Physics 80 Lab Manual

Michael Mulhearn, Emilija Pantic

February 24, 2021

Contents

Chapter 1

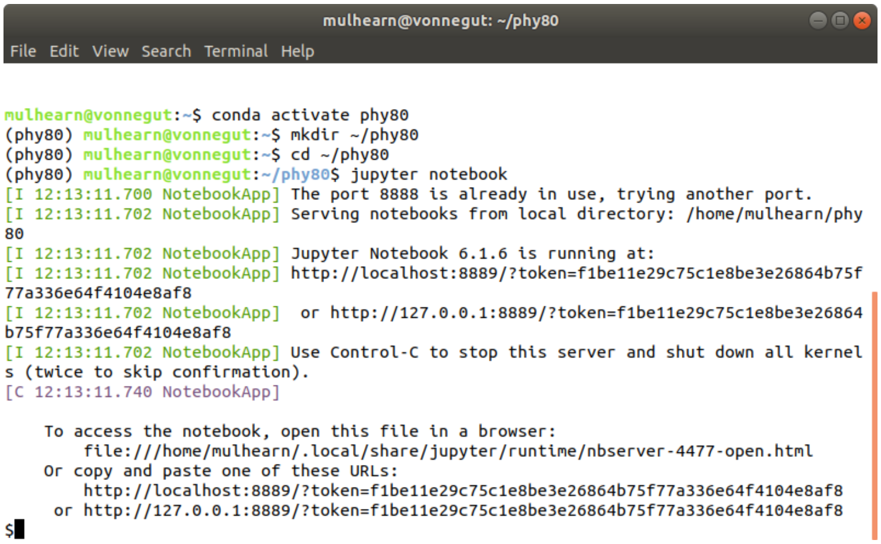
Introduction to Scientific Python

1.1 Introduction

This lab will highlight key Scientific Python features starting from Python fundamentals.

1.2 Starting a Jupyter notebook

This course will make extensive use of the Jupyter Notebook interface to Scientific Python, which is well suited to academic work (including independent research) because it combines code with output in digestable chunks. Even when the end product is a polished peice of software, much of the development occurs in the sort of interactive sessions that Jupyter Notebooks provide.

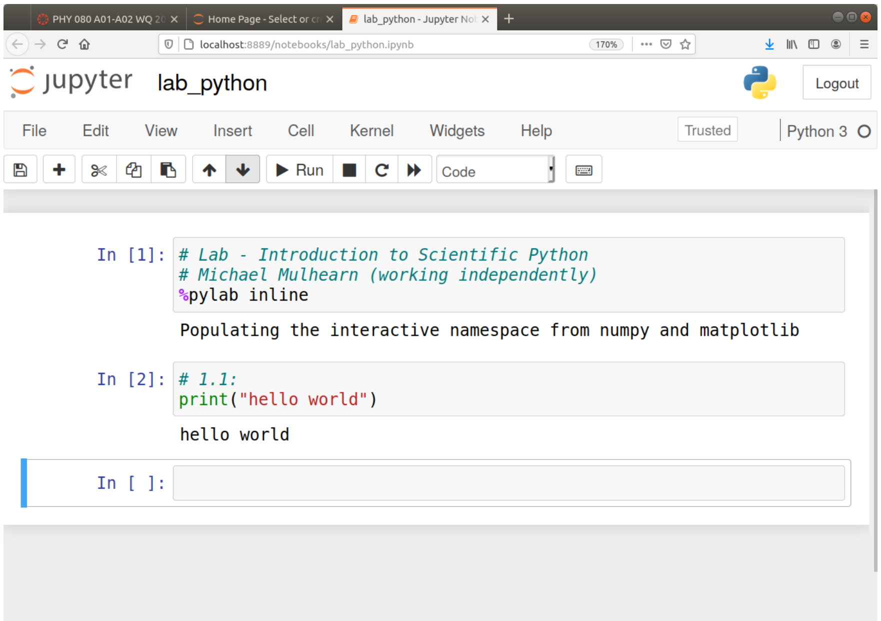

mulhearn@vonnegut: ~/phy80
File Edit View Search Terminal Help

```
mulhearn@vonnegut:~$ conda activate phy80
(phy80) mulhearn@vonnegut:~$ mkdir ~/phy80
(phy80) mulhearn@vonnegut:~$ cd ~/phy80
(phy80) mulhearn@vonnegut:~/phy80$ jupyter notebook
[I 12:13:11.700 NotebookApp] The port 8888 is already in use, trying another port.
[I 12:13:11.702 NotebookApp] Serving notebooks from local directory: /home/mulhearn/phy
80
[I 12:13:11.702 NotebookApp] Jupyter Notebook 6.1.6 is running at:
[I 12:13:11.702 NotebookApp] http://localhost:8889/?token=f1be11e29c75c1e8be3e26864b75f
77a336e64f4104e8af8
[I 12:13:11.702 NotebookApp]  or http://127.0.0.1:8889/?token=f1be11e29c75c1e8be3e26864
b75f77a336e64f4104e8af8
[I 12:13:11.702 NotebookApp] Use Control-C to stop this server and shut down all kernel
s (twice to skip confirmation).
[C 12:13:11.740 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/mulhearn/.local/share/jupyter/runtime/nbserver-4477-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=f1be11e29c75c1e8be3e26864b75f77a336e64f4104e8af8
     or http://127.0.0.1:8889/?token=f1be11e29c75c1e8be3e26864b75f77a336e64f4104e8af8
$
```

Figure 1.1: Example starting Jupyter Notebook from the Linux command line. In Windows, you will need to open the Anaconda Prompt instead of a terminal.

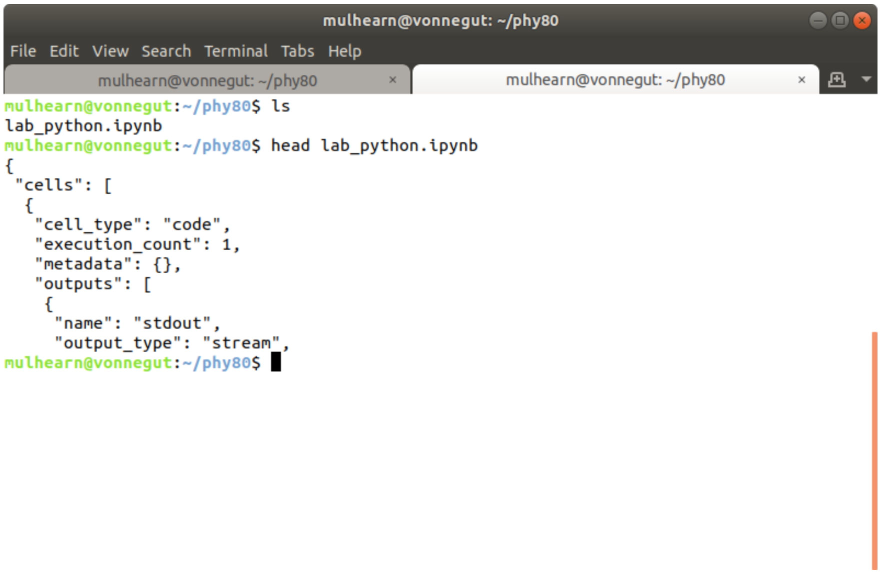
After following the software installation instructions on the course website, activate the Physics 80 environment with:

```
$ conda activate phy80
```



```
In [1]: # Lab - Introduction to Scientific Python
        # Michael Mulhearn (working independently)
        %pylab inline

        Populating the interactive namespace from numpy and matplotlib

In [2]: # 1.1:
        print("hello world")

        hello world

In [ ]:
```

Figure 1.2: The Hello World example Jupyter Notebook.


mulhearn@vonnegut: ~/phy80
File
Edit
View
Search
Terminal
Tabs
Help
mulhearn@vonnegut: ~/phy80
mulhearn@vonnegut: ~/phy80

```
mulhearn@vonnegut:~/phy80$ ls
lab_python.ipynb
mulhearn@vonnegut:~/phy80$ head lab_python.ipynb
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 1,
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
mulhearn@vonnegut:~/phy80$ 
```

Figure 1.3: Example showing the saved Jupyter notebook. Notice that notebook file (ipynb) is not human readable on its own: it requires the Jupyter software to render it in a human readable form.

Then, navigate to a working directory for this session, and start the notebook with:

```
$ jupyter notebook
```

This should start the Jupyter Notebook server and open a client in your web browser. An example starting a Jupyter Notebook from Linux is shown in Fig. 1.1.

You should create one Jupyter Notebook per lab assignment, by choosing the New (Python 3) option in your client. Change the name of your notebook to something that clearly identifies the lab. Start each lab with comments (starting with "#" symbol) indicating the title of the lab, then your name followed by your lab partners. See the first cell of Fig. 1.2 for an example. This first cell is also a good place to issue the ipython "magic function":

```
%pylab inline
```

which will setup the notebook for inline plots and load the numpy and matplotlib libraries for you.

Each assignment will consist of a number of steps, clearly numbered like this one, you first step:

Jupyter Notebook 1.1. Print "hello world" using the python print command.

To keep your notebook clear, label cells (such as this one) with a comment for the assignment step number, as in the second cell of Fig. 1.2. You only need to label one cell if the assignment is fullfilled across several cells.

Jupyter Notebook checkpoints your work automatically. You should be able to see your notebook saved in the working directory where you started, as in Fig. 1.3. Notice that while the notebook file is ASCII text, it is not a human readable format. The Jupyter software is needed to render the notebook in a human readable way. To make your grader's life easier, you will be submitting PDF versions of your notebook, once all of the tasks are completed and the output is visible. There are several ways to make a PDF file from your notebook, but the most reliable is to use the "Print Preview" option to view the notebook as a PDF file within your browser, then use the print feature of your browser to print the page as a PDF file. Try this now, and make sure you can create a legible PDF file, but do not submit it to the course site, as you still have more to do. Always keep your python notebook file (ipynb) even after you submit the assignment. If you have problems, you can reproduce a PDF file from the notebook file, but it is tedious to reproduce your notebook from PDF. If you have problems producing the PDF file, you can submit the "ipynb" file as a temporary work-around, but work with your TA to sort out the problem as quickly as possible.

1.3 Scientific Python lecture notes

A link to the Scientific Python lecture notes is available on the course website. These lecture notes are a community-based effort which is well-maintained and constantly improving. This section contains example problems designed to reinforce the key concepts from the first chapter of the Scientific Python Lecture Notes (SPLN). Nearly all of Chapter 1 is useful, but these examples single out the most essential sections for getting started with compuation in this course.

Read through SPLN 1.2.2 and complete the following exercises in your notebook:

Jupyter Notebook 1.2. Make some of your own simple calculations demonstrating the use of integers, floats, and Boolean variables. Make your own list of strings, and demonstrate a few list operations.

Jupyter Notebook 1.3. Try to predict the output of this code snippet:

```
a=3
b=a
a=2 #update
print(b)
```

Run the code and check the output. Are integers mutable or immutable? At the line marked #update, is a being changed by assignment or modification in place?

Jupyter Notebook 1.4. Try to predict the output of this code snippet:

```
a=[3]
b=a
a[0]=2 #update
print(b[0])
```

Run the code and check the output. What type of data format is a? Is that a mutable data type? At the line marked #update, is a being changed by assignment or by modification in place? Change that line so as to do the opposite. Does the program output change?

Read through SPLN 1.2.3 and complete the following exercises in your notebook:

Jupyter Notebook 1.5. Use a for loop to print the first 10 powers of 3: 1,3,9,27,...

Jupyter Notebook 1.6. Use a while loop to find the first number n for which the sum $1^2 + 2^2 + 3^2 + ... + n^2$ exceeds 1000.

Jupyter Notebook 1.7. Use a for loop to calculate $n!$ by simply multiplying every value from n to 1.

Jupyter Notebook 1.8. Write a program to calculate and list the first 10 prime numbers after 100. Use only basic python features such as while loops, for loops, conditionals, assignment(=), equality (==), multiplication (*), and the integer divide operation (//). Do not use the mod operation (%).

Read through SPLN 1.4.1.1-3, 1.4.1.5 and complete the following problems:

Jupyter Notebook 1.9. Use the np.arange to define a 1-D numpy array of integers from 10 to 30 (inclusive). Uses slices to print (a) every third element (10,13,..), (b) the last five elements (26,27,...,30), and (c) every other element in reverse order (30, 28, 26, ..., 10).

Jupyter Notebook 1.10. Define variables a and b as numpy arrays of floats each of length three. Write a code snippet to calculate and print the vector dot product of $a \cdot b$ using an explicit for loop, i.e. loop over indexes 0,1,2 and increment a sum by the product. This is how you write this program in a language such a C. Now use the python * and np.sum operation to compute the dot product in a single line, with no explicit for loop. The elimination of tedious explicit for loops is why programming in Scientific Python is much more fun than any other common language, once you get used to the idea.

The Scientific Python Lecture Notes are an excellent resource. These examples and sections are a good starting point for the work we will be doing in this course, but remember that there is much more available for you to refer to in the future!

1.4 Submitting your assignment

Before submitting, take some time to clean up your assignments to remove anything superfluous and place the exercises in the correct order. You can also add comments as needed to make your work clear. You can use the Cell → All Output → Clear and Cell → Run All commands to make sure that all your output is up to date with the cell source.

When you are satisfied with your work, print the PDF file as described earlier and submit it to the course website.

Chapter 2

Introduction to Plotting

2.1 Introduction

This lab will introduce calculations and plotting techniques using numpy arrays within Scientific Python. See the previous lab for details on how to produce and submit an ipython notebook as a PDF file.

2.2 Plotting discrete data and continuous functions

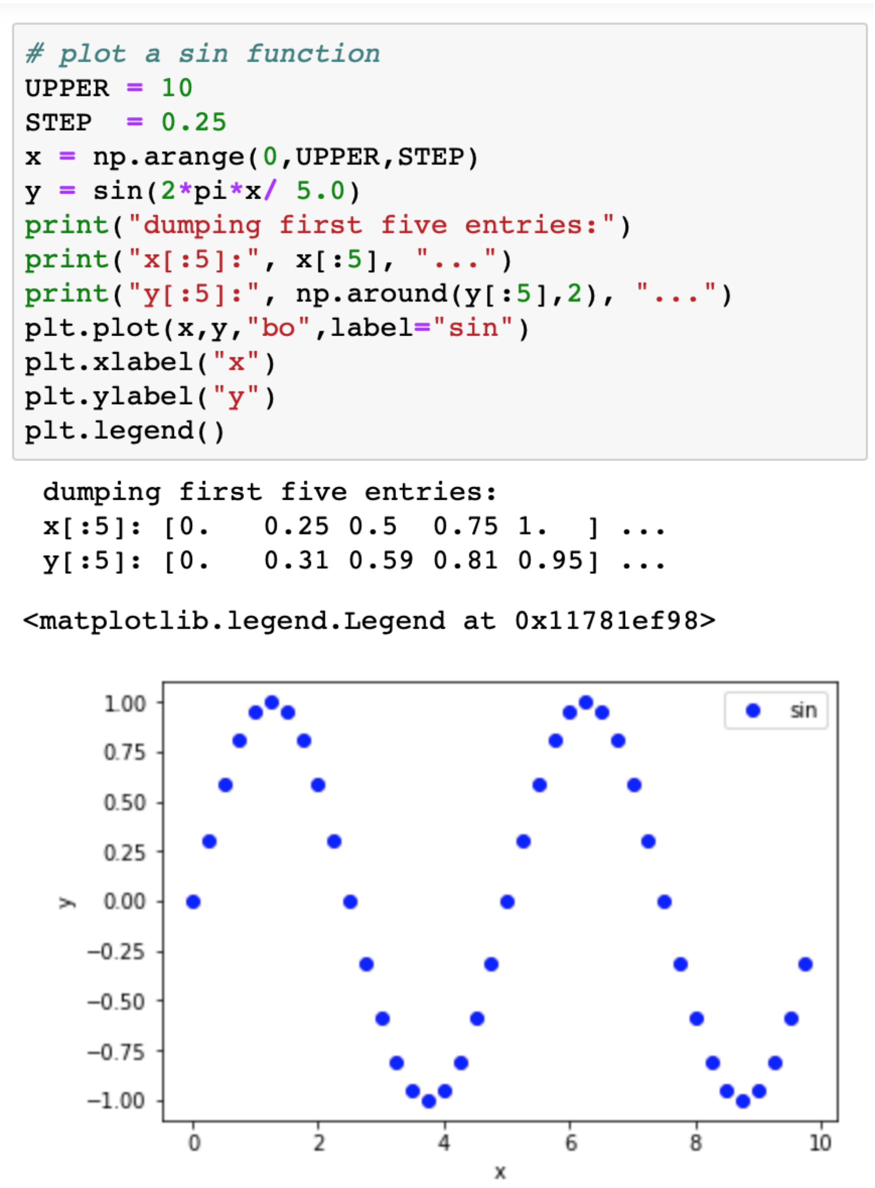
Consider the Jupyter Notebook example in Fig. 2.1 which plots a sine function sampled at discrete values. Note the following key features, which you will use repeatedly today and in future labs:

- 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.ylabel`.
- 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 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 2.1: Sine function sampled at discrete values.

