Physics 116C Scientific Python Exercises

May 3, 2019

Chapter 1

Introduction to Plotting

1.1 Introduction

This exercise will introduce calculations and plotting techniques using numpy arrays within Scientific Python.

1.2 Plotting discrete data and continuous functions

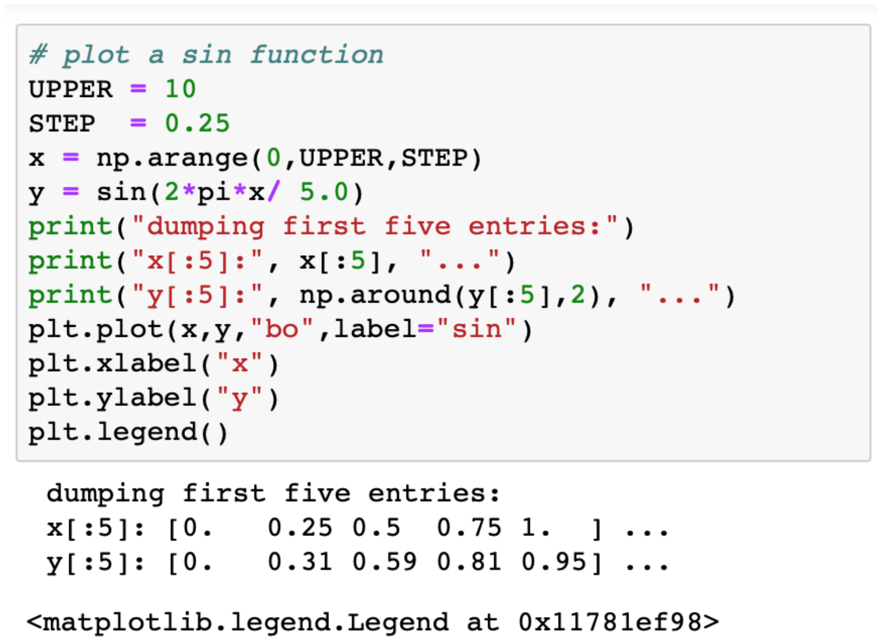
Consider the Jupyter notebook example in Fig. 1.1 which plots a sine function sampled at discrete values. Note the following key features, which you will use repeatedly today and in future labs:

- Use of global variables `UPPER` and `STEP` at the top of the snippet, allowing for easy adjustment of parameters that affect the plot.
- Use of `np.arange` to define an array of x values.
- Creation of the array y, defined by $y = \sin(2\pi x/5)$ for each value of x. One of the great joys of using numpy is the ability to avoid getting bogged down with explicit for loops.
- Use of slicing techniques `x[:5]` to show only the first five entries for debugging.
- Plotting the arrays of x and y values with `plt.plot` using the `"bo"` option for blue circles.
- Defining appropriate axis labels with `plt.xlabel` and `plt.xlabel`.
- Creation of a legend using the `label` option to `plt.plot` and the plot.legend() command.

Notice that even in this simple example, I've added some intermediate feedback from my code in the form of the screen dumps of the first few values of x and y. It's a common pitfall to try and rush ahead to the final product when programming. But it is much faster and reliable to break your task into small steps, and establish feedback at each small step. To plot a continuous function with Scientific Python, you will still use discrete data but:

- Use much finer binning of the x-axis variable to draw a smooth curve.
- Use the line option `"-"` or dashed line `"--"` instead of points with `"o"`.

Plot 1: Plot the quadratic function $y = x^2$ with the following requirements:



```
# plot a sin function
UPPER = 10
STEP  = 0.25
x = np.arange(0,UPPER,STEP)
y = sin(2*pi*x/ 5.0)
print("dumping first five entries:")
print("x[:5]:", x[:5], "...")
print("y[:5]:", np.around(y[:5],2), "...")
plt.plot(x,y,"bo",label="sin")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

```
 dumping first five entries:
 x[:5]: [0.   0.25 0.5  0.75 1.  ] ...
 y[:5]: [0.   0.31 0.59 0.81 0.95] ...

