Physics 40 Lab Manual

Michael Mulhearn

October 4, 2021

Contents

Chapter 1

Installation of Scientific Python

1.1 Introduction

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

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

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

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

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

1.2 Installing Miniconda3

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

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

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

```
echo $SHELL
```

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

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

1.2.1 Installing under Windows

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

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

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

If prompted, you should choose to:

- Accept the license / terms of use.

- Install for just the current user, not all users.

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

```
conda ——version
```

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

1.2.2 Installing on a Chromebook

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

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

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

1.2.3 Installing Miniconda3 under Linux or macOS

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

```
conda ——version
```

If you see something like:

```
conda 4.9.2
```

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

```
conda: command not found
```

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

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

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

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

- Accept the license / terms of use.

- Do not install for all users, but just one the current user.

- Do allow the installer to issue "conda init".

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

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

```
conda config —set auto_activate_base false
```

Confirm that you have successfully installed conda by typing

```
conda —version
```

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

1.3   Installing the Physics 40 Conda Environment

Make sure your conda is fully up to date with:

```
conda update conda
```

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

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

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

1.4   Starting a Jupyter notebook

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

To activate the phy40 environment type:

```
conda activate phy40
```

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

```
conda deactivate
```

Launch jupyter notebook with:

```
jupyter notebook
```

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

Figure 1.2: The Hello World example Jupyter Notebook.

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

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

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

```
%pylab inline
```

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

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

△ **Jupyter Notebook Exercise 1.1:** Print "hello world" using the python print command.

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

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

your python notebook file (ipynb) even after you submit the assignment. If you have problems, you can reproduce a PDF file from the notebook file, but it is tedious to reproduce your notebook from PDF. If you have problems producing the PDF file, you can submit the "ipynb" file as a temporary work-around, but work with your TA to sort out the problem as quickly as possible.

1.5 Submitting your assignment

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

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

Chapter 2

Binary Numbers

2.1   Introduction

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

2.2   Preparation

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

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

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

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

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

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

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

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

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

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

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

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 commonly represented in hexidecimal (base 16) where two digits of hexadecimal 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

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

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 64206 in decimal, hexadecimal, and binary. Hint: just reuse the carefully formatted print statement from the example above.

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

△ **Jupyter Notebook Exercise 2.7:** 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 significand m is a real number with a finite number of significant figures and the exponent n is an integer. The base B is ten for scientific notation but typically two in a floating point representation. The exponent n is typically chosen so that there is only one digit before the decimal point in the base B, e.g. 3.173×10^{-8} for scientific notation.

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

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

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

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

```
eps = 1.0
while eps + 1 > 1:
    eps = eps / 2
eps = eps * 2
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 signficant figures using 52 bits:

$$m = 1.m_1 m_2 m_3 ... m_{52}$$

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

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

2.5 Other Data Types

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

Python includes strings as a basic type:

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

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

Python includes complex numbers as a basic type:

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

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

△ **Jupyter Notebook Exercise 2.15:** Define a complex number 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:

```python
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:

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

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

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

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

△ **Jupyter Notebook Exercise 2.17:** It's good to read the documentation, but it's a useful skill to figure things out for yourself too! Without looking up the documentation, write a snippet of code to determine for yourself if complex numbers are mutable (like lists) or immutable (like integers). 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

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:

```
1    fa := 0 # set fa to 0
2    fb := 1 # set fb to 1
3  repeat n times:
4      fc := fa + fb
5      print fc to screen
6      fa := fb
7      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:

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

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

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

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

3.5 Geometric Series

The geometric series

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

converges for $|r| < 1$ to:

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

We will demonstrate this numerically.

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

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

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

3.6 Refinements

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

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

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

3.7 Fibonacci Integer Right Triangles

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

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

and

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

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

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

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

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

Chapter 4

The Quadratic Equation and Prime Numbers

4.1   Introduction

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

4.2   Preparation

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

```python
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.

```python
# 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:

```python
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:

```python
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:

```
1     D := b² - 4ac
2    if (D=0):
3        calculate the single solution from quadratic
4        print single solution
5    if (D<0):
6        print no solutions
7    if (D>0):
8        calculate both solutions from quadratic formula
9        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 three more test cases with integer solutions and use them to test your function more thoroughly.

△ **Jupyter Notebook Exercise 4.4:** (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:

```
1      if (a<2):
2          return false
3      i := 2
4      while (i ≤ √a):
5          if (a%i =0):
6              return false
7          i := i + 1
8      return true
```

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

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

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

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

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

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

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

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

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

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

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

4.5   The Lucky Number of Euler

Euler discovered that the formula:
$$k^2 + k + 41$$

produces prime numbers for $0 \leq k \leq 39$. Perhaps you can beat Euler at his own game!
    Consider quadratics of the form:
$$k^2 + ak + b$$

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

$\triangle$ **Jupyter Notebook Exercise 4.8:** (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

Arrays and Plotting

5.1 Introduction

This lab will introduce a fundamental element of scientific python the numpy array, and show how they are used to produce plots.

5.2 Preparation

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

```
%pylab inline
```

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

A Numpy array is a grid of values. Unlike Python lists, the elements of a numpy array all have the same data type, which makes them much more computionally efficient. 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:

```python
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:

```python
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.

