2_plotting

April 16, 2024

1 Plotting data

File as PDF

The graphical representation of data plotting is one of the most important tools for evaluating and understanding scientific data and theoretical predictions. However, plotting is not a part of core Python but is provided through one of several possible library modules. The de factor standard for plotting in Python is MatPlotLib. There are nevertheless several other very good modules like PlotLy, Seaborn, Bokeh and other available.

Because MatPlotLib is an external library (in fact it's a collection of libraries) it must be imported into any routine that uses it. MatPlotLib makes extensive use of NumPy so the two should be imported together. Therefore, for any program for which you would like to produce 2-d plots, you should include the lines

```
import matplotlib.pyplot as plt
```

This already implies one way, the implicit interface of `pyplot` to create figures and plot. In general there are two interfaces:

- an **implicit** "pyplot" interface that keeps track of the last Figure and Axes created, and adds Artists to the object it thinks the user wants.
- an **explicit** "Axes" interface that uses methods on a Figure or Axes object to create other Artists, and build a visualization step by step. This has also been called an "object-oriented" interface.

We will use most of the the the `pyplot` interface as in the examples below. The section *Additional Plotting* will refer to the explicit programming of figures.

The plotting libraries are extremely rich and we will not be able to cover all plot types and details here. This is therefore more a collection of code examples, you may use as inspiration.

[131]:
```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
```

We can set some of the parameters for the appearance of graphs globally. In case you still want to modify a part of it, you can set individual parameters later during plotting. The command used here is the

```
plt.rcParams.update()
```

function, which takes a dictionary with the specific parameters as key.

[167]:
```
plt.rcParams.update({'font.size': 10,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 10,
                     'xtick.labelsize' : 9,
                     'ytick.labelsize' : 9,
                     'legend.fontsize' : 8,
                     'contour.linewidth' : 1,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',
                     'figure.figsize': (4, 3),
                     'figure.dpi': 150 })
```

1.1 Simple Plotting - Implicit Version

There are several levels of of access to plotting available in MatPlotLib. We will use most of the time the commands that use some defaults. This is simple as the layout of the graphs is automatically adjusted by MatPlotLib. At the end of this section, we will also shortly address more advanced access

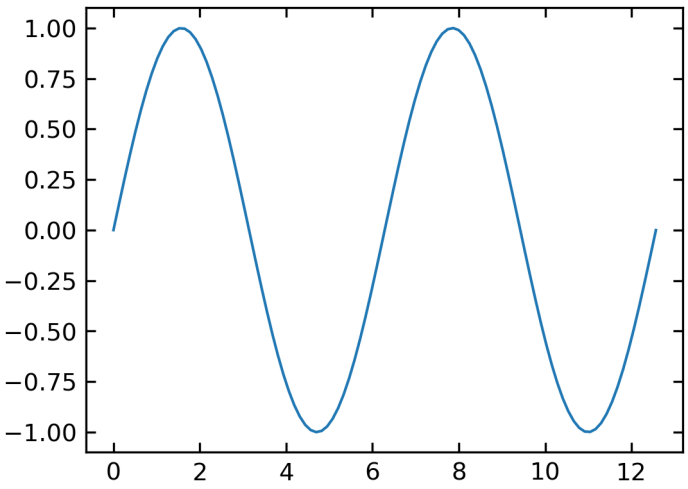
1.1.1 Line Plot

A line plot ist created with the

`plt.plot(x,y)`

command. You may, however, modify in the parameters the appearance of the plot to a scatter plot. But by default, it creates a line plot.

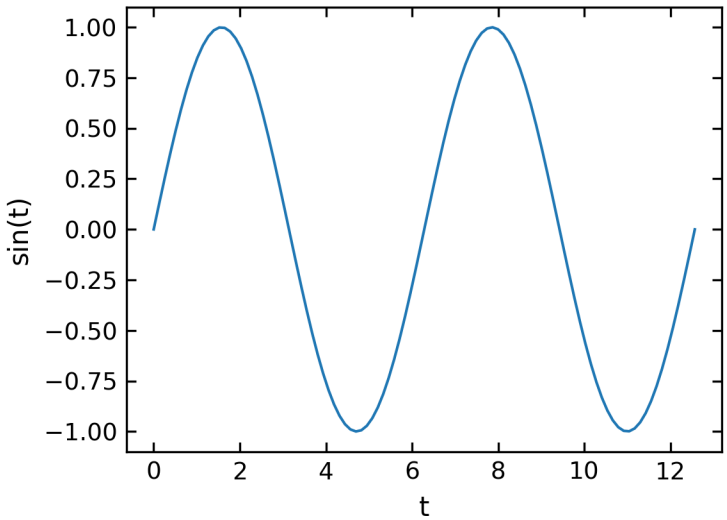
[89]:
```
x = np.linspace(0, 4.*np.pi, 100)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```



Axis Labels We should be always keen to make the diagrams as readable as possible. So we will add some axis labels.

```
plt.xlabel('x-label')
plt.ylabel('y-label')
```

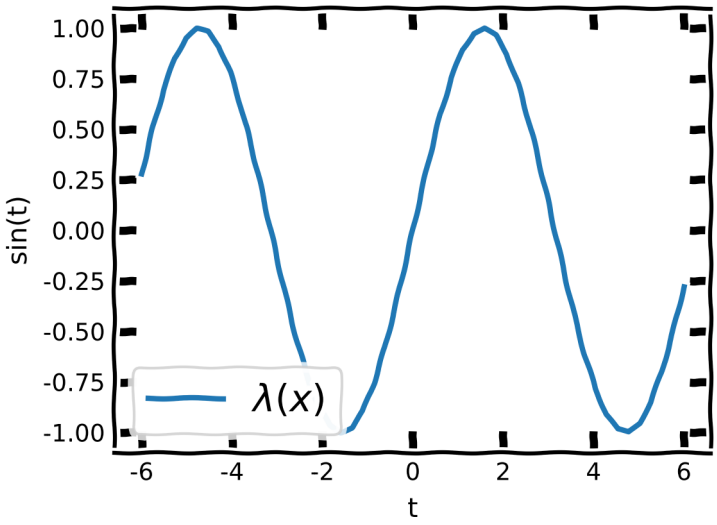
[90]:
```
plt.plot(x,np.sin(x)) # using x-axis
plt.xlabel('t') # set the x-axis label
plt.ylabel('sin(t)') # set the y-axis label
plt.show()
```



Legends