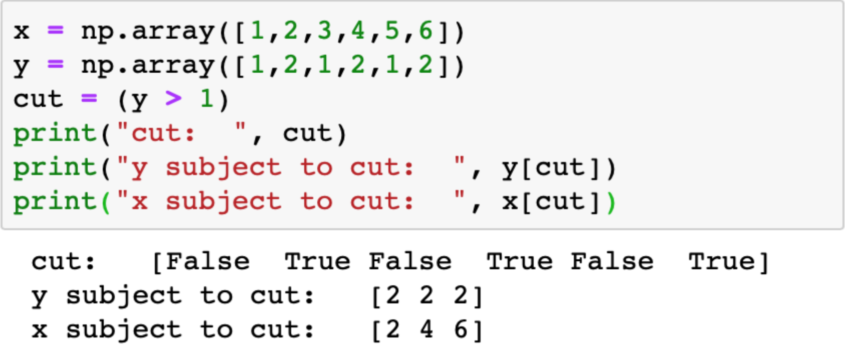
Jupyter Notebook 2.1.

Plot the sinc function with the following requirements:

- Plot in the range $x = (-5, 5)$.
- Plot discrete samples with a step size of 0.5 using blue circles.
- On the same axis, plot the corresponding smooth function using a red solid line. The smooth function should use much finer binning to approximate a continuous curve.
- Add appropriate axis labels.
- Add a legend for the "discrete" and the "smooth" function.

Use the numpy sinc function, e.g. `y = np.sinc(x)`.

2.3 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 2.2: 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. 2.2. 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 2.1 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 2.1: 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])
```

Jupyter Notebook 2.2.

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. **This is a sign-off point for the lab.**

2.4 The Logistics Map

The logistics map is the recurrence relation

$$x_{n+1} = r\, x_n\, (1 - x_n)$$

with the variable x between 0 and 1. The variable x can be thought to represent the ratio of a population to its maximum possible value. The population increases due to birth and decreases due to starvation as the population approaches it’s maximum value (x near 1). This leads to the non-linear relationship that defines the logistic map. The mapping keeps the variable x between 0 and 1 as long as the parameter r is in the range $[0, 4]$.

The logistics map is frequently encountered as a simple example of a chaotic system emerging from a simple non-linear system. If we consider the long term behavior of the population x as a function of the parameter r, as shown in Fig. 2.3, we see that for values of r less than 3 the population approaches a single fixed value. At the value $r = 3$ the non-linear system exhibits bifurcation with the population oscillating between two values. As r increase, further bifurcations occur at an ever increasing rate until the systems exhibits chaotic behavior alternating with occasional returns to stable oscillations.

The long term behavior of the logistics map can be easily modeled in Scientific Python. A start is shown in Fig. 2.4 where you should understand:

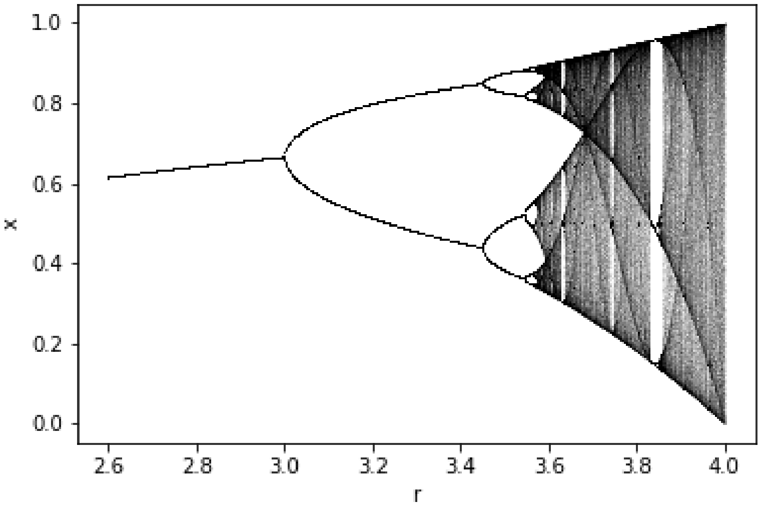


Figure 2.3: Long term behavior of the logistics map.

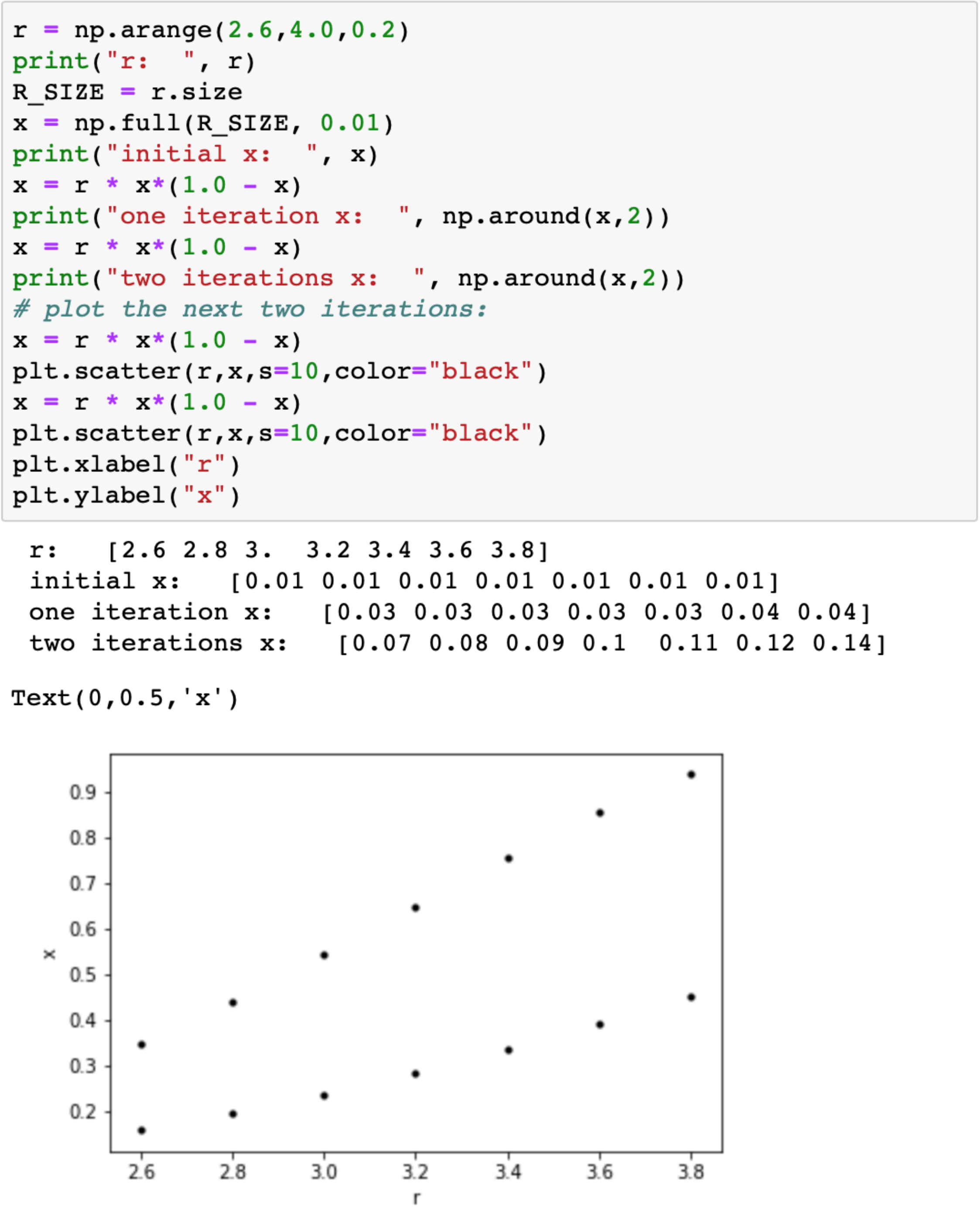
- An array of r values is defined.
- An array of x values of the same size as r is defined and initialized to an arbitrary non-zero value (0.01).
- Two example iterations of the logistic map are applied.
- The next two iterations of the values of x are plotted as function of r on the same plot.

Jupyter Notebook 2.3.

Reproduce the figure in Fig. 2.3 by doing the following:

- Define two global variables `ITER = 10` and `PLOT = 5`.
- Apply the logistics map `ITER` times by using a for loop.
- Apply the logistics map an additional `PLOT` times, plotting the values of x as a function of r, as in the example, each time.

You'll observe the long term behavior by increasing the value of `ITER` to a large value, such as 10,000. You'll see the full dependence on r by decreasing the step size in the initialization of the numpy array r to something like 0.001. You'll observe the chaotic behavior by increasing the value of `PLOT` to 100 or even 1000 iterations. To make a prettier plot using finer points (once you have a large number of points) you can reduce the size by adjusting the `s=10` parameter in the call to `plt.scatter` to something like `s=0.0001`.



```
r = np.arange(2.6,4.0,0.2)
print("r:  ", r)
R_SIZE = r.size
x = np.full(R_SIZE, 0.01)
print("initial x:  ", x)
x = r * x*(1.0 - x)
print("one iteration x:  ", np.around(x,2))
x = r * x*(1.0 - x)
print("two iterations x:  ", np.around(x,2))
# plot the next two iterations:
x = r * x*(1.0 - x)
plt.scatter(r,x,s=10,color="black")
x = r * x*(1.0 - x)
plt.scatter(r,x,s=10,color="black")
plt.xlabel("r")
plt.ylabel("x")
```

```
r:   [2.6 2.8 3.  3.2 3.4 3.6 3.8]
initial x:   [0.01 0.01 0.01 0.01 0.01 0.01 0.01]
one iteration x:   [0.03 0.03 0.03 0.03 0.03 0.04 0.04]
two iterations x:   [0.07 0.08 0.09 0.1  0.11 0.12 0.14]
```

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

Figure 2.4: Modeling the logistics map.

Chapter 3

The Monte Carlo Method

3.1 Introduction

This lab, which will take two lab sessions to complete, introduces the Monte Carlo method, an approach to solving a wide range of problems by repeatedly drawing random numbers from a probability distribution. You will produce a sequence of pseudorandom numbers. You will use a histogram to directly compare values of random variables to a probability distribution function. With these preliminaries in hand, you will explore several widely used Monte Calro techniques: Monte Carlo integration, the rejection method, and the transform method. You will finish by looking at the evolution of entropy during diffusion, as modeled by a random walk.

3.2 Generating random numbers

The Monte Carlo method relies on the generation of random numbers, so we will start there. The numbers we generate using computers are actually "pseudorandom" numbers, because they are deterministically obtained from an algorithm. However, the algorithm is choosen so that the numbers appear random for practical purposes. This is no small concern. Much of the computational work in the early 1970's had to be redone because of the widespread use of a deeply flawed pseudorandom number generator called RANDU.

In this section, you will generate a pseudorandom number sequence using the linear congruential method. This sequence is determined iteratively from the simple relationship:

$$I_{n+1} = (a * I_n + c) \mod M$$

Recall that $x \mod y$ (coded as `x % y` in python) is the remainder after integer division $x//y$. Each I_n is called a seed, and the initial seed I_0 must be provided e.g. by the user. Notice that the seeds are all integers in the range from 0 to $(M-1)$. If we wish to convert these seeds into a random variable x in the range from 0 to L, we simply use $x_n = L * I_n/M$. As long as M is much larger than L, x is approximately continous.

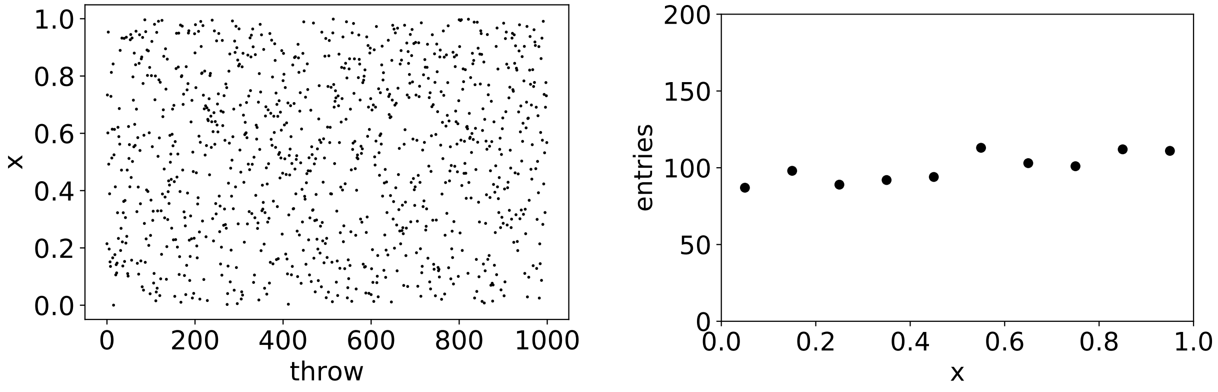
The algorithm works because the product $a * I_n$ is generally many times larger than M, so the remainder is effectively a uniform random number. The effectiveness of this algorithm is highly dependend on the choice of a,c, and M. Choose poorly and you get RANDU. Choose wisely and you get the highly regarded algorithm of Park and Miller. We will do the latter and use $a = 7^5$,

$c = 0$, and $M = 2^{31} - 1$.

Jupyter Notebook 3.1.

Generate a sequence of ten uniform random variables in the range $[0, 1]$ from the Park-Miller sequence, using an initial seed of one. Check your code by testing that the generator returns a **seed** of 1043618065 after 10000 calls. Change the initial seed to a value of your choice and report the first 10 random values. If you like, round to two decimal places using `np.around` to tidy up your output.

3.3 Visualizing distributions



(a) (b)

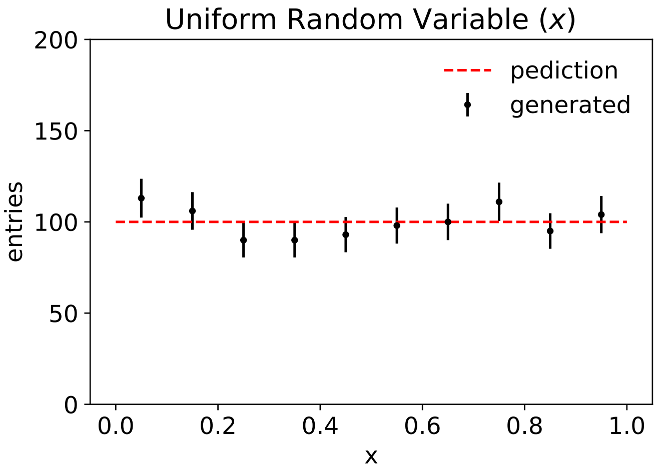
Figure 3.1: The (a) x value of uniform random throws versus throw number and (b) corresponding histogram.

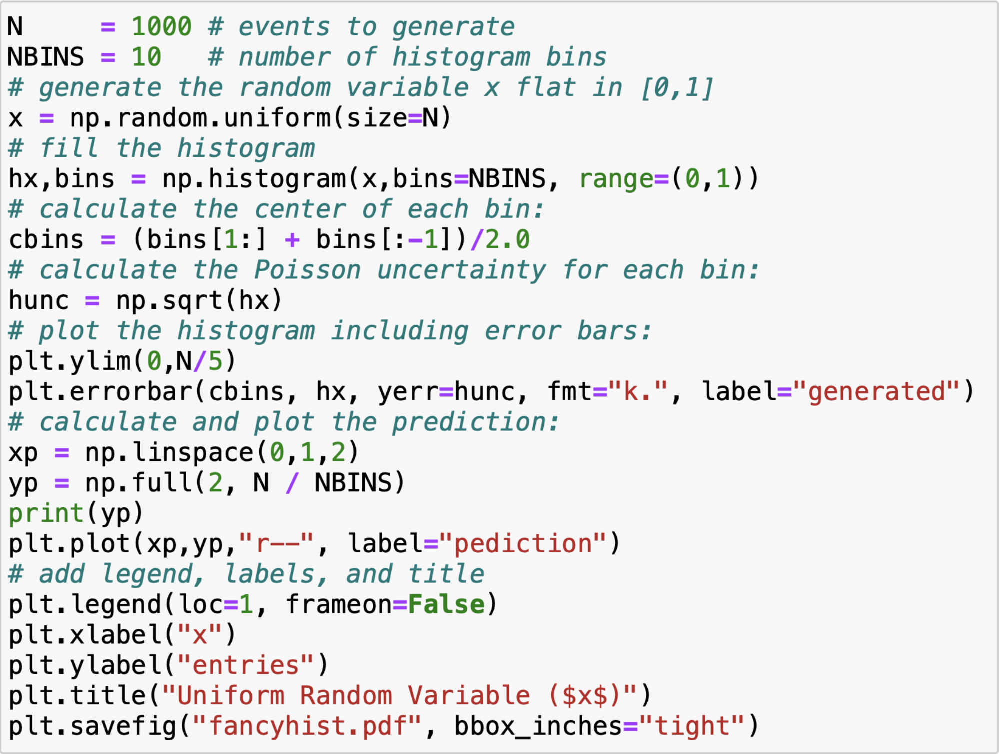
How can we verify that our Park-Miller random number generator produces uniform random numbers in the range 0 to 1? The associated PDF is just $p(x) = 1$, which we know how to plot, but how can we compare this function to a sequence of numbers like $[0.21, 0.85, 0.33, ...]$? An initial attempt might look like Fig. 3.1a, where we have simply plotted the x value of each throw versus the number of the throw. Unfortunately, this plot isn't particularly helpful. If we zoomed in, we could determine from the plot the x value associated with each throw. This is simply too much information.

For interpreting a list of values as a distribution, there is only one tool of choice: the histogram. To histogram our data, we divide the entire range $[0, 1]$ into smaller ranges called *bins*. Let's start with 10 bins as an example. In this case, the first bin covers the range from 0 to 0.1, or more precisely, the half-open interval $[0, 0.1)$ which includes 0 and 0.099, but not 0.1. The second bin would have range $[0.1, 0.2)$, the third bin would have range $[0.2, 0.3)$ and so on, up to the last bin which would cover $[0.9, 1.0]$. To *fill* a histogram, you count the number of values that fall within the range of each bin. So in our example, the value 0.21 would add one to the count for the third bin, which has range $[0.2, 0.3)$. After filling, the histogram consists of a count associated with each bin range.

Fig. 3.1b shows a histogram filled with 1000 random throws drawn from a uniform random number generator. While we can now see the shape of the distribution, we still don't have quite enough information to answer the question, is this flat? It is certainly not perfectly flat!

The first feature we will need to add to the plot is the inclusion of *error bars* to indicate the statistical uncertainty in our histogram values. Each histogram contains a *count* n which should follow a Poisson distribution with variance $\sigma^2 = n$. Error bars are traditionally drawn with the size σ, which in this case is $\sigma = \sqrt{n}$. So when drawing a histogram, the uncertainty in each bin is just the square root of the histogram value. That is the beauty of Poisson statistics! If we have a count, we know the statistical uncertainty.