5.3   Plotting with Scentific Python

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

```python
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:

```python
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:

```python
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.

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

5.4   The Logistics Map

The logistics map is the recurrence relation

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

with the variable x between 0 and 1. The variable x can be thought to represent the ratio of a population to its maximum possible value. The population increases due to birth and decreases
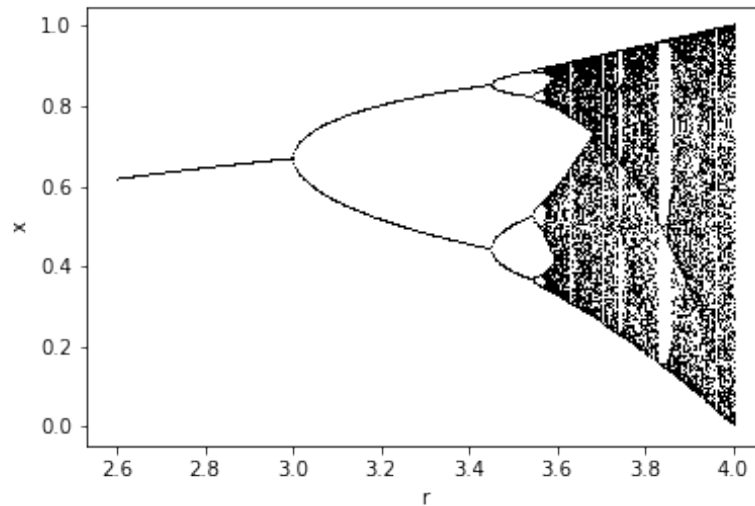
Figure 5.1: Long-term behavior of the logistics map as a function of parameter r.

due to starvation as the population approaches it's maximum value (x near 1). This leads to the non-linear relationship that defines the logistic map. The mapping keeps the variable x between 0 and 1 as long as the parameter r is in the range $[0, 4]$.

The logistics map is frequently encountered as a simple example of a chaotic system emerging from a simple non-linear system. The long term behavior of the logistic map is showin in Fig. 5.2. For each value of r, the x values encountered during 100 iterations after thowring out the first 1000 iterations are shown (i.e. we see x_i for i from 1000 to 1100.)

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

△ **Jupyter Notebook Exercise 5.3:** Implement a function `def logmap(x, r)` which returns the next iteration (x_{n+1}) of the logistic map for parameter r and $x_n = x$. Test your code by showing that for $r = 3.0$ and $x_0 = 0.1$ the next five iterations are: 0.27, 0.5913, 0.725, 0.5981 and 0.7211.
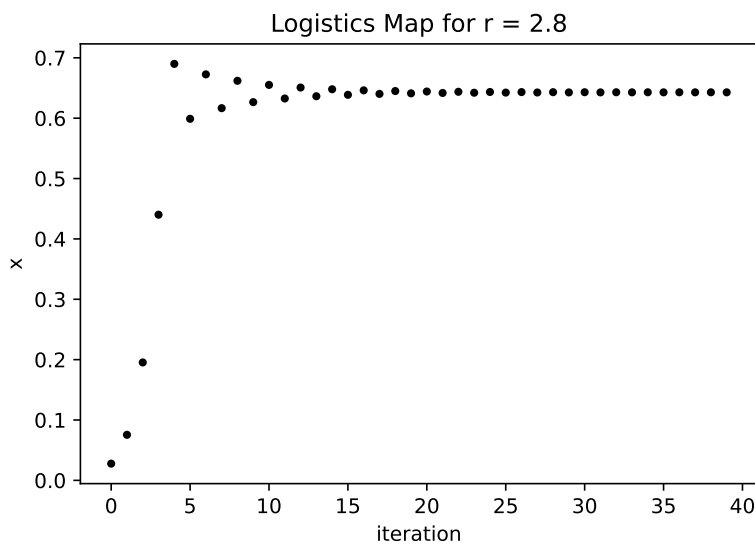
△ **Jupyter Notebook Exercise 5.4:**

Implement a function `def logmap(x, r)` which returns the next iteration (x_{n+1}) of the logistic map for parameter r and $x_n = x$. Test your code by showing that for $r = 3.0$ and $x_0 = 0.1$ the next five iterations are: 0.27, 0.5913, 0.725, 0.5981 and 0.7211.

△ **Jupyter Notebook Exercise 5.5:** For

The long term behavior of the logistics map can be easily modeled in Scientific Python.

- An array of r values is defined.

- An array of x values of the same size as r is defined and initialized to an arbitrary non-zero value (0.01).

- Two example iterations of the logistic map are applied.

- The next two iterations of the values of x are plotted as function of r on the same plot.

Figure 5.2: Convergence of the logistic map for $r = 2.8$

△ **Jupyter Notebook Exercise 5.6:** Reproduce the figure in Fig. 5.2 by doing the following:

- Define two global variables `ITER = 10` and `PLOT = 5`.

- Apply the logistics map `ITER` times by using a for loop.

- Apply the logistics map an additional `PLOT` times, plotting the values of x as a function of r, as in the example, each time.

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

Chapter 6

Differentiation and Projectile Motion

6.1   Introduction

6.2   Preparation

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

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

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

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

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

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

def f(i):
    return i**2

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

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

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

6.3   Numerical Differentiation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 Projectile Motion

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

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

and ensure you get the corret output:

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

```python
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)
```

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

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

△ **Jupyter Notebook Exercise 6.7:** Simulate projectile motion using your function `euler` as described above.

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

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

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

6.5   Projectile Motion with Drag

We can model drag as a deceleration:
$$\vec{a} = -k|\vec{v}|\vec{v}$$

where $k = 0.00622$ m$^-$1 for typical baseballs .

△ **Jupyter Notebook Exercise 6.9:** Extend your simulation to include the effect of drag. Plot the trajectory without drag and the trajectory with drag in the same plot. Include a legend and (as always) label all axes.

△ **Jupyter Notebook Exercise 6.10:** Plot the kinetic energy, potential energy, and total energy as a function of time, just as before. Is total energy conserved?