

Lots of plots with matplotlib

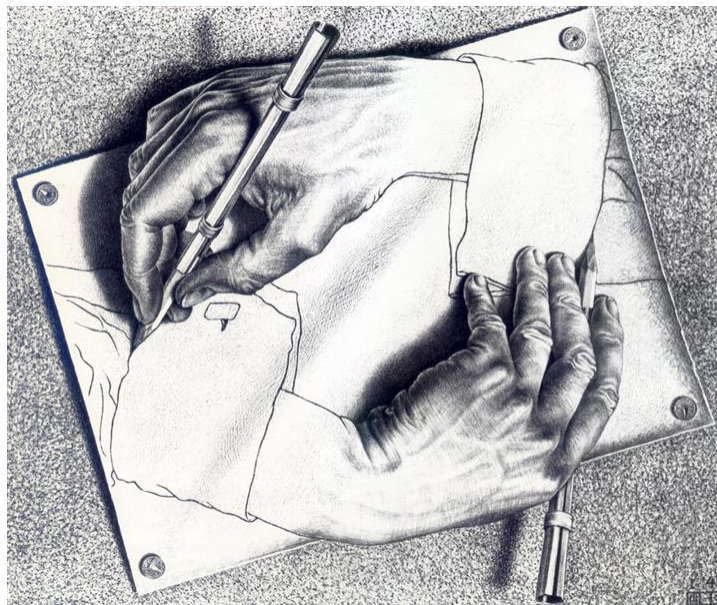
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4

Spring 2025

Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/matplotlib_plus/matplotlib_plus.pdf



Graphics

- `matplotlib` includes many options beyond simple line plots;
- Several plots can be grouped into a single figure;
- 3D plotting options are available;
- In some cases, we will want to read a datafile first;
- A plot can be improved and annotated;
- Abstract graphs and networks can be created with `vizgraph` and `pydotplus`;

1 A variety of plots

We have made many line plots, a few histograms and examples of filled figures. But the `matplotlib` library allows us many options for the graphical presentation of data:

- **line plot**, a sequence of points (x_i, y_i) connected by lines;
- **line plots**, several sets of points (x_i, y_i) , each connected by lines;
- **subplots**, a single figure involving several plots;
- **area between plot**, the area between two curves is filled with color;
- **scatter plot**, a set of points (x_i, y_i) indicated by markers;

- **bar plot**, a set of data y_i displayed as bar heights;
- **horizontal bar plot**, data y_i displayed as bar lengths (good for labels);
- **histogram**, data to be grouped into bins before bar plotting;
- **contour lines**, contour lines of a function $f(x, y)$;
- **contour colors**, contour colors of a function $f(x, y)$;
- **surface plot**, a 3D plot of the surface $z = f(x, y)$;
- **vector plot**, a 2D plot using arrows for a gradient or flow field;
- **filled polygons**, a plot of colored polygons;
- **USA plot**, a schematic of the USA;
- **graph plot**, a set of labeled points p_i , and a list of two-way connections ($p_i \leftrightarrow p_j$);
- **digraph plot** of a tree structure, representing a branching process;
- **digraph plot** of a network, a set of labeled points p_i , and a list of one-way connections ($p_i \rightarrow p_j$).

2 Getting the data files

Many of the following plotting exercises start by reading data from a file. For instance, the first exercise reads from the file *bulgaria_data.txt*. Thus, to carry out the exercise on your computer, you need to download a copy of this file.

As we have seen in previous exercises, if you do not have a copy of this file, it can be downloaded from the class website. Another way to get it is to go to

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/math1800_2023.html

which will give you a menu that includes the files we are interested in.

For this set of exercises, you will want to copy the following data files:

- *album_data.txt*
- *bulgaria_data.txt*
- *geyser_data.txt*
- *price_data.txt*
- *schoolyear_data.csv*
- *snowfall_data.txt*
- *volcano_data.txt*

To read a text file (those ending in *.txt*, use the command like

```
data = np.loadtxt ( 'bulgaria_data.txt' )
```

In this case, `data` will be a 30×2 array of years and populations, corresponding to the fact that the text file contains 30 lines, each containing a pair of values.

The *.csv* files (“comma separated values”) are a little more complicated to read, and instructions for how to do that will be given with the corresponding exercise.

3 The graphviz and pydotplus libraries

For some of the later exercises, we will need to make some plots of mathematical graphs and networks that `matplotlib` can’t handle. Instead, we will rely on `graphviz` and `pydotplus`. If you want to do these exercises, you may need to download these libraries.

The `graphviz` home page is <https://graphviz.org/>.

