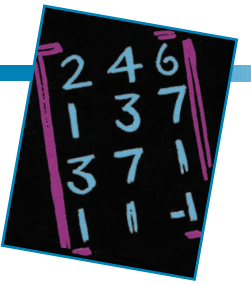
Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu



MATPLOTLIB: A 2D GRAPHICS ENVIRONMENT

By John D. Hunter

Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems.

I began learning Python in 2001, mainly as a way to procrastinate during the final stages of preparing my dissertation. I was hooked in short order. My numerical and statistical workflow at the time was a mix of Fortran, C++, and Matlab; I used the file system to communicate between the three. Python's ability to integrate seamlessly and transparently with high-performance compiled code dramatically simplified this workflow, but I couldn't quite wean myself from the breadth, quality, and ease of use that the Matlab environment offered for graphics.

Around that time, I began a fairly substantial project that involved writing an application to analyze human electrocorticography (ECoG) signals registered with 3D medical image data, and I debated long and hard about using Python or Matlab. On balance—and after much hand wringing—Matlab won, primarily because of its excellent graphics and secondarily because of its widespread use in the ECoG community. The application quickly grew in complexity, ultimately incorporating 3D medical image visualizations, 2D ECoG displays, spectral and time-series analyses, and the data structures required to represent human subject data. The networking support included data files served up over HTTP, metadata served up over MySQL, and some Web Common Gateway Interface (CGI) forms thrown into the mix for good measure. Not everyone knows that Matlab embeds its own Java Virtual Machine (JVM), which makes it possible to handle all of these things, but making them work together became increasingly painful, and I eventually hit the wall and decided to start all over again in Python.

The first step was to find a suitable replacement for the Matlab 2D graphics engine (the Visualization Toolkit [VTK] in Python provided 3D-visualization support that was more than adequate for my purposes). Although a score of graphics packages were and are readily available for Python, none met all my needs: they had to be embeddable in a GUI for application development, support different platforms, offer extremely high-quality raster and vector (primarily PostScript) hardcopy output for publication, provide support for mathematical expressions, and work interactively from the shell.

I wrote matplotlib to satisfy these needs, concentrating initially on the first requirement so I could get up and running with my ECoG application (the `pbrain` component of the "neuroimaging in Python project" at http://nipy.scipy.org; also, see p. 52 in this issue) and then gradually adding support for the others, with generous contributions from the matplotlib community. Because I was intimately familiar with Matlab and happy with its graphics environment, I followed the advice of Edward Tufte ("copy the great architectures") and T.S. Elliot ("talent imitates, but genius steals") and reverse-engineered the basic Matlab interface. Figure 1 shows a screenshot of the `pbrain` ECoG viewer I wrote in matplotlib.

The latest release of matplotlib runs on all major operating systems, with binaries for Macintosh's OS X, Microsoft Windows, and the major Linux distributions; it can be embedded in GUIs written in GTK, WX, Tk, Qt, and FLTK; has vector output in PostScript, Scalable Vector Graphics (SVG), and PDF; supports TeX and LaTeX for text and mathematical expressions; supports major 2D plot types and interactive graphics, including *xy* plots, bar charts, pie charts, scatter plots, images, contouring, animation, picking, event handling, and annotations; and is distributed under a permissive license based on the one from the Python Software Foundation. Along with a large community of users and developers, several institutions also use and support matplotlib development, including the Space Telescope Science Institute and the Jet Propulsion Laboratory.

Getting Started: A Simple Example

Matplotlib has a Matlab emulation environment called

PyLab, which is a simple wrapper of the matplotlib API. Although many die-hard Pythonistas bristle at PyLab's Matlab-like syntax and its `from pylab import *` examples, which dump the PyLab and NumPy functionality into a single namespace for ease of use, this feature is an essential selling point for many teachers whose students aren't programmers and don't want to be: they just want to get up and running. For many of these students, Matlab is the only exposure to programming they've ever had, and the ability to leverage that knowledge is often a critical point for teachers trying to bring Python into the science classroom. In Figure 2 , I've enabled the `usetex` parameter in the matplotlib configuration file so LaTeX can generate both the text and the equations.

