Physics 40 Lab Manual

Michael Mulhearn (original), Eric Prebys

December 1, 2025

Contents

Chapter 1

Lab1: 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 Miniconda

This course is based on Python and Jupyter Lab[1]. It's possible that you already have one or both of these things installed on your computer, but we will be using "Miniconda"[2] to make an environment with specific versions of both, as well as all supporting libraries, so that everyone is on exactly the same page regardless of what type of computer they have.

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 64-bit versions of Windows, Linux, and macOS. It should also work on all Chromebooks released since 2019, which capable of running

[1] Jupyter Lab a newer interface that adds a bit more functionality to Jupyter Notebooks. We'll use the terms "Jupyter Lab" and "Jupyter Notebook" interchangeably

[2] Miniconda is a lightweight version of Conda. If you already have Conda, that's fine, too.

a Linux environment. If you have an older Chromebook or a 32-bit OS, please see the instructors to determine an alternate solution.

Once you have determined your OS type, go to

```
https://www.anaconda.com/download/success
```

and follow the appropriate instructions for your OS below.

1.2.1 Installing Miniconda under Windows

Download and run the "Windows→64 Bit Graphical Installer" file. Accept all defaults, then proceed to Section 1.3.

1.2.2 Installing Miniconda under MacOS

Go do the download page and download the graphical installer corresponding to your processor. If you're not sure what type of processor you have, click "Apple→About this Mac". If the chip says "Apple Silicon M1/2/3", then select "Apple silicon". Otherwise, select "Intel".

Click on the downloaded package installer and select all defaults.

Proceed to Section 1.3.

1.2.3 Installing Miniconda under Linux

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 download the appropriate command line installer. If you're not sure what type of processor you have, open a terminal window and type:

```
lscpu
```

The first line will tell you what the architecture is. If it's "x86_64", then download the "64-bit (x86) Installer". Otherwise, if it's "aarch64", then download the file "64-bit (AWS Graviton2/ARM64) Installer".

Open a terminal window and navigate to where you downloaded the file (usually "Downloads"). You'll need to make it executable and execute it by typing

```
chmod +x Miniconda3-latest-Linux-(architecture).sh
./Miniconda3-latest-Linux-(architecture).sh
```

Where "(architecture)" is the version you downloaded. This will start the installation process. Accept all defaults.

Proceed to Section 1.3.

1.2.4 Installing Miniconda 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 instructions for installing under Linux in Section 1.2.3.

1.3 Installing the Physics 40 Conda Environment

We'll be setting up our conda environment and starting our Jupyter notebooks from a terminal window.

- On Windows, use the special anaconda window by going to "Start→All apps→Anaconda (miniconda3)→Anaconda prompt".
- On MacOS or Linux, just enter "terminal" in the search bar. Note that if you had a terminal open when you installed miniconda, you'll need to close it and restart it. Make sure to add it to the dock (MacOS) or favorites (Linux).

All subsequent commands will be typed in the terminal window.

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, type:

```
conda create -n phy40 python=3.13 numpy scipy matplotlib ipython jupyter
```

Answer "y" when prompted. This command creates an environment called "phy40" and installs the libraries "numpy", "scipy", "matplotlib", "ipython", and "jupyter". If needed, additional libaries can be installed from within the environment.

It's a good idea to create a directory to save all your lab work. You can put this directory anywhere you want, but for the moment, we'll assume it's right under your home directory, so in your terminal window, type

```
mkdir phy40
```

1.4 Starting Jupyter Lab

You'll notice that after you install miniconda on MacOS or Linux systems, "(base)" will start appearing before each command prompt, which is somewhat annoying. To get rid of this behavior, type

```
conda config --set auto_activate false
```

On Windows, the "(base)" prefix will only appear on an anaconda command window, and there is no way to remove it.

This course will make extensive use of the Jupyter Lab 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 piece of software, much of the initial code development can be done in the interactive session that Jupyter Lab provides.

Open you command window and type

```
cd phy40
conda activate phy40
```

This will put you in your working directory and activate the "phy40" environment you just created. You should see "(phy40)" tacked onto the beginning of each command line after that.

Note that any time you open a new terminal for a lab, you will need to go to the correct directory and activate the phy40 environment!

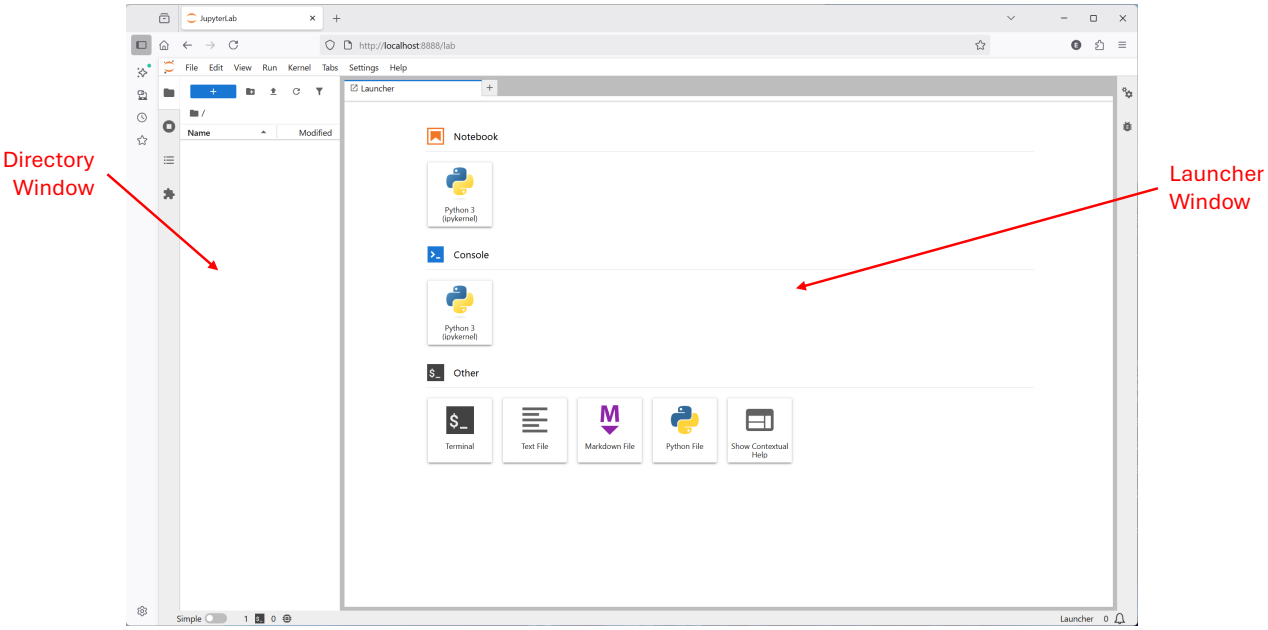


Figure 1.1: Initial appearance of the Jupyter Lab interface. The major difference with the older Jupyter Notebook interface is the navigation window at left.

Launch Jupyter Lab with:

```
jupyter lab
```

This should start the Jupyter Lab server and open a client in your default web browser which looks like Figure 1.1. The navigation pane at the left is the major difference between Jupyter Lab and the older Jupyter Notebook client. It shows the directory where you launched the client. It's empty at the moment because it's a new directory. You can create subdirectories, but Jupyter security features prevent you from navigating out of of the directory tree from which you launched the client.

1.5 Your First Notebook

You should create one Jupyter Notebook per lab assignment, by choosing the New (Python 3) option in your client. You will now see a file appear at the left called "Untitled.ipynb". You can

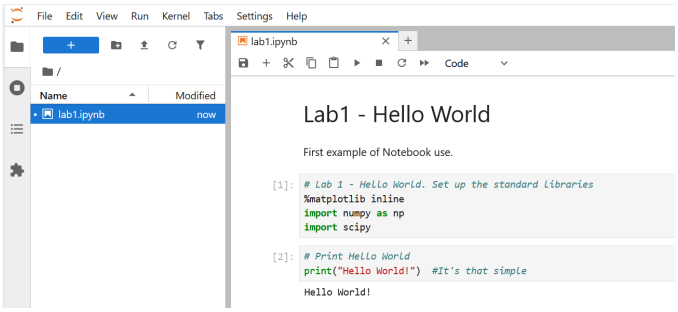


Figure 1.2: The Hello World example Jupyter Notebook.

change the name by right-clicking either on the file name or the name at the top of the tab. Change it to "lab1.ipynb".

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.

Jupyter Notebooks are divided into "cells", which can be executed sequentially or individually in any order. The three types of cells are:

- **Code (default):** Executable Python code
- **Markdown:** A powerful markup language, used for commentary, documentation, etc.
- **Raw:** Raw text.

We're going to put some comments in the first cell. Type

```
#Lab1 - Hello World
First example of notebook use
```

Select this cell, use the pull down menu above to change the type from "Code" to "Markdown", and then click the little "play" (triangle) to execute the cell. You should see that the using the "#" symbol in your markup caused the first line to be rendered in a large font, as shown in Figure 1.2.

Now use the "+" icon to create two more cells and populate them as shown in the figure. The first cell sets up the "matplotlib" plotting library two work inline in the notebook, while the next two lines import the "numpy" and "scipy" libraries. We don't actually need these libraries for this lab, but it's good boiler plate for the future.

The second cell prints "Hello World!". Note that it's good practice to comment your code.

Execute the cells by selecting them and click the "play" icon.

Jupyter Notebook checkpoints your work automatically, but you can explicitly save it by clicking the "save" icon. Do this, and then right click on the file name at left and select "Open with...→ Editor". This shows you what the actual source of the Python notebook looks like, as shown in Figure 1.3 The Jupyter client translates this into the readable format you see.

1.6 Saving your Work

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

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "id": "9608ece4-f404-4aa0-8794-b90a8e7ad814",
   "metadata": {},
   "source": [
    "# Lab1 - Hello World\n",
    "\n",
    "First example of Notebook use."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 1,
   "id": "75256df1-9e70-4ddb-ae11-968b7b7e10d2",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Lab 1 - Hello World. Set up the standard libraries\n",
    "%matplotlib inline\n",
    "import numpy as np\n",
    "import scipy"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 2,
   "id": "08f01a01-9a08-44a6-b8d9-dbefe6eeb6d7",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "Hello World!\n"
     ]
    }
   ],
   "source": [
    "# Print Hello World\n",
    "print(\"Hello World!\")  #It's that simple"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "f3ca9199-0a20-4446-a6a6-0938c8047cee",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
"metadata": {
 "kernelspec": {
  "display_name": "Python 3 (ipykernel)",
  "language": "python",
  "name": "python3"
 },
 "language_info": {
  "codemirror_mode": {
   "name": "ipython",
   "version": 3
  },
```


Figure 1.3: Source code of Jupyter notebook.

your notebook, but the most reliable is to do "File→Print.." and then select "Save to PDF" as the destination. 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.

1.7 Closing your Session

It's best practice to end your session by doing "File→Shut Down" to stop the kernel, and then closing the browser window. This will free up the terminal.

If you simply close the browser window, it will leave the client running in the terminal. You can stop this by closing the terminal window on all systems or by hitting <cntl>-C on MacOS or Linux.

If the terminal remains open, you can exit the phy40 environment if you wish by typing

```
conda deactivate
```

1.8 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 **both** the PDF file and notebook file to the course website.

Chapter 2

Lab 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".

Python will try to guess what type of variable you want based on how you initialize it. In this case, it interpreted "121" as an integer. On the other hand, add a couple of lines of code where you add a "." to the end of 121 and see how this changes the type.

There will be times when you want to have control over exactly what type of variable you have. One such case is if you plan to pass variables to a routine written in another language.

You can explicitly type the variable a by setting it to "float(121)". Try this and and then print the type and value again.

△ **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.

In this lab, it will be convenient to know how to calculate the absolute value of a number, which you can do with the python `abs` function:

```
a = -1.5
b = abs(a)
print(b)
```

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

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

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)
```

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 65322 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 satellite 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:** This is a paper and pencil exercise. You can do it on paper and pencil and submit a scanned PDF, or you can just record you steps as comments in a jupyter notebook cell. 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 significant 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
print(eps)
```

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 significant 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)
```

The strict equality condition $x == 0$ is not met because of limited floating point precision. Instead of strict equality, check that x is near zero with a condition like:

$$|x| < 10\epsilon$$

where ϵ is the machine precision and the factor of 10 is a conservative factor to allow for round-off errors that might be a bit larger than the best possible precision ϵ. Add such a condition to the code above and show that this looser definition of equality now holds. In general, when using approximations (like floating point numbers) you can only check things to within the accuracy of the approximation!

