Python Exercises

April 20, 2020

Chapter 1

Probability Distributions

1.1 Introduction

In this lab, you will create your own numerical simulation of a binomial process, and compare the results of your simulation with the PDFs for the binomial, Poisson, and Gaussian processes in the appropriate limits. For this lab there are only Jupyter notebook entries.

This lab includes a number of code snippets to illustrate the ideas that are being discussed. However, the entire source listing is not available to you, as this would amount to giving away the answer, and we all learn best by doing things ourselves! The examples should help you understand what you should do, but you will have to write your own code. **Do not expect the code snippets as written to simply work for your code without any modification... they are not intended to!**

1.2 Simulating the Binomial Process

Your first task is to create a Monte Carlo simulation for a binomial process. The Monte Carlo method, named after the casino in Monaco, refers to the repeated sampling of random variables to obtain numerical results.

Suppose one single experiment consists of n_{try} trials with a probability ϵ of success. The outcome of each experiment is a single number from 0 to n_{try} reporting the number of the n_{try} trials from that particular experiment which were successful. To study the distribution of outcomes, you will repeat the experiment n_{exp} times.

There are library functions that will simulate this process for you, but for this lab you will create your own simulation. While developing your code, start with a small test. For instance $n_{\text{try}} = 3$ and $n_{\text{exp}} = 5$, as shown in the example below. Implement your Monte Carlo simulation in the following manner:

- Create a 2-D array of shape n_{exp} by n_{try} filled with random values chosen uniformly from 0 to 1.0:

```
nexp = 5
ntry = 3
x = np.random.uniform(size=(nexp,ntry))
print(x)
```

```
[[0.90348221 0.39308051 0.62396996]
 [0.6378774  0.88049907 0.29917202]
 [0.70219827 0.90320616 0.88138193]
 [0.4057498  0.45244662 0.26707032]
 [0.16286487 0.8892147  0.14847623]]
```

This array associates a random value with each trial from each experiment.

- Consider a trial successful if the randomly chosen value is less than ϵ. In this example, taking $\epsilon = 0.5$ and assuming 1 indicates success and 0 a failure, we obtain:

```
[[0 1 0]
 [0 0 1]
 [0 0 0]
 [1 1 1]
 [1 0 1]]
```

In the first simulated experiment, only the second trial was successful. In the second experiment, only the last trial successful. And so on.

- Next, count the number of trials that were successful in each experiment. The result will be a 1-D array of length n_{exp}. Consider using the `np.sum` function with the `axis` parameter (If documentation is unclear to you, check the examples). In this example, we obtain the array of outcomes m:

```
[1 1 0 3 2]
```

This is the outcome of our five simulated experiments. The first and second experiment have one out of three trials successful, the third experiment had zero out of three trials successful, and so on.

Work through the example and make sure you see how each result follows from the previous step. Then implement and test your own version of this algorithm in Scientific Python. If you are an experienced programmer, you should put your simulation into a function like:

```
def throw_binomial(nexp,ntry,eps):
    x = np.random.uniform(size=(nexp,ntry))
    #...your simulation follows...
```

This will make your code much easier to manage, because you can simply call this function each time you need to run your simulation, but it is optional. Cutting and pasting is also permissible.

When validating a numerical calculation, think hard about good test cases. For instance, if you only test with the value of $\epsilon = 0.5$, you won't catch a bug that mistakes success for failure. Try a few different test cases, with reasonably small values for n_{try} and n_{exp} and check your numerical simulation. Boundaries often make a good check, for instance $\epsilon = 0$ and $\epsilon = 1$.

Another effective validation strategy is to check known mathematical relations using your simulation. For instance, we know that the mean value of a Binomial distribution is given by:

$$\bar{m} = n_{\text{try}} \cdot \epsilon \tag{1.1}$$

and that the variance is given by:

$$\sigma^2 = n_{\text{try}} \cdot \epsilon \, (1 - \epsilon) \tag{1.2}$$

and so these predicted values, calculated from your parameters ϵ and n_{try} can be compared to the mean and variance of the outcome array m from your simulation, calculated using the numpy functions `np.mean` and `np.var`. A comparison with $n_{\text{exp}} = 1000$, $n_{\text{try}} = 10$, and $\epsilon = 0.5$ should result in something like:

```
print("mean expected:    ", ntry * eps)
print("mean simulated:   ", np.mean(m))
print("var expected:     ", ntry * eps * (1.0 - eps))
print("var simulated:    ", np.var(m))
```

```
mean expected:     5.0
mean simulated:    5.015
var expected:      2.5
var simulated:     2.3847750000000003
```

Jupyter Notebook 1.1. Produce a large number of pseudo-experiments by setting $n_{\text{exp}} = 1000$ and comment out any debugging print statements which will get very long. Pick two well chosen test cases with different values of n_{try} and ϵ and compare the expected and simulated mean and variances.

The simulated values will fluctuate by a fractional amount $1/\sqrt{n_{\text{exp}}}$. So for $n_{\text{exp}} = 1000$, we expect the expected and simulated values to agree within about 3%.

Jupyter Notebook 1.2. Increase the number of pseudo-experiments to $n_{\text{exp}} = 100000$. The agreement should improve to better than 1%. Be certain to print your test cases and the results in a clear way in your notebook.

