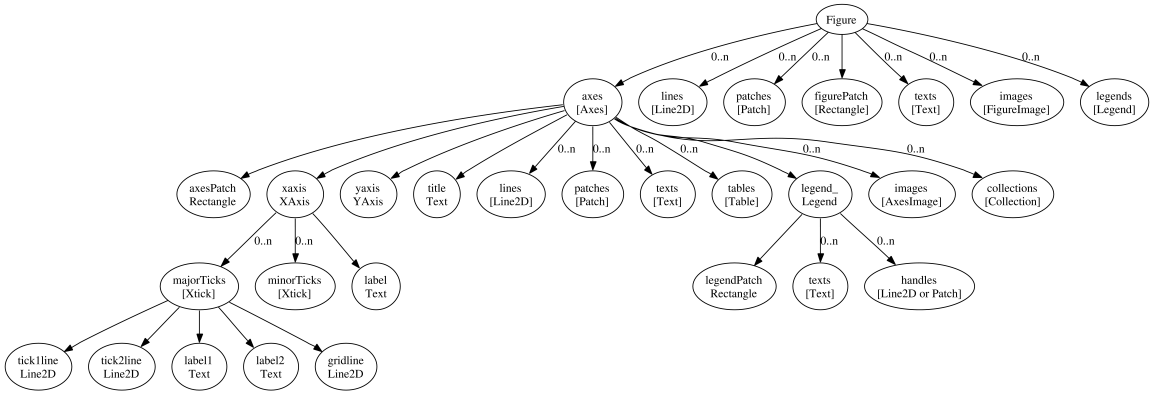

Figure 10.4: The Artist containment hierarchy. The top level Artist is the `matplotlib.figure.Figure`, which contains all of the other Artist instances. The attribute names are given in lower case, and the object type is listed below in upper case. If the attribute is a sequence, for example the figure contains a list of Axes, then the connecting edge is labeled $0 \ldots n$ and the object type is in square brackets to indicate a list, eg [Axes]. Some redundant information is omitted; for example, the yaxis contains the equivalent objects that the xaxis contains, the minorTicks have the same containment hierarchy as the majorTicks, and so on.

axes, which in turn are interfaces to the `Gcf` class that does the real lifting.

Line 2: `plot([1,2,3])`

All of the `pylab` functions are defined similarly: they get the current axes and forward the call on to the corresponding `matplotlib.axes.Axes` method, which does the real work. The `plot` command in the example below calls `gca` to get the current axes. If no figure or axes has been defined at the time of this call, the are created one on the fly using default parameters from the rc file; ultimately the `new_figure_manager` backend method is called to provide new figures when needed, and the default `subplot(111)` is added to the figure if no other axes has been defined.

The `new_figure_manager` method deserves a bit more attention, because this creates the central object that contains all the other objects relevant to the creation of a single figure. `matplotlib.backend_bases.FigureManagerBase` is a container class for the figure window (a GUI window) and figure canvas (a GUI widget which can be drawn upon). The figure canvas derives from `matplotlib.backend_bases.FigureCanvasBase`, and contains the `matplotlib.figure.Figure` instance.

Once the current axes is obtained by `gca`, `plot` forwards the call to `Axes.plot`. If an exception is raise, the backend