```
N     = 1000 # events to generate
NBINS = 10   # number of histogram bins
# generate the random variable x flat in [0,1]
x = np.random.uniform(size=N)
# fill the histogram
hx,bins = np.histogram(x,bins=NBINS, range=(0,1))
# calculate the center of each bin:
cbins = (bins[1:] + bins[:-1])/2.0
# calculate the Poisson uncertainty for each bin:
hunc = np.sqrt(hx)
# plot the histogram including error bars:
plt.ylim(0,N/5)
plt.errorbar(cbins, hx, yerr=hunc, fmt="k.", label="generated")
# calculate and plot the prediction:
xp = np.linspace(0,1,2)
yp = np.full(2, N / NBINS)
print(yp)
plt.plot(xp,yp,"r--", label="pediction")
# add legend, labels, and title
plt.legend(loc=1, frameon=False)
plt.xlabel("x")
plt.ylabel("entries")
plt.title("Uniform Random Variable ($x$)")
plt.savefig("fancyhist.pdf", bbox_inches="tight")
```

Figure 3.2: Histogram of data drawn from a flat distribution compared to prediction, with the code used to produce the plot.

We'll also want to add the prediction to the plot, assuming a flat distribution for the contents of each bin. In this case, we generated N events and we have N_{BINS} histogram bins, which should therefore each contain an equal share: N/N_{BINS}. The resulting histogram, along with the code used to generate it, is shown in Fig. 3.2. With the prediction and errorbars included in the plot, one can now see that these generated values are indeed quite consistent with a flat prediction. All of the bins are within nearly one-sigma. With this number of bins, it is not uncommon to see a two-sigma descrepancy.

You will produce many histograms in this class, so you will need to (eventually) understand every single line in this example code. Take the time to read through the documentation for the key functions like `np.histogram` and `np.random.uniform`, available on the web (see numpy.org or just search "np.histrogram python"). A big part of learning to program effectively, is learning how to read and understand software documentation correctly and efficiently.

There are a few important features to notice:

- The x-values are contained in an `np.array` filled with uniform random variables generated by calling the `np.random.uniform` function.
- The function `np.histogram` is used to calculate a histogram from these x values. The call requests `NBINS=10` histogram bins, in range $[0, 1]$. Don't confuse the python tuple `(0,1)` used to indicate this range as indicating an open interval... often the computing language differs significantly from math notation, as is the case here!
- The `np.histogram` function returns two items we need: a `np.array` containing the count for each bin (`hx`) and a `np.array` of bin edges (`bins`)
- We want to plot the count over the center of each bin, not one of the edges, so we calculate the quantity `cbins` which is an `np.array` containing the center of each bin. You'll use this trick a lot, so make sure you understand what it is doing!
- The uncertainty on each bin `hunc` is calculated as the square root of the bin counts `hx`.
- We use the somewhat poorly named `np.errorbar` function to plot **both** the histogram central value `and` the errorbar in each bin.
- We draw the prediction as a straight line defined by two points defined by `xp` and `yp`.

Jupyter Notebook 3.2.

Modify the example code to generate a histogram for uniform random numbers generated from your Park-Miller sequence instead of `np.random.uniform`. Increase the number of events to `N=10000`. Increase the number of bins to `NBINS=20`. Does your code appear to produce uniform random numbers?

To answer a question in your notebook, simply add a cell and answer the question as a comment (each line starting with `#`).

3.4 Calculating the value of π

Hopefully 2021 will see the return of parties, so let's start by examing a surefire way to be the life of the party: determining the constant π by throwing toothpicks! The procedure is simple: you cut a peice of paper to a width of four toothpicks, then draw two vertical lines separated by the width of two tooth picks. Take turns tossing toothpicks, as in Fig. 3.3.

From the geometry of the setup, it can be shown that the probability that a toothpick which is entirely on the paper also crosses a line is given by $1/\pi$. Therefore, one can measure π by counting

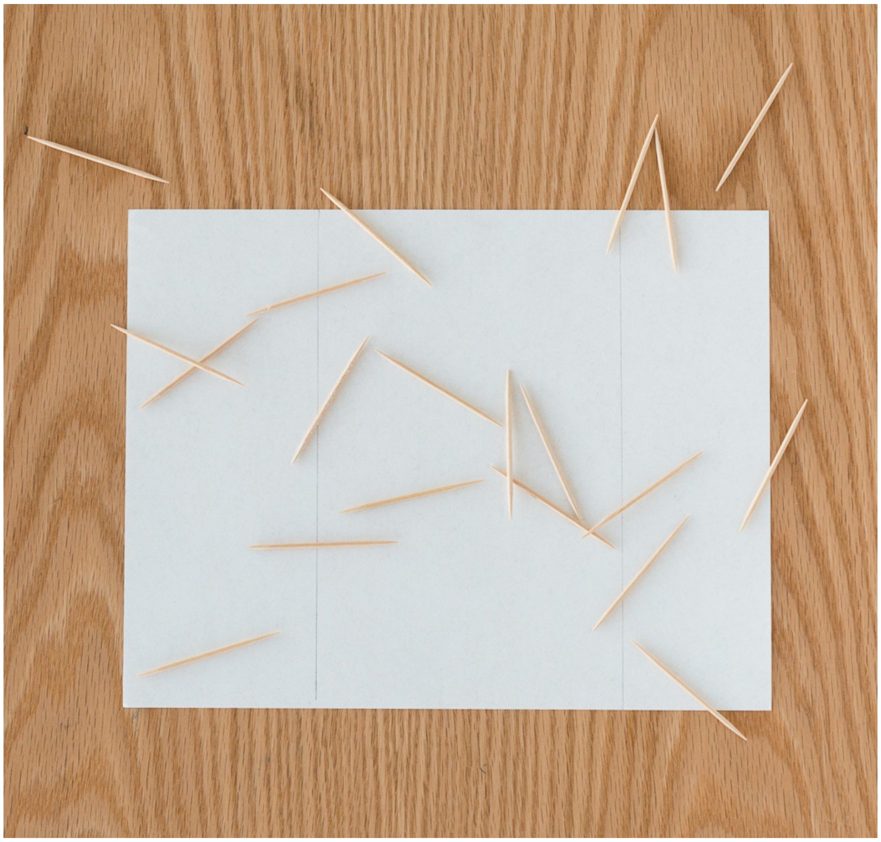


Figure 3.3: Determining π by throwing toothpicks.

the total number of toothpicks that landed entirely on the page and dividing by the number of those toothpicks that crossed a line. This is, in essence, the Monte Carlo method.

An easier Monte Carlo method to implement computationally is shown in Fig. 3.4 along with the code used to generate the plot. The idea is to throw points uniformly in the unit square of area 1. Much like in the toothpick example, the value of π can be determined by counting the number of generated points that also landed within the unit circle.

The key features of the example code are:

- The x and y values are each contained in an `np.array` filled with uniform random variables in $[0, 1]$ by the `np.random.uniform` function.
- A mask `inside` is created to indicate which points are inside the circle. Recall that the mask is an `np.array` of True or False values, with the same length as the x and y arrays. For example `x[inside]` is an np.array containing just the subset of `x` which are inside the circle.

Jupyter Notebook 3.3.

Starting from the example code, determine the numerical value of π using the Monte Carlo method. The easiest way to obtain the count you need is to apply the function `np.sum` to an appropriate mask. When counting a mask, each True is treated as a one, and each False is treated as zero. Work out the relationship between π and the fraction of events in the unit circle, and use your count to numerically determine the value of π. Increase the number of generated events and confirm that your calculated value of π approaches the known value.

Jupyter Notebook 3.4.

This is an example of a binomial process, because points are either inside or outside the circle. So we expect the number of events in the circle to follow the binomial distribution with $\sigma^2 = n\epsilon(1-\epsilon)$. In this case, n is the total number of generated events and ϵ is the fraction that fall inside the unit circle. The statistical uncertainty on your measured value of π works out to be:

$$\sigma_\pi = \sqrt{\frac{\pi\,(4-\pi)}{n}}$$

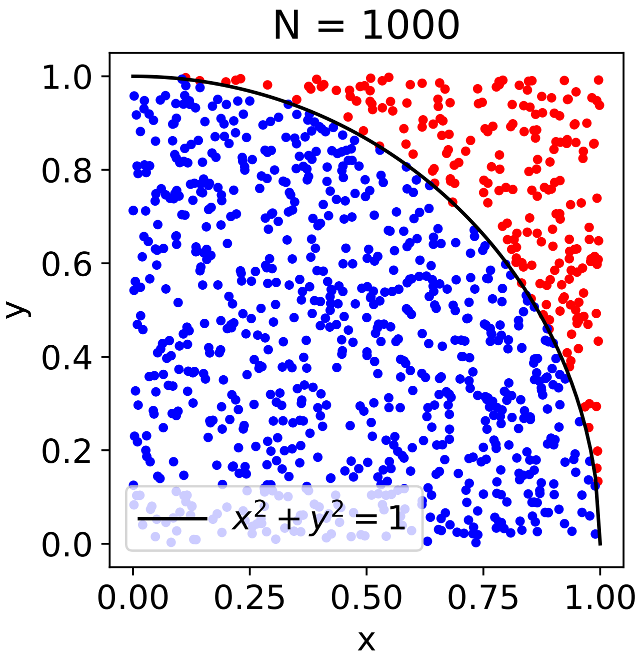
where n is the number of generated events. Does your measured value of π agree with the known value within your statistical uncertainty?

3.5 Monte Carlo integration

The Monte Carlo method can also be used to numerically integrate a function. Monte Carlo integration methods generally only outperform deterministic methods when the number of dimensions is large, but we can illustrate the method most easily in one dimension. In this section, you'll use the Monte Carlo method to perform the integral:

$$\int_0^\pi \sin^2\theta\, d\theta$$

To do so, you should make a copy of your solution from the previous section and modify it in the following manner:



```
N = 1000 # number of throws:
# throw x and y variables uniform in [0,1]
x   = np.random.uniform(size=N)
y   = np.random.uniform(size=N)
# calculate the radius squared and masks
#  for the points inside and outside the circle
rsq = x**2 + y**2
inside  = (rsq<=1)
outside = np.logical_not(inside)
# set aspect ratio so that a circle looks like a circle:
plt.axes().set_aspect('equal')
# plot outside points as red, inside as blue
plt.plot(x[outside],y[outside],"r.")
plt.plot(x[inside],y[inside],"b.")
# draw a curve for the circle:
xfin   = np.linspace(0,1,100)
yfin   = sqrt(1 - xfin * xfin)
plt.plot(xfin, yfin, "k-",label="$x^2+y^2=1$")
# add labels, title, and legend
plt.xlabel("x")
plt.ylabel("y")
plt.title("N = "+str(N))
plt.legend(loc=3)
plt.savefig("pimc.pdf", bbox_inches="tight")
```

Figure 3.4: Monte Carlo Determination π .

- Instead of thowing x in $[0, 1]$, throw θ in $[0, \pi]$. This means the area of the rectangle A is now π instead of 1.
- Count the number of throws that land below the integral $y < \sin^2\theta$.
- Determine the area under the curve as the fraction of the throws under the curve times the total area of the rectangle A.
- The statistical uncertainty in this case is $\pi/(2\sqrt{n})$ where n is the number of generated events.

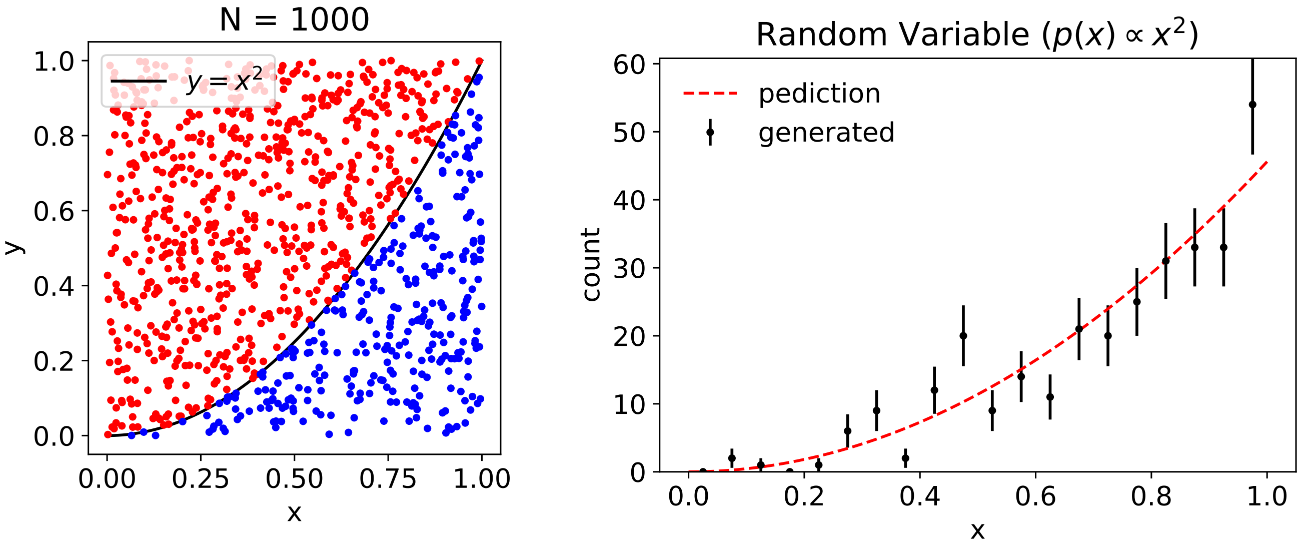
Jupyter Notebook 3.5.

Use the Monte Carlo method to calculate the integral:

$$\int_0^\pi \sin^2\theta\, d\theta$$

Make a plot similar to that of Fig. 3.4 showing the thows above the curve in red and below the curve in blue. Calculate the integral and statistical uncertainty and compare it to the value you obtain analytically.

3.6 The Rejection method



(a)

(b)

Figure 3.5: Monte Carlo rejection method applied to $p(x) \propto x^2$. Uniformly generated points (a) are rejected (red) if they are above the PDF, and the x values of points below the PDF (blue) are selected. A histogram (b) of the selected x values shows that they follow the PDF.

We now know how to generate uniform random numbers, but suppose we need a random variable thrown according to a non-uniform probability distribution $p(x)$? Fig. 3.5 demonstrates one approach, which closely follows the procedure for numerical integration using the Monte Carlo technique.

The rejection method produces random variables in a range from 0 to L according to any desired PDF $p(x)$. Start by finding a value Y which is at least as large as the maximum value of $p(x)$ for x in $[0, L]$. Then:

- Throw x as a uniform random variable in range $[0, L]$.
- Throw y as a uniform random variable in range $[0, Y]$.
- If $y > p(x)$ reject the x value and start over, otherwise, use the x value as one throw.

Repeat these steps as necessary until a sufficent number of x values have been selected.

The rejection method works because the probability of an x value being selected is, by construction, proportional to $p(x)$. Since the x values were initially chosen from a flat distribution, the selected x values will follow the $p(x)$ distribution. You can visualize this in Fig. 3.5 which leaves very little doubt that the x values of the blue points will follow the PDF. Notice that it isn't even necessary for $p(x)$ to be normalized for this procedure to work: any function proportional to the PDF of interest will do.

To produce a smooth function such as the quadratic prediction of Fig. 3.5b, make sure you use plenty of x values (around 100 at least), via `np.arange` or `np.linspace`, just as you did in the Plotting lab. When comparing a PDF to histogrammed data (as you will do for the second plot below) you will need to normalize it appropriately. The number of throws we expect to find in a bin with edges at a and b is given by

$$N \cdot \int_a^b p(x)\, dx \; = N \cdot p(x^*) \cdot (b - a)$$

The integral is simply the probability that one throw ends up in the range, which we scale by the total number of throws N. The equality holds for at least one x^* in the range $[a, b]$ and $(b - a)$ is simply the bin size. Therefore, we can formulate a prediction from a normalized PDF $p(x)$ to data from N throws used to fill a histogram with bin sizes Δx as the smooth function resulting from:

$$N \cdot p(x) \cdot \Delta x$$

This is a technique we will use over and over again, so make sure you understand it!

Jupyter Notebook 3.6.

Use the rejection method to generate random numbers in the region from [0,1] that follow a distribution $p(x) \propto x^2$. You'll do the following:

- Note that there is no need to normalize the PDF when using the rejection method, so use $p(x) = x^2$.
- In our x range $[0, 1]$, $p(x)$ has maximum value at $x = 1$ so set $Y = p(1) = 1^2 = 1$.
- Throw x as a uniform random variable in the range $[0, 1]$.
- As $Y = 1$, throw y as a uniform random variable in range $[0, 1]$.
- If $y > x^2$ reject the x value and try a new set of x and y values, otherwise, use the x value as one throw.

Throw 1000 (unselected) x values, and produce a plot like that of Fig. 3.5a showing your selected points in blue, your rejected points in red, and the selection function ($p(x) = x^2$).

Jupyter Notebook 3.7.

Increase the number of (unselected) x values thrown to 10,000. Count the number N of selected x values. Generate a plot like that of Fig. 3.5b comparing the distribution of your selected x values to the prediction, which in this case is given by:

$$N \cdot 3x^2 \cdot \Delta x$$

Be careful to use the number of selected x values for N, not the total thrown (10000) before rejection.

3.7 The transformation method

Suppose that you need to throw random variables according to an exponential distribution $p(x) = \exp(-x)$. This PDF is defined for $[0, +\infty)$ and properly normalized across this range as you can verify:

$$\int_0^{+\infty} \exp(-x)\, dx = 1$$

The first problem is that we can only generate uniform random variables up to a finite value L, not $+\infty$. But let's suppose we are willing to work aound this by simply cutting off the PDF at some large value, like say we won't produce values with $x > 100$.

With this change, the rejection method will work in principle. But it still has a major shortcoming. Since $p(x)$ has a maximum value of 1, and x ranges from 0 to 100, the rectangle we will be filling with uniform random points has area 100. But our PDF, even when integrated to $+\infty$, only has area 1. So less than one out of every 100 points we throw will be selected. Perhaps we can live with this, but then what if we need to go out to $x = 1000000$. Now only one out of every million points will be selected. In many scenarios, the rejection method becomes too computationally inefficient to be of any practical value.

In these case, we can use the transformation method instead of the rejection method. The transformation method is premised on the fact that for *any* normalized PDF, we must have

$$p(x) \geq 0$$

everywhere and

$$\int_{-\infty}^{+\infty} p(x)\; dx = 1$$

as long as we take care to set $p(x) = 0$ outside our range for x. It follows from these properties that for any value of y in the range $[0, 1]$ there is a unique largest x value for which:

$$\int_{-\infty}^{x} p(x)\; dx = y \tag{3.1}$$

From the fundamental theorem of calculus, we see that:

$$dy = p(x)\, dx$$

If the variables y are drawn from a uniform distribution with PDF $q(y) = 1$, then we see that:

$$\int_{y_1}^{y_2} q(y)\, dy = \int_{x_1}^{x_2} p(x)\, dx.$$

for x_i and y_i related by Eqn. 3.1. This shows that while y is a uniform random variable ($q(y) = 1$), the corresponding x values will distributed according to the desired PDF $p(x)$.

That provides the mathematical justification for the transformation method, which starts by finding the inverse function $f^{-1}(y)$ for:

$$y = f(x) = \int_{-\infty}^{x} p(x)\, dx$$

Then the procedure is:

- Throw y as a uniform random variable in $[0, 1]$.
- Find $x = f^{-1}(y)$

The x values determined in this way will be drawn from the $p(x)$ distribution.

There is an intuitive explanation for why this works. The y value is essentially a fraction of the probability integrated by the PDF. In a region of x where $p(x)$ is relatively large, the integral is changing rapidly and so a large range of y values map to this region of x-values. In a region of x where $p(x)$ is relatively small, the integral is not changing rapidly and so a small range of y values map to this region of x-values.

Let's see how this applies to our exponential function. In this case we calculate:

$$y = f(x) = \int_{0}^{x} \exp(-x)\, dx = 1 - \exp(-x)$$

which we invert to find:

$$x = -\ln(1 - y)$$

To determine values of the random variable x, we follow this procedure:

- Throw a y value flat in [0,1]
- Caculate $x = -\ln(1 - y)$

Repeat to produce as many x values as needed. Notice that this procedure gives one usable x value for every random throw.

Jupyter Notebook 3.8.

Use the transformation method as described to generate 10,000 values of a random variable thrown from an exponential function. Produce a plot like that of of Fig. 3.5b comparing the distribution of your generated events to the prediction for $p(x) = \exp(-x)$. Remember to properly normalize your prediction based on the bin size and number of events thrown.

3.8 Particle diffusion

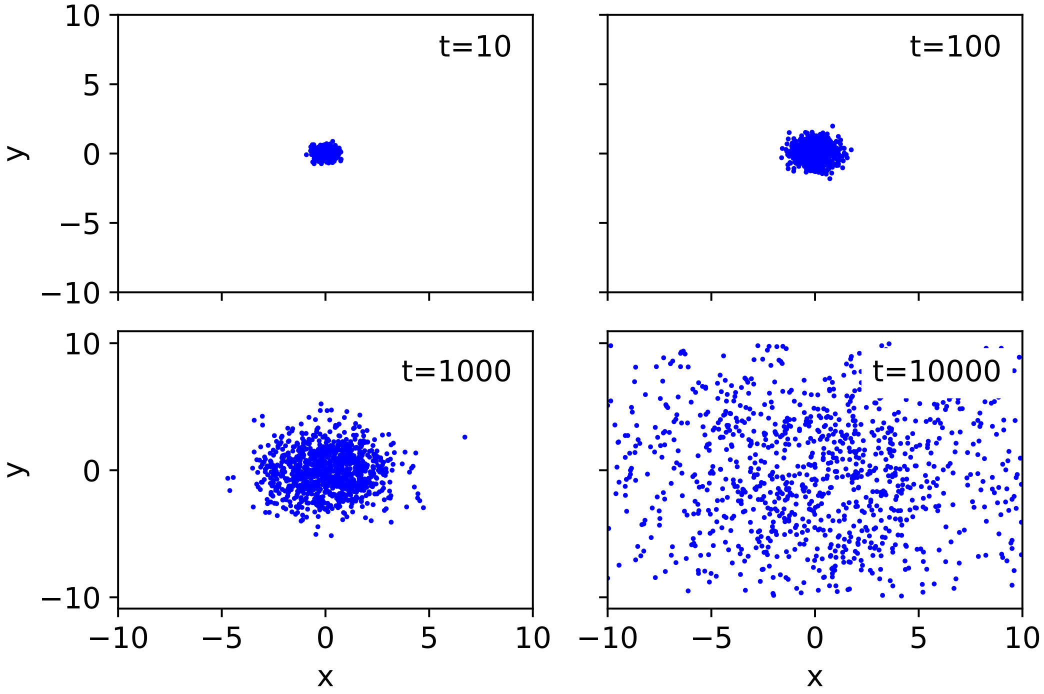


Figure 3.6: Simulation of the diffusion of a drop of particles at four different times.

In this section, we will model the diffusion of a drop of particles in a medium, as in Fig. 3.6, shown as snapshots at four different times. The starting point for the simulation, at $t = 0$ is shown in Fig. 3.7 along with the code used to produce it. The entire state of the system is contained in the arrays `x` and `y` which contain the x and y positions of each particle.

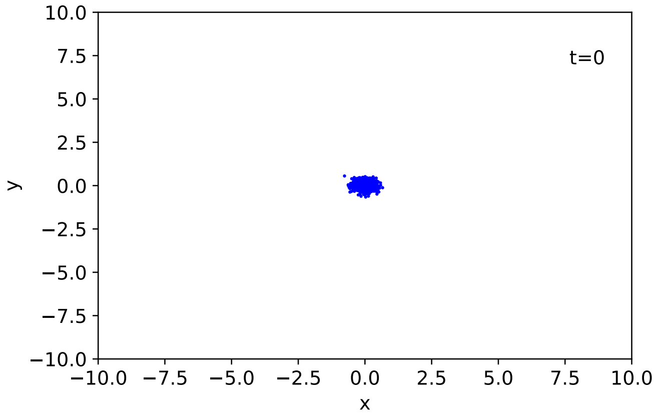
The diffusion process is modeled by a random walk. During each update (for one time step) the x and y values of each particle should be randomly increased or decreased by an amount `STEP=0.2` Any particles that would leave the boundaries of the region $[-L, L]$ as a result should be moved back into the region. The numpy functions `np.random.choice` and `np.clip` are useful here.

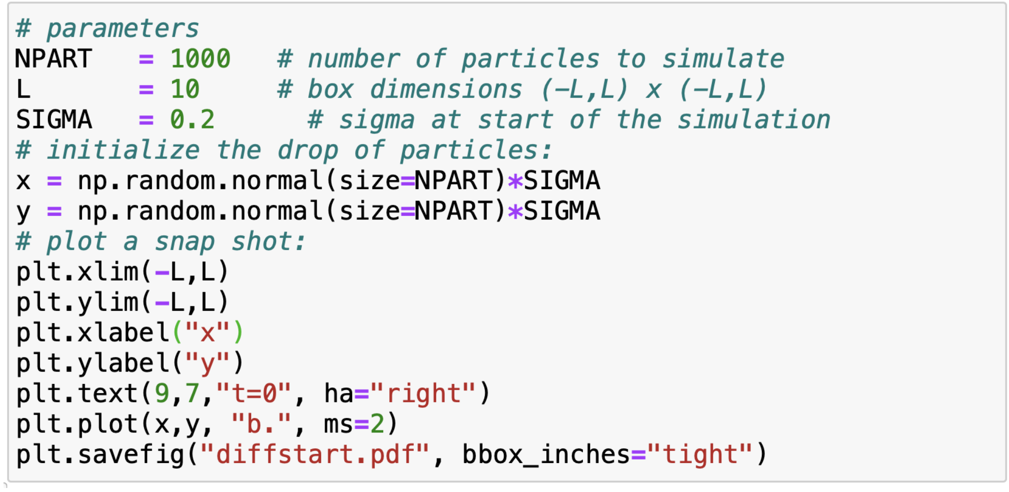
Despite the symmetry of the random walk, the system clearly evolves by diffusing outward over time. This can be seen as a consequence of the second law of thermodynamics. Calculating the entropy from the microscopic state of continuous particles is a bit tricky. The approach we will use is based on the Gibb's entropy. We divide the area into cells, and determine the fraction of the particles f_i in each cell i. We calculate the entropy as:

$$S = \sum_i f_i \ln f_i$$

A python function which calculates the entropy in this manner:

```
from scipy import stats
def entropy(x,y,l,sbins):
    h,xbins,ybins=np.histogram2d(x,y,bins=sbins,range=[[-l,l],[-l,l]])
    return stats.entropy(h.flatten())
```