Let's look at a sample session from IPython, the interactive Python shell that is matplotlib-aware in PyLab mode (also see p. 21 in this issue):

```
> ipython -pylab
IPython 0.7.3 — An enhanced Interactive Python.
   Welcome to pylab, a matplotlib-based
   Python environment. For more information,
   type 'help(pylab)'.

In [1]: subplot(111)
In [2]: t = arange(0.0,3.01,0.01)
In [3]: s = sin(2*pi*t)
In [4]: c = sin(4*pi*t)
In [5]: fill(t, s, 'blue', t, c, 'green',
   alpha=0.3);
In [6]: title(r'\TeX\ is
No.$\displaystyle\sum_{n=1}^\infty
   \frac{-e^{i\pi}}{2^n}$!')
```

IPython detects which GUI windowing system you want to use by inspecting your matplotlib configuration, imports the PyLab namespace, and then makes the necessary threading calls so you can work interactively with a GUI mainloop such as GTK's.

Images, Color Mapping, Contouring, and Color Bars

In addition to simple line plots, you can fairly easily create more sophisticated graphs, including color-mapped images with contouring and labeling, in just a few lines of Python. For pseudocolor images, matplotlib supports various image-interpolation and color-mapping schemes. For interpolation, you can choose "nearest" (which does a nearest neighbor interpolation for those who just want to see their raw data), "bilinear," "bicubic," and 14 other interpolation methods for smoothing data. For color mapping, all the classic color maps from Matlab are available (`gray`, `jet`, `hot`, `copper`, `bone`, and so on) as well as scores more. You can also define custom color maps.

Let's look at a Python script that computes a bivariate Gaussian distribution plotted as a grayscale image and then overlays contour lines using the heated object scale `hot` color map:

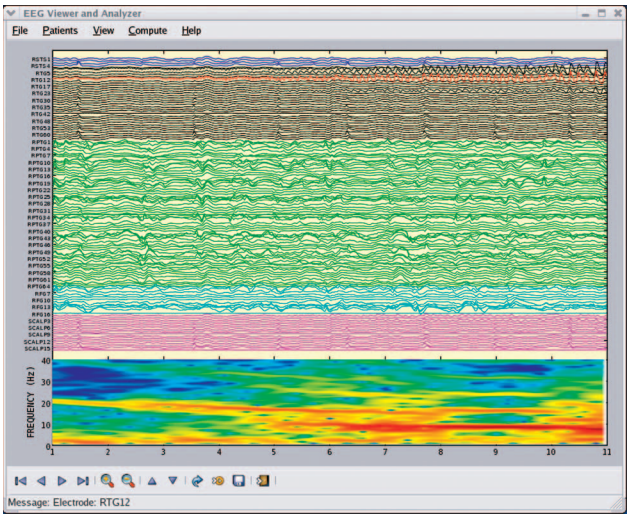


Figure 1. The PBrain project. An electrocorticography (ECoG) viewer, written in matplotlib and embedded in a `pygtk` application.

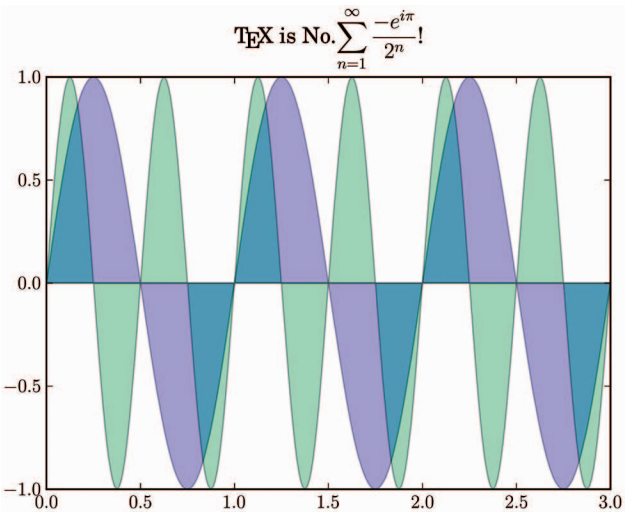


Figure 2. LaTeX support. Setting the `usetex` option in the matplotlib configuration file enables LaTeX generation of the text and equations in a matplotlib figure, as this screenshot shows.