Google Colab It's already available!
Windows: download and run graphviz-3.0.0 installer
Macintosh with MacPorts: sudo port install graphviz
Linux: pip3 install graphviz

The `pydotplus` home page is <https://pypi.org/project/pydotplus/>. The library is already available on Google Colab. If you are working on your own local copy of Python, you may be able to install `pydotplus` with the command:

```
pip install pydotplus
```

4 A Line Plot of the BULGARIA Data

Load the text data file `bulgaria_data.txt`, which records a sequence (y_i, p_i) , of years and population. The population counts are given at somewhat irregular intervals. Instead of connecting the data points in a curve, we want to show just the data points.

```
import matplotlib.pyplot as plt
import numpy as np

filename = 'bulgaria_data.txt'
data = np.loadtxt ( filename )
#
# Split data into separate year and population vectors.
#
year = data[:,0]
population = data[:,1]
#
# Create a plot of the data points.
#
plt.plot ( year, population, 'b-' ) # Draw a blue line
#
# Show the plot.
#
plt.show ( )
```

The result of the Bulgaria example is not particularly pretty, and could be more professional-looking. We can try again, adding a title, axis labels, and a grid. The revised plotting code would be:

```
plt.plot ( year, population, 'r-' ) # a red line this time
plt.grid ( True )
plt.xlabel ( '<— Year —>' )
plt.ylabel ( '<— Population —>' )
plt.title ( 'The population of Bulgaria' )
plt.show ( )
```

We might wish to draw attention to the fact that, in 1985, the population reached its maximum value of 8,975,291. We can do this using the `annotate()` function, and we can add a red dot to highlight the position.

```
plt.annotate ( 'Max population', [ 1985, 8975291 ] )
plt.plot ( 1985, 8975291, 'ro' )
```

If we want to save this plot, we might simply call `plt.savefig('bulgaria.png')`; This call should come before the call to `plt.show()`.

You can use pretty much any name you want for the graphics file that you create. Python guesses the actual file format from the filename extension. Aside from `'png'`, other formats include `pdf` and `svg`.

5 Line Plots: the PRICE Data

The file *price_data.txt* is a table of average monthly prices for 11 consumer products, between February 2008 and February 2018. There are 241 records, and each record contains 13 items: the month, the year, and the prices of 11 goods. Columns 3, 10 and 13 contain the prices of bananas, gas, and milk, respectively. We want a single plot that shows line plots for all three items together.

For our “x” coordinate we will create a vector of length 241 which represents the month index.

```
price = np.loadtxt ( 'price_data.txt' )
month_num = price.shape[0]
month = np.arange ( month_num )
```

Now we can simply issue a plot command for each pair of (month,price) values. All three curves will appear in the same figure. To make the curves more visible, we request `linewidth=3`, rather than the default value of 1.

```
plt.plot ( month, price[:,2], linewidth = 3 )
plt.plot ( month, price[:,9], linewidth = 3 )
plt.plot ( month, price[:,12], linewidth = 3 )
```

Now that we have our three curves plotted, we can “decorate” the figure with a grid, labels and a legend. A legend helps to interpret a plot containing multiple curves, by displaying a small chart pairing a line color with a line label.

```
plt.grid ( True )
plt.legend ( [ 'Bananas', 'Gas', 'Milk' ] )
plt.xlabel ( '<— Month index —>' )
plt.ylabel ( '<— Price ($) —>' )
plt.title ( 'Prices, Feb 2008–Feb 2018' )
```

6 Subplots: the PRICE Data (again!)

Again, we read the price data. This time, we also set up an array of labels to be used when we make our subplots.

```
price = np.loadtxt ( 'price_data.txt' )
month_num = price.shape[0]
month = np.arange ( month_num )

label = [ \
    'Month', 'Year', 'Bananas', 'Oranges', 'Bread', \
    'Tomatoes', 'Chicken', 'Electricity', 'Eggs', 'Gasoline', \
    'Beef', 'Heating gas', 'Milk' ];
```

We want to show the prices of six of the commodities in the *price_data.txt*, but each item will appear in its own plot. We will lay out the plots in two rows and three columns. To do this, we first issue the `subplots(rows,columns)` command:

```
plt.clf ( )

m = 2
n = 3
fig, ax = plt.subplots ( m, n )

fig.suptitle ( '2x3 Price Subplots' )
```

The variable `fig` is a way to refer to the entire figure. Meanwhile `ax[i,j]` is a way to refer to the subplot (or “axis”) in row `i` and column `j`.