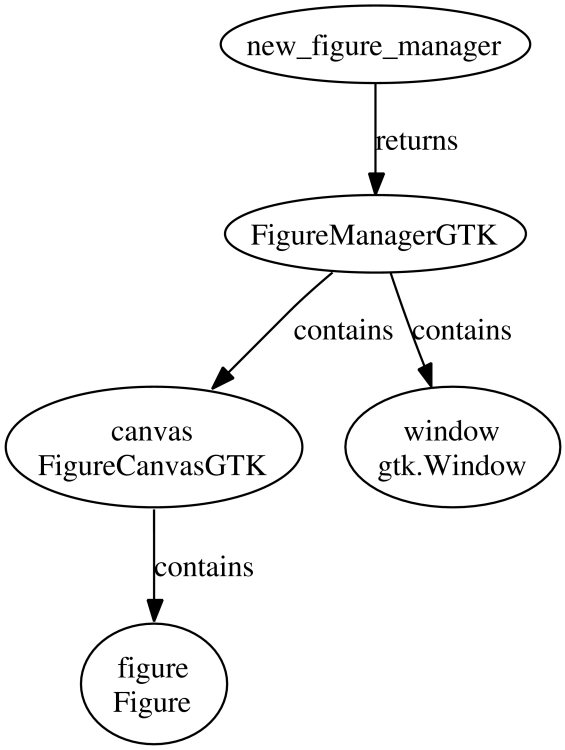

Figure 10.5: The pylab interface function `new_figure_manager` returns a backend dependent concrete implementation of `matplotlib.backend_bases.FigureManagerBase`, which contains the figure canvas and figure window. The attribute names are shown in lower case, and the backend dependent classes are shown in upper case. The standard attribute naming system allows the MATLAB^{TM} interface to make calls across backends to the figure canvas and figure.

`error_msg` method is called with the traceback to display it. If the code is successful, the backend `draw_if_interactive` method is called which will update the plot if the rc parameter `interactive` is *True*, and finally the return value is returned.

```
def plot(*args, **kwargs):
    try:
        lines =  gca().plot(*args, **kwargs)
    except ValueError, msg:
        msg = raise_msg_to_str(msg)
        error_msg(msg)
    else:
        draw_if_interactive()
        return lines
plot.__doc__ = Axes.plot.__doc__
```

The `matplotlib.axes.Axes.plot` method parses the `*args` and **kwargs, creates the requested line objects, and adds them to its list of `Line2D` instances. It will also extract the x and y data range and use these to update the data limits of the axes, which is turn will be used to autoscale the view limits. No drawing code is actually issued, but is deferred until later.

Line 3: `show()`

`show` is an interface to realize and show the GUI windows. For image backends, eg Agg, PS or SVG, it is superfluous. The image backends will draw the figure on a call to savefig, and ignore a `show` call. Each GUI backend defines `show` to realize all of the GUI windows, and start the GUI mainloop. For this reason, the call to `show` is blocking, and should be the last line of the script. Here is a representative show method from `matplotlib.backends.backend_gtk`

```
def show(mainloop=True):
    """
    Show all the figures and enter the gtk main loop

    This should be the last line of your script
    """

    for manager in Gcf.get_all_fig_managers():
        manager.window.show()

    if gtk.main_level() == 0 and mainloop:
        if gtk.pygtk_version >= (2,4,0):  gtk.main()
        else:                             gtk.mainloop()
```

Typically, the GUI backends binds the realize or expose event of the GUI window to ultimately trigger the `Figure.draw` method of the Figure instance contained by the `FigureCanvas`. In the show function above, `manager.window.show()` will trigger an expose event in pygtk. The gtk backend binds the expose event to the `FigureCanvasGTK.expose_event` method. If the canvas has not yet been drawn, the `expose_event` method will create a RendererGTK instance (which derives from the common drawing API in `matplotlib.backends.RendererBase`) and then call `Figure.draw(renderer)`, which in turn passes the draw command on to each `Artist` instance it contains; see Figure 10.4 for the `Artist` containment hierarchy. Each `Artist` instance defines the draw method, and contains a transform to transform itself to display coordinates. For example, the Line2D instance will transform its x and y data to display coordinates, and then call the appropriate renderer method, eg `RendererGTK.draw_lines`, which expects x and y data in display coordinates. In this case, the GTK renderer `draw_lines` method makes the appropriate calls to the GTK drawing API, and the screen is updated; see Figure 10.6.

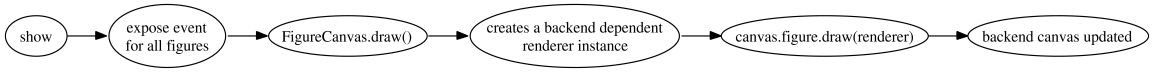

Figure 10.6: The typical sequence of steps triggered in the backend code by the call to `show` that ultimately gets the ink on the canvas.