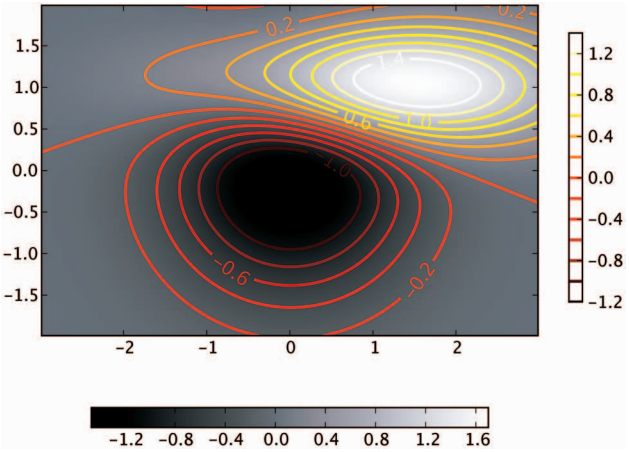


Figure 3. Images, contours, and color mapping. This screenshot from matplotlib illustrates how to add contour lines to luminosity images; note that the use of multiple color maps (`gray` and `hot`) and color bars (`continuous` and `discrete`) are supported.

```
from pylab import figure, cm, nx, show
from matplotlib.mlab import meshgrid, \
   bivariate_normal

delta = 0.025
x = nx.arange(-3.0, 3.0, delta)
y = nx.arange(-2.0, 2.0, delta)
X, Y = meshgrid(x, y)
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10.0 * (Z2 - Z1) # difference of Gaussians

fig = figure()
ax = fig.add_subplot(111)

# make a grayscale image
im = ax.imshow(Z, interpolation='bilinear',
   cmap=cm.gray, extent=(-3,3,-2,2),
   origin='lower')
levels = nx.arange(-1.2, 1.6, 0.2)

# do a contour using a "hot" colormap for
# the lines
cs = ax.contour(Z, levels, linewidths=2,
   cmap=cm.hot, extent=(-3,3,-2,2),
   origin='lower')

# label every 2nd contour inline
ax.clabel(cs, levels[1::2], inline=1,
   fmt='%1.1f')

# make a colorbar for the contour lines
cbar = fig.colorbar(cs, shrink=0.8,
   extend='both')

# we can still add a colorbar for the image, too.
cbarim = fig.colorbar(im, orientation=
   'horizontal', shrink=0.8)

# This makes the original colorbar look a bit
# out of place, so let's improve its position.
l,b,w,h = ax.get_position()
ll,bb,ww,hh = cbar.ax.get_position()
cbar.ax.set_position([ll, b+0.1*h, ww, h*0.8])

show()
```

Figure 3 shows the output of this Python script, with multiple color maps and color bars supported in a single axes, as well as continuous (the horizontal `gray` bar) and discrete color bars (the vertical `hot` bar).

Interactive Plotting

To facilitate interactive work, matplotlib provides access to basic GUI events, such as `button_press_event`, `mouse_motion_event`, `key_press_event`, `draw_event`, and so on; you can also register with these events to receive callbacks. In addition to the GUI-provided information, we attach matplotlib-specific data—if you connect to the `button_press_event`, for example, you can get the button press's *x* and *y* location in the display space, the `xdata` and `ydata` coordinates in the user space, which axes the click occurred in, and the underlying GUI event that generated the callback.

Event Handling

Matplotlib abstracts GUI event handling across the five major GUIs it supports, so event-handling code written in matplotlib works across many different GUIs. Let's look at a simple example that reports the *x* and *y* locations in the display and user spaces with a mouse click:

```
from pylab import figure, show
fig = figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])

def onpress(event):
  if not event.inaxes: return
  print 'click'
  print '\tuser space: x=%1.3f, y=%1.3f'%(
```

```
      event.xdata, event.ydata)
  print '\tdisplay space: x=%1.3f, y=%1.3f'%(
      event.x, event.y)

fig.canvas.mpl_connect('button_press_event',
   onpress)
show()
```

Basic events like button and key presses, mouse motion, and canvas drawing are supported across the five GUIs that matplotlib supports, thus matplotlib code written for one GUI will port without changes to another.