<matplotlib.legend.Legend at 0x11781ef98>
```

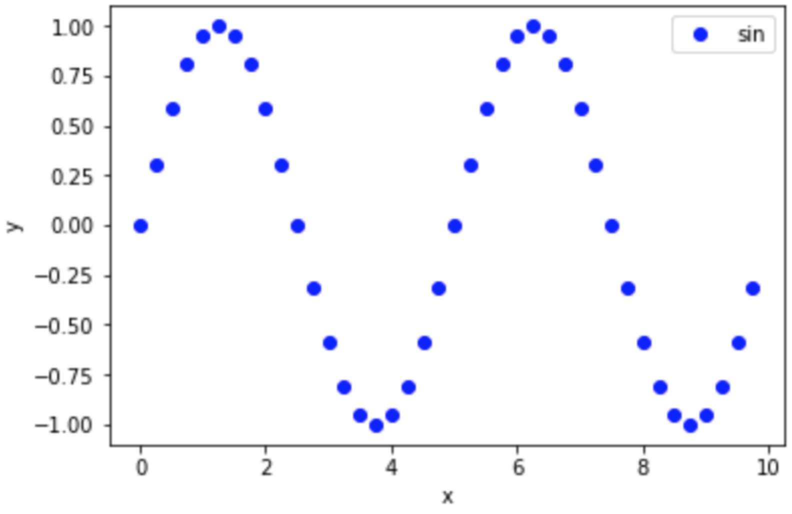
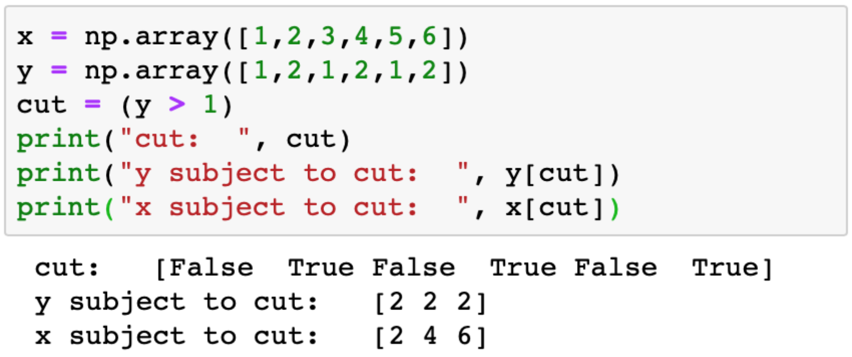


Figure 1.1: Circuit for verifying Ohm's law as a (a) circuit diagram, and (b) implemented using your lab equipment.

- Plot in the range $x = [0, 20)$.
- Plot discrete samples with a step size of 2 using blue circles.
- On the same axis, plot the corresponding smooth function using a red solid line.
- Add appropriate axis labels.
- Add a legend for the “discrete” and the “smooth” function.

1.3 Multivariate analysis using boolean masks



```
x = np.array([1,2,3,4,5,6])
y = np.array([1,2,1,2,1,2])
cut = (y > 1)
print("cut:  ", cut)
print("y subject to cut:  ", y[cut])
print("x subject to cut:  ", x[cut])
```

