Computational Physics With Python

Dr. Eric Ayars

California State University, Chico

Chapter 0

Useful Introductory Python

0.0 Making graphs

Python is a scripting language. A script consists of a list of commands, which the Python interpreter changes into machine code one line at a time. Those lines are then executed by the computer.

For most of this course we'll be putting together long lists of fairly complicated commands —programs— and trying to make those programs do something useful for us. But as an appetizer, let's take a look at using Python with individual commands, rather than entire programs; we can still try to make those commands useful!

Start by opening a terminal window.[1] Start an interactive Python session, with pylab extensions[2], by typing the command ipython −−pylab followed by a return. After a few seconds, you will see a welcome message and a prompt:

```
In [1]:
```

Since this chapter is presumbly about graphing, let's start by giving Python something to graph:

```
In [1]: x = array([1,2,3,4,5])
In [2]: y = x+3
```

[1] In all examples, this book will assume that you are using a Unix-based computer: either Linux or Macintosh. If you are using a Windows machine and are for some reason unable or unwilling to upgrade that machine to Linux, you can still use Python on a command line by installing the Python(x,y) package and opening an "iPython" window.

[2] All this terminology will be explained eventually. For now, just use it and enjoy the results.

Next, we'll tell Python to graph y versus x, using red $\times$ symbols:

```
In [3]: plot(x,y,'rx')
Out[3]: [<matplotlib.lines.Line2D at (gibberish)>]
```

In addition to the nearly useless Out[] statement in your terminal window, you will note that a new window opens showing a graph with red $\times$'s.

The graph is ugly, so let's clean it up a bit. Enter the following commands at the iPython prompt, and see what they do to the graph window: (I've left out the In []: and Out[]: prompts.)

```
title('My first graph')
xlabel('Time (fortnights)')
ylabel('Distance (furlongs)')
xlim(0, 6)
ylim(0, 10)
```

In the end, you should get something that looks like figure 0.

Let's take a moment to talk about what's we've done so far. For starters, x and y are *variables*. Variables in Python are essentially storage bins: x in this case is an address which points to a memory bin somewhere in the computer that contains an *array* of 5 numbers. Python variables can point to bins containing just about anything: different types of numbers, lists, files on the hard drive, strings of text characters, true/false values, other bits of Python code, *whatever*! When any other line in the Python script refers to a variable, Python looks at the appropriate memory bin and pulls out those contents. When Python gets our second line

```
In [2]: y = x+3
```

It pulls out the x array, adds three to everything in that array, puts the resulting array in another memory bin, and makes y point to that new bin.

The plot command plot(x,y,'rx') creates a new figure window if none exists, then makes a graph in that window. The first item in parenthesis is the x data, the second is the y data, and the third is a description of how the data should be represented on the graph, in this case red $\times$ symbols.

Here's a more complex example to try. Entering these commands at the iPython prompt will give you a graph like figure 1:

```
time = linspace(0.0, 10.0, 100)
height = exp(-time/3.0)*sin(time*3)
figure()
```

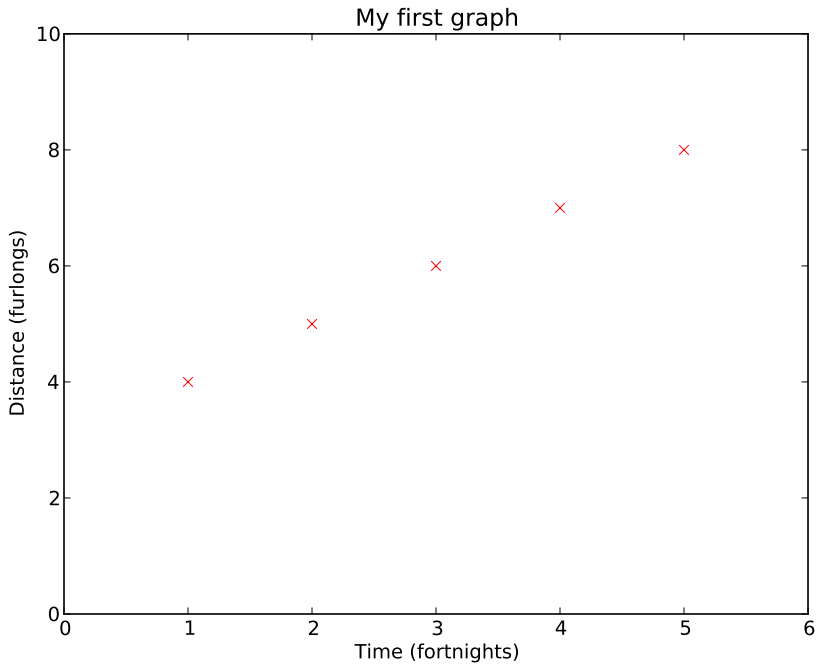

Figure 0: A simple graph made interactively with iPython.

```
plot(time, height, 'm-^')
plot(time, 0.3*sin(time*3), 'g-')
legend(['damped', 'constant amplitude'], loc='upper right')
xlabel('Time (s)')
```