1.3 Histogram for the Binomial Process

We use histograms to represent the distribution of numerical data. In this case, our histogram simply reports the number of times each particular outcome occurs. Fill an output array m with the simulated results of $n_{\text{exp}} = 1000$ experiments each consisting of $n_{\text{try}} = 10$ trials with success rate $\epsilon = 0.25$. There are eleven possible outcomes for each of these experiments: $0, 1, 2, \ldots, 10$. We want to know how often each of these outcomes occurred.

Jupyter Notebook 1.3. Construct and plot a histogram from your simulated results in array m using the `np.histogram` function:

```
counts,edges = np.histogram(m,bins=11,range=(0,11))
print("counts:     ", counts)
print("total:      ", np.sum(counts))
print("bin edges:  ", edges)
```

```
counts:      [ 65 172 284 269 122  64  19   4   1   0   0]
total:       1000
bin edges:   [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
```

This calculates a histogram with eleven bins covering the range from zero to eleven, and reports the number of outcomes from m that occur in each bin. It returns two arrays, which we save as `counts` and `edges`. We interpret the `counts` array as follows: the first entry tells us that 65 experiments had zero successes, the second entry that 172 experiments had one success, the third entry that 284 experiments had two successes, and so on. The exact values will vary each time you run the simulation, but in all cases the sum across all bins will equal the total number of experiments $n_{\text{exp}} = 1000$.

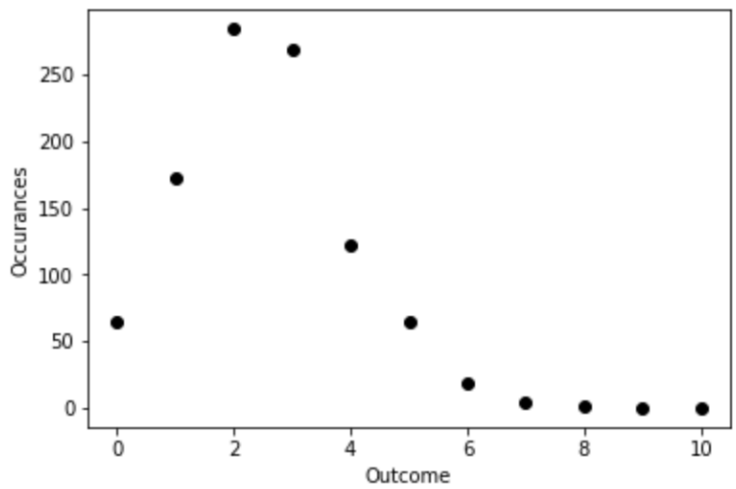
Histograms are most often used to handle continuous data, so you have to take a bit more care when using them to display discrete data (integers) as we are doing here. Consider the bin edges array `edges` that was returned in this example:

```
bin edges:   [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
```

which you will notice has length 12. This is because the array contains the **edges** of eleven consecutive bins. Technically the first bin is the count of all outcomes in the range from zero (inclusive) to just below one (exclusive). The next bin is the count of all outcomes in the range from one (inclusive) to just below two (exclusive). In this case, however, we are using discrete data, and so there are no entries with values like 1.73 to consider. All the entries in the first bin are from experiments with the outcome zero, while all the entries in the second bin are from the outcome one. The best way to plot this histogram, therefore, is to associate each count with the leading bin edge, that is:

```
plt.plot(edges[:-1],counts,"ko")
plt.xlabel("Outcome")
plt.ylabel("Occurances")
print("edges[:-1]:  ", edges[:-1])
```

```
edges[:-1]:   [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```



Notice the essential trick for plotting histogram with discrete data in integer bins is to use the slice `edges[:-1]` of the bin edges data

```
edges[:-1]:   [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

as the x values for plotting the occurrences of each outcome. This slice (all but the last entry) essentially throws out the superfluous bin edge 11. As we will see later, this trick only works for discrete data in integer bins. Notice that in resulting plot, we have the number of occurances at each of the eleven possible outcomes correctly centered over the numbers $0, 1, 2, \ldots, 10$.

1.4 Error Bars

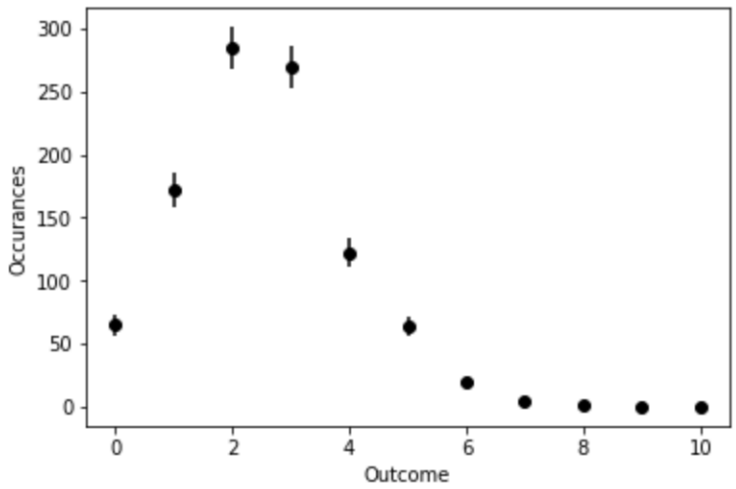
Jupyter Notebook 1.4. Add error bars to your histogram and make a new plot.