```
cut:   [False  True False  True False  True]
y subject to cut:   [2 2 2]
x subject to cut:   [2 4 6]
```

Figure 1.2: Using boolean masks to cut on variable y.

A powerful technique in Scientific Python for performing analysis involving multiple variables uses boolean masks as shown in Fig. 1.2. In the example:

- Two numpy arrays x and y `of the same length` are defined to contain the collected data.
- The cut defined by $y > 1$ is a boolean array of the same length as x and y which is true at indices where the condition is met and false where it is not.
- The subset of the entire y array defined by `y[cut]` consists only of those entries of y for which the condition $y > 1$ is met.
- The subset of the entire x array defined by `x[cut]` consists only of those entries of x for which the condition $y > 1$ is met for the corresponding y value.

The last item shows the real power of this technique, one can look at one variable subject to constraints on another variable.

Next consider the sample data in Table 1.1 which comes from experimental measurements of a voltage level v at discrete times t. The measurement is subject to a high-frequency noise monitoring by the variable n. The noise is only present for $n > 6.0$. A straightforward way to load this data into scientific python is by defining numpy arrays for each variable as follows:

Table 1.1: Sample data for a voltage measurement subject to high frequency noise.

t (s)	v (V)	n
0.4	0.25	2.8
1.1	2.37	7.3
1.4	1.69	9.7
1.9	0.93	1.3
2.5	-1.0	6.2
3.0	0.95	4.8
3.4	1.22	6.9
4.1	0.54	4.0
4.4	0.37	1.9
4.8	0.13	4.0
5.5	-2.04	9.5
6.2	-2.06	8.7
6.5	-0.81	2.3
7.0	-0.95	5.3
7.5	0.98	9.7
7.9	0.27	8.3
8.5	-0.81	0.1
9.0	-0.59	5.1
9.4	-0.37	4.4
9.9	0.56	9.9

```
t = np.array([0.4, 1.1, 1.4, 1.9, 2.5, 3.0, 3.4, 4.1, 4.4, 4.8,
                    5.5, 6.2, 6.5, 7.0, 7.5, 7.9, 8.5, 9.0, 9.4, 9.9])
v = np.array([ 0.25, 2.37, 1.69, 0.93, -1.0, 0.95, 1.22,
                     0.54, 0.37, 0.13, -2.04, -2.06, -0.81, -0.95,
                     0.98, 0.27, -0.81, -0.59, -0.37, 0.56])
n = np.array([2.8, 7.3, 9.7, 1.3, 6.2, 4.8, 6.9, 4.0, 1.9, 4.0,
                     9.5, 8.7, 2.3, 5.3, 9.7, 8.3, 0.1, 5.1, 4.4, 9.9])
```

Plot 2 Prepare a plot of the sample data subject to the following:

- Plot the voltage as a function of time as discrete data using red points.
- Define the boolean array `keep` based on the noise reducing condition $n <= 6.0$.
- Plot the voltage as a function of time, subject to the noise reducing condition using blue points.
- Plot the function $\sin(2\pi x/10)$ as a smooth function.
- Add appropriate axis labels.
- Add a legend for "raw" data with no cut, "clean" data with noise removed, and your continuous "sin" function.

Your plot will reveal a clear sine function in the discrete data (after noise removal) consistent with the continuous function.

Chapter 2

Histograms

2.1 Introduction

In this exercise, you will learn how to produce and display histograms, and compare histogram data with a distribution function.

2.2 Histogram of Simulated Data

First we will generate fake data, called Monte Carlo data, in order to have something to plot. We'll use the `random.norm` function to produce 200 events randomly drawn from a Gaussian distribution with mean of 5, and sigma of 1.5:

```
NEVT = 200
dat = np.random.normal(loc=10.0,scale=3.0,size=NEVT)
print(dat[:5])
```

```
[ 9.34696311 12.46436606 14.44383343 13.99559213  8.914403
88]
```

Notice the scipy names for the Gaussian parameters are `loc` for the mean, and `scale` for σ. The first five events of the 200 produced are printed to the screen as a quick check of our work.

Next we'll produce a histogram for this simulated data, as shown in Fig. 2.1 using the scipy `histogram` function:

```
counts,edges = np.histogram(dat,bins=10,range=(0,20))
```

where we have specified 10 bins, uniformly covering the range from 0 to 20. The function returns to arrays, which we save as `counts` and `edges`. The `counts` array contains the bin contents, the count of the number of values in each bin:

```
counts:       [ 1  6 13 29 50 56 24 16  4  1]
```

The edges array contains the edges of the bins:

```
bin edges:    [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
```

```
counts,edges = np.histogram(dat,bins=10,range=(0,20))
cbins = (edges[1:] + edges[:-1])/2.0
print("counts:      ", counts)
print("total:       ", np.sum(counts))
print("bin edges:   ", edges)
print("cbins:       ", cbins)
```

