Physics 40 Lab Manual

Michael Mulhearn

September 18, 2021

Contents

Chapter 1

Installation of Scientific Python

1.1 Introduction

In this lab, you will install the software which we will be using in phy40. This is an assignment, and will be graded. You should submit a text file containing a log of all the steps you took to install the software on your computer. Make this log as specific as possible, an entry might be:

```
Downloaded windows installer from:
 https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe
```

Keeping this log will also make it easier for you to get help if you have problems.

If you run into problems, do some research on a web search tool (Google, for example) to become better informed and to see if you can overcome the problem on your own before asking for help. This is an important technique in getting help with technical problems that will serve you well even outside of this class. You will find it more easy to get useful technical help, from the sort of people most capable of offering it, when it is clear from your question that you are informed and have already tried all of the obvious things. If you are still stuck after trying to solve the problem for yourself, then contact your TA or instructor with specific technical details about what is failing, and include your installation log.

If you do find a problem with these instructions or manage to overcome a technical problem yourself, make sure to note it in your log and inform your TA, in case it is helpful for other students.

1.2 Installing Miniconda3

We will be using Miniconda3 based on Python 3.7 for data analysis using Jupyter notebooks. Miniconda is a lightweight package which we can use to install all of the remaining analysis software we will need in a consistent manner across all different operating systems.

Determine which OS type and version you have on the desktop or laptop computer that you will be using for your coursework. The software here will work under Windows, Linux, or macOS. It should also work on all Chromebooks released since 2019, and some earlier Chromebooks. You should also check whether you have a 32-bit or 64-bit OS (you can find instructions for how to determine this for your particular OS version with a Google search.) Most desktop or laptop computers built in the last ten years are 64-bit.

If you are using Linux or macOS, then from within a terminal type:

```
echo $SHELL
```

to determine the shell you are using (typically "bash" these days). Record all of this information in your installation log file.

Once you have determined your OS type and version, follow the instructions below approprate to your operating system.

1.2.1 Installing under Windows

If you have already installed a version of conda (e.g. Anaconda or Miniconda) then you do not need to re-install it. Instead, find the the Anaconda Prompt in the Application menu and run it.

If you need to install Miniconda3, then download and run the appropriate installer from:

`https://docs.conda.io/en/latest/miniconda.html#`

If prompted, you should choose to:

- Accept the license / terms of use.
- Install for just the current user, not all users.

Once installed, check that you can run the "Anaconda Prompt". From the prompt, check that you can run:

```
conda --version
```

and note the output in your installation log. Then proceed to Section 1.3.

1.2.2 Installing on a Chromebook

You will need to activate Linux on your Chromebook, according to the instructions here:

`https://www.codecademy.com/articles/programming-locally-on-chromebook`

Then follow the insturctions for installing under Linux. If your Chromebook predates 2019 and does not support Linux, contact your instructor for alternative arrangements.

1.2.3 Installing Miniconda3 under Linux or macOS

If you believe you already already have a version of conda installed (such as miniconda or ananconda) , check by running

```
conda --version
```

If you see something like:

```
conda 4.9.2
```

as output (even if the version is different) then you do indeed already have conda installed, with the base environment activated, and you can skip ahead to Section 1.3. If instead you get a message like:

```
conda: command not found
```

then the easiest solution is to simply proceed with these instructions.

To install Miniconda, download the appropriate installer for your OS here:

https://docs.conda.io/en/latest/miniconda.html\#

For macOS, you can choose between a "package" or "bash" version. I find it easier to follow the bash version, but the package version will work too. I recommend you make the following choices if prompted:

- Accept the license / terms of use.
- Do not install for all users, but just one the current user.
- Do allow the installer to issue "conda init".

During the installation, take note of the install location in your log.

After installation with these settings, conda will automatically activate the "base" conda environment. If this annoys you, as it does me, or interferes with other software you are using, you can turn off this agressive behavior with:

```
conda config --set auto_activate_base false
```

Confirm that you have successfully installed conda by typing

```
conda --version
```

Record the output in your installation log, and proceed to Section 1.3.

1.3 Installing the Physics 40 Conda Environment

Make sure your conda is fully up to date with:

```
conda update conda
```

Then follow the prompts, e.g. selecting "y" as needed to update any out-of-date packages.

We'll be using a conda environment specifically for phy40 to avoid conflicts with any other projects on your computer, and to ensure that we all have the same software installed. To create our environment:

```
conda create -n phy40 python=3.9 numpy scipy matplotlib ipython jupyter
```

language=csh

1.4 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 initial code development can be done in the interactive session that Jupyter Notebooks provide.

To activate the phy40 environment type:

```
conda activate phy40
```

When you are done with Phy 40 for the day you can deactivate this environment (later) with:

```
conda deactivate
```

Launch jupyter notebook with:

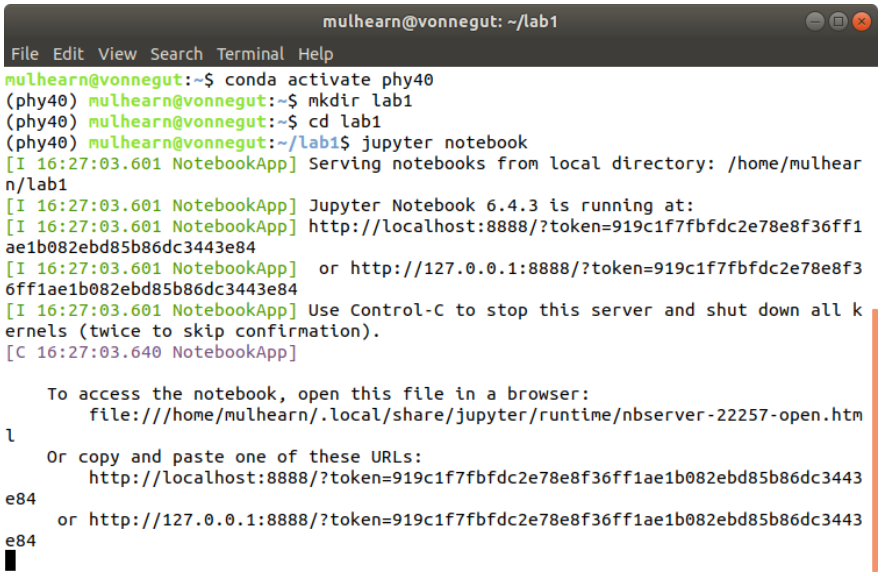


Figure 1.1: Example starting Jupyter Notebook from the Linux command line. In Windows, you will need to open the Anaconda Prompt instead of a terminal.

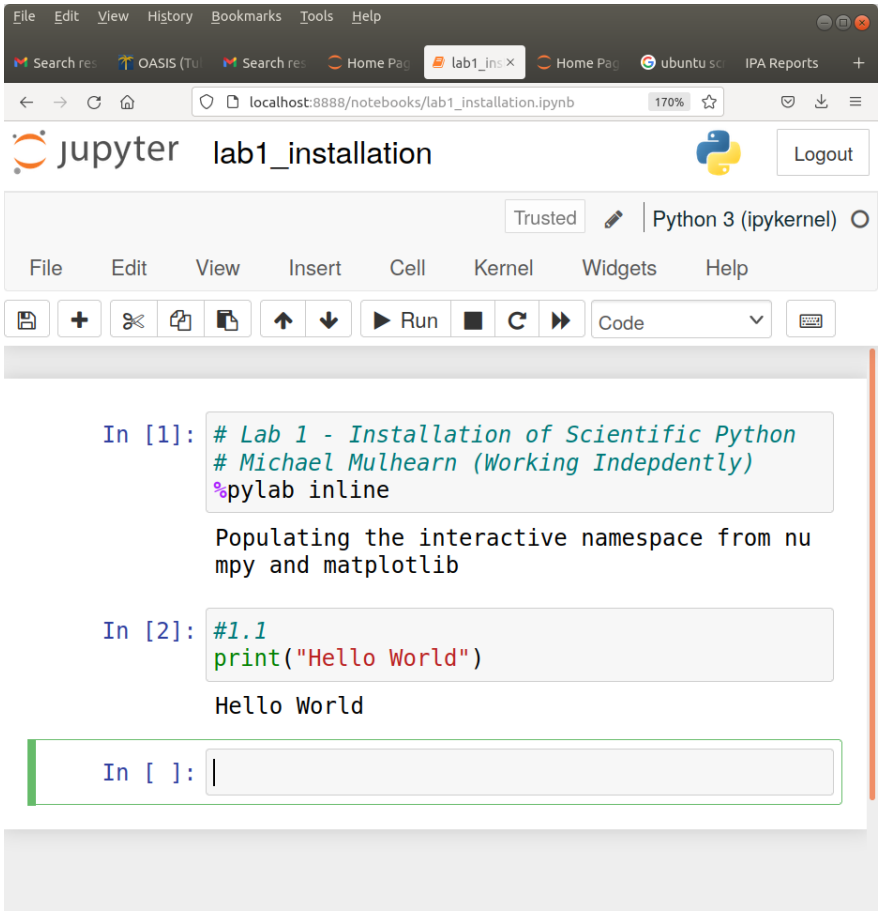


Figure 1.2: The Hello World example Jupyter Notebook.