Run your simulation a few times and you will observe that the simulated values fluctuate slightly each time you run your code. But how much should we expect these values to fluctuate? If you consider a single bin of your histogram in isolation, it contains a single count N, which is itself the result of a simple counting experiment. Counting experiments are described by the Poisson distribution. Our best estimate for the mean value λ of this counting experiment is simply our count N. For the Poisson distribution the variance $\sigma^2 = \lambda$, and so we expect $\sigma = \sqrt{\lambda} = \sqrt{N}$. We expect each bin in our histograms to fluctuate by an amount σ which is the square root of the value of the bin.

To aid in interpreting histograms, it is useful to indicate the amount we expect each bin to fluctuate by adding to the data point an "error bar" with a length equal to the square root of the size of the number of events in the bin:

```
errs = counts**0.5
plt.errorbar(edges[:-1],counts,yerr=errs,fmt="ko")
plt.xlabel("Outcome")
plt.ylabel("Occurances")
```

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



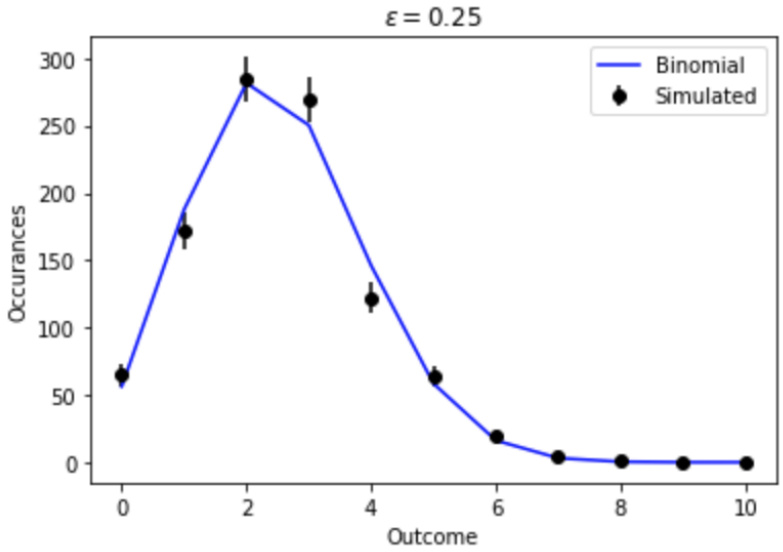
This provides an intuitive visual indication for the size of the statistical fluctuations associated with the counts reported by the histogram. Notice that the poorly named `errorbar` function plots *both* the data point and the error part, so it is a replacement for the `plot` function, not something you must call in addition.

1.5 Comparison with Binomial PDF

Next we'd like to compare our Monte Carlo simulation with the PDF for the binomial distribution. We'll use the `scipy.stats.binom.pmf` function, which is the PDF for the discrete binomial distribution available in Scientific Python. This PDF is only non-zero at integer values, so we'll simply evaluate it at each discrete occurrence, i.e. the same slice `edges[0:-1]` which we used to plot the histogram:

```
from scipy.stats import binom
errs = counts**0.5
plt.errorbar(edges[:-1],counts,yerr=errs,fmt="ko",
             label="Simulated")
plt.xlabel("Outcome")
plt.ylabel("Occurances")
xpred = edges[:-1]
ypred = nexp * binom.pmf(xpred, ntry,eps)
plt.plot(xpred, ypred,"b-",label="Binomial")
plt.legend()
plt.title("$\epsilon=0.25$")
```

```
Text(0.5,1,'$\\epsilon=0.25$')
```



Notice that we scale the PDF by the number of experiments n_{exp}. The PDF is the expected frequency of each outcome for a single experiment, but we are plotting the number of occurrences for n_{exp} experiments.

Jupyter Notebook 1.5. Compare the output of your Monte Carlo simulation with the Binomial distribution PDF with $n_{\text{exp}} = 1000$ and $n_{\text{try}} = 10$, and $\epsilon = 0.75$. See example for $\epsilon = 0.25$ in the plot above.

1.6 The Poisson Limit

The Poisson distribution follows from the Binomial distribution in the limit that $n_{\text{try}} \to \infty$. Recall that the mean value of the Poisson distribution is $\bar{m} = \epsilon\, n_{\text{try}}$. If we kept the success rate ϵ as a

parameter, than any finite value of ϵ would cause the mean of the distribution to diverge to infinity. If instead we hold the new parameter λ constant, and set:

$$\epsilon = \frac{\lambda}{n_{\rm try}}$$

we see that $\epsilon \to 0$ as $n_{\rm try} \to \infty$ and the mean of the Poisson distribution remains at the fixed value λ.