```
counts:      [ 1   6 13 29 50 56 24 16   4   1]
total:       200
bin edges:   [ 0.   2.   4.   6.   8. 10. 12. 14. 16. 18. 20.]
cbins:       [ 1.   3.   5.   7.   9. 11. 13. 15. 17. 19.]
```

Figure 2.1: Example creating a histogram.

You'll notice that 10 consecutive bins have 11 edges. When plotting continuous data, plot the contents at the center of each bin:

```
cbins = (edges[:-1] + edges[1:])/2.0
```

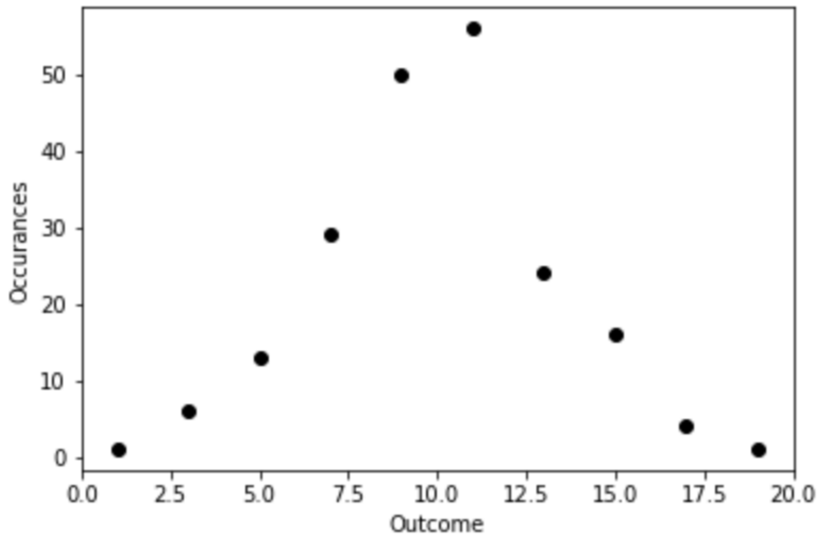
Here the two slices `edges[:-1]` and `edges[1:]` are all but the last and all but the first edges. The average of the two is the center of each bin:

```
cbins:    [ 1.  3.  5.  7.  9. 11. 13. 15. 17. 19.]
```

We can take a quick look at the histogram data with the `plot` function:

```
plt.plot(cbins,counts,"ko")
plt.xlim(0,20)
plt.xlabel("Outcome")
plt.ylabel("Occurances")
```

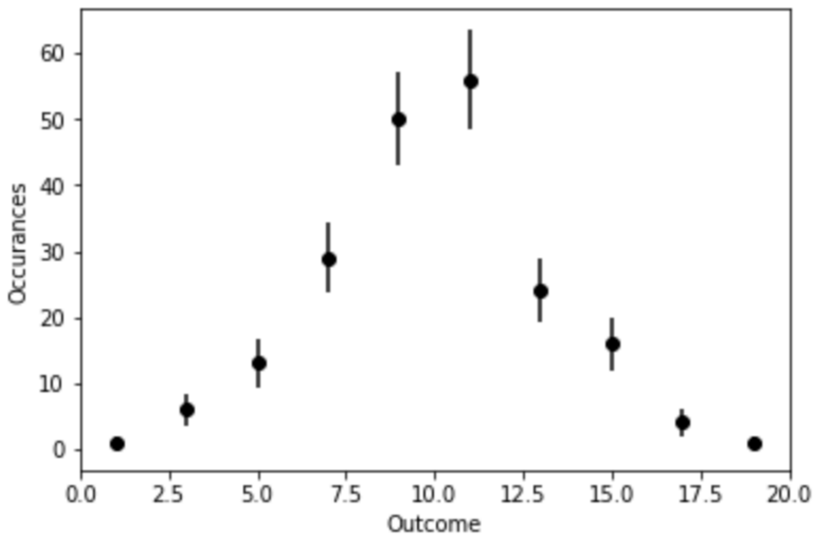
```
Text(0,0.5,'Occurances')
```



The content of each bin is a single number, a count, and is therefore subject to the Poisson distribution. We can estimate the mean of the Poisson distribution by the measured value, so that $\lambda = N$. For the Poisson distribution, the variance $\sigma^2 = \lambda$, and so $\sigma = \sqrt{N}$. It is customary to draw a line of size $\sqrt{N}$ when plotting a histogram value N. This is an example of an error bar, which indicates how well our measurement has determined a particular value.