The linspace() function is very useful. Instead of having to type in values for all the time axis points, we just tell Python that we want linearly-spaced numbers from (in this case) 0.0 through 10.0, and we want 100 of them. This makes a nice x-axis for the graph. The second line makes an array called 'height', each element of which is calculated from the corresponding element in 'time' The figure() command makes a new figure window. The first plot command is straightforward (with some new color and symbol indicators), but the second plot line is different. In that second line we just put a calculation in place of our y values. This is perfectly fine with Python: it just needs an array there, and does not care whether it's an array that was retrieved from a memory bin (i.e. 'height') or an array calculated on the

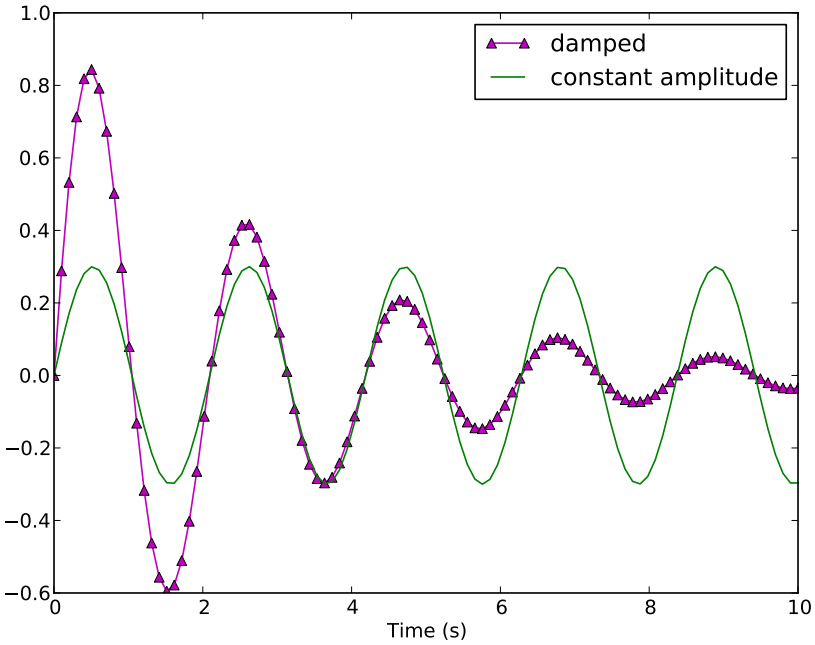

Figure 1: More complicated graphing example.

spot. The legend() command was given two parameters. The first parameter is a *list*[3]:

```
['damped', 'constant amplitude']
```

Lists are indicated with square brackets, and the list elements are separated by commas. In this list, the two list elements are strings; strings are sequences of characters delimited (generally) by either single or double quotes. The second parameter in the legend() call is a labeled option: these are often built in to functions where it's desirable to build the functions with a default value but still have the option of changing that value if needed[4].

[3]See section 1.3.
[4]See section 1.7.

0.1 Libraries

By itself, Python does not do plots. It doesn't even do trig functions or square roots. But when you start iPython with the '-pylab' option, you are telling it to load optional *libraries* that expand the functionality of the Python language. The specific libraries loaded by '-pylab' are mathematical and scientific in nature; but Python libraries are available to read web pages, create 3D animations, parse XML files, pilot autonomous aircraft, and just about anything else you can imagine. It's easy to make libraries in Python, and you'll learn how as you work your way through this class. But you will find that for many problems someone has already written a Python library that solves the problem, and the quickest and best way of solving the problem is to figure out how to use their library!

For plotting, the preferred Python library is "matplotlib". That's the library being used for the plots you've made in this chapter so far; but we've barely scratched the surface of what the matplotlib library is capable of doing. Take a look online at the "matplotlib gallery": `http://matplotlib.org/gallery.html`. This should give you some idea of the capabilities of matplotlib. This page very useful: clicking on a plot that shows something similar to what you want to create gives example code showing how that graph was created!

Another extremely useful library for physicists is the 'LINPACK' linear algebra package. This package provides very fast routines for calculating *anything* having to do with matrices: eigenvalues, eigenvectors, solutions of systems of linear equations, and so on. It's loaded under the name 'linalg' when you use ipython −−pylab.