We'll explore this limit numerically simply by taking $n_{\rm try}$ to the large (but finite) value of 1000. Re-run your Monte Carlo simulation using the parameters $n_{\rm try} = 1000$, $n_{\rm exp} = 1000$, and $\epsilon = \lambda/n_{\rm try}$. For now, take $\lambda = 2.0$. Instead of the Binomial distribution, compare your simulation to the Poisson distribution PDF using the `poisson.pmf` function:

```
from scipy.stats import poisson
xpred = edges[:-1]
ypred = nexp * poisson.pmf(xpred, lamb)
```

Note that the first argument of the `pmf` function is the array of positions to evaluate the function at, while the second is the Poisson parameter λ. Also note that sadly `lambda` is a reserved word in python, and so you cannot use it as a variable name.

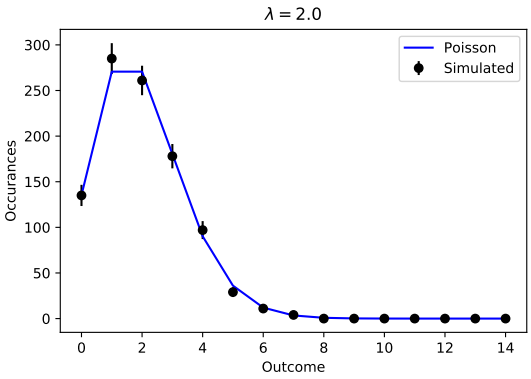


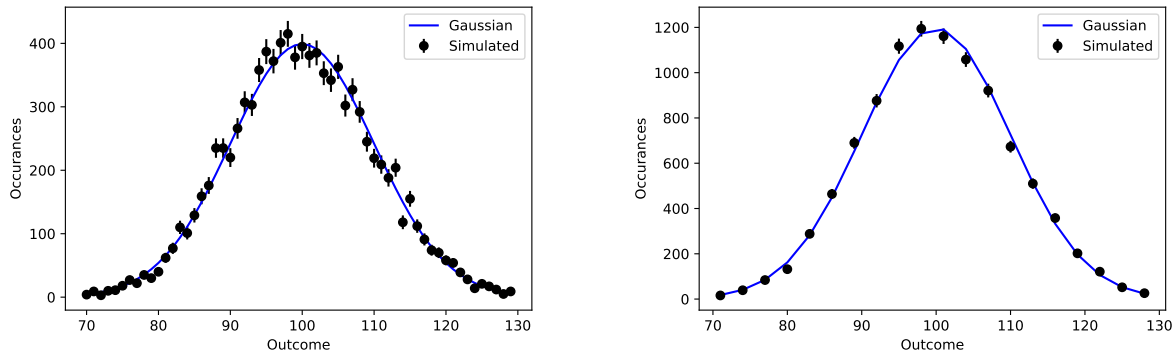
Figure 1.1: Example of the expected result for the Monte Carlo simulation data in comparison to the Poisson PDF for for $\lambda = 2$.

Jupyter Notebook 1.6. In the Poisson limit, compare the output of your Monte Carlo simulation to the Poisson PDF for $\lambda = 5.2$. Plot the histogram with 15 bins for the outcomes: 0,1,2,3,...,14. An example for $\lambda = 2$ is shown in Fig. 1.1.

1.7 The Gaussian Limit

As the mean value λ of the Poisson distribution gets larger, the Poisson distribution resembles the Gaussian distribution. We will simulate this numerically by taking our Monte Carlo simulation in the Poisson limit, as above, with $\lambda = 100$. Initially use integer bins inclusively in the range 70 to 130, e.g.:

```
counts,edges = np.histogram(m,bins=60,range=(70,130))
```



(a) (b)

Figure 1.2: Example of the expected result for the Monte Carlo simulation data in comparison to the Gaussian PDF for $\lambda = 100$.

and compare with the Gaussian (also called normal) distribution PDF, e.g.: `scipy.stats.norm.pdf` function:

```
from scipy.stats import norm
xpred = edges[:-1]
ypred = nexp * norm.pdf(xpred, loc=lamb, scale=lamb**0.5).
```

Jupyter Notebook 1.7. Set the parameter `loc` to the mean value, and the parameter `scale` to σ. Recall that in the Poisson limit $\sigma^2 = \lambda$, which is why we set `scale=lamb**0.5` in the example. This should reproduce the plot in Fig. 1.2a.

You should see that 60 bins is rather unwieldy. We'll reduce the number of bins, but that's actually a bit more complicated than you might expect: non-integer bins with discrete data is about the most challenging binning you can tackle. You'll have to do the following:

- While keeping the range of 70 to 130, set the number bins to `bins=20` when filling your histogram.
- Our trick to use edges[:-1] will no longer work, since now the data for each bin is associated with a range of values in the bin. Each bin is now 3 integers wide. If we live this as is, edges[:-1] will position the x value of the bin at its leading edge: 70, 73, 76 , ... and the data will be plotted with an observable bias. For continuous data, we often simply use the middle of the bin:

  ```
  cbins = (edges[:-1] + edges[1:])/2.
  ```

  This would position the x value of the bins at 71.5, 74.5. 77.5, ... and that seems more reasonable choice. In this specific case of discrete data which doesn't extend all the way to the right edge this is still slightly biased. The first bin in our example represents the count of all outcomes in the range from 70 (inclusive) to below 73 (exclusive). So its center should be at 71. To be precise for this specific case, the x position we should use for plotting the contents of each bin is:

```
cbins = (edges[:-1] + edges[1:] -1 )/2
```

- The normalization of the PDF to the data now requires an additional scale factor to account for the wider bins (which integrate more probability). You need to scale by an additional factor of 3 to account for the fact that each bin is 3 integers wide.