```
errs = counts**0.5
plt.errorbar(cbins,counts,yerr=errs,fmt="ko")
plt.xlim(0,20)
plt.xlabel("Outcome")
plt.ylabel("Occurances")
```

```
Text(0,0.5,'Occurances')
```



2.3 Comparison to Distribution Function

Our theoretical models often predict a PDF for some observable variable x. As experimentalists, we are often therefore concerned with the question as to whether our collected data for an observable x is consistent with the theoretical PDF. A visual approach to answering this question is to plot the data in a histogram, and to draw the PDF as a curve normalized to the histogram.

To predict the number of events in a bin with edges x_{lower} and x_{upper}, in principle we need to integrate the PDF and normalize to the number of experiments:

$$N_{\mathrm{pred}} = N_{\mathrm{meas}} \int_{x_{\mathrm{lower}}}^{x_{\mathrm{upper}}} p(x)\, dx$$

In practice, we generally choose the bin sizes small enough that the PDF is approximately constant during the entire bin, and in this case, the prediction can be taken as:

$$N_{\mathrm{pred}} = N_{\mathrm{meas}}\, \Delta x\; p(x)$$

where Δx is the width of each bin. This scale factor $N_{\mathrm{meas}}\, \Delta x$ allows us to compare a continuous function to data accumulated within bins.

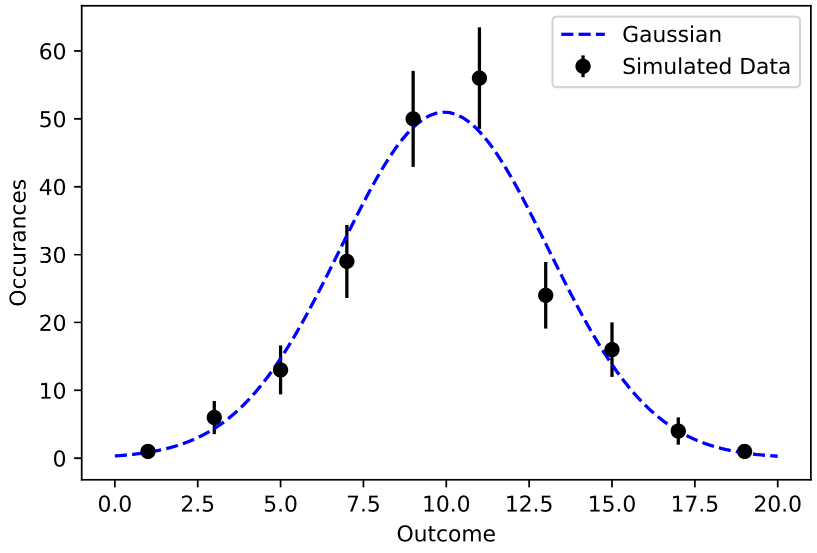


Figure 2.2: Comparison of simulated data to Guassian PDF.

Plot 1: Reproduce the plot in Fig. 2.2 by comparing the simulated data to the Gaussian distribution function available from the stats.norm.pdf function. To use the `stats` library, you will need to import it:

```
from scipy import stats
```

You'll need to appropriately scale the PDF.

Plot 2: Repeat the plot with 10,000 events and 50 bins in the range from $(0, 20)$.

Chapter 3

Curve Fitting

3.1 Introduction

In this exercise, you will learn about curve fitting with Scientific Python function `curve_fit`. Given a function to fit $f(x;p)$, with p representing any number of parameters, and a set of measurements y_i and points x_i, the `curve_fit` function determines the best fit parameters by minimizing:

$$\chi^2 = \sum_i \frac{(f(x_i;p) - y_i)^2}{\sigma_i^2}.$$

If the uncertainties σ_i are not specified, the function assumes $\sigma_i = 1$, and still finds the correct minimum if the actual uncertainties are identical to one another.

3.2 Fitting a Straight Line

An example using Scientific Pythons `curve_fit` function to fit a straight line to data is shown in Fig. 3.2. A block of code defining the function we wish to fit, in this case, a straight line, is defined as a function:

```
def line_func(x, a, b):
    return a*x + b
```

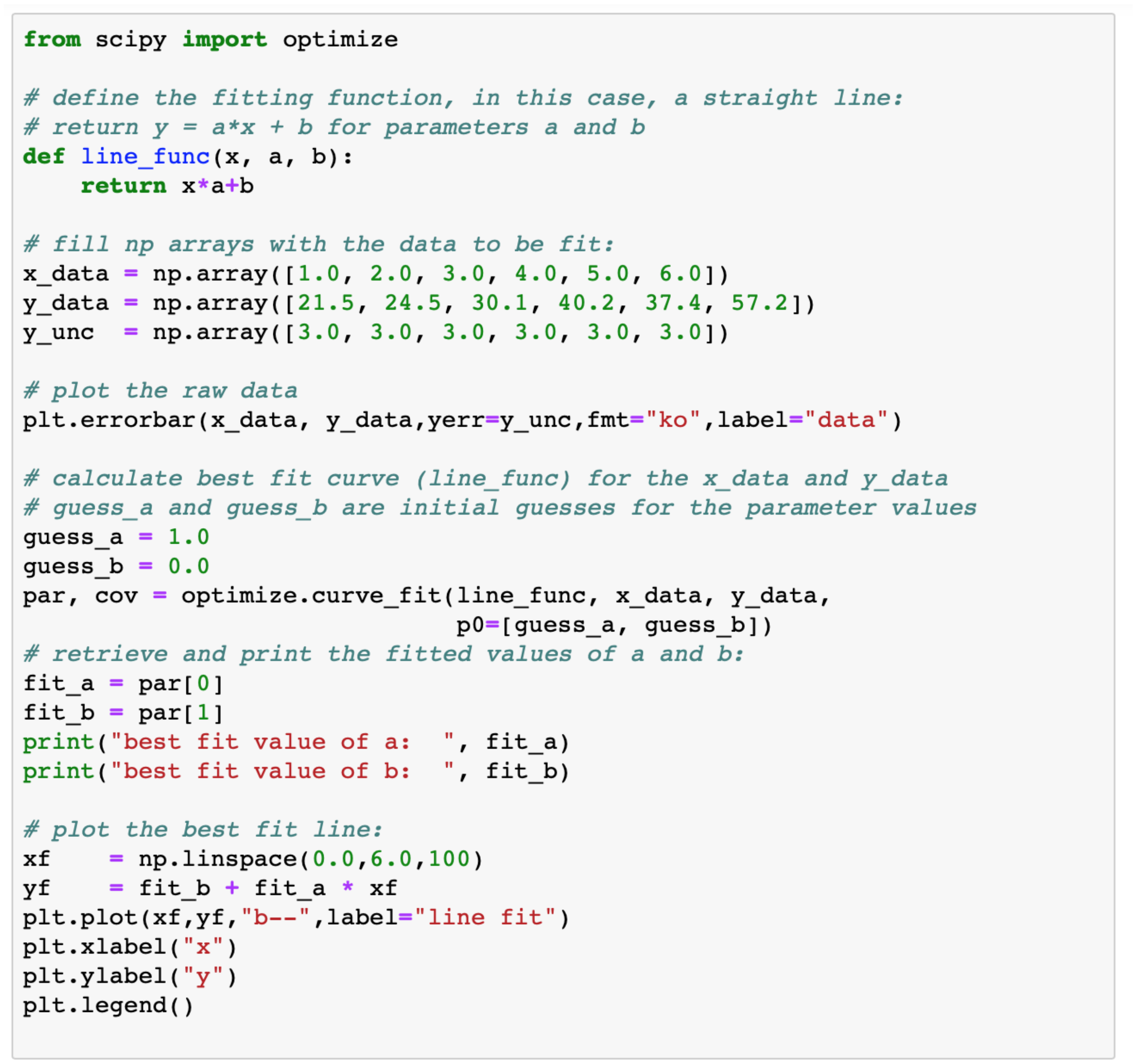
In this case, the function requires three parameters (in the computer science sense) the x data in a numpy array as function parameter x, the slope as function parameter a, and the intercept as function parameter b. When called, the function returns the x data multiplied by the value a, with the value b added. We don't directly call this function, but in principle, it could be called like:

```
y_data = line_func(x_data, 2.0, 0.0)
```

to create a numpy array `y_data` constructed from `x_data` with slope 2 and intercept 0.

The next section filling numpy arrays containing the data, and plotting it with error bars should be familiar by now. The fit itself is performed by the line:

```
par, cov = optimize.curve_fit(line_func, x_data, y_data, p0=[guess_a, guess_b])
```



```python
from scipy import optimize

# define the fitting function, in this case, a straight line:
# return y = a*x + b for parameters a and b
def line_func(x, a, b):
    return x*a+b

# fill np arrays with the data to be fit:
x_data = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
y_data = np.array([21.5, 24.5, 30.1, 40.2, 37.4, 57.2])
y_unc  = np.array([3.0, 3.0, 3.0, 3.0, 3.0, 3.0])

# plot the raw data
plt.errorbar(x_data, y_data,yerr=y_unc,fmt="ko",label="data")

# calculate best fit curve (line_func) for the x_data and y_data
# guess_a and guess_b are initial guesses for the parameter values
guess_a = 1.0
guess_b = 0.0
par, cov = optimize.curve_fit(line_func, x_data, y_data,
                              p0=[guess_a, guess_b])
# retrieve and print the fitted values of a and b:
fit_a = par[0]
fit_b = par[1]
print("best fit value of a:  ", fit_a)
print("best fit value of b:  ", fit_b)

# plot the best fit line:
xf    = np.linspace(0.0,6.0,100)
yf    = fit_b + fit_a * xf
plt.plot(xf,yf,"b--",label="line fit")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

Figure 3.1: Example fitting data to straight line.

```
best fit value of a:   6.494285714297698
best fit value of b:   12.420000000027086

<matplotlib.legend.Legend at 0x101e725f60>
```

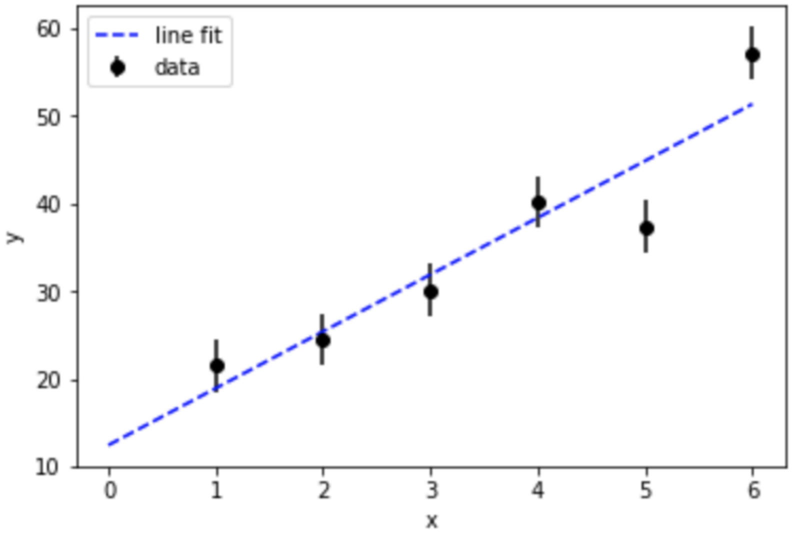


Figure 3.2: Output of example fit.