△ **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 = 3;
b = 1.2343E-17;
sum = 0
sum += a;
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 with value of $1+i$ and multiply it by it's complex conjugate. 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] = 3
print(a[0])
print(a is b)
```

What happened here? This illustrates a very important concept in Python. When you equate something to an *immutable* object, like an integer or string, it creates a *new and separate* variable. This is known as "pass by value". In fact, if you try to change the value of an immutable object, it will actually delete the old object and create a new one with the same name. The following is a list of the most common immutable objects in Python

- int
- float
- complex
- book
- str
- tuple
- bytes
- frozenset

On the other hand, almost all other objects are *mutable*. In the example above, the first line created an array. When you equated "b" to "a" on the second line, it didn't copy the array, but rather passed a pointer to the original array. This is knows as "pass by object reference" or "pass by assignment.

△ **Jupyter Notebook Exercise 2.17:** Copying a mutable object.

There's a good chance that what you did in the previous example was not what you wanted to do.

Redo the previous example, but change the second line to

```
    b = a.copy()
```

and see how it changes the behavior.

Note, both Java and Python use the default pass by reference behavior for complex objects and require you to explicitly make a copy if that's what you want. This is generally considered a big improvement over C++, which defaults to passing by value, which can result in a minor typo causing you to accidentally copy *enormous* amounts of data.

△ **Jupyter Notebook Exercise 2.18:** 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). We are able to change just a part of a list (like `a[0] = 3`). But can we change part of a complex number (like `z.real`). Try it and find out!

(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

Lab 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 a challenge problem in lieu of some of the other problems: see the final problem for the details.

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 the function, usually in a different cell:

```
loop(5)
```

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

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

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

```
3
6
9
12
```

In future problems, we'll describe this output simply as 3, 6, 9, 12. We won't be picky about whitespace unless we discuss it explicitly. One way to complete this is to use the three arguments 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 k in range(n):
          sum = sum + 1 + k
          #print("k: ", k, "\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:** 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:** Extend the Arithmetic series function to include parameters a and d. Show that it works.

3.7 Maclaurin series

A Taylor series is an expansion of of a function about a point based on the derivatives at that point. A Maclaurin series is a special case of a Taylor series, which is expanded about 0. It is defined by

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 ...$$

This is equivalent of approximate a function with a polynomial. Larger is the polynomial, better is the approximation even at points of the function far away from the referent point.

Let's star defining a function `f(x) = cos(x)`. For this exercise, we will use the numpy package that we will use extensively later in the course.

```
import numpy as np
def myfunc(x):
    return np.cos(x)
```

Let's now define a second order polynomial that approximate this function (around 0) and let's write a python function with this polynomial. To find this function you compute the derivatives of f(x) and can use the Maclaurin series:

$$P(x) = 1 - \frac{1}{2!}x^2$$

```
def polynomialorder2(x):
    return  1 - 1/2. * x**2
```

△ **Jupyter Notebook Exercise 3.9:** Compute the derivative of the function f(x) around the point 0 and define the polynomial that approximate the function f(x) with order 4, 6 and 8. Define a new function for each of these polynomials.

△ **Jupyter Notebook Exercise 3.10:** Investigate how good is your polynomial or second order to approximate f(x) = cos(x) when you start to go far away from x=0. In particular, fix a value for x (x = 1) and compare in absolute value the difference between `f(x) - polynomialorder2(x)`

```
x = 1
print(np.abs(myfun(x) - polynomialorder2(x)))
```

△ **Jupyter Notebook Exercise 3.11:** Investigate how good are the polynomial of orders 2 (P2), 4 (P4), 6 (P6) and 8 (P8) to approximate the function f(x). Find how far can you go with x and still have difference between f(x) and your polynomials (P2,P4,P6,P8) < than .1. You can do this by trial and error, but try to find the x value corresponding to a .1 error within .1.

3.8 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 3.12:** Explicitly verify these properties for the first four Fibonacci integer right triangles.

Chapter 4

Lab 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. For now, let's assume that a, b, and c are all integers.

In this case, the solution to the quadratic equation can be calculated as follows:

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

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 one more case with integer solutions and use it to test your function more thoroughly.

△ **Jupyter Notebook Exercise 4.4:** As we discussed in a previous lab, Python can work with imaginary numbers, but you will need to import `cmath` and use `cmath.sqrt()` rather than `math.sqrt()`. Make this change and verify that it works with all the examples, including the one with no real solutions.

△ **Jupyter Notebook Exercise 4.5:** (Optional) The condition that the discriminant is exactly zero (`D==0`) is problematic when applied to floating point numbers. (Why?) In example is for a=.1 b=.3 c =0.225 which should have only one real solution. Test you function with this test case. Modify the conditionals in your function to account for floating point precision and test it.

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 4.6:** 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 4.7:** 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. With that in mind...

△ **Jupyter Notebook Exercise 4.8:** Write a routine to simplify the fraction

$$\frac{a}{b}$$

as much as possible. You should return two integers as a Tuple, like this:

```
def simplify(a,b):
    # Your code
    return a_simp,b_simp
```

It should use the following algorithm

```
1   a_simp := a
2   b_simp := b
3   i := 1
4   while i less than or equal to the minimum of a_simp and b_simp:
5       if i is a factor of a_simp and b_simp:
6           a_simp :=a_simp / i
7           b_simp :=b_simp / i
```

Test this algorithm with (3,12), (51,9), and (3,11), with code like:

```
a,b = simplify(3,12)
print("{}/{}".format(a,b))
```

△ **Jupyter Notebook Exercise 4.9:** Write 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.

```
# 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))
```

Note that you don't have to worry about simplifying the fraction until the final step, for which you can use the `simplify()` routine you wrote in the previous exercise.

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 4.10:** (Optional) Find the the values of a and b which produce the largest number of prime numbers. Restrict yourself to $|a| \leq 1000$ and $|b| \leq 1000$.

If you do complete this optional problem, then nice work, hot shot, but remember that Euler found his without using a computer!

Chapter 5

Lab 5: Arrays, Plotting and Chaos

5.1 Introduction

This lab will introduce a fundamental element of scientific python, the numpy array, and use them to produce plots. We will also consider the non-linear logistics map and examine bifurcation in the approach to chaos.

If you can complete all of the plots in Section 5.6 including the optional plot and *without any instructor help* then you may omit the plots from the preceding sections.

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.
For this lab you will need the "pandas" library, which we did not install when we created the environment. If you have not already done so, after activating the environment, type

```
pip install pandas
```

on the command line.

This is the first lab that relies on inline plotting, so make sure you are starting your notebook with the "line magic" and install the required libraries:

```
%matplotlib inline
import matplotlib.pyplot as plt.
import numpy as np
import pandas as pd
```

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 computationally efficient. Choices for the data type include the built-in python integer, float, and bool types. The numpy library provides a wide range of analysis tools that are mostly centered on the numpy array type.

Numpy arrays can be constructed easily 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))
```

This is conveninent when you have specific values you want to define by hand. Another possibility is to construct the numpy array by calling a function designed specifically for the purpose:

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

Both `linspace` and `arange` allow you to specify the range of values you want, but with `linspace` you specify the number of points you want whereas with `arange` you specify the step size.

One of the great joys of using numpy arrays comes from the fact that most operators are applied elementwise automatically, without the need to explictly write a for loop:

```
a = np.arange(0,5,1)
b = 10
print(a)
print(b)
print(a+b)
```

Notice how the value of b (10) is added to *every element* of a, without the need to explicitly loop over every element. Try modify the example code to multiply every element of a by b. Then try raising each element of a to the power of 2.

△ **Jupyter Notebook Exercise 5.1:** Use numpy arange and elementwise operations to implement a function `def powers(a, n)` which returns a numpy array containing the first powers of a from 1 to a^n. So for example `print(powers(2,4)` should output `[ 1 2 4 8 16]`
vskip 1cm

Numpy arrays can also be built up element by element using the append function:

```
a = np.array([])
print(a)
a = np.append(a, 1)
print(a)
a = np.append(a, 3)
print(a)
a = np.append(a, 4)
print(a)
```

This example creates an empty numpy array and then adds one element at a time.

△ **Jupyter Notebook Exercise 5.2:** Run the snippet above and observe the output.

△ **Jupyter Notebook Exercise 5.3:** Implement a function `def recur(n,x)` which returns an array containining the first n values of the recurrance relationship $x_{i+1} = 2\,x_i + 1$ starting from $x_0 = x$. Use the following algorithm:

```
1  Parameter x # starting value
2  Parameter n # number of iterations
3  Create empty array a
4  Repeat n times:
5     append x to array a
6     x := 2x+1
7  print a
```

Test your code for $x = 1$ and $n = 5$ and ensure that it produces the correct output: `[ 1. 3. 7. 15. 31.]`.

5.3 Plotting with Scientific 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 5.4:** Plot the $sinc(x)$ function as a smooth line in the x range from -5 to 5. Add appropriate labels to each axes. Include a legend identifying the sinc function. For the line color, use any color other than red or blue.

5.4 Plotting with Pandas

Pandas is a powerful tool for manipulating data. It's essentially an internal spreadsheet. Download the file "lab5.csv" from "Files/labs/lab5" at the Canvas site. This is a comma separated file with 3 columns: "x", "f1", and "f2".

△ **Jupyter Notebook Exercise 5.5:** Read this file in with

```
    df = pd.read_csv('lab5.csv')
```

Create a new column, which is the product of the first two, by typing

```
    df['f3'] = df['f1'] * df['f2']
```

Now plot all three functions on the same plot by typing

```
    plt.plot(df['x'],df['f1'],label='f1')
```

etc. Include a legend and a label on the x axis.

5.5 Plot and Distributions

Use `numpy.random.normal` to create an array with 10000 elements. This array of number will follow a Gaussian distribution (use $\mu = 10$ and $\sigma = 3$).

△ **Jupyter Notebook Exercise 5.6:** Histogram this distribution using the plt.hist() function. Remember to add labels on the axes and a title for the plot.