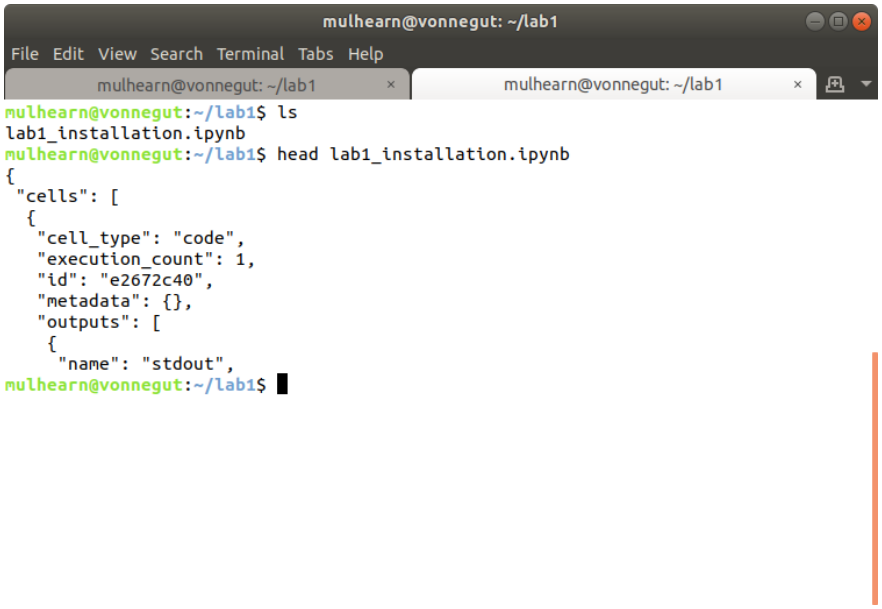


Figure 1.3: Example showing the saved Jupyter notebook. Notice that notebook file (ipynb) is not human readable on its own: it requires the Jupyter software to render it in a human readable form.

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, your first step:

△ **Jupyter Notebook Exercise 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.

1.5 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

Binary Numbers

2.1 Introduction

At the heart of numerical analysis, naturally, you will find numbers. In this lab, we will explore the basic data types in Python, with particular emphasis on the computer representation of integers and real numbers. All modern programming languages do an admirable job of hiding the limitations of the computer representations of these mathematical concepts. In this chapter, we will deliberately explore their limitations.

2.2 Preparation

This lab will rely on the material from Section 1.2.2 of the Scientific Python Lecture notes.

△ **Jupyter Notebook Exercise 2.1:** Enter the following code into a cell and check the output:

```
a = 121
print(type(a))
print(a)
```

Next, in the same cell, add a line at the end setting a to a real value, `a = 1.34`, and print the type and value again. Check the output. Next, set a to Boolean value, `a = False`, and print the type and value yet again.

In many languages, such as C and C++, variables are strictly typed: you would have to decide at the start whether you want a to be of integer, float, or boolean type, and then you would not be able to change to a different type later. Python variables are references to objects, which means they only point to memory locations that contain objects with all of the data and functionality associated with that object. When you write $a = 121$ it is interpreted as "set variable a to point to a location in memory that contains a class of type integer with the value 121".

△ **Jupyter Notebook Exercise 2.2:** Enter the following code into a cell and check the output:

```
a = 12
b = a
b = 5
print(a)
print(b)
```

Why is the output "12, 5" instead of "5, 5"? When you write `b = a`, the variable b points to the same Integer class that a points to. So when you write "b = 5" why doesn't the value of a change as well? This code snippet shows that it does not. The reason is that Integers are *immutable* objects in python... their values cannot be changed. So when you write `b = 5` it is interpreted as "variable b points to a (new) Integer with value 5." The variable a continues to point to the Integer with value 12. The "is" operator used like this:

```
print(a is b)
```

tells you if a references the same object as b. Add to compiles of this line to your snippet in order to clarify the situation. One call should return True and the other False.

△ **Jupyter Notebook Exercise 2.3:** If I set a variable a to an integer value and I set b to the same integer value, do a and b refer to the same object, or to two different objects with the same value? Write a snippet of code (three lines) to find out.

2.3 Binary Representation of Integers

Computer hardware is based on digital logic: the electrical voltage of a signal is either high or low, which correspond to a mathematical zero or one. A digital clock is used to ensure that signals are only sampled at particular times, when they are guaranteed not to be in transition from a zero to one or vice versa. The Arithmetic-Logic Unit (ALU) uses digital logic gates (such as AND or OR) to perform calculations. For example, it is possible to build an adder that uses only NAND gates.

Because digital signals have only two states (zero and one), the most natural way to represet numbers in a computer is using the base two, which we call binary. In the familiar base ten, we have ten digits (from 0 to 9) and the place value increases by a factor of 10 with each digit moving toward the left. In binary, we have only two digits (0 and 1) and the place value increase by factors of two. A single digit in binary is referred to as a "bit". You add columns quickly when counting in binary: zero(0), one(1), two(10), three(11), four(100), five(101), and so on. For efficient operations, computers often group eight bits together to form a byte. Digital values are therefore commonly represented in hexidecimal (base 16) where two digits of hexidecimal describes one byte. See Table 2.1, which you can produce for yourself in Python like this:

```
print("dec: hex: bin:")
for d in range(16):
    print("{0:<2d}   0x{0:01x}   0b{0:04b}".format(d))
```

It is conventional to preprend binary numbers with "0b" and hexadecimal with "0x" otherwise we wouldn't know whether a "10" represents ten, sixteen, or two! Notice that the largest number that can be written with n bits is $2^n - 1$.

The mathematical notion of an integer can be naturally implemented by computer hardware. Although integers are represented in binary in the hardware, modern compilers and languages generally print them to screen as decimal by default. One caveat is that computers do not have an unlimited number of bits. Many computer languages use 64-bit integers, which means that only the integer values from 0 to 18446744073709551615 can be represented:

```
x = 2**64-1
print(x)
```

Table 2.1: The numbers 0 to 15 in decimal, hexadecimal, and binary.

dec:	hex:	bin:	dec:	hex:	bin:
0	0x0	0b0000	8	0x8	0b1000
1	0x1	0b0001	9	0x9	0b1001
2	0x2	0b0010	10	0xa	0b1010
3	0x3	0b0011	11	0xb	0b1011
4	0x4	0b0100	12	0xc	0b1100
5	0x5	0b0101	13	0xd	0b1101
6	0x6	0b0110	14	0xe	0b1110
7	0x7	0b0111	15	0xf	0b1111

For signed integers, one bit is used to indicate the sign (positive or negative) and so a 64-bit signed integer can represent integer values from -9223372036854775808 to 9223372036854775807. As long as an integer value is within the range covered by the integer type, the integer value can be perfectly represented.

Python uses arbitrary sized integers: it simply adds more bits as needed to represent any number. For an extremely large number, you will eventually reach practical limitations on the amount of memory and processing time available in the computer, which will limit how large of an integer can be calculated.

△ **Jupyter Notebook Exercise 2.4:** See for yourself just how huge integers can be in python by entering:

```
x = 2**8000
print(x)
```

and checking the output.

△ **Jupyter Notebook Exercise 2.5:** Print the integer 64206 in decimal, hexadecimal, and binary. Hint: just reuse the carefully formatted print statement from the example above.

△ **Jupyter Notebook Exercise 2.6:** Suppose you are tasked with rewriting the firmware for a distant satelite which has just lost one line from an eight-bit digital communications bus due to radiation damage. You now have only seven working bits! What is the maximum sized unsigned integer which you could write on this degraded seven-bit bus? What range of signed integers could you write?

△ **Jupyter Notebook Exercise 2.7:** Let's consider a four-bit signed integer, so zero is 0000 and one is 0001. Suppose the upper bit is reserved for sign, so 1XXX is a negative number. An obvious choice for representing -1 would be 1001, but there is a better choice. Consider that:

$$(-1) + 1 = 0$$

Well, if we simply ignore the last carry bit (5th bit):

$$1111 + 1 = 10000 = 0000$$

So if we define 1111 as -1, we can treat addition with negative numbers exactly the same as adding ordinary numbers. Find the representation for -2 such that

$$-2 + 2 = 10000 = 0000$$

then show that:

$$-1 + -1 = -2.$$

Python does not use this trick, but many other languages do.

2.4 Binary Representation of Real Numbers

Representing real numbers presents much more of challenge. There are an uncountably infinite number of real numbers between any two distinct rational numbers, but a computer has only finite memory and therefore a finite number of states. It is impossible for computers to exactly represent every real number. Instead, computers represent real numbers with an approximate floating point representation much like we use for scientific notation,

$$x = m \times B^n$$

where the significand m is a real number with a finite number of significant figures and the exponent n is an integer. The base B is ten for scientific notation but typically two in a floating point representation. The exponent n is typically chosen so that there is only one digit before the decimal point in the base B, e.g. 3.173×10^{-8} for scientific notation.

The limited precision of the discriminant can lead to challenges when using floating point numbers. The floating point precision is specified by the parameter ϵ (epsilon) which is the difference between one and the next highest number larger than one that can be represented. For scientific notation with four significant digits, $\epsilon = 0.001$, because we cannot represent anything between 1.000×10^0 and 1.001×10^0 with only four significant figures.

△ **Jupyter Notebook Exercise 2.8:** Determine the Python floating point ϵ by running

```
import sys
print(sys.float_info.epsilon)
```

△ **Jupyter Notebook Exercise 2.9:** Determine the Python floating point ϵ for yourself by running:

```
eps = 1.0
while eps + 1 > 1:
    eps = eps / 2
eps = eps * 2
```

Here the while loop continues running the indented code until the condition $\epsilon + 1 = \epsilon$ is met.

△ **Jupyter Notebook Exercise 2.10:** Python uses the IEEE 754 double-precision floating-point format. This format uses 64-bits overall, with 52 bits reserved for the significant. The standard uses a clever trick to save one bit, by requiring that the leading bit of the significant m is one, and defining 53 signficant figures using 52 bits:

$$m = 1.m_1m_2m_3...m_{52}$$

Here each m_i represents an individual bit (0 or 1). Predict the parameter ϵ and compare with the above.

△ **Jupyter Notebook Exercise 2.11:** It seems like ϵ should be small enough to simply ignore it in most cases, but in fact it shows up quite clearly if you apply strict equality to floating point quanties. To see the problem, run this code, checking if $\sin(\pi)$ is zero:

```
x = np.sin(np.pi)
print(x)
print(x==0)
```

Devise an alternative condition to `x==0` that properly accounts for floating point precision.