Picking

All matplotlib `Artist` primitives define a method `pick` that supports picking so that users can interactively select objects in the plot scene via mouse clicks. Users can also define threshold tolerance criteria in distance units—for example, to define a hit if the mouse click is within five printer's points of the graphical object—or provide a custom hit-testing function. Let's print the data points that fall within five points of the mouse-click location:

```
from pylab import figure, nx, show
fig = figure()
ax1 = fig.add_subplot(111)

# 5 points tolerance
line, = ax1.plot(nx.mlab.rand(100), 'o',
   picker=5)

def onpick(event):
  line = event.artist
  xdata = line.get_xdata()
  ydata = line.get_ydata()
  ind = event.ind
  print 'data: ', zip(nx.take(xdata, ind),
     nx.take(ydata, ind))

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

More elaborate examples, such as defining custom hit-testing functions, are located in the examples directory of the matplotlib source distribution.

Matplotlib Toolkits

Matplotlib supports toolkits for domain-specific plotting functionality that's either too big or too narrow in purpose for the main distribution. Jeffrey Whitaker of the US National Oceanic and Atmospheric Association offers the excellent `basemap` toolkit for plotting data on map projections (http://matplotlib.sourceforge.net/matplotlib.toolkits.basemap.basemap.html). Some of the available projections include cylindrical equidistant, mercator, lambert conformal conic, lambert azimuthal equal area, albers equal area conic, stereographic, and many others. The `basemap` toolkit ships with the `proj4` library and Python wrappers to do the projections. Coastlines, political boundaries, and rivers are available in four resolutions: crude, low, intermediate, and high. The toolkit also supports contouring and annotations. Figure 4 shows an orthographic map projection of the Earth from the perspective of a satellite looking down at 50N, 100W using low-resolution coastlines and from this code:

```
from matplotlib.toolkits.basemap import Basemap
from pylab import nx, show

# don't plot features that are smaller than
# 1000 square km.
map = Basemap(projection='ortho',lat_0=50,
   lon_0=-100, resolution='l',
   area_thresh=1000.)

# draw coastlines, country boundaries, fill
# continents.
map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral')

# draw the edge of the map projection region
# (the projection limb)
map.drawmapboundary()

# draw lat/lon grid lines every 30 degrees.
map.drawmeridians(nx.arange(0,360,30))
map.drawparallels(nx.arange(-90,90,30))
show()
```

Many more sophisticated examples including annotations, alternate projections, and detailed political and geographic boundaries are available in the examples directory of the `basemap` toolkit, which also ships with shape files for the boundaries.

Figure 4. Low-resolution satellite view of the Earth using the matplotlib `basemap` toolkit. Higher-resolution political and geographic boundaries, as well as a wealth of map projections, are available as configuration options in the toolkit.

The Matplotlib API

At its highest level, the matplotlib API has three basic classes: `FigureCanvasBase` is the canvas onto which the scene is painted, analogous to a painter's canvas; `RendererBase` is the object used to paint on the canvas, analogous to a paintbrush; and `Artist` is the object that knows how to use a renderer to paint on a canvas. `Artist` is also where most of the interesting stuff happens; basic graphics primitives such as `Line2D`, `Polygon`, and `Text` all derive from this base class. Higher-level artists such as `Tick` (for creating tick lines and labels) contain layout algorithms and lower-level primitive artists to handle the drawing of the tick line (`Line2D`), grid line (`Line2D`), and tick label (`Text`). At the highest level, the `Figure` instance itself is an `Artist` that contains one or more `Axes` instances—the `subplot` command in Figure 2 creates an `Axes` instance.

The basic drawing pipeline is fairly straightforward. For concreteness, let's look at the `Agg` back end. `Agg` is the core matplotlib raster back end that uses the antigrain C++ rendering engine to create pixel buffers with support for antialiasing and alpha transparency (see www.antigrain.com). `FigureCanvasAgg` creates the pixel buffer, and `RendererAgg` provides low-level methods for drawing onto the canvas—for example, with `draw_lines` or `draw_polygon`. The canvas is created with a reference to `Figure`, which is the top-level `Artist` that contains all other artists. This provides a rigid segregation between `Figure` and the output formats. Let's look at a complete example that uses the `Agg` canvas to make a PNG output file:

```
from matplotlib.backends.backend_agg import \
  FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('agg_demo.png')
```

To create a different output format, such as PostScript, we need only change the first line to `from matplotlib.backends.backend_ps import FigureCanvasPS as FigureCanvas`. The method `canvas.draw()` in our code example creates a back-end-specific renderer and forwards the `draw` call to `Figure`. The `draw` method looks like this:

```
class FigureCanvasAgg(FigureCanvasBase):
  def draw(self):
    renderer = RendererAgg(self.width,
       self.height, ...)
    self.figure.draw(renderer)
```

Every `Artist` must implement the `draw` method; the call to `Figure.draw` calls `Axes.draw` for each `Axes` in the `Figure`. In turn, `Axes.draw` calls `Line2D.draw` for every line in the `Axes`, and so on, until all the matplotlib `Artists` contained in the figure are drawn. Let's look at the code for the top-level `Line2D.draw` method, which closes the circle between the high-level matplotlib `Artists` and the low-level primitive `renderer` methods:

```
class Line2D(Artist):
  def draw(self, renderer):
    x, y = self.get_transformed_xy()
    gc = renderer.new_gc()
    gc.set_foreground(self._color)
    gc.set_antialiased(self._antialiased)
    gc.set_linewidth(self._linewidth)
    gc.set_alpha(self._alpha)
    renderer.draw_lines(gc, x, y)
```

This code illustrates the encapsulation of the back-end renderer from the matplotlib `Artist`: the `Line2D` class knows

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. 2007 annual subscription rates: $45 for Computer Society members (print plus online), $76 (sister society), and $100 (individual nonmember). For AIP society members, 2007 annual subscription rates are $45 (print plus online). For more information on other subscription prices, see www.computer.org/subscribe/ or https://www.aip.org/forms/journal_catalog/order_form_fs.html. Computer Society back issues cost $20 for members, $96 for nonmembers; AIP back issues cost $22 for members.

Postmaster: Send undelivered copies and address changes to *Computing in Science & Engineering*, 445 Hoes Ln., Piscataway, NJ 08855. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Corporation (Canadian distribution) publications mail agreement number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8 Canada. Printed in the USA.

that the `Renderer` instance has a `draw lines` method, but doesn't know what kind of output it renders to, be it a PostScript canvas or GTK `DrawingArea`. Likewise, the renderer and canvas don't know anything about the matplotlib coordinate system or figure hierarchy or primitive geometry types; instead, they rely on the individual artists to do the layout and transformation and make the appropriate primitive renderer calls. Although this design isn't always ideal—for example, it doesn't exploit some of the features in a given output specification—it does keep the back ends reasonably simple and dumb, allowing us to quickly add support for a new format.

Matplotlib has achieved many of its early goals; its current objectives include improving performance for real-time plotting and offering better support for an arbitrary number of scales for a given axis, user coordinate systems, and basic 3D graphics (http://matplotlib.sf.net/goals.html). The latter goal is one of the most frequent requests from matplotlib users.

The requirement for high-quality 3D graphics and visualization in Python is mostly solved via the wrapping of VTK; MayaVi2, which is part of the Enthought Toolsuite (http://code.enthought.com), provides a convenient Matlab-like interface to VTK's fairly complex functionality. Matplotlib provides some rudimentary 3D graphics, and we want to improve them so that our users can rely on basic functionality for surfaces, meshes, and 3D scatterplots without having to depend on the VTK installation. But for anything more sophisticated, we prefer to stick to our core competency—high-quality interactive scientific 2D graphs—rather than trying to replicate the already fantastic graphics provided by VTK and the MayaVi wrappers.

John D. Hunter is Senior Research Programmer and Analyst at Tradelink. His research interests include scientific visualization and event-driven trading strategies. Hunter has a PhD in neurobiology from the University of Chicago. Contact him at jdh2358@gmail.com.