Now we can work through each subplot. Notice that our plotting commands are prefaced by a reference to `ax[]`, to indicate which subplot we are working on. The variable `k` allows us to choose the column of data to plot. We want to skip the first two columns in the data file, because those are simply the month and years.

```
k = 1
for i in range ( 0, m ):
    for j in range ( 0, n ):
        k = k + 1
        ax[i,j].plot ( month, price[:,k] )
        ax[i,j].set_title ( label[k] )
plt.show ( )
plt.close ( )
```

7 An “Area Between” Plot: the INTEGRAL Data

The integral of a function $f(x)$ over the limits $a \leq x \leq b$ can be pictured as the area “under the curve”, that is, the area between the x axis and the curve $y = f(x)$. If the curve goes below the x axis, this portion counts as negative area.

As an example, we’d like to plot $y = \sin(x)$ and illustrate the integral from $\frac{\pi}{3} \leq x \leq \pi$, displaying positive area in blue and negative area in red. The command `fill_between()` fills in the colors.

```
x1 = np.linspace ( -0.25, 6.5, 101 ) # plot range
y1 = np.sin ( x1 )

x2 = np.linspace ( np.pi / 3.0, np.pi, 101 ) # positive integral range
y2_hi = np.sin ( x2 )
y2_lo = np.zeros ( 101 )

x3 = np.linspace ( np.pi, 4.0 * np.pi / 3.0, 101 ) # negative integral range
y3_hi = np.sin ( x3 )
y3_lo = np.zeros ( 101 )

plt.plot ( x1, y1, 'k-', linewidth = 2 ) # Curve
plt.fill_between ( x2, y2_lo, y2_hi, linewidth = 0, color = 'b' ) # Positive part
plt.fill_between ( x3, y3_lo, y3_hi, linewidth = 0, color = 'r' ) # Negative part
plt.plot ( [ -0.25, 6.5 ], [ 0.0, 0.0 ], 'k-' ) # Draw x-axis

plt.show()
```

If we like the plot, we can label it and save it.

```
plt.grid ( True )
plt.title ( 'Integral = Blue area - red area' )
plt.xlabel ( '<— x —>' )
plt.ylabel ( '<— y = sin(x) —>' )
filename = 'integral_between.png'
plt.savefig ( filename )
print ( ' Graphics saved as "' + filename + "' )
plt.show()
plt.close ( )
```

8 A Scatter Plot: the GEYSER Data

The file *geyser_data.txt* contains 272 observations of the Old Faithful geyser. Each record lists the eruption duration in minutes, and the pause in minutes until the next eruption. A line plot of the durations, or of the pauses, doesn't show a clear pattern. However, it might seem reasonable that the duration and successive pause variables are related. A long eruption might mean a long subsequent wait, for instance. Such patterns can be investigated using a scatter plot, in which we simply put a mark at each point (x, y) for which we have data.

Load the text data file *geyser_data.txt*, and split the data:

```
data = np.loadtxt ( filename )
duration = data[:,0]
pause = data[:,1]
```

We do **not** want to use the `plot()` command. Instead we use `scatter`:

```
plt.scatter ( duration , pause )
```

If we wish to use an 'o' for the marker, of size 5, and color red, we could give the fancier command:

```
plt.scatter ( duration , pause , s = 5 , c = 'r' , marker = 'o' )
```

More information is available by `help (plt.scatter)`.

This example shows how a scatter plot can reveal interesting patterns in data. What relationship between eruption duration and pause duration is suggested here?

9 A Bar Plot: the ALBUM Data

Line plots connect pairs of data values in which x is steadily increasing and y is varying smoothly. A bar plot also considers data pairs, but the variable playing the x role might be an increasing numeric value, or it might simply be a set of named categories. The main purpose of a bar plot is to emphasize the variation in height of the bars, inviting the user to draw a conclusion as to the pattern.

Our first example considers the sales of music albums over a period of years. We could instead display the data using a line plot, but, by using bars, the change in behavior over the years is much more dramatically displayed.

The file *album_data.txt* lists the year, and total music album sales, for each year from 2007 to 2017. We begin by reading the data from the file, and copying out the year and sales information separately:

```
filename = 'album_data.txt'
data = np.loadtxt ( filename )
year = data[:,0]
sales = data[:,1]
```

To create a bar plot, we can call the `matplotlib` function `bar()`:

```
plt.bar ( year , sales )
```

One of the useful optional parameters is `width=value`, which defaults to 0.8. This draws bars that are only 80% of the maximum width, thus leaving gaps to make the bars more clearly separate. You can easily specify this value, and adjust it as you like. Similarly, the parameter `color` has the default value 'b', but you can change the color of the bars to green by specifying `color='g'` instead.