△ **Jupyter Notebook Exercise 2.12:** Here is another case where floating point precision joins the chat uninvited:

```
x = 0.1
y = x+x+x
print(y == 0.3)
```

Devise an alternative to `y==0.3` that properly accounts for floating point precision.

△ **Jupyter Notebook Exercise 2.13:** Personally, I prefer my zero's to look like zero. When floating point limitations are making them look non-zero, I like to clean then up with rounding, like this:

```
x = np.sin(2*np.pi)
print(x)
x = np.around(x,15)
print(x)
```

Yeck! What is -0??!! IEEE 754 defines two zeros -0 and 0. -0 is used to indicate that 0 was reached by rounding a negative number. This is so that $1/-0$ can be interpreted as $-\infty$ and $1/0$ as $+\infty$. If you just want to make this go away, add "+0":

```
x = np.around(x,15)+0
```

Show that this works.

△ **Jupyter Notebook Exercise 2.14:** Consider the following code:

```
a = 5;
b = 1.2343E-17;
sum = 0
sum += 5;
for i in range(1000000):
    sum = sum + b
print(sum)
```

Is there any problem here? If there is, fix it by changing only the *order* of the lines of code.

2.5 Other Data Types

Integers and floating point numbers are the real work horses of computational physics. We'll add numpy arrays in a future lab. This section will briefly introduce the remaining types.

Python includes strings as a basic type:

```
s = "hello world"
s = s + " (it's been a strange few years)"
print(s)
print(type(s))
print(s[6],s[4],s[6])
print(type(s[0]))
```

Strings are *immutable* objects that contain textual data. If you have used other languages, you might expect `s[0]` to be a "character" but in Python it is a string of length one. There is no built-in character type.

Python includes complex numbers as a basic type:

```
z = 1 + 2j
print(type(z))
print(z.real)
print(z.imag)
print(z.conjugate())
```

This is our first example of an object oriented programming (OOP) class interface. To compute the complex conjugate of z, an ordinary function would need to be passed z as an argument, or else it would not know which complex value to use for the computation. But `z.conjugate()` is a *method* of the class complex. The method is tied to the instance of complex number z by the "." and has access to all of the data it needs from z. Similarly, the `real` and `imag` are member data of class complex: they are the integers that contain the real and imaginary parts of z. Objects play a central role in Python, but in a refeshingly understated and reserved manner. It is enough for now to understand that `z.conjugate()` is much like a function that already has z as a parameter, and `z.imag` and `r.real` are just ways to access the data contained in z.

△ **Jupyter Notebook Exercise 2.15:** Define a complex number $1 + i$ and multiply it by it's complex number. Show that the resulting complex number has zero for it's imaginary part.

Python provides Lists as a native container of python objects. We'll make much more use of numpy arrays, which are better suited to numerical analysis, but Python Lists occassionally play a role for various bookkeeping tasks:

```
L = ["hello", 1, 2, 3+2j, 3.45, "green"]
print(L[0])
print(L[1])
print(L[5])
L[0] = "goodbye"
print(L)
```

Here the List L contains a variety of (admittedly rather useless) objects. These objects can be referred to individually by their index. One place where lists really shine is in looping over a custom list of values:

```
for i in [1,5,10,50,100]:
    print(i)
```

△ **Jupyter Notebook Exercise 2.16:** Run the following code:

```
a = [1,2,3,4,5]
b = a
b[0] = 1
print(a[0])
print(a is b)
```

(Spits out coffee) "What the???!!" Lists are *mutable* which makes them fundamentally different from *immutable* integers. Here we assign b to point to the same object as a (a list) and then change an entry in that list. Even after the change, a and b point to the same object. This is only possible because the list object is mutable.

△ **Jupyter Notebook Exercise 2.17:** It's good to read the documentation, but it's a useful skill to figure things out for yourself too! Without looking up the documentation, write a snippet of code to determine for yourself if complex numbers are mutable (like lists) or immutable (like integers). (It's OK if your code throws an error here, but you can also comment it out if you are the sort of person that can't possibly leave it alone)

Chapter 3

Sequences and Series

3.1 Introduction

In this lab, we will apply for loops to study sequences and series. If you already have programming experience, you can complete the challenge problem in lieu of the other problems.

3.2 Preparation

This lab will rely on the material from Sections 1.2.1 to 1.2.4 of the Scientific Python Lecture notes. Most of the problems can be completed using a simple functions containg a single for loop, such as in this function:

```
  def loop(n):
    for i in range(n):
        print(i)
```

To run the code in the function, you call function, usually in a different cell:

```
loop(5)
```

△ **Jupyter Notebook Exercise 3.1:** Create a new function:

```
def powers(a,n):
   # your code here ...
```

that prints the first n powers of a. For example `mult(3,4)` should output:

```
1
3
6
9
```

In future problems, we'll describe this output simply as 1, 3, 6, 9. We won't be picky about whitespace unless we discuss it explicitly. One way to complete this is to use the three options of `range(start,stop,step)`.

3.3 Fibonacci Sequence

The Fibonacci numbers are a sequence of numbers satisfying the recursion relationship:

$$F_{n+2} = F_n + F_{n+1}$$

with $F_0 = 0$ and $F_1 = 1$. The sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

This sequence can be generated numerically by an algorithm such as this one:

```
fa := 0 # set fa to 0
fb := 1 # set fb to 1
repeat n times:
    fc := fa + fb
    print fc to screen
    fa := fb
    fb := fc
```

Note that this is not python syntax. What is the importance of the last two lines? Would the algorithm work if we exchanged their order?

△ **Jupyter Notebook Exercise 3.2:** Use the algorithm described above to implement a new function `fib(n)` which prints the next n Fibbonacci numbers after the initial 0 and 1.

3.4 Arithmetic Series

The finite arithmetic series

$$S_n = \sum_{k=0}^{n} (a + kd) = a + (a + d) + (a + 2d) + \ldots + (a + nd)$$

sums to the average of the first and last terms times the number of terms:

$$S_n = (n + 1)\frac{a + (a + nd)}{2} \tag{3.1}$$

We will assume $a = d = 1$ and calculate this finite series numerically using the following function:

```
def arith(n):
    sum = 0
    for j in range(1,n+1):
        sum = sum + j
        #print("j: ", j, "\t sum: ", sum)
    return sum
```

Type in this function and see how it works by uncommenting the print statement (delete the # symbol that starts a comment) and calling it as `arith(5)`. The use of print statements in a loop like this or at each stage of a calculation is a simple, effective and classic debugging technique. You test your code with the print statements included, keeping n small so you don't fill your whole screen with output. Once your code is working, you comment out the unneeded print statements so that the interpreter ignores them and you no longer see the unneeded output. Why not

just delete them? You can, but experience shows that if you do, you will need the line again shortly!

△ **Jupyter Notebook Exercise 3.3:** Obtain the sum of the first n terms of arithmetic series with `sum = arith(n)` for three different values of n. Each time, show that sum returned by the function matches the expected sum.

3.5 Geometric Series

The geometric series

$$\sum_{k=0}^{\infty} ar^k = a + ar + ar^2 + ar^3 + \dots$$

converges for $|r| < 1$ to:

$$\frac{a}{1-r}. \tag{3.2}$$

We will demonstrate this numerically.

△ **Jupyter Notebook Exercise 3.4:** Implement a function `geom(a,r,n)` which calculates sum of the first n terms of the geometric series with kth term ar^k. Show that it agrees with Eqn. 3.2 for $a = 2$, $r = 0.5$ $n = 100$.

△ **Jupyter Notebook Exercise 3.5:** Call you geometric series function again for $a = 3$, $r = 0.8$ and $n = 100$. Compare with the expected output calculate within python and with pencil and paper. Do they agree exactly? If not, do they agree within the floating point precision?

△ **Jupyter Notebook Exercise 3.6:** Now compare your calculated sum with Eqn. 3.2 for $a = 1$, $r = -0.9$ $n = 100$. How is the agreement? Increase n and see what happens. Why do you suppose this series is slower to converge?

3.6 Refinements

There are a few refinements you can make to your code. Don't change your working code from previous examples! Instead, copy the previous version to a new cell and make your refinements there. You don't even need to change the name of the function, Python will happily overwrite the old function implementation when it reaches the cell with the new version. Make these code improvements:

△ **Jupyter Notebook Exercise 3.7:** (Optional) Improve your Fibbonacci function so that prints the first n numbers including the initital two numbers "0" and "1". Make sure it works properly for $n = 0$, $n = 1$, $n = 2$, and so on.

△ **Jupyter Notebook Exercise 3.8:** (Optional) Extend the Arithmetic series function to include parameters a and d. Show that it works.

3.7 Fibonacci Integer Right Triangles

Starting with the number 5, every second Fibonacci number is the length of the hypotenuse of a right triangle with integer sides. The first two are:

$$5^2 = 3^2 + 4^2$$

and

$$13^2 = 5^2 + 12^2.$$

Furthermore, from the second triangle onward, the middle side is the sum of the lengths of the sides of the previous triangle, for example:

$$12 = 3 + 4 + 5.$$

△ **Jupyter Notebook Exercise 3.9:** (Optional Challenge) Use numerical methods to explicitly verify these properties for the first n Fibonacci integer right triangles.

If you would prefer, you may submit the Optional Challenge problem plus the problems from Section 3.5 to complete the assignment.

Chapter 4

The Quadratic Equation and Prime Numbers

4.1 Introduction

In this lab, we will make more extensive use of conditional statements to implement algorithms which solve the quadratic equation, identify prime numbers, and add fractions. An optional challenge problem, The Lucky Number of Euler, explores how the quadratic equation can generate prime numbers.

4.2 Preparation

This lab will rely on the material from Sections 1.2.1 to 1.2.4 of the Scientific Python Lecture notes. We'll now be making frequent use of conditional statements:

```
def compare(a,b):
    if (a==b):
        print("a equals b")
    elif (a<b):
        print ("a is less than b")
    else:
        print ("a is greater than b")
```

Notice that Python uses `==` for comparison. You will get a syntax error if you use `a=b` instead.

