A Crash Course in Computational Physics

Using Python to Model Physical Systems

Course Prepared by:
William Matzko

Computational Physics Course Notes Based on Python 3

Department of Physics and Astronomy
George Mason University
Fairfax, Virginia
January 2020

Contents

Acknowledgments

This course is based on the introductory computational physics course I took at GMU as an undergrad. I appreciate Adam Jacobs' (who previously taught this workshop) notes and advice on teaching this workshop. Some talking points in various chapters and many derivations originate from various online sources, including Wikipedia, Stackoverflow, and other professors' class notes from different universities.

Most of these notes were written in the late hours of the night. Typos and mistakes are bound to crop up, all of which are my fault. Mistakes will be corrected when found.

Chapter1
Motivation Behind Computational Physics

People have been able to do science for thousands of years without computers. In terms of physics, Isaac Newton, Carl Friedrich Gauss, James Clerk Maxwell, and many others pioneered major fundamental breakthroughs in our understanding of the physical world–and none of them had computers! Is this to say that we do not inherently need computers to do physics? To an extent, yes. However, the mysteries at the forefront of modern physics are much more complex than the 'older' problems these scientists dealt with. Indeed, to make progress in virtually any frontier topic of physics today, one is inextricably forced to make use of computing resources. Often times, such computation is done in the form of simulations. For instance, when assessing our understanding of the early universe, we can design simulations and see how they unfold. We can compare the results of these simulations with observed properties of the universe to determine if we are in agreement. If our simulations match observed reality, then it's a safe bet that we have a good understanding of that area of physics. On the other hand, if they do not match reality, then that is a sign we do not understand the physics of the situation. A concrete example of this is what is called the 'core-cusp' problem. Our best, most detailed simulations predict the density profile of dark matter in galaxies to 'spike' towards the center of the galaxies. However, actually measuring the density profile of galaxies (with gravitational lensing or stellar kinematics) tells us that the profile is 'flat' towards the center. Many solutions have been posed to this problem, but we still don't fully understand why theory differ from observation. Obviously, if it weren't for computers, these types of simulations wouldn't even be possible!

As we will soon see, computers are highly useful even when we aren't pushing the bounds of human knowledge. From tedious tasks humans don't want to do (sorting lists, organizing thousands of files, etc.) or simply can't (solving coupled differential equations), computers come to our rescue and make these tasks not only possible, but comparatively easy. Our goal in this course is to become comfortable with applying computer techniques to solve a wide array of problems, particularly problems in physics. In order to do this, we must first settle on a programming language. For reasons that will be discussed in the next section, I have chosen to use Python for this course.

Chapter2
What is Python?

From python.org, "Python is an interpreted, object-oriented, high-level programming language with dynamic semantics." That's a lot of jargon, so let's break down what this is actually saying.

An implementation of a programming language can be either 'interpreted' (e.g. Python, PHP, Javascript) or compiled (e.g. C, C++, Fortran). Interpreted languages go through a program line by line with a different program (an 'interpreter') and execute each command. Compiled languages are first converted into machine code that a processor can execute. Doing this conversion requires an additional step of 'building' the code so the processor can make sense of it. At face value, it might seem like compiled languages are slower, but that is not generally the case. In fact, compiled languages cut out a lot of the overhead involved in interpreted languages and can run much faster. However, it is generally slower to test/debug compiled languages because they need to be 're-built' every time you want to run them.

Similarly, a language can be 'object-oriented,' 'functional,' or 'procedural.' In the case of object-oriented languages (e.g. Python, C++, Javascript, PHP), information is stored in various 'objects' (like classes, lists, etc.) that can be passed around the code freely. Functionanl languages (e.g. Haskell) are based solely on functions; one main function or 'task' is comprised of many smaller functions. Procedural languages (e.g. C, Fortran) essentially 'break everything down' into smaller sub-procedures until the procedure is simple enough to solve. An advantage of object-oriented programming is that it is easy to modify the code once written–you only need to change one thing in one place (in theory). There is much more to these 'types' of programming languages than I have discussed, but going into detail about it all isn't within our scope.

A language can also either be 'high-level' or 'low-level.' Essentially, high-level languages (e.g. Python, C, C++, Java) are human readable, easy to maintain, and can run on any platform. Low-level languages (e.g. assembly, machine code) are not human readable, are difficult to maintain, and are platform-dependent. One advantage of low-level languages is that they are more memory efficient than high level languages.

The term 'dynamic semantics' simply means that variables can be assigned to any type of data (e.g. integer, string, etc.). Some languages, such as C, require you to explicitly declare the type of data (integer, string, etc.) the variable will hold.

The above is a somewhat technical overview of what exactly Python is, but it can all effectively be summed up by saying 'Python is an easy and powerful programming language to use.' It's also free, which is a huge plus. For these reasons, I have chosen to teach this computational physics course using Python. It should go without saying that if you become comfortable using Python to solve physics problems, you can easily adapt to solve other problems as well. Python is genuinely a fantastic, all-purpose, programming language that is actively used by numerous scientists.

Chapter3 Mathematics Review

The purpose of this section is to knock off any rust you might have and give a quick crash course of mathematics necessary in (computational) physics. I anticipate that you are at least familiar with vectors and calculus, and perhaps linear algebra. I do not expect you to have detailed knowledge of ordinary differential equations; we will just cover the bare basics.

3.1 Vectors

Depending on the context, vector can have a variety of meanings. In our case, it is sufficient to think of a vector as any quantity with magnitude (e.g. 5, 60, 120000) and direction (e.g. left, North, down). I will denote a vector, **r**, by using **boldface**. We can denote components of vectors using unit vectors $\hat{x}, \hat{y}, \hat{z}$, or by listing the components in parentheses ( ) or $< >$. As an example, we could write some vector **r** in terms of its x, y, and z components as

$$\mathbf{r} = 5\hat{x} + 2\hat{y} - 3\hat{z}$$

or

$$\mathbf{r} = (5, 2, -3).$$

Vector addition and subtraction is done component-wise, as if you were just adding/subtracting the regular numbers. There are two ways to perform vector multiplication. The first is the scalar, or 'dot' product, which can be defined in two ways:

$$\mathbf{a} \cdot \mathbf{b} = (a_x b_x + a_y b_y + a_z b_z)$$

$$\mathbf{a} \cdot \mathbf{b} = ||a||\,||b||\cos(\theta)$$

where subscripts stand for the i-th component (e.g. x component), the double vertical bars stand for the magnitude of a quantity, and θ is the angle between the two vectors **a** and **b**. The second way to perform multiplication is with the cross product, which can again be defined in two ways:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$||\mathbf{a} \times \mathbf{b}|| = ||a||\,||b||\sin(\theta),$$

where the vertical bar around the matrix denotes the determinate, to be discussed more in section 3.3.

There are many vector identities and properties. If you find that you are in need of review, I suggest consulting Paul's Online Math Notes:

http://tutorial.math.lamar.edu/Classes/CalcII/VectorsIntro.aspx

3.2 Calculus

We will make extensive use of derivatives and integrals. Recall that derivatives tell us about the slope of functions, and integrals give us the area under functions. If you need a review, I again suggest you consult Paul's Online Math Notes:

http://tutorial.math.lamar.edu/Classes/CalcI/CalcI.aspx

We should only really need concepts from Calculus 1, but knowledge of Calculus II and multivariable calculus is certainly a plus.

A note on notation for derivatives. We can denote the derivative of a function $f(x)$ with respect to x in a few different ways, most commonly

$$\frac{df}{dx} \text{ or } f'(x).$$

Time derivatives are commonly denoted with a 'dot' above the quantity:

$$\frac{df}{dt} = \dot{f}.$$

3.3 Linear Algebra

Linear algebra is full of many interesting concepts, but there are a few I want to highlight. First, is just the notion of a matrix. A matrix has m rows and n columns and is called an $m \times n$ matrix (be careful, the order does matter!). We can think about a matrix simply as a collection of vectors (while it might not be obvious, functions themselves can be vectors). An easy way to see this is to convert a system of equations into matrix form. Let's consider the following system of equations:

$$\begin{aligned} x + y &= 4 \\ 4x - 8y &= 7. \end{aligned}$$

We can write this in matrix form:

$$\begin{bmatrix} 1 & 1 \\ 4 & -8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}.$$

This might seem strange at first, but the two forms are entirely equivalent. This leads us to the notion of matrix multiplication, which is simply the dot product of the i-th row with the j-th column. Carrying out the multiplication yields

$$\begin{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} 4 & -8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$
$$\begin{bmatrix} x+y \\ 4x-8y \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

We can then set the top row of the left hand side equation to the top row of the right hand side and similarly for the other row. It is important to note that you *cannot* generally multiply any two matrices. Suppose we have two matrices that are $m \times n$ and $m' \times n'$. In order for matrix multiplication to be defined, we *must* have the condition $n = m'$. This operation results in a $m \times n'$ matrix. Most matrices we will deal with though are $n \times n$ matrices, or 'square' matrices (having the same number of rows and columns).

The next concept to discuss is the determinant of a matrix. Determinates are needed in the evaluation of cross products, and in determining the inverses of matrices (which will be discussed next). Determinates are only defined for square matrices, and can be computed with the following formulas:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - cd$$
$$\begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \hat{x}(a_y b_z - a_z b_y) - \hat{y}(a_x b_z - a_z b_x) + \hat{z}(a_x b_y - a_y b_x)$$

Determinates of matrices larger than 3×3 are unwieldy and not done by hand. If the determinate of a matrix is zero, then the matrix is said to be 'singular.'

Next we have the notion of an inverse matrix. The inverse of a matrix $\mathbb{A}$ (I use $\mathbb{BLACKBOARD}$ $\mathbb{BOLD}$ to denote matrices) is often denoted $\mathbb{A}^{-1}$. The inverse of a 2×2 matrix $\mathbb{A}$ is relatively easy to calculate:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\det(\mathbb{A})} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where $\det(\mathbb{A})$ is of course the determinate of the matrix. There is a straightforward formula for computing the inverse of a 3×3 matrix, but it isn't intuitive. For higher dimensional matrices, the inverse can be computed using Gaussian elimination (discussed in section (5.1)). There is a quick check in determining if a matrix has an inverse at all. From the above formula, it should be clear that if the determinate of the matrix is zero, then the matrix does not have an inverse. To calculate the inverse of any matrix, we always need to compute 1/det. So, if a matrix is singular, no matter its dimension, then it won't have an inverse.

Lastly, there are three terms I wish to discuss: 'upper triangular,' 'lower triangular', and 'diagonalized.' Generally, these terms only apply to square matrices, but it can be possible to stretch the definition of the first two terms to incorporate non-square matrices. Let's first discuss what a diagonal matrix is. A diagonal matrix is a matrix whose elements are all zero, except for elements along the main diagonal. The main diagonal extends from the top left of the matrix on the bottom right of the matrix. For instance, the following matrix is diagonal:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -7 \end{bmatrix}.$$

The term 'upper triangular' refers to a $n \times n$ matrix whose elements are all zero *below* the main diagonal. Upper triangular matrices are often denoted 'U.' Lower triangular matrices are matrices whose elements *above* the main diagonal are all zero, and are often denoted by 'L.' Below are examples of upper and lower triangular matrices respectively:

$$L = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 4 & 0 \\ 3 & 3 & 1 \end{bmatrix}, \ U = \begin{bmatrix} 3 & 3 & 1 \\ 0 & 4 & 9 \\ 0 & 0 & 7 \end{bmatrix}$$

Note that in both cases, the diagonals must be non-zero.

The importance of linear algebra in computational physics cannot be overstated. Unfortunately, we will only have time to cover a few applications of linear algebra in solving systems of equations. These applications will be further discussed in Chapter 5.

3.4 Ordinary Differential Equations

An ordinary differential equation (ODE) is simply an equation that involves derivatives. Solutions to such equations are certain types of functions. Sometimes, we can look at a differential equation and 'guess' a solution. For instance, it is relatively easy to guess the solution to the following differential equation:

$$y''(x) + y(x) = 0.$$

The interpretation of the above equation is straightforward; we seek a function such that the second derivative of the function added to the original function gives zero. An obvious solution is $y(x) = \sin(x)$. We can easily verify that this is a solution by plugging it in to the above equation and noting that the result does indeed equal zero. We could have just as easily chosen $y(x) = \cos(x)$. This hints at a more important general fact: for a *linear* homogeneous (the equation equals zero) ODE, a linear combination of solutions is also a solution. So, the above equation also has the solution

$$y(x) = c_1 \sin(x) + c_2 \cos(x)$$

where c_1 and c_2 are constant numbers. In fact, this is the most general solution to the above ODE–it isn't always enough to just put $\sin(x)$ or $\cos(x)$ (or whatever other function you find) as *the* solution–you should always try to consider the most general solution possible so you don't lose any information.

In many instances though, we cannot reasonably guess a solution. There are many methods one can use to solve ODE's, but a common, easy (and the physicist s favorite) is a technique called separation of variables. This technique is applicable if we can move all terms involving x on one side, and all terms involving y on the other side. This is best illustrated through an example. Suppose we have the ODE

$$y'(x) = ky,$$

where k is a constant. We seek the derivative of a function that is equal to the original function times a constant. This is something we could easily guess as well–it's just an exponential function $y(x) = e^{kx}$. But, we can show this using the technique of separation of variables. Since $y'(x) = dy/dx$, we can write the above ODE as

$$\begin{aligned}\frac{dy}{dx} &= ky\\ \frac{dy}{y} &= kdx.\end{aligned}$$

where I have just multiplied both sides by dx/y. To some of you, this might seem strange. You have probably been told that dy/dx is not a fraction and shouldn't be treated as such, but that's not entirely correct. There is a complicated rabbit hole here, but we won't go there. Just take my word that you can treat dy/dx as a fraction–it works after all. We can now just integrate both sides of the equation:

$$\begin{aligned}\int \frac{dy}{y} &= k \int dx\\ \ln(y) &= kx + C\end{aligned}$$

where C is of course our integration constant. Raising both sides to the power of e to clear the natural log, we obtain

$$\begin{aligned}y &= e^C e^{kx}\\ y &= ce^{kx}\end{aligned}$$

where c is just an arbitrary constant (e raised to an arbitrary constant is still an arbitrary constant). Thus, we have found our solution to the above ODE. Evidently, the most complicated thing we need to do when using this technique is integrate. It should come as no surprise that the integrals can become very hard depending on the situation, and in some cases are simply not possible to solve. In that event, we must use numerical techniques, which will be discussed in Chapter 11.

In this course, it won't be necessary to employ the use of more complex strategies to solving differential equations. Most of the work will be done numerically, which will generally require us to separate variables in the manner we did above. Still, if you're interested on other techniques, I suggest you consult Paul's Online Math Notes:

http://tutorial.math.lamar.edu/Classes/DE/DE.aspx

Chapter4
Basics of Python

At last, we are ready to begin coding with Python. It is difficult to find a good starting place. When someone first learns a programming language, the hardest part is often 'thinking like a computer.' Computers do exactly what you tell them to do, nothing more, nothing less. You need to translate the ideas in your head into simple, methodological, instructions a computer can follow, and it's not always clear how to do this. Therein lies the challenge of programming. The other hurdle to overcome is syntax. You may know how to communicate your ideas to a computer, but you need to figure out how exactly to tell it. For example, if you know how to populate an array with random numbers in one programming language, you can probably figure out how to do the same thing in other programming languages–the basic idea is the same. However, the exact way you do that varies from language to language. Some languages require colons (':') in certain areas, while others do not. Some require 'end' statements in certain areas, while others do not. So on and so forth. The good news is, most languages have good documentation that enable you to look up whatever syntax you need. Python is no exception. The only drawback is that the documentation can be overwhelming, as it typically discusses a lot of information very quickly. One great feature about Python is its popularity, which means you can often Google your problems and someone, somewhere will have the same issue with you (and might have found a fix!). For future reference, official Python documentation can be found here:

https://docs.python.org/3.6/

As mentioned before, it can also be beneficial to just Google any error or question you might have. For example, if you want to know how to create a dictionary in Python, you can simply Google "How to make a dictionary in Python." You'll find many sources on this topic–many of which are pretty good!

There are subtle differences between Python 2 and Python 3. Throughout these notes, and in class, I will be using Python 3 simply because it is the more up-to-date version of Python. I strongly recommend that all of you use Python 3 instead of Python 2!

The only way to get good at programming is to practice. It's important to not get hung-up on writing the perfect code. For now, just focus on writing programs that work and make sense to you. For now, I'll give an overview of the basic syntax/operations/functions in Python.

4.1 Operations

The 4 basic operations addition, subtraction, multiplication, and division are readily available in Python via '+', '−', '∗', '\' respectively. To raise something to a power, we use '∗∗'; Notice that we do *not* use a carat ('ˆ'), as is common in other languages. Trig functions and logarithms are not strictly recognized in Python. In other words, typing cos(0) will not return 1, it will return an error saying 'cos' is not defined. To use functions like these, you will need to import something called a library function, which will be discussed shortly. One last important operator is the modulus operator '%.' This will divide one number by another and return the remainder of the division. An example would be

6%3

which will return 0, because 3 goes evenly into 6. If we used 4 instead of 3, then the value returned would be 2, because 4 goes into 6 once and there is 2 leftover.

4.2 Data Types

There are various types of information Python can hold, called data types. Many of these are familiar: integers, floats (numbers with digits after a decimal point), complex numbers, strings (plain words), and booleans (e.g. True/False, 0/1, etc.). Python is usually smart enough to recognize the data type of something you input. For instance, Python will interpret 3 as an integer, but 3.0 as a float. Strings are input using either single or double quotes. For example 'spider' or "The dog is cute" are appropriate ways to declare words and/or phrases. To determine what data type Python thinks something is, you can use the type() function. As an example, you can write

type(8.00)

to check the data type of '8.00.' This will be important when we discuss comparisons in Python. You can change the data type of most objects in Python easily. You can use int(), float(), and str() to convert something to an integer, floating point number, or string respectively.

4.3 Variables, Print Statements & Comments

Variables are incredibly useful in Python. They effectively act as a 'short cut' for longer expressions, in the same sense that you can define a variable **r** to be short hand for $x\hat{x}+y\hat{y}+z\hat{z}$. Variables can store any sort of data type you want, and can be named almost anything you like. For instance, let's say we want to include the string 'Oops, you selected an invalid option! Please try again.' in multiple places in our code. It is a pain to write that out, so let's just make a variable that references it. We could simply assign that phrase to the variable 'x' by writing x = ‘Oops, you selected an invalid option! Please try again.’