Especially for bar plots, specifying a background grid via `plt.grid (True)` makes it easier for the viewer to compare the heights of different bars.

10 A Horizontal Bar Plot: the SCHOOLYEAR Data

Usually, a bar plot exhibits the bars vertically. The length of the bar represents a count, but what does the position mean? Frequently, the bars represent counts of data over bins of equal size, and evenly spaced. But other times, the data is associated with different people, or different countries, or other items, so that each bar needs a readable label.

For such data, a horizontal bar plot might be best. The labels might appear on the left, with a horizontal bar to the right, as suggested by this simple typewritten example:

```
Votes in favor: xxxxxxxxxxxx
Votes against: xxxxxxxxxxxxxxxxxxxx
Votes withheld: xxxx
```

For this exercise, our data will involve the length of the school year in days, for various countries. We need to read a CSV data file, *schoolyear_data.csv*, in which the first item is the name of a country, and the second is the number of data in a school year. The first four lines of this file are:

```
"Country", "Days"
"Belgium (French)", 175
"Belgium (Flemish)", 160
"British Columbia", 185
```

so you can see that line #1 gives names for the two pieces of data, and subsequent lines are a country name in quotes, a comma, and a number of days. Data like this is very common, but a little tricky to read.

We will use `genfromtxt()` to read the data. But now, because the data is not simply numeric, we need to include some information describing the type of each item. We describe the type of the first item as `'U20'`, that is, a Unicode text string of up to 20 characters. The second item is listed as `'i4'`, that is, a 32 bit signed integer. So here's how we read from the file, split out our two data items, reverse the order of the data (so that the alphabetical order reads down, not up!) and make an index variable `country`:

```
types = [ 'U20', 'i4' ]
data = np.genfromtxt ( 'schoolyear_data.csv', dtype = types, delimiter=',', names = True )

days = data[ 'Days' ]
name = data[ 'Country' ]

days = np.flip ( days )
name = np.flip ( name )

n = len ( days )
country = np.linspace ( 1, n, n )
```

The point here is that the bars will be meaningless unless we can label them, preferably with the country name. That would be very difficult with vertical bars, but we may be able to manage it with horizontal bars, since the labels will read across. So we call `barh()` to make the bars, and then specify the country names as “y tick marks”

```
plt.barh ( country, days )
plt.grid ( True )
plt.title ( 'Schoolyear lengths', fontsize = 16 )
plt.xlabel ( '<— Days —>', fontsize = 16 )
plt.yticks ( country, name, fontsize = 6 )
```

We specify the `fontsize` in order to keep the country names small enough to fit. Again, a background grid might be helpful for comparing the lengths of various bars.

11 A Histogram: the SNOWFALL Data

Suppose we want to display a large set of observations of some numerical quantity, (x_1, x_2, \dots, x_n) . If the data has some slowly varying trend, a line plot might be suitable. If the data is not too numerous, a bar plot might work. But if the data is numerous and somewhat chaotic, we are better off with a *histogram*, which simplifies the data by grouping it into a small number of bins. A histogram then shows how many data items fall into each range.

When preparing a histogram, it's important to pick a reasonable number of bins for the plot. A good number of bins will allow the data to suggest a smoothly varying behavior, for instance. Specifying too few or too many bins will result in an ugly and uninformative plot.

The students at Michigan Technical University have been collecting snowfall data for years and updating the records at

<https://www.mtu.edu/alumni/favorites/snowfall/>

The file *snowfall_data.txt* contains data from 1890 to 2022. Column 9 (counting from 0!) of the data lists the total snowfall for that year. That is the data we want to display.

```
data = np.loadtxt ( filename )
inches = data[:,9]

plt.hist ( inches , rwidth = 0.95 )
```

The `rwidth=0.95` argument specifies that the bars in the plot should be just 95% of their maximum width.

The argument `bins=n` can be used to specify the number of bins to be n . If this is not specified, a suitable value is chosen automatically. Experiment with choosing this value yourself. Even small changes to n can sometimes make a noticeable difference in the plot. As stated before, using n too large or too small will result in an awfully ugly plot.

What features of the data does the histogram allow you to see right away?

To understand the difference between a histogram and a bar plot, try calling `plt.bar()` directly for this same data. Does the bar plot do a good job of presenting the information?

12 A Contour Line Plot: the VOLCANO Data

We may have a collection of measurements of temperature, or elevation, or pollution levels, sampled at regular points on an (x, y) grid. This means that we are really interested in a 3D plot, of $z(x, y)$. One way to display such data on a 2D screen or piece of paper is to use some kind of contour plot. On a map of a mountainous area, for instance, contour lines are used to indicate points that have the same height: 100 meters, 200 meters, and so on. By choosing the right number of such contour lines for your data, you can suggest the pattern formed by your 3D data.