We'll also use of the modulo operator % extensively. The modulo operation $a\%b$ returns the remainder from the integer division a/b.

```
# is b a factor of a?
def isfactor(a,b):
    if (a%b == 0):
        return True;
    return False
```

Why does `a%b == 0` mean that b is a factor of a?

△ **Jupyter Notebook Exercise 4.1:** Consider this verbose code snippet:

```
for i in range(100):
    print("on index ", i)
```

Which prints the current index on every iteration. Use the modulo operator to modify the code so that it only prints the index every 10 iterations. This is a classic trick!

We will also be using while loops, which repeat a block of code until a condition is met:

```
count=0
while(count<10):
    print(count)
    count = count+1
```

4.3 Quadratic Formula

The Quadratic equation:

$$ax^2 + bx + c = 0$$

has solutions which are given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{4.1}$$

The number of unique real solutions depends on the quantity in the square root, which is called the discriminant:

$$b^2 - 4ac$$

If this is positive there are two real solutions, it it is one there is one real solution, and if it is negative there are no real solutions.

The solution to the quadratic equation can be calculate as follows:

```
D := b^2 - 4ac
if (D<0):
    print no solutions
if (D>0):
    calculate both solutions from quadratic formula
    print both solution
if (D=0):
    calculate the single solution from quadratic
    print single solution
```

Since D is calculated from parameters a, b, and c which might be floating point, we need to take some care to evaluate the condition that `D==0` within the precision of the floating point operation.

A test case with one real solution is:

$$(x-1)(x-1) = x^2 - 2x + 1.$$

A test case with two real solution is:

$$(x-1)(x+1) = x^2 - 1.$$

A test case with zero real solutions is:

$$(x-i)(x+i) = x^2 + 1.$$

△ **Jupyter Notebook Exercise 4.2:** Implement a function `quad(a,b,c)` which reports the solutions to the quadratic equation and verify it with the test cases shown.

△ **Jupyter Notebook Exercise 4.3:** Calculate three more test cases with integer solutions and use them to test your function more thoroughly.

△ **Jupyter Notebook Exercise 4.4:** A particular tricky test case is a=.1 b=.3 c =0.225 which should have only one real solution. Test you function against this test case.

4.4 Prime Numbers

A prime number is a number that has two and only two factors: itself and one. One is not prime, but two is. We can determine if a number a is prime as follows:

```
if (a<2):
    return false
i := 2
while (i ≤ √a):
    if (a%i=0):
        return false
    i := i + 1
return true
```

Why is there no need to check for factors larger than $\sqrt{a}$?

△ **Jupyter Notebook Exercise 4.5:** Implement a function `isprime(a)` which returns `True` if the integer a is prime and `False` if not.

Suppose that next we want to find the first n prime numbers greater than or equal to a number A. We can simply check if A, $A+1$, $A+2$, and so on are prime until we find the first n. But we do not know how many numbers we will have to check before finding n that are prime. This is a case for a `while` loop.

△ **Jupyter Notebook Exercise 4.6:** Find the first n prime numbers greater than A using a while loop and your `isprime` function. Test it for $n = 10$ and $A = 0$, then for $A = 1000000000$. Try that with paper and pencil!

Notice how we broke this problem of finding primes into two parts: determining whether or not a number is prime or not, then testing and counting prime numbers. We thoroughly tested the first part before using it in the second. This is an essential approach to solving computational problems: break complicated tasks down into smaller tasks which can be tested separately.

△ **Jupyter Notebook Exercise 4.7:** Implement a function which computes the fraction

$$\frac{n}{d} = \frac{a}{b} + \frac{c}{d}$$

from integer inputs a,b,c and d and returns integers n and d. Returning $n > d$ is allowed, but n/d should be a simplified fraction with a greatest common factor of one. You can return two integers as a Tuple, like this:

```
# function which  adds fractions
def addfrac(a,b,c,d):
    n=d=0
    #your code...
    return n, d

# calling function:
n,d =addfrac(1,2,1,3)
print("{0}/{1}".format(n,d))
```

For full credit, you must factorize (see what I did there?) this problem into two parts: your `addfrac` funtion should call a second function that does one well defined task.

4.5 The Lucky Number of Euler

Euler discovered that the formula:

$$k^2 + k + 41$$

produces prime numbers for $0 \leq k \leq 39$. Perhaps you can beat Euler at his own game!

Consider quadratics of the form:

$$k^2 + ak + b$$

For each integer value of a and b, there is a maximum number n such that the quadratic formula produces prime numbers for $0 \leq k < n$

△ **Jupyter Notebook Exercise 4.8:** Find the the values of a and b which produce the largest number of prime numbers. Restrict yourself to $|a| \leq 1000$ and $|b| \leq 1000$.

Chapter 5

Arrays and Plotting

5.1 Introduction

5.2 Preparation

This lab will rely on the material from Sections 1.4.1 to 1.4.2 and 1.5.1 to 1.5.2 of the Scientific Python Lecture notes. This is the first lab that relies on inline plotting, so make sure you are starting your notebook with the "line magic":

```
%pylab inline
```

This will load the numpy library as np, the matplotlib.pyplot library as plt, and setup the matplotlib backend to imbed plots in your notebook.

A Numpy array is a grid of values. Unlike Python lists, the elements of a numpy array all have the same data type, which makes them much more computionally efficient. The numpy library provides a wide range of analysis tools that are mostly centered on the numpy array type. Numpy arrays can be constructed from a Python list:

```
a = np.array([1.3,7.2,4.1,0.0])
b = np.array([[1,2],[3,4]])
print(a)
print(b)
print(np.shape(a))
print(np.shape(b))
```

or they can be constructed from numpy function designed for the purpose:

```
a = np.linspace(0,1,11)
print(a)
b = np.arange(0,5,1)
print(b)
```

5.3 Plotting with Scentific Python

Basic plotting in Python requires two numpy arrives: one for the x coordinates and one for the y coordinates. Consider the following very simple plot:

```
x = np.array([0.0, 1.0, 2.0, 3.0, 4.0,  5.0])
y = np.array([0.3, 3.2, 5.8, 9.0, 12.4, 14.7])
```

```
plot(x,y,"bo")
```

Here, the "bo" options specifies blue circles. Now consider:

```
x = np.linspace(0, 1, 100)
y = np.sin(np.pi*x)
plt.plot(x,y,"r-")
```

Here the "r-" option specifies red line. Including 100 points (as done here) results produces a smooth looking curve.

Now promise me that you will never make another plot without labeling the x and y axes! Here's another example will all the bells and whistles you need to make a professional looking plot:

```
UPPER = 2
LOWER = 0
tau   = 2*np.pi
x = np.linspace(LOWER,UPPER,100)
s = np.sin(tau*x)
c = np.cos(tau*x)
plt.plot(x,s,"b-",label="sin")
plt.plot(x,c,"r-",label="cos")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Two Periods of a Sine and Cosine")
plt.legend(frameon=False)
plt.show()
```

Make sure you understand all of the features demonstrated here:

- Variables `UPPER` and `LOWER` located at the top of the snippet, allowing for easy adjustment of parameters that affect the plot.
- Use of `np.linspace` to define an array of x values, with plenty of them (100) to produce nice smooth curves.
- Creation of two different arrays of y values, one for sin and one for cos.
- Plotting the arrays of x and y values with `plt.plot` using the "-" option for a line and color blue("b") for sin and red("r") for cos.
- Defining appropriate axis labels with `plt.xlabel` and `plt.ylabel`.
- Adding a title with `plt.title`
- Creation of a legend using the `label` optional argument to `plt.plot` and the plot.legend() command. Removing the frame with option `frameon=False`

It is written so concisely and intuitively, you might not even notice what is going on with the line:

```
s = np.sin(tau*x)
```

Remember that x here is a numpy array of 100 elements. The `tau*x` multiples every element of x by our value tau. The `np.sin(tau*x)` then takes the sine of each element. The resulting numpy array, also of 100 elements, is referenced by variable s. Each element of s contains $\sin(\tau x)$ for the corresponding element of the array x. It takes some getting used to for programmers used

to explicitly writing for loops for things like this, but ultimately, the fact that python handles so much of this bookkeeping for us is what makes it a very fun language to work with.

△ **Jupyter Notebook Exercise 5.1:** △ **Jupyter Notebook Exercise 5.2:** Plot the sinc function as a smooth line in the x range from -5 to 5. Add appropriate axes labels. Include a legend identifying the sinc function. For the line color, use any color other than red or blue.

5.4 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. 5.1, 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.

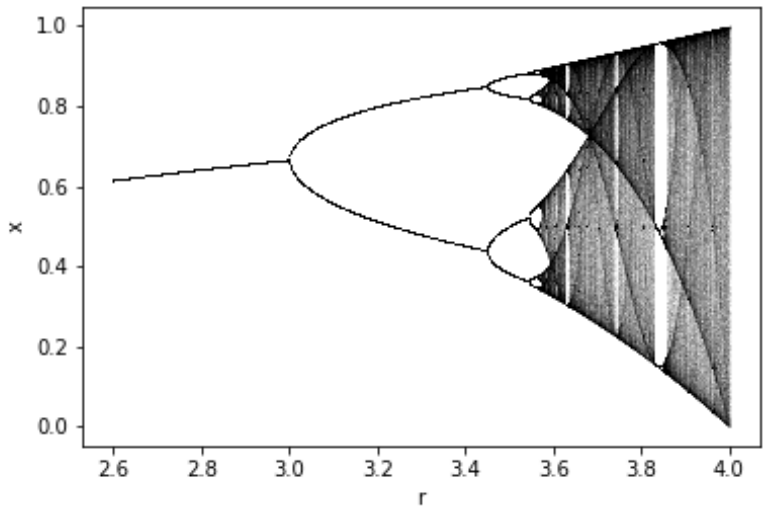


Figure 5.1: Long term behavior of the logistics map.