Jupyter Notebook 1.8. Using the techniques described above, reproduce Fig 1.2b, which shows that the Binomial distributions becomes a Gaussian distribution as the mean value of the distribution becomes large.

Chapter 2

The Central Limit Theorem and Experimental Uncertainties

2.1 Introduction

In this lab, you will produce a numerical demonstration of the central limit theorem. You will also model the propagation of uncertainties and compare with the calculated uncertainties. For this lab there are only Jupyter notebook entries.

2.2 Sampling Distributions

Scientific python provides functions to draw random samples according to various distributions. In today's lab, we will draw samples uniformly in the interval $[-1, 1]$, as demonstrated in Fig. 2.1. The line

```
r = np.random.uniform(low=-1.0, high=1.0, size=NEXP)
```

creates a NumPy array `r` which contains `NEXP` entries, with each entry chosen uniformly and randomly in the range from -1 to 1. In the example, these events are displayed in a histogram. When plotting histograms with plenty of statistics (one million entries here) and fine binning (60 bins here) it is usually preferable to use lines instead of points with error bars for plotting the histograms, as is done in this example. Notice, however, that even with one million events, there are still statistical fluctuations which prevent the curve from being perfectly smooth.

In Fig. 2.2, entries are instead drawn from the Gaussian distribution with the line:

```
r = np.random.normal(loc=5.0,scale=1.5,size=NEXP).
```

The histogram is plotted with a logarithmic y scale:

```
plt.semilogy()
```

which results in the Gaussian distribution appearing as a parabola. The histogram is compared to the Gaussian PDF appropriately normalized:

```
x = np.linspace(MIN,MAX,100)
y = NEXP*binsize*stats.norm.pdf(x,loc=MEAN,scale=SIGMA)
```

```
NEXP  = 1000000
NBINS = 60
MAX   = 3
r = np.random.uniform(low=-1.0, high=1.0, size=NEXP)
h,edges = np.histogram(r,bins=NBINS,range=(-MAX,MAX))
cbins = (edges[:-1] + edges[1:])/2.0
plt.plot(cbins,h,"k-")
plt.xlabel("$x$")
plt.ylabel("Entries")
```

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

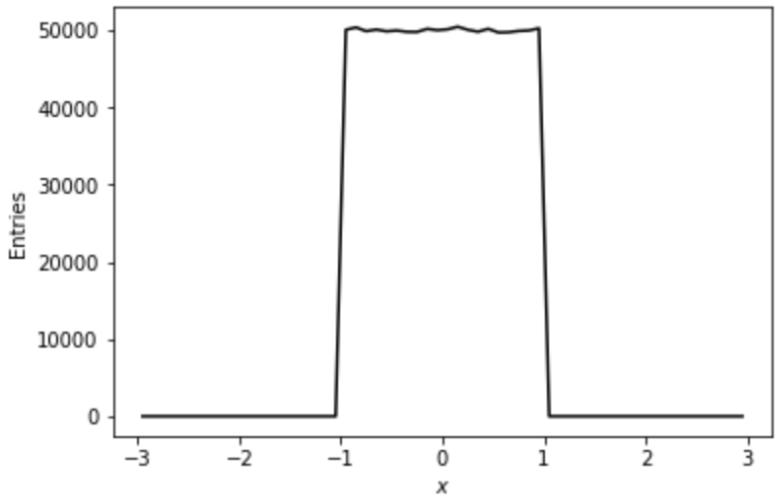


Figure 2.1: Sampling from the uniform distribution.

```
from scipy import stats
NEXP  = 200000
NBINS = 60
MIN   = 0
MAX   = 10
MEAN  = 5.0
SIGMA = 1.5
r = np.random.normal(loc=5.0,scale=1.5,size=NEXP)
h,edges = np.histogram(r,bins=NBINS,range=(MIN,MAX))
cbins = (edges[:-1] + edges[1:])/2.0
plt.plot(cbins,h,"k-",label="Monte-Carlo")
binsize = float(MAX-MIN)/NBINS
x = np.linspace(MIN,MAX,100)
y = NEXP*binsize*stats.norm.pdf(x,loc=MEAN,scale=SIGMA)
plt.plot(x,y,"r--",label="PDF")
plt.xlabel("$x$")
plt.ylabel("Entries")
plt.semilogy()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1a26a4ce10>
```

