PyPlot

Graphics can be produced in Python via a variety of packages. We will use a Python plotting package that is part of MatPlotLib, for which documentation can be found at matplotlib.org. This handout covers a very small subset of the graphics methods available and provides only a brief description of each one, so use of the documentation web site will be necessary.

The plotting library must be imported, and we will assume in these examples an **import** statement similar to those for `numpy` and `math` as

```
import matplotlib.pyplot as pl
```

enabling simplified calls like

```
import pl.plot(x,y)
```

The plotting routines can be grouped in three different categories. The main set does most of the design and plotting: *plot*, *hist*, *contour*, and *scatter* are general types of graph. Another group adds labels, axes, and related annotation, such as *title* and *xlabel*.The third set starts, completes, and saves graphics, including *show*, *figure*, and *savefig*.

plot and scatter

The ***plot*** function produces connected curve plots or scatter plots, with the capability of combining them. The most basic call is

```
plot( x, y )
```

in which `x` and `y` are one-dimensional, floating-point arrays of coordinates to be connected with a curve. To produce a scatter plot, add a format argument.

```
plot( x, y, 'ro')
```

in which `'ro'` causes the points to be plotted individually with red circles. The color abbreviations and marker or line designators used in formats are in tables at the end of this handout. More than one curve or marker set can be plotted on a graph.

```
plot( x, y, 'y--', u, v, 'b*')
```

in which `x` and `y` must be of the same size and will be plotted with a dashed-yellow curve, and `u` and `v` must be the same size as each other but not necessarily the same size as the `(x,y)` set, and the `(u,v)` set will be plotted with blue stars. To have those curves and scatter plots labeled within a figure, use two different *plot* calls and the `label` keyword argument which provides information for a later call to *legend*.

```
plot( x, y, 'y--', label='smoothed curve')
plot( u, v, 'b*', label='raw data')
```

The ***scatter*** function is more limited and only puts markers at a set of points. A good use for it is to mark the raw data locations under a contour plot.

```
scatter( x, y, marker='o', c='r')
```

will plot a red circle at each (x,y) location.

hist

The ***hist*** function creates a **histogram** – a form of bar chart that reduces a list of numbers to the frequency of their occurrence in bins, which are defined as short subranges of the data, defined by values at the boundaries between ranges. In the simplest form, it only needs a list of numbers.

```
hist( x )
```

will determine the range of values in `x` (highest and lowest values, break that range into 10 evenly spaced bins, count the number of values in each bin, and plot the resulting frequencies. Arguments can be used to change the default behavior in a variety of ways.

```
hist( x, bins=nbins )
```

where *nbins* is an integer number that controls how many bins are used instead of the default 10.

```
hist( x, bins=list_of_bins )
```

where *list_of_bins* is a sequence of numbers that become the actual bin edges, allowing uneven bins to be specified.

```
hist( x, cumulative=1 )
```

changes the histogram values from regular frequencies into *cumulative* frequencies in which the value of each histogram bar is the number of points less than or equal to the right-edge value of the bin. Setting `cumulative=-1` produces a reverse cumulative chart in which each bar is the number of points greater than or equal to the left-edge value.

```
hist( x, range=[x_min,x_max] )
```

restricts the range of the x axis, so that data outside that range will be ignored.

contour and imshow

The ***contour*** and ***contourf*** functions produces contour plots, as might be expected, with *contourf* doing color fill between contours whereas *contour* just draws the curves. Turning a set of data into a contour plot often requires an intermediate step. Looking at the online Gallery for example codes may be particularly helpful in this case.

Contouring routines normally require a two-dimensional array of values arrayed in a grid. If the grid is defined by a set of evenly spaced x values and evenly spaced y values, then a z array in two dimensions that contains all the possible pairs of x and y coordinates can be contoured as simply as

```
contour( x, y, z )
```

If the x and y values do not actually matter, this can be simplified to

```
contour( z )
```

in which case the x and y coordinates will assume each grid point has a spacing of 1 (i.e., the axes will count position numbers on the grid).