A map of a volcano has been created by creating an 87×67 grid of (x, y) points, and recording a table of the vertical height $z(x, y)$ at each point. This data has been stored in the text file *volcano_data.txt*. The first 5 rows and columns of this data look like:

```
1.00e+02 1.00e+02 1.01e+02 1.01e+02 1.01e+02 ...
1.01e+02 1.01e+02 1.02e+02 1.02e+02 1.02e+02 ...
1.02e+02 1.02e+02 1.03e+02 1.03e+02 1.03e+02 ...
1.03e+02 1.03e+02 1.04e+02 1.04e+02 1.04e+02 ...
1.04e+02 1.04e+02 1.05e+02 1.05e+02 1.05e+02 ...
...
```


We can read the z data from the file using `np.loadtxt()`.

```
zmat = np.loadtxt ( 'volcano_data.txt', dtype = 'f', delimiter = ' ' )
```

Here, `dtype='f'` is saying the data involves real numbers, while `delimiter = ' '` indicates that values are separated by blank spaces, rather than, say, commas, as we saw in CSV files.

In order to use the contour plotting function, we need to supply values for (x, y) as well as z . For this plot, the exact locations don't matter, and we will just assume that the data is equally spaced. In that case, we can use `np.linspace()` to make lists of x and y coordinates. We actually need to create a two-dimensional table of (x, y) data. This is done by calling `np.meshgrid()`.

```
m, n = zmat.shape
xvec = np.linspace ( 0, n-1, n ) # xvec = ( 0, 1, ..., n-1 )
yvec = np.linspace ( 0, m-1, m )
xmat, ymat = np.meshgrid ( xvec, yvec )
```

Now we are ready to call `plt.contour()`. We include a value for the number of contour levels. We can also specify that the contour lines have a color that varies with value. To do so, we specify a particular colormap. In order to use this option, we must also load the `cm` library from `matplotlib`:

```
from matplotlib import cm
levels = 15
plt.contour ( xmat, ymat, zmat, levels, cmap = cm.coolwarm )
```

13 A Contour Color Plot: the VOLCANO Data

You may have noticed that using color for the contour lines of the volcano data made it easier to “read” the image. An even more vivid picture can often be made by dropping the contour lines and simply using color by itself.

We have already seen in the previous discussion how to create the arrays `xmat`, `ymat`, `zmat`. To create a color contour plot, we call `plt.contourf()` instead of `plt.contour()`.

Since the colors you choose will dominate the image, it can be important to find a colormap that suits your taste. There is a large set of such maps available in `matplotlib`. You might like to redraw your plot using one of the following:

- `viridis`
- `Greys`
- `binary`
- `PiYG`
- `twilight`
- `Pastell`
- `flag`

Finally, note that you may be able to get an even more dramatic image by combining filled color contours with contour lines. In that case, however, you should plot the filled color first, and then request contour lines; otherwise, the contour lines will disappear underneath the color polygons!

14 A Surface Plot: the VOLCANO data

The volcano data, of course, represents a 3-dimensional object. We can get a 3D plot if we first import the appropriate function

```
from mpl_toolkits.mplot3d import Axes3D
```

and then after defining the data in the usual way, request a surface plot:

```
fig = plt.figure ( )
ax = fig.add_subplot ( 111, projection = '3d' )
ax.plot_surface ( xmat, ymat, zmat, \
    cmap = cm.coolwarm, edgecolor = 'none' )
```

When the surface plot is displayed to the screen, you should be able to manipulate it, turn it around, change the angle and so on. We will see this idea in some more detail with the “sombbrero” function next.

15 A Surface Plot: the SOMBRERO Function

A contour plot can also be useful to examine the behavior of a function $z = f(x, y)$; even better is a 3D surface plot. In this case, we have to generate the (x, y) grid values, and set up a way to evaluate the function.

Because we want to do 3D plotting, we have to import the `Axes3D` library from `matplotlib`. We will also try using a colormap for the surface, so let's import `cm` as well.

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

For this example, we are interested in the function $z = \sin(r)/r$ where $r = \sqrt{x^2 + y^2}$. Since our plot domain will include the point $(0,0)$, we might want to replace the `nan` value at 0 by the value 1.

```
xvec = np.linspace( -8.0, 8.0, 33 )
yvec = np.linspace( -8.0, 8.0, 33 )
xmat, ymat = np.meshgrid ( xvec, yvec )
rmat = np.sqrt ( xmat**2 + ymat**2 )
zmat = np.sin ( rmat ) / rmat
zmat[rmat==0] = 1.0
```

At this point, having created the arrays `xmat`, `ymat`, `zmat`, we could easily create a contour line or color contour plot of our data. You may want to try this, so that you can compare these contour results with the surface plot we are about to make. While surface plots can be much more dramatic, it is sometimes the case that a simple contour plot is more readable.