```
plt.plot(..., label=r'$\sin(x)$')
plt.legend(loc='lower left')
```

[138]:
```
with plt.xkcd():
    plt.rcParams['font.family'] = 'DejaVu Sans'
    plt.plot(x,np.sin(x),label=r"$\lambda(x)$") #define an additional label
    plt.xlabel('t')
    plt.ylabel('sin(t)')

    plt.legend(loc='lower left') #add the legend
    plt.show();
```



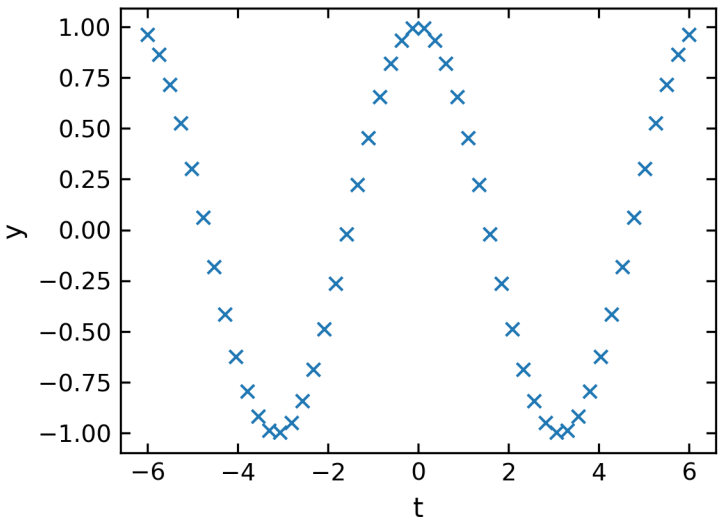
1.1.2 Scatter plot

If you prefer to use symbols for plotting just use the

`plt.scatter(x,y)`

command of pylab. Note that the scatter command requires a x and y values and you can set the marker symbol (see an overview of the marker symbols).

[139]:
```
plt.scatter(x,np.cos(x),marker='x')
plt.xlabel('t') # set the x-axis label
plt.ylabel('y') # set the y-axis label
plt.show()
```



1.1.3 Histograms

A very useful plotting command is also the *hist* command. It generates a histogram of the data provided. If only the data is given, bins are calculated automatically. If you supply an array of intervalls with `hist(data,bins=b)`, where `b` is and array, the hist command calculates the histogram for the supplied bins. `density=True` normalizes the area below the histogram to 1. The hist command not only returns the graph, but also the occurrences and bins.

Physics Interlude Probability density for finding an oscillating particle

We want to use this occasion to combine histograms with some physics problem. Lets have a look at the simple harmonic oscillators in one dimension, which as you remember follows the following equation of motion.

$$\ddot{x}(t) = -\omega^2 x(t) \tag{1}$$

The solution of that equation of motion for an initial elongation Δx at $t = 0$ is given by

$$x(t) = \Delta x \cos(\omega t) \tag{2}$$

If you now need to calculate the probability to find the spring at a certain elongation you need to calculate the time the oscillator spends at different positions. The time dt spend in the interval $[x(t),x(t) + dx]$ depends on the speed, i.e.

$$v(t) = \frac{dx}{dt} = -\omega \Delta x \sin(\omega t) \tag{3}$$

The probability to find the oscillator at a certain intervall then is the fraction of time residing in this intervall normalized by the half the oscillation period $T/2$.

$$\frac{dt}{T/2} = \frac{1}{T/2}\frac{dx}{v(t)} = \frac{1}{T/2}\frac{-dx}{\omega\Delta x \sin(\omega t)} \tag{4}$$

As the frequency of the oscillator is $\omega = 2\pi/T$ we can replace T by $T = 2\pi/\omega$ which yields

$$p(x)dx = \frac{1}{\pi\Delta x}\frac{dx}{\sqrt{1-\left(\frac{x(t)}{\Delta x}\right)^2}} \tag{5}$$

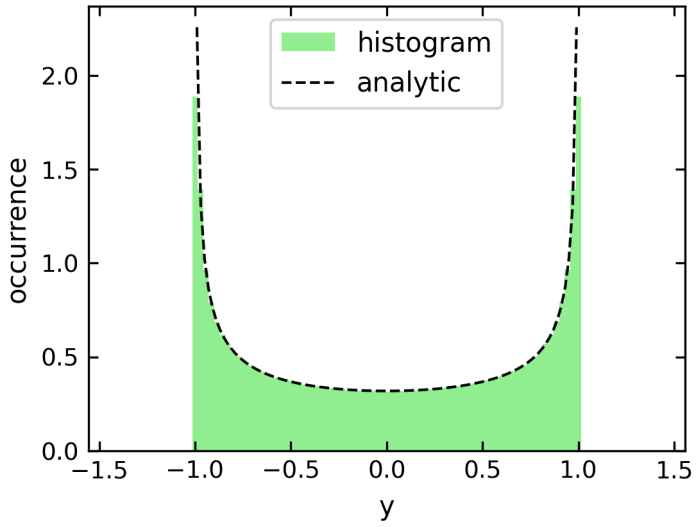
This is the probability density of finding an oscillating spring at a certain elongation $x(t)$. If you look at the example more closely, it tells you, that you find and elongation more likely when the speed of the mass is low. This is even a more general issue in non-equilibrium physics. If cells or cars are moving with variable speed, they are more likely to be found at places where they are slow.

This can be also addressed with the histogram function. If we use the solution of the equation of motion and evaluate the position at equidistant times, the the historgram over the corresponding positions will tell us the probability of finding a certain position value if normalized properly.

Solution:

[93]:
```
y=np.cos(t)
```

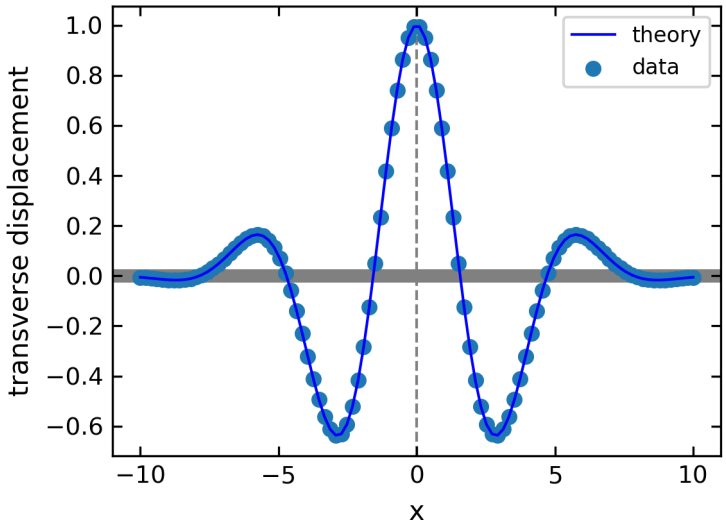
[94]:
```
t=np.linspace(0,np.pi,10000)
y=np.cos(t)
ymin=np.mean(y)-2*np.std(y)
ymax=np.mean(y)+2*np.std(y)
b=np.linspace(ymin,ymax,100)