Appendix A

A sample matplotlibrc

```
### MATPLOTLIBRC FORMAT

# This is a sample matplotlib configuration file.  It should be placed
# in HOME/.matplotlib/matplotlibrc (unix/linux like systems) and
# C:\Documents and Settings\yourname\.matplotlib (win32 systems)
#
# By default, the installer will overwrite the existing file in the
# install path, so if you want to preserve your's, please move it to
# your HOME dir and set the environment variable if necessary.
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting
#
# Blank lines, or lines starting with a comment symbol, are ignored,
# as are trailing comments.  Other lines must have the format
#
#   key : val   # optional comment
#
# Colors: for the color values below, you can either use
#  - a matplotlib color string, such as r, k, or b
#  - an rgb tuple, such as (1.0, 0.5, 0.0)
#  - a hex string, such as ff00ff  (no '#' symbol)
#  - a scalar grayscale intensity such as 0.75
#  - a legal html color name, eg red, blue, darkslategray

#### CONFIGURATION BEGINS HERE
# the default backend; one of GTK GTKAgg GTKCairo FltkAgg QtAgg TkAgg
#     Agg Cairo GD GDK Paint PS PDF SVG Template
backend      : Agg
numerix      : numpy  # numpy, Numeric or numarray
interactive  : False      # see http://matplotlib.sourceforge.net/interactive.html
#toolbar      : toolbar2   # None | classic | toolbar2
#timezone     : UTC        # a pytz timezone string, eg US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location.  This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata


### LINES
```

```
# See http://matplotlib.sourceforge.net/matplotlib.lines.html for more
# information on line properties.
#lines.linewidth   : 1.0     # line width in points
#lines.linestyle   : -       # solid line
#lines.color       : blue
#lines.marker      : None    # the default marker
#lines.markeredgewidth  : 0.5     # the line width around the marker symbol
#lines.markersize  : 6           # markersize, in points
#lines.dash_joinstyle : miter        # miter|round|bevel
#lines.dash_capstyle : butt          # butt|round|projecting
#lines.solid_joinstyle : miter       # miter|round|bevel
#lines.solid_capstyle : projecting   # butt|round|projecting
#lines.antialiased : True         # render lines in antialised (no jaggies)

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles.  See
# http://matplotlib.sourceforge.net/matplotlib.patches.html for more
# information on patch properties
#patch.linewidth        : 1.0     # edge width in points
#patch.facecolor        : blue
#patch.edgecolor        : black
#patch.antialiased      : True    # render patches in antialised (no jaggies)

### FONT
#
# font properties used by text.Text.  See
# http://matplotlib.sourceforge.net/matplotlib.font_manager.html for more
# information on font properties.  The 6 font properties used for font
# matching are given below with their default values.
#
# The font.family property has five values: 'serif' (e.g. Times),
# 'sans-serif' (e.g. Helvetica), 'cursive' (e.g. Zapf-Chancery),
# 'fantasy' (e.g. Western), and 'monospace' (e.g. Courier).  Each of
# these font families has a default list of font names in decreasing
# order of priority associated with them.
#
# The font.style property has three values: normal (or roman), italic
# or oblique.  The oblique style will be used for italic, if it is not
# present.
#
# The font.variant property has two values: normal or small-caps.  For
# TrueType fonts, which are scalable fonts, small-caps is equivalent
# to using a font size of 'smaller', or about 83% of the current font
# size.
#
# The font.weight property has effectively 13 values: normal, bold,
# bolder, lighter, 100, 200, 300, ..., 900.  Normal is the same as
# 400, and bold is 700.  bolder and lighter are relative values with
# respect to the current weight.
#
# The font.stretch property has 11 values: ultra-condensed,
# extra-condensed, condensed, semi-condensed, normal, semi-expanded,
# expanded, extra-expanded, ultra-expanded, wider, and narrower.  This
# property is not currently implemented.
#
```

```
# The font.size property is the default font size for text, given in pts.
# 12pt is the standard value.
#
#font.family         : sans-serif
#font.style          : normal
#font.variant        : normal
#font.weight         : medium
#font.stretch        : normal
# note that font.size controls default text sizes.  To configure
# special text sizes tick labels, axes, labels, title, etc, see the rc
# settings for axes and ticks. Special text sizes can be defined
# relative to font.size, using the following values: xx-small, x-small,
# small, medium, large, x-large, xx-large, larger, or smaller
#font.size           : 12.0
#font.serif          : Bitstream Vera Serif, New Century Schoolbook, Century
    ...Schoolbook L, Utopia, ITC Bookman, Bookman, Nimbus Roman No9 L, Times New Roman,
    ... Times, Palatino, Charter, serif
#font.sans-serif     : Bitstream Vera Sans, Lucida Grande, Verdana, Geneva, Lucid,
    ...Arial, Helvetica, Avant Garde, sans-serif
#font.cursive        : Apple Chancery, Textile, Zapf Chancery, Sand, cursive
#font.fantasy        : Comic Sans MS, Chicago, Charcoal, Impact, Western, fantasy
#font.monospace      : Bitstream Vera Sans Mono, Andale Mono, Nimbus Mono L, Courier
    ...New, Courier, Fixed, Terminal, monospace

### TEXT
# text properties used by text.Text.  See
# http://matplotlib.sourceforge.net/matplotlib.text.html for more
# information on text properties

#text.color          : black
#text.usetex         : False  # use latex for all text handling.  For more
    ...information, see
                           # http://www.scipy.org/Wiki/Cookbook/Matplotlib/UsingTex
#text.dvipnghack     : False  # some versions of dvipng don't handle
                           # alpha channel properly.  Use True to correct and flush
                           # ~/.matplotlib/tex.cache before testing

### AXES
# default face and edge color, default tick sizes,
# default fontsizes for ticklabels, and so on.  See
# http://matplotlib.sourceforge.net/matplotlib.axes.html#Axes
#axes.hold           : True    # whether to clear the axes by default on