Example 0.1.1
In electronics, Kirchhoff's laws are used to solve for the currents through components in circuit networks. Applying these laws gives us systems of linear equations, which can then be expressed as matrix equations, such as

$$\begin{bmatrix} -13 & 2 & 4 \\ 2 & -11 & 6 \\ 4 & 6 & -15 \end{bmatrix} \begin{bmatrix} I_A \\ I_B \\ I_C \end{bmatrix} = \begin{bmatrix} 5 \\ -10 \\ 5 \end{bmatrix} \tag{1}$$

This can be solved algebraically without too much difficulty, or one can simply solve it with LINPACK:

```
A = matrix([ [-13,2,4], [2,-11,6], [4,6,-15] ])
```

```
B = array([5,-10,5])
linalg.solve(A,B)
--> array([-0.28624535, 0.81040892, -0,08550186])
```

One can easily verify that the three values returned by linalg.solve() are the solutions for I_A, I_B, and I_C.

LINPACK can also provide eigenvalues and eigenvectors of matrices as well, using linalg.eig(). It should be noted that the size of the matrix that LINPACK can handle is limited only by the memory available on your computer.

0.2 Reading data from files

It's unlikely that you would be particularly excited by the prospect of manually typing in data from every experiment. The whole point of computers, after all, is to *save* us effort! Python can read data from text files quite well. We'll discuss this ability more in later in section 1.8, but for now here's a quick and dirty way of reading data files for graphing.

We'll start with a data file like that shown in table 1. This data file (which actually goes on for another three thousand lines) is from a lab experiment in another course at this university, and a copy has been provided[5]. Start iPython/pylab if it's not open already, and then use the loadtxt() function to read columns of data directly into Python variables:

Table 1: File microphones.txt

#Frequency	Mic 1	Mic 2
10.000	0.654	0.192
11.000	0.127	0.032
12.000	0.120	0.030
13.000	0.146	0.031
14.000	0.155	0.033
15.000	0.175	0.036
...		

[5] `/export/classes/phys312/examples/microphones.txt`

```
frequency, mic1, mic2 = loadtxt('microphones.txt', unpack = True)
```

The loadtxt() function takes one required argument: the file name. (You may need to adjust the file name (microphones.txt) to reflect the location of the actual file on your computer, or move the file to a more convenient location.) There are a number of optional arguments: one we're using here is "unpack", which tells loadtxt() that the file contains columns of data that should be returnend in separate arrays. In this case, we've told Python to call those arrays 'frequency', 'mic1', and 'mic2'. The loadtxt() function is very handy, and reasonably intelligent. By default, it will ignore any line that begins with '#', as it assumes that such lines are comments; and it will assume the columns are separated by tabs. By giving it different optional arguments you can tell it to only read certain rows, or use commas as delimiters, etc. It will choke, though, if the number of items in each row is not identical, or if there are items that it can't interpret as numbers.

Now that we've loaded the data, we can plot it as before:

```
figure()
plot(frequency, mic1, 'r-', frequency, mic2, 'b-')
xlabel('Frequency (Hz)')
ylabel('Amplitude (arbitrary units)')
legend(['Microphone 1', 'Microphone 2'])
```

See figure 2.

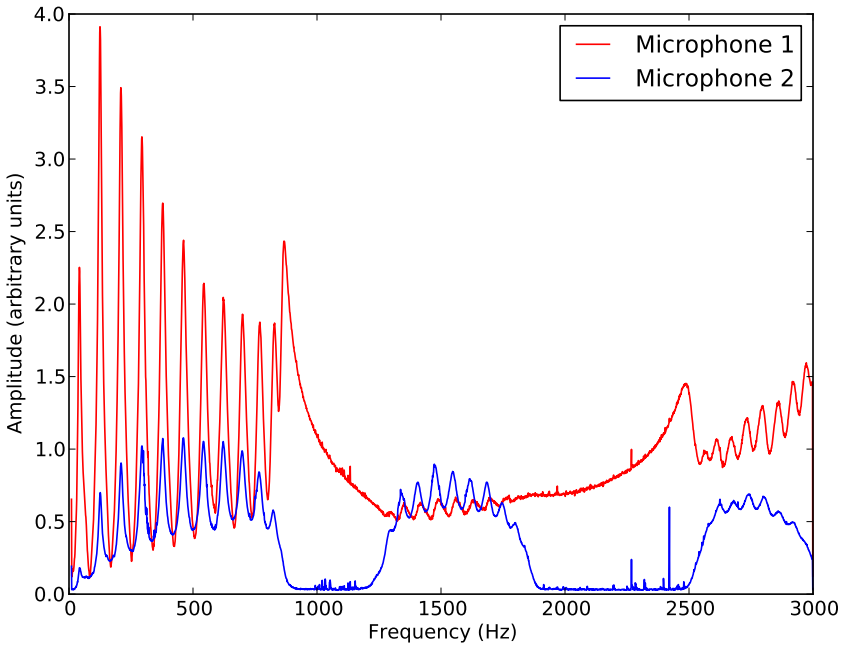

Figure 2: Data from 'microphones.txt'

0.3 Problems

0-0 Graph both of the following functions on a single figure, with a usefully-sized scale.