△ **Jupyter Notebook Exercise 5.7:** Implement a function `def Gaussian(x, sigma, mu)` which, given sigma and mu returns a Gaussian. Use the following Gaussian formula to create this function. The try to overlap this function to your plot to confirm that the random.normal distribution is a Gaussian distribution. Remember, you will have to scale this to match your histogram.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

5.6 The Logistics Map

The logistics map is the recurrence relation

$$p_{i+1} = r\ p_i\ (1 - p_i)$$

which calculates the value of p for step $i + 1$ from step i. The variable p can represent the ratio of a population to its maximum possible value, and each iteration (n) is a step in time (such as one year). Each year, the population increases due to birth and decreases due to starvation as the population approaches it's maximum value (p near 1). The growth (or decline) of the population is controlled by the growth rate parameter r. We will only consider r in the range from $[0, 4]$ which keeps p in the range $[0, 1]$. This is a simple non-linear model which illustrates chaotic behavior.

△ **Jupyter Notebook Exercise 5.8:** Implement a function `def logmap(p, r)` which returns the next iteration (p_{i+1}) of the logistic map for parameter r and $p_i = p$. Test your code by showing that for $r = 3.0$ and $p_0 = 0.1$ the next five iterations are: 0.27, 0.5913, 0.725, 0.5981 and 0.7211.

△ **Jupyter Notebook Exercise 5.9:** Use your `logmap` function to build an array containing the first n entries in the logistics map starting from $p_0 = 0.1$. Check that the output is correct by comparing with the results from the previous exercise for $n = 6$.

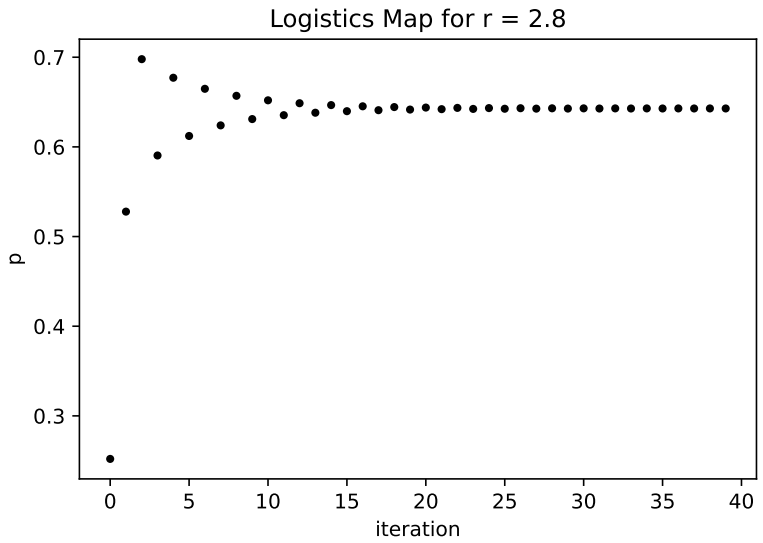


Figure 5.1: Convergence of the logistic map for $r = 2.8$

△ **Jupyter Notebook Exercise 5.10:** Plot the time evolution of the logistics map for $r = 2.8$ for 40 iterations, starting from $p_0 = 0.1$ as in Fig. 5.1. To create a plot, you'll need an array containing the p values (these correspond to the y axis in the plot) which you can construct as in the previous

exercise. But what about the x axis values? Since your array of p values contains $[p_0, p_1, p_2 \ldots p_n]$ the corresponding array of indices is simply $[0, 1, 2 \ldots 3]$ which you can construct using `np.arrange`.

Your results from the previous exercise should reproduce Fig. 5.1. This shows that for $r = 2.8$ the logistics map converges to a value of about 0.64. But this non-linear recursion relationship does not always converge to a single value.

△ **Jupyter Notebook Exercise 5.11:** Plot the time evolution of the logistics map for $r = 3.2$ for 40 iterations, starting from $p_0 = 0.1$.

Notice that now the system oscillates between two values (near 0.5 and 0.8).

△ **Jupyter Notebook Exercise 5.12:** Plot the time evolution of the logistics map for $r = 3.5$ for 40 iterations, starting from $p_0 = 0.1$.

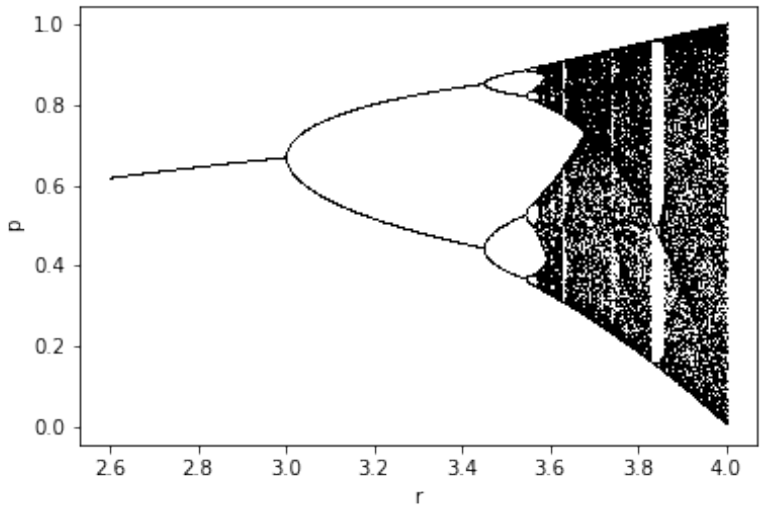


Figure 5.2: Long-term behavior of the logistics map as a function of parameter r.

Notice that now the system oscillates between four values. This is an example of bifurcation in the approach to chaos. To show this more clearly, we'd like to produce a plot as in Fig. 5.2. For each r value, this shows the p values from 100 iterations *after* the first 1000 iterations. This shows the *long term behavior* of the logistics map. We can clearly see that for $r = 2.8$ the p values converge to one single value and for $r = 3.2$ there are two values just as in the previous exercises. These bifurcations continue until the system becomes chaotic (oscillating between many different values) with occassional windows of stability.

To produce this plot for yourself, start by considering this snippet:

```
Nr = 5
r = np.linspace(2.6,4,Nr)
p = logmap(0.1,r)
print(p)
p = logmap(p,r)
```

```
print(p)
```

Here we create a numpy array of r values, and pass that to our `logmap` function instead of a single value. The operations within the function are applied elementwise to the array, and the result is that instead of a single p value, the call to to `logmap(0.1,r)` returns an array of p values, one for each r value. This is just what we need to make the plot in Fig. 5.2.

△ **Jupyter Notebook Exercise 5.13:** Run (and understand!) the snippet and make a plot of p versus r. Understand that you are plotting p_2 as a function r! Use the `\"k,\"` format option (black pixels) when plotting. Comment out the print statements and increase Nr to 100.

△ **Jupyter Notebook Exercise 5.14:** Make a plot that is *almost* like that of Fig. 5.2 by plotting p_{1001} versus r. Hint: in the code above, instead of one call to `p = logmap(p, r)` use a for loop which calls this 1000 times.

You should start to see features of the Fig. 5.2 but you are only plotting one p value for each r. To see the bifurcations and chaos, you'll need to plot about 100 p values at each r value.

△ **Jupyter Notebook Exercise 5.15:** Reproduce the plot in Fig. 5.2. Hint: instead of plotting just p_{1001} as in the previous exercise, plot the next 100 values as well. Increase the number of r values plotted to 1000. Add appropriate labels to each axis.

△ **Jupyter Notebook Exercise 5.16:** (Optional) The bifurcation diagram which you have constructed exhibits self-similarity. If you zoom into an appropriate region of the diagram, you will find a diagram which looks quite similar to the original diagram. Produce a plot that demonstrates this self-similiarity by zooming into a particular region.

Chapter 6

Lab 6: Differentiation and Projectile Motion

6.1 Introduction

In this lab we will apply numerical differentiation to elementary functions. We'll apply the Euler method to compute the trajectory of a projectile, and compare our results with the analytic solution. We'll also show that our numerical technique easily accommodates air resistance, a problem that has no analytic solution using elementary functions.

6.2 Preparation

Our code is going to get 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.

In this lab, we will also 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 the 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, we will encounter problems with floating point precision.

△ **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 of 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 $f(x) = \sin(x)$ using the `np.sin` function. 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

In lecture, we derived the Euler Method for iteratively determining the trajectory of a particle:

$$\begin{aligned} \vec{v}_{n+1} &= \vec{v}_n + \tau\vec{a}_n \\ \vec{r}_{n+1} &= \vec{r}_n + \tau\vec{v}_n \end{aligned}$$

△ **Jupyter Notebook Exercise 6.6:** Implement a function

```
def euler(dt, x, y, vx, vy, ax, ay):
    # your code here
    return x, y, vx, vy
```

which calculates the next iteration of x,y,v_x, and v_y from the current values of x,y,v_x,v_y,a_x and a_y. Notice that this function returns several different variables at once using a comma separated list

referred to as tuple in python. To retrieve the individual variables from the tuple, simply call the function like this:

```
x,y,vx,vy = euler(x,y,vx,vy,ax,ay)
```

One downside of this convenient approach is that you must get the order of the variables correct! 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 that you get the correct output:

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

We will use the Euler Method to simulate projectile motion. We'll take the initial velocity to be 20 m/s and take $g = 9.8$ m/s^2. Here's a snippet of code that sets these constants and determines the x and y coordinates of the initial velocity from an angle θ, which is set to 45°:

```
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)
```

The trajectory of the particle can be determined using the following algorithm:

```
Create an empty array tjx # will contain x positions of the trajectory
Create an empty array tjy # will contain y positions of the trajectory
while y ≥ 0:
    Append the x position to tjx
    Append the y position to tjy
    Compute the next values of x,y,v_x and v_y using the Euler Method
Plot tjy versus tjx
```

Notice that the algorithm stops just before the projectile reaches $y \leq 0$.

△ **Jupyter Notebook Exercise 6.7:** Use the Euler Method to plot the trajectory of a projectile with the initial conditions described above.

△ **Jupyter Notebook Exercise 6.8:** Derive an expression (paper and pencil) for the maximum range of the trajectory and evaluate the range for these initial conditions. Are the results consistent?

△ **Jupyter Notebook Exercise 6.9:** Extend your simulation to record v_x and v_y at each step along with the x and y positions. Take the mass of the projectile to be $m = 0.215$ 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 versus 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

In this section we will consider the effect of air restistance on a baseball thrown a 22 m/s. 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 6.10:** 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 6.11:** Including the effect of air restistance, plot the kinetic energy, potential energy, and total energy (kinetic plus potential) of the projectile as a function of time. Is the total energy of the projectile conserved?

Chapter 7

Lab 7: Pendulum Motion

7.1 Introduction

In this lab we will apply the Euler and Verlet methods to the pendulum problem. We will compare the results of the Verlet problem to the small angle approximation.

If you prefer, you can complete the shorter, but more challenging sequence of problems: 7.8, 7.10, and at least two of the optional challenge problems. You will only receive minimal instructor help while you are taking this approach.

7.2 Preparation

Suppose you want to produce a plot of $f(t) = A\exp(-t/\lambda)$ versus t from $t = 0$ to 10 and $A = \lambda = 5$. For a visually smooth plot, you want about $N = 100$ points, but we'll start with a smaller number $N = 11$ for easy debugging. You create an array containing the 11 time values you want to plot:

```
MAX = 10
N = 11
t = np.linspace(0,MAX,N)
print(t)
```

You calculate the $y = f(t)$ values like this:

```
A    = 5
LAMB = 5
y = A*np.exp(-t/LAMB)   # !!!
print(y)
```

Make sure that you thoroughly understand the line marked "!!!". That is calculating one y value for every t value, and so y is an array with the same shape and size as t:

```
print("t shape: ", np.shape(t), "y shape: ", np.shape(y))
print("t size:  ", np.size(t),  "y size: ", np.size(y))
```

With two arrays of the same shape, plotting them is a simple matter. Here we use the red line format, and add some axes labels:

```
plt.plot(t,y,"r-")
plt.xlabel("t (s)")
plt.ylabel("y")
```

If you look closely, you'll see kinks in the plot for $N = 11$. Increase to $N = 100$ for a visually smooth plot.