```
# parameters
NPART    = 1000   # number of particles to simulate
L        = 10     # box dimensions (-L,L) x (-L,L)
SIGMA    = 0.2      # sigma at start of the simulation
# initialize the drop of particles:
x = np.random.normal(size=NPART)*SIGMA
y = np.random.normal(size=NPART)*SIGMA
# plot a snap shot:
plt.xlim(-L,L)
plt.ylim(-L,L)
plt.xlabel("x")
plt.ylabel("y")
plt.text(9,7,"t=0", ha="right")
plt.plot(x,y, "b.", ms=2)
plt.savefig("diffstart.pdf", bbox_inches="tight")
```

Figure 3.7: Snapshot of the simulation at the start, along with the code used to produce it.

The function takes as input parameters the position arrays `x` and `y`, the boundary distance `l` (set it to `L` and the number of bins in each dimension `sbins` (set it to 20). The function returns the entropy of the current state of the system described by `x` and `y`.

Jupyter Notebook 3.9.

Starting from the example code, implement a random walk to model the diffusion process, and plot four snap shops showing the evolution of the system.

Jupyter Notebook 3.10.

Calculate and record the entropy of the system as it evolves, and plot the entropy as a function of time.

Chapter 4

Limits of Distributions

4.1 Introduction

In this lab, we will use a Monte Carlo simulation to demonstrate the convergence of statistical distributions. We'll see that the binomial distribution approaches the Poisson distribution for a large number of trials, and that the Poisson distribution approaches the Gaussian distribution for a large values of the mean value λ. You will produce your own numerical demonstration of the Central Limit Theorem, by considering the average value of uniform random variables.

4.2 Poisson Limit of the Binomial Distribtion

We showed in lecture that the Binomial distribution for n independent trials with success probability ϵ approaches a Poisson distribution with mean value $\lambda = \epsilon \cdot n$ as $n \to \infty$. We will demonstrate this numerically by using a large but finite value for the number of trials n.

A simulation of a binomial process is demonstrated in Fig. 4.1 which will be a good start for the exercises. While working through the example, note a few key features:

- Note the distinction between the number of trials `NTRY` and the number of random throws `THROWS`. The simulation throws `THROWS` random variables m each of which represents the outcome of an binomial process with `NTRY` trials. It's easy to get these quantities confused!

- The simulated binomial outcomes are contained in the array `m` which is filled by calling the `np.random.binomial` function which is designed for this specific purpose. It produces an array of the specified size containing values randomly drawn from the binomial distribution with `n` trials of `p` success probability. Here the computing languages uses `p` for the parameter we call ϵ. That's life!

- We want to keep the parameter λ constant as we vary n so we set the success probability $\epsilon = \lambda/n$ with the line `EPS = LAMBDA / NTRY`.

- We calculate and plot a histogram in much the same way as in previous exercises. But notice that instead of center of each bin, we bplot using the left edge of each bin, determined as `lbins = bins[:-1]`. The reason for this choice is described in more detail below.

- We compare our simulated outcomes to the probability mass function (PMF) for the Binomial process evaluated at the integer values in the array `lbins`

- The prediction for each bin is normalized by multiplying the PMF (which represents a single throw) by the number of throws `THROWS`. There is no factor for the bin size in this normalization, because the bin size is 1.

Binning integer data: when plotting a histogram filled from continuous data, we plotted the count for each bin over the central x value for the bin. So the number of entries in the range $[0, 1)$ would be plotted over the value 0.5. This is a good choice for continuous data, because continuous data fills the entire range $[0, 1)$. But in this example, our random variables are integers. The only value that can increment the bin at $[0, 1)$ is the value (the range is open on the right, and does not include 1, which is left for the next bin!) In this case, it is misleading to plot a count of the value 0 over the center of the bin at 0.5. Instead, when plotting histograms of integer data with integer bins, we plot the data over the left edge of the bin. For example, the count of 0 values will be placed directly over 0.

Jupyter Notebook 4.1.

Adjust the example code so that the simulated binomial process is compared to the Poisson PMF instead of the binomial PMF. Look up the `np.stats.poisson.pmf` function. **Make sure that you leave the simulation unchanged:** the point of this exercise is to simulate a binomial process but compare it to a Poisson PDF. Leave the number of trials (and other parameters) unchanged. The agreement between the Binomial simulation and the Poisson PDF should not agree for six trials. Do your results here confirm that?

Jupyter Notebook 4.2.

Increase the number of trials in your simulated process to 100. Does your binomial process now resemble a Poisson PDF?

4.3 Gaussian Limit of the Poisson Distribtion

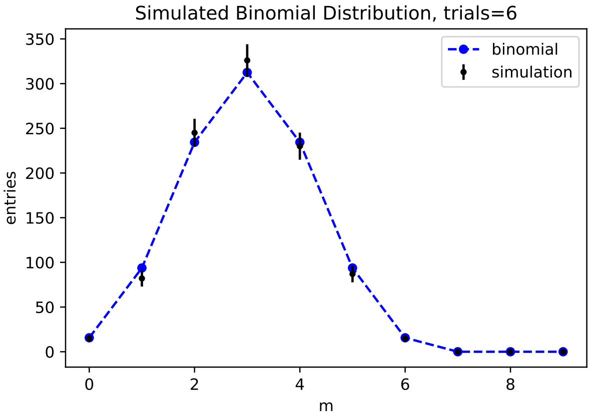
We discussed in lecture that in the limit $\lambda \to \infty$, the Poisson distribution will approach a Gaussian distribution with mean value λ and $\sigma = \sqrt{\lambda}$. We will demonstrate this result numerically using a simulated Poisson process and a large but finite value of λ.

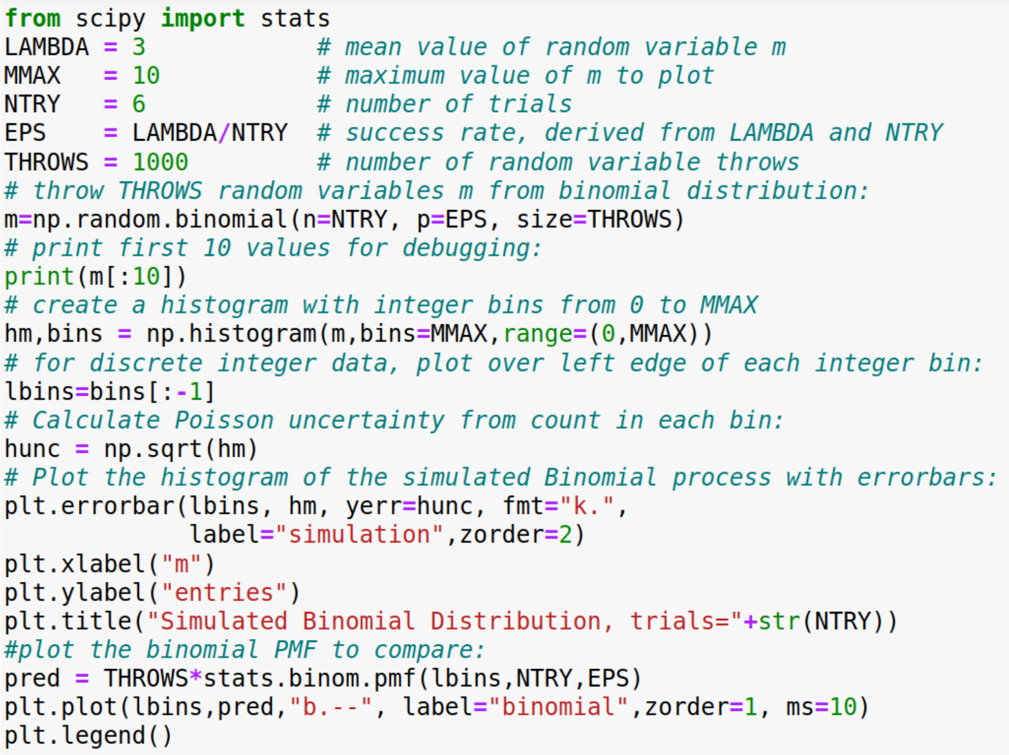
A simulation of a Poisson process is demonstrated in Fig. 4.2, which is similar in many respects to the previous example. Note a few features:

- The simulated Poisson outcomes are contained in the array `m` which is filled by calling the `np.random.poisson` function which is designed for this specific purpose. It produces an array of the specified size containing values drawn from the poisson distribution with mean `lam` (the parameter we call λ).

- The x-values used as locations to plot the histogram counts are contained in the array `ibins`. These are determined from the bin edges as the largest integer smaller than the center of the bin. This reduces to the left edge for integer bins (as in the previous example) but will continue to work as the bin size increases to include multiple integers.

- The outcomes are now being compared to the continuous Gaussian PDF. For this reason, a finely binned set of of x-values are contained in the `xf` array, filled using the `np.linspace` function. This will plot a nice smooth function, appropriate for a continuous function.





```
from scipy import stats
LAMBDA = 3            # mean value of random variable m
MMAX   = 10           # maximum value of m to plot
NTRY   = 6            # number of trials
EPS    = LAMBDA/NTRY  # success rate, derived from LAMBDA and NTRY
THROWS = 1000         # number of random variable throws
# throw THROWS random variables m from binomial distribution:
m=np.random.binomial(n=NTRY, p=EPS, size=THROWS)
# print first 10 values for debugging:
print(m[:10])
# create a histogram with integer bins from 0 to MMAX
hm,bins = np.histogram(m,bins=MMAX,range=(0,MMAX))
# for discrete integer data, plot over left edge of each integer bin:
lbins=bins[:-1]
# Calculate Poisson uncertainty from count in each bin:
hunc = np.sqrt(hm)
# Plot the histogram of the simulated Binomial process with errorbars:
plt.errorbar(lbins, hm, yerr=hunc, fmt="k.",
             label="simulation",zorder=2)
plt.xlabel("m")
plt.ylabel("entries")
plt.title("Simulated Binomial Distribution, trials="+str(NTRY))
#plot the binomial PMF to compare:
pred = THROWS*stats.binom.pmf(lbins,NTRY,EPS)
plt.plot(lbins,pred,"b.--", label="binomial",zorder=1, ms=10)
plt.legend()
```

Figure 4.1: Simulation of a binomial process.

- The prediction is normalized by multiplying the PDF (which represents a single throw) by the number of throws `THROWS` and the bin size. The bin size factor is needed because we are using a probability density function and the bin size will be different than one in the exercises.
- The parameters for the Gaussian PDF are poorly choosen in the example, on purpose.

Jupyter Notebook 4.3.

Lookup the function `scipy.status.norm.pdf`. Set the parameters `loc` and `scale` to values consistent with the simulation. Leave $\lambda = 3$ for now and plot your results. How does the Gaussian distribution compare to Poisson distribution at $\lambda = 3$?

Jupyter Notebook 4.4.

Now set $\lambda = 100$. Adjust the range for plotting to $[70, 130]$ and set the number of bins to 20. The Gaussian distribution distribution should agree closely with the Poisson distribution at this point. Is that what your simulation shows?

4.4 The Central Limit Theorem

The central limit theorem states that a properly normalized sum of independent random variables will tend toward the Gaussian distribution, even if the random variables are drawn from a different distribution. In this section, you will demonstrate this fact by showing that the average of `NSUM` uniform random variables does in fact follow a Gaussian distribution.

When solving these exercises, you should consider the following points:

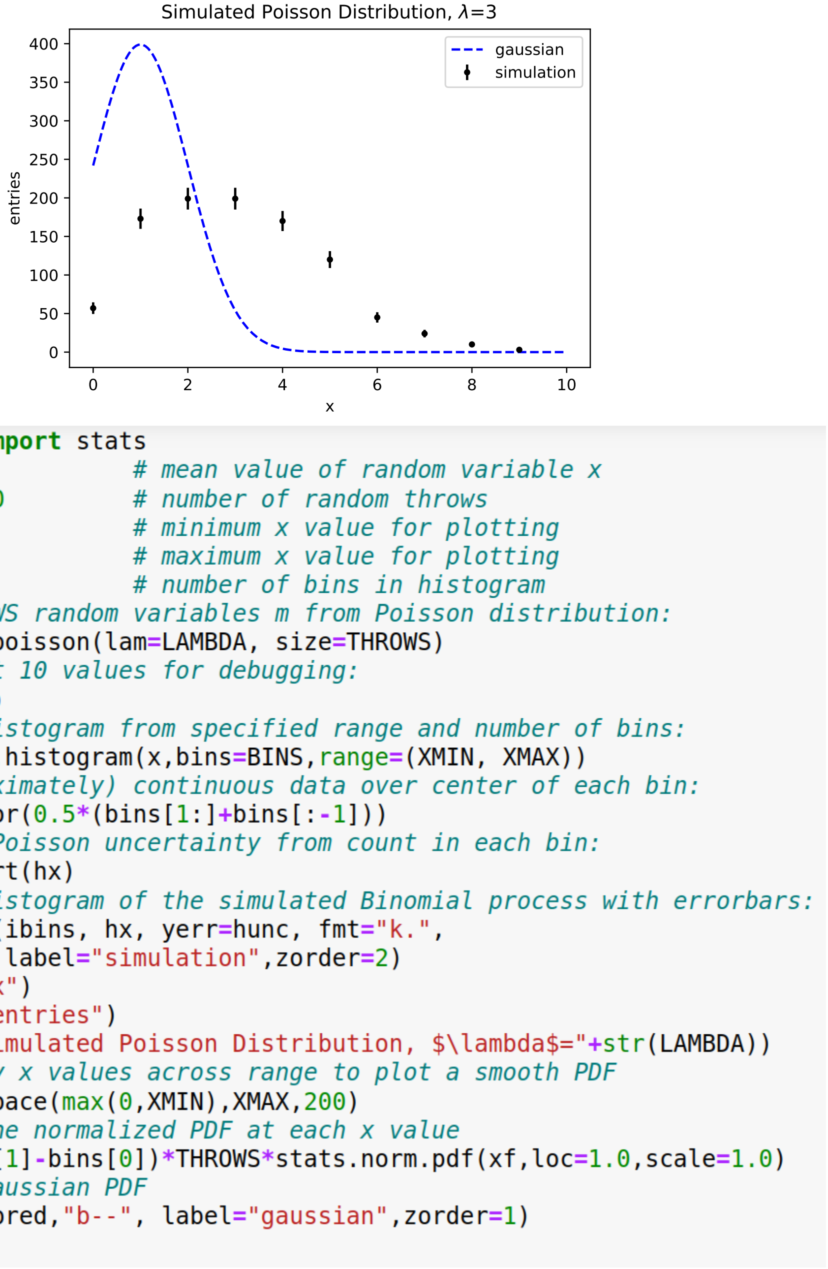
- You'll want to simulate a large number `THROWS` of average values, where each avearge value is based on `NSUM` individual uniform random variables. Produce your uniform random variables in the range $[-1, 1]$ using the `np.random.uniform` function.
- As all of the variables in this section are continuous, you should use the bin centers when plotting histograms, as in previous labs, calculated as `cbins = (bins[1:]  + bins[:-1])/2`.
- You will be comparing your simulated results to the Gaussian PDF. The best estimates for the mean and variance of the Gaussian from our sample of simulated data is provided by the sample mean and sample variance, which are calculated by the `np.mean` and `np.var` funtions. Make sure to normalize your PDF to obtain your prediction.

Jupyter Notebook 4.5.

Generate `THROWS=10000` averages of `NSUM=100` random variables thrown from uniform distribution in the range $[-1, 1]$. Plot a histogram of these average values across an appropriate range that clearly shows the bell shaped Gaussian distribution, using about 50 bins. On the same axes, plot a Gaussian PDF, appropriately normalized, to compare with your simulation.

Jupyter Notebook 4.6.

Using the statistical concepts we developed in lecture, you should be able to calculate, with pencil, the expected mean and variance of the average values. Determine these predicted values and compare the values reported by `np.var` and `np.mean` for your simulation.



```
from scipy import stats
LAMBDA = 3               # mean value of random variable x
THROWS = 1000            # number of random throws
XMIN   = 0               # minimum x value for plotting
XMAX   = 10              # maximum x value for plotting
BINS   = 10              # number of bins in histogram
# throw THROWS random variables m from Poisson distribution:
x=np.random.poisson(lam=LAMBDA, size=THROWS)
# print first 10 values for debugging:
print(x[:10])
# create a histogram from specified range and number of bins:
hx,bins = np.histogram(x,bins=BINS,range=(XMIN, XMAX))
# plot (aproximately) continuous data over center of each bin:
ibins=np.floor(0.5*(bins[1:]+bins[:-1]))
# Calculate Poisson uncertainty from count in each bin:
hunc = np.sqrt(hx)
# Plot the histogram of the simulated Binomial process with errorbars:
plt.errorbar(ibins, hx, yerr=hunc, fmt="k.",
             label="simulation",zorder=2)
plt.xlabel("x")
plt.ylabel("entries")
plt.title("Simulated Poisson Distribution, $\lambda$="+str(LAMBDA))
# choose many x values across range to plot a smooth PDF
xf = np.linspace(max(0,XMIN),XMAX,200)
# evaluate the normalized PDF at each x value
pred = (bins[1]-bins[0])*THROWS*stats.norm.pdf(xf,loc=1.0,scale=1.0)
# plot the Gaussian PDF
plt.plot(xf,pred,"b--", label="gaussian",zorder=1)
plt.legend()
```

Figure 4.2: Simulation of a binomial process.

Chapter 5

Experimental Uncertainties

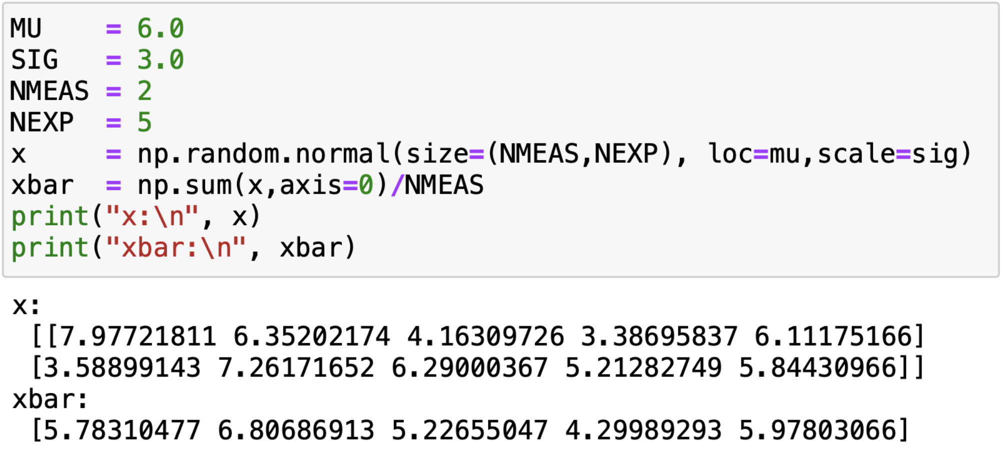
5.1 Introduction

In this lab, you will explore how the sample mean and sample variance can be used to estimate experimental uncertainties. You will also use the Monte Carlo method to explore the standard propogation of uncertainties to calculated quantities.

5.2 Sample Mean and Sample Variance

The Monte Carlo simulation of experimental data is an essential tool of experimental physics in large part because, unlike in our real experiments, we know the true values of the parameters used to produce our simulated experimental data.

In this section, we will use the Monte Carlo method to examine the statistical properites of the sample mean and sample variance, from a series of experiments each making repeated measurement of a single quanity x. We'll produce our simulated experiments with known parameters μ and σ, which are the "true values" of the simulation. We'll then calculate experimental estimates for these quantities and compare them to the known true values.