# plot the histogram
n, bins, _ =plt.hist(np.
  ↪cos(t),bins=b,density=True,color='lightgreen',label='histogram')

#plot the analytical solution
x=np.linspace(-0.99,0.99,100)
plt.plot(x,1/(np.pi*np.sqrt(1-x**2)),'k--',label='analytic')

plt.xlabel('y') # set the x-axis label
plt.ylabel('occurrence') # set the y-axis label
plt.legend()
plt.show()
```



1.1.4 Combined plots

You can combine multiple data with the same axes by stacking multiple plots.

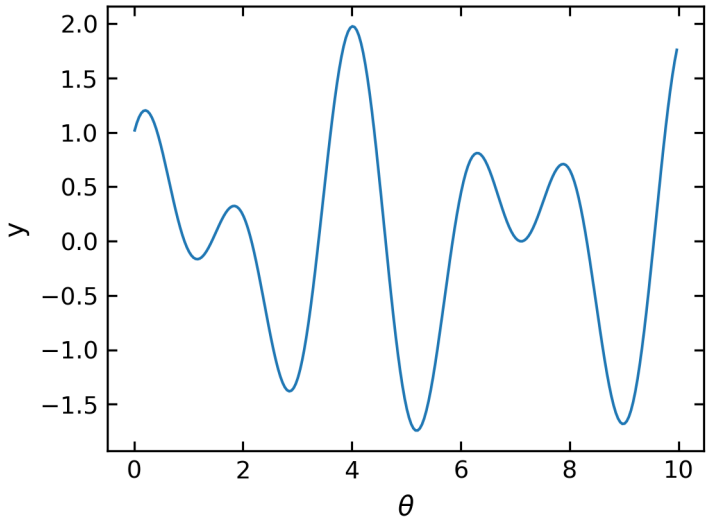
[156]:
```
# create x and y arrays for theory
x = np.linspace(-10., 10., 100)
y = np.cos(x) * np.exp(-(x/4.47)**2)

# create plot
plt.figure(1, figsize = (4,3),dpi=150)

plt.plot(x, y, 'b-', label='theory')
plt.scatter(x, y,  label="data")

plt.axhline(color = 'gray', linewidth=5,zorder=-1)
plt.axvline(ls='--',color = 'gray', zorder=-1)


plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()
```



1.2 Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class. Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. For scientific papers, I recommend using PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command). In some cases, PGF can also be good alternative.

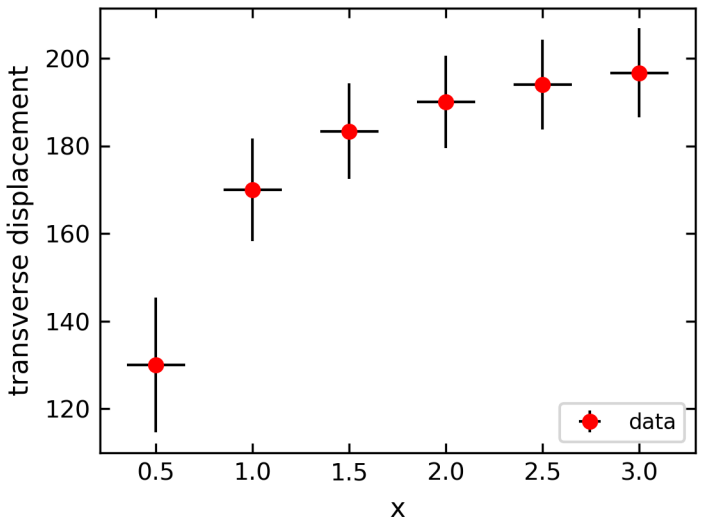
[157]:
```
theta = np.arange(0.01, 10., 0.04)
ytan = np.sin(2*theta)+np.cos(3.1*theta)
plt.figure(1, figsize = (4,3),dpi=150)
plt.plot(theta, ytan)
plt.xlabel(r'$\theta$')
plt.ylabel('y')
plt.savefig('filename.pdf',transparent=True)
plt.show()
```



1.3 Plots with error bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty that exists in the measurement of each data point. The MatPlotLib function errorbar plots data with error bars attached. It can be used in a way that either replaces or augments the plot function. Both vertical and horizontal error bars can be displayed. The figure below illustrates the use of error bars.

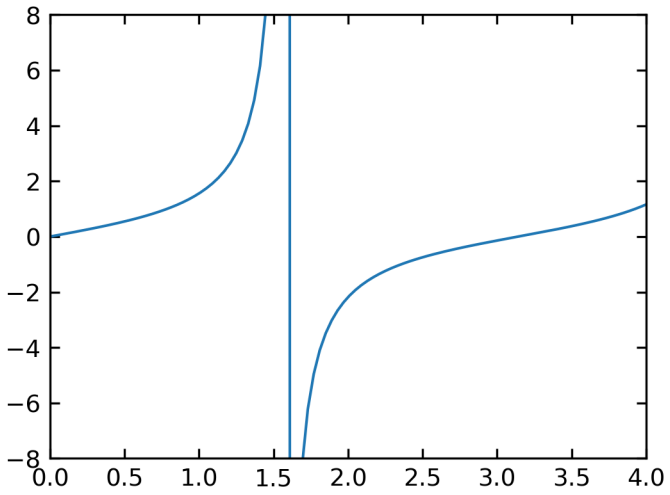
[158]:
```
plt.figure(1, figsize = (4,3),dpi=150)
xdata=np.arange(0.5,3.5,0.5)
ydata=210-40/xdata

yerror=2e3/ydata

plt.errorbar(xdata, ydata, fmt="ro", label="data",
             xerr=0.15, yerr=yerror, ecolor="black")

plt.xlabel("x")
plt.ylabel("transverse displacement")
plt.legend(loc="lower right")