The long term behavior of the logistics map can be easily modeled in Scientific Python. A start is shown in Fig. 5.2 where you should understand:

- An array of r values is defined.

```
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')
```

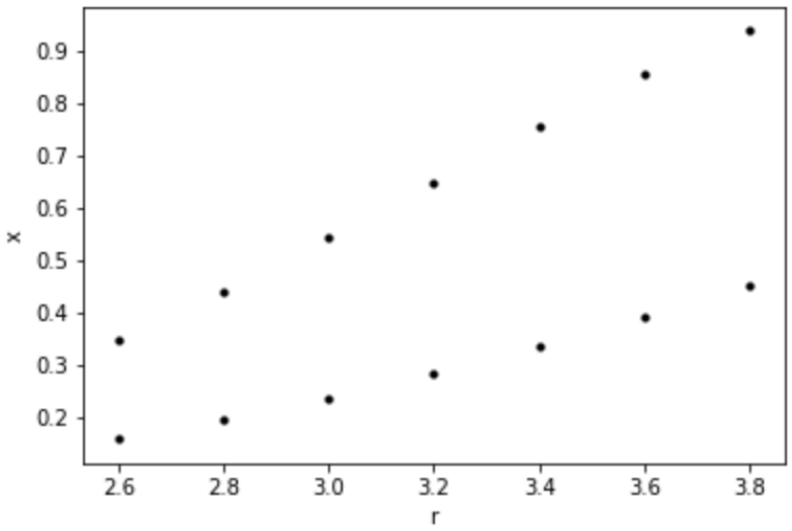


Figure 5.2: Modeling the logistics map.

- 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 Exercise 5.3:** Reproduce the figure in Fig. 5.1 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`.

Chapter 6

Differentiation and Projectile Motion

6.1 Introduction

6.2 Preparation

Our codes are getting complicated enough that we will need to pay some attention to variable scope, as illustrated here:

```
i=1
j=2
k=3
def f(i,j):
    print(i,j,k)
f(i,j)
f(j,i)
```

Try to predict the output of this snippet before running it. The first three lines define integers i,j, and k. These have global scope, which means they can be accessed from anywhere. The function `f(i,j)` has parameters i and j which have a scope limited to the function f. Even though they have the same name as the global variables i and j, they are independent quantities. Within the function f the variable i is the first parameter, and j is the second parameter. Because they have the same name, the global variables i and j are *shadowed* by the local parameters i and j. The global variable k is not shadowed.

When `f(j,i)` the paramter i is set to the global variable j, and the parameter k is set to the global variable i. There is no parameter k only global variable k. This is why the output from the call is "2 1 3".

In this lab, we will be passing a function as an argument to another function, as in this simple example:

```
def show(f):
    print(f(2))
    print(f(3))

def f(i):
    return i**2

def g(i):
    return i**3
```

```
show(f)
show(g)
```

Here the `show` function takes another function `f` as an argument. Within the show function, the function `f` is called using paranthesis just like any other function, as in `f(2)` We define two additional functions, `f` which returns the square of its argument, and `g` which returns the cube. When `show(f)` the output 4 and 9. When `show(g)` the output 8 and 27. Run the code as is, and also check what happens if you define `g(i)` to require a second argument as in `g(i,j)`.

6.3 Numerical Differentiation

In lecture we derived the right derivative (aka foward derivative) formula

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

for numerically determining the derivative of the function f. Remember we do not calculate the $\mathcal{O}(h)$ term, that indicates that the truncation error is of order h. We also derived centered derivative formula:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

To evaluate a derivative using any of these formulas, we need to choose an appropriate value of h. If h is too large, the truncation error (the amount the estimated value differs from the actual value) will dominate. If h is too small, floating point precision will likely be the limit.

△ **Jupyter Notebook Exercise 6.1:** Implement the right derivative formula as `right(f, x, h)` where f is the function to be evaluated, x is the location to evaluate the derivative, and h is the step size for the numerical integration. Check you code on several functions with known derivatives, like this:

```
def f(x):  # derivative 0
    return 2
def g(x):  # derivative 3
    return 3*x
def h(x):  # derivative 4x
    return 2*x**2

print(right(f,1,0.01))
print(right(g,1,0.01))
print(right(h,1,0.01))
```

△ **Jupyter Notebook Exercise 6.2:** Implement the center derivative function as `center` and test it in the same manner as in the previous exercise for `right`.

△ **Jupyter Notebook Exercise 6.3:** Compare the performance of `right` and `center` like this:

```
def f(x):  # derivative 6x**2
    return 2*x**3

print("right:", right(f,1,0.1),   "center:", center(f,1,0.1))
print("right:", right(f,1,0.01),  "center:", center(f,1,0.01))
print("right:", right(f,1,0.001), "center:", center(f,1,0.001))
```

Recall that the truncation error goes as h for the right derivative and as h^2 for the center derivative. Are these results consistent with that expecation?

From now on, we will use the center derivative function only due to its better performance. We can plot the derivative a function like this:

```
def f(x):
    return 0.5*x**2

x = np.linspace(0,1,100)
y = center(f, x, 0.1)
plt.plot(x,ya,"-b")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Notice how the argument x passed to the function `right(f,x,h)` and then to `f(x)` is now a numpy array of 100 values from 0 to 1. The derivative is now evalued at 100 places with a single call.

△ **Jupyter Notebook Exercise 6.4:** Define $f(x) = x^3$. Use your `center` function to evaluate it's derivative $f'(x)$ in the x range $[-2, 2]$. Plot both $f(x)$ and $f'(x)$ in that range (in the same plot) using different colors for each. Add a legend and axis labels.

△ **Jupyter Notebook Exercise 6.5:** Define $f(x) = sin(x)$. Use your `center` function to evaluate it's derivative $f'(x)$ in the x range $[0, 2\pi]$. Plot $f(x)$, $f'(x)$, and $cos(x)$ in that range (all in the same plot) using different colors for each. Add a legend and axis labels.

6.4 Projectile Motion

△ **Jupyter Notebook Exercise 6.6:** Check you implementation against the following test values:

```
print(np.around(euler(0.134, 0.659, 0.282, 0.662, 0.643, 0.900, 0.451),2))
print(np.around(euler(0.924, 0.959, 0.575, 0.299, 0.710, 0.699, 0.471),2))
```

and ensure you get the corret output:

```
[0.75 0.37 0.78 0.7 ]
[1.24 1.23 0.94 1.15]
```

```
tau = 2*np.pi
vi    = 20    # [m/s]
g     = 9.8   # [m/s^2]
theta = tau/8
dt = 0.01 # [s]
x  = 0    # [m]
y  = 0    # [m]
vx = vi*np.cos(theta)
vy = vi*np.sin(theta)
```

```
tjx = np.array([x])
tjy = np.array([y])
while(y>=0):
    x,y,vx,vy = euler(dt,x,y,vx,vy,0,-g)
    tjx = np.append(tjx,  x)
    tjy = np.append(tjy,  y)
```

```
plt.plot(tjx, tjy)
plt.xlabel("x [m]")
plt.xlabel("y [m]")
plt.show()
```

△ **Jupyter Notebook Exercise 6.7:** Simulate projectile motion using your function `euler` as described above.
△ **Jupyter Notebook Exercise 6.8:** Extend your simulation to record vx and vy at each step along with the x and y positions. Take the mass of the projectile to be $m = 0.145$ kg and plot the kinetic energy, potential energy, and total energy as a function of time. To build an array containing the time of each step for plotting quanties against time, you can do:

```
t = np.arange(tjx.size)*dt
```

Include a legend. The x and y axes have changed to time and energy, so make certain to change the axes labels.

6.5 Projectile Motion with Drag

We can model drag as a deceleration:

$$\vec{a} = -k|\vec{v}|\vec{v}$$

where $k = 0.00622$ m$^-$1 for typical baseballs .
△ **Jupyter Notebook Exercise 6.9:** Extend your simulation to include the effect of drag. Plot the trajectory without drag and the trajectory with drag in the same plot. Include a legend and (as always) label all axes.
△ **Jupyter Notebook Exercise 6.10:** Plot the kinetic energy, potential energy, and total energy as a function of time, just as before. Is total energy conserved?

Chapter 7

Pendulum Motion

7.1 Introduction

7.2 Pendulum Motion

Chapter 8

Orbital Motion

8.1 Introduction

8.2 Orbital Motion

Chapter 9

Integration

9.1 Introduction

9.2 Integration

9.3 Gaussian Distribution

9.4 Eliptic Function

Chapter 10

The Monte Carlo Method

10.1 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.

10.2 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:

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

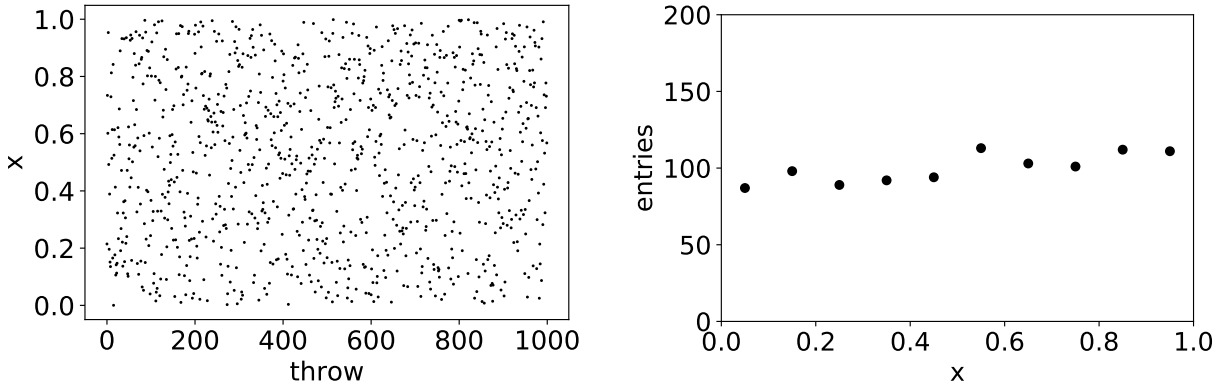
Recall that $x \mod y$ (coded as `x % y` in python) is the remainder after integer division $x//y$. Each I_n is called a seed, and the initial seed I_0 must be provided e.g. by the user. Notice that the seeds are all integers in the range from 0 to $(M-1)$. If we wish to convert these seeds into a random 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 Exercise 10.1:** 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.

10.3 Visualizing distributions



(a) (b)

Figure 10.1: The (a) x value of uniform random throws versus throw number and (b) corresponding histogram.

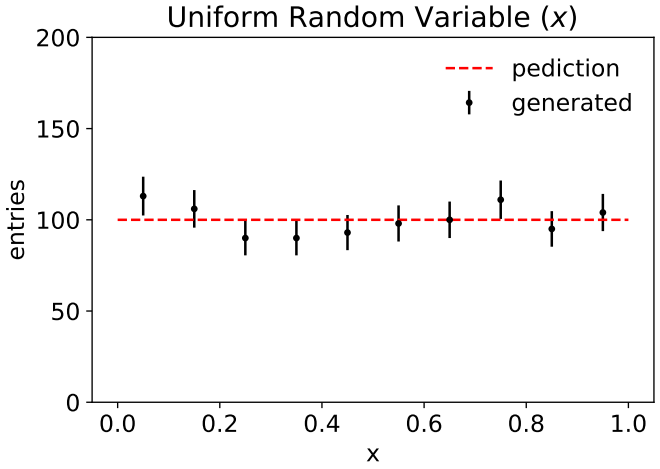
How can we verify that our Park-Miller random number generator produces uniform random numbers in the range 0 to 1? The associated PDF is just $p(x) = 1$, which we know how to plot, but how can we compare this function to a sequence of numbers like $[0.21, 0.85, 0.33, ...]$? An initial attempt might look like Fig. 10.1a, 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. 10.1b 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. Error bars are conventionally drawn with a size equal to the standard deviation of the measured value, σ. Each histogram contains a *count* n. As we will see in lecture, this count is drawn from a Poisson distribution, and our best estimate for the standard deviation σ associated with a count n is simply $\sqrt{n}$. So when drawing a histogram,

the uncertainty in each bin is simply 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 10.2: 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. 10.2. 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 Exercise 10.2:** 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 `#`).

10.4 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. 10.3.

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 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. 10.4 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.

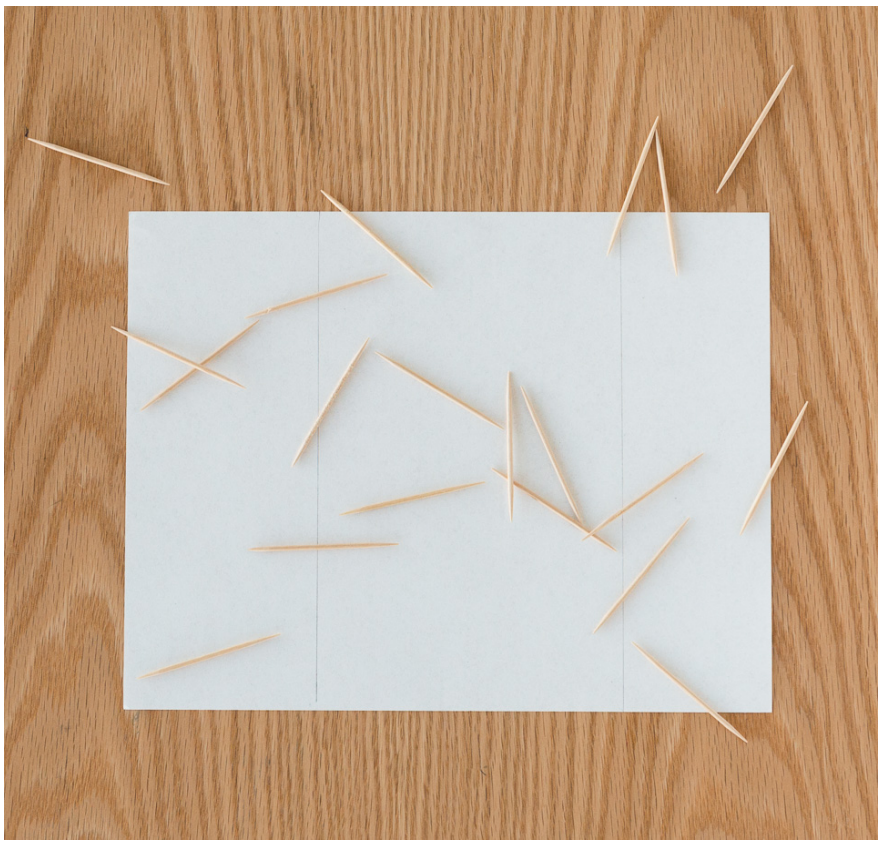
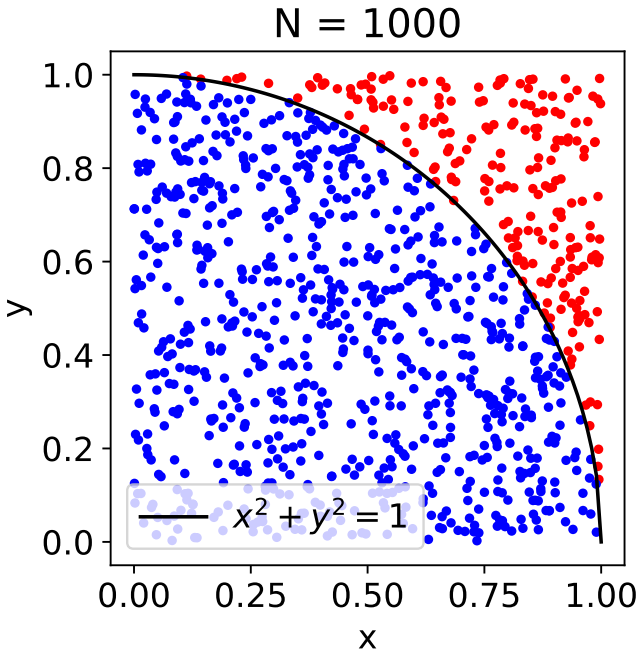


Figure 10.3: Determining π by throwing toothpicks.




```
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 circule
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 10.4: Monte Carlo Determination π .

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 Exercise 10.3:** 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 Exercise 10.4:** 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?

10.5 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:

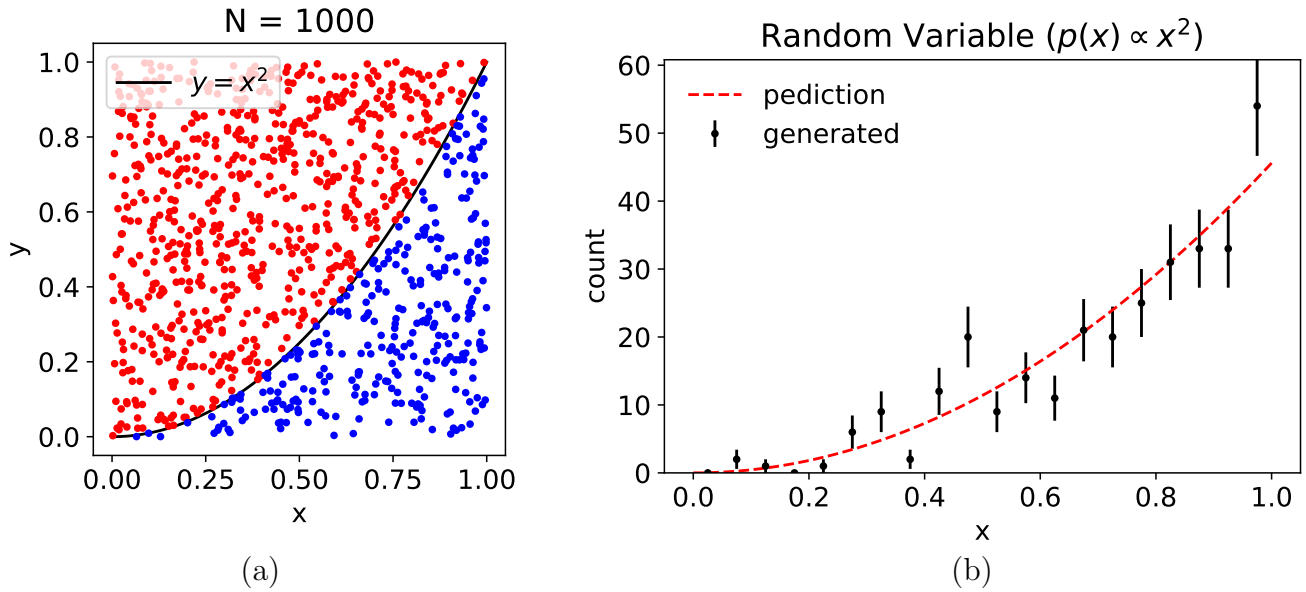
- Instead of thowing x in $[0, 1]$, throw θ in $[0, \pi]$. This means the area of the rectangle A is now π instead of 1.
- Count the number of throws that land below the integral $y < \sin^2\theta$.
- Determine the area under the curve as the fraction of the throws under the curve times the total area 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 Exercise 10.5:** 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. 10.4 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.

10.6 The Rejection method



(a) (b)

Figure 10.5: 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. 10.5 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. 10.5 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. 10.5b, 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 Exercise 10.6:** 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. 10.5a showing your selected points in blue, your rejected points in red, and the selection function ($p(x) = x^2$).

△ **Jupyter Notebook Exercise 10.7:** 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. 10.5b 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.