(a)
$$x^4 e^{-2x}$$

(b)
$$\left[x^2 e^{-x} sin(x^2)\right]^2$$

Make sure your figure has legend, range, title, axis labels, and so on.

0-1 The data shown in figure 2 is most usefully analyzed by looking at the *ratio* of the two microphone signals. Plot this ratio, with frequency on the x axis. Be sure to clean up the graph with appropriate scales, axes labels, and a title.

0-2 The file Ba137.txt contains two columns. The first is counts from a Geiger counter, the second is time in seconds.

(a) Make a useful graph of this data.

(b) If this data follows an exponential curve, then plotting the natural log of the data (or plotting the raw data on a logrithmic scale) will result in a straight line. Determine whether this is the case, and explain your conclusion with —you guessed it— an appropriate graph.

0-3 The data in file Ba137.txt is actual data from a radioactive decay experiment; the first column is the number of decays N, the second is the time t in seconds. We'd like to know the half-life $t_{1/2}$ of ^{137}Ba. It should follow the decay equation

$$N = N_o e^{-\lambda t}$$

where $\lambda = \frac{\log 2}{t_{1/2}}$. Using the techniques you've learned in this chapter, load the data from file Ba137.txt into appropriately-named variables in an ipython session. Experiment with different values of N and λ and plot the resulting equation on top of the data. (Python uses exp() calculate the exponential function: i.e. y = A∗exp(−L∗time)) Don't worry about automating this process yet (unless you *really* want to!) just try adjusting things by hand until the equation matches the data pretty well. What is your best estimate for $t_{1/2}$?

0-4 The normal modes and angular frequencies of those modes for a linear system of four coupled oscillators of mass m, separated by springs of equal strength k, are given by the eigenvectors and eigenvalues of M, shown below.

$$M = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

(The eigenvalues give the angular frequencies ω in units of $\sqrt{\frac{k}{m}}$.) Find those angular frequencies.

0-5 Create a single plot that shows separate graphs of position, velocity, and acceleration for an object in free-fall. Your plot should have a single horizontal time axis and separate stacked graphs showing position, velocity, and acceleration each on their own vertical axis. (See figure 3.) The online matplotlib gallery will probably be helpful! Print the graph, with your name in the title.

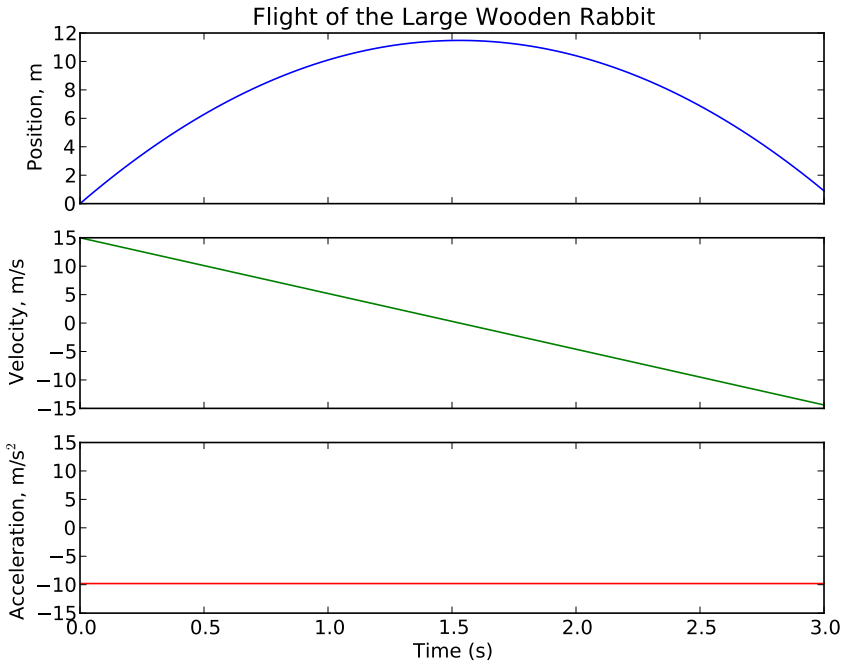

Figure 3: Sample three-graph plot

Chapter 1

Python Basics

1.0 The Python Interpreter

Python is a computer program which converts human-friendly commands into computer instructions. It is an *interpreter*. It's written in another language; most often C++, which is more powerful and much faster, but also harder to use.[1]

There is a fundamental difference between interpreted languages (Python, for example) and compiled languages such as C++. In a compiled language, all the instructions are analyzed and converted to machine code by the compiler *before* the program is run. Once this compilation process is finished, the program can run very fast. In an interpreted language, each command is analyzed and converted "on the fly". This process makes interpreted languages significantly slower; but the advantage to programming in interpreted languages is that they're easier to tweak and debug because you don't have to re-compile the program after every change.