You are expected to do all of this on your own, from a prompt like this:

△ **Jupyter Notebook Exercise 7.1:** Plot the function $f(t) = A\exp(-t/\lambda)$ from $t = 0$ to 10, $A = 5$ and $\lambda = 5$ as a red line.

Now suppose instead that you already have a set of y-values in a `np.array` `yarr`, and you would like to plot y vs t, knowing that these y values were sampled starting at $t = 0$ with a uniform step size of $\tau = 0.2$ between each sample, i.e. at $t = 0, 0.2, 0.4, \ldots$. In this case, you would create a `np.array` containing your time values as:

```
tau = 0.2
t = np.arange(yarr.size)*tau
```

△ **Jupyter Notebook Exercise 7.2:** Starting from the y values contained in:

```
yarr = np.array([1,1,1,2,4,7,3,2,1,1,0,0])
```

which you know correspond to time values starting at $t = 0$ with constant step size $\tau = 0.5$. Plot y vs t as blue circles and add axes labels.

7.3 Pendulum Motion

In lecture we showed a pendulum of length L can be described by the angle θ with respect to vertical (rest position), the angular velocity:

$$\omega = \frac{d\theta}{dt}$$

and the angular acceleration:

$$\alpha = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}$$

In a constant gravitational field with acceleration g, the angular acceleration is:

$$\alpha = -\frac{g}{L}\sin\theta$$

Which we can write as:

$$\alpha = -\omega_L^2 \sin\theta$$

where

$$\omega_L = \sqrt{\frac{g}{L}}.$$

This problem, which is quite simple to pose, has no analytic solution in terms of elementary functions. Instead, we will rely on numerical techniques to solve it.

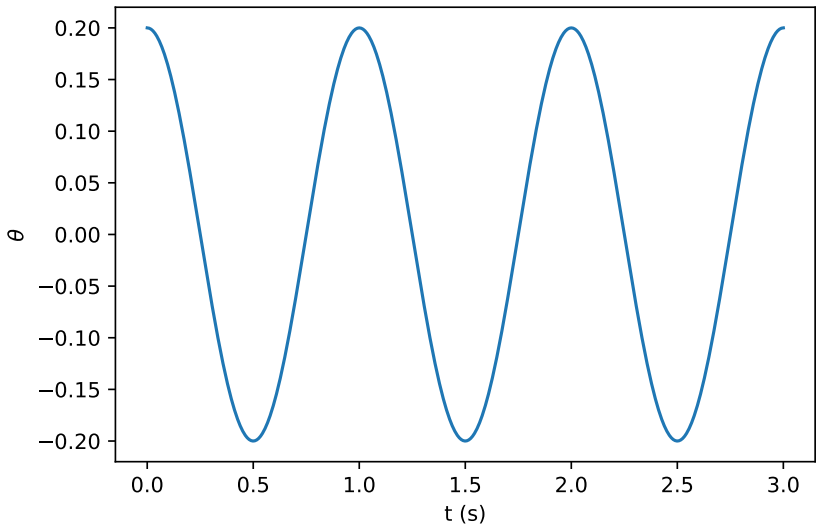


Figure 7.1: The small angle approximation for pendulum motion with period $T_L = 1$ s

.

7.4 Small Angle Approximation

Analytic solutions are the precious gems that we use to validate our numerical techniques. We'll start our analysis in a region where we can solve the problem analytically. For small displacements of the pendulum, θ is small, and so:

$$\sin\theta \approx \theta$$

and

$$\alpha = -\omega_L^2 \sin\theta \approx -\omega_L^2 \theta$$

The differential equation which describes the motion is:

$$\frac{d^2\theta}{dt^2} = -\omega_L^2 \theta$$

We showed in lecture that if the initial angular velocity is zero ($\omega_0 = 0$) and the initial theta position is θ_0, then the solution is:

$$\theta(t) = \theta_0 \cos(\omega_L t) \tag{7.1}$$

as plotted in Fig. 7.1. The pendulum rocks back and forth following a sine wave with period:

$$T_L = \frac{2\pi}{\omega_L} = 2\pi\sqrt{\frac{L}{g}}$$

△ **Jupyter Notebook Exercise 7.3:** Calculate ω_L for a pendulum with $L = 1$ m and $g = 9.8$ m/s^2. What are the units of ω_L?

△ **Jupyter Notebook Exercise 7.4:** Suppose you want to construct a pendulum such that for small oscillations the period $T_L = 1$ s. What should be the value of ω_L? What length L should you use, assuming that $g = 9.8$ m/s?

△ **Jupyter Notebook Exercise 7.5:** Reproduce the plot in Fig. 7.1. Set the constants T_L, ω_L, θ_0, and N as follows:

```
TL = 1           # period in seconds
wL = 2*np.pi/TL  # small-angle angular frequency of pendulum
A  = 0.2         # theta at t=0 (amplitude)
```

You must complete this problem **without using an explicit for loop**. To set the y axis label to the fancy θ do:

```
plt.ylabel("$\\theta$")
```

.

7.5 The Failure of the Euler Method

The Euler equations for angle θ and angular velocity ω are:

$$\theta_{n+1} = \theta_n + \tau\,\omega \tag{7.2}$$
$$\omega_{n+1} = \omega_n + \tau\,\alpha \tag{7.3}$$

where α is the angular acceleration and τ is the time step.

△ **Jupyter Notebook Exercise 7.6:** Implement a function

```
def euler(tau, theta, omega, alpha):
    # your code here
    return theta, omega
```

which implements one iteration of the Euler method. Test your `euler` function with these test values:

```
print(np.around(euler(-0.01, -0.28, -0.30, -0.06),3))
print(np.around(euler( 0.94,  0.32, -0.85, -0.86),3))
print(np.around(euler( 0.92, -0.16,  0.38, -0.32),3))
print(np.around(euler( 0.31,  0.12, -0.91, -0.76),3))
print(np.around(euler(-0.14,  0.96,  0.66, -0.73),3))
```

which should produce the following output:

```
[-0.277 -0.299]
[-0.479 -1.658]
[0.19  0.086]
[-0.162 -1.146]
[0.868 0.762]
```

You'll use this now thoroughly tested function more below. Don't change it!

△ **Jupyter Notebook Exercise 7.7:** Apply the Euler method to the problem of small oscillations of a pendulum with $T_L = 1$ s as in Fig. 7.1. First, set the parameters of your code just as before:

```
TL = 1           # period in seconds
wL = 2*np.pi/TL  # small-angle angular frequency of pendulum
A  = 0.2         # theta at t=0 (amplitude)
```

Then implement the Euler method as follows:

```
θ := A
ω := 0
τ := 0.0003
Create an empty array tjth which will contain θ positions of the trajectory
Append θ to the array tjth.
Repeat N times:
    Calculate α = −ω_L^2 θ
    Update θ and ω for acceleration α and time step τ by calling euler().
    Append θ to the array tjth.
Create an array t containting N+1 appropriately spaced time values.
Plot tjth versus t
```

As always, first debug your code using a small value for N. Then, you should reproduce something that closely resembles Fig. 7.1 with $N = 10000$.

△ **Jupyter Notebook Exercise 7.8:** Repeat the exercise above (you can use cut and paste) with $\tau = 0.01$ and $N = 1000$. Yikes! Is energy conserved?

7.6 The Verlet Method

The Verlet Equation for this problem is:

$$\theta_{n+1} \;=\; 2\theta_n - \theta_{n-1} + \tau^2\,\alpha \tag{7.4}$$

Notice that with the Verlet method, we will not need to calculate angular velocity ω in order to get the θ trajectory.

△ **Jupyter Notebook Exercise 7.9:** Implement a function

```
def verlet(tau, theta, oldth, alpha):
    # your code here
    return theta
```

which returns θ_{n+1} from θ_n =theta and θ_{n-1} =oldth using the verlet method. Test your `verlet` function with these test values:

```
print(np.around(verlet(-0.01, -0.28, -0.30, -0.06),3))
print(np.around(verlet( 0.94,  0.32, -0.85, -0.86),3))
print(np.around(verlet( 0.92, -0.16,  0.38, -0.32),3))
print(np.around(verlet( 0.31,  0.12, -0.91, -0.76),3))
print(np.around(verlet(-0.14,  0.96,  0.66, -0.73),3))
```

which should produce the following output:

```
-0.26
0.73
-0.971
1.077
1.246
```

You'll use this now thoroughly tested function more below. Don't change it!

△ **Jupyter Notebook Exercise 7.10:** In a previous exercise you showed that the Euler method, when applied to the problem of small oscillations of a pendulum with $T_L = 1$ s for $\tau = 0.01$ and $N = 1000$, is unstable. Instead, apply the Verlet method. You can reuse (by copying and pasting) much of your code from that previous exercise with a few changes:

- You will call your `verlet` function instead of `euler`.
- You no longer need to keep track of ω (`omega`)
- You will now have to keep track of two θ values at all times. At each updat:

$$(\theta_n, \theta_{n-1}) \to (\theta_{n+1}, \theta_n)$$

- You can start things off with `oldth = theta = A`

With this method, you should produce many oscillations with no sign of instability.

△ **Jupyter Notebook Exercise 7.11:** So far, we have been using the small angle approximation. Modify your code to use the exact formula for the angular acceleration $\alpha = \omega_L^2 \sin\theta$ and set the initial position to $A = 2$. This is a trivial change to your numerical simulation, but it makes an analytic solution impossible!

△ **Jupyter Notebook Exercise 7.12:** (Optional Challenge) Run you Verlet analysis for $N = 1000$ steps for $A = 3$ and set τ appropriately so that you see a bit more than one period of motion. Plot the trajectory as a black line and read off the period T. Superimpose a plot of a cosine with amplitude A and period T.

△ **Jupyter Notebook Exercise 7.13:** (Optional Challenge) We've been starting things off with approximately zero angular velocity by setting `oldth = theta = A`. You can add velocity to the initial state with:

```
V = 1
theta = A
oldth = A - tau*V
```

Give the pendulum enough of a whack that it reaches all the way to top and keeps going. When plotting this trajectory, you can use `np.mod` to keep the theta values in the range from -π to π if you want.

△ **Jupyter Notebook Exercise 7.14:** (Optional Challenge) Fix the Euler method for this pendulum problem by forcing energy to be conserved at each step. Compare your results to the Verlet method.

Chapter 8

Lab 8: Roots and Integrals

8.1 Introduction

In this lab, we will implement numerical alogirthms for root-finding and integration. We will apply the root finding algorithm to a classic problem from quantum mechanics: the bound-state energy levels of particle in finite potential well. We will apply numerical integration to the calculation of the periods of a simple pendulum.

8.2 Roots of a Linear Function

It is usually best to start simple when developing code, so we'll develop our algorithms using a simple linear function with a root at $x = 3$.

△ **Jupyter Notebook Exercise 8.1:** Implement $f(x) = 5\,(x - 3)$ as a python function

```
def f(x):
    # your code ...
```

Calculate the derivative of f and implement it as the function:

```
def fp(x):
    # your code ...
```

Check you code with:

```
for x in [-1,3,4]:
    print(f(x), fp(x))
```

which should return:

```
-20 5
0 5
5 5
```

.