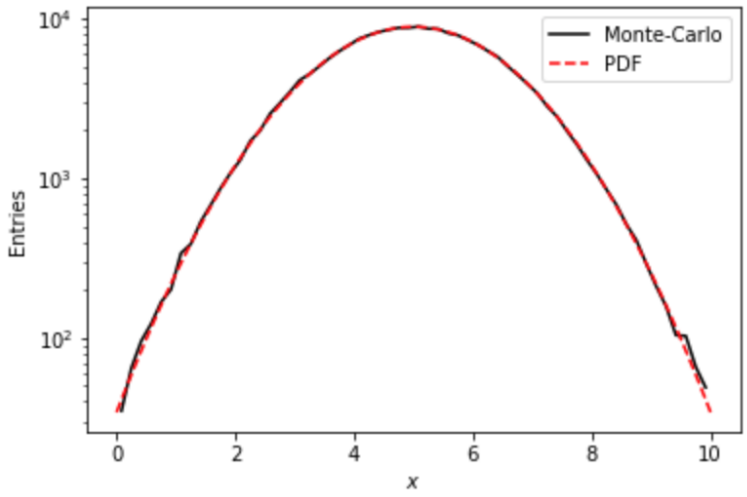


Figure 2.2: Sampling from the Gaussian distribution and comparison with the Gaussian PDF.

2.3 Demonstration of the Central Limit Theorem

In this section, you'll show that average value of random variables chosen uniformly from -1 to 1 approaches a Gaussian distribution, consistent with the central limit theorem. First create a 2-D array of size `NEXP` by `NAVG` filled with uniform random values in the interval from -1 to 1, as follows:

```
r = np.random.uniform(low=-1.0, high=1.0, size=(NEXP,NAVG))
```

Then calculate averages values from `NAVG` entries:

```
x = np.sum(r, axis=1)/float(NAVG)
```

From the Central Limit Theorem, we expect the entries in x to approach a Gaussian distribution.

Jupyter Notebook 2.1. Set `NEXP` to 1000000 for plenty of statistics. Produce three different histograms with 40 bins covering the range from -1.2 to 1.2 for three values `NAVG`: 1,2, and 3. Plot all three histogram in the same graph with appropriate legend.

Your plot will show that already for three contributions to the average, the result looks quite Gaussian on a linear scale. For more precise comparison, we will use a log scale and compare to the PDF.

Jupyter Notebook 2.2. Calculate `NEXP`= 1000000 average values `x` for `NAVG`= 10. Calculate the mean value of the entries in `x` using the `np.mean` function. Calculate σ for the entries in x by taking the square root of the output from the `np.var` (variance) function. Produce a histograms with 20 bins covering the range from -0.5 to 0.5 for the average values. Compare with a Gaussian distribution, appropriately normalized, using your calculated values from the mean and sigma. Plot both the histogram and PDF on the same graph, including an appropriate legend. Use a logarithmic y axis.

2.4 Propagation of Uncertainties

Consider two measured values $a \pm \sigma_a$ and $b \pm \sigma_b$. If we calculate the quantity $c = a + b$ or $c = a - b$, the uncertainty on the calculated value c is given by:

$$\sigma_c = \sqrt{\sigma_a^2 + \sigma_b^2}.$$

If instead, we calculate $c = a * b$ or $c = a/b$ the fractional uncertainty on c is given by:

$$\frac{\sigma_c}{c} = \sqrt{\left(\frac{\sigma_a}{a}\right)^2 + \left(\frac{\sigma_b}{b}\right)^2}.$$

In this section, you'll develop a numerical simulation for the propagation of uncertainties under addition, subtraction, multiplication, and division. An example, for $c = a + b$ is shown in Fig. 2.3.

Pick values for a, b, σ_a and σ_b for simulating subtraction: $c = a - b$. Record them out in your logbook. Choose values that are different from what is plotted in Fig. 2.3.

Jupyter Notebook 2.3. Simulate the measurement a by drawing 100,000 random samples from a Gaussian distribution with mean a and sigma σ_a, and do likewise for b. Calculate the values $c = a - b$ from the a and b values. Plot the distribution of all three variables (as in Fig. 2.3) as histograms with 50 bins and an appropriate range. Calculate the mean and variance of the simulated c values and compare to your expectations from the standard propagation of uncertainties.

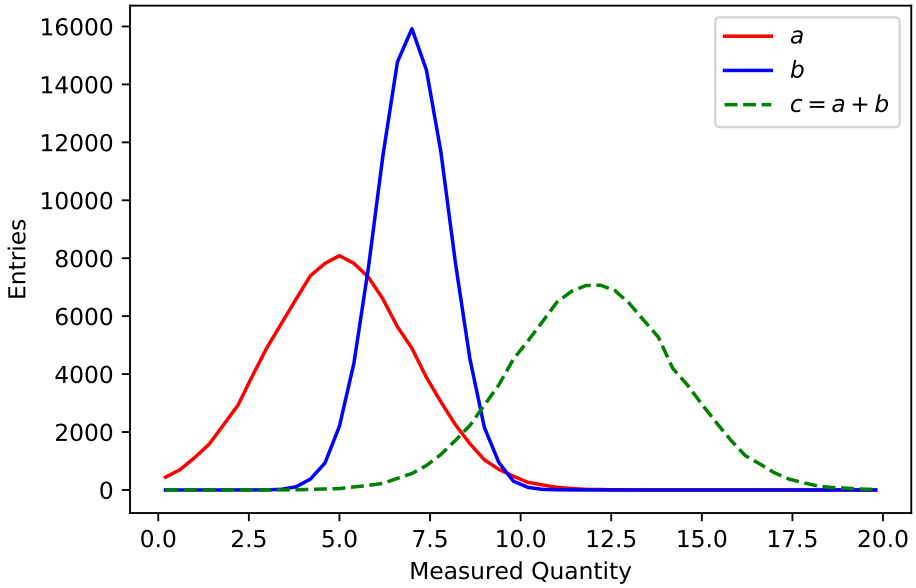


Figure 2.3: Simulation of many measurements of the quantity $c = a + b$.