plt.show()
```



1.3.1 Setting plotting limits and excluding data

If you want to zoom in to s specific region of a plot you can set the limits of the individual axes.

[159]:
```
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)
plt.figure(figsize=(4,3),dpi=150)
plt.plot(theta, ytan)
plt.xlim(0, 4)
plt.ylim(-8, 8) # restricts range of y axis from -8 to +8 plt.
↪axhline(color="gray", zorder=-1)
plt.show()
```

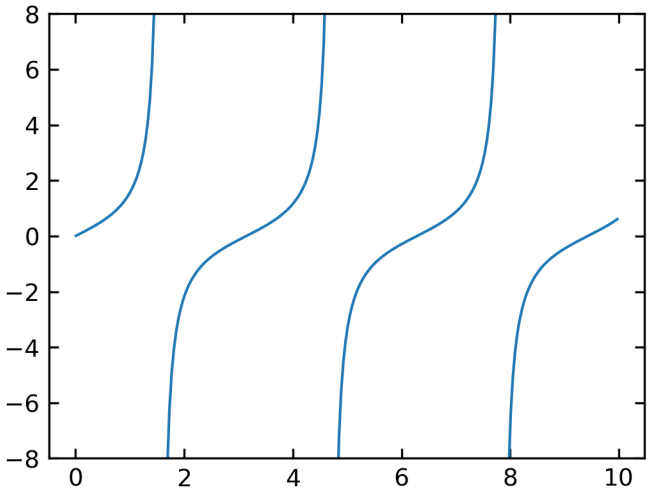


Masked arrays Sometimes you encounter situations, when you wish to mask some of the data of your plot, because they are not showing real data as the vertical lines in the plot above. For this purpose, you can mask the data arrays in various ways to not show up. The example below uses the

`np.ma.masked_where()`

function of NumPy, which takes a condition as the first argument and what should be returned if that condition is fulfilled.

[160]:
```
theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

#exclude specific data from plotting
ytanM = np.ma.masked_where(np.abs(ytan)>20., ytan)

plt.figure(figsize=(4,3),dpi=150)
plt.plot(theta, ytanM)
plt.ylim(-8, 8)
plt.show()
```



If you look at the resulting array, you will find, that the entries have not been removed but replaced by `--`, so the values are not existent and thefore not plotted.

1.4 Logarithmic plots

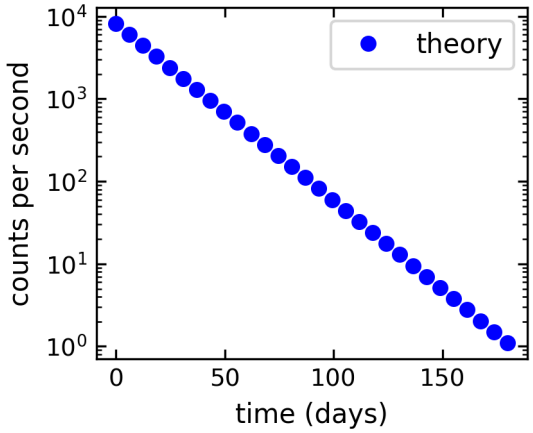
Data sets can span many orders of magnitude from fractional quantities much smaller than unity to values much larger than unity. In such cases it is often useful to plot the data on logarithmic axes.

1.4.1 Semi-log plots

For data sets that vary exponentially in the independent variable, it is often useful to use one or more logarithmic axes. Radioactive decay of unstable nuclei, for example, exhibits an exponential decrease in the number of particles emitted from the nuclei as a function of time.

MatPlotLib provides two functions for making semi-logarithmic plots, semilogx and semilogy, for creating plots with logarithmic x and y axes, with linear y and x axes, respectively. We illustrate their use in the program below, which made the above plots.

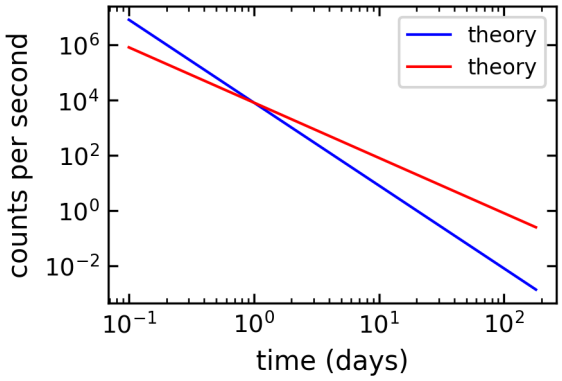
[100]:
```
# create theoretical curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0, 180, 30)
N = N0 * np.exp(-t/tau)

# create plot
plt.figure(1, figsize = (3,2.5),dpi=150)

plt.semilogy(t, N, 'bo', label="theory")
plt.xlabel('time (days)')
plt.ylabel('counts per second')
plt.legend(loc='upper right')

plt.tight_layout()
plt.show()
```



1.4.2 Log-log plots

MatPlotLib can also make log-log or double-logarithmic plots using the function loglog. It is useful when both the x and y data span many orders of magnitude. Data that are described by a power law $y = Ax^b$, where A and b are constants, appear as straight lines when plotted on a log-log plot. Again, the loglog function works just like the plot function but with logarithmic axes.

[161]:
```
# create theoretical fitting curve
tau = 20.2 # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
N0 = 8200. # Initial count rate (per second)
t = np.linspace(0.1, 180, 128)
N = N0 * t**(-3)
N1 = N0 * t**(-2)