```
MU    = 6.0
SIG   = 3.0
NMEAS = 2
NEXP  = 5
x     = np.random.normal(size=(NMEAS,NEXP), loc=mu,scale=sig)
xbar  = np.sum(x,axis=0)/NMEAS
print("x:\n", x)
print("xbar:\n", xbar)
```

```
x:
 [[7.97721811 6.35202174 4.16309726 3.38695837 6.11175166]
 [3.58899143 7.26171652 6.29000367 5.21282749 5.84430966]]
xbar:
 [5.78310477 6.80686913 5.22655047 4.29989293 5.97803066]
```

Figure 5.1: Five simulated experiments each averaging two measurements.

As we will need to simulate multiple experiments, and each experiment will make multiple measurements, our simulated measurements will be in the form of a 2-D numpy array, as showin in Fig. 5.1. When looking through the example, note the following key features:

- There are two counts to consider, the number of experiments `NEXP` and the number of measurements `NMEAS`. For example, if we want to study the statistics of experiments involving only a few measurements, we can still simulate many experiments, each with only a few measurements.
- We simulate measurements by throwing random numbers from a Gaussian distribution, using the function `np.random.normal`. The argument `loc` is the mean and the argument `scale` is σ.
- We compute sums using the `np.sum` and use the argument `axis` to sum the columns. After summing the columns in our `NMEASxNEXP` matrix, we are left with a 1-D array with one sum per experiment.
- Notice that there are no for loops. The C++ version of this code would use two. It might take some getting used to, but this is one of the features that makes Python less error prone and more fun to write. You spend less time on boring bookkeeping of counts and more time thinking about the problem itself.

An experiment with N measurements of a quantity x can estimate any $\langle f(x)\rangle$ as:

$$\bar{f} = \frac{1}{N}\sum_i f(x_i)$$

This is because the measurements x are drawn from the distribution $P(x)$ associated with the experiment and so the sum is an approximation for the integral:

$$\langle f(x)\rangle = \int f(x_i)P(x)dx$$

This is why $\bar{x}$ is a good experimental estimate for μ. We can use the same technique to estimate the experimental uncertainty σ as the standard deviation of the measured values. While the variance is defined as:

$$\sigma^2 = \langle (x-\mu)^2\rangle = (x-\mu)^2 P(x)dx$$

we can estimate it from the data as:

$$\sigma^2_\mu = \frac{1}{N}\sum_i (x_i-\mu)^2$$

provided that we know the true value of μ, which we do in this case. We'll call this quantity the sample variance with respect to μ. When using Monte Carlo simulation, it is often illuminating to use "truth" information, as we do here, but keep in mind that there is no corresponding experimental technique!

Jupyter Notebook 5.1.

Simulate 100,000 experiments each making 10 measurements of a value x, drawn from a Gaussian distribution with $\mu = 5$ and $\sigma = 2$. Use the summing technique (along an axis) to calculate the sample variance with respect to μ:

$$\sigma_\mu^2 = \frac{1}{N} \sum_i (x_i - \mu)^2$$

Make certain to use the true value $\mu = 5$. As always, start with a small numbers (e.g, 10 experiments each making 3 measurements) while developing and debugging your code. Then scale up to the requested numbers once your code is working reliably. Calculate the ratio of σ_μ^2 to the true value $\sigma^2 = 4$.

Calculating the sample variance with respect to μ requires knowing the true value of μ. This situation does sometimes arise in the lab, such as when you are calibrating your device with a known input. In this case, you know the true value μ, and you can estimate the uncertainty of your measurement σ by calculating σ_μ.

Often, however, the entire purpose of your experiment is to measure the unknown quantity μ. Supposing one did not know the uncertainty of the measurement σ, how could you obtain an estimate. The best you can do, in this case, is use your best estimate of μ which is $\bar{x}$ and calculate

$$s_N^2 = \frac{1}{N} \sum_i (x_i - \bar{x})^2.$$

The quantity s_N^2 is called the sample variance, and the quantity s_N is called the sample standard deviation.

Jupyter Notebook 5.2.

Modify your code to calculate the sample variance:

$$s_N^2 = \frac{1}{N} \sum_i (x_i - \bar{x})^2.$$

Use the `np.sum` function as in the previous exercise. Calculate the ratio of s_n^2 to the true value $\sigma^2 = 4$.

You results should show that for $N = 10$ the sample variance s_N^2 is biased from the true value of σ^2. It turns out that

$$\langle s_N^2 \rangle = \frac{N-1}{N} \sigma^2 \tag{5.1}$$

which is an optional exercise in the lecture notes. A plausible explanation for this effect comes from noting that the quantity:

$$\sum_i (x_i - \mu)$$

is unknown whereas our definition of $\bar{x}$ implies that

$$\sum_i (x_i - \bar{x}) = 0$$

Which is a fixed constraint on the N values used to compute the sum in s_N^2. This constraint reduces the number of independent measurements to $N - 1$, biasing the result by the factor $(N - 1)/N$.

Understanding this bias allows us to caculate the *unbiased* sample variance as:

$$s^2 = \frac{1}{N-1}\sum_i (x_i - \bar{x})^2$$

The use of $N-1$ instead of N is called Bessel's correction.

Jupyter Notebook 5.3.

Modify your code to calculate the unbiased sample variance:

$$s^2 = \frac{1}{N-1}\sum_i (x_i - \bar{x})^2.$$

Use the `np.sum` function as in the previous exercise. Calculate the ratio of s_n^2 to the true value $\sigma^2 = 4$.

Numpy provides the functions `np.mean` and `np.var` to calculate the sample mean and sample variance. You can use the argument `ddof` to select between the biased and unbiased sample variation.

Jupyter Notebook 5.4.

Modify your code to calculate mean, biased sample variance and unbiased sample variance, using the numpy functions `np.mean` and `np.var`.

We showed in lecture that the uncertainty on the sample mean $\bar{x}$ of N measured values is smaller than the uncertainty $sigma_x$ of the individual measurements:

$$\sigma(\bar{x}) = \sigma_x/\sqrt{N}$$

This relationship illustrated in Fig. reffig:uncmean. When looking through the example code, notice the following features:

- The x-axis is plotted on a log scale, by the function call `plt.semilogx`
- If the x values `nf` for plotting a smooth function where choosen uniformly in N, there would be roughly 10,000 times as many points above $N = 10^3$ as below $N = 10^1$. To make the plot smooth everywhere would require a *lot* of points. Instead, the x-axis values are determined as $N = 10^a$ where a is chosen uniformly in the range $[0, 4]$. This is a very useful technique!

Jupyter Notebook 5.5.

Simulate 10,000 experiments each of which make N measures of a quantity x with $\mu = 0$ and $\sigma = 1$. Calculate the unbiased sample variance of each experiment, and then the mean of these sample variances across all experiments. Plot the results as function of N for $N = 1, 10, 10^2, 10^3$ and 10^4. On the same axis, included the prediciton from Fig. 5.2.

Jupyter Notebook 5.6.

Modify your previous example so that *both* the x-axis and y-axis have a log scale. Why does the function now appear linear?

```python
MAXPOW = 4
nf = np.power(10,np.linspace(0,MAXPOW,200))
sf = 1./np.sqrt(nf)
plt.title("Uncertainty on the Mean")
plt.semilogx()
plt.plot(nf,sf,"b--",label="$1/\sqrt{N}$")
plt.xlabel("N")
plt.ylabel("$\sigma$(mean) / $\sigma$(x)")
plt.legend()
plt.savefig("uncmean.pdf", bbox_inches="tight")
```

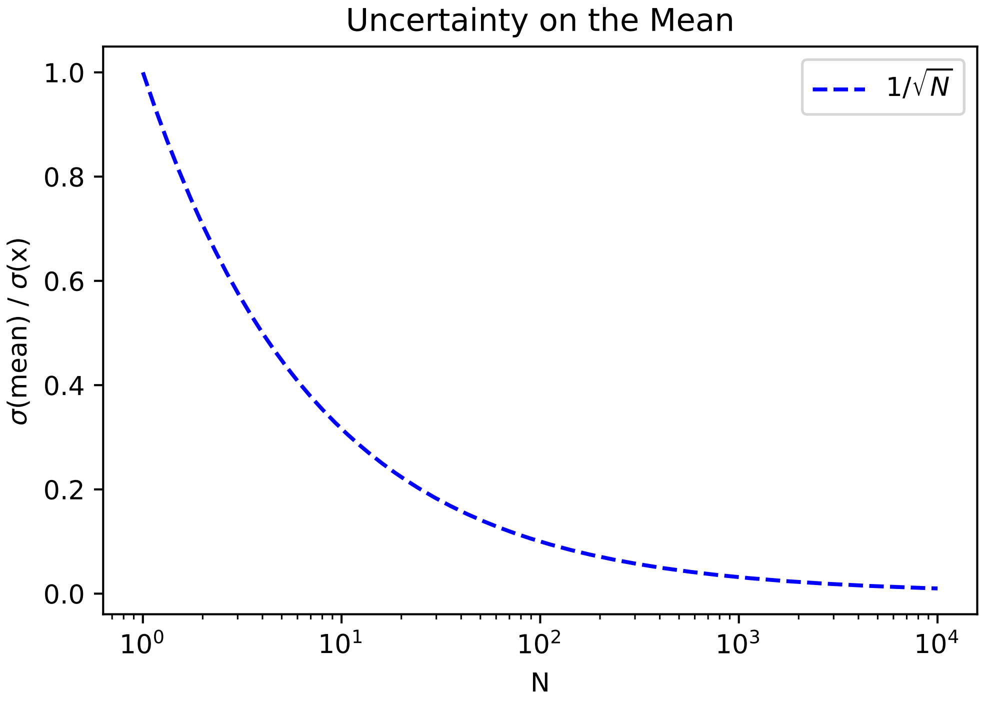


Figure 5.2: The uncertainty on the mean versus number of measurements.

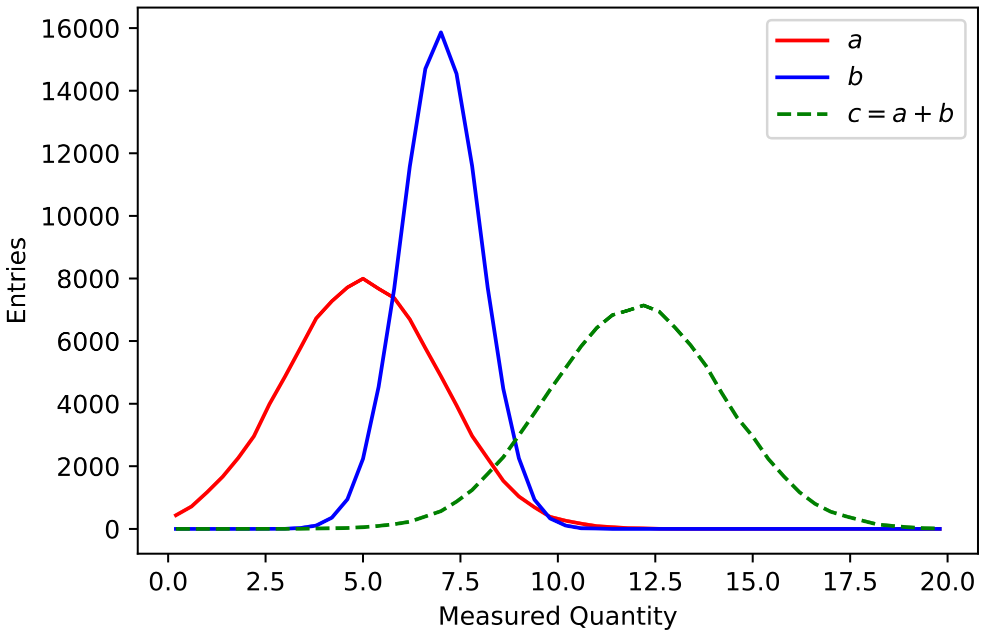


Figure 5.3: Simulation of many measurements of the quantity $c = a + b$.

5.3 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 $c = a \cdot b$ 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. 5.3.

Jupyter Notebook 5.7.

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. 5.3) as histograms with 50 bins and an appropriate range. Use values of a and b that are different from the example. Calculate the mean and variance of the simulated c values and compare to your expectations from the standard propagation of uncertainties.

Jupyter Notebook 5.8. Repeat the previous exercise but for multiplication. Use new values for a and b if you like.

Our general formula for propogating uncertainties uses a Taylor explansion approximation. This means that the approximation is not valid if the function is varying wildly within the variable uncertainties. This condition can be easily broken for division a/b.

Jupyter Notebook 5.9. Repeat the previous exercise but for division. Find values for a and b which cause the standard propogation of uncertainty to fail. What should an experimenter do in such case. (Hint: you are looking at it!)

Chapter 6

Curve Fitting

6.1 Introduction

In this lab, 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:

$$\chi^2 = \sum_i \frac{(f(x_i;p) - y_i)^2}{\sigma_i^2}. \tag{6.1}$$

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.

6.2 Fitting a Straight Line

An example using Scientific Pythons `curve_fit` function to fit a straight line to data is shown in Fig. 6.1. 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
```

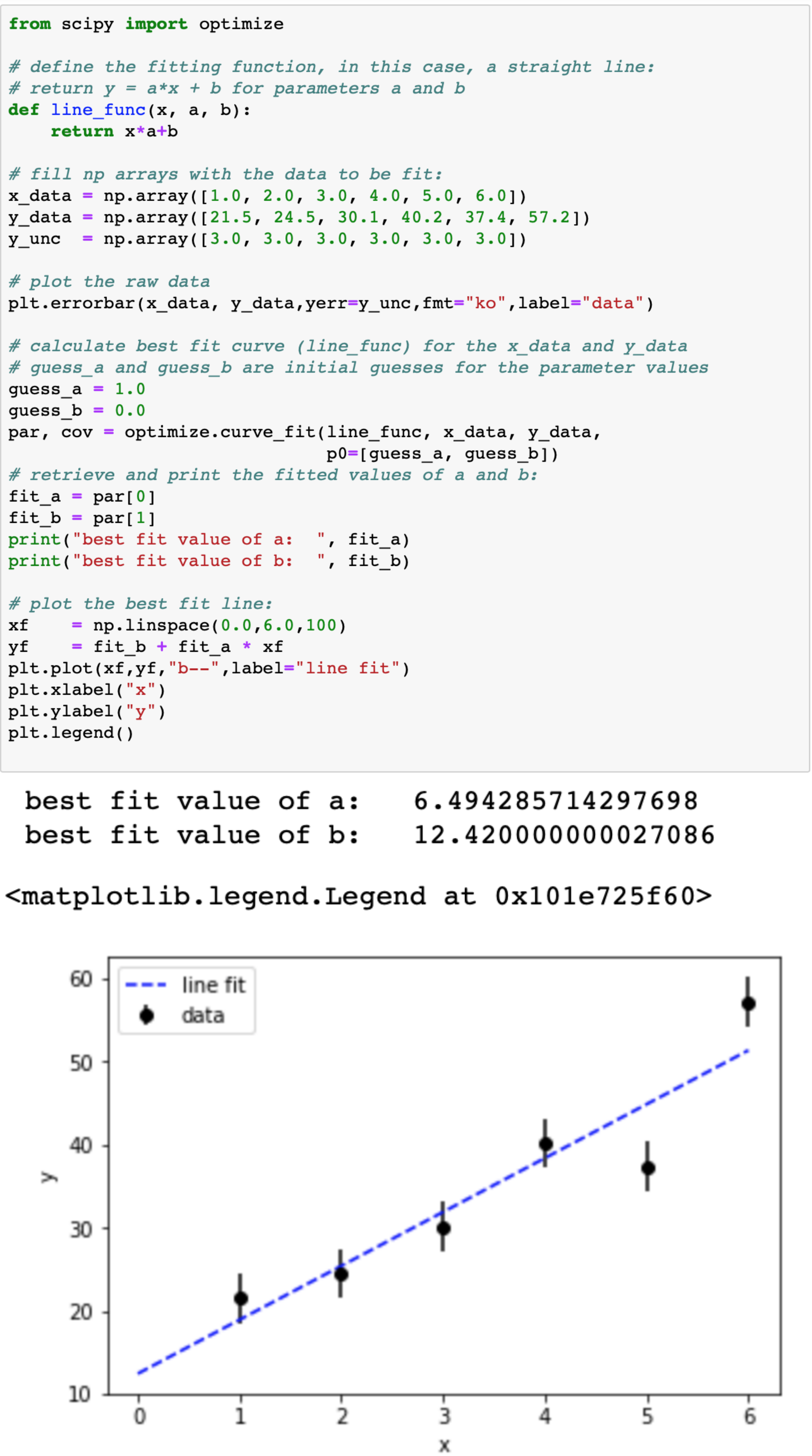
In this case, the function requires three parameters (in the computer science sense, not the mathematical sense) the x data in a numpy array as function parameter x, the slope as function parameter a, and the intercept as function parameter b. When called, the function returns the x data multiplied by 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 6.1: Example fitting data to straight line.