Now, whenever we write ‘x’ in our code, we are actually writing this entire string! We can change the content of our variable ‘x’ whenever we like (the previous information stored in the variable will be erased). We could have just as easily chosen ‘y’ to be the variable, or almost anything else. A word of warning, though. It is good practice to use variables that describe what they are holding. For instance, instead of ‘x’, we might want to use err_msg. Not only does this make the code more readable, it will save you the headache of figuring out what you were thinking when you look at your code months later!

As I’ve hinted at, we are free to name variables almost anything, but there are some restrictions. They are as follows:

- A variable must start with a letter or underscore (‘_’)
- A variable cannot start with a number
- A variable can only contain upper/lower case letters, numbers 0-9, and underscores
- Variables are CaSe SeNsItIvE (age, Age, and AGE are three different variables!)

Of course, you can use any operations on a variable. Python has specific reserved, or built-in, functions that you cannot name your variables as. For instance, you cannot name a variable max or min, as Python will think you’re trying to find the maximum/minimum of something.

One of the most useful functions in Python (and coding languages in general) is the print() function. This function will display whatever information you choose to print in the console. This is helpful in debugging your code and returning certain information about your program to its users. As an example, if we assign the variable ‘x’ to the string we did above and write print(x), we would see ‘Oops, you selected an invalid option! Please try again.’ appear in our console.

We can also be more verbose in our print statements. We can add in whatever explanatory text we want to our print statement so we know exactly what it is we’re reading in in the console (this is helpful if we have many print statements in our code). This is accomplished via print statement formatting. For example, we could write

print(”The error message is {0}”.format(x))

which returns

The error message is Oops, you selected an invalid option! Please try again.

There are two things to notice here. The first is the .format at the end of the quote. This is necessary to tell Python what we want to put inside our print statement. The second is the {0} which tells Python where we want that information in our print statement. The 0 is a positional argument, which will put the *first* object given in the .format() statement at that location. If we used 1, it would try to put the second object in the .format() statement into that location, but would fail because there is no second object in that .format() statement we wrote. This might be a little confusing, so let's look at another example. We can also use the {} to format the number of digits we want in our print statement if we are dealing with decimal numbers. For instance, we can write the following code:

```
mean, std = 2.53928472, 0.24838239
print("The mean is {0:.2f} and the standard deviation is {1:.2f}".format(mean,std))
```

which will return

```
The mean is 2.54 and the standard deviation is 0.25
```

The colon tells Python that we are about to specify a formatting that is not the default format. The .2 tells Python that we are formatting to two decimal places and the f says we are displaying a floating point number. There are many more options for print statement formatting, but we'll address those on an as-needed basis.

An underrated feature of all programming languages is the comment feature. This feature allows you to write in plain text without worrying about screwing up your code. In effect, you can tell people what a line of code does in plain English, directly in the code, without generating syntax errors. In Python, this is achieved by simply using '#' and then typing your commend. Everything after the # is effectively ignored when Python reads your code. For instance, we can comment the above snippet of code as follows:

```
mean, std = 2.53928472, 0.24838239 #assign mean and std
#print the values
print("The mean is {0:.2f} and the standard deviation is {1:.2f}".format(mean,std))
```

and the output will be the same. Many people do not think it is necessary to comment their code, but they are wrong. Similar to making appropriate, descriptive variable names, commenting your code enhances its readability and communicates to other people who might be trying to modify your code what the purpose of a line or section is. In general, comments should be concise–they shouldn't span multiple lines. I encourage you all to comment you code as thoroughly as you can, as this will help with developing and debugging your code.

4.4 Data Structures

There are four types of data structures I will discuss here: lists, arrays, tuples, and dictionaries. It should be noted that Python does not have arrays by default and must be made with a package called NumPy. I will discuss packages/libraries more in a later section, but it will suffice to say that NumPy is abbreviated in code as np, and if we write np.array() it means that we are converting something to a NumPy array. First, let's discuss lists.

A list is exactly what it sounds like, a collection of objects and/or variables in one place. Lists are made in python by rectangle brackets [ ]. You can manually add objects into your list (integers, floats, strings, etc.) by encasing them between brackets and separating them with commas. For example, let's make a list called 'data' and have it store a string, integer, and float. This is done by writing

data = [2, 5.39, 'red']

We can access specific elements in the above list. Note that Python lists (and NumPy arrays) are zero-indexed, meaning that the first object in a list/array has an index of 0, the second object has an index of 1, and so on. This can be confusing, but not much we can do about it. To access a specific object in a list, we simply put square brackets next to the variable holding the list, and within the brackets indicate which element you want. Coding this looks like:

data[0], data[-1]

Here, data[0] will return 2, and data[-1] will return the last object in the list, which is 'red.' We can also replace specific elements in lists using the same technique. If we write data[0] = 9, then the first element in data, 2, will be replaced with 9. We can also obtain the position of an object in a list by using the .index argument. This is coded like

data.index(5.39)

which will return 2. If we try to use this method to find an object that is not in the list, Python will spit out an error telling us that what we're looking for is not in the specified list. If we have duplicate values in a list and use the .index() argument, Python only returns the index of the first duplicate element. You can specify an additional argument in .index() after you input what element you want to find that sets the 'starting element' Python will begin at when searching your list. For example, data.index(4.5, 2) will return the index of the first occurrence of 4.5, ignoring the first two elements in the list.

An important aspect of lists (and arrays) in Python is something called 'slicing'. Essentially, we can 'break up' or 'slice' a list to obtain specific sub-sections of that list. Suppose we have a list called *data*. Then we can use the following syntax to obtain various subsets of that list:

```
data[start:stop] # items start through stop-1
data[start:] # items start through the rest of the array
data[:stop] # items from the beginning through stop-1
data[:] # a copy of the whole array
data[start:stop:step] # go through using some step size
```

Some examples of this are

```
a[::-1]    # all items in the array, reversed
a[1::-1]   # the first two items, reversed
a[:-3:-1]  # the last two items, reversed
a[-3::-1]  # everything except the last two items, reversed
```

Slice notation can be very confusing when you're first learning it. It's always helpful to print out exactly what you're slicing when you aren't sure what you're doing!

NumPy arrays are very similar to lists. The key difference is that you can perform operations on arrays, while you cannot perform operations on lists. For instance, let's make the list data2 = [3, 5, 8]. If we write data2/2, Python will return an error saying something about unsupported operand types between list and int. In other words, we can't divide lists by integers. If we want to perform such an operation, we will first have to convert data2 to a NumPy array. This can be done by typing

```
data2 = np.array(data2)
```

which will update the variable data2 to a NumPy array. Now, if we write data2/2, all of the elements in our array will be divided by two. We can access individual elements in arrays in the same way as lists. Unfortunately, NumPy arrays do not support the use of .index(). Instead, we can use a different argument, from NumPy, called np.where(). The array name should be listed first, then you should use a double equal sign, '==', followed by the element you want the position of. An example would look like:

```
np.where(data2==5)
```

which will return the (rather strange) result

```
(array([1], dtype=int64),)
```

This is telling us that 5 is the second element in the array (remember, these arrays start at zero) and that the data type (dtype) of that element is a 64 bit integer. Notice that this result is encased in parentheses, ( ), and is actually a tuple data type (which will be discussed next). If we just want to get the index itself, and not the extra information, we must access the first element of this tuple. The method for doing so is the same as for lists; we just write np.where(data2==5)[0]. This returns

```
array([1], dtype=int64)
```

and is a NumPy array data type. We must then access the first element of this array, which is done the same way. So, to get only the index, we must write

```
np.where(data2==5)[0][0]
```

This 'nesting' can get a little confusing, and is a common source of error. Make sure you know what these arguments are returning!

The next data type to discuss is a tuple. Tuples are denoted with parentheses and are 'immutable.' This means that once a tuple is made, the content of it cannot change. Recall that we could replace any element in a list with a different element at any time. Such a thing is not possible with tuples. This is useful if we want to store static values in a safe place, so they can't accidentally be changed later in a program. Other than than being immutable, tuples aren't significantly different from lists in Python.

The last data type to discuss is a dictionary. With dictionaries, we can associate various keywords with sets of objects (numbers, strings, etc.). Dictionaries are encased in curly brackets, { }, and can be written as follows:

fruit = {'apple':('red', 2), 'banana':('yellow', 3)}

Dictionaries are indexed by 'keys,' meaning that each key in a dictionary corresponds to some set of values. The keys in the above dictionary are 'apple' and 'banana.' Each key has a color and number associated with it. We can access the keys and corresponding elements in a familiar way:

fruit['apple']

returns the tuple

('red', 2)

and accessing these elements is done the same way as before. Of course, we could have specified a list using [ ] instead of a tuple. We can also make nested dictionaries in this fashion. The syntax for adding a new key to a pre-existing dictionary looks like

fruit['oranage'] = ('orange', 4)

In general, dictionary keys can be any non-mutable object, such as a string or a number. Dictionary keys cannot be lists, for instance.

There are other data structures used in Python, such as sets, but it's unlikely that we will need to use those. The above information is not intended to be an exhaustive description of these data types; rather, it is meant to be a starting point. You will find out how to work with these data types more as we do practice exercises and as you work on your projects. Part of the goal is to make you comfortable and efficient with looking up relevant information on the internet. Everyone eventually gets stuck somewhere, knowing how to un-stuck yourself is a crucial skill to develop!

4.5 For and While Loops & If statements

For loops, while loops, and if statements are critical parts of every programming language. It is important that you are comfortable using each of these. Let's look at for-loops first.

A for-loop can be thought of as a 'counting cycle.' They are useful when we want to repeat a code a certain number of times. The basic syntax of a for loop is as follows:

```
for <variable> in <something>:
    <code here>
```

In place of <variable>, it is common to just use a single letter, such as i or x. This variable can be though of as a looping index or element. This will become clearer in a moment when I give an example. In place of <something>, there are two common arguments. The first utilizes the range() function in Python. range() by itself isn't particularly useful, but when used in a for-loop in this manner it specifies a numerical bound to loop over. One implementation of this is printing out the first n numbers in rapid succession. This is done by writing

```
for i in range(0,4):
    print(i)
```

which will return the output

```
0
1
2
3
```

Notice that this output does *not* include the upper bound of our range() argument. Also notice the variable we chose to print–we must use the preceding variable to utilize the range() argument in this manner. Instead of range(), we can specify a list, or array, or tuple, to loop through. Such a thing would look like this:

```
data = [3.4, 8, 16, -6]
for i in data:
    print(i)
```

which will return the following output:

```
3.4
8
16
-6
```

Instead of accessing the elements in the list *data* directly, we could have used the range argument and wrote print(data[i]). The output is the same regardless of the method, but different situations may find one method easier then the other.

In both of the above cases, you must observe **where** I have put the print statement–it is indented 4 spaces from the left margin. **THIS IS ABSOLUTELY CRITICAL AND IS A DEFINING FEATURE OF PYTHON**. All loops, if-statements and functions (discussed later) must have their arguments indented in order for Python to understand your code. To be fair, we could have used 3 spaces, or 5 spaces, but most style guides recommend 4 spaces. Generally, I don't care about your coding style. However, I must insist that you use 4 spaces whenever you indent something. **Never use tab to indent**. Unless your code editor is set up in a particular way, tab-indents will cause immense headaches and are not worth dealing with. You have to trust me on this.

This is the basic idea of a for-loop. It enables you to loop through some list or range of numbers, and is incredibly useful. We will have many examples that make use of for-loops.

The next type of loop to discuss is a while loop. While loops are similar to for-loops, but instead of running for a specific interval, they run until a condition is met. This condition can be anything, and is specified during the creation of the while loop. For instance, if we want to print the integers between 0 and 5 using a while loop, we could write:

```
z = 0
while z <= 5:
    print(z)
    z = z + 1
```

which gives the output

```
0
1
2
3
4
5
```

Note that we must first initialize a value of z, otherwise Python will complain that it doesn't know what z is. You'll notice that there is a funny statement in the while loop: $z = z + 1$. Mathematically, this statement doesn't make much sense–it's an equation with no solution. However, coding-wise, this statement makes perfect sense. It tells Python to take whatever the value of z is and add 1 to it. This must be done if we want to print different values of z and terminate the loop. Had we not included this line, we would be stuck in an infinite loop that prints 0. If you ever find yourself stuck in an infinite loop, or otherwise want to stop your code from running, you can just press CTRL + c in your Python shell to halt execution of a program.

Note that if we want to generate the same output with a for-loop, we would just write:

```
for i in range(0,6):
    print(i)
```

While loops are useful when you don't know how many times you need to iterate over something. Of course, depending on the task, one loop might be better than the other.

Lastly, we discuss the if-statement. This is another incredibly useful tool in Python (and other languages). The if-statement checks to see if a condition is met or not. If that condition is met (or not), a subsequent block of code is (or isn't) executed. The best way to see this is with an example. Suppose I want to assign the variable x to some integer, and if the integer is positive I print ”Positive!” and if it's negative I print ”Negative!”. This can be done by writing:

```
x = 3
if x >= 0:
    print("Positive!")
elif x < 0:
    print("Negative!")
else:
    pass
```

Of course, this code will return the string "Positive!". The 'elif' statement is not always necessary. If you have secondary conditions, you can use elif to enforce them instead of making a new if-statement altogether. In principle, you can have an unlimited number of elif statements. The 'else' statement is for when none of your conditions are met. If Python goes through and determines that the condition specific in the if statement and elif statement are not met, then it will execute the block of code in else:. It is not required to include an else statement. If your conditions are not met, Python will simply do nothing and move on to the next bit of code to execute. The reserved word 'pass' is just short for 'do nothing.' When Python comes across this statement, it will not do anything. This is actually useful, because in every single if statement (or loop) you write, you must have *something* inside the statement/loop.

If-statements, for-loops, and while-loops are critical programming tools, so it is imperative you become comfortable using them. We will have plenty of practice with these during your projects and exercises.

4.6 Matrices

Python doesn't really have an explicit matrix object, but we can effectively make a matrix by using nested lists. For instance, if we wanted to make a 3×3 matrix whose elements are 1-9, we would write:

```
m = [[1,2,3], [4,5,6], [7,8,9]]
```

where I have assigned the matrix to the variable m. This is effectively a matrix; we can access each row and individual element using the same techniques discussed in section (4.4). We can also make matrices uses NumPy arrays. We can simply write m = np.array(m) using the above matrix made of lists to obtain a 'matrix' as a NumPy array. In addition to being able to do operations on the matrix now, if we print m it will also be formatted like a matrix. By this, I mean the output of a print statement looks like a proper matrix as opposed to a single line of nested lists:

```
m = [[1,2,3],[4,5,6], [7,8,9]]
print(m, type(m))
m = np.array(m)
print(m, type(m))
```

gives us the output

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]] <class 'list'>
[[1 2 3]
 [4 5 6]
 [7 8 9]] <class 'numpy.ndarray'>
```

There are other ways to obtain matrices if we don't want to write them out explicitly. Indeed, if we want a 10×10 matrix, it is prohibitive to write it explicitly as we've done above. Let's suppose we want to make such a 10×10 matrix whose elements are 1-100. One way we could do this is by writing:

```
m = [] #initialize list
for i in range(100):
    m.append(i+1) #add elements to list

m = np.array(m) #turn to array so can use reshape
m = m.reshape(10,10)# turn m into 10x10 matrix

#m = np.reshape(m, (10,10)) is equivalent syntax
print(m)
```

which will give us the output

```
[[  1   2   3   4   5   6   7   8   9  10]
 [ 11  12  13  14  15  16  17  18  19  20]
 [ 21  22  23  24  25  26  27  28  29  30]
 [ 31  32  33  34  35  36  37  38  39  40]
 [ 41  42  43  44  45  46  47  48  49  50]
 [ 51  52  53  54  55  56  57  58  59  60]
 [ 61  62  63  64  65  66  67  68  69  70]
 [ 71  72  73  74  75  76  77  78  79  80]
 [ 81  82  83  84  85  86  87  88  89  90]
 [ 91  92  93  94  95  96  97  98  99 100]]
```

There are two new arguments/functions I have used here. The first is .append(), which is only applicable to lists. This .append() function will add a new element to the list in the last position. The other is np.reshape(), which simply allows us to change the shape of an array to something else.

When possible, you should always use NumPy arrays as your matrix data type. This makes your life significantly easier.

4.7 Functions

Imagine you write a script that generates a list of odd numbers. Suppose this script is 10 lines long and there are multiple places in your code where you need to generate a list of random numbers (the length of the list not always being the same). You could, in theory, copy/paste your script into your code verbatim whenever you needed to make a new list. This is, however, ugly and inefficient. You've already made the script, so it doesn't make sense to keep rewriting it and slightly tweaking it for particular situations. Instead, we can use something called a function to entirely eliminate this redundancy. Functions are generally useful for containing large blocks of code that can take various arguments depending on the situation. They are designed, in part, to make your code more compact and readable. The code inside a function will only be executed when that function is called, so you are free to put as many functions in your code, wherever you want, without worrying about code being executed out of order. The basic syntax of a function is as follows:

```
def <name>(<arguments>):
    <code>
    return <something>
```

Functions always begin with def. This tells Python that you are beginning a function. The function name follows the same rules as variable names, but can otherwise be anything. If applicable, you can put your arguments of the function inside the parentheses ( ). Normally, if you write something like z = z + 1 in your code without defining z, Python spits out an error complaining z is not defined. However, if we write this exactly line in our function and make z and argument of that function, Python will not complain. This is because z is some parameter/argument we must specify when we call the function. Functions can have any number of arguments. When you call your function, you must keep in mind the position of the arguments in the function. For instance, suppose we have the following script

```
def add(a, b):
    a = a + 5
    b = b + 2
    return a, b

a, b = add(1, 2)
```

Python will *always* interpret the first argument in add(a,b) as a and the second as b. So, be careful! You don't want to pass the correct arguments in an incorrect order into your function, otherwise you could get the wrong results. Lastly, note the line that has return in it. Functions usually, but don't have to, return a value (or list, array, etc.). If you do not include a return statement and call your function, the code inside your function will execute, but you won't really be able to use anything that block of code generates (unless you declare global variables, but don't fiddle with those).

Sometimes, we want to define basic mathematical functions like x^2. We could do so using the above method of making functions, but there is a better, more compact, way. This method uses something called lambda functions, which is basically shorthand notation for what we've done above, except lambda functions can only take one expression. The basic syntax for a lambda function is