Another benefit of an interpreted language is that one can experiment with simple Python commands by using the Python interpreter directly from the command line. In addition to the iPython method shown in the previous chapter, it's possible to use the Python interpreter directly. From a terminal window (Macintosh or Linux) type `python<enter>`, or open a window in "Idle" (Windows). You will get the Python prompt: `>>>`. Try some simple mathematical expressions, such as `6*7<enter>`. The Python interpreter takes each line of input you give it and attempts to make sense of it: if it can, it replies with what it got.

[1] There are versions of Python written in other languages, such as Java & C#. There is also a version of Python written in Python, which is somewhat disturbing.

You can use Python as a very powerful calculator if you want. It can also store value in variables. Try this:

```
x = 4
y = 16
x*y
x**y
y/x
x**y**x
```

That last one may take a moment or two: Python is actually calculating the value of $4^{(16^4)}$, which is a rather huge number.

In addition to taking commands one line at a time, the Python interpreter can take a file containing a list of commands, called a *program*. The rest of this book, and course, is about putting together programs so as to solve physics problems.

1.1 Comments

A program is a set of instructions that a computer can follow. As such, it has to be comprehensible by the computer, or it won't run at all. The rest of this chapter is concerned with the specifics of making the program comprehensible to the computer, but it's worthwhile to spend a little time here at the beginning to talk about making the program comprehensible to humans.

Python is pretty good in terms of comprehensibility. It's a language that doesn't require a lot of obscure punctuation or symbols that mean different things in different contexts. But there are still two very important things to keep in mind when you are writing any computer code:

(1) The next person to read the code will not know what you were thinking when you write the code.

(2) If you are the next person to read the code, rule #1 will still apply.

Because of this, it is absolutely critical to comment your code. Comments are bits of text in the program that the computer ignores. They are there solely for the benefit of any human readers.

Here's an example Python program:

```
#!/usr/bin/env python
"""
```

```
tenPrimes.py

    Here's a simple Python program to print the first 10
    prime numbers. It uses the function IsPrime(), which
    doesn't exist yet, so don't take the program too
    seriously until you write that function.
"""

# Initialize the prime counter
count = 0

# "number" is used for the number we're testing
# Start with 2, since it's the first prime.
number = 2

# Main loop to test each number
while count < 10:
    if IsPrime(number):      # The function IsPrime() should return
                             # a true/false value, depending on
                             # whether number is prime. This
                             # function is not built in, so we'd
                             # have to write it elsewhere.

        print number         # The number is prime, so print it.
        count = count + 1    # Add one to our count of primes so far.
    number = number + 1      # Add one to our number so we can check
                             # the next integer.
```

Anything that follows # is a comment. The computer ignores the comments, but they make the program easier for humans to read.

There is a second type of comment in that program also. Near the beginning there is a block of text delimited by three double-quotes: """. This is a multi-line string, which we'll talk more about later. The string doesn't *do* anything in this case, and isn't used for anything by the rest of the program, so Python promptly forgets it and it serves the same purpose as a comment. This specific type of comment is used by the `pydoc` program as documentation, so if you were to type the command `pydoc tenPrimes.py` the response would consist of that block of text. It is good practice to include such a comment at the beginning of each Python program. This comment should include a brief description of the program, instructions on how to use it, and the author & date.

There is one special comment at the beginning of the program:

#!/usr/bin/env python. This line is specific to Unix machines[2]. When the characters *#!* (called "hash-bang") appear as the first two characters in a file, Unix systems take what follows as an indicator of what the file is supposed to be. In this case, the file is supposed to be used by whatever the program `/usr/bin/env` considers to be the `python` environment.

Compare the program above with the following functionally identical program:

```
count = 0
number = 2
while count < 10:
    if IsPrime(number):
        print number
        count = count + 1
    number = number + 1
```

The second program might take less disk space but disk space is cheap and plentiful. Use the commented version.

1.2 Simple Input & Output

The raw_input() command takes user keystrokes and assigns them, as a raw string of characters, to a variable. The input() command does nearly the same, the only difference being that it first tries to make numeric sense of the characters. Either command can give a prompt string, if desired.