10.7 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:

$$\int_{-\infty}^{x} p(x)\, dx = y \tag{10.1}$$

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. 10.1. 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 Exercise 10.8:** 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. 10.5b 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.

10.8 Particle diffusion

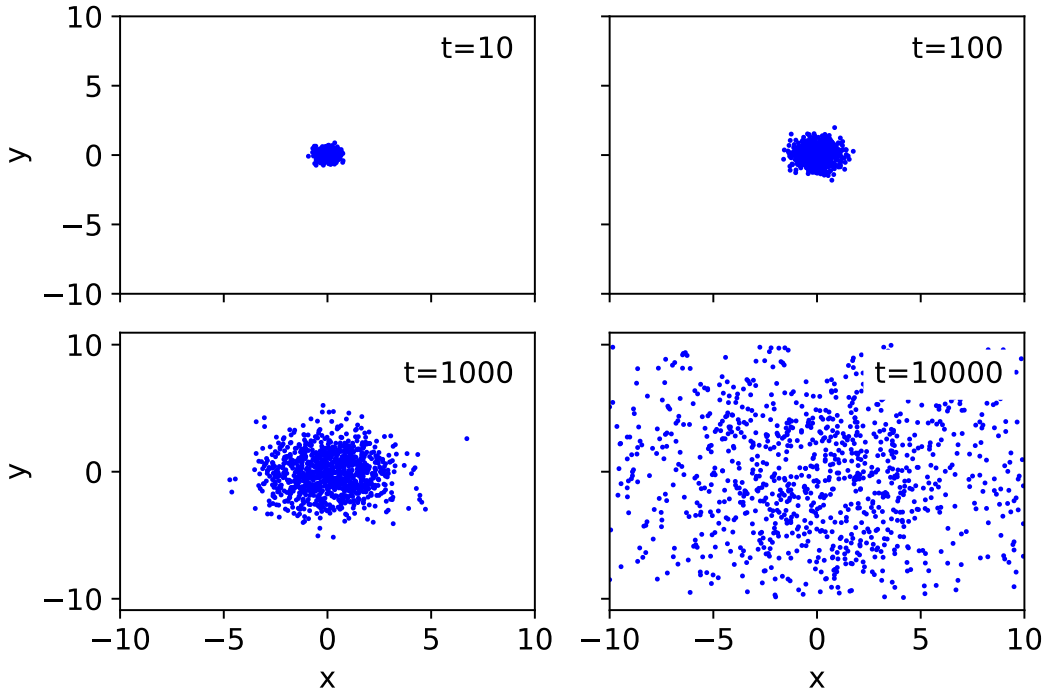


Figure 10.6: Simulation of the diffusion of a drop of particles at four different times.

In this section, we will model the diffusion of a drop of particles in a medium, as in Fig. 10.6, shown as snapshots at four different times. The starting point for the simulation, at $t = 0$ is shown in Fig. 10.7 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.

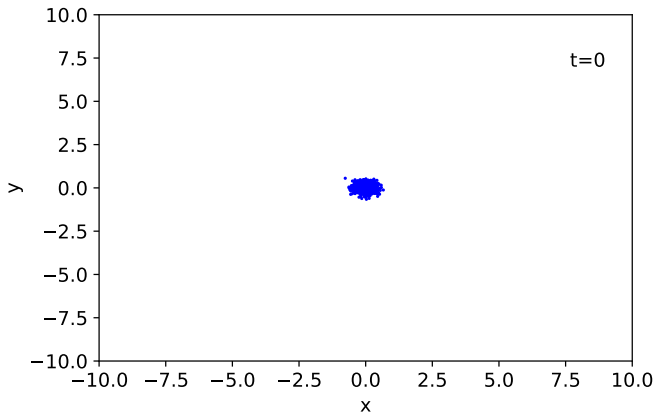
The diffusion process is modeled by a random walk. During each update (for one time step) the x and y values of each particle should be randomly increased or decreased by an amount `STEP=0.2` Any particles that would leave the boundaries of the region $[-L, L]$ as a result should be moved back into the 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 10.7: 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 Exercise 10.9:** 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 Exercise 10.10:** Calculate and record the entropy of the system as it evolves, and plot the entropy as a function of time.

Chapter 11

Simulation of an Ideal Gas

11.1 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:

$$P(v_x) = \sqrt{\frac{m}{2\pi k_{\rm B} T}} \exp\left(-\frac{m v_x^2}{2 k_{\rm B} T}\right) \tag{11.1}$$

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

$$P(v_y) = \sqrt{\frac{m}{2\pi k_{\rm B} T}} \exp\left(-\frac{m v_y^2}{2 k_{\rm B} T}\right). \tag{11.2}$$

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:

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

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:

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

Integrating over all possible directions θ, we obtain:

$$\begin{aligned} P(v) &= \int_0^{2\pi} P(v, \theta) d\theta \\ &= \int_0^{2\pi} \frac{mv}{2\pi k_{\rm B} T} \exp\left(-\frac{m v^2}{2 k_{\rm B} T}\right) \\ P(v) &= \frac{mv}{k_{\rm B} T} \exp\left(-\frac{m v^2}{2 k_{\rm B} T}\right) \end{aligned} \tag{11.3}$$

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.

11.2 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.

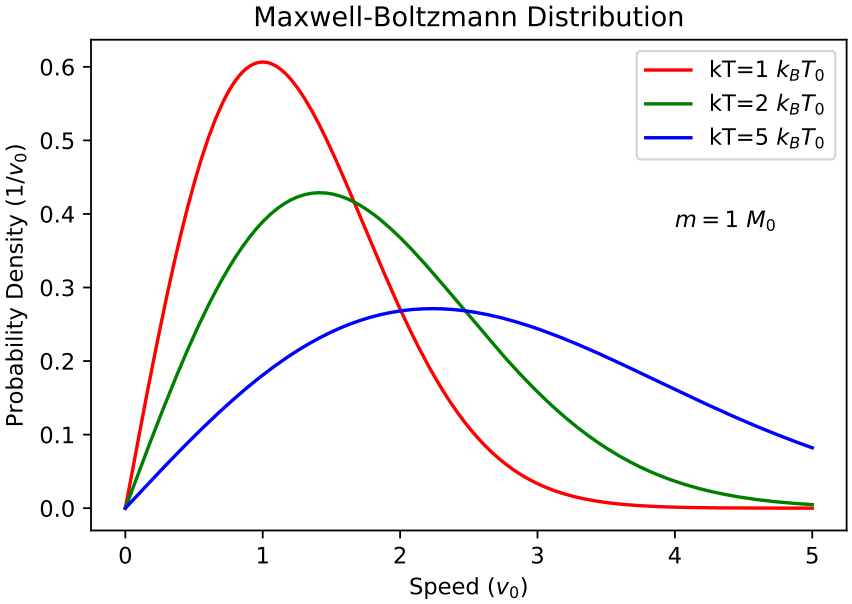
To start, we choose a reference temperature near the temperature we would like to simluate, say $T_0 = 293$ K. All temperatures in the simulation will be in units of this reference temperature. So a temperature `T=1.2` in the program will be $1.2\,T_0 = 352$ K in SI units. Our model also includes mass, 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. 11.1 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 11.1: The Maxwell-Boltzmann distribution using a system of units appropriate for a numerical simulation, along with the code used to produce the plot.

11.3 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. 11.2, 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:

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

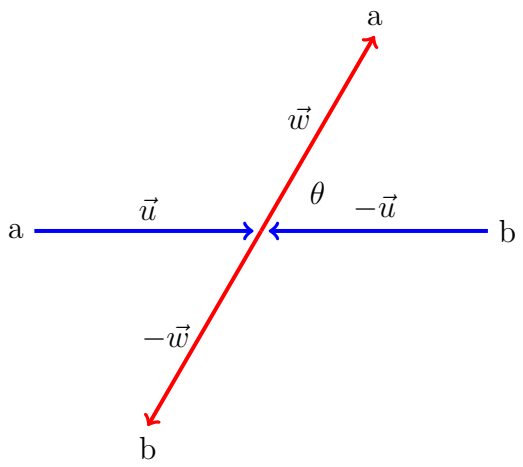


Figure 11.2: The collision model in the center-of-mass: incoming molecule a with velocity $\vec{u}$ collides with the incoming particle b of identical mass with velocity $-\vec{u}$. Particle a is scattered by angle θ and leaves with velocity $\vec{w}$, while particle b leaves with velocity $\vec{w}$. The magnitude of the final and initial velocities are the same: $|\vec{u}| = |\vec{w}|$.

11.4 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:

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

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. 11.3. This function takes as input the velocity components `ax`, `ay`, `bx`, `by` as defined above plus the scattering angle `theta`. In Fig. 11.3, the function simply returns the velocity components unchanged. You should modify the function to implement the scattering algoirthm described above.

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

Figure 11.3: 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.

```python
# 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.]
```

```python
#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 11.4: Example collisions along the x and y axis.

△ **Jupyter Notebook Exercise 11.1:** Implement the collision algorithm as a function as in Fig. 11.3 and test it using example collisions from Fig. 11.4.

When testing your code, start with easy, special cases, such as used in Fig. 11.3. 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 Exercise 11.2:** Test your collision algorithm using the example collisions from Fig. 11.5.

```
# 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 11.5: More complicated example collisions.

11.5 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 Exercise 11.3:** 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 11.2 because it has not yet reached thermal equilibrium.

11.6 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 about 10 times as many collisions as gas molecules in order to reach thermal equilibrium.
△ **Jupyter Notebook Exercise 11.4:** 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 11.2.

11.7 Temperature of an Ideal Gas

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

$$k_b T = m \frac{\langle v_x^2 \rangle + \langle v_y^2 \rangle}{2} \tag{11.4}$$

You can estimate $\langle v_x^2 \rangle$ from your simulation as `np.mean(vx**2)`.
△ **Jupyter Notebook Exercise 11.5:** Estimate kT of the gas using Equation 11.4 before and after simulating collisions. The values should remain near the expected value 4/3.