The more typical case of having randomly located data requires that a grid be created first, which can be accomplished with the *griddata* function from the mlab library. Also, the default *contour* call does not label the contour lines, which requires interface with the *clabel* function. Both of these concepts are better learned by examining the source code examples in the PyPlot gallery, rather than by a brief description here. However, the following example takes three one-dimensional arrays of the same size, `x`, `y`, and `z`, creates evenly spaced `xg` and `yg` "grid" definitions by dividing the range of those two variables into 20 equal segments, uses the *griddata* function (which must be imported from another sublibrary) to create a two-dimensional `zg` array with interpolated values of `z` at all the possible combinations of `xg` and `yg`, and then plots a contour diagram of the resulting grid.

```
import matplotlib.pyplot as pl
import matplotlib.mlab as mlab
import matplotlib.pyplot as pl
    ⋮
xmin = min(x); xmax=max(x); xinc=(xmax - xmin)/20.0
ymin = min(y); ymax=max(y); yinc=(ymax - ymin)/20.0
xg = np.arange(xmin, xmax, xinc )
yg = np.arange(ymin, ymax, yinc )
zg = mlab.griddata(x, y, z, xg, yg, interp='linear')
pl.contour(xg, yg, zg)
```

The ***imshow*** function also requires as input a two-dimensional array, such as `z` in the contouring examples. Instead of doing contour lines, *imshow* produces a color map. Thus,

```
imshow( z )
```

would produce a color map of `z` as a third dimension over the two-dimensional space defined by its index positions.

One means of creating a two-dimensional array is to read an image file, in which the color or intensity values of the array become the `z` values. This requires importing an additional library. In the following example, `raster.png` is assumed to be an existing image file, PNG format implied by the filetype. The file will be read by the ***image*** function from the `matplotlib.image` library.

```
import matplotlib.pyplot as pl
import matplotlib.image as img
    ⋮
a = img.imread("raster.png")
pl.imshow( a )
```

The preceding calls only reproduce the image in `raster.png` as a PyPlot graph. However, they enable pixel by pixel image manipulation using Python arithmetic and programming on the values in the array `a`. Figuring out exactly what to do with this is an advanced topic.

The *imread* function recognizes the most commonly used image file formats, including PNG, TIFF, JPEG, and GIF.

Both *contour* and *imshow* have additional arguments and graphics resources that can be used to control colors of contours, color maps of the image, axis range and labeling, and labeling. As usual, starting with examples in the online PyPlot gallery to emulate and modify is the best way to learn the options.

fill and fill_between

The ***fill*** function can be used to color a space. In the following example, assume that `x` and `y` are each one-dimensional arrays of the same size, providing coordinates of the vertices of polygon. To fill that polygon with red color, use

```
fill( x, y, 'r')
```

where the color codes are from the same table at the end of this handout that cover marker and line colors. For simple polygons, coordinates may be specified directly. The next call would make a blue rectangle of height 3 and width 5.

```
fill( [0.0, 5.0, 5.0, 0.0], [0.0, 0.0, 3.0, 3.0], 'b')
```

To fill the space between two curves, use ***fill_between***. In this example `y1` and `y2` are two different sets of y-coordinates both corresponding to the same set of x-coordinates. The following call will shade the space between the two curves green:

```
fill_between( x, y1, y2, 'g' )
```

Axes and Labels

For annotation of axes and the top of the graph, use any or all of the following calls in the obvious manner

```
xlabel( 'Label text for x axis' )
ylabel( 'Label text for y axis' )
title( 'Label text for top of graph' )
```

For annotation of individual curves, use ***legend***. With a blank argument list, `legend()`, a box will be plotted in the upper right with text from `label` arguments. If the *plot* statements do not have labels, the following is an example of constructing a legend and moving it from the default position.

```
legend( ('smoothed curve', 'raw data'), loc='upper left')
```

Note the extra parentheses around the list of labels. For the `loc` string, most reaonable combinations of `upper`, `lower` or `center` in vertical with `left`, `right`, or `center` in horizontal will work. See the online documentation for allowable locations and other options.