# create plot
plt.figure(1, figsize = (3,2),dpi=150)
plt.loglog(t, N, 'b-', label="theory")
plt.loglog(t, N1, 'r-', label="theory")
#plt.plot(time, counts, 'ro', label="data")
plt.xlabel('time (days)')
plt.ylabel('counts per second')
plt.legend(loc='upper right')
plt.show()
```



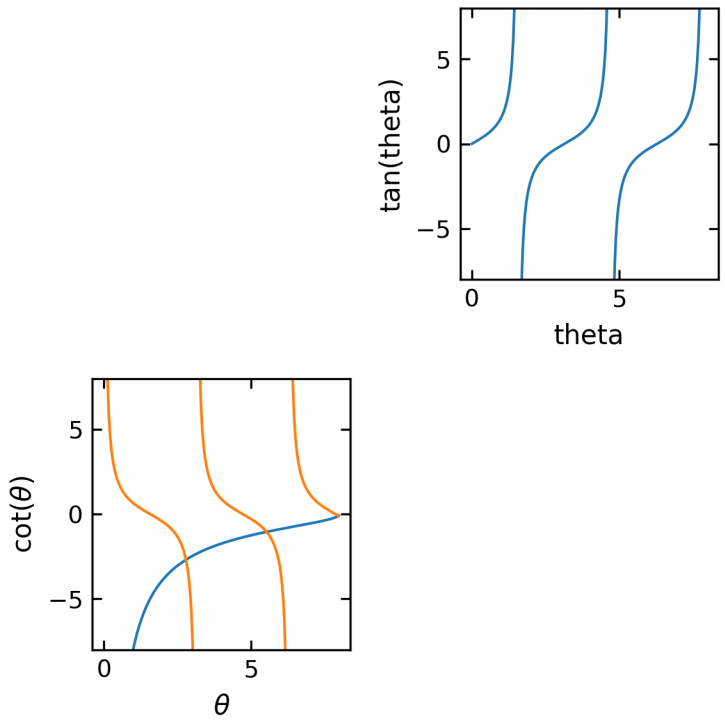
1.5 Arranging multiple plots

Often you want to create two or more graphs and place them next to one another, generally because they are related to each other in some way.

[162]:
```
theta = np.arange(0.01, 8., 0.04)
y = np.sqrt((8./theta)**2-1.)
ytan = np.tan(theta)
ytan = np.ma.masked_where(np.abs(ytan)>20., ytan)
ycot = 1./np.tan(theta)

ycot = np.ma.masked_where(np.abs(ycot)>20., ycot)
```

[163]:
```
plt.figure(1,figsize=(4,4),dpi=150)
plt.subplot(2, 2, 2)
```

```
plt.plot(theta, ytan)
plt.ylim(-8, 8)
plt.xlabel("theta")
plt.ylabel("tan(theta)")

plt.subplot(2, 2, 3)
plt.plot(theta, -y)
plt.plot(theta, ycot)
plt.ylim(-8, 8)
plt.xlabel(r'$\theta$')
plt.ylabel(r'cot($\theta$)')

plt.tight_layout()
plt.show()
```



1.6 Contour and Density Plots

A contour plots are useful tools to study two dimensional data, meaning $Z(X, Y)$. A contour plots isolines of the function Z.

1.6.1 Simple contour plot

Note: Physics Interlude

Interference of spherical waves

We want to use contour plots to explore the interference of two spherical waves. A spherical wave can be given by a complex spatial amplitude

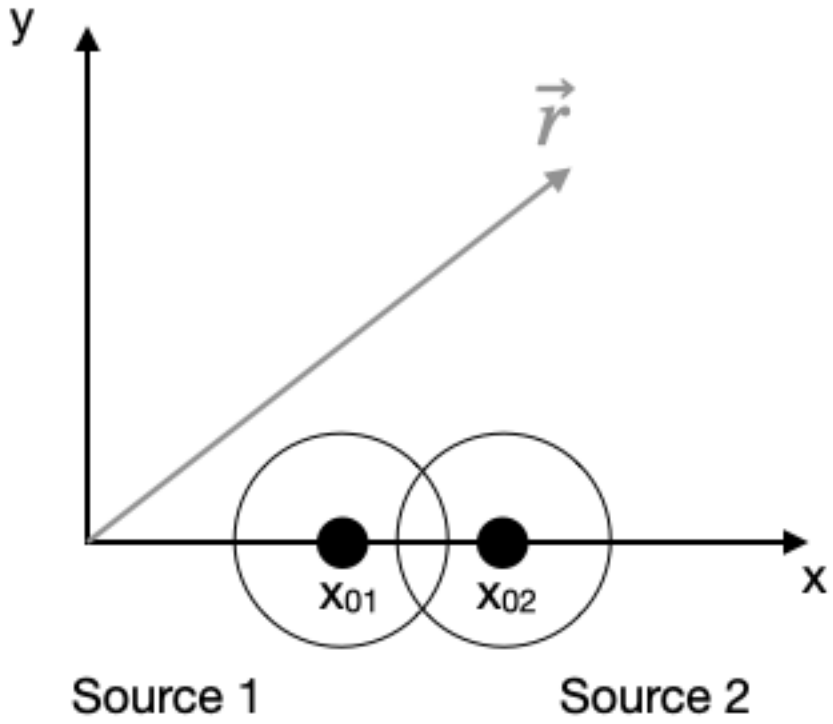
$$U(r) = U_0 \frac{e^{-i\,kr}}{r} \tag{6}$$

where U_0 is the amplitude, k the magnitude of the k-vector, i.e. $k = 2\pi/\lambda$ and $r = \sqrt{x^2 + y^2}$ the distance, here in 2 dimensions. Note that the total wavefunction also contains a temporal part such that

$$U(r,t) = U_0 \frac{e^{-i\,kr}}{r} e^{i\omega t} \tag{7}$$

We will however ignore the temporal factor, as we are interested in the spatial interference pattern with a time-averaged intensity.

To show interference, we just use two of those monochromatic waves located at $\vec{r}_1$ and $\vec{r}_2$.



To obtain the total amplitude we have to sum up the wavefunctions of the two sources.

$$U(\vec{r}) = U_{01} \frac{e^{-i\,k|\vec{r}-\vec{r}_1|}}{|\vec{r}-\vec{r}_1|} + U_{02} \frac{e^{-i\,k|\vec{r}-\vec{r}_2|}}{|\vec{r}-\vec{r}_2|} \tag{8}$$

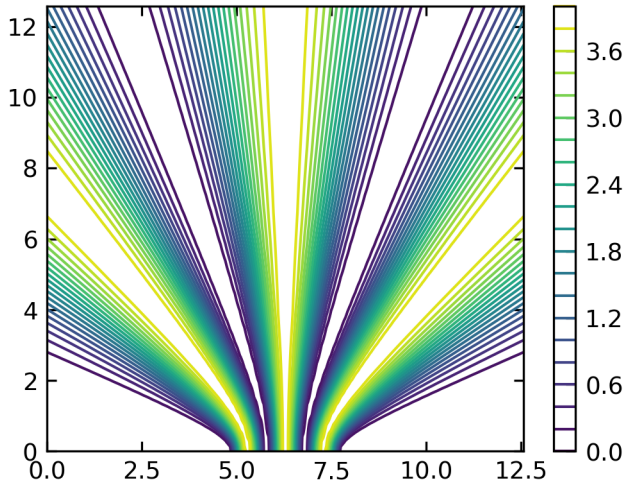
The intensity of the wave at a position is then related to the magnitude square of the wavefunction

$$I(r) \propto |U(r)|^2 \tag{9}$$

To keep it simple we will skip the $1/r$ amplitude decay.

[168]:
```
lmda = 2 # defines the wavelength
x01=1.5*np.pi # location of the first source, y01=0
x02=2.5*np.pi # location of the second source, y02=0
x = np.linspace(0, 4*np.pi, 100)
y = np.linspace(0, 4*np.pi, 100)