```
lambda <arguments>: <code>
```

As an example, suppose we wanted to write x^2 in this manner. Then we would simply write:

```
x_squared = lambda x: x**2
```

Of course, I could have chosen whatever name I wanted in place of x_squared. One practical use of functions (particularly lambda functions) is when we are fitting curves to data. To do so, we must specify the general function we are trying to fit, and that function must be written using Python functions. If we're fitting many different types of trends, then this shorthand notation for functions is quite useful.

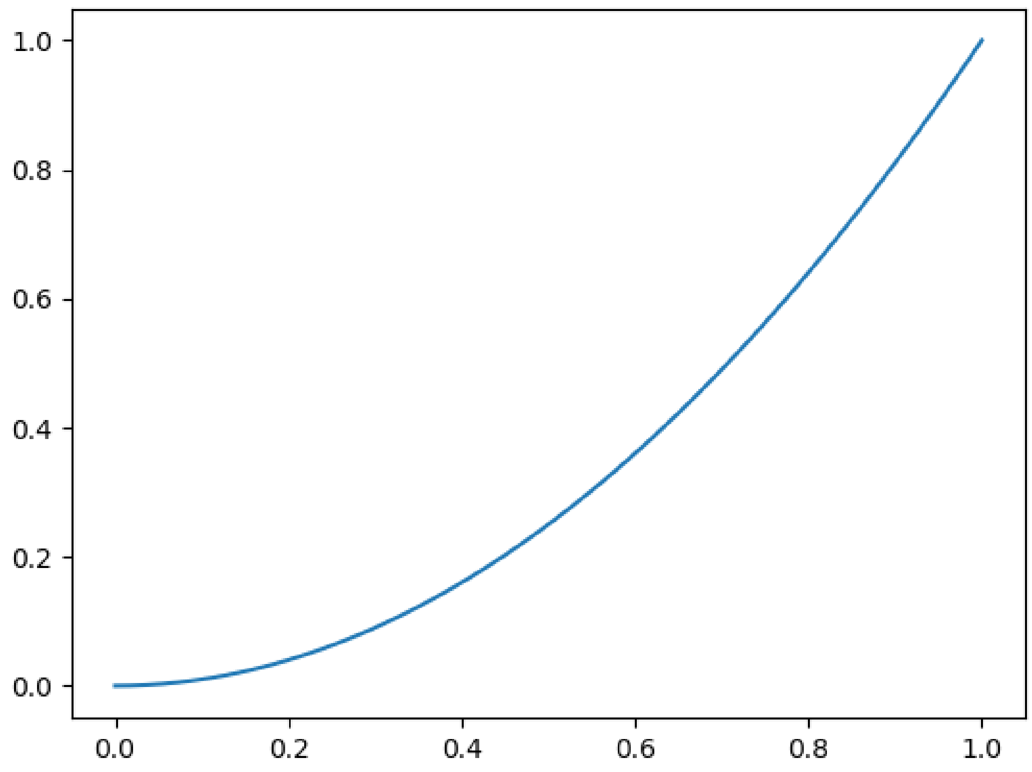
4.8 Plotting

Practically speaking, all plotting in Python is done using the package Matplotlib (packages will be discussed in section (4.9)). For now, it suffices to say that Matplotlib's tools are accessed using plt.<something>. So, if you see a line that uses plt. just note we're using something from Matplotlib. Note that this is essentially the same plotting backend as Matlab–if you can plot something in Matlab, you can plot it the same way in Python and vice-versa.

There are many different plotting techniques and styles you can use. For now, let's just start with the most basic plotting routines. Generally, when we begin a new figure, we begin with plt.figure() (strictly speaking you don't need this line, but it's good practice to include it). Inside the parentheses here, you can specify arguments of your plot, such as figure size. This line effectively creates a blank canvas for you to begin your figure. The next line to include, which is required, is plt.plot(). In general, the syntax for plt.plot() is

plt.plot(x,y)

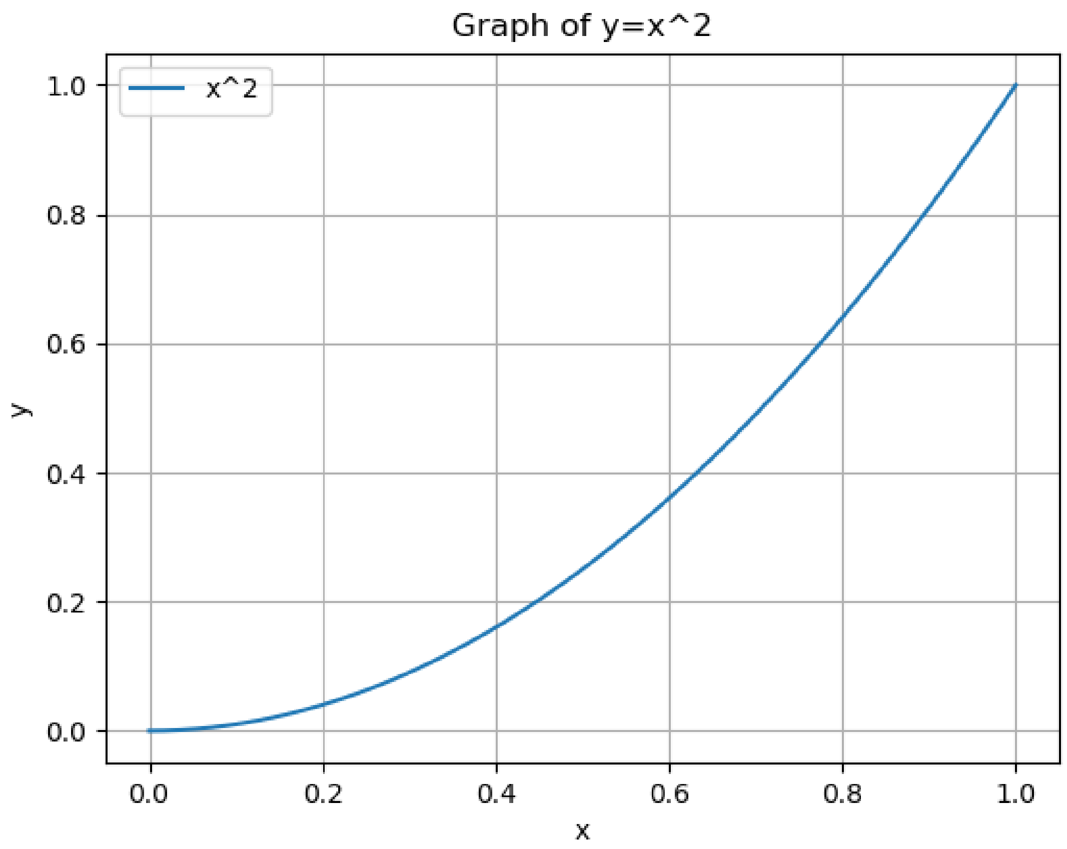
where x and y are lists or arrays that contain the data you want to plot. Strictly speaking, that's all you are *required* to do, but this leaves you with a horribly bare and ugly plot. Such a code will generate the following plot (after giving it some x and y data):



Does this look like a good plot to you? Your answer should be 'no.' Why? It's missing...almost everything else a plot should have! We need to, at minimum, include a title and axes labels. However, it many graphs it is also useful to include a grid (for readability), and a legend (in case you have multiple things going on in your plot). Thankfully, all of these things are extremely easy to add. It's best to show you this by example:

```
x = np.linspace(0,1,1000)
y = x**2
plt.figure()
plt.plot(x,y, label='x^2') #must have label here, this goes in the legend
plt.xlabel('x') #make basic label on x axis
plt.ylabel('y') #same for y axis
plt.title('Graph of y=x^2') #make title
plt.grid() #add gridlines for readability
plt.legend()#add legend
plt.savefig('plt.png')#save figure
#plt.show() displays figure instead of saving
```

which produces the following figure:



This looks *much* better. However, we have not even scratched the surface of the parameters you can tweak in these plots. You would be (and will be) amazed at the level of fine-tuning you can do–if you know the syntax! You'll notice that in the creation of this plot, I have used the function np.linspace(). This generates an array of linearly spaced numbers between some lower bound (0 in this case) and upper bound (1 in this case), with a certain number of elements between the two numbers (1000 in this case). This is a common way to graph functions. Note that the more elements we have in x, the smoother our graph will be.

This plotting tutorial is intentionally short. Part of the reason for this is that I want you to figure out more complicated plots on your own. The other part is that I will include a large example at the end of Chapter 4 that has a lot of fine plotting details, which you can adapt to suit your needs. The best place to figure out Syntax is the official matplotlib documentation:

https://matplotlib.org/3.1.1/users/index.html

4.9 Packages & Package Management

You've seen me reference a couple of things that are not originally in Python: NumPy and matplotlib. What exactly are these things? Simply put, they are extensions to Python. You'll commonly see such things referred to as either packages or libraries. One of the greatest aspects of Python is the sheer number of packages available. There's a Python package for nearly everything you can think of: machine learning, special functions (spherical harmonics, Bessel functions, etc.), optimization, linear algebra, and so much more. These packages often provide short cut commands to tasks you would have to otherwise hard-code yourself. One example of this is NumPy's transpose function, np.transpose(). This will easily transpose any matrix you give it–no need to code anything yourself here! The same is true for matrix multiplication, and many other mathematical operations/methods. The downside is that you don't always understand what's going on 'under the hood.' As such, the use of packages can cut down on instructional value, so sometimes professors will ask you to write something out explicitly even though there is a package for it. Regardless, packages are an amazing benefit, and should be utilized when possible/permitted.

How do you obtain these packages? It's actually incredibly simple. There are two ways to do it, depending on how you have installed Python. If you just installed Python directly from Python's website (and have installed Python 3 or newer), then it should have come with the package management tool 'pip.' Pip is a command-line function that automatically downloads and installs packages from the Python Package Index (a repository where pretty much everyone uploads their packages to). You must type all pip-related commands into a command line terminal. On Windows, you can just go into your search bar and type 'cmd' then hit enter. A terminal should open up. From there, you can simply type

```
pip install ⟨package name⟩
```

It really is that simple...if everything is configured correctly. If not, there severe anguish can result. Let's hope we avoid that. However, in this workshop, I'm hoping that all of you are using the Anaconda distribution of Python. This distribution comes with its own terminal, the Anaconda Prompt (on Windows, simply type this into your search bar and it should come up). The command to install packages is then conda install -c conda-forge ⟨package name⟩

Evidently, this method has a few more arguments to it. the -c conda-forge specifies the repository you're looking in. For some reason, Anaconda's packages aren't all located in one place (though most of them are). You can also install packages via the Anaconda Navigator by clicking around the GUI interface, but I feel cooler using the command prompt to install my packages.

Some of the most popular (and important) Python packages are NumPy, matplotlib, SciPy, random, and pandas. The first two we've seen already. SciPy is a library for scientific computing and has many excellent functions that we will work with eventually. Random is a great package for obtaining randomized data, either completely random or drawn according to some distribution. Finally, pandas is used to read data sets into Python. In general, you should consider these packages available to you at all times in this workshop; you can use them and their contents at you leisure unless you are specifically told not to.

A quick work about style. It is customary to include all of your imports at the top of you code. It is also common to abbreviate long imports with nicknames, which can be declared upon import of the package. For instance, it is common to write NumPy as np, matplotlib.plt as plt, and pandas as pd.

4.10 Reading & Writing Data Files

There are many ways to let Python read data sets. For all serious work, pandas is the go-to package. We are commonly interested in loading a .csv or .txt file into Python. The basic syntax for doing so is

pd.read_csv(⟨path/to/file⟩)

Pandas is usually smart about reading files; it can pick out whether you have a .txt file or .csv file and read it correctly. Sometimes, though, it struggles and needs extra specifications. A common argument is header=None which tells Pandas that your file has no header. This argument, and other arguments, are included after you specify the file path and file. We can deal with other issues related to Pandas as they arise.

Often times, we want to direct the output of our program into a text file, or .csv file. This can be done using Pandas, but it can also be done, in a more intuitive way, without library packages. The basic syntax for generating and writing to a file looks like:

```
file = open("<file name>", "w") # file can be any variable, w stands for 'write'
file.write(<something>)
file.close()
```

The file name must include the desired file extension, such as .txt or .csv. Note that you do not need indentation here. Formatting the content of your file can be tricky, but such as discussion should be on a case-by-base basis.

4.11 Random Number Generation

The random package is the go-to package for generating random numbers. When importing this package, it is useful to just use

from random import *

This imports everything in the random library, so you are free to use any commands associated with random(). There are many tricks we can do with this package, but it is sufficient, for now, to show you the following 3 examples.

First, to generate a random number between 0 and 1, we simply use

```
random()
```

This returns said random number. If we want a random integer, then we may write

```
randint(<lower bound>, <upper bound>)
```

which will generate a random integer between (and including) the bounds. Finally, we can draw numbers randomly from a Gaussian (normal) distribution with:

```
gauss(<mean>, <stnd deviation>)
```

This returns a random number following a Gaussian distribution with a given mean and standard deviation.

Simulating data often requires some amount of randomness, and this randomness can be created using this package. Alternatively, NumPy offers random number generation as well.

4.12 Examples

A lot of information has been thrown at you with minimal examples. In this section, I will state a couple problems and provide my commented code as a solution. While these examples will not be exhaustive, the exercises I will assign you will be. The only way for you to learn how to code, is to code. So, I want you all to have as much practice with the basics as possible–struggling through getting your code to work is part of the learning process. Still, I hope the following examples are helpful. I encourage you to try to work these examples out on your own. or at least think about how you would approach these, before looking at my solution.

Ex 1) Write a program that computes π using the Leibniz formula. Determine the error in approximation as a function of the number of iterations (terms in the sum) and computational time. Make plots that show 1) your approximation of pi as a function of the number of iterations, 2) error in approximation vs. number of iterations , and 3) error in approximation vs computational time. Decorate your plots appropriately.

Solution:

```
import numpy as np #import necessary modules
import time #keep track of computation time
import matplotlib.pyplot as plt

pi = np.pi #lazy declartation of pi

param = [] #store result of each iteration

for i in range(0,6): #number of iterations as power of 10
    start = time.perf_counter() #begin timer
    s = 0 #initialize variable to hold the sums
    for k in range(0,10**i): #state bounds of sum
        s += (-1)**k/(2*k+1) #compute pi
    s *= 4  #multiply result by 4, s= s*4
    end = time.perf_counter() #end timer

    tot = end-start #calculate total run time

    #append value of pi, time, and number of iterations
    param.append([s, tot, '{:.1E}'.format(10**i)])

#initialize lists to make plots
val = [] #value of pi
ctime = [] #computation time
niter = [] #number of iterations
err = [] #error of approximation

for i in param: #extract values from param so can plot them
    val.append(i[0])
    ctime.append(i[1])
    niter.append(i[2])
    errform = 100*np.abs(i[0]-pi)/pi #compute error
    err.append(errform)

#plot results
#If enclosed in $$, references LaTeX environment for typesetting
#I do not expect $$ to work for you
plt.figure(figsize=(14,10))
#marker details what kind of data point to use
plt.plot(niter, val, color = 'black', marker = 'o', linestyle='solid')
plt.title('Leibniz Approximation of $\pi$', fontsize=30) #decorate plot
plt.xlabel('Number of Iterations',fontsize=18)
plt.ylabel('Approximation of $\pi$', fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.grid()
plt.axhline(pi, linestyle='--', color='orange')
for i,v in enumerate(val): #add numbers next to data points
    plt.text(i,v+0.03, '{0:.6f}'.format(v), ha='center')
    #ha = horizontal alignment
plt.show()
#make the other plot
plt.figure(figsize=(14,10))
plt.plot(niter, err, color = 'black', marker = 'o', linestyle='solid')
plt.title('Leibniz Approximation of $\pi$ Error', fontsize = 30)
plt.xlabel('Number of Iterations',fontsize = 18)
plt.ylabel('Approximation of $\pi$ Error (%)', fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.grid()
for i,v in enumerate(err):
    plt.text(i+0.07,v+0.5, '{0:.3f}'.format(v), ha='center')
plt.show()

plt.figure(figsize=(14,10))
plt.plot(ctime, err, color = 'black', marker = 'o', linestyle='solid')
plt.title('Leibniz Approximation of $\pi$ Computational Time',fontsize=30)
plt.xlabel('Computation Time (s)', fontsize=18)
plt.ylabel('Approximation of $\pi$ Error',fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.grid()
#plt.axhline(pi, linestyle='--', color='orange')
for i, v in zip(ctime,err):
    plt.text(i+0.0001, v+0.5, '{0:.6f}'.format(v), ha='center')
plt.show()
```

Ex 2) Write a program that multiplies two matrices. Do not use NumPy's built in matrix multiplication function.
Solution:

```
import numpy as np

def mmult(a,b):
    if a.shape[1] != b.shape[0]: #check matrix dims
        print("Error: Incorrect Matrix Dimensions")
        raise ValueError #exit code if can't mult
    #initialize matrix to hold values of mult
    c = np.zeros(shape=(a.shape[0], b.shape[1]))
    d = 0 #hold dot product
    for i in range(0, a.shape[0]):
        for j in range(0, b.shape[1]):
            d += a[i,:]*b[:,j]#multiple ith row jth column
            c[i,j] = sum(d) #add results of d
            d = 0 #reset d and start process over
    print(c)
    return c

#make arbitrary matrices
a = [[1,2,9], [3,4,3], [5,5,5]]
b = [[1,2,1], [3,4,8], [2,2,2]]
#convert to arrays
a = np.array(a)
b = np.array(b)
#call function
mmult(a,b)
```

Ex 3) Write a program that uses the Babylonian method of calculating square roots and cube roots of any arbitrary user-defined number, given a reasonable initial guess for the root of the number.
Solution:

```
#The Babylonian method for aproximating square roots is
# x_(n+1) = (x_n + S/x_n)/2,
#where x_n gets closer and closer to the correct value and
#S is the square root in question.
#We must begin with some guess, x_0.