Example 1.2.1

```
name = raw_input("what is your name? ")
```

After the above line, the variable 'name' will contain the characters you type, whether they be "King Arthur of Britain" or "3.141592".

```
y = input("What is your quest? ")
```

The value of y, after you press enter, will be the computer's best guess as to the numeric value of your entry. "3.141592" would result in y being approximately π. "To find the Holy Grail" would cause an error.

[2]Including Macintosh & Linux, see appendix A

In order to get your carefully calculated results out of the computer and onto the monitor, you need the **print** command. This command sends the value of its arguments to the screen.

Example 1.2.2

```
e = 2.71828
print "Hello, world"
```

⟶ Hello world

```
print e
```

⟶ 2.71828

```
print "Euler's number is approximately ", e, "."
```

⟶ Euler's number is approximately 2.71828 .

Note in example 1.2.2 that the comma can be used to concatenate outputs. The comma can also be used to suppress the newline character that would otherwise come automatically at the end of the output. This use of the comma can allow you to make one print statement ending in a comma, then another print statement some lines further in the program, and have the output of both statements appear on one line of the screen.

It is also possible to specify the format of the output, using "string formatting". The most common format indicators used for our purposes are given in table 1.1. To use these format indicators, include them in an output string and then add a percent sign and the desired value to insert at the end of the print statement.

Example 1.2.3

```
pi = 3.141592
print "Decimal: %d" % pi
```

⟶ 3

```
print "Floating Point, two decimal places: %0.2f" % pi
```

⟶ 3.14

```
print "Scientific, two D.P, total width 10: %10.2e" % pi
```

%xd	Decimal (integer) value, with (optional) total width x.
%x.yf	Floating Point value, x wide with y decimal places.Note that the output will contain more than x characters if necessary to show y decimal places plus the decimal point.
%x.ye	Scientific notation, x wide with y decimal places.
%x.5g	"General" notation: switches between floating point and scientific as appropriate.
%xs	String of characters, with (optional) total width x.
+	A "+" character immediately after the % sign will force indication of the sign of the number, even if it is positive. Negative numbers will be indicated, regardless.

Table 1.1: Common string formatting indicators

$\longrightarrow$ 3.14e00

Note the two extra spaces at the front of the output in that final example. If we had given the format as "%2.2e", the output would have been the same numerically, but without those two blank spaces at the beginning. The output format expands as necessary, but always takes up *at least* as much space as specified.

Should you need to include more than one formatted variable in your output, go right ahead: just put the variables, grouped with parenthesis, after the % sign. Put them in the right order, of course: the first value will go into the first string formatting code, the second into the second, and so on.

Example 1.2.4

```
pi = 3.141592
e = 2.718282
sum = pi + e
print "The sum of %0.3f and %0.3f is %0.3f." % (pi, e, sum)
```

$\longrightarrow$ The sum of 3.142 and 2.718 is 5.860.

String formatting can be used for *any* strings, not just print statements. You can use string formatting to build up strings you want to use later, or send to a file, or whatever:

```
ComplicatedString = "Student %s scored %d on the final exam,
    for a grade of %s." % (name, FinalExamScore, FinalGrade)
```

1.3 Variables

It's worth our time to spend a bit of time discussing how Python handles variables. When Python interprets a line such as x=5, it starts from the right hand side and works its way towards the left. So given the statement x=5, the Python interpreter takes the "5", recognizes it as an integer, and stashes it in an integer-sized "box" in memory. It then takes the label x and uses it as a pointer to that memory location. (See figure 1.0.)

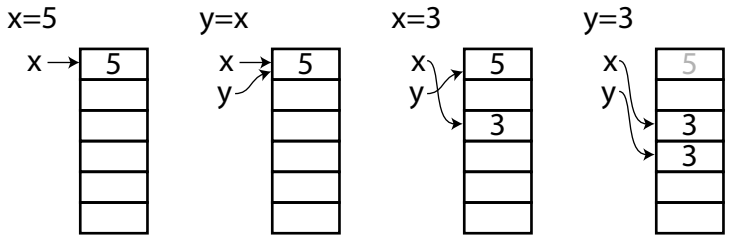

Figure 1.0: Variable assignment statements, and how Python handles them. The boxes represent locations in the computer's memory.

The statement y=x is analyzed the same way. Python starts from the right (x) and, recognizing x as a pointer to a memory location, makes y a pointer to the same memory location. At this point, both x and y are pointing to the same location in memory, and you could change the value of both *if you could change the contents of that location.* You can't, generally...

Continuing the example shown in figure 1.0, you now give Python the statement x=3. As always, this is analyzed from right to left: "3" is placed in some memory location, and x becomes a pointer to that location. This doesn't change y, though, since y is a pointer to the location containing "5".