This performs a fit of the function `line_func` defined above to the x and y data contained in the arrays `x_data` and `y_data`. Numerical fits generally find the local minimum, which is not necessarily the global minimum of interest. It is important therefore, especially for complicated fits, to provide an initial guess near the expected fit values. These are provided to the optional, named, function parameter `p0`, which is set to the python list `[guess_a, guess_b]` which contains our initial guesses for the fit parameters a and b. The function performs a least-squares fit to find the best values of a and b which are returned as the numpy array `par`. The function also returns the covariance matrix as the numpy array `cov`.

The remaining code simply uses the best fit values to plot the fitted function as a dashed line. Numerical fits are fickle. Even if you are only interested in the fitted value, you should always plot the best-fit function and compare the results to your data as in important check for your work.

Table 3.1: Sample data for straight line fit.

x	$y \pm \sigma_y$
1.0	15.9 ± 3.0
2.0	23.6 ± 3.0
3.0	33.9 ± 3.0
4.0	39.7 ± 3.0
5.0	45.0 ± 10.0
6.0	32.4 ± 20.0

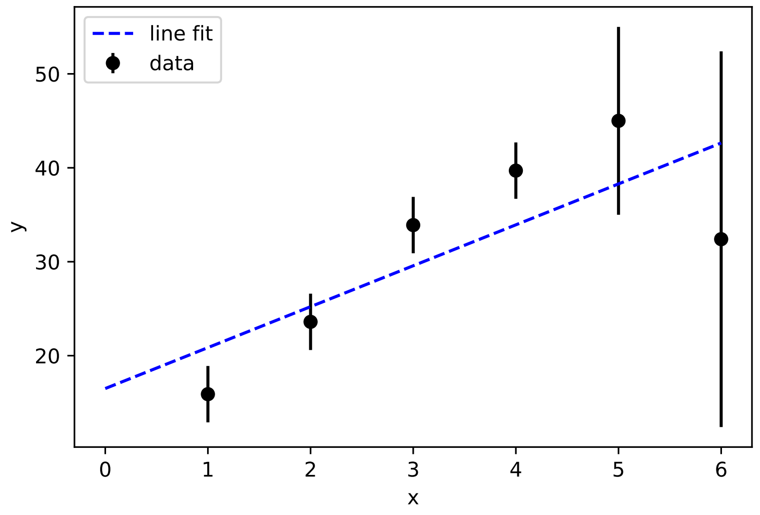


Figure 3.3: This linear fit is biased by the failure to properly account for uncertainties.

Produce **Plot 1** by applying code like that of Fig. 3.2 to the data in Table 3.1 to reproduce the plot in Fig. 3.3.

Notice that the last two data points have larger uncertainties than the other data points. However, the call to the curve_fit function does not provide the parameter uncertainties, and so the function assumes that they all have the value 1. In this case, since the uncertainties are not in fact all the same, the function does not find the correct minimum. The answer is clearly biased toward the poorly measured points, because the function gives these points the same weight as all of the other points.

Look-up the `curve_fit` function and the optional parameter `sigma`. For **Plot 2**, provide the correct uncertainties to the fit. You should observe that the fit results is no longer biased, and more closely tracks the well constrained left side of the plot.

3.3 Fitting a Sine Curve

In this section, you will fit the sample data to a sine function:

$$y = A \sin(kx).$$

Use the following sample data:

x	y	x	y	x	y
0	5.3	4	-9.7	8	15.7
1	15.0	5	-17.4	9	18.5
2	19.2	6	-20.5	10	8.6
3	6.8	7	2.1		

Assume the uncertainty is the same for each y value: $\sigma_i = 2$. For **Plot 3**, plot the data including error bars and the best-fit sine wave.