11.8 The Maxwell-Boltzmann Distribution

In this section, you'll reproduce the instructor plots of Fig. 11.6 using your own numerical simulation.
△ **Jupyter Notebook Exercise 11.6:** 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 11.2. The results should resemble the right side of Fig. 11.6, which were produced with `NGAS=10000`.
△ **Jupyter Notebook Exercise 11.7:** 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 11.3. The results should resemble the left side of Fig. 11.6, which were produced with `NGAS=10000`.

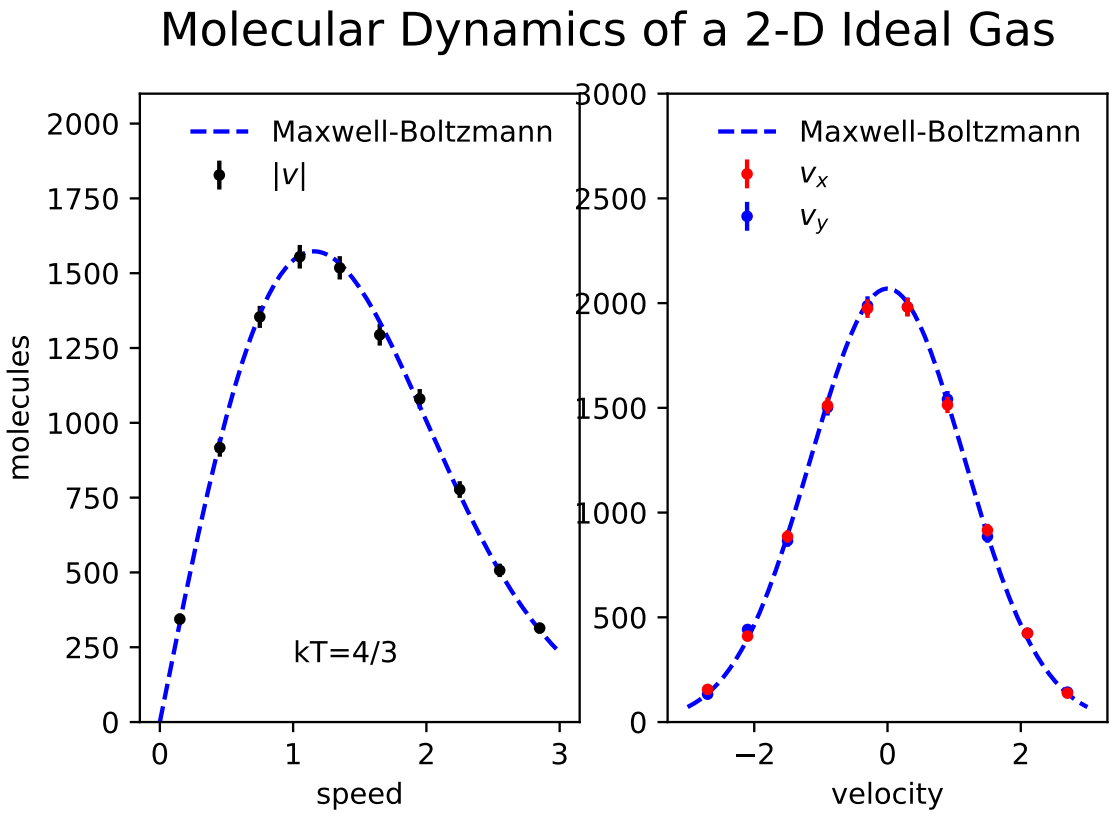


Figure 11.6: Instructor plots.

Chapter 12

Vectors

12.1 Introduction

In this lab, we will use numpy arrays to investigate the properties of vectors in Euclidean space.

12.2 Vector Dot Product and Cross Product

In this lab, we will use numpy arrays of length 3 to represent vectors in a Euclidean vector space. So for example:

```
a = np.array([1.2, -3.2, 2.1])
```

is how we will represent the vector $\vec{a}$ with $a_x = 1.2$, $a_y = -3.2$, and $a_z = 2.1$. To obtain the components of the vector, we retrieve the appropriate element of the array, so a_x is `a[0]`, a_y is `a[1]`, and a_z is `a[2]`.

We construct the unit vectors $\hat{x}$, $\hat{y}$ and $\hat{z}$ as:

```
xhat = np.array([1, 0, 0])
yhat = np.array([0, 1, 0])
zhat = np.array([0, 0, 1])
```

which provides an alternative way to define vectors:

$$\vec{b} = 2\hat{x} - 4\hat{y} + 1\hat{z}$$

using the equivalent python code:

```
b = 2*xhat - 4*yhat + 1*zhat
```

The dot product of two vectors is defined as:

$$\vec{a} \cdot \vec{b} = a_x\, b_x + a_y\, b_y + a_z\, b_z.$$

We can determine the magnitude of vector using the dot product:

$$|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}}$$

The cross product of two vectors is defined as:

$$\vec{a} \times \vec{b} = (a_y\, b_z - a_z\, b_y)\, \hat{x} + (a_z\, b_x - a_x\, b_z)\, \hat{y} + (a_x\, b_y - a_y\, b_x)\, \hat{z}$$

△ **Jupyter Notebook Exercise 12.1:** Implement a function `dot(a,b)` which returns the dot product of vectors a and b. You do not need to use a for loop in your implementation if you do not want to: explicitly calling each of three components is simple and clear. Confirm that your code is working by calculating $\vec{b} \cdot \hat{x}$, $\vec{b} \cdot \hat{y}$, and $\vec{b} \cdot \hat{z}$.

△ **Jupyter Notebook Exercise 12.2:** Implement a function `mag(a)` which returns the magnitude of the vector a by calling your `dot(a,b)` function. Show that your function works by comparing the magnitude of $\vec{b}$ return by your function with what you expect from paper and pencil.

△ **Jupyter Notebook Exercise 12.3:** Implement a function `cross(a,b)` which returns the cross products of the vectors a and b. Show that your function works for:

$$\begin{aligned} \hat{x} \times \hat{x} &= 0 \\ \hat{x} \times \hat{y} &= \hat{z} \\ \hat{x} \times \hat{z} &= -\hat{y} \\ \hat{y} \times \hat{x} &= -\hat{z} \\ \hat{y} \times \hat{z} &= \hat{x} \end{aligned}$$

The follow example demonstrates that:

$$\vec{a} \cdot \vec{b} \leq |\vec{a}|\,|\vec{b}|$$

by generating many random vectors and testing:

```
TOL = 10*np.finfo(float).eps
fail = 0
for i in range(10000):
    ra = np.random.rand(3)
    rb = np.random.rand(3)
    x = dot(ra,rb)-mag(ra)*mag(rb)
    if (x>TOL):
        fail = fail+1
print("failures:  ", fail)
```

Notice the use of a tolerance instead of strict comparison with zero to account for the floating point precision.

△ **Jupyter Notebook Exercise 12.4:** Demonstrate

$$|\vec{a} + \vec{b}| < |\vec{a}| + |\vec{b}|$$

△ **Jupyter Notebook Exercise 12.5:** Demonstrate the Jacobi identity:

$$\vec{a} \times (\vec{b} \times \vec{c}) + \vec{b} \times (\vec{c} \times \vec{a}) + \vec{c} \times (\vec{a} \times \vec{b}) = 0$$

12.3 Numpy Tools

The numpy tools for dot and cross products are `np.dot` and `np.cross`

```
print(np.dot([1,2,3],[0,0,1]))
print(np.cross([1,0,0],[0,1,0]))
```

12.4 Motion of a Charged Particle in a Magnetic Field

Chapter 13

Matrices, Rotations and Parity

13.1 Introduction

13.2 Preparation

13.3 Rotations and Parity

```
def rotz(theta):
    cs = np.cos(theta)
    sn = np.sin(theta)
    R  = np.array([[cs, sn, 0],[-sn, cs, 0], [0, 0, 1]])
    R=np.around(R,decimals=15)+0
    return R
```

△ **Jupyter Notebook Exercise 13.1:** With $\tau \equiv 2\pi$ print the rotation matrix for $\theta = 0, \tau/4, \tau/2, 3\tau/4, \tau$ corresponding to zero, quarter turn, half turn, 3/4 turn, and a full turn.

△ **Jupyter Notebook Exercise 13.2:** Show that $R(\vec{a} \times \vec{b}) = (R\vec{a}) \times (R\vec{a})$.

△ **Jupyter Notebook Exercise 13.3:** Show that $P(\vec{a} \times \vec{b}) \neq (P\vec{a}) \times (P\vec{a})$.

Chapter 14

Potential Energy and Symmetry

14.1 Introduction

14.2 Integration

Chapter 15

Scattering

15.1 Introduction

15.2 Scattering

Chapter 16

Roots

Chapter 17

Linear Algebra

17.1 Introduction

Chapter 18

Debugging

In our context, debugging is the process of finding and removing mistakes, called bugs, from your software. Singling this process out is a bit deceptive, it makes it seems distinct from software development, as if you should write your software, and then debug it. Indeed many students start this way, but it is a painful and ineffective approach. Experienced programs debug *while* developing their code.

The fundamental approach to debugging (which works equally well outside of programming) is to break every problem down into simple, well defined parts, and then thoroughly test each part. When one part does not work, you break it down into smaller parts. This process can be quite simple, such as adding print statements to each step of a complicated calculation. It can also be quite advanced, such as when teams of experienced software developers use automated builds and a suite of integration tests that validate every proposed change to code before it is accepted. Experienced programs still produce bugs, they just get better at squashing them.

There are a number of well-loved techniques to debugging:

- Print statements.
- Start with a simple problem.
- Test on special cases.
- Use paper and pencil.
- Decrease the size.
- Establish feedback.
- Write modular code.
- Maintain unit tests.