Finally, if you now give Python the command y=3, y will end up pointing at some third memory location containing yet another "3". It will not be the same memory location as x, since from Python's perspective there is no reason it *should* be the same location. At this point, nothing is pointing at the memory location containing "5", and that location is free to be used for something else.

This right-to-left interpretation allows you to do some very useful —if mathematically improbable— things. For example, the command x = x+1 is perfectly legal in Python (as well as in nearly every other computer language.) If $x = 3$, as in the end of figure 1.0, Python would start from the right $(x + 1)$ and calculate that to be "4". It would then assign x to be a pointer to that "4". It is also perfectly legal in Python to say w = x = y = z = "Dead Parrot". In this case, each of those variables would end up pointing at the exact same spot in memory, until they were used for something else.

Python also allows you to assign more than one variable at a time. The statement a,b = 3,5 works because Python analyzes the right half first and sees it as a pair of numbers, then assigns that pair to the pair of variables on the left.[3] This can be used in some very handy ways.

Example 1.3.1
You want to swap two variable values.

```
x,y = y,x
```

Example 1.3.2
If you start with $a = b = 1$, what would be the result of repeated uses of this command?

```
a,b = b, a+b
```

Generally, a deep knowledge of how Python manages variables like this is not necessary. But there are occasions when changing a variable's value changes the contents of that box in memory rather than changing the address pointed to by the variable. Keep this in mind when you're dealing with matrices. It's something to be aware of!

Variable Names

Variable names can contain letters, numbers, and the underscore character. They must start with a letter. Names are case sensitive, so "Time" is not the same as "time".

[3] Technically, it sees both pairs as "tuples". See page 22.

It's good practice when naming variables to choose your names so that the code is "self-commenting". The variable names r and R are legal, and someone reading your computer code might guess that they refer to radii; but the names CylinderRadius and SphereRadius are much better. The extra time you spend typing those more descriptive variable names will be *more* than made up by the time you save debugging your code!

Variable Types

There are many different types of variables in Python. The two broad divisions in these types are *numeric* types and *sequence* types. Numeric types hold single numbers, such as "42", "3.1415", and $2 - 3i$. Sequence types hold multiple objects, which may be single numbers, or individual characters, or even collections of different types of things.

One of the strengths (and pitfalls) of Python is that it automatically converts between types as necessary, if possible.

Numeric Types

Integer The integer is the simplest numeric type in Python. Integers are perfect for counting items, or keeping track of how often you've done something.

The maximum integer is $2^{31} - 1 = 2,147,483,647$.

Integers don't divide quite like you'd expect, though! In Python, $1/2 = 0$, because 2 goes into 1 zero times.

Long Integer Integers larger than 2,147,483,647 are stored, automatically, as long integers. These are indicated by a trailing "L" when you print them, unless you use string formatting to remove it.

The maximum size of a long integer is limited only by the memory in your computer. Integers will automatically convert to long integers if necessary.

Float The "floating point" type is a number containing a decimal point. 2.718, 3.14159, and 6.626×10^{-34} are all floating point numbers. So is 3.0. Floats require more memory to store than do integers, and they are generally slower in calculations, but at least $1.0/2.0 = 0.5$ as one would expect.

It's important to note that Python will "upconvert" types if necessary. So, for example, if you tell Python to calculate the value of 1.0/2,

Python will convert the integer 2 to the float 2.0 and then do the math.There is a trade-off in speed for this convenience, though.

Complex Complex numbers are built-in in Python, which uses $j \equiv \sqrt{-1}$. It is perfectly legal to say x = 0.5 + 1.2j in Python, and it does complex arithmetic correctly.

Sequence Types

Sequence types in Python are collections of items which are referred to by one variable name. Individual items within the sequence are separated by commas, and referred to by an *index* in square brackets after the sequence name. This is easier to demonstrate than explain, so...

Example 1.3.3

```
Pythons = ("Cleese", "Palin", "Idle", "Chapman",
    "Jones", "Gilliam")
print Pythons[2]
```

⟶ Idle

Note that the index starts counting from zero:

```
print Pythons[0]
```

⟶ Cleese

Negative numbers start counting from the end, backwards:

```
print Pythons[-1], Pythons[-2]
```

⟶ Gilliam Jones

One can also specify a "slice" of the sequence:

```
print Pythons[1:3]
```

⟶ ('Palin', 'Idle')

Note that in that last example, what is printed is another (shorter) sequence. Note also that the range [1:3] tells Python to start with item 1 and go *up to* item 3. Item 3 is not included.

Now let's examine some of the specific types of sequence in Python.

Tuple Tuples are indicated by parentheses: (). Items in tuples can be any other data type, including other tuples. Tuples are *immutable*, meaning that once defined their contents cannot change.

List Lists are indicated by square brackets: [ ]. Lists are pretty much the same as tuples, but they are *mutable*: individual items in a list may be changed. Lists can contain any other data type, including other lists.

String A string is a sequence of characters. Strings are delimited by either single or double quotes: " " or ' '. Strings are immutable, like tuples. Unlike lists or tuples, strings can only include characters.

There are also some special characters in strings. To indicate a <tab> character, use "\t". For a newline character, use "\n".

The # character indicates a comment in Python, so if you put # in a string the rest of the string will be ignored by the Python interpreter. The way to get around this is to "escape" the character with "\", such as "\#". This causes Python to recognize that the # is meant as just a # character, rather than the *meaning* of #. Similarly, \" will allow you to put a double-quote inside a double-quoted string, and \' will allow use of a single-quote inside a single-quoted string.

An alternate way of indicating strings is to bracket them in triple double-quotes. This allows you to have a string that spans multiple lines, including tabs and other special chacters. A triple double-quoted string within a Python program which is not assigned to a variable or otherwise used by Python will be taken to be documentation by the pydoc program.

Dictionary Dictionaries are indicated by curly brackets: { }. They are different from the other built-in sequence types in Python in that instead of numeric indices they use "keys", which are string labels. Dictionaries allow some very powerful coding, but we won't be using them in this course so you'll have to learn them elsewhere.

As mentioned in the description of lists above, a list can contain other lists. A list of lists sounds almost like a 2-dimensional array, or matrix. You can use them as such, and the way you would refer to elements in the matrix is to tack indices onto the end of the previous index.

Example 1.3.4

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

The variable "matrix" is now a 3×3 array. To change the first item in the second row from 4 to 0, we would use

```
matrix[1][0] = 0
```

(Remember that the indices start from zero!)

This is almost what we want for matrices in computational physics. Almost. The [i][j] method of indexing is somewhat clumsy, for starters: it'd be nice to use [i,j] notation like we do in everything else. Another problem with using lists of lists for matrices is that addition doesn't work like we'd expect. When lists or tuples or strings are added, Python just sticks them together end-to-end, which is mathematically useless.

Example 1.3.5

```
matrix1 = [ [1, 2], [3, 4] ]
matrix2 = [ [0, 0], [1, 1] ]
print matrix1 + matrix2
```

$\longrightarrow$ [[1, 2], [3, 4] [0, 0], [1, 1]]

The best way of doing matrices in Python is to use the SciPy or NumPy packages, which we will introduce later.

Sequence Tricks

If you are calculating a list of N numbers, it's often handy to have the list exist first, and then fill it with numbers as you do the calculation. One easy way to create an empty list with the length needed is multiplication:

```
LongList = []*N
```

After the above command, LongList will be a list of N blank elements, which you can refer to as you figure out what those elements should be.

Sometimes you may not know exactly how many list elements you need until you've done the calculation, though. Being able to add elements to the end of a list, thus making the list longer, would be ideal in this case; and Python provides for this with list.append(). Here's an example:

Example 1.3.6

You want a list of calculated values, but you don't know exactly how many you need ahead of time.

```
# Start by creating the first list element.
# Even an empty element will do -- there just must
# be something to tell Python that the variable
# is a list rather than something else.
Values = []
# Now do your calculations,
NewValue = MuchCalculation(YadaYadaYada)
# and each time you find another value just append
# it to the list:
Values.append(NewValue)
# This will increase the length of Values[] by one,
# and the new element at the end of Values[] will
# be NewValue.
```

Another handy trick is sorting. If, for example, in the previous example you wanted to sort your list of values numerically after you'd calculated them all, this would do it:

```
Values.sort()
```

After that, the list Values would contain the same information but in numeric order.

Generally speaking, the sort operation only works if all the elements of the list are the same type. Trying to sort a mix of numbers and strings and other sequences is a recipe for disaster, and Python will just give you an error and stop.

Sequences have many other useful built-in operations, more than can be covered here. Google is my preferred way of finding what they are: If there's something you think you *should* be able to do with a sequence, Google "Python list (or string, or tuple) <action>", whatever that action might be. This will usually come up with something! There are operations for finding substrings in strings, changing case in strings, removing non-printing characters, etc...

Ranges

It is often necessary to create a list of numbers for the computer to use. If you're making a graph, for example, it'd be nice to quickly generate a list of numbers to put along the bottom axis. Python has a built-in function to do this for us: "range()". The range() function takes up to three parameters: *Start* (optional), *Stop* (required), and *Step* (optional). It generates a list of

integers beginning with