8.3 Bisection Method

In the bisection method, we start from values a_1 and b_1 where $f(a_1) \cdot f(b_1) \leq 0$. This condition insures that the range $[a_1, b_1]$ contains at least one root. The bisection method halves the interval with each iteration, chosing the half that contains at least one root. Given a_n and b_n, we calculate:

$$c_n = (a_n + b_n)/2$$

we then determine:

$$a_{n+1} = \begin{cases} a_n & f(a_n) \cdot f(c_n) \leq 0 \\ c_n & \text{(otherwise)} \end{cases}$$

and

$$b_{n+1} = \begin{cases} c_n & f(a_n) \cdot f(c_n) \leq 0 \\ b_n & \text{(otherwise)} \end{cases}$$

△ **Jupyter Notebook Exercise 8.2:** Implement the python function

```
def bisection(f, a, b):
    # your code
    return a, b # updated values
```

which, given the function f=$f(x)$ and interval defined by a=a_n b=b_n, returns the smaller interval defined by a=a_{n+1} and b=b_{n+1} using the bisection algorithm. Test your code with:

```
print(bisection(f,0,4))
print(bisection(f,3,4))
```

which should have output:

```
(2.0, 4)
(3, 3.5)
```

8.4 Newton's Method

We can use Newton's method to find the roots of a function $f(x)$ if we know its derivative $f'(x)$. From our current best estimate for the root x_n we calculate a better estimate as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

△ **Jupyter Notebook Exercise 8.3:** Implement the python function:

```
def newton(f,fp,x):
    # your code
    return x # updated value
```

which, given the function f=$f(x)$, it's derivative fp=$f'(x)$, and the current best estimate for the root x=x_n, returns the improved estimate x_{n+1}. Test your code as:

```
for x in [-100,5,1000]:
    print(newton(f,fp,x))
```

which should return the values 3, 3, and 3. Why does one single iteration of Netwon's method find the root in this case?

8.5 Secant Method

When we do not know the derivative a function, we can use the secant method. The secant method provides an improved estimate for the root x_{n+1} from the previous two estimates x_n and $x_n - 1$ which are used to estimate the derivative and then apply Newton's method:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

△ **Jupyter Notebook Exercise 8.4:** Implement the python function:

```
def secant(f,a,b):
    # your code here
    return b,c
```

which, given the function f=$f(x)$, and two previous estimates for the root a=x_{n-1} and b=x_n, returns the updated estimates $b = x_n$ and $c = x_{n+1}$ using the secant method. Test your code as:

```
print(secant(f,0,4))
print(secant(f,5,3))
```

which should return

```
(4, 3.0)
(3, 3.0)
```

8.6 Roots of a Quadratic Function

In this section, we will test your root finding algorithms on a quadratic equation:

$$g(x) = (x - 1)(x - 4) \tag{8.1}$$

△ **Jupyter Notebook Exercise 8.5:** Define a python function `g(x)` which returns the value $g(x)$. Calculate the derivative $g'(x)$ analytically, and define `gp(x)` which returns $g'(x)$. Plot $g(x)$ and $g'(x)$ for $0 < x < 5$. Be sure to include axis labels and a legend.

△ **Jupyter Notebook Exercise 8.6:** Apply five iterations of your function `bisection` to the range (0,2.4). After each iteration, print out a, b, $g(a)$, and $g(b)$. Did you approach the correct root?

△ **Jupyter Notebook Exercise 8.7:** Starting from the value x=2.4, apply five iterations of your function `newton`. After each iteration, print x and $g(x)$ Did you approach a root?

△ **Jupyter Notebook Exercise 8.8:** Starting from estimates a=0 and b=2.5, apply five iterations of your function `secant`. After each iteration, print out a, b, $g(a)$, and $g(b)$. Did you approach the correct root?

8.7 Particle in a Finite Potential Well

Suppose a particle of mass m and energy E is located in a finite potential well of form:

$$V(x) = \begin{cases} V_0 & x \leq -L/2 \\ 0 & -L/2 < x < L/2 \\ V_0 & L/2 \leq x \end{cases}$$

We are going to consider the interesting situation where $E < V_0$, that is, when the particle is bound by the potential. For simplicity, we will also assume the particle is in a symmetric state, such that wave function ψ has the property $\psi(-x) = \psi(x)$. In this case, the solutions to the Schrodinger Equation satisfy the transcendental equation:

$$v \tan v = \sqrt{v_0^2 - v^2} \tag{8.2}$$

where

$$v = L\sqrt{\frac{m\,E}{2\hbar}}$$

and

$$v_0^2 = \frac{m\,L^2\,V_0}{2\hbar}$$

are both dimensionless quantities, related to the particle energy E and the potential energy of the well V_0. If you haven't yet encountered this essential problem from quantum mechanics, you can just take my word about Equation 8.2 for now.

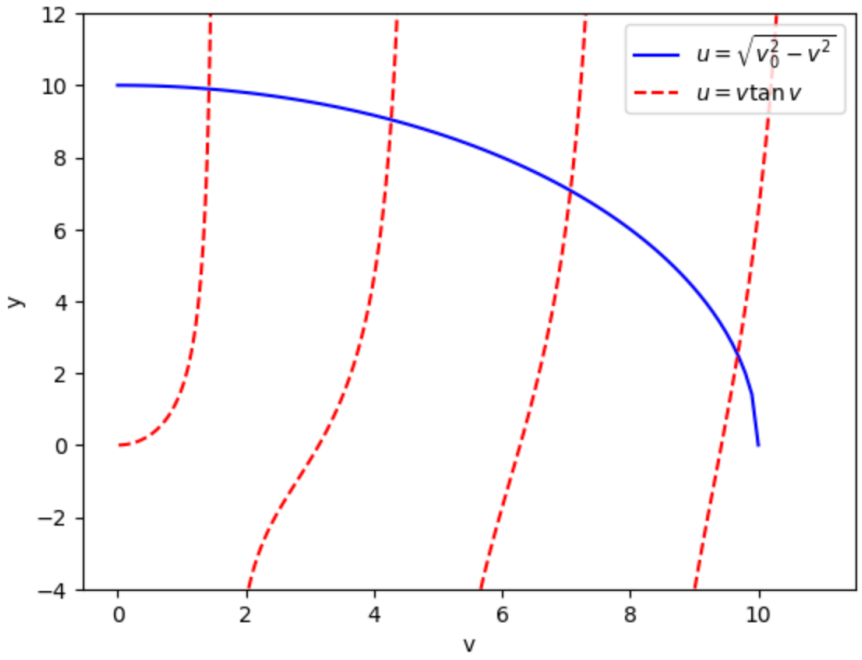


Figure 8.1: Graphical solution to Equation 8.2 for $v_0 = 10$

Most quantum mechanical textbooks suggest that the solutions to Equation 8.2 may be obtained "graphically", by plotting $u = v\tan v$ and $u = \sqrt{v_0^2 - v^2}$ and observing where they intersect. This

approach is illustrated in Fig. 8.1 for $v_0 = 10$. You can see from the plot that there are four places where these curves intersect. In a classical system, the particle could have any $E < V_0$, but in quantum mechanical system, the particle may have only one of the four energy values corresponding to the four intersection points. The number of interesections, and where they occur, depends on the particular value of v_0.

△ **Jupyter Notebook Exercise 8.9:** Reproduce Fig. 8.1. If you attempt to plot the dashed red lines all at once, you will introduce superflous vertical lines from when $\tan v$ switches from $+\infty$ to $-\infty$. To avoid this, plot each continuous region of $u = v \tan v$ separately, in the regions $(-\pi/2, \pi/2)$, then $(\pi/2, 3\pi/2)$, and so on. You may find it useful to remove problematic end points from a numpy array by slicing them off like this `x=x[1:-1]`. Instead of the math formulas, your legend may use the simpler labels LHS and RHS , which refer to the left hand side and right hand side of Equation 8.2. Note: you will have to use 'r-' for each tangent plot to keep it from using different colors. Also, only put a label on the first one, so you don't get a bunch of separate legend entries.

Determining the bound state energy levels of quantum system is of fundamental importance. For just one example, we are able to determine the chemical composition of stars using spectroscopy. The light frequencies which are readily absorbed by each element are determined from the difference in the bound state energy levels. In this section, we will use numerical techniques to accurately determine the values of v which satisfy Equation 8.2. This is equivalent to finding the bound-state energy levels. We will do so by finding the roots of the equation:

$$h(x) = v \tan v - \sqrt{v_0^2 - v^2} \tag{8.3}$$

△ **Jupyter Notebook Exercise 8.10:** Define a function:

```
def h(x):
    # your code
```

which implements $h(x)$ from Equation 8.3 for $v_0 = 10$. Also define a function for the analytical derivative $h'(x)$[1] as

```
def hp(x):
    # your code
```

Test these with

```
for v in [1,1.5,2,4.5]:
    print(np.around(h(v),2),np.around(hp(v),2))
```

This should return the values -8.39, 11.27, -14.17, and 11.94 for $h(x)$ and 5.08, 314.03, 9.57, and 106.41 for $h'(x)$

For each of the following exercises, you will find the third root; i.e. the one between $v = 6$ and $v = 8$, for the case where $v_0 = 10$ In each case, you will continue to iterate until $|h(x)| < .0001$. In each case, print the total number of iterations required and the final value of $h(x)$

[1] Numpy has no secant function so use $\frac{d}{dx} \tan x = \frac{1}{\cos^2 x}$.

△ **Jupyter Notebook Exercise 8.11:** Starting with $a = 6$ and $b = 8$, use the bisection method to find the root. Use $h(a)$ to test the value after each iteration.

△ **Jupyter Notebook Exercise 8.12:** Starting with $a = 6$ and $b = 8$, use the secant method to find the root. Use $h(a)$ to test the value after each iteration.

△ **Jupyter Notebook Exercise 8.13:** Starting with $x = 6$ and your definition of $h'(x)$, use Newton's method to find the root.

8.8 Trapezoid Method

The trapezoid method approximates the definite integral

$$I = \int_a^b f(x)\,dx$$

from the function evaluated at n evenly spaced points

$$I_T = \frac{h}{2}(f(a) + f(b)) + h\sum_{i=1}^{n-2} f(x_i)$$

where

$$h = \frac{b-a}{n-1}$$

and

$$x_i = a + h \cdot i; \quad 0 \leq i < n$$

The truncation error is:

$$I - I_T = \mathcal{O}(h^2).$$

Notice that the sum is over the interior points, which have twice the weight of the end points at a and b.

△ **Jupyter Notebook Exercise 8.14:** Implement the python function:

```
def trap(f,a,b,n):
    # your code
    return sum
```

Which returns the integral of f=$f(x)$ from a to b, using the trapezoid method from n evenly spaced points. Test your code by approximating

$$\int_0^\pi \sin(x)dx$$

with $n = 2, 3$ and 4

```
print(np.around(trap(np.sin,0,np.pi,2),2))
print(np.around(trap(np.sin,0,np.pi,3),2))
print(np.around(trap(np.sin,0,np.pi,4),2))
```

which should output the values 0.00, 1.57, and 1.81. Explain the zero value.

8.9 Iterative Trapezoid Method

In the iterative trapezoid method, we halve the spaces between points during each interation, so that:

$$h_0 = (b-a),\ h_1 = \frac{b-a}{2},\ \ldots\ ,\ h_m = \frac{b-a}{2^m}$$

Note that these correspond to $n = 2$, $n = 3 \rightarrow n = 2^m + 1$ in our function in the previous section.