X, Y = np.meshgrid(x, y)
I= np.abs( np.exp(-1j*np.sqrt((X-x01)**2+Y**2)*2*np.pi/lmda)
          +np.exp(-1j*np.sqrt((X-x02)**2+Y**2)*2*np.pi/lmda))**2
```

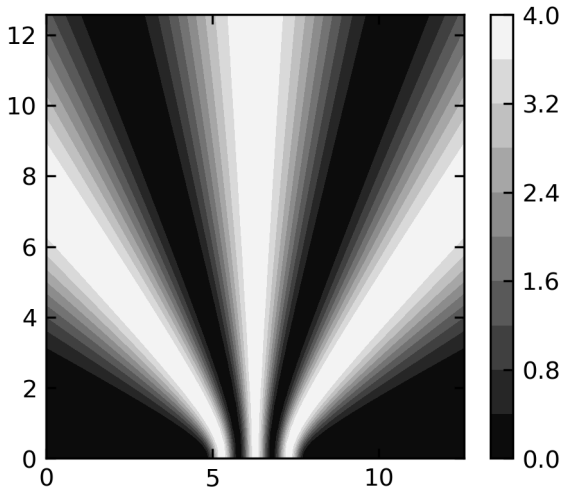
[169]:
```
plt.figure(1,figsize=(4,3),dpi=150)

#contour plot
plt.contour(X, Y, I, 20)
plt.colorbar()
plt.show()
```



1.6.2 Color contour plot

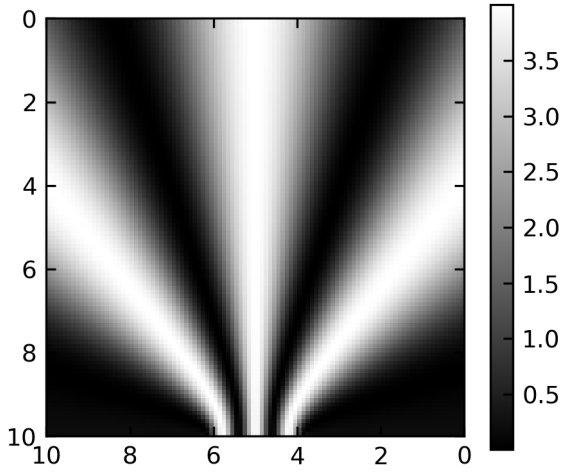
[170]:
```
plt.figure(1,figsize=(3.5,3),dpi=150)
plt.contourf(X, Y, I, 10, cmap='gray')
plt.colorbar()
```

```
plt.show()
```



1.6.3 Image plot

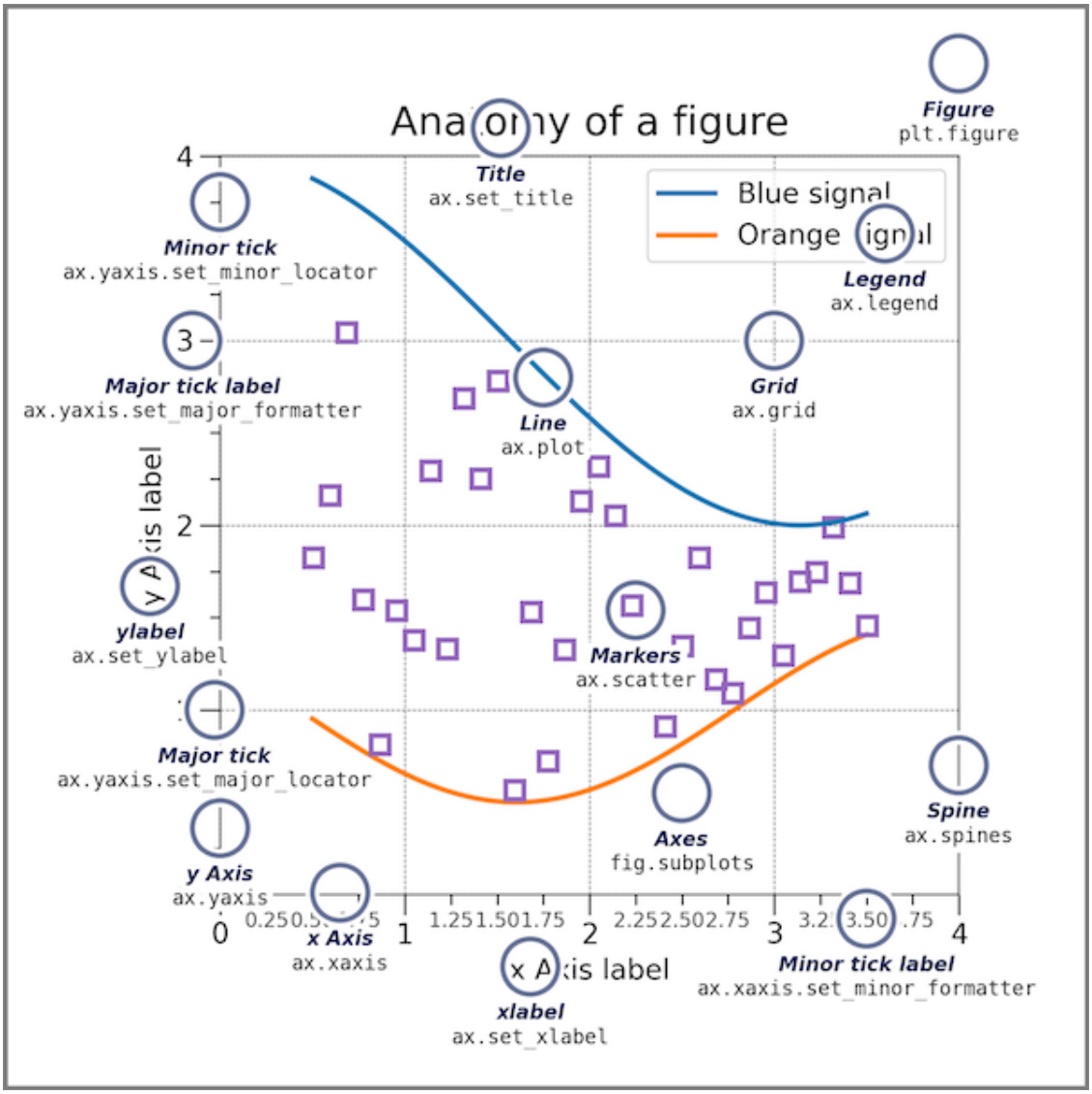
[171]:
```
plt.figure(1,figsize=(3.5,3),dpi=150)
plt.imshow( I,extent=[10,0,0,10],cmap='gray');
plt.ylim(10,0)
plt.colorbar()
plt.show()
```



1.7 Additional Plotting - Explicit Version

While we have so far largely relied on the default setting and the automatic arrangement of plots, there is also a way to precisely design your plot. Python provides the tools of object oriented programming and thus modules provide classes which can be instanced into objects. This explicit interfaces allows you to control all details without the automatisms of `pyplot`.

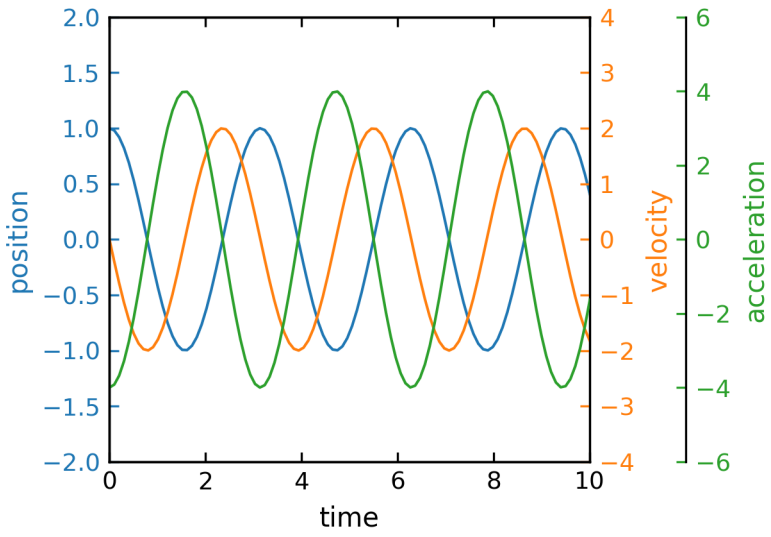
The figure below, which is taken from the matplotlib documentation website shows the sets of commands and the objects in the figure, the commands refer to. It is a nice reference, when creating a figure.



1.7.1 Plots with Multiple Spines

Sometimes it is very useful to plot different quantities in the same plot with the same x-axis but with different y-axes. Here is some example, where each line plot has its own y-axis.

[172]:
```
fig, ax = plt.subplots()