#axes.facecolor      : white   # axes background color
#axes.edgecolor      : black   # axes edge color
#axes.linewidth      : 1.0     # edge linewidth
#axes.grid           : False   # display grid or not
#axes.titlesize      : 14      # fontsize of the axes title
#axes.labelsize      : 12      # fontsize of the x any y labels
#axes.labelcolor     : black
#axes.axisbelow      : False   # whether axis gridlines and ticks are below
                             # the axes elements (lines, text, etc)
#axes.formatter.limits : -7, 7 # use scientific notation if log10
                             # of the axis range is smaller than the
                             # first or larger than the second

#polaraxes.grid      : True    # display grid on polar axes
```

```
### TICKS
# see http://matplotlib.sourceforge.net/matplotlib.axis.html#Ticks
#xtick.major.size     : 4      # major tick size in points
#xtick.minor.size     : 2      # minor tick size in points
xtick.major.pad       : 8      # distance to major tick label in points
xtick.minor.pad       : 8      # distance to the minor tick label in points
#xtick.color          : k      # color of the tick labels
#xtick.labelsize      : 12     # fontsize of the tick labels
#xtick.direction      : in     # direction: in or out

#ytick.major.size     : 4      # major tick size in points
#ytick.minor.size     : 2      # minor tick size in points
ytick.major.pad       : 8      # distance to major tick label in points
ytick.minor.pad       : 8      # distance to the minor tick label in points
#ytick.color          : k      # color of the tick labels
#ytick.labelsize      : 12     # fontsize of the tick labels
#ytick.direction      : in     # direction: in or out


### GRIDS
#grid.color       :   black   # grid color
#grid.linestyle   :   :       # dotted
#grid.linewidth   :   0.5     # in points

### Legend
#legend.isaxes        : True
#legend.numpoints     : 2      # the number of points in the legend line
#legend.fontsize      : 14
#legend.pad           : 0.2    # the fractional whitespace inside the legend border
#legend.markerscale   : 1.0    # the relative size of legend markers vs. original
# the following dimensions are in axes coords
#legend.labelsep      : 0.010  # the vertical space between the legend entries
#legend.handlelen     : 0.05   # the length of the legend lines
#legend.handletextsep : 0.02   # the space between the legend line and legend text
#legend.axespad       : 0.02   # the border between the axes and legend edge
#legend.shadow        : False

### FIGURE
# See http://matplotlib.sourceforge.net/matplotlib.figure.html#Figure
#figure.figsize   : 8, 6    # figure size in inches
#figure.dpi       : 80      # figure dots per inch
#figure.facecolor : 0.75    # figure facecolor; 0.75 is scalar gray
#figure.edgecolor : white   # figure edgecolor

# The figure subplot parameters.  All dimensions are fraction of the
# figure width or height
#figure.subplot.left   : 0.125  # the left side of the subplots of the figure
#figure.subplot.right  : 0.9    # the right side of the subplots of the figure
#figure.subplot.bottom : 0.1    # the bottom of the subplots of the figure
#figure.subplot.top    : 0.9    # the top of the subplots of the figure
#figure.subplot.wspace : 0.2    # the amount of width reserved for blank space between
    ... subplots
#figure.subplot.hspace : 0.2    # the amount of height reserved for white space between
    ... subplots
```

```
### IMAGES
#image.aspect : equal             # equal | auto | a number
#image.interpolation  : bilinear  # see help(imshow) for options
#image.cmap   : jet               # gray | jet etc...
#image.lut    : 256               # the size of the colormap lookup table
#image.origin : upper             # lower | upper


### CONTOUR PLOTS
#contour.negative_linestyle :  6.0, 6.0 # negative contour dashstyle (size in points)

### SAVING FIGURES
# the default savefig params can be different for the GUI backends.
# Eg, you may want a higher resolution, or to make the figure
# background white
#savefig.dpi       : 100      # figure dots per inch
#savefig.facecolor : white    # figure facecolor when saving
#savefig.edgecolor : white    # figure edgecolor when saving

# tk backend params
#tk.window_focus   : False    # Maintain shell focus for TkAgg
#tk.pythoninspect  : False    # tk sets PYTHONINSEPCT

# ps backend params
#ps.papersize      : letter   # auto, letter, legal, ledger, A0-A10, B0-B10
#ps.useafm         : False    # use of afm fonts, results in small files
#ps.usedistiller   : False    # can be: None, ghostscript or xpdf
                                # Experimental: may produce smaller files.
                                # xpdf intended for production of
                                    ...publication quality files,
                                # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res  : 6000      # dpi

# pdf backend params
#pdf.compression   : 6 # integer from 0 to 9
                     # 0 disables compression (good for debugging)

# svg backend params
#svg.image_inline : True       # write raster image data directly into the svg file
#svg.image_noscale : False     # suppress scaling of raster data embedded in SVG

# Set the verbose flags.  This controls how much information
# matplotlib gives you at runtime and where it goes.  Ther verbosity
# levels are: silent, helpful, debug, debug-annoying.  Any level is
# inclusive of all the levels below it.  If you setting is debug,
# you'll get all the debug and helpful messages.  When submitting
# problems to the mailing-list, please set verbose to helpful or debug
# and paste the output into your report.
#
# The fileo gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, eg --verbose-helpful.
```

```
#
# You can access the verbose instance in your code
#   from matplotlib import verbose.
#verbose.level   : silent      # one of silent, helpful, debug, debug-annoying
#verbose.fileo   : sys.stdout  # a log filename, sys.stdout or sys.stderr
```