We then define I_m as

$$I_m = \text{trap}(f, a, b, (2^m + 1))$$

△ **Jupyter Notebook Exercise 8.15:** Write an iterative routine

```
def itertrap(f,a,b,eps):
    # your code
    return I, m # Integral and iteration number
```

which starts at $m = 0$ and continues to call the iter routine with updated values of $n = 2^m - 1$ until

$$|I_m - I_{m-1}| < eps$$

It then returns the integral I and the final value of m.

Use this calculate the definite integral:

$$I = \int_0^\pi \sin(x)\, dx$$

to an accurace of $eps = 10^{-6}$. What is the final value? How many iterations did it take? (Remember, the number of iterations is equal to one more than the final value of m.

8.10 Period of a Pendulum

In this section we will consider the oscillation of a pendulum of length L in a graviational field with magnitude g. For small angles, the period of a pendulum oscillation is approximately a constant value:

$$T_0 = 2\pi\frac{L}{g}$$

For oscillations at larger angles, the period of the oscillation cannot be calculated analytically. However, it can be expressed as a definite integral:

$$\frac{T}{T_0} = \frac{2}{\pi}\int_0^{\pi/2} \frac{1}{\sqrt{1 - k^2 \sin u}}\, du \tag{8.4}$$

where:

$$k = \sin\frac{\theta_0}{2}$$

△ **Jupyter Notebook Exercise 8.16:** Estimate the ratio T/T_0 for $\theta_0 = 1$ using your iterative trapezoid routine. Iterate until the estimated truncation error is less than 10^{-6}. What is the final value of I? How many iterations did it take?

Chapter 9

Lab 9: Monte Carlo Simulation

9.1 Introduction

In this lab, we will introduce the Monte Carlo method, an approach to solving a wide range of problems by repeatedly drawing random numbers. You will use histograms to compare randomly thrown values to their corresponding probability distributions functions. You will measure π and compute one integral using a Monte Carlo method.

9.2 Preparation

In this exercise, we will make use of slicing and boolean masks, which are covered in sections 1.4.1.5 and 1.4.1.7 of the Scientific Python lecture notes. Make sure you are comfortable enough with these concepts to understand the following exercises:

△ **Jupyter Notebook Exercise 9.1:** Run the following snippet, and make sure you understand the output:

```
x = np.arange(10)
print("x       = ", x)
# all but the first element:
print("x[1:]   = ", x[1:])
# all but the last element:
print("x[:-1]  = ", x[:-1])
# the first 4 elements:
print("x[:4]   = ", x[:4])
# elements from index 2 to 6:
print("x[2:7]  = ", x[2:7])
# elements in reverse order:
print("x[::-1] = ", x[::-1])
```

△ **Jupyter Notebook Exercise 9.2:** Run the following snippet,

```
x = np.arange(5)
print("x =          ", x)
print("midpoints:   ", (x[1:]+x[:-1])/2)
```

which demonstrates a trick we will be using later, for calculating the midpoints between each value in x. Note that the first array has five entries, the second has four.

△ **Jupyter Notebook Exercise 9.3:** Run the following snippet, and make sure you understand the output:

```
x = np.array([1,2,5,1,5,1,8,2])
print("x        = ", x)
mask = x > 2
print("mask     = ", mask)
print("x[mask] = ", x[mask])
```

△ **Jupyter Notebook Exercise 9.4:** Run the following snippet,

```
x = np.array([7,2,10,4,2,9,1,3])
mask = x>3
np.sum(mask)
```

which demonstrates a trick we will using later, for counting the number of entries in an array which satisfy a particular condition.

9.3 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:

$$I_{n+1} = (a\, I_n + c) \bmod M$$

Recall that x mod y (coded as `x % y` in Python) is the remainder after integer division (`x//y` in Python). Each I_n is called a seed, and the initial seed I_0 must be provided to start the sequence. 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 continuous.

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 9.5:** Define a function:

```
def parkmiller(i):
    # your code here
    return i # updated value
```

which, given a seed i, returns the next seed in the Park and Miller algorithm. Make sure a, c, and M are integers or the algorithm will not work properly! Check your code by testing that for a initial seed of one, the generator returns a **seed** of 1043618065 after 10000 calls.

△ **Jupyter Notebook Exercise 9.6:** Use your `parkmiller(i)` function to fill a numpy array `xarr` with 5 randomly thrown x values in the range $[0, 1]$. Check you code with:

```
print(np.around(xarr,2))
```

Now that we have seen how randomly number are generated, we will use the standard numpy tool to produce array of randomly drawn values as needed for us:

△ **Jupyter Notebook Exercise 9.7:** Use the `np.random.uniform` to create an array `xarr` of five randomly thrown x values in the range $[0, 1]$. Check you code with:

```
print(np.around(xarr,2))
```

9.4 Probability Density Functions and Histograms

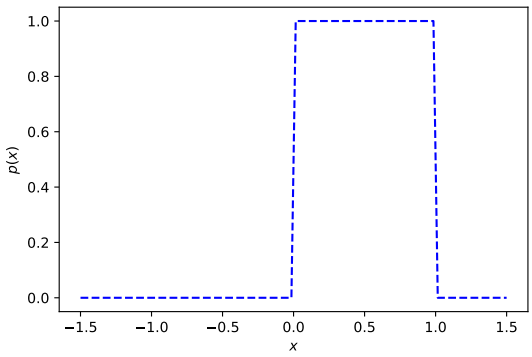


Figure 9.1: The PDF for the process of throwing a random variable uniformly in the region $[0, 1]$.

When I generate ten random numbers from the Park and Miller algorithm, I get the following ten values:

```
[0.6448101  0.32334154 0.40125851 0.95175061 0.07252577 0.94062374
 0.06316977 0.69433175 0.63370098 0.61235022]
```

We say that these ten numbers are thrown (like dice) or drawn (like cards?) uniformly in the interval from $[0, 1]$. How can we describe this process of thowing random numbers in terms of probability? The probability of drawing a particular number, like 0.06316977 is extremely small. If the computer

had unlimited precision, the probability of drawing any particular number would drop all the way to zero. Probability by itself doesn't seem to be very useful here. The solution is to recognize that it is the probability of throwing a number in some region $[a, b]$ which is non-zero. Instead of a probality, we describe this process by a probability density function (PDF), in this case:

$$p(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{9.1}$$

This PDF is illustrated in Fig. 9.1. To find the probability P that we would throw x in some interval $[a, b]$, we integrate the probability density function:

$$P = \int_a^b p(x)\, dx$$

For example, the probability that we throw a number less than one half is:

$$P = \int_{-\infty}^{\frac{1}{2}} p(x)\, dx = \int_0^{\frac{1}{2}} 1\, dx = \frac{1}{2}$$

As are all PDFs, this one is normalized to a total probability of one:

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

△ **Jupyter Notebook Exercise 9.8:** Define a python function:

```
def pdf(x):
    # your code here
    return p
```

which implements the PDF from Equation 9.1. Use it to create x and y values for plotting as:

```
xf = np.linspace(-1.5,1.5,100)
yf = pdf(xf)
```

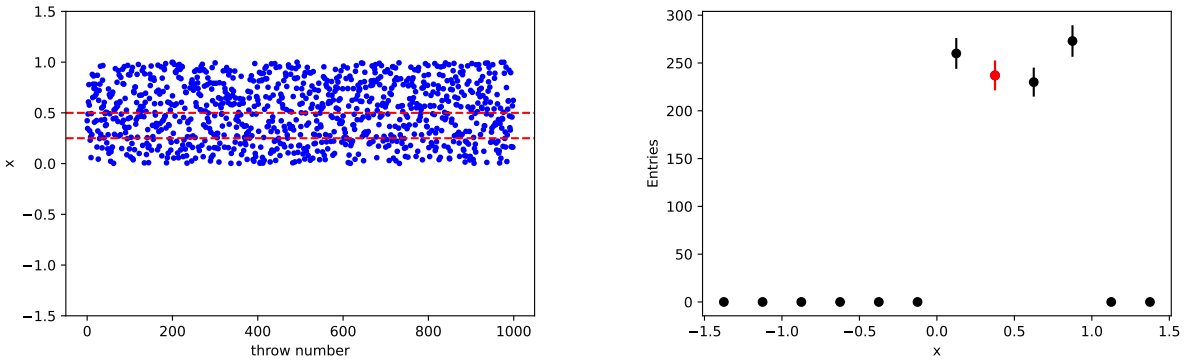
Reproduce the plot in Fig. 9.1. Hint: for this to work, you have to make sure the function `pdf(x)` can handle properly the case that x is a numpy array, and return a numpy array of the same size.

But how can we be verify that our numbers drawn from the Park and Miller algorithm:

```
[0.6448101  0.32334154 0.40125851 0.95175061 0.07252577 0.94062374
 0.06316977 0.69433175 0.63370098 0.61235022]
```

are actually being drawn from the PDF in Fig. 9.1? The tool of choice for seeing the "shape" of a list of values is the histogram, and the process of building one is illustruated in Fig. 9.2. First, let's draw 1000 random values. One way to visualize these values is shown in Fig. 9.2a, which simply plots each value above the throw number (from 0 to 1000). To build a histogram, we devide the x range into small regions, called *bins*. For example, we have a bin from 0.25 to 0.5, as indicated by the dashed red lines. The number of blue points contained within that range is 237. In Fig. 9.2b, we plot the count as the red point. The y-value of the point is the count 237. The x-value is the middle of the bin:

$$(0.50 + 0.25)/2 = 0.375$$



(a) (b)

Figure 9.2: The 1000 measurements of variable x in (a) are used to produce the histogram in (b). The red data point in (b) is the count of the number of entries in range indicated by the red dashed lines in (a).

We continue this process for as many bins as we wish to plot. The result is the entire set of points in Fig. 9.2b which we call a histogram. A histogram is a set of bins, with a count corresponding to each bin.

Notice that each data point in our histogram has an error bar: the verticle line passing through each point in the histogram. A fundamental result from statistics is that the best estimate for the statistical uncertainty on a count N of independent occurances is simply $\sqrt{N}$. The error bars in Fig. 9.2b have been taken as the square root of the count in each bin.

Fortunately, the process of calculating a histogram from an array of values is handled by the python function `np.histogram`, which you will use in the following exercise:

△ **Jupyter Notebook Exercise 9.9:** Run the following code snippet:

```
NTOT = 1000  # total number of events thrown
NBIN = 12    # number of bins in histogram
XMIN = -1.5  # maximum X value
XMAX = 1.5   # minimum X value
# throw NTOT random values uniformly in [0,1]:
xarr = np.random.uniform(size=NTOT)
#create a histogram from the random values in xarr:
# hx:   the histogram counts (length NBIN)
# edges: the bin edges (length NBIN+1)
hx,edges = np.histogram(xarr,bins=NBIN,range=(XMIN,XMAX))
# calculate the center of each bin, for plotting:
cbins = (edges[1:]+edges[:-1])/2
# calculate the error in each bin as the square root of the count
err    = hx**0.5
# plot the histogram, including errorbars, using the errorbar function:
plt.errorbar(cbins,hx,yerr=err,fmt="ko",label="Histogram")
# add the labels
plt.xlabel("x")
plt.ylabel("Entries")
```

to construct a histogram like that of Fig. 9.2b.