Table 6.1: 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.

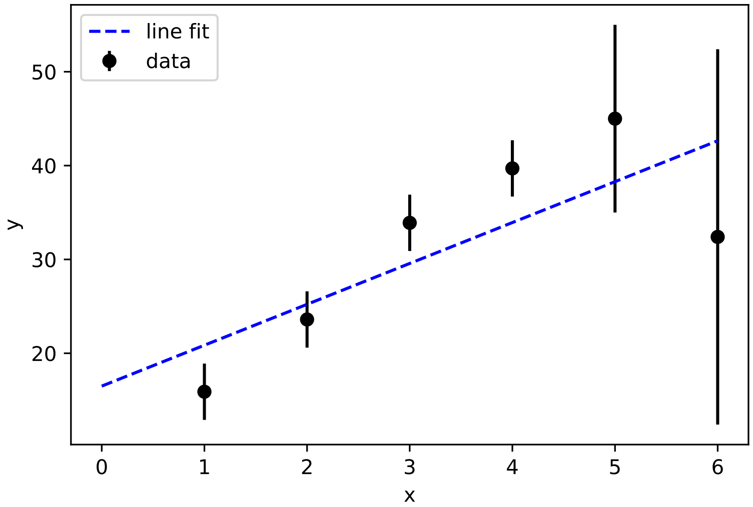


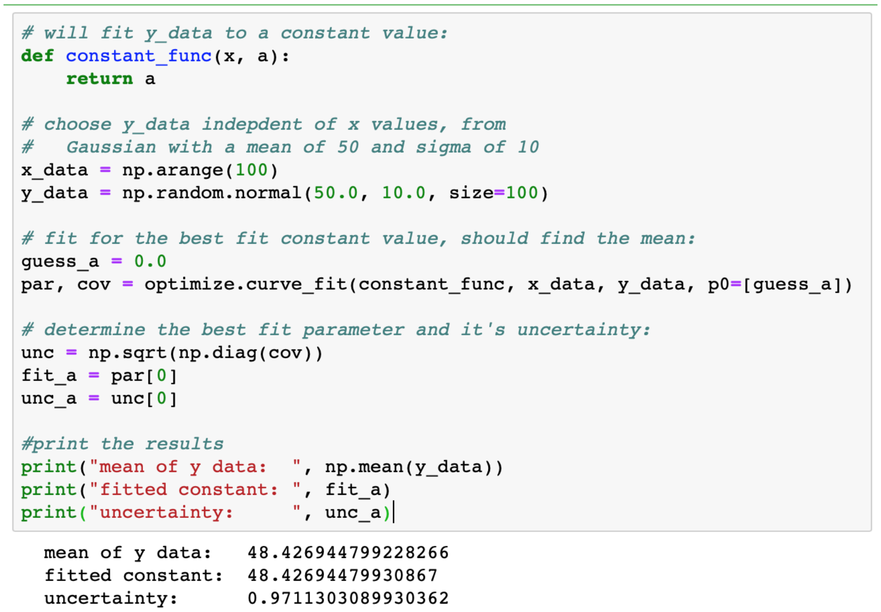
Figure 6.2: This linear fit is biased by the failure to properly account for uncertainties. An unbiased fit would track the well constrained points more closely.

Jupyter Notebook 6.1. Apply code like that of Fig. 6.1 to the data in Table 6.1. Plot the data including error bars and the best-fit function to obtain a result like that of Fig. 6.2.

Notice that the last two data points in Table 6.1 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. 6.2) toward the poorly measured points, because the function gives these points the same weight as all of the other points.

Jupyter Notebook 6.2. 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.

6.3 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 6.3: 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. 6.3, 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:

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

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 6.3. Repeat the fit in Fig. 6.3, but set the uncertainties on the y values to 10 and also set `absolute_sigma = True` in the fit. Record the uncertainty.

Jupyter Notebook 6.4. 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.

6.4 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 6.5. 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`.

Chapter 7

Simulation of an Ideal Gas

7.1 Introduction

For an ideal gas composed of molecules with mass m at temperature T, the probability density for the component of velocity in the x direction (v_x) is given by:

$$P(v_x) = \sqrt{\frac{m}{2\pi k_{\mathrm{B}} T}} \exp\left(-\frac{m v_x^2}{2 k_{\mathrm{B}} T}\right) \tag{7.1}$$

where k_{B} is Boltzmann's constant. Similary for the y direction:

$$P(v_y) = \sqrt{\frac{m}{2\pi k_{\mathrm{B}} T}} \exp\left(-\frac{m v_y^2}{2 k_{\mathrm{B}} T}\right). \tag{7.2}$$

For simplicity, we will be simulating a gas in two dimensions. The infinitesimal probability associated with a velocity (v_x, v_y) is given by:

$$\begin{aligned} P(v_x, v_y)\, dv_x\, dv_y &= P(v_x)\, dv_x\, P(v_y)\, dv_y \\ &= \frac{m}{2\pi k_{\mathrm{B}} T} \exp\left(-\frac{m(v_x^2 + v_y^2)}{2 k_{\mathrm{B}} T}\right) dv_x\, dv_y \\ &= \frac{m v}{2\pi k_{\mathrm{B}} T} \exp\left(-\frac{m v^2}{2 k_{\mathrm{B}} T}\right) d\theta\, dv \end{aligned}$$

where we have changed to polar coordinates v and θ in the usual manner with area differential $dv_x\, dv_y = v\, dv\, d\theta$. This allows us to read off the probability density in polar coordintes:

$$P(v, \theta) = \frac{m v}{2\pi k_{\mathrm{B}} T} \exp\left(-\frac{m v^2}{2 k_{\mathrm{B}} T}\right)$$

Integrating over all possible directions θ, we obtain:

$$\begin{aligned} P(v) &= \int_0^{2\pi} P(v, \theta) d\theta \\ &= \int_0^{2\pi} \frac{m v}{2\pi k_{\mathrm{B}} T} \exp\left(-\frac{m v^2}{2 k_{\mathrm{B}} T}\right) \\ P(v) &= \frac{m v}{k_{\mathrm{B}} T} \exp\left(-\frac{m v^2}{2 k_{\mathrm{B}} T}\right) \end{aligned} \tag{7.3}$$

which is the Maxwell-Boltzmann distribution for an ideal gas in two dimensions. This is the probability density for a gas molecule to have speed v.

In this lab, we will create a simple numerical simulation of an ideal gas and verify that the velocity of the gas follows the Maxwell-Boltzmann distribution.

7.2 System of Units

Choosing an effective system of units is essential for building a well-behaved numerical simulation. Consider the Maxwell-Boltzmann distribution, which involves the following SI values:

- Boltzmann's constant: $k_{\mathrm{B}} = 1.38 \times 10^{-23}$ J/K
- Molecular masses: e.g. N_2 with $m = 4.65 \times 10^{-26}$ kg.
- Temperature: e.g. room temperature $T = 293$ K.

The smallest number greater than zero that a computer can represent with a single-precision floating point number is approximately 10^{-38}. Representing the SI value of Boltzmann's constant at 10^{-23} uses a large fraction of this precision before we even begin our calculation. Numerical algorithms using floating point numbers work best when the values involved in the calculation are near one.

It is usually best, therefore, to devise an alternate system of units for any numerical simulation which keeps the values of variables of interest as near one as possible. We will call this the numerical system of units.

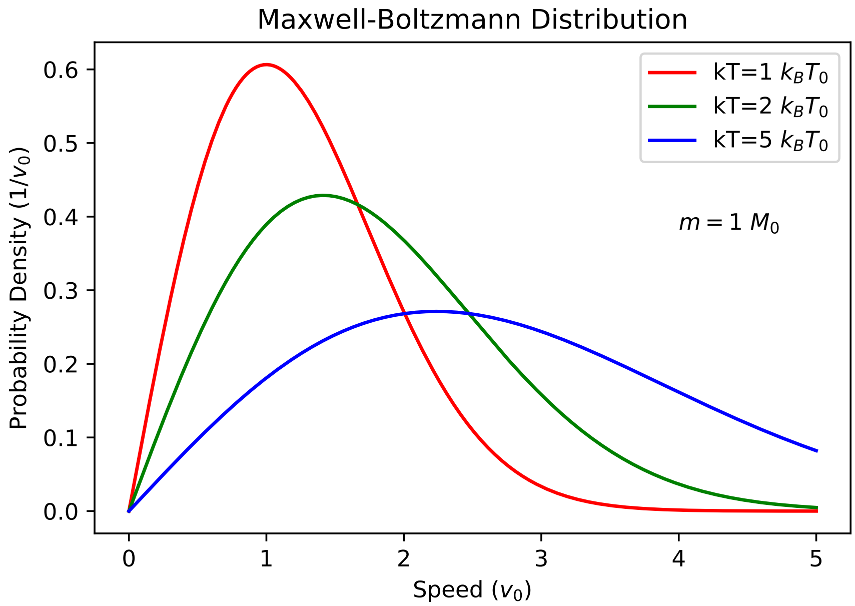
To start, we choose a reference temperature near the temperature we would like to simluate, say $T_0 = 293$ K. All temperatures in the simulation will be in units of this reference temperature. So a temperature `T=1.2` in the program will be $1.2\,T_0 = 352$ K in SI units. Our model also includes mass, so we choose a reference mass near the mass of the molecules we will be simulating, say $M_0 = 4.65 \times 10^{-26}$ kg. A mass `m=2.1` in our program would have an SI value value of $2.1 M_0 = 9.8 \times 10^{-26}$ kg.

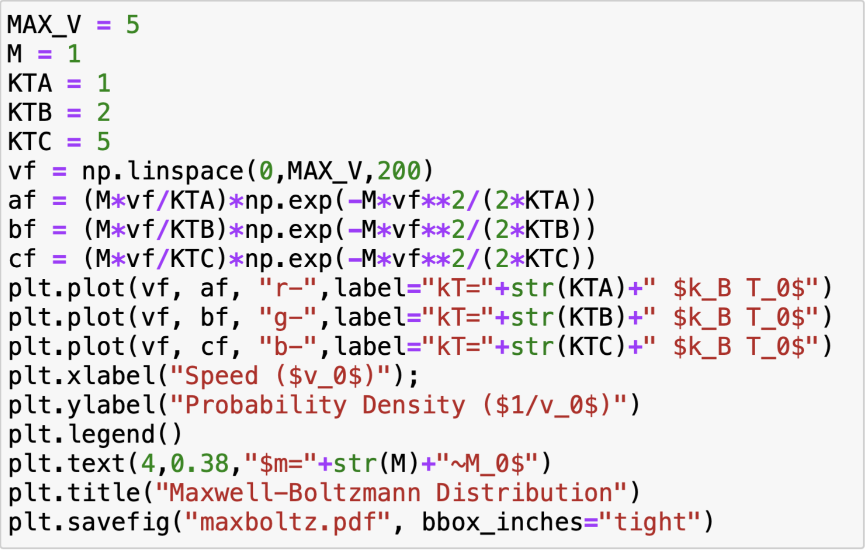
The physics we will simulate involves Boltzmann's constant k_{B} which will have a value of one in our program. This sets the reference energy from our reference temeperature. For example, an energy `kT = 3` in our program will have an SI value of $k_{\mathrm{B}}T = 3\ k_B T_0 = 1.21 \times 10^{-20}\ J$. The reference energy and reference mass together define a reference velocity:

$$V_0 = \sqrt{\frac{k_b T_0}{M_0}} = 295 \text{ m/s}.$$

The only time the actual values choosen for the numerical system of units are needed is if you need to convert inputs in SI units to the numerical system of units, or convert the results of your simulation to SI units. In this lab, we will specify all inputs and report all results using the numerical system of units. **So there is no need for specific values such as $M_0 = 2.32 \times 10^{-25}$ kg to appear anywhere in your program.** If such values do appear, outside of comments, you are certainly making a mistake!

As an example, the Maxwell-Boltzmann distribution is plotted in Fig. 7.1 alongside the code used to produce it. Notice how for `kT` and `m` near one, the typical velcities are also near one. This is sign of good numerical system of units. Notice also that Boltzmann's constant or any other small or large numbers in SI units do not appear anywhere in the code.





```
MAX_V = 5
M = 1
KTA = 1
KTB = 2
KTC = 5
vf = np.linspace(0,MAX_V,200)
af = (M*vf/KTA)*np.exp(-M*vf**2/(2*KTA))
bf = (M*vf/KTB)*np.exp(-M*vf**2/(2*KTB))
cf = (M*vf/KTC)*np.exp(-M*vf**2/(2*KTC))
plt.plot(vf, af, "r-",label="kT="+str(KTA)+" $k_B T_0$")
plt.plot(vf, bf, "g-",label="kT="+str(KTB)+" $k_B T_0$")
plt.plot(vf, cf, "b-",label="kT="+str(KTC)+" $k_B T_0$")
plt.xlabel("Speed ($v_0$)");
plt.ylabel("Probability Density ($1/v_0$)")
plt.legend()
plt.text(4,0.38,"$m="+str(M)+"~M_0$")
plt.title("Maxwell-Boltzmann Distribution")
plt.savefig("maxboltz.pdf", bbox_inches="tight")
```

Figure 7.1: The Maxwell-Boltzmann distribution using a system of units appropriate for a numerical simulation, along with the code used to produce the plot.