#Prompt user for inputs

S = float(input("Input the number you want to calculate the root of: "))
x_0=float(input("Input the initial guess for the root of that number: "))

def sqrt(S, x_0):
    calc = (S)**0.5 #Python's calculated value
    x=[x_0] #List for storing approximations
    for n in range(1, 100): #Arbitrary max number of iterations
        x.append(n) #Add placeholder element to list
        x[n] = (x[n-1] + S/x[n-1])/2 #Replace with correct value in list
        reldif = abs((x[n]-calc)*100/calc)#calculate absolute value relative diff
        if reldif < 0.0001: #Terminate upon desired precision, could use while loop
            break #break out of loop
    print("")#Readability in console
    #apply string formatting to make nice outputs
    print("The relative difference is {0} %".format(reldif))
    print("")
    print("The value of sqrt({0}) is (to 4 decimal places) {1:.4f}".format(S, x[-1]))

sqrt(S, x_0)

#A similar approach is taken for calculating cube roots.
#The modified formula we must use is
#x_(n+1) = (2*x_n + S/(x_n)^2)/3

def cuberoot(S, x_0):
    calc = (S)**(1/3) #same idea as above
    x=[x_0]
    for n in range(1, 100):
        x.append(n)
        x[n] = (2*x[n-1] + S/(x[n-1])**2)/3
        reldif = (x[n]-calc)*100/calc
        if reldif < 0.0001:
            break
    print("")#Readability
    print("The relative difference is {0} %".format(reldif))
    print("")
    print("The value of cbrt({0}) is (to 4 decimal places) {1:.4f}".format(S, x[-1]))

#cuberoot(S, x_0)

#uncomment function call to use desired function when program is called
```

Hopefully these examples give you some concrete ideas about strategies to use when solving problems and insight into Python's syntax. If you still feel shaky, don't worry! There will be many, many exercises you all will work through with problems similar to these.

Now that the basics are 'covered,' we can begin to move on to actual computational physics/programming!

Chapter5
Systems of Linear Equations

The fundamental problem we are solving in this Chapter is, given a system of linear equations, how do we solve for every unknown? Mathematically, we can write this question as

$$\mathbb{A}\mathbf{x} = \mathbf{b}$$

Where $\mathbb{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. There are many computer techniques to accomplish this, but we will only discuss four. Central to all of these techniques is Gaussian elimination. From there, we will discuss LU decomposition, matrix inverses, and finally NumPy's equation solving package. All of these techniques will make use of matrix manipulations.

5.1 Gaussian Elimination

Gaussian elimination is the first method we will use to solve a system of linear equations. In principle, this works for any system of equations–it doesn't have to be square. Suppose we are given a system of linear equations:

$$x + 3y + z = 9 \tag{5.1}$$

$$x + y - z = 1 \tag{5.2}$$

$$3x + 11y + 5 = 35 \tag{5.3}$$

We can rewrite this system in matrix form:

$$\begin{bmatrix} 1 & 3 & 1 \\ 1 & 1 & -1 \\ 3 & 11 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ 35 \end{bmatrix}. \tag{5.4}$$

Before we can perform Gaussian elimination, we need to make an augmented matrix using equation (5.4). This is done by simply 'sticking together' the square matrix and column matrix on the right hand side, and separating them with a thin line (this is just the notation):

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{array}\right]. \tag{5.5}$$

Now, we may begin to apply Gaussian elimination. We are allowed to do three operations, called row operations, on this augmented matrix:

- Swap two rows
- Multiply a row by a non-zero number
- Add one row to another

The goal of these three operations is to obtain an upper triangular matrix, or obtain as many zeros in the lower left ‘corner’ as possible. Once this is accomplished, the matrix is said to be in row echelon form. We can further reduce the matrix by making the diagonal elements (or the leading entry in each row) a 1, at which point the matrix is said to be in reduced row echelon form. The definition of row echelon form encompasses the following criteria:

- All rows with at least one non-zero element are above any rows whose elements are all zero
- The leading coefficient, or ‘pivot’ (first non-zero number from the left) of a non-zero row is always to the right of the leading coefficient of the row immediately above it
- All entries in a column below the leading coefficient are zero

Similarly, for reduced row echelon form, we must meet the following criteria:

- Matrix is in row echelon form
- The leading coefficient in each non-zero row is 1 (called a leading 1)
- Each column containing a leading 1 has zeros everywhere else

This might be a little confusing, but below are a couple examples of row echelon form and reduced row echelon form for non-square matrices:

$$\begin{bmatrix} 3 & 0 & 0 & 3 \\ 0 & 2 & 7 & 5 \\ 0 & 0 & 0 & 9 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 7 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

The matrix on the left is in row echelon form, while the matrix on the right is in reduced row echelon form. A matrix in reduced row echelon form is also in row echelon form.

By performing Gaussian elimination, we are effectively ‘simplifying’ the system of equations. The row operations we do are equivalent to algebraically legal manipulations on a system of equations; we’re just expressing it in a different form. So, looking back at our example matrix, equation (5.5), we can apply these row operations and see what we obtain:

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 5 & 35 \end{array}\right] \xrightarrow{3*L_2} \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 3 & 3 & -3 & 3 \\ 3 & 11 & 5 & 35 \end{array}\right] \xrightarrow[L_2*1/3]{L_3-L_2,\, L_3*(1/8)} \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 0 & 1 & 1 & 4 \end{array}\right] \tag{5.7}$$
$$\xrightarrow{L_2-L_1,\, l_2*1/2} \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -1 & -1 & -4 \\ 0 & 1 & 1 & 4 \end{array}\right] \xrightarrow[\text{REF}]{L_3+L_2,\, L_2*(-1)} \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow[\text{RREF}]{L_1-3*L_2} \left[\begin{array}{ccc|c} 1 & 0 & -2 & -3 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

REF and RREF stand for row echelon form and reduced row echelon form respectively. If this is the first time you're seeing this, you are probably a little confused, which is to be expected. How do I know what row operations I need to do? What's my goal? As stated earlier, the goal is to obtain a matrix in REF. Ideally, you should put the matrix in RREF. The exact way you do that is somewhat arbitrary, but some pathways lead to dead ends. In some sense, this is just trial and error–it isn't immediately obvious how we can tell a computer to do this for us. Before we discuss how to implement such an algorithm in Python, let's look at what we've actually done here. We have manipulated the original system of equations (equations (5.1)-(5.3)) into the form of a matrix in equation (5.7). Looking at the matrix in RREF, we can immediately read off a new system of equations:

$$x - 2z = -3 \tag{5.8}$$

$$y + z = 4 \tag{5.9}$$

Notice that we only have a system of two equations now that do not have explicit solutions–the best we can obtain is x, y and z defined in terms of each other. I hope, however, that you can see the utility of this method. If our matrix in RREF just has a single 1 in each row, we have immediately obtained our solution for x, y, and z! Regardless, we will almost certainly obtain a simpler system of equations to work with.

This is the exact method many computers use to solve systems of linear equations. The question we must answer now is 'How do we instruct Python to do this for us?' I will not make you write your own code for this; if you have completed the chapter 4 exercises then you should have all the tools you need to write such a program. Doing so, however, takes incredibly careful thought and is time-consuming. Instead, I will give you a code that does the computation for you and ask that you go through and understand each line. The following code should be able to solve most linear systems, provided the solution exists and the number of equations is equal to the number of unknowns (i.e. we have a square matrix):

```
#Courtesy of stackoverflow:
#shorturl.at/hiQ08 to website with Q+A
#Slightly improved for readability and special cases

def linsolv(A,b):
    n = len(A) #find dimensionality of matrix A
    M = A #reassign matrix for convenience
    #check if matrix is square or not
    for i in M:
        if n != len(i):
            print("Error: Only square matrices supported")
            raise ValueError

    print("Original matrix is {0}\n".format(M)) #see what we're working with
```

```
i = 0 #initialize counter for indices
for x in M: #loop through rows of M
    x.append(b[i]) #make augmented matrix
    i += 1 #cycle indices to augment correctly
print("Augmented matrix is {0}\n".format(M))

for k in range(n): #do pivoting
    print(k,n)
    for i in range(k,n):
        if abs(M[i][k]) > abs(M[k][k]):
            print("Pivoting...\n") #want largest pivot in upper left
            M[k], M[i] = M[i], M[k]#row swaps
            print("{0}\n".format(M))
        else:
            pass

    for j in range(k+1,n): #get into REF
         print('Puting \n{0}\ninto row echelon form...\n'.format(M))
         q = M[j][k] / M[k][k] #find what to mult row by
         for m in range(k, n+1):
             M[j][m] -= q*M[k][m] #subtract rows

print("Row echelon form is \n{0}\n".format(M))

print("Obtaining solutions...\n")
x = [0 for i in range(n)] #initialize array of zeros to hold solution

for i in range(n): #simple solution existence check
    #check if last row of REF has all zeros augmented with non-zero
    #simple solution existence check
    if all(i==0 for i in M[n-1][0:-1:1]) and M[n-1][-1] != 0:
        print("System has no solution")
        return "Null"

x[n-1] =float(M[n-1][n])/M[n-1][n-1] #get one solution
print("First solution is {0}/{1}\n".format(M[n-1][n],M[n-1][n-1]))
print("Performing back substitution...\n")

for i in range (n-1,-1,-1): #do back sub
    print("Obtaining {0}-th solution \n".format(i))
    z = 0
    for j in range(i+1, n): #substitute remaining solution(s) into
                            #systems of equations and solve
        print("Note that 0 + {1}*{2} = {3}\n"\
              .format(z, M[i][j], x[j], z + M[i][j]*x[j]))
            z += M[i][j]*x[j]
        x[i] = (M[i][n] - z)/M[i][i]
        print("Solution {0} is ({1} - {2})/{3}\n"\
              .format(i, M[i][n], z, M[i][i]))

    print(x) #print solution
    return x

A = [[1,2,3],[3,4,5], [4,5,6]] #sample input matrix, LHS of system
b = [7,8,9] #sample input vector, RHS of system

ans = linsolv(A,b) #Call function
```

This code might be intimidating at first, but if you spend some time with it I'm sure it'll make more sense. In my opinion, the most confusing parts about the code are the bounds on the for-loops. Python's zero-indexing doesn't help here either. Don't be afraid to use trial and error to get your bounds right!

With that, we've covered all we need to on Gaussian elimination and its relation to solving linear systems of equations. While this is indeed how many computers solve systems of equations, it is not always the most practical method, as we will discuss in the next section.

5.2 LU Decomposition

Many problems require solving equations of the form

$$\mathbb{A}\mathbf{x} = \mathbf{b} \tag{5.10}$$

for the same matrix $\mathbb{A}$, but *multiple* vectors $\mathbf{b}$. For such situations, it is inefficient to use Gaussian elimination over and over again. Instead, we break the matrix $\mathbb{A}$ into a lower triangular matrix and upper triangular matrix such that

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a31 & a_{32} & a33 \end{bmatrix} = \mathbb{L}\mathbb{U} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{5.11}$$

How exactly does this help us though? Substituting $\mathbb{L}\mathbb{U}$ into equation (5.10) for $\mathbb{A}$, we have

$$\mathbb{L}\mathbb{U}\mathbf{x} = \mathbf{b} \tag{5.12}$$

$$\mathbb{L}\mathbf{z} = \mathbf{b} \tag{5.13}$$

$$\mathbf{z} = \mathbb{U}\mathbf{x}. \tag{5.14}$$

So, the strategy is to first solve equation (5.13) for some new unknown set of vectors $\mathbf{z}$, then use $\mathbf{z}$ to solve for $\mathbf{x}$ in equation (5.14).

Now, how exactly do we find $\mathbb{L}$ and $\mathbb{U}$? It turns out that there are a couple simple formulas for the elements of each matrix:

$$u_{ij} = \mathbb{A}_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik} \tag{5.15}$$

$$l_{ij} = \frac{1}{u_{jj}} \left(\mathbb{A}_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik} \right) \tag{5.16}$$

There are alternative ways to find the decomposition, but I find that this is the most straightforward. It's important to note that the LU decomposition is not unique unless we require the diagonals of either $\mathbb{L}$ or $\mathbb{U}$ to be 1. Typically, we choose $\mathbb{L}$ to have 1's on its diagonal.

Once we have $\mathbb{L}$ and $\mathbb{U}$, what do we do with them? We simply follow the prescription outlined in equations (5.12) and (5.13). That is, we solve the new systems of equations. However, thanks to our decomposition, these systems are quick and easy to solve. First, we must solve equation (5.13) for **z** using a technique called forward substitution:

$$y_1 = b_1,\ y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j\ ,\ i \in [2, n] \tag{5.17}$$

where n is the size of **b**. We may then take this result, which gives us **z**, and solve for **x** in equation (5.14) using a technique called back substitution:

$$x_{-1} = y_{-1}/u_{-1,-1}\ ,\ x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{=ij+1}^{n} u_{ij} x_j \right)\ ,\ i \in [i, n-1] \tag{5.18}$$

where an index of -1 stands for the last element in some vector and n is again the size of the vector **z**.

We have all of the information we need to solve a system of equations with LU decomposition; we just need to code an algorithm that implements all of these techniques. I will give you a code that implements LU decomposition and back substitution. It will be an exercise for you to write an algorithm that implements forward substitution. The code that does the LU decomposition is given below:

```
#LU decomposition algorithm courtesy of shorturl.at/knoQ4

def LU(A):
    M=A

    L = [[0.0]*len(A) for i in range(len(A))]
    U = [[0.0]*len(A) for i in range(len(A))]
    for j in range(len(M)):
        L[j][j] = 1
        for i in range(j+1):
            s = sum(U[k][j] * L[i][k] for k in range(i))
            U[i][j] = A[i][j] - s
        for i in range(j,len(A)):
            s2 = sum(U[k][j]* L[i][k] for k in range(j))
            L[i][j] = (A[i][j] - s2)/U[j][j]

    return L,U
```

The code that does back substitution is given below:

```
def bsub(U,z):
    U = A
    z = z
    x = [[0.0]*len(U) for i in range(len(L))]
    x[-1] = z[-1]/U[-1][-1]
    for i in range(len(A)-2, -1, -1):
        x[i] = 1/U[i][i] * (z[i] - sum([U[i][j]*x[j] for j in range(i+1,len(A))]))
    print(x)
    return x
```

Neither of these codes are particularly difficult to follow, provided you're familiar with Python's syntax. The most difficult parts are getting the index bounds right and computing the sums.

There are a couple of further applications of LU decomposition that should be discussed. First, we can use LU decomposition to compute the determinate of a matrix. To see how, we need to take a step back and consider something called a permutation matrix $\mathbb{P}$, which is equal to the identity matrix. Strictly speaking, I should have included this matrix in the decomposition (such a thing is know as an LUP decomposition and ensure numerical stability). Just as we are allowed to interchange rows in Gaussian elimination, we may do the same in LU decomposition to avoid any division by zero. When doing these row changes, we will also interchange the rows of the permutation matrix. Symbolically, we may represent this as

$$\mathbb{P}\mathbb{A} = \mathbb{L}\mathbb{U} \tag{5.19}$$

$$\implies \mathbb{A} = \mathbb{P}^{-1}\mathbb{L}\mathbb{U} \tag{5.20}$$

$$\implies \det(\mathbb{A}) = \det(\mathbb{P}^{-1})\det(\mathbb{L})\det(\mathbb{U}) = (-1)^S \prod_{i=1}^{n} l_{ii} \prod_{i=1}^{n} u_{ii}. \tag{5.21}$$

Here, S is the number of row swaps used in the decomposition algorithm, and the multiplication follows because the determinates of triangular matrices are just the product of their diagonal elements (can you show this?).

Next, we can use LU decomposition to calculate matrix inverses. Before we do this, though, note that one way to compute inverses is to augment the matrix with the identity matrix. Then, perform Gaussian elimination until the matrix on the left hand side is the identity matrix. In other words, augment your matrix with the identity matrix and put it in reduced row echelon form. Such a process (intermediate steps omitted) might look like this:

$$\left[\begin{array}{ccc|ccc} 2 & 4 & 6 & 1 & 0 & 0 \\ 0 & -1 & -8 & 0 & 1 & 0 \\ 0 & 0 & 96 & 0 & 0 & 1 \end{array}\right] \rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1/2 & 2 & 13/96 \\ 0 & 1 & 0 & 0 & -1 & -1/12 \\ 0 & 0 & 1 & 0 & 0 & 1/96 \end{array}\right] \tag{5.22}$$

Now, since we can obviously write the identity matrix as

$$\mathbb{A}\mathbb{A}^{-1} = \mathbb{I} = [e_1, e_2, \ldots, e_n] \tag{5.23}$$

where e_n's are column matrices of the identity matrix, it is clear that we have an equation of the form

$$\mathbb{A}\mathbf{x_i} = \mathbf{b_i} \tag{5.24}$$

if we let $\mathbf{b_i}$ be a column vector of the identity matrix and $\mathbf{x_i} = [x_1, x_2, \ldots x_n]$ be an analogous column vector for each column of $\mathbb{A}^{-1}$. We may then break apart $\mathbb{A}$ into its LU decomposition and solve each system of equations to obtain each $\mathbf{x_i}$ using the methods discussed above. Once we have all elements of $\mathbf{x_i}$, we will have found the inverse matrix.

You might be wondering which method is more efficient. After all, finding inverses using LU decomposition seems like much more work. To an extent, it is; however, both methods can be useful depending on your situation. For sufficiently small matrices, say less than 4×4, it is more computationally efficient to use LU decomposition. Further, algorithms to obtain reduced row echelon form are often numerically unstable–they don't work that well. In principle, though, it can be done.

In practice, though, matrix inverses are seldom needed. Inverse matrices just give us yet another method of solving linear systems of equations; however, it can take significant computational effort to calculate the inverse. When other, more efficient, methods exist, it simply isn't productive to fiddle with inverses.

5.3 Library Packages

This chapter has been focusing on solving systems of linear equations. I have shown you how to do Gaussian elimination and LU decomposition, and how to use those methods to solve systems of linear equations. Everything we have done (except perhaps Gaussian elimination), can be readily done using library packages. Indeed, all of those long lines of code can essentially be replaced by a single line. So, why have I bothered to go through all of this effort when simpler (and more practical) options are available? The answer is simple: I wanted to show you how these functions worked. They are not black magic nor voodoo–they have logical methods behind them. Further, this is also good coding practice. In some tasks, you will find that there is no explicit library function that does what you want. In those cases, it is critical that you know how to put together a program (that often relies on existing library packages) that suits your specific purpose. Still, library packages are incredibly useful (and in more complex codes, essential), so we shall discuss them now.

5.3.1 NumPy.linalg.solve()

As does the above code, this package only works for square matrices. The basic syntax is

$$\text{np.linalgsolve(A,b)}$$

where A is some square matrix and b is some vector. The output is a numpy array that contains the solutions $x_1, x_2, \ldots, x_n$. That's all there is to it...See, if I would have just tossed this at you, I would run out of material to talk about!

5.3.2 scipy.linalg.lu()

This library function gives the LUP decomposition of a matrix $\mathbb{A} = \mathbb{PLU}$. The syntax is

$$\text{scipy.linalg.lu(A, permute\_l = False, overwrite\_a=False, check\_finite=True)}$$

where A is the matrix you want to decompose. There are some optional parameters you can specify. Permute_l toggles whether or not the algorithm swaps rows. Check_finite runs through your matrix and determines if there are any infinities or NaN's that may cause problems in the computation. I'm not sure what overwrite does. Regardless, this will return the permutation matrix $\mathbb{P}$ and the lower and upper decompositions *in that order.* It is critical to note that last point about the order in which scipy returns the matrices. If you want to assign the outputs of this scipy function to variables, you *must* write your command like this:

$$a, b, c = \text{scipy.linalg.solve(A)}$$

Of course, a, b, and c can be whatever name you want, but no matter what the name is, a will be your permutation matrix, b will be your lower decomposition matrix, and c will be your upper decomposition matrix. Do not get these mixed up!

There isn't an explicit library function that explicitly does Gaussian elimination for us (at least, that I could find). This just illustrate the importance of knowing how to code and not needing to rely on library functions!

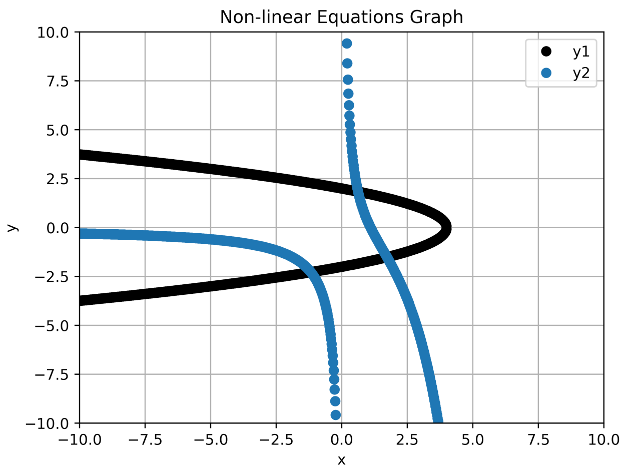
Chapter6
Solving Non-Linear Equations

6.1 Graphical Methods with fsolve()

We've seen how to solve linear systems of equations, but how do we solve non-linear systems, or just equations that aren't linear? Suppose our system of equations is

$$x + y^2 = 4 \tag{6.1}$$
$$e^x + xy = 3 \tag{6.2}$$

Go ahead and try putting this into Wolfram Mathematica or any other online calculator, such as SymboLab (if you don't know what either of these are, don't worry about it). Go ahead, I'll wait. You'll probably find that these two websites *cannot* solve the above system of equations–the computation time is just too great. On one had, I'm a little surprised Wolfram can't do this. On the other, it illustrates the importance of knowing numerical methods and coding! So, how might we go about solving this? When faced with any such system of non-linear equations, the first thing we should *always* do is graph it (if possible). Graphing the above system in Python is fairly easy, since we can explicitly solve for y in both equations. If we are presented with a case where we can not explicitly solve for y (e.g. $x^2 \sin(x) = ye^y$), we may use SymPy, which offers implicit equation plotting. However, we'll only cross the SymPy bridge if we are out of other options, and for now that is not the case. Plotting equations (6.1) and (6.2) in Python (you should do this yourself too, as an exercise) gives us the following plot:



This graph immediately tells us very important information. From this plot, we know that this system of equations has 3 real solutions, and we know *approximate* values for these solutions (just estimate from the graph). In general, though, we would like to be a little more precise, so how do we hone in on these solutions? For this particular problem, we can use two methods: Newton's method, and scipy's fsolve() function. For this particular system, I will only discuss the latter. It is possible to use Newton's Method to solve this, but doing so involves Jacobian matrices and some other tedious aspects I'd prefer not to go in to. Don't worry, I'm not abandoning hard-coding algorithms! I will use them explicitly in this Chapter, but for simpler problems.

The package scipy.optimize.fsolve() is fairly straightforward to use. The syntax is

scipy.optimize.fsolve(f, x0)

where f is a **Python function** with at least one argument, and $x0$ is (a list) of initial guesses for the solution. First, we must make a function that returns both equations (6.1) and (6.2). We also require the argument to only take one vector argument, else SciPy complains about required positional arguments. Once that is done, we may call fsolve with a list of two initial guesses, one for x and one for y. Doing this might look like the following:

```
import scipy

def eqn(p):
    x,y = p
    return (x + y**2-4, np.exp(x) + x*y - 3)

x,y = scipy.optimize.fsolve(eqn, [1,1])
print(x,y)

Output: 0.6203445234801195 1.8383839306750887

x,y = scipy.optimize.fsolve(eqn, [-1,-1])
print(x,y)

Output: -1.1831793154500128 -2.276659683714106
```

I have included the outputs of each print statement in the above block of code fo convenience. There are two important takeaways here. First, note that we must rewrite our functions such that they are equal to zero. Second, the initial guesses we give fsolve() are critical in determining what solution we obtain. Notice that we only get one pair of solutions per initial guess! By now, you should hopefully see the necessity in graphing our systems. If we hadn't, we might assume the function only has one solution, or we might not realize this function has 3 solutions (I didn't bother finding the third one though).

This illustrates a powerful tool at our disposal, scipy's fsolve() function. It is an excellent tool to check our answers with when we explicitly write root-finding algorithms in the next couple sections. There are two methods I will discuss algorithms for: Newton's Method, and the bisection method.

6.2 Newton's Method

Consider an arbitrary function $f(x)$. Expanding it in a Taylor series around $x = x_0$ gives us

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \mathcal{O}(x^2) \tag{6.3}$$

Suppose we have a root of equation (6.3), so $f(x) = 0$. This value is taken as the next approximation of the root we are trying to find. So, setting $f(x) = 0$ and $x \to x_{n+1}$ and solving for x_{n+1}, we obtain

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{6.4}$$

Equation (6.4) is known as Newton's Method, and is incredibly versatile and powerful, provided we give it a good initial guess, x_0. If we keep iterating using this formula, we should eventually arrive at *a* root of the function $f(x)$. If a function has multiple roots, just as we've seen above, we'll need to give the algorithm a close initial guess in order to converge on that solution. In theory, we can iterate forever. However, at some point, our solution, x_{n+1}, will stop changing significantly. At that point, we can end our algorithm and say we have found a root. Since this algorithm is rather straight forward, I will have you code it as an exercise.

6.3 Bisection Method

Like Newton's Method, the goal of the Bisection Method is to determine the roots of some function $f(x)$. Unlike Newton's method, this method is slower to converge on a solution. Indeed, this method is more commonly used to obtain sufficient initial guesses, which are then used in other algorithms to hone in on the exact root.

In any case, the Bisection Method is based on the intermediate value theorem. Suppose we have a continuous function $f(x)$ defined on some interval $[a, b]$. If the function changes signs (the function is positive, then becomes negative, or vice-versa) on $[a, b]$, then there is a root somewhere in this interval. We then compute the midpoint m of the interval. Now, we have two possible intervals to choose from: $[a, m]$ or $[m, b]$. One of these intervals is guaranteed to have a sign change (unless m happens to be a root, which is very unlikely). We choose whichever interval has a sign change, then begin the process all over again.

This algorithm isn't particularly difficult to code either, but I'll give you a code that does it. You may want to model your code for Newton's Method after this one, since it includes a solution tolerance range and max iteration cut off. While features like this aren't strictly necessary, they make your code much more usable. Anyways, here's the code for the Bisection Method:

```
def bisect(f, I):
    '''
    Bisect Method for root finding.
    f is a function, I is an interval with a sign flip
    '''

    I = I
    y = f
    n_max = 100 #max number of iterations
    tol = 1e-16 #tolerance to accept a root
    n = 0 #count iterations
    while n <= n_max:
        n+=1
        m = (I[0] + I[1])/2
        #print(m)
        if abs(y(m)) < tol:
            print("Root found, terminating algorithm at {0} steps"\
                  .format(n))
            print("The root is {0}".format(m))
            return (m)
        if (y(m) > 0 and y(I[0]) > 0) or (y(m)<0 and y(I[0]) <0):
            I[0] = m
        else:
            I[1] = m
        #print("New interval is {0}".format(I))
        if n == n_max:
            print("Max iterations reached, terminating algorithm at {0} steps"\
                  .format(n))
            return m

#test inputs
f = lambda x: x**4+5*x**3+9*x+3
I = [-1,0]

root = bisect(f,I)
print(root)

Output:
Root found, terminating algorithm at 54 steps
The root is -0.31679024794235283
-0.31679024794235283
```

The three quotes at the top of the function create what's called a 'doc string', which is typically information about a function and the arguments it takes. They aren't necessary, but can be helpful.

In practice, fsolve() is usually sufficient to find roots–you don't need to code your own algorithm. Again, though, it's good to see how these packages work. You will get more practice with each of these methods in the corresponding exercise worksheet.

Chapter7 Curve Fitting

One of the most important aspects of data analysis is fitting a model to our data. These models are meant to describe general trends/parameters in our data and can be used to make predictions about the system we are analyzing. For instance, we could have data about the apparent visual magnitude of a star as a function of local light pollution. If we plot this data, we'll probably see a trend. Using that data, we may want to estimate the apparent magnitude of a star elsewhere on Earth given some amount of light pollution, or estimate the apparent magnitude of the star outside of our data range. To answer these questions, we need to fit a model to our data. In general, we can fit any function we want to our data, but some make more sense than others (we wouldn't fit an exponential curve to data that appears logarithmic!). Caution should be taken, though, especially when extrapolating data outside of our given data points–what may look like a clear trend may only be so in a certain limit. In other words, physical reality may substantially deviate from your model if you don't have enough reliable data points!

In this Chapter, we will look at simple linear least squares regression, along with a library function from SciPy that fits virtually any function to any data set. In practice, people seldom need to code their own program to fit their data–they just use library packages. However, as always, it is instructive to know some theory behind how these packages work. With that said, let's begin looking at the methodology behind linear least squares fitting.

7.1 Linear Least Squares

Linear least squares, otherwise referred to as simple linear regression or ordinary least squares regression, is one of the more common algorithms for determining how one variable changes with respect to a different variable. As the name implies, this technique only works for data which appear to be linear.

Of course, given some apparently linear set of data, there are many (infinitely many) linear lines we can draw through that data, some better than others. How exactly do we determine what line is the'best' fit? In technical terms, we want to 'minimize the residuals' of our fit. Simply put, a residual is the difference between our model and our data point. If there is no residual, then our model goes directly though our data point. Larger residuals indicate that the data point is far from our model and may not be a good fit (or that particular data point is an outlier). It is exceedingly unlikely for any model to fit *perfectly* to a set of data–there's always experimental uncertainties that introduce some scatter in our data. As such, if your model perfectly fits your data, you should worry about the validity of your model, since it might fail to describe other similar situations.

There is a simple formulation for linear least squares fitting. Suppose we have a scatter plot of n (x, y) data points. We want to find the best fit line, denoted $\hat{y} = mx + b$, that reduces the sum of the squared errors/residuals in y. In other words, we want to minimize

$$F \equiv \sum (y_i - \hat{y}_i)^2 = \sum (y_i - b - mx_i)^2 . \tag{7.1}$$

From standard multivariable calculus, we know that F will be minimized at m and b when the *partial* derivatives $\frac{\partial F}{\partial m} = 0$ and $\frac{\partial F}{\partial b} = 0$, respectively. Computing these partial derivatives yields

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{n} -2 (y_i - b - mx_i) = 2 \left(na + b \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i \right) = 0. \tag{7.2}$$

We can divide equation (7.2) by 2 and solve for b to obtain

$$b = \frac{1}{n} \sum_{i=1}^{n} y_i - m \frac{1}{n} \sum_{i=1}^{n} x_i = \bar{y} - m\bar{x} \tag{7.3}$$

where the bar above a quantity denotes the average of that quantity. The next partial derivative tells us

$$\frac{\partial F}{\partial m} = \sum_{i=1}^{n} -2 \left(y_i x_i - b x_i - b x_i^2 \right) = 0 \tag{7.4}$$

Substituting in what we found for b and dividing by -2 gives us

$$\frac{\partial F}{\partial m} = \sum_{i=1}^{n} \left(x_y y_i - x_i \bar{y} + m x_i \bar{x} - b x_i^2 \right) = 0. \tag{7.5}$$

Separating equation (7.5) into two sums and solving for m yields

$$m = \frac{\sum_{i=1}^{n} (x_i y_i - x_i \bar{y})}{\sum_{i=1}^{n} (x_i^2 - x_i \bar{x})} = \frac{\sum_{i=1}^{n} (x_i y_i) - n\bar{x}\bar{y}}{\sum_{i=1}^{n} (x_i)^2 - n\bar{x}^2}. \tag{7.6}$$

It is possible to write m in several different ways by manipulating the sums. The most useful manipulation tells us that

$$m = \frac{\text{cov}(x, y)}{\text{var}(x)} \tag{7.7}$$

where 'cov' stands for the covariance and 'var' stands for the variance.

Equations (7.3) and (7.6) (or (7.7)) must then be calculated in order to fit a line to your data using linear least squares. Computing all of the sums may be tedious, but otherwise this isn't a particularly difficult algorithm to code in Python. I will leave the implementation of this as an exercise.

Once we have our fit, how do we determine how do we determine the quality of the fit? One method of doing this is by using an R^2 value. Loosely speaking, this is a measure of how well our fitted model can predict data outside of our given data range, or how well the variability in our data is explained by our model. Generally speaking, R^2 can range from 0 to 1, with 1 being a 'perfect' fit. We can compute R^2 with the following formula:

$$R^2 = 1 - \frac{\sum_i^n (f_i - y_i)^2}{\sum_i^n (\bar{y} - y_i)^2} \tag{7.8}$$

where y_i are your actual data points and f_i is the corresponding model value.

R^2 is a good metric to judge the quality of various models in relation to each other. There are other metrics we can compute, such as reduced χ^2, that tell us how well our data fit a distribution, which is more suited for testing the quality of the model itself. We don't need to concern ourselves with the minor details, though.

7.2 Scipy's curve_fit()

Our data, of course, can behave non-linearly. For these cases, it's still possible to code your own algorithm to fit the data, but less practical. For most general purposes, we may use SciPy's curve fitting package:

scipy.optimize.fsolve(f, xdata, ydata, p0, sigma)

Here, f is the function you want to fit to your data. As before, it must be defined using Python functions. The second and third arguments are self explanatory. p0 is a list containing initial guesses for your fit parameters (e.g. amplitude, phase, etc.). Sigma is an optional argument that allows you to tell scipy there is uncertainty in your y-data. If specified, that uncertainty will be taken into account during the fit. This is of great use in realistic cases, but for our purposes we do not need to concern ourselves with this additional argument.

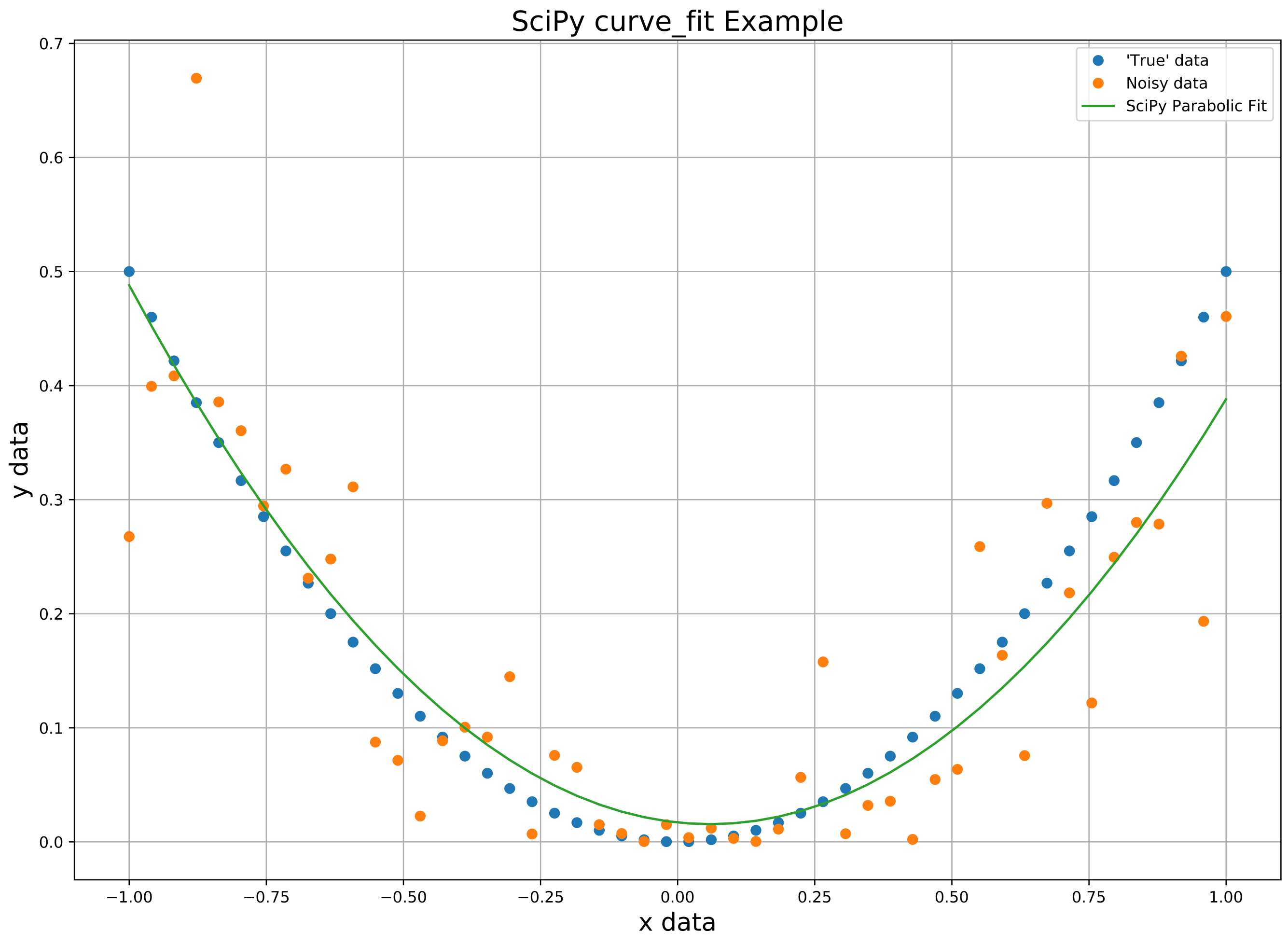
This function returns two arrays: popt and pcov **in that order**. The former gives the optimal parameters of your fit (e.g. slope and y-intercept) in order they were specified in your function. The latter is a covariance matrix, whose **diagonal** elements are equal to the *variance* of your parameters, in the order they were specified in the function you defined. In other words, the square root of these diagonal elements correspond to the uncertainty in SciPy's fit for each free parameter in your model. It is absolutely critical that you **specify the independent variable (usually x) as the first argument of your function**, else SciPy's fit will not work.

Let's see an example of how this function works. Below is a script that will generate a simple parabola, add noise to it (to make our data more realistic), and fit the data:

```
import numpy as np #get relevant imports
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import random

data = np.linspace(-1, 1, 50) #generate x data

#random.seed(1) #random library way of making noise
#noise= [random.gauss(0,0.25) for i in range(data.size)]


np.random.seed(3) #to generate the 'same' random data

noise = np.random.normal(0,0.15, data.shape) #NumPy's way

new_data = data+noise #add noise to data

old_curve = 0.5*data**2 #make curve without noise

curve = 0.5*new_data**2 #make curve with noise

f = lambda x, a, b, c: a*x**2 + b*x+c #define general parabola

popt, cov = curve_fit(f, data, curve) #calculate fit

#obtain values and uncertainties of each parameter
a_f, a_f_unc = round(popt[0], 3), round(np.sqrt(cov[0][0]), 3)
b_f, b_f_unc = round(popt[1], 3), round(np.sqrt(cov[1][1]), 3)
c_f, c_f_unc = round(popt[2], 3), round(np.sqrt(cov[2][2]), 3)

sciline = a_f * data**2 + b_f*data + c_f #make fit line

#make plot
plt.figure(figsize = (14,10))
plt.plot(data, old_curve, 'o', label = "'True' data")
plt.plot(data, curve, 'o', label = 'Noisy data')
plt.plot(data, sciline, label = 'SciPy Parabolic Fit')
plt.xlabel('x data', fontsize=16)
plt.ylabel('y data', fontsize=16)
plt.title('SciPy curve_fit Example', fontsize=18)
plt.grid()
plt.legend()
#add caption to figure to display fit values and uncertainties
cap = "Fit parameters: a = {0} $\pm$ {1}, b = {2} $\pm$ {3}, c = {4} $\pm$ {5}"\
.format(a_f,a_f_unc, b_f,b_f_unc, c_f,c_f_unc)
```

```
plt.text(-1, -0.12, cap, fontsize = 16)
plt.savefig('Curve_fit_ex.png', bbox_inches = 'tight', dpi = 1000)
#plt.show()
```

The output of this code is the following plot:



Fit parameters are a = 0.421 ± 0.037, b = -0.05 ± 0.02, c = 0.017 ± 0.017

Notice that the fit values for a and b are actually not the correct value within their $1 - \sigma$ uncertainties even for this relatively simple data set with minimum scatter. So, you should always interpret the results of your fit carefully. In other words, don't assume that your fit perfectly matches reality within uncertainties! You may also notice that I didn't specify any initial guesses for these parameters. As you may verify, adding in the exact value of each parameter as the initial guess actually does not change the fit parameters nor uncertainties.

This example will be enough to get you started. Keep in mind that when you define your fitting function, you should choose the most general function possible. For example, when fitting a parabola, assume a form of $ax^2 + bx + c$ and not just ax^2. Similarly, for a sine wave, you should assume a form of $A \sin(bx - c)$ with a, b, c scaling constants.

Chapter8 Interpolation

In the previous chapter, we covered curve fitting. To determine data outside of our collected data range, we can just assume our best fit line also models that region of data. For instance, the example using SciPy i the previous chapter plotted the parabola in the range $[-1, 1]$, but it's reasonable to assume the parabolic trend continues indefinitely. To obtain a reasonable estimate for what our data might be outside $[-1, 1]$, we can just use the same best fit parameters and trend line. In reality, this may not always be the case–trends can suddenly deviate from expected patterns (e.g. Planck's law for black body radiation). But, what if we have holes *within* our data set that we want to fill? We could just fit our data as in the previous chapter and use the best fit line, but suppose we can't find a good fit function. Then, we're forced to use a technique called interpolation. This technique allows us to construct new data points given a set of known data points. There are many types of interpolation, which are based on the 'curve' we want to interpolate on. For instance, if we expect the data between two data points to be linear, we would use linear interpolation. If we expect the data between to points to follow a polynomial curve, we may use polynomial interpolation.

As in the previous chapter, I will discuss the simple case of linear interpolation. For more complicated interpolations, we will use SciPy's interpolation function.

8.1 Linear Interpolation

As mentioned above, linear interpolation takes two known data points and generates a new data point between them, assuming that point must lie on a line between the two data points. What does this look like mathematically? Suppose we have two known points, (x_0, y_0) and (x_1, y_1). If we want to find a point between these two points that lies on a line, called (x, y), we may use the equation for slopes:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \tag{8.1}$$

We may easily solve equation (8.1) for y to obtain the formula we use for linear interpolation:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0} \tag{8.2}$$

So, if we plug in the x-coordinate of the point we want to interpolate into equation (8.2), we know where it should lie in our data set. This isn't a particularly difficult formula to implement in Python, so I will leave it as an exercise for you. Note that we can also use this method to approximate functions themselves. If we have a complicated function, we can use interpolation on the known data points to figure out how the function could behave inside the range of the given data points. An example of this will be given in the next section.

Another use of linear interpolation is to 'thin out' a data set while retaining its general trend. Sometimes, we can have data sets with millions of points, and doing manipulations on that data set can take a long time. In some situations, we can obtain equivalent results by interpolating our data at specific (large) intervals. For example, suppose we have a data set with 2 million data points that plots a decaying curve over 400 seconds. We could interpolate our data at, say, every 10 seconds. Doing so will 'thin out' our data set and make it more efficient to work with, while preserving the overall trend of the decay. In situations like this, we may also apply equation (8.2), while just making sure we interpolate at the correct time intervals. This is also not particularly difficult to code in Python, so I will leave this as an exercise for you.

8.2 SciPy's Interpolate

For more complicated interpolations (e.g. polynomial), we may use SciPy's interpolation function. There are many types of interpolation available through this function. One 'class' is univariate interpolation, which is interpolation with familiar 1-dimensional (x, y) data. The other class is multivariate interpolation, which is interpolation for multidimensional data (e.g. a function of $f(x, y)$). For our purposes, it is sufficient to work with the 1 dimensional univeriate interpolation. The function we then use is

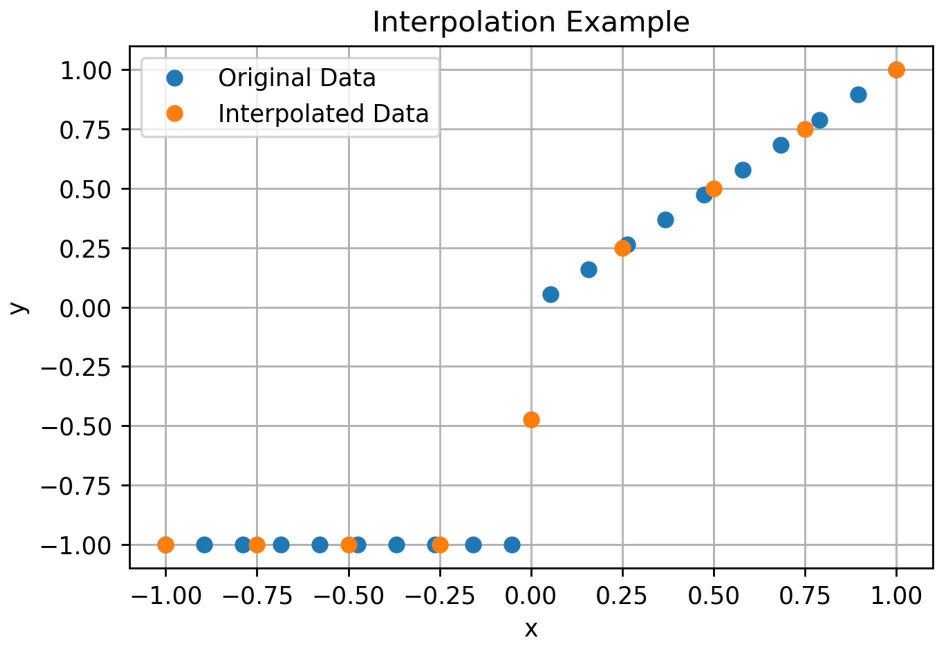
scipy.interpolate.interp1d(x,y,kind='linear')

Here, x and y are simply your x and y data arrays. The 'kind' argument specifies what interpolation SciPy does. By default, it will do linear interpolation, but you may also choose to do quadratic, cubic, or other kinds of interpolation. This function returns another function that approximates the function given in the (x, y) data set. Let's look at a simple example where we use interpolation to approximate a piecewise function. The code below makes a basic piecewise function, then uses interp1d to obtain new data points inside the range of the piecewise function:

```
import numpy as np #get necessary imports
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

x= np.linspace(-1,1,20) #make original x data

#make piecewise function
#see np.piecewise documentation for more details on this
y = np.piecewise(x, [x > 0, x <= 0], [lambda x: x, -1])

f = interp1d(x,y, kind='linear') #do interpolation
```

```
xnew = np.linspace(-1,1,9) #make new x values
g = f(xnew) #put them in interpolation function

#make nice plot
plt.plot(figsize=(14,10))
plt.plot(x,y, 'o', label = 'Original Data')
plt.plot(xnew, g, 'o', label = 'Interpolated Data')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Interpolation Example")
plt.grid()
plt.legend()
plt.savefig('Interpolation_ex.png', dpi = 250)
```

This results in the following graph:



Note that this interpolation isn't quite correct given how the piecewise function is defined. The orange data point at $x = 0.0$ should have a value of -1.0, but clearly it doesn't. We could have avoided having this particular point showing up in our data if we chose to have an even number of points in our *xnew* array. Regardless, the nature of linear interpolation will always attempt to model this discontinuity as a line despite the fact there shouldn't be any data in that discontinuity. This is another reminder that we must be careful about how we apply library functions to our problems! While there is much more to linear interpolation, we do not have the time to cover it. For our purposes, though, this is a good enough look at the basics.

Chapter9 Numerical Integration

Numerical integration is an important concept in computational physics. Many functions do not have elementary antiderivatives, and many others are simply unintegrable analytically. For cases like these, we may resort to numerical methods to approximate the area under the function for some interval $[a, b]$. For many purposes, particularly in simulation work, this is all we need. There are various methods of numerical integration. We will discuss 4 of these methods: Riemann sums, trapezoid rule, Simpson's rule, and SciPy's quad integration package.

9.1 Riemann Sums

When you were first learning about integration, you probably saw the notion of Riemann sums. In effect, we use n rectangles to approximate the area under a curve. Of course, there are three different ways we can draw our rectangles under the curve. We can have either the left or right edges of the rectangle 'touching' the curve, or we can have the midpoint of the rectangle touch the curve. These are termed the left, right, and midpoint Riemann sums, respectively. Their mathematical formulations are as follows:

$$A = w * \sum_{i=0}^{n-1} f(x_i) \quad \text{(left)} \tag{9.1}$$

$$A = w * \sum_{i=1}^{n} f(x_i) \quad \text{(right)} \tag{9.2}$$

$$A = w * \sum_{i=0}^{n} f(x_{i+1/2}) \quad \text{(midpoint)} \tag{9.3}$$

where n is the number of rectangles and w is the width of the rectangle. If the lower and upper bounds are a and b respectively, then the width of the rectangle is given by $w = (b - a)/n$.

These formulas aren't very difficult to code, but the process gets repetitive. So, you will be given code that computes the left and right Riemann sums, and an exercise that asks you to compute midpoint sums. The code for the left Riemann sum is given below:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as pat
```

```
def left_Riemann(f, a, b, n):
    '''
    Computes left Riemann sum
    f = function to integrate, must define actual function
    a = lower bound on integration
    b = upper bound on integration
    n = number of rectangles to use
    '''
    w = (b-a)/n #calculate width of rectangles
    x = np.linspace(a,b, n+1)#obtain x values for rectangles
    xf = np.linspace(a,b, (b-a)*100) #obtain x values for function
    #Think about why we need n+1 and not n
    c = 0 #initialize sum
    plt.figure() #begin figure
    plt.plot(xf, f(xf), label = 'f(x)') #plot curve
    ax = plt.gca() #get current axes, need to put in rectangles

    for i in range(n): #do sum
        #calculate rectangles, see matplotlib.patches.Rectangle for syntax
        rect = pat.Rectangle((x[i], 0), w, f(x[i]), fill=False)
        ax.add_patch(rect) #add rectangls to plot
        c += f(x[i]) #compute sum
    c *= w #compute sum
    plt.title("Left Riemann Sum")#add decorations to plot
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.show() #no grid, blurs rectangles
    return c
```

The code for right Riemann sums is again given below:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as pat

def right_Riemann(f, a, b, n):
    '''
    Computes Right Riemann sum
    f = function to integrate, must define actual function
    a = lower bound on integration
    b = upper bound on integration
    n = number of rectangles to use
    '''
    w = (b-a)/n #same logic as left_Riemann
    x = np.linspace(a,b,n+1)
    xf = np.linspace(a,b, (b-a)*100)
    plt.figure()
    plt.plot(xf,f(xf), label = 'f(x)')
    ax = plt.gca()
    c = 0

    for i in range(1,n+1): #start at 1 else get rectangle outside lower bound
        #shift rectangle plots by w to get positioned right
        rect = pat.Rectangle((x[i]-w, 0), w, f(x[i]), fill = False)
        ax.add_patch(rect)
        c += f(x[i])
    c *= w
    plt.title("Right Riemann Sum")
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.show()

    return c
```

As usual, you are encouraged to understand how these actually work. Note that for monotonically increasing functions, the left Riemann sum underestimates the area, while for a monotonically decreasing function it overestimates the area. Vice-versa for Right Riemann sums. Of course, as we approach infinity, the left, right, and midpoint sums converge to the same area.

9.2 Trapezoid Rule

If you graph the rectangles associated with Riemann sums, for most functions (and sufficiently low n), you'll notice that the rectangles don't 'hug' the curve very well. It should come as no surprise that we do not necessarily have to use rectangles to approximate the area under curves. Another commonly used shape is a trapezoid, which is usually a better approximation than rectangles–we'll obtain more accurate estimates of the area with fewer trapezoids.

Using trapezoids, we may approximate an integral as

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}\sum_{k=0}^{n}\left[f(x_{k+1}) + f(x_k)\right] \tag{9.4}$$

where n is the number of trapezoids and $\Delta x \equiv (b-a)/n$.

Implementing equation (9.4) is rather straightforward and is left as an exercise.

9.3 Simpson's Rule

The trapezoid rule certainly 'hugs' many functions better than the rectangles in the Riemann sum. However, functions with significant curves (such as $\sin(x)$ or x^4) aren't perfectly 'hugged' by trapezoids either. What else can we use besides trapezoids? We can use quadratic polynomials! Such functions will approximate functions with steep curves quite well. There are a few variants of Simpson's rule. The most basic one approximates an integral of a function $f(x)$ from a to b as follows:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{b-a}{2}\right) + f(b)\right) \tag{9.5}$$

Just like it's better to use more trapezoids and rectangles for computing areas, it's also better to use more polynomials for approximating intervals. Indeed, equation (9.5) only fits one parabola. To fit multiple parabolas, we may use something called composite Simpson's rule, which instead approximates the integral as

$$\int_a^b f(x)dx \approx \frac{b-a}{3n}\sum_{i=1}^{n/2}\left(f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\right) \tag{9.6}$$

where n is strictly **even**. For $n = 2$, equation (9.6) is identical to equation (9.5). This rule is fairly simple to implement is Python as well, but it's difficult to graphically show the quadratic polynomials on the graph. For that reason, I will give you a function that computes composite Simpson's rule. The function is below:

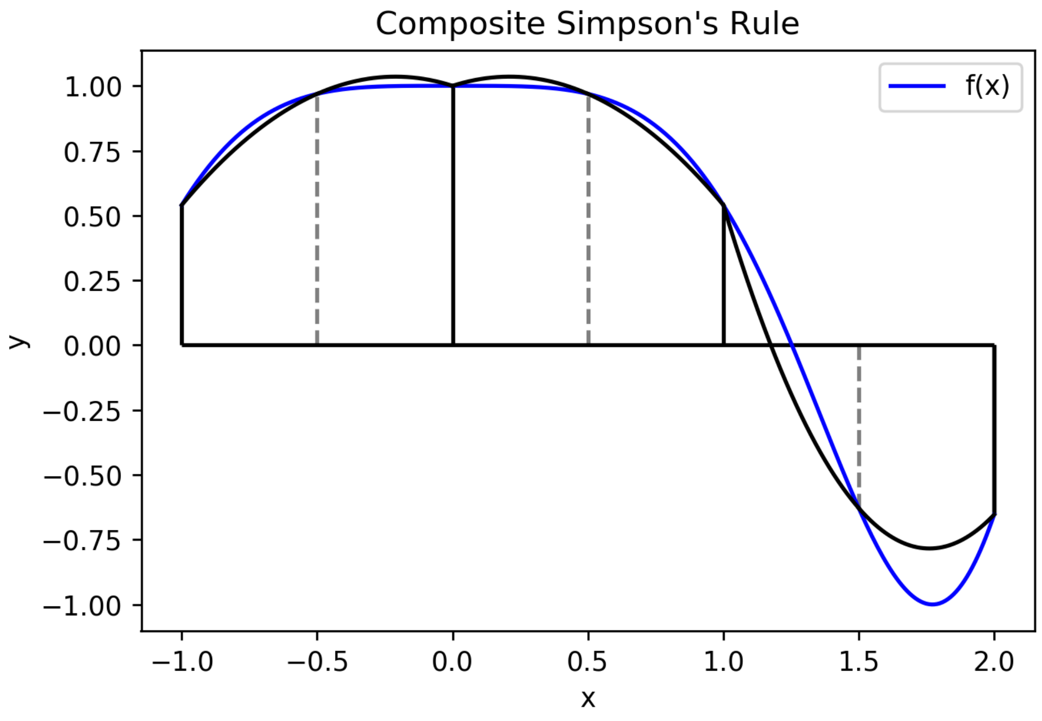
```
import numpy as np
import matplotlib.pyplot as plt