There are some lines of particular importance in the code snippet, which you will need to understand to succeed in this lab. This line:

```
hx,edges = np.histogram(xarr,bins=NBIN,range=(XMIN,XMAX))
```

actually creates the histogram for the array `xarr`. It creates `NBIN=12` bins in range from `XMIN=-1.5` to `XMAX=1.5`. The count for each bin is contained in the array `hx` which has length `NBIN`. The edges of the bins are contained in the array `bins` which has length `NBIN+1`. Because they are different lengths, you cannot simply plot `hx` versus `bins`. Instead, we calculate the bin centers with the line:

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

which uses slicing to produce an array of length `NBIN` containing the bin centers. We calculate the statistical error for each point in the histogram as the square root of the count:

```
err   = hx**0.5
```

The line:

```
plt.errorbar(cbins,hx,yerr=err,fmt="ko",label="generated")
```

plots the histogram with errorbars. Sometimes students are confused by the name of the `plt.errorbar` function: it plots both the histogram and the errorbars!

△ **Jupyter Notebook Exercise 9.10:** Modify the code snippet from the previous example to create and draw a histogram from 1000 random values thrown in the range $[0, 1]$ using your Park and Miller algorithm.

Your histogram in the previous exercise should look much like that of Fig. 9.2b. We can also see that it has the same shape as the PDF in Fig. 9.1. But the histogram has a maximum value of around 270, whereas the PDF has a maximum value of 1. This is because the histogram presents a count and the PDF presents a probability density. For N_{tot} total events thrown, we can use the PDF $p(x)$ to predict the number of events N_{ab} in a bin with edges a and b as:

$$N_{ab} = N_{\text{tot}} \int_a^b p(x)\, dx$$

where $p(x)$ is the PDF, N_{tot} are the number of throws. From the mean value theorem we can find a particular x^* in the range $[a, b]$ such that

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

and so we have:

$$N_{ab} = N_{\text{tot}}\, \Delta x \cdot p(x^*)$$

where $\Delta x = b - a$ is the bin size. As a practical matter, instead of calculating N_{ab} for each bin, we simply plot the PDF scaled as:

$$N(x) = N_{\text{tot}}\, \Delta x \cdot p(x)$$

which allows us to directly compare a PDF to the histogram. In our case, the scale factor is:

$$N_{\text{tot}}\, \Delta x = 1000 * (0.50 - 0.25) = 250.$$

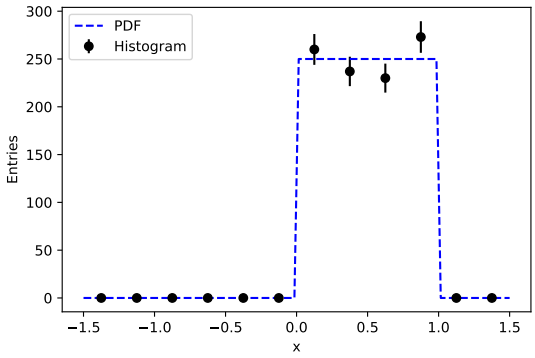


Figure 9.3: Direct comparison of a histogram containing 1000 events thrown uniformly in the range $[0, 1]$ with the corresponding PDF, scaled appropriately.

A comparison of the histogram with the PDF scaled by this factor is shown in Fig. 9.3.

△ **Jupyter Notebook Exercise 9.11:** Add a plot of the scaled PDF to your histogram, to produce a figure like that of Fig. 9.3

9.5 Calculating the value of π

You can be the life of your next party by showing off how to determine the constant π by throwing toothpicks! The procedure is simple: you cut a piece 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. 9.4.

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. 9.5 which was generated with following code:

```
N = 1000 # number of random values to throw
# throw N x and y random variables uniform in [0,1]
x = np.random.uniform(size = N)
y = np.random.uniform(size = N)
# determine which (x,y) position or inside/outside the unit circle:
rsq = x**2 + y**2
inside  = rsq<=1
outside = np.logical_not(inside)
# set aspect ratio to 1 so unit circle looks like a circle.
plt.axes().set_aspect('equal')
# plot inside as blue dots and outside as red dots
plt.plot(x[inside],y[inside],"b.",label="inside")
plt.plot(x[outside],y[outside],"r.",label="outside")
# plot the unit circle:
xfin = np.linspace(0,1,100)
```

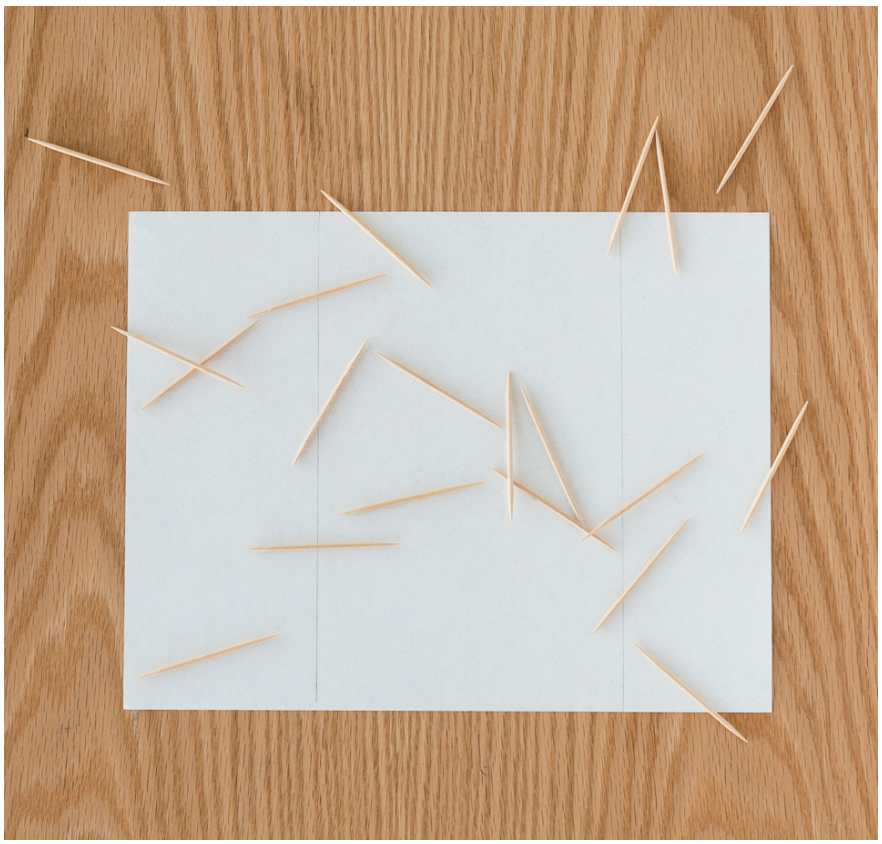


Figure 9.4: Determining π by throwing toothpicks.

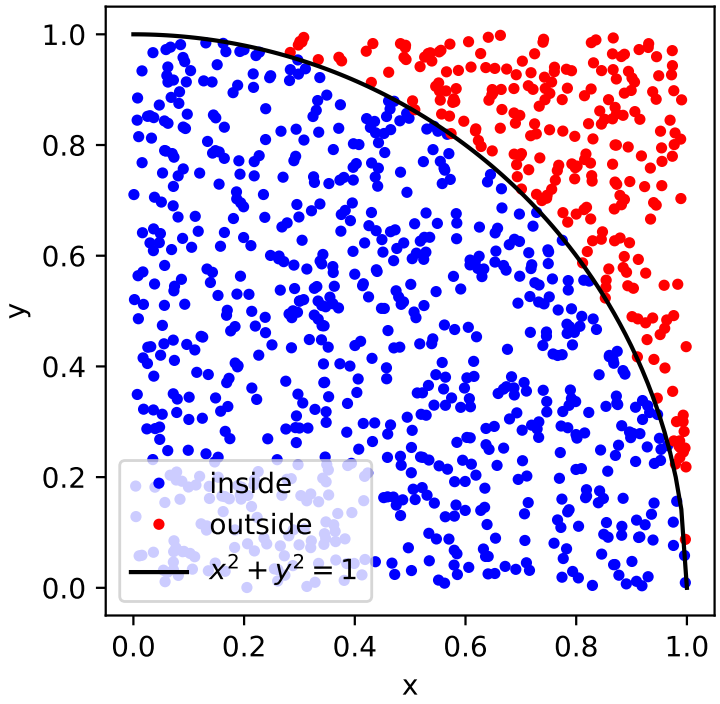


Figure 9.5: Monte Carlo Determination π .

```
yfin = sqrt(1-xfin**2)
plt.plot(xfin,yfin,"k-",label="$x^2+y^2=1$")
# add labels and legends:
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc=3)
```

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 which also landed within the unit circle. Make sure you understand the example code, particular how masks are used to draw the blue and red dots.

△ **Jupyter Notebook Exercise 9.12:** Run the example code. Then, count the number of points inside the unit circle using:

```
n_inside = np.sum(inside)
```

Estimate π using the Monte Carlo method. Hint: based on area, what fraction of total events do you expect to find within the unit circle?

△ **Jupyter Notebook Exercise 9.13:** This is an example of a binomial process, and 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?

9.6 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 9.14:** 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. 9.5 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.

Chapter 10

Lab 10: Ideal Gas

10.1 Introduction

In this lab, you will construct a Monte Carlo simulation of a 2-D ideal gas, and show that the molecular velocities follow the Maxwell-Boltzman distribution.

10.2 Ideal Gas in Two Dimensions

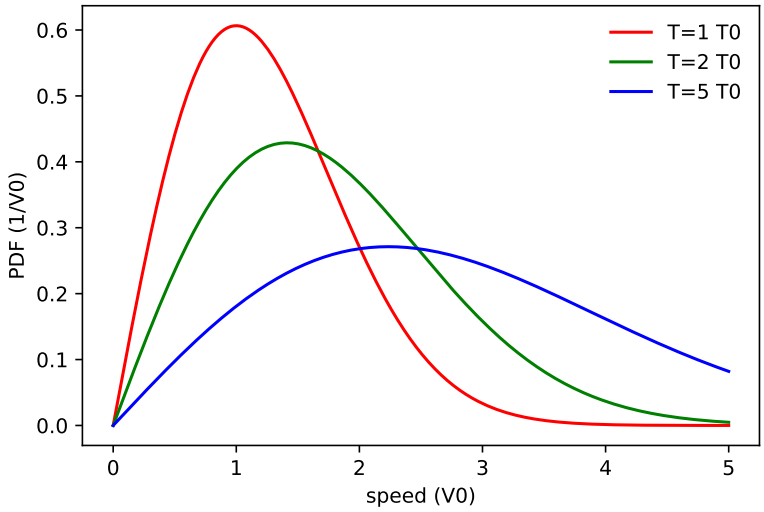


Figure 10.1: The Maxwell-Boltzmann distribution, for three different temperatures, using the system of units chosen for the numerical simulation.

For the remainder of this lab, we will calculate the statistical properties of a simulated ideal gas and compare to the theoretical prediction. Our ideal gas will be composed of molecules with mass m and held at temperature T. To keep things simple, we will consider a two dimensional gas.