Pick values for a, b, σ_a and σ_b for simulating division: $c = a/b$. Record them in your logbook. Choose values that will allow you to see the distributions of a, b, and c all on the same plot.

Jupyter Notebook 2.4. Simulate the measurements a and b using the same procedure as in the previous plots, but with your new values. Calculate the values of $c = a/b$ from the a and b values. Plot the distribution of all three variables (as in Fig. 2.3) as histograms with 50 bins and an appropriate range. Calculate the mean and variance of the simulated c values and compare to your expectations from the standard propagation of uncertainties. (Note that for certain values of a and b and their uncertainties, the assumptions in the standard propogation of uncertainties are not met!)

Chapter 3

Curve Fitting

3.1 Introduction

In this lab, you will learn about curve fitting with Scientific Python function `curve_fit`. For this lab there are only Jupyter notebook entries.

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:

$$\chi^2 = \sum_i \frac{(f(x_i;p) - y_i)^2}{\sigma_i^2}. \tag{3.1}$$

If the uncertainties σ_i are not specified, the function assumes $\sigma_i = 1$, which still finds the correct minimum if the 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. 3.1. 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, not the mathematical 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])
```

```
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()
```

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

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

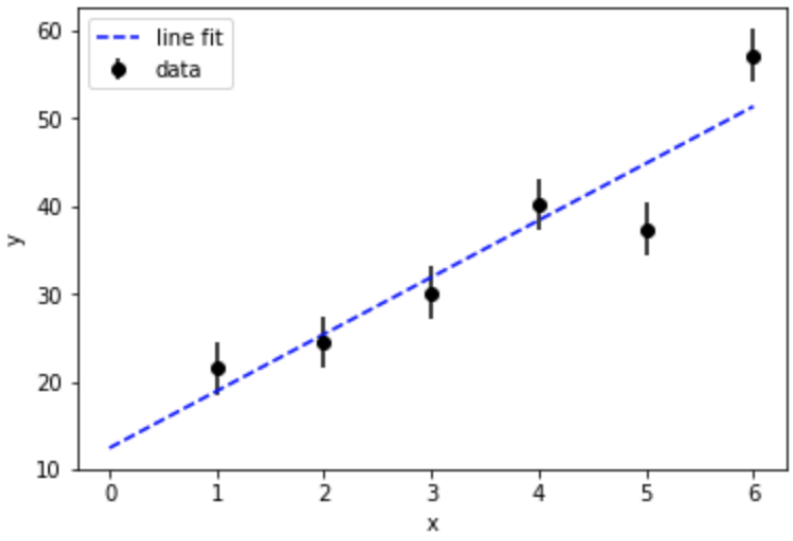


Figure 3.1: Example fitting data to straight line.

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

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.

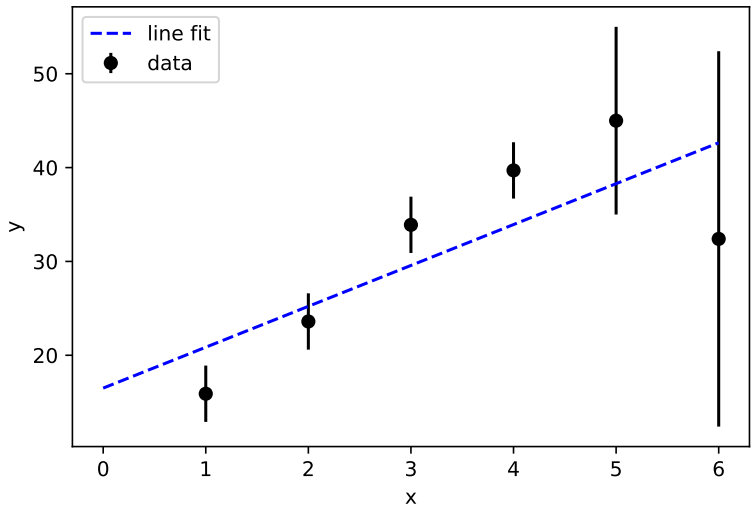


Figure 3.2: This linear fit is biased by the failure to properly account for uncertainties. An unbiased fit would track the well constrained points more closely.

Jupyter Notebook 3.1. Apply code like that of Fig. 3.1 to the data in Table 3.1. Plot the data including error bars and the best-fit function to obtain a result like that of Fig. 3.2.

Notice that the last two data points in Table 3.1 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 (as in Fig. 3.2) toward the poorly measured points, because the function gives these points the same weight as all of the other points.

Jupyter Notebook 3.2. Look-up the `curve_fit` function and the optional parameter `sigma`. Provide the correct uncertainties to the fit and make a new plot with the data and 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 Parameter Uncertainties