def quadratic(x,y):
    '''
    Determine quadratic from 3 points
    x = array of x-coordinates of points
    y = array of y-coordinates of points
    '''
    m = []
    for i in x:
        e1 = i**2
        e2 = i
        e3 = 1
        m.append([e1,e2,e3])
    aq, bq, cq = np.linalg.solve(m,y)
    return aq,bq,cq

def simpson(f,a,b,n):
```

```
'''
Use composite Simpson's Rule to compute area under curve
f = function to integrate
a = lower bound
b = upper bound
n = number of polynomials, must be even
'''
if n%2 != 0:
    print("Error: n must be even")
    raise ValueError

h = (b-a)/(n) #take care of prefactor
x = np.linspace(a,b,n+1) #generate x data for trapezoids
xf = np.linspace(a,b, (b-a)*100) #generate x data for function
c = 0 #initialize sum
plt.figure() #begin figure
plt.plot(xf, f(xf), label = 'f(x)', color = 'b') #plot function
plt.hlines(0,a,b) #plot base line, assume at zero
for i in range(1,int(n/2)+1): #compute sum
    #obtain quadratic equation from points given in formula
    aq,bq,cq = quadratic([x[2*i-2],x[2*i-1], x[2*i] ], \
                         [f(x[2*i-2]), f(x[2*i-1]), f(x[2*i])])
    xg = np.linspace(x[2*i-2], x[2*i-1], 1000)#x data for nth quadratic
    g =  aq*xg**2 + bq*xg + cq #y data for quadratics
    plt.plot(xg, g, color = 'k') #plot first quadratic
    plt.vlines(xg[0], 0, g[0]) #plot line separating quadratic blocks
    #add dashed line separating individual quadratics
    plt.vlines(xg[-1], 0, f(xg[-1]), linestyle='--', alpha = 0.5)
    xg = np.linspace(x[2*i-1], x[2*i], 1000)#x data for n+1 th quadratic
    g =  aq*xg**2 + bq*xg + cq #same plotting idea
    plt.plot(xg, g, color = 'k')
    plt.vlines(x[-1], f(x[-1]), 0)

    c +=  f(x[2*i-2]) + 4* f(x[2*i-1]) + f(x[2*i]) #compute sum

c *= h/3 #multiply by prefactor
plt.title('Composite Simpson\'s Rule') #decorate plot
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
return c
```

By far the most difficult part of the above code is determining how to place the quadratic polynomials on the graph. If you're hopelessly confused by that part, don't worry. The general approach, though, is that we want to compute three points: a beginning, midpoint, and endpoint. Using those three points, we draw a parabola that should approximate the function $f(x)$. In order to draw that parabola, we must use those three points to solve a system of equations to obtain the constants a, b, c in the usual quadratic equation. Once those constants are known, we may plot that parabola over the appropriate interval. At the midpoint of each quadratic, I have drawn a faint dashed line to separate 'block' quadratic into two distinct quadratics. A solid vertical line distinguishes separate 'blocks' of quadratics. This will make more sense if you run the code and play around with a few examples. For concreteness, let's just take a look at one quick example using the above code. If we want to integrate $f(x) = \cos(x^2)$ over the interval $[-1, 2]$ with $n = 6$, we would write, in our script that contains the simpson function,

```
f = lambda x: np.cos(x**2)
a = -1
b = 2
n = 6
simpson(f,a,b,n)
```

which gives us an area of about 1.3676 and the following figure:



Notice that for $n = 6$, we still don't 'hug' the function very nicely. However, for $n = 18$, we approximate the function almost perfectly.

Strictly speaking, Simpson's Rule will give us *exact* results if we're integrating polynomials up to and including third degree. This fact makes Simpson's Rule particularly useful when approximating integrals.

9.4 SciPy Quad

Of course, there are built-in functions that will do numerical integration for us. One of the most common is quad from SciPy. Its syntax is

scipy.integrate.quad(func, a, b)

where 'func' is the function you are integrating (defined using Python functions of course) and a and b are the lower and upper bounds respectively. The output of this function is the result of the integral and the absolute error. SciPy also has support for double and triple integrals, but we won't concern ourselves with those. For good measure, lets use this function to integrate $\cos(x^2)$ and compare it with our result using Simpson's Rule in the previous section. Below is a script that uses quad to do this:

```
import numpy as np
from scipy.integrate import quad

f = lambda x: np.cos(x**2)
a = -1
b = 2

ans, err, = quad(f, a, b)

print(ans, err)

output: 1.3659857003334883 3.81351836039706e-09
```

For convenience, the output of the code is included in the above block of code. Notice that our result using Simpson's Rule with $n = 6$ was about 1.3676, which is fairly close to the 'actual' value. Of course, making n larger and larger will make our approximation using Simpson's method.

In practice, most people will just use library functions (such as quad from SciPy) to evaluate integrals. However, there are cases where quad does not give accurate results (e.g. if a step function is integrated over a large boundary). In such cases, it might be useful to write your own integration function.

Chapter10
Numerical Differentiation

At first blush, numerical differentiation may not seem as straight forward as numerical integration. However, if we consider the basic definition of a derivative, we can easily see how we might go about approximating derivatives numerically. There are various approximation we may use, some better than others. We will discuss what are known as the forward, backward, and central difference formulas for derivative approximation, as well as a formula for second order derivatives. Finally, we'll discuss SciPy's numerical differentiation function.

10.1 First Order Forward, Backward,Central Difference Formulas

Recall that the definition of a derivative is

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{10.1}$$

Based on equation (10.1), it is reasonable to expect that we can approximate the derivative of a function as

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} \tag{10.2}$$

for some arbitrarily small value of h. Equation (10.2) is known as the forward difference approximation of a derivative. For a discrete set of data points, we may write equation (10.2) as

$$\frac{df(x)}{dx} \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \tag{10.3}$$

where i stands for the i-th data point. Looking at equation (10.3), we can note that it is possible to use the 'previous' data point (i.e. the i-1 th data point) instead of the next data point (i.e. the i+1 th data point). These formulas are called the backwards difference formulas, and can be expressed as

$$\frac{df(x)}{dx} \approx \frac{f(x) - f(x-h)}{h} \tag{10.4}$$

$$\frac{df(x)}{dx} \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \tag{10.5}$$

where equation (10.4) is for general functions and equation (10.5) is for discrete data. Naturally, we may wonder if there is an 'average' difference formula to compute the derivative. Indeed, there is such a formula, called the central difference formula. It is obtained by calculating the slope of the line passing through the points $(x+h, f(x+h))$ and $(x-h, f(x-h))$, which is

$$\frac{df(x)}{dx} \approx \frac{f(x+h)-f(x-h)}{x+h-(x-h)} = \frac{f(x+h)-f(x-h)}{2h} \tag{10.6}$$

or, for discrete, data,

$$\frac{df(x)}{dx} \approx \frac{f(x_{i+1})-f(x_{i-1})}{2h}. \tag{10.7}$$

It turns out (and is fairly easy to see) that the central difference formula for the derivative is a more accurate approach to numerically approximating derivatives.

It is remarkably easy to implement these formulas in Python. A function that calculates the derivative of some given function $f(x)$, specified via Python functions, is given below:

```
#based on math.ubc.ca/~pwalls/math-python/differentiation/differentiation/
def derivative(f, x, method='central', h=0.01):
    '''
    Compute the difference formula for f'(x) with step size h.
    f = function to differentiate
    x = point to differentiate. May be single value or list
    '''

    if method == 'central':
        return (f(x + h) - f(x - h))/(2*h)
    elif method == 'forward':
        return (f(x + h) - f(x))/h
    elif method == 'backward':
        return (f(x) - f(x - h))/h
    else:
        raise ValueError("Method must be 'central', 'forward' or 'backward'.")
```

There will be some exercises for you on working with this function and plotting derivatives. For our purposes, this is all we need to cover about first order difference formulas.

10.2 Second Order Central Difference Derivatives

Unsurprisingly, there are numerous formulas for calculating higher order derivatives. However, we will only look at one of the simpler ones that immediately follows from the previous sections. Since the central difference approximation is the most accurate, we will only concern ourselves with finding a second order formula for that, and not worry about second order forward or backwards difference formulas.

It should be intuitively clear that if we want a central difference approximation for a second order derivative, we should compute

$$\frac{d^2f(x)}{dx^2} \approx \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} = \frac{f(x+h)-2f(x)+f(x-h)}{h^2}. \tag{10.8}$$

The implementation of this formula is very similar to the implementation of the first order derivatives. I will leave it as an exercise for you to write a function that implements equation (10.8).

10.3 SciPy's Derivative

As usual, SciPy has a package that does exactly what we've been doing. The syntax for this function is

scipy.misc.derivative(func, x0, dx, n)

where 'func' is the Python function you are differentiating, $x0$ is the point at which the derivative is computed, dx is the step-size, and n is the order of the derivative (first, second, etc.). This function uses the central difference method, and returns the value of the derivative at $x0$. Application of this formula is trivial, so an example is not included.

There is one important point to note in regards to the step size h (or dx as it's taken in SciPy). Realistically, we want our step size to be a small as possible (in the actual limit, it should approach zero after all). However, there is a danger in making the step size too small. The danger lies in a topic we won't cover in detail in this course, which is machine precision and round-off error. Essentially, when dealing with numbers that are very large or very small (say, to the order of $10^{\pm 15}$), the computer's ability to do arithmetic plunges. This is because it simply cannot accurately store numbers of those magnitude (in Python, you can circumvent this problem using the decimal package, which allows for arbitrary precision, but we don't need to worry about that). Practically, this means that if our step size is too small, both SciPy and the function we write will **not** give accurate approximations of the derivative. An example of this will be given as an exercise. Remember, be careful and always do sanity checks on results computers spit out!

Chapter11
Numerical Methods of Ordinary Differential Equations

Nothing is more important in physics than differential equations. Every fundamental law of nature can, in some form or another, be expressed as a differential equation. Even for relatively simple systems, such as a double pendulum, the equations of motion consist of coupled ordinary differential equations that cannot be solved analytically. In fact, for most realistic problems–and even for many idealized problems–we cannot solve the equations of motion of a system analytically. Numerical methods here are absolutely essential if we want to determine how a physical system evolves with time.

As expected, there are many methods we can use to numerically solve ordinary differential equations (ODEs). Some have more utility than others. In this chapter, we will only cover a small handful, but it should be sufficient to give you an overview of these numerical methods.

11.1 Forward Euler

Perhaps the simplest way to numerically solve an ODE is with the Forward Euler method. This method can be formulated such that it applies to both first order and second order differential equations. Let's start with the formulation for solving first order ODEs.

This method applies to differential equations of the form

$$y'(t_n) = f(t_n, y(t_n)),\ y(t_0) = y_0 \tag{11.1}$$

Let us expand the function $y(t_n + h)$, where $h \equiv t_n - t_0 = \text{constant}$ is some small step size, in a Taylor series around t_n. Doing so gives us

$$y(t_n + h) \equiv y_{n+1} = y(t_n) + y'(t_n)(t_n + h - t_n) + \mathcal{O}((h^2)) \approx y(t_n) + hf(t_n, y(t_n)) \tag{11.2}$$

The approximation comes from ignoring terms in the Taylor expansion of second order and higher. Equation (11.2) tells us how we can find y as time progresses, which is what we were trying to find. This equation is known as the Forward Euler method, and is a simple and easy approximation to many common ODEs. I will leave the implementation of this method as an exercise for you.

Suppose, now, we have a second order ODE of the form

$$y''(t_n) = f(t_n, y(t_n)) \tag{11.3}$$

To solve a second order ODE like equation (11.3), we can try to rewrite it as a system of first order ODEs, for which we already know the solution. In this case, it is easy to do so. If we let

$$v(t_n) = y'(t_n), \tag{11.4}$$

we may rewrite equation (11.3) as

$$v'(t_n) = f(t_n, y(t_n)) \tag{11.5}$$

which we just saw (equation (11.2)) has the solution of

$$v(t_n + h) \equiv v_{n+1} = v(t_n) + hf(t_n, y(t_n)). \tag{11.6}$$

We still must find a solution for equation (11.4), but this is trivial to integrate analytically. Assume we have the initial conditions $v(t_0) = v_0$ and $y(t_0) = y_0$. Then we may write equation (11.4) as

$$\begin{aligned} v(t_n) \equiv v_n = \frac{dy}{dt} &\implies dy = v_n dt \\ &\implies \int_{y_0}^{y_n} y = v_n \int_{t_0}^{t_n} dt \\ &\implies y_n = y_0 + hv_n. \end{aligned} \tag{11.7}$$

Note that we assume the velocity is constant over a time step h, so we may pull v_n outside the integral in the above calculation. Equations (11.6) and (11.7) are thus the formulation for solving a second order ODE using the Forward Euler method.

Note that when solving a first order ODE, we only need one initial condition (for the initial position of the system). When solving a second order ODE, we must be given two initial conditions: one for the initial position and one for the initial velocity. I will again leave the implementation of this as an exercise.

While the Forward Euler method may be easy and quick to program, it has a potentially fatal drawback. Notably, it does *not* conserve energy. So, if we are trying to model a periodic system (e.g. a pendulum or planetary motion) our approximation of the motion will become less and less accurate the longer our simulation runs. For this reason, Forward Euler isn't an ideal choice for modeling periodic systems for extended time frames. In these cases, we want to use some other algorithm.

11.2 Euler-Cromer

The Euler-Cromer method is a simple modification of the Forward Euler method for second order ODEs. Note that in the Forward Euler method, we calculate the position, then the velocity at each iteration. In the Euler-Cromer method, we just calculate the velocity first, then the the position. It follows that the formulation of the Euler-Cromer method is simply

$$v_{n+1} = v_n + dtf(t_n, y(t_n)) \tag{11.8}$$
$$y_{n+1} = y_n + dtv_{n+1} \tag{11.9}$$

Unlike Forward Euler, Euler-Cromer *does* conserve energy...almost. Using this method, energy is not perfectly conserved–the total energy of the system oscillates. However, these oscillations do not become unbound, i.e. the energy will fluctuate between two values indefinitely.

This method isn't too difficult to program, but you need to have at least one example in this section to go off of. So, below is a function that implements the Euler-Cromer method:

```
def EulerCromer(f, a, b, y0, v0, dt = 0.01):
    '''
    Solve a second order ODE via Euler-Cromer method
    f = function to solve. MUST be of form f(y(t), t)
    Even if f has no y(t) dependence, it must be specified in function
    a = lower bound
    b = upper bound
    y0 = initial state of system; starting position
    v0 = initial state of system; starting velocity
    dt = step size (think time step)
    '''
    nt = int((b-a)/dt + 1) #calculate number of points to use
    t = np.linspace(a, b, nt) #make time array
    y = np.zeros(nt) #initialize y array
    v = np.zeros(nt)# initialize v array
    y[0] = y0 #insert initial y guess
    v[0] = v0 #insert initial v guess
    plt.figure() #begin figure
    plt.title("Euler-Cromer Approximation")
    plt.xlabel('t') #add axes labels
    plt.ylabel('f(t)')
    for i in range(nt-1): # do the algorithm
        v[i+1] = v[i] + dt*f(y[i], t[i])
        y[i+1] = y[i] + dt*v[i+1]

    plt.plot(t, y, 'o', label = 'EC, position')#make plots
    plt.plot(t,v,'o', label = 'EC, velocity')