fig.subplots_adjust(right=0.75)
fig.subplots_adjust(right=0.75)

twin1 = ax.twinx()
twin2 = ax.twinx()

# Offset the right spine of twin2.  The ticks and label have already been
# placed on the right by twinx above.
twin2.spines.right.set_position(("axes", 1.2))

time=np.linspace(0,10,100)
omega=2
p1, = ax.plot(time,np.cos(omega*time), "C0", label="position")
p2, = twin1.plot(time,-omega*np.sin(omega*time), "C1", label="velocity")
p3, = twin2.plot(time, -omega**2*np.cos(omega*time), "C2", label="acceleration")

ax.set(xlim=(0, 10), ylim=(-2, 2), xlabel="time", ylabel="position")
twin1.set(ylim=(-4, 4), ylabel="velocity")
twin2.set(ylim=(-6, 6), ylabel="acceleration")

ax.yaxis.label.set_color(p1.get_color())
twin1.yaxis.label.set_color(p2.get_color())
twin2.yaxis.label.set_color(p3.get_color())

ax.tick_params(axis='y', colors=p1.get_color())
twin1.tick_params(axis='y', colors=p2.get_color())
twin2.tick_params(axis='y', colors=p3.get_color())

#ax.legend(handles=[p1, p2, p3])

plt.show()
```

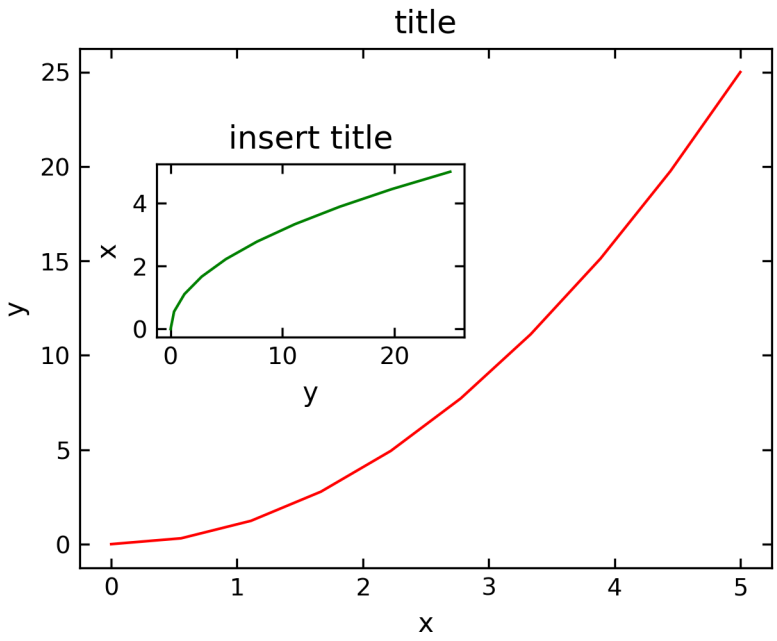


1.7.2 Insets

Insets are plots within plots using their own axes. We therefore need to create two axes systems, if we want to have a main plot and and inset.

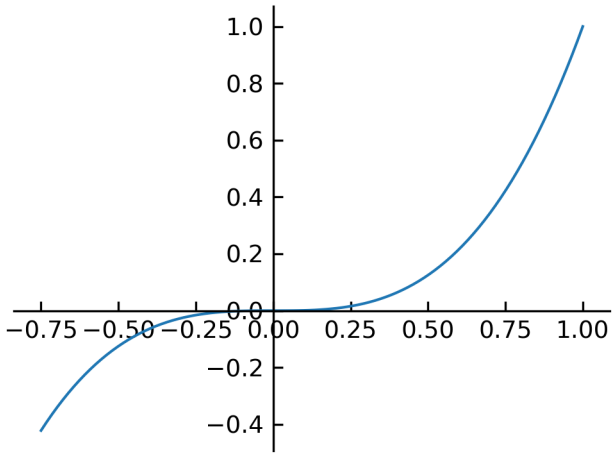
[173]:
```
x = np.linspace(0, 5, 10)
y = x ** 2
```

[174]:
```
fig = plt.figure(figsize=(4,3),dpi=150)

axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('insert title')
plt.show()
```



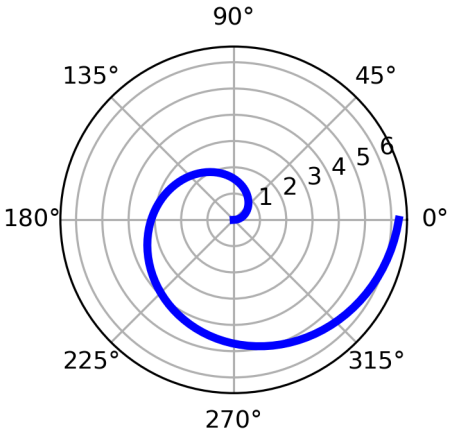
1.7.3 Spine axis

[175]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```



1.7.4 Polar plot

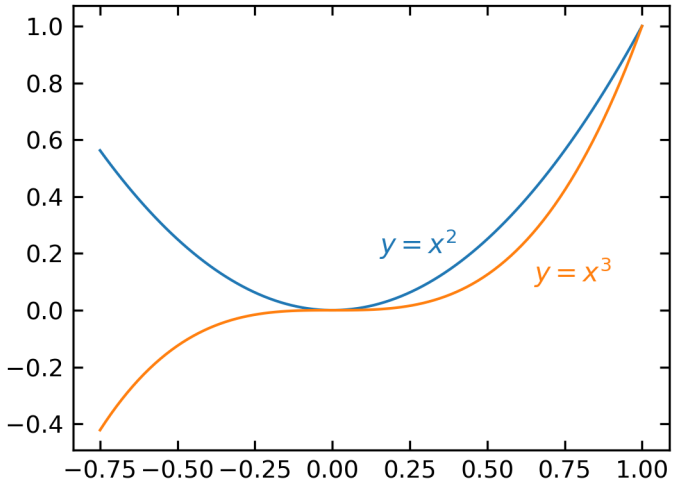
[176]:
```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = np.linspace(0, 2 * np.pi, 100)
ax.plot(t, t, color='blue', lw=3);
```



1.7.5 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

[177]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", color="C0")
ax.text(0.65, 0.1, r"$y=x^3$", color="C1");
```



1.7.6 3D Plotting

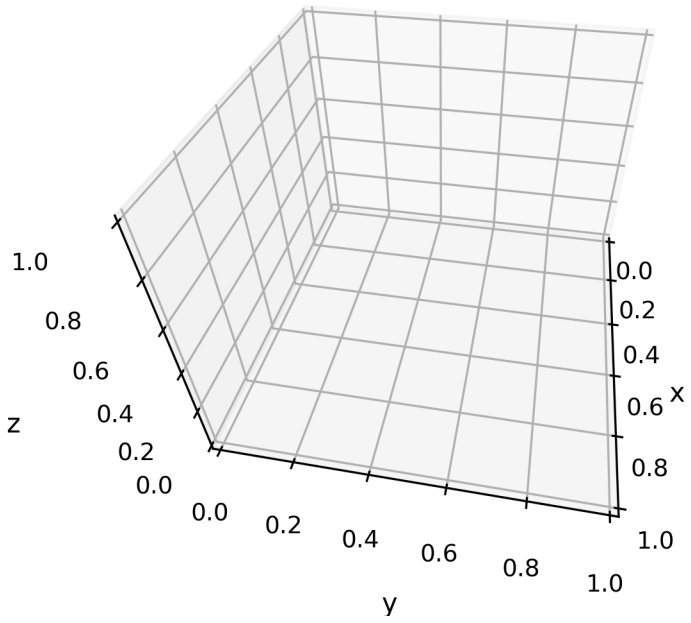
Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the mplot3d toolkit, included with the main Matplotlib installation:

[178]:
```
from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword projection='3d' to any of the normal axes creation routines:

Projection Scence

[179]:
```
fig=plt.figure(figsize=(4,4))
ax = plt.axes(projection='3d')
#ax.set_proj_type('ortho')
ax.set_proj_type('persp', focal_length=0.2)
ax.view_init(elev=40., azim=10)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```

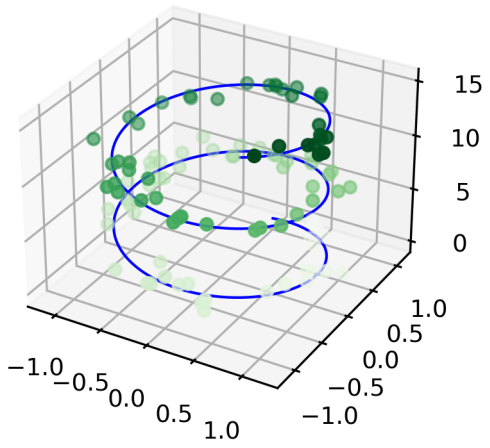


With this three-dimensional axes enabled, we can now plot a variety of three-dimensional plot types. Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook; recall that to use interactive figures, you can use %matplotlib notebook rather than %matplotlib inline when running this code.

Line Plotting in 3D from sets of (x, y, z) triples. In analogy with the more common two-dimensional plots discussed earlier, these can be created using the `ax.plot3D` and `ax.scatter3D` functions. The call signature for these is nearly identical to that of their two-dimensional counterparts, so you can refer to Simple Line Plots and Simple Scatter Plots for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some points drawn randomly near the line:

[180]:
```
plt.figure(figsize=(4,3))

#create the axes
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'blue')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
```

```
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens')
plt.show()
```



Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page. While the three-dimensional effect is sometimes difficult to see within a static image, an interactive view can lead to some nice intuition about the layout of the points. Use the `scatter3D` or the `plot3D` method to plot a random walk in 3-dimensions in your exercise.

Surface Plotting A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized:

[181]:
```
x = np.linspace(-6, 6, 50)
y = np.linspace(-6, 6, 60)

X, Y = np.meshgrid(x, y)
Z=np.sin(X)*np.sin(Y)
```

[182]:
```
np.shape(Z)
```

[182]: (60, 50)

[183]:
```
plt.figure(figsize=(4,3))
ax = plt.axes(projection='3d')
ax.view_init(30, 50)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface')
```

```
plt.show()
```