7.3 Collision Model

At the heart of your numerical simulation is the collision model. It is the collisions of molecules that will allow your simulated gas to reach thermal equilibrium. We will use the simple elastic collision of identical mass particles, as illustrated in Fig. 7.2, as our collision model. We consider particles a and b with velocities $\vec{v_a}$ and $\vec{v_b}$ in the lab frame. The velocity of particle a in the CMS frame before the collision is

$$\vec{u} = \frac{\vec{v_a} - \vec{v_b}}{2}.$$

The collision rotates the velocity of particle a by the scattering angle θ so that the velocity $\vec{w}$ after the collision is

$$\begin{pmatrix} w_x \\ w_y \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix}$$

In the lab frame, the velocity of molecule a changes by an amount:

$$\Delta\vec{v_a} = \vec{w} - \vec{u}$$

and the velocity of molecule b changes by an amount:

$$\Delta\vec{v_b} = \vec{u} - \vec{w}$$

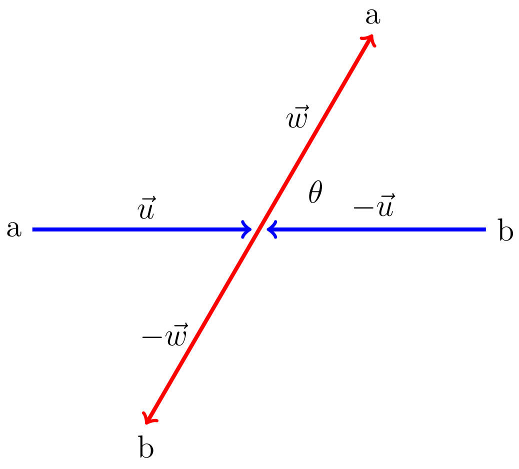


Figure 7.2: The collision model in the center-of-mass: incoming molecule a with velocity $\vec{u}$ collides with the incoming particle b of identical mass with velocity $-\vec{u}$. Particle a is scattered by angle θ and leaves with velocity $\vec{w}$, while particle b leaves with velocity $\vec{w}$. The magnitude of the final and initial velocities are the same: $|\vec{u}| = |\vec{w}|$.

7.4 Implementing the Collision Model

Our Python implementation for the collision will be computed in terms of the components of the velocity vectors of molecule a and molecule b:

$$\begin{aligned}\vec{v_a} &= \begin{pmatrix} a_x \\ a_y \end{pmatrix} \\ \vec{v_b} &= \begin{pmatrix} b_x \\ b_y \end{pmatrix}\end{aligned}$$

We'll use the Python variable names `ax`, `ay`, `bx`, and `by` to refer to a_x, a_y, b_x, and b_y.

First calculate the x and y component of $\vec{u}$ as:

$$\begin{aligned}u_x &\equiv \frac{a_x - b_x}{2} \\ u_y &\equiv \frac{a_y - b_y}{2}\end{aligned}$$

Then compute the x and y component to the change in velocity of particle a and particle b:

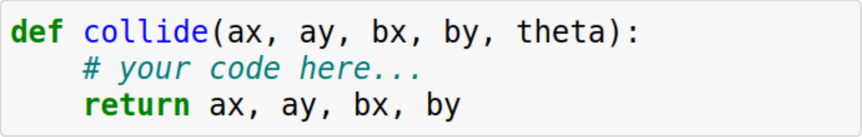
$$\begin{aligned}\Delta a_x &= (\cos\theta - 1)\, u_x + \sin\theta\, u_y \\ \Delta a_y &= (\cos\theta - 1)\, u_y - \sin\theta\, u_x \\ \Delta b_x &= (1 - \cos\theta)\, u_x - \sin\theta\, u_y \\ \Delta b_y &= (1 - \cos\theta)\, u_y + \sin\theta\, u_x\end{aligned}$$

Finally, update the x and y components of the particle velocities to their value after the collision:

$$\begin{aligned}a_x &\to a_x + \Delta a_x \\ a_y &\to a_y + \Delta a_y \\ b_x &\to b_x + \Delta b_x \\ b_y &\to b_y + \Delta b_y\end{aligned}$$

In essential technique for programming complicated task is dividing complicated tasks into smaller tasks, and thoroughly testing the smaller tasks. You cannot program effectively until you master this technique. I've taught students programming for many years, and the students that finish last are invariably the ones that rush to complete their entire program and then try to test and debug it. This approach always fails because when you do not get the right answer, and you won't, ever, on the first try, you have absolutely no idea what part of a very long chain of calculations is not programmed correctly.

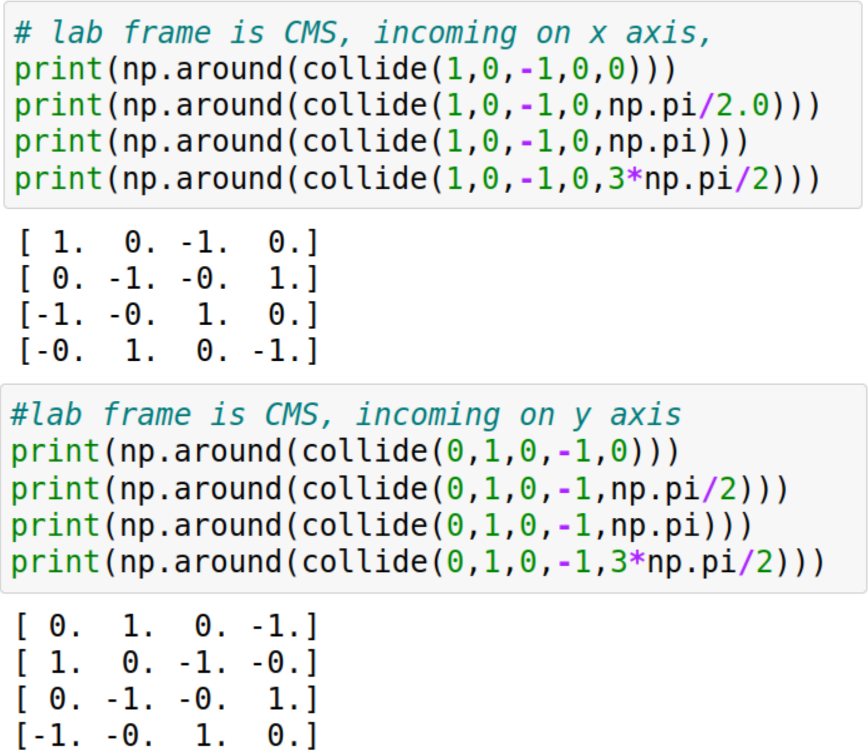
To use this approach in the lab, we'll be implementing the collision algorithm as a function, exactly as in Fig. 7.3. This function takes as input the velocity components `ax`, `ay`, `bx`, `by` as defined above plus the scattering angle `theta`. In Fig. 7.3, the function simply returns the velocity components unchanged. You should modify the function to implement the scattering algoirthm described above.



```
def collide(ax, ay, bx, by, theta):
    # your code here...
    return ax, ay, bx, by
```

Figure 7.3: Collision function.

Normally at this point, you would have to devise your own test to validate your code. One technique, that would work here, is to calculate a few examples and then compare your program output to what you obtained with paper and pencil. For this lab, I will provide some specific example calculations for you to validate your collision function.



```
# lab frame is CMS, incoming on x axis,
print(np.around(collide(1,0,-1,0,0)))
print(np.around(collide(1,0,-1,0,np.pi/2.0)))
print(np.around(collide(1,0,-1,0,np.pi)))
print(np.around(collide(1,0,-1,0,3*np.pi/2)))
```

```
[ 1.  0. -1.  0.]
[ 0. -1. -0.  1.]
[-1. -0.  1.  0.]
[-0.  1.  0. -1.]
```

```
#lab frame is CMS, incoming on y axis
print(np.around(collide(0,1,0,-1,0)))
print(np.around(collide(0,1,0,-1,np.pi/2)))
print(np.around(collide(0,1,0,-1,np.pi)))
print(np.around(collide(0,1,0,-1,3*np.pi/2)))
```

```
[ 0.  1.  0. -1.]
[ 1.  0. -1. -0.]
[ 0. -1. -0.  1.]
[-1. -0.  1.  0.]
```

Figure 7.4: Example collisions along the x and y axis.

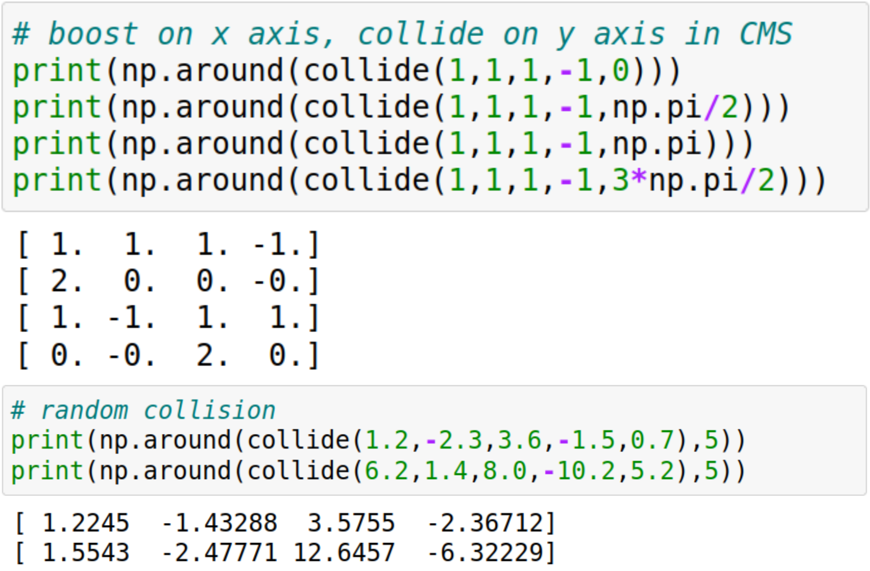
Jupyter Notebook 7.1.

Implement the collision algorithm as a function as in Fig. 7.3 and test it using example collisions from Fig. 7.4.

When testing your code, start with easy, special cases, such as used in Fig. 7.3. This helps makes it clearer where the program is failing. Once your code works on the simple cases, escalate to more complicated examples.

Jupyter Notebook 7.2.

Test your collision algorithm using the example collisions from Fig. 7.5.



```
# boost on x axis, collide on y axis in CMS
print(np.around(collide(1,1,1,-1,0)))
print(np.around(collide(1,1,1,-1,np.pi/2)))
print(np.around(collide(1,1,1,-1,np.pi)))
print(np.around(collide(1,1,1,-1,3*np.pi/2)))
```

```
[ 1.  1.  1. -1.]
[ 2.  0.  0. -0.]
[ 1. -1.  1.  1.]
[ 0. -0.  2.  0.]
```

```
# random collision
print(np.around(collide(1.2,-2.3,3.6,-1.5,0.7),5))
print(np.around(collide(6.2,1.4,8.0,-10.2,5.2),5))
```

```
[ 1.2245  -1.43288  3.5755  -2.36712]
[ 1.5543  -2.47771 12.6457  -6.32229]
```

Figure 7.5: More complicated example collisions.

7.5 Initializing the Simulated Ideal Gas

You will be modeling an ideal gas by direct Monte Carlo simulation of NGAS representative molecules. We will use NGAS=1000 intially, and you should use an even lower value while debugging. We'll assume that the mass of each molecule in the gas is M_0, or in the numerical system of units M=1.

The state of your simulation will be completely contained in two numpy arrays vx and vy, each of length NGAS, which contain the velocities of the particles in units of $V_0 = \sqrt{k_b T_0 / M_0}$. Remember, the simulation uses a system of units that should keep velocities near 1, so values such as 2.2, -3.1, 0.8, -0.01 are all likely, and correspond to speeds up to several hundred meters per second in SI units. On the other hand, the presence of extremely small values, like 5.3E-23, and extremely large values like 1.2E18 and -8.2E28 are symptoms of bugs.

Jupyter Notebook 7.3. Initialize both velocity arrays vx and vy of length NGAS with values choosen as uniform random variables in the range $[-2, 2]$. Fill two histograms, one with v_x and one with v_y, with an approriate range and 20 bins. You should see that the velocities are distributed uniformly (a flat distribution). The distribution does not yet resemble the Gaussian shape of Equation 7.2 because it has not yet reached thermal equilibrium.

7.6 Collisions of an Ideal Gas

To reach thermal equilibrium, you'll need to simulate collisions betweens pairs of molecules in your gas. For each collision, do the following:

- Choose two molecules at random as particles a and b. (See np.random.choice.)
- Choose a random value θ uniformly in the range $[0, 2\pi]$ (See np.random.uniform.)
- Call your collision function with components of the velocity vectors for particles a and b and the scattering angle θ.

- Update the velocity of particles a and b from the return value of your collision funciton

For this model, you will need above 10 times as many collisions as gas molecule in order to reach thermal equilibrium.

Jupyter Notebook 7.4. For `NGAS=1000` simulate `NCOLL = 10000` collisions as described above. Fill two histograms, one with v_x and one with v_y, with an approriate range and 20 bins. After reaching thermal equilibrium, the distributions should resemble a Gaussian as predicted by Equation 7.2.

7.7 Temperature of an Ideal Gas

The temperature of the gas is related to the mean kinetic energy by:

$$k_b T = m \, \frac{\langle v_x^2 \rangle + \langle v_y^2 \rangle}{2} \tag{7.4}$$

You can estimate $\langle v_x^2 \rangle$ from your simulation as `np.mean(vx**2)`.

Jupyter Notebook 7.5. Estimate kT of the gas using Equation 7.4 before and after simulating collisions. The values should remain near the expected value 4/3.

7.8 The Maxwell-Boltzmann Distribution

In this section, you'll reproduce the instructor plots of Fig. 7.6 using your own numerical simulation.

Jupyter Notebook 7.6. After your simulation reaches equilibrium, fill two histograms, one with v_x and one with v_y, with an approriate range and 10 bins. Compare with the prediction from Equation 7.2. The results should resemble the right side of Fig. 7.6, which were produced with `NGAS=10000`.

Jupyter Notebook 7.7. After your simulation reaches equilibrium, fill a histograms with the magnitude of the velocity v, with an approriate range and 10 bins. Compare with the prediction from Equation 7.3. The results should resemble the left side of Fig. 7.6, which were produced with `NGAS=10000`.

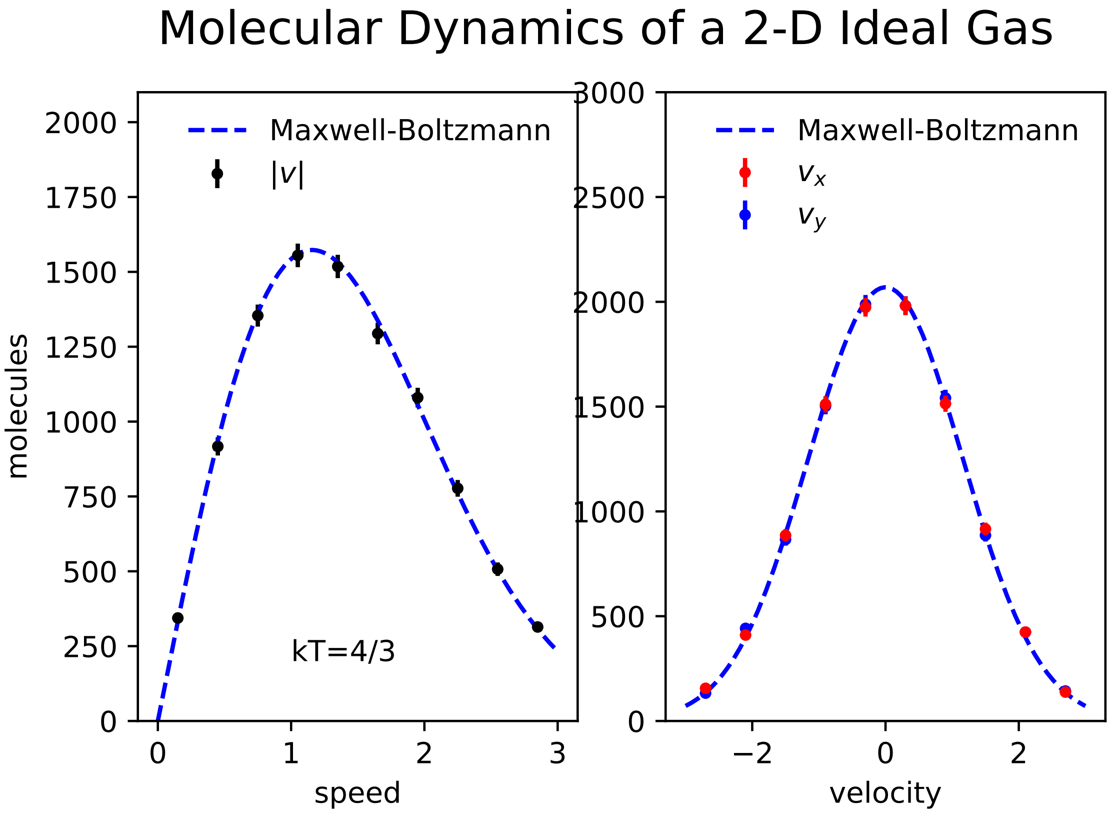


Figure 7.6: Instructor plots.