Because we want to make a 3D plot, we have to use a more elaborate way to define our figure and axis. Then we call `ax.plot_surface()` to make the surface plot. Notice that we again can choose a colormap. By default, the surface will be covered by grid lines, which can be quite distracting. For our plot, we can suppress these gridlines by specifying `edgecolor='none'`.

```
fig = plt.figure ( )
ax = fig.add_subplot ( 111, projection='3d' )
ax.plot_surface ( xmat, ymat, zmat, \
    cmap = cm.ocean, edgecolor = 'none' )
```

16 A Vector Plot: the FLOW Function

Suppose we want to visualize the direction and magnitude of the wind over some 2D region. While a contour plot deals with a scalar function $z(x, y)$, now we are considering a vector function $(u, v)(x, y)$. The standard way to represent a velocity field like this is to draw arrows at each point (x, y) , of the appropriate direction and magnitude.

Another case in which a vector plot is needed is when we are trying to minimize a function $z(x, y)$, in which case we want to plot the gradient vectors $(dz/dx, dz/dy)(x, y)$.

The `matplotlib` function `quiver()` can produce the plots we want. All we need to supply is the arrays of x, y, u, v values.

In the following example, we consider a flow field over the unit square, for which we have an explicit formula:

$$u = -(x^4 - 2x^3 + x^2)(2y^3 - 3y^2 + y)$$

$$v = (2x^3 - 3x^2 + x)(y^4 - 2y^3 + y^2)$$

In the usual way, we use `np.linspace()` to define vectors `xvec` and `yvec`, then `np.meshgrid()` to create tables `X` and `Y`. Using the flow formula above, we can then create tables `U` and `V`.

Now it's simple to set up our plot:

```
m = 16
n = 16

xvec = np.linspace ( 0, 1.0, n )
yvec = np.linspace ( 0, 1.0, m )
X, Y = np.meshgrid ( xvec, yvec )

U = - ( X**4 - 2 * X**3 + X**2 ) * ( 2 * Y**3 - 3 * Y**2 + Y )
V = ( 2 * X**3 - 3 * X**2 + X ) * ( Y**4 - 2 * Y**3 + Y**2 )

plt.quiver ( X, Y, U, V, color = 'c' )
plt.axis ( 'Equal' )
plt.show ( )
```

Here, the extra argument `color='c'` has the arrows drawn in cyan color.

Additional arguments to `quiver()` can alter the arrow length or thickness. Vector plots are a fundamental tool when working in fields such as computational fluid dynamics, electromagnetism, and climate modeling.

17 Filled Polygons: the ELL data

Many geometric images can be constructed by combining a number of colorful polygons. A checkerboard, for instance, can be drawn using 32 red and 32 black squares. A 6×10 rectangle can be tiled by the 12 pentominoes (distinct shapes of 5 squares), each of which can be shown in a different color. A polygon can be triangulated, with each triangle a different color.

Such images are known as *filled polygon plots*. The `matplotlib` library offers the function `plt.fill()` to display such figures. The format of the call can be

```
plt.fill ( x, y ) # one polygon, default color
plt.fill ( x, y, c ) # one polygon, color 'c'
plt.fill ( x1, y1, x2, y2, ..., xn, yn ) # n polygons
plt.fill ( x1, y1, c1, x2, y2, c2 ) # 2 polygons, colored 'c1', 'c2'
```

Our target for this exercise is to create an L-shaped figure composed of three polygonal shapes colored blue, red, and cyan. Here is a sketch of what we want:

```
4 +-+
  |B|
3 + +
  |B|
```

```

2 + +-+----+
  |B B|R R|
1 +-----+ +
  |C C C|R|
0 +-----+ +
  0 1 2 3 4