Appendix B

mathtext symbols

Appendix C

matplotlib source code license

matplotlib is distributed under the Python Software Foundation (PSF) license, which permits commercial and noncommercial free use and redistribution as long as the conditions below are met. The VERSION string below is replaced by the current matplotlib version number with each release.

```
LICENSE AGREEMENT FOR MATPLOTLIB VERSION
--------------------------------------

1. This LICENSE AGREEMENT is between the John D. Hunter ("JDH"), and the
Individual or Organization ("Licensee") accessing and otherwise using
matplotlib software in source or binary form and its associated
documentation.

2. Subject to the terms and conditions of this License Agreement, JDH
hereby grants Licensee a nonexclusive, royalty-free, world-wide license
to reproduce, analyze, test, perform and/or display publicly, prepare
derivative works, distribute, and otherwise use matplotlib VERSION
alone or in any derivative version, provided, however, that JDH's
License Agreement and JDH's notice of copyright, i.e., "Copyright (c)
2002-2007 John D. Hunter; All Rights Reserved" are retained in
matplotlib VERSION alone or in any derivative version prepared by
Licensee.

3. In the event Licensee prepares a derivative work that is based on or
incorporates matplotlib VERSION or any part thereof, and wants to
make the derivative work available to others as provided herein, then
Licensee hereby agrees to include in any such work a brief summary of
the changes made to matplotlib VERSION.

4. JDH is making matplotlib VERSION available to Licensee on an "AS
IS" basis.  JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR
IMPLIED.  BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND
DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS
FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB VERSION
WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB
VERSION FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR
LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING
```

MATPLOTLIB VERSION, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF
THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material
breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any
relationship of agency, partnership, or joint venture between JDH and
Licensee. This License Agreement does not grant permission to use JDH
trademarks or trade name in a trademark sense to endorse or promote
products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib VERSION,
Licensee agrees to be bound by the terms and conditions of this License
Agreement.

Bibliography

Julius S. Bendat and Allan G. Piersol. *Random Data: Analysis and Measurement Procedures*. John Wiley & Sons, New York, 1986.

Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. Chapman & Hall/CRC, second edition edition, 2002.