```
# will fit y_data to a constant value:
def constant_func(x, a):
    return a

# choose y_data indepdent of x values, from
#   Gaussian with a mean of 50 and sigma of 10
x_data = np.arange(100)
y_data = np.random.normal(50.0, 10.0, size=100)

# fit for the best fit constant value, should find the mean:
guess_a = 0.0
par, cov = optimize.curve_fit(constant_func, x_data, y_data, p0=[guess_a])

# determine the best fit parameter and it's uncertainty:
unc = np.sqrt(np.diag(cov))
fit_a = par[0]
unc_a = unc[0]

#print the results
print("mean of y data:  ", np.mean(y_data))
print("fitted constant: ", fit_a)
print("uncertainty:     ", unc_a)
```

```
mean of y data:   48.426944799228266
fitted constant:  48.42694479930867
uncertainty:      0.9711303089930362
```

Figure 3.3: Example obtaining parameter uncertainties.

We will show in lecture that the uncertainty σ_{p_i} on the ith parameter p_i can be determined from the second derivative of the χ^2 function:

$$\frac{d^2\chi^2}{dp_i^2} = \frac{2}{\sigma_{p_i}^2}.$$

In general, for M parameters, the $M \times M$ covariance matrix is calculated as:

$$C(i,j) = 2 \cdot \left(\frac{d^2\chi^2}{dp_i dp_j} \right)^{-1}$$

from which we can see that the diagonals are simply the parameter uncertainties squared:

$$C(i,i) = 2 \cdot \left(\frac{d^2\chi^2}{dp_i^2} \right)^{-1} = \sigma_{p_i}^2.$$

The off-diagonal elements contain information about how parameters are correlated, and for a well designed fit function they should be close to zero.

The `curve_fit` function returns both the best fit parameter values and the covariance matrix:

```
par, cov = optimize.curve_fit(...)
```

For a fit with M parameters, we can obtain an array containing the M parameter uncertainties from the square root of the diagonals of the $M \times M$ covariance matrix:

```
unc = np.sqrt(np.diag(cov))
```

An example is shown in Fig. 3.3, for a single parameter. In this case, the y-values are simply 100 random numbers drawn from a Gaussian distribution with mean $m = 50$ and a width $\sigma_y = 10$. The best fit constant value is simply the mean of the y-values. The uncertainty on the mean value should be:

$$\sigma_m = \sigma_y/\sqrt{N} = 10/\sqrt{100} = 1.0$$

Indeed we obtain a best fit constant value and it's uncertainty close to these expected values.

It's surprising actually, that the fit returns the correct uncertainty. Look through the code carefully and notice that nowhere has the uncertainty on the y parameters σ_y been provided to the fit. So how can it possibly deduce the correct uncertainty $\sigma_m = \sigma_y/\sqrt{N}$?

The answer is that behind the scenes, the `curve_fit` function is being really quite clever (too clever, in my opinion, for a default behavior!) By default, the covariance matrix returned by the `curve_fit` function is scaled by the factor:

$$\alpha = \frac{\chi^2_{\rm min}}{\rm NDF}$$

the minimum value of the χ^2 divided by the number of degrees of freedom (number of data points minus number of parameters). We'll show in lecture that α is around 1 for a least-squares fit with an appropriate model and correct uncertainties. So nominally this factor is one, and has no effect. But consider what happens if the actual uncertainties are σ while the χ^2 used in the fit assumes they are, for example, "1". In this case, the calculated χ^2 is:

$$\chi^2 = \sum_i \frac{(f(x_i;p) - y_i)^2}{1}$$

which differs from the correct χ^2:

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

by a factor of σ^2. This means that while the correct value for α is nearly one, the calculated value of alpha will be σ^2. This is precisely the factor needed to scale the squared parameter uncertainties to account for the fact that the initial uncertainty was σ but we assumed 1.

This behavior is controlled by the parameter `absolute_sigma`. By default, the function sets `absolute_sigma = False` and scales the covariance matrix as just described. On the other hand, if you want to simply use the provided uncertainties without re-scaling the covariance matrix, you must remember to set `absolute_sigma=False`. I think this is a really poor choice of default behavior... it's really quite a fancy thing to do implicitly. In cases when you know the uncertainty on your data points, this re-scaling actually results in less correct estimate for the uncertainties. This is because for a good model with proper uncertainties, the factor α is near one, but not exactly one.

Jupyter Notebook 3.3. Repeat the fit in Fig. 3.3, but set the uncertainties on the y values to 10 and also set `absolute_sigma = True` in the fit. Record the uncertainty.

Jupyter Notebook 3.4. Leave the uncertainties on the y values unspecified and set `absolute_sigma = True` in the fit. You should obtain an uncertainty of 0.1. Record your value and explain why you obtained this value.

As a rule of thumb, when using `curve_fit`, if you provide explicit uncertainties, you should remember to set `absolute_sigma=True`. And really, for precision work, you should almost always be providing explicit uncertainties.

3.4 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: $\sigma_i = 2$.

Jupyter Notebook 3.5. Plot the sample data including error bars and the best-fit sine wave. Print the best fit parameter values and their uncertainties. This fit is highly sensitive to the initial guess, so make sure you provide good starting values close the correct answer. Remember to set `absolute_sigma=True`.