```

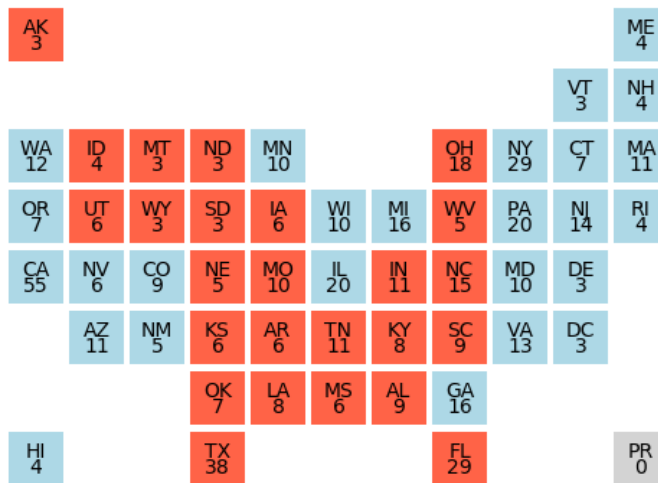
You should see that 'B' polygon has $x1=[0,2,2,1,1,0]$, $y1 = [1, 1, 2,2,4,4]$ and $c1='b'$. Determine $x2$, $y2$, $c2$ and $x3$, $y3$, $c3$ for the 'C' and 'R' polygons. Call `plt.fill()` to create the filled polygon version of this plot.

You may notice that your plot is slightly distorted. Squares are rectangular instead of squares. This is because `matplotlib`, like `MATLAB`, chooses by default a rectangular picture frame, and stretches that data to fit it, if necessary. To avoid this unpleasant habit, we can simply say

```
plt.axis ( 'equal' )
```

18 Filled Polygons: the USA data

In many times, a map of the US is displayed, with each state represented as a square, arranged roughly in the proper shape. The squares can be colored in a way that indicates cold and warm, Republican and Democrat, high, medium and low income, and so on. Using the same idea of filled polygons that we saw in the ELL example, it is possible (with patience), to create images like the following:



Since the details are tedious, we will not go through the trouble of suggesting how the plot is made. Looking at the plot, it should be clear how to reproduce a similar result.

19 A Graph Plot: the NETWORK data

Mathematically, a graph is a set of *nodes*, some of which are connected by *edges*. (A tree is a special kind of graph which is connected, and has no circuits.) In a graph, the edges usually represent two-way connections. We will see one-way connections in a moment, in the directed graph example.

We will again use `graphviz` for this example. The primary difference from the tree plot is that now we import the `Graph` library.

```
from graphviz import Graph
```

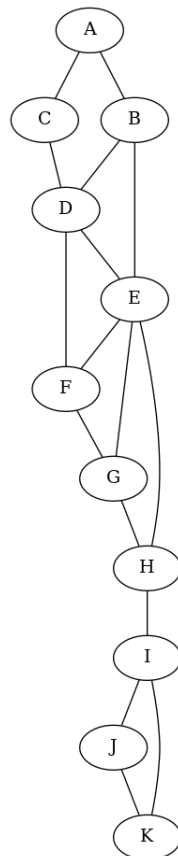
We begin by calling `Graph` to create an object we will name `dot`.

```
dot = Graph ( comment = 'A network ', format = 'png' )
```

Now, just like we did for the tree example, we must list the nodes, with a short identifier and a label, followed by a list of the edges. This procedure will seem identical to what we did for the tree, and so we leave most of the work to you. Here is just a bit of the commands you will need:

```
dot.node ( '1', 'A' )  
...  
dot.node ( '11', 'K' )  
  
dot.edge ( '1', '2' )  
...  
dot.edge ( '10', '11' )
```

To fill in the details, look at the information in the following plot:



Once you have specified the nodes and edges, make your plot:

```
dot.render ( 'network_graph.dot', view = True )
```

20 A Digraph (directed graph) plot: the GENEALOGY data

A tree plot is way to illustrate a process which involves branching or dependence. A tree plot can illustrate all the paths that start at the initial point and branch one way or another at decision points.

The locations on a tree are called *nodes*, and the node representing the initial state is called the *root node*. If we assign each node an identifying index, then the structure of the tree can be described if each node identifies its “parent node”, which connects it back to the root node.

In order to make our plot, we are going to need to install **graphviz** and **pydotplus**, and then import the directed graph library from **graphviz**.

```
from graphviz import Digraph
```

For our data, we will assume we have a simple sort of genealogy, in which one person is the ancestor (and hence will be represented by the root node). This ancestor had several immediate descendants, and so an arrow will be directed from the ancestor node to the nodes of each of these descendants. If any of these descendants had children, more nodes and arrows will be drawn, in a natural way.

The **graphviz** library allows us to specify labels for each node, and the arrows we wish to have drawn from one node to another.