A fundamental result from statistical mechanics is that the propability of finding a single gas

molecule in a state with energy E is proportional to the Boltzman factor:

$$P(E) \propto \exp\left(-\frac{E}{k_{\rm B}T}\right)$$

where $k_{\rm B}$ is Boltzmann's constant. Since the ideal gas has no interactions, its energy is purely kinetic energy, and in two dimensions this is given by:

$$E = \frac{1}{2}\, m\,(v_x^2 + v_y^2)$$

So we can conclude:

$$P(v_x, v_y) \propto \exp\left(-\frac{m\,(v_x^2+v_y^2)}{2k_{\rm B}T}\right) = \exp\left(-\frac{m\,v_x^2}{2kT}\right)\cdot\exp\left(-\frac{m\,v_y^2}{2kT}\right)$$

We can infer that the probability density for the component of velocity in x direction is given by:

$$P(v_x) = \sqrt{\frac{m}{2\pi k_{\rm B}T}}\exp\left(-\frac{mv_x^2}{2k_{\rm B}T}\right) \tag{10.1}$$

where we have calculated the normalization constant such that:

$$\int_{-infty}^{+\infty} P(v_x)dv_x = 1.$$

Similary:

$$P(v_y) = \sqrt{\frac{m}{2\pi k_{\rm B}T}}\exp\left(-\frac{mv_y^2}{2k_{\rm B}T}\right) \tag{10.2}$$

Now we shall find the PDF associated with a particular speed v. We consider the infinitesimal probability for a particular velocity

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

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:

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

Integrating over all possible directions θ, we obtain:

$$\begin{aligned} P(v) &= \int_0^{2\pi} P(v,\theta)d\theta \\ &= \int_0^{2\pi}\frac{mv}{2\pi k_{\rm B}T}\exp\left(-\frac{mv^2}{2k_{\rm B}T}\right) \\ P(v) &= \frac{mv}{k_{\rm B}T}\exp\left(-\frac{mv^2}{2k_{\rm B}T}\right) \end{aligned} \tag{10.3}$$

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. It is illustrated in Fig. 10.1. In this lab, we will create a simple numerical simulation of an ideal gas and verify that the velocity of the gas follows this distribution.

10.3 System of Units

Choosing an effective system of units is essential for building a well-behaved numerical simulation. By now you have hopefully learned the wisdom of solving problems analytically using only variables, plugging actual numbers into your equations only if necessary and only at the very end. Numerical techniques generally depend on using actual numerical values, but by making a wise choice for a computational system of units, we can recover the same universality and clarity that variables provide to analytic solutions.

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! If you are living entirely within the numerical system of units, there is no need to even define M_0: this is how you recover universality in numerical simulations.

△ **Jupyter Notebook Exercise 10.1:** Using the numerical system of units:

```
# computational system of units:
M  = 1 # mass of gas particles, M0 = 4.65E-26 kg
T  = 1 # Temperature of gas, T0 = 293 K
kb = 1 # Boltzmans constant
```

and the Maxwell-Boltzman distribution:

```
def mbspeed(v):
    return (M*v / (kb*T))*np.exp(-M*v**2/(2*kb*T))
```

plot the Maxwell-Boltzman distribution for v = 0 to 5 V_0. Compare with Fig. 10.1.

Notice how the relevant velocities for T=1 are near v=1. 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 (only in comments).

△ **Jupyter Notebook Exercise 10.2:** Suppose we set $M_0 = 10^{-15}$ kg instead. Would your plot in the previous exercise need to be changed? What about Fig. 10.1? From Fig. 10.1, read off the peak of the distribution for T = 2 T0. What is that in SI units, assuming $M_0 = 4.65 \times 10^{-26}$ kg? What if $M_0 = 10^{-15}$ kg? Do we even need to specify M_0 if we do not care about the specific SI values? Is choosing a computational system of units with variables near one the numerical analysis equivalent to solving a problem using variables only?

10.4 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. 10.2, 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

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

and the velocity of particle b is $-\vec{u}$.

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

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

The velocity of particle b after scattering is $-\vec{w}$.

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:

$$\Delta\vec{v_b} = (-\vec{w}) - (-\vec{u}) = -\Delta\vec{v_a}$$

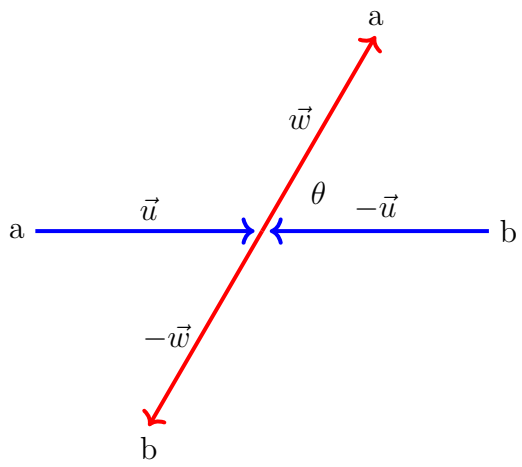


Figure 10.2: The collision model in the center-of-mass: incoming molecule a with velocity $\vec{u}$ collides with the incoming particle b of identical mass with velocity $-\vec{u}$. Particle a is scattered by angle θ and leaves with velocity $\vec{w}$, while particle b leaves with velocity $\vec{w}$. The magnitude of the final and initial velocities are the same: $|\vec{u}| = |\vec{w}|$.

10.5 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:

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

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

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

△ **Jupyter Notebook Exercise 10.3:** Implement the collision model as a Python function:

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

which takes the lab frame veclocitys of particle a (ax,ay) and b (bx, by) and returns the lab frame velocities after scattering by an angle theta. Test your code with some simple cases first, rotations by zero,quarter,half,and three quarters of a full turn:

```
tau = 2*np.pi # Using 2 pi is like saying twice half-way...
# lab frame is cms, incoming on x axis:
print(np.around(collide(1,0,-1,0,0),2)+0)
print(np.around(collide(1,0,-1,0,tau/4),2)+0)
print(np.around(collide(1,0,-1,0,tau/2),2)+0)
print(np.around(collide(1,0,-1,0,3*tau/4),2)+0)
```

which should have the output:

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

And:

```
# lab frame is cms, incoming on y axis:
print(np.around(collide(0,1,0,-1,0),2)+0)
print(np.around(collide(0,1,0,-1,tau/4),2)+0)
print(np.around(collide(0,1,0,-1,tau/2),2)+0)
print(np.around(collide(0,1,0,-1,3*tau/4),2)+0)
```

which should have the output:

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

When debugging code, start with simple test cases. If you get the wrong answer, it will be easy to see where the calculation is going wrong. Once your code works on the simple tests cases, start adding more complexity:

△ **Jupyter Notebook Exercise 10.4:** Test your collision function where the lab frame is not the CMS frame:

```
# boost along x axis, incoming on y axis:
print(np.around(collide(1,1,1,-1,0),2)+0)
print(np.around(collide(1,1,1,-1,tau/4),2)+0)
print(np.around(collide(1,1,1,-1,tau/2),2)+0)
print(np.around(collide(1,1,1,-1,3*tau/4),2)+0)
```

which should output:

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

△ **Jupyter Notebook Exercise 10.5:** Test your collision function with randomly choosen values:

```
# test with random values:
print(np.around(collide(6.24, 1.78, 3.35, 5.98, 3.19),2))
print(np.around(collide(4.07, 4.69, 1.61, 4.54, 2.46),2))
print(np.around(collide(5.28, 2.99, 4.77, 5.22, 3.15),2))
print(np.around(collide(2.84, 5.37, 5.47, 6.16, 1.59),2))
```

which should output:

```
[3.25 5.91 6.34 1.85]
[1.84 5.33 3.84 3.9 ]
[4.76 5.22 5.29 2.99]
[4.58 4.46 3.73 7.07]
```

10.6 Initializing the Simulated Ideal Gas

We will be modeling an ideal gas by direct Monte Carlo simulation of `NGAS` representative molecules. 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 of 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 10.6:** Start with a small number of molecules `NGAS=5`. Use the `np.random.uniform` function to initialize the `vx` and `vy` arrays with random values choosen uniformly in the range (-2,2). Print your arrays and see if the output is reasonable.

Our simulation is going to grow to contain at least 1000 molecules, so printing the values of each molecule is not going to work. So we will want to visualize it as a histogram.

△ **Jupyter Notebook Exercise 10.7:** Increase NGAS to 1000, and plot the v_x array as a histogram:

```
hvx,bins = histogram(vx,bins=20,range=(-5,5))
cbins = (bins[1:]+bins[:-1])/2
plt.errorbar(cbins,hvx,yerr=np.sqrt(hvx),fmt="o")
```

△ **Jupyter Notebook Exercise 10.8:** Plot the v_y distribution as a histogram.

Your histograms should reveal that your distribution of velocities is flat, just like in Fig. 9.3. The gas molecules have not yet reached thermal equilibrium with each other!

10.7 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 10.9:** For `NGAS=1000` moleculres, simulated `NCOLL=10000` collisions using the algorithm described above. Plot the vx and vy distributions as a histogram:

```
hvx,bins = histogram(vx,bins=20,range=(-5,5))
hvy,bins = histogram(vy,bins=20,range=(-5,5))
cbins = (bins[1:]+bins[:-1])/2
plt.errorbar(cbins,hvx,yerr=np.sqrt(hvx),fmt="bo",label="vx")
plt.errorbar(cbins,hvy,yerr=np.sqrt(hvx),fmt="ro",label="vy")
plt.xlabel("Velocity (V0)")
plt.ylabel("Molecules")
plt.legend()
```

10.8 Temperature of an Ideal Gas

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

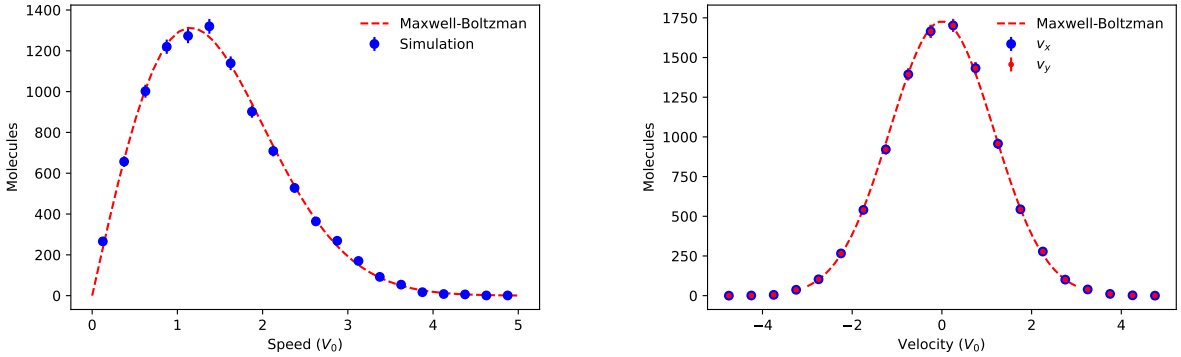
$$k_b T = m \frac{\langle v_x^2 \rangle + \langle v_y^2 \rangle}{2} \tag{10.4}$$

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

△ **Jupyter Notebook Exercise 10.10:** Estimate kT of the gas using Equation 10.4 before and after simulating collisions. The values should remain near the expected value 4/3.

10.9 The Maxwell-Boltzmann Distribution

In this section, you'll reproduce the instructor plots of Fig. 10.3 using your own numerical simulation.



(a) (b)

Figure 10.3: Instructor plots.

△ **Jupyter Notebook Exercise 10.11:** After your simulation reaches equilibrium, plot two histograms, one with v_x and one with v_y, with an approriate range and 10 bins. Directly compare your histograms with the PDF from Equation 10.1, scaled appropriately. The results should resemble the right side of Fig. 10.3, which were produced with `NGAS=10000`.

△ **Jupyter Notebook Exercise 10.12:** 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 10.3. The results should resemble the left side of Fig. 10.3, which were produced with `NGAS=10000`.