Text strings can be added anywhere in a graph using the ***annotate*** function. The location of the text string `(x,y)` is in the graph-relative coordinate system, so use coordinates as if they were data point locations is possible. The first example just puts a character string defined as object `s` into the graph at position `(x,y)`:

```
annotate( s, xy=(x,y) )
```

Annotate can also be used to put string `s` at location `(x2,y2)` with an arrow pointing at location `(x,y)`. This requires use of an `arrowprops` keyword, which is best learned by gallery examples.

The ***arrow*** function can be used for putting an arrow on a plot, mostly commonly for North arrows in our class. A common usage is

```
arrow( xstart, ystart, xlength, ylength, head_width=hw, head_length=hl)
```

in which *xstart* and *ystart* are replaced with the coordinates of the arrow's starting point, *xlength* and *ylength* are replaced with length in each direction, and *hw* and *hl* are replaced with the width and length of the arrowhead. All positions and sizes are in the coordinate system of the graph, meaning the coordinate system shown on the axes.

The ***axis*** function provides several different controls on axis appearance. Often, the default choices are good enough and axis is not needed. Here are some useful examples when the default is not adequate.

```
axis([xmin, xmax, ymin, ymax])
```

Sets the minimum and maximum values for each axis, using a list of numbers.

```
axis('off')
```

Turns off axis lines and labels.

```
axis('equal')
```

Changes limits of the axes so that both cover the same range, i.e., so that *xmax - xmin* equals *ymax - ymin*. Useful when two-dimensional shapes need to be preserved, as in mapping.

```
axis('scaled')
```

which changes the *length* of the axes in order to preserve two-dimensional shapes. Also see

```
axis('tight')
```

shows all the data but maximizes the use of the graphics window.

```
axlimits = axis()
```

The blank argument list turns axis into an inquiry function, in which *axlimits* is a list of the *xmin, xmax, ymin, ymax* being used.

Control of one or the other axis limits without needing to specify all of them is possible with ***xlim*** and ***ylim***, which have the arguments `(xmin,xmax)` and `(ymin,ymax)`, respectively. These may be overridden by `axis` calls. Examples:

```
xlim(xmin=0)
```

sets the bottom limit of the x axis while leaving the y axis alone and living the upperlimit of the x axis automatic, while

```
ylim(ymin=-10,ymax=20)
```

sets both limits for the y axis but leaves the x axis to be set automatically.

Completing and saving figures

The ***show*** function tells Python to plot the graph defined by the above calls; usually it is the last call of the graphics program for a single plot.

```
show()
```

The ***figure*** function may not be necessary in our course. It starts a new figure, gives it a number, and optionally changes the size. It is more useful in programs that work on more than one figure at a time, especially for panel plots. To give a figure a number other than 1, such as whatever number is currently stored in `k`, for example,

```
figure(k)
```

The ***savefig*** function is called just before ***show*** to save a file to disk in some format. The main argument is to provide a filename, from which the file format will usually be deduced. The following three examples save a file in the current directory (where the Python program is running), in a subdirectory, and to the Desktop using an absolute path. The file formats will be PNG, PDF, and JPEG, respectively.

```
savefig('figure1.png')
savefig('graphs/figure2.pdf')
savefig('/Users/students/Desktop/figure3.jpg')
```

Colors, Linestyles, and Markers

colors

`'b'`	blue	`'g'`	green	`'r'`	red	`'c'`	cyan
`'m'`	magenta	`'y'`	yellow	`'k'`	black	`'w'`	white

line styles

`'-'`	solid (default)	`'--'`	dashed	`'-.'`	dash-dot	`':'`	dotted

markers

`'.'`	point (default)	`','`	pixel	`'o'`	circle	`'s'`	square
`'v'`	triangle down	`'^'`	triangle up	`'<'`	triangle left	`'>'`	triangle right
`'p'`	pentagon	`'h'`	hexagon	`'d'`	diamond	`'+'`	plus
`'*'`	star	`'x'`	X	`'|'`	vertical line	`'_'`	horizontal line