We begin by creating an object we name **dot**, adding a comment, and indicating that the final figure should be saved as a **png** file.

```
dot = Digraph ( comment = 'A genealogy tree', format = 'png' )
```

Now we want to describe the nodes. We begin by requesting that the nodes be drawn with an egg shape. Then we list each node, giving a short name (which might as well be just a quoted numerical index), and a label, for each.

```
dot.attr ( 'node', shape = 'egg' )
dot.node ( '1', 'Alan' )
dot.node ( '2', 'Bert' )
dot.node ( '3', 'Chad' )
dot.node ( '4', 'Dian' )
dot.node ( '5', 'Enid' )
dot.node ( '6', 'Fran' )
dot.node ( '7', 'Gert' )
dot.node ( '8', 'Hank' )
dot.node ( '9', 'Iona' )
dot.node ( '10', 'Jean' )
dot.node ( '11', 'Kate' )
dot.node ( '12', 'Lynn' )
```

Now we specify the connections between nodes:

```
dot.edge ( '1', '2' )
dot.edge ( '1', '3' )

dot.edge ( '2', '4' )
dot.edge ( '2', '6' )

dot.edge ( '3', '5' )
dot.edge ( '3', '10' )
```

```

dot.edge ( '4', '7' )
dot.edge ( '4', '8' )
dot.edge ( '4', '9' )

dot.edge ( '5', '11' )
dot.edge ( '5', '12' )

```

Finally, we can ask for a printed description of the plot, as verification, and then we render the image to a file, to be called *genealogy_tree.dot.png*, and optionally display the plot to the screen.

```

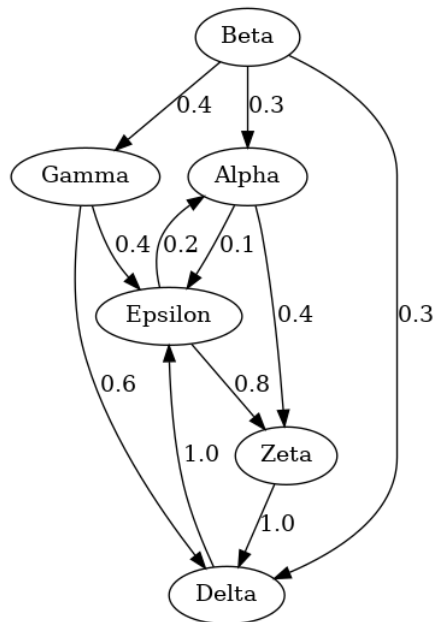
print ( dot.source )
dot.render ( 'genealogy_digraph.dot', view = True )

```

21 A DiGraph Plot: the WEB data

A directed graph displays nodes connected by one-way edges. There may be a value associated with each edge, representing a weight, length or probability of access. A good display should include that information.

Let us consider the following network, and assume that nodes 1 through 6 are labeled Alpha, Beta, Gamma, Delta, Epsilon and Zeta, respectively.



To create a corresponding image, we will again use the `Digraph` function from the `graphviz` library.

```

dot = Digraph ( comment = 'A network', format = 'png' )

```

We will specify nodes as before, but this time, we will add a third argument to the edge specifications, a label to be printed, which indicates some property of that edge. Working from the plot above, you should be able to fill in the missing details:

```

dot.node ( '1', 'Alpha' )
...
dot.node ( '6', 'Zeta' )

dot.edge ( '1', '5', '0.1' )

```

```
dot.edge ( '1', '6', '0.9' )
...
dot.edge ( '6', '4', '1.0' )
```

And of course, once we are done with the specifications, we can view our plot and save it as a file:

```
dot.render ( 'web-digraph.dot', view = True )
```

22 Conclusion

A variety of plots can be made for illustrating computational work. Once the basic plot has appeared, it's important to consider ways to improve or “decorate” it. Line plots may look fine on a computer screen, but usually should be drawn with a larger `linewidth` for clarity, especially when being displayed in print or on a slide presentation. A background grid improves the observer's ability to estimate or compare data in the plot. Labels, titles, and legends change a plot from a picture to a “document”, something that tells a story even if you aren't there to state it. Occasionally, it's necessary to force a plot to use the same units in x and y , especially when a geometric object is being displayed.

When we began plotting, we simply used the `plot()` command from `matplotlib`. However, for 3D plotting and certain other applications, it may be useful or necessary to work with functions like

```
fig = plt.figure ( )
ax = fig.add_subplot ( 111, projection='3d' )
```

or

```
fig, ax = plt.subplots ( )
```

You should try to find more examples of these functions and get to know them better.

The `matplotlib` library currently doesn't have good abilities for drawing mathematical trees, graphs and networks, and so we have looked at `graphviz`. These kinds of plots are constantly coming up in computational, mathematical and scientific work. It's important to pick up the skills you need in order to get plots that express your ideas. The `graphviz` package has online help and a useful document to help you.