    plt.legend() #add decorations to graph
    plt.grid()
    plt.show()
    return y, v
```

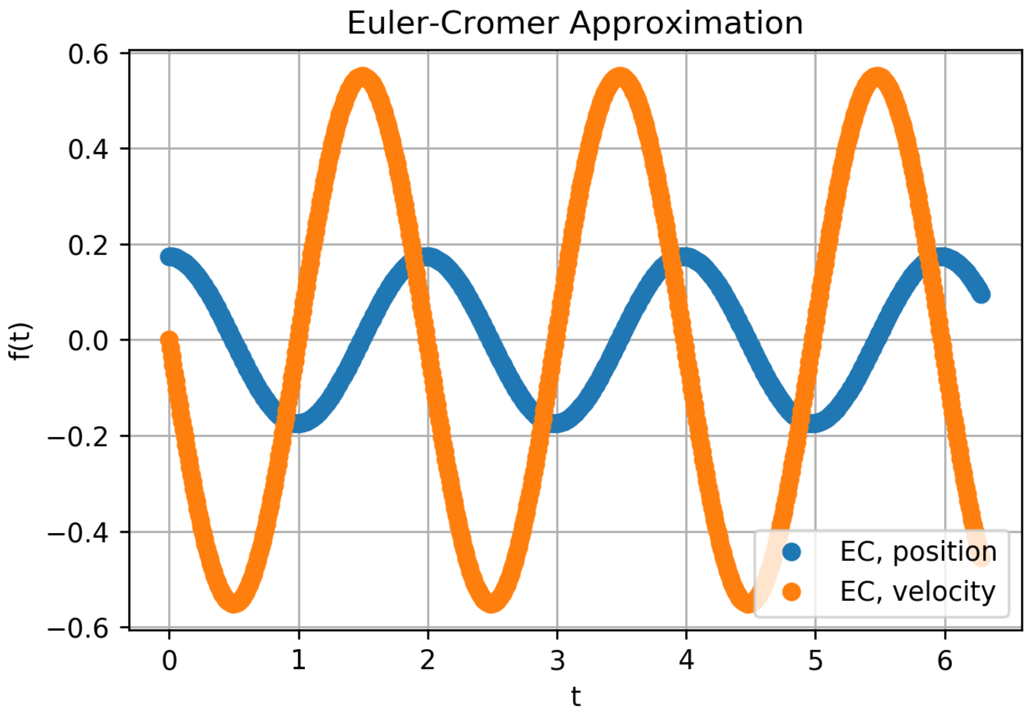
Note that the position in which you specify the arguments of your function f do matter. In the above code, you must write a Python function of $f(y(t), t)$, **not** $f(t, y(t))$. To illustrate this function, let's consider the basic equation of motion of a simple pendulum:

$$\ddot{\theta} = -a\sin(\theta) \tag{11.10}$$

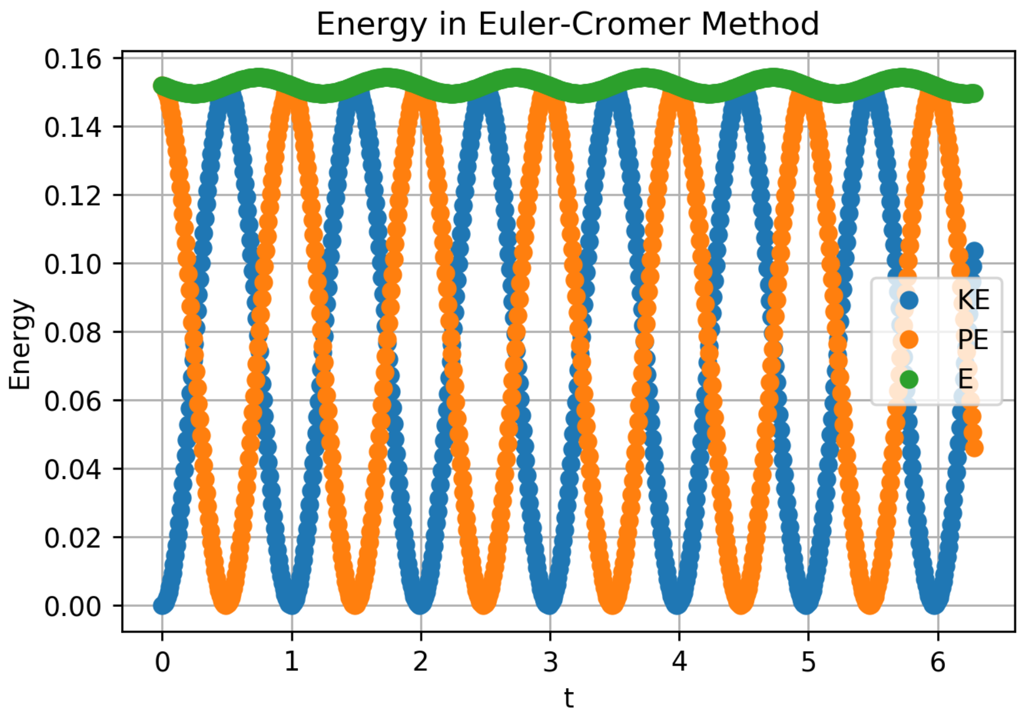
where a is some constant greater than zero. If we solve for the motion and velocity of the pendulum using the Euler-Cromer method and plot the result, the code will look like:

```
f = lambda y, t: -10*np.sin(y)
a = 0
b = 6
y0 = np.pi/180 * 1 #convert to radians
v0 = 0

ans2 = velVerlet(f, a, b, y0, v0)
```

which will produce the following graph:



While not explicitly shown in the above code, we can fairly easily make a graph of the energy over time in the Euler-Cromer method (again, this will be an exercise for you):



These graphs are more or less what we expect, at least in terms of extreme behavior. In other words, when the velocity is zero, the position of the pendulum is either a maximum or minimum. When the velocity is at a maximum or minimum, the pendulum is at its zero point crossing. Similarly, in the energy graph, the kinetic energy (KE) and potetial energy (PE) are mirrors of each other, just as we would expect.

Again, note that the total energy E is actually not constant, but rather oscillates in time. There is a different algorithm, velocity Verlet, that has less severe total energy oscillations for this same system, and is generally more stable. Before we move on to that method, we should first consider the basic Verlet method.

11.3 Verlet

This method of ODE approximation is also a commonly used algorithm that is rather easy to implement. This method is intended to solve ODEs of the form $y'' = f(y(t))$. As we did in equation (11.2), let us consider the Taylor expansion of two functions: $y(t_n + h)$ and $y(t_n - h)$:

$$y(t_n + h) = y(t_n) + hy'(t_n) + \frac{1}{2}y''(t_n)h^2 + \mathcal{O}(h^3) \tag{11.11}$$

$$y(t_n - h) = y(t_n) - hy'(t_n) + \frac{1}{2}y''(t_n)h^2 + \mathcal{O}(h^3) \tag{11.12}$$

Adding equations (11.11) and (11.12) and slightly rearranging gives us

$$y(t_n + h) = 2y(t_n) - y(t_n - h) + f(y(y))h^2 + \mathcal{O}(h^4) \tag{11.13}$$
$$\leftrightarrow y_{n+1} = 2y_n - y_{n-1} + f(y(t_n))h^2. \tag{11.14}$$

It so happens that the third order terms in the Taylor expansion cancel out when adding equations (11.11) and (11.12), but this is only significant if we discuss error analysis, which we aren't. So, equation (11.14) is known as the Verlet method of integration.

Notice that equation (11.14) is not 'self-starting.' We must know the initial position, $y(t_0)$ *and* the position at the first time step, $y(t_1)$. To approximate $y(t_1)$, we may use the same Taylor expansion as we did in equation (11.11) but expand it around t_0. Doing so gives us

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}y''(t_0)h^2 \tag{11.15}$$
$$\leftrightarrow y_{n+1} = y_n + hv_0 + \frac{1}{2}f(y_0)h^2 \tag{11.16}$$

So, equations (11.14) and (11.16) constitute the Verlet algorithm. Notice that, except for equation (11.16), this algorithm does not depend on the velocity of the system. As you might expect, it is crucial for us to have correct initial conditions, else we won't accurately model the system.

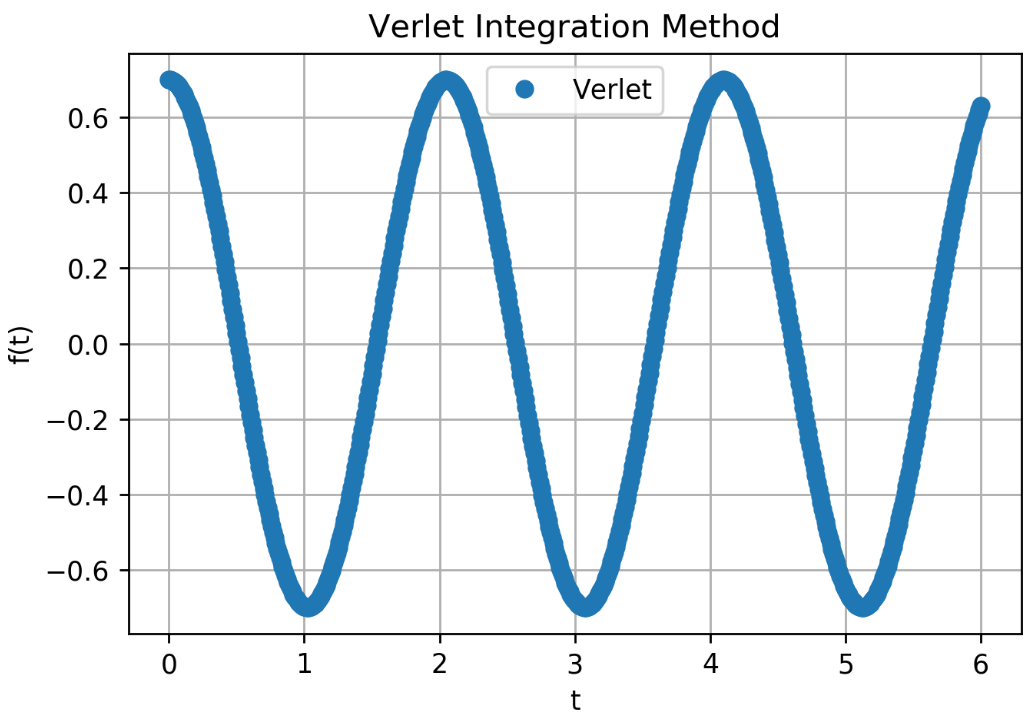
As an example, a function that implements and graphs the position of a system in the Verlet method is given below:

```
def Verlet(f, a, b, y0, v0, dt = 0.01):
    '''
    Use Verlet method to solve ODE and plot solution
    f = function to solve. MUST be of form D^2(y) = F(y(t))
    a = lower bound
    b = upper bound
    y0 = initial state of system; starting position
    v0 = initial state of system, starting velocity
    dx = step size (think time step)
    comp = boolean; if true, also plots ODE int solution
    '''

    nt = int((b-a)/dt + 1) #set number of time steps
    t = np.linspace(a, b, nt) #get times to calculate system
    y = np.zeros(nt) #initialize y array
    y[0] = y0 #set initial condition
    y[1] = y[0] + v0*dt + 1/2 * f(y[0]) * dt**2 #set other initial condition
    plt.figure()# begin figure
    plt.title("Verlet Integration Method")
    plt.xlabel("t")
    plt.ylabel("f(t)")
    for i in range(1, nt-1): #do algorithm
        y[i+1] = 2*y[i] - y[i-1] + f(y[i])*dt**2

    plt.plot(t, y, 'o', label = 'Verlet') #mke plot
    plt.grid() #add plot details
    plt.legend()
    plt.show()

    return y
```

As an example, let's again do the simple pendulum. Our input to this function would then look like

```
f = lambda y: -10*np.sin(y)
a = 0
b = 6
y0 = np.pi/180 * 40 #radians!
v0 = 0

ans = velVerlet(f, a, b, y0, v0, dt = 0.01)
```

which will give us the following plot:



As expected, we obtain sinusoidal oscillations that do not become unbound as we plot more and more cycles. Now, let's take a look at how we can incorporate velocities explicitly in this method.

11.4 Velocity-Verlet

The velocity-Verlet is more commonly used than the regular Verlet algorithm, largely because we do not need to know an initial condition at the first time step. As usual, let's first begin by expanding $y(t_n + h)$ in a Taylor series about t_n. To second order, this gives us

$$y(t_n + h) \approx y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(t_n). \tag{11.17}$$

Now, we want to explicitly incorporate velocities, so let us also do a Taylor expansion of $y' = v(t_n + h)$ about t_0. This gives

$$v(t_n + h) \approx v(t_n) + hv'(t_n) + \frac{1}{2}h^2 v''(t_n). \tag{11.18}$$

Now, we need to eliminate v''. This can be done by, surprise, a Taylor expansion of v' to first order:

$$v'(t_n + h) \approx v'(t_n) + hv''(t_n) \tag{11.19}$$

$$\implies v''(t_n) = \frac{v'(t_n + h) - v'(t_n)}{h} \tag{11.20}$$

Substituting equation (11.20) into (11.18) and simplifying, we finally obtain

$$v(t_n + h) = \frac{1}{2}h(v'(t_n + h) + v'(t_n)) \leftrightarrow v_{n+1} = v_n + \frac{1}{2}h(f(y_{n+1}) + f(y_n)) \tag{11.21}$$

Equations (11.17) and (11.21) thus constitute the velocity-Verlet algorithm. I leave its implementation as an exercise for you.

11.5 SciPy's ODEint

Lastly, we can solve an ODE using SciPy's ODEint. The syntax for this function is

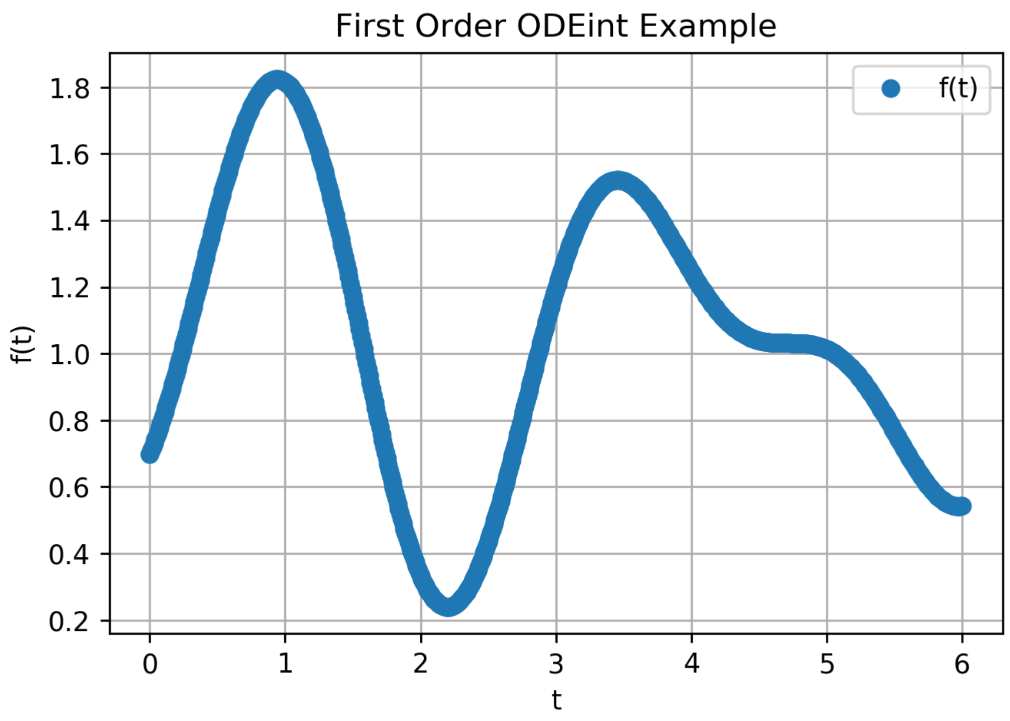
scipy.integrate.odeint(func, y0, t)

where 'func' is the ODE you want to integrate, defined using a Python function, $y0$ is the initial position (or a list/array of initial guesses), and t is an array of times for which we evaluate the ODE. It is designed to solve ODEs of the form $dy/dt = f(y, t)$. This function returns an array of values for y. If we use this to solve a second order ODE, it will return an array of value pairs, each corresponding to the single ODE we broke the second order ODE into. This will make more sense with an example or two.

Let's take a look at two examples, one for a first order ODE and another for a second order ODE. In both cases, we'll evaluate over the ODE over the interval $[0, 6]$. Using ODEint to solve $\frac{dy}{dt} = cos(2x) + sin(3x)$, we have

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

a = 0
b = 6
dt = 0.01
nt = int((b-a)/dt + 1)
t = np.linspace(a, b, nt)
y0 = np.pi/180 * 40
v0 = 0
#need to assume function of f(y, x) else ODEint complains
f = lambda y, x: np.cos(2*x) + np.sin(3*x) #dy/dt = cos(3x) + sin(2x)

ans= odeint(f, y0, t)
plt.figure()
plt.plot(t, ans, 'o', label = 'f(t)')
plt.title("First Order ODEint Example")
plt.xlabel('t')
plt.ylabel('f(t)')
plt.grid()
plt.legend()
plt.show()
```

This gives us the output

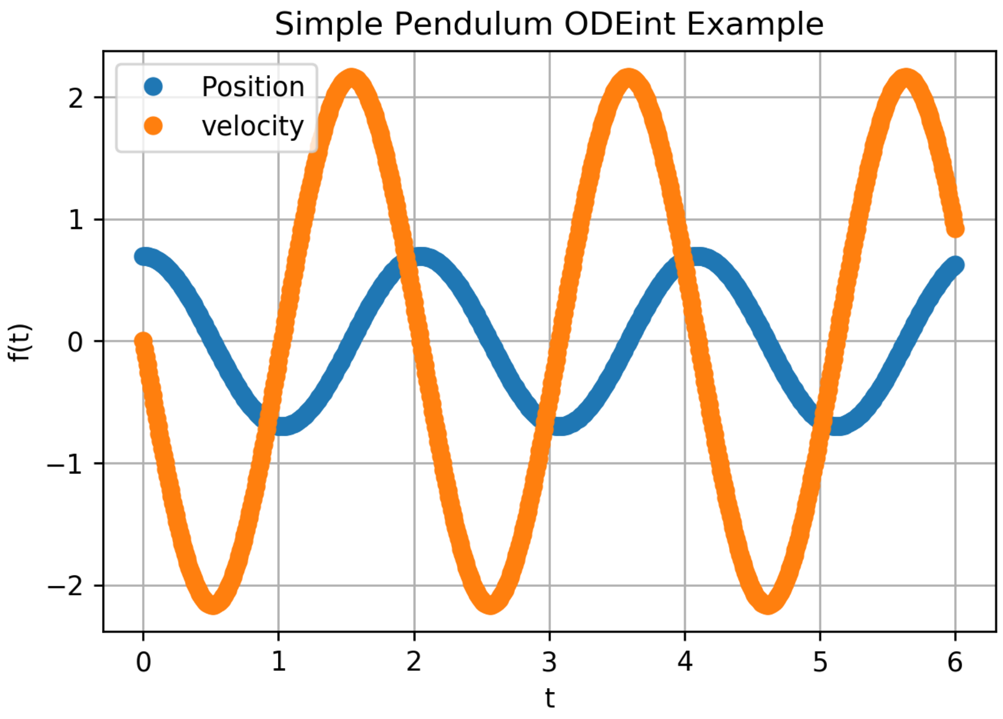


Notice that even though our ODE is only explicitly dependent on t, not y, we still must specify our lambda function with y as the first argument. Otherwise, ODEint will complain about the number of arguments in lambda.

Next, let's solve the equation of motion of a pendulum $\ddot{y} = -a\sin(y)$. As we did earlier, we want to break this second order ODE into two first order ODEs. We can do that by assigning $v = y'$ and $v' = -a\sin(y)$. In practice, this would look like

```
def func(z, t):
    y,v = z
    return v, -10*np.sin(y)

ans2 = odeint(func, [y0, v0], t)
ans2p, ans2v = list(zip(*ans2)) #unzip lists
plt.figure()
plt.plot(t, ans2p, 'o', label = 'Position')
plt.plot(t, ans2v, 'o', label = 'velocity')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Simple Pendulum ODEint Example')
plt.grid()
plt.legend()
plt.show()
```

Which gives us the figure



While ODEint can get the job done, it isn't uncommon for people to explicitly use one of the algorithms discussed above, or a completely different one. ODEint uses a solver called the Livermore Solver for Ordinary Differential Equations, which is a collection of various ODE solvers from a FORTRAN package that I'm not familiar with. Different ODE techniques are useful for different situations, so it's useful to know a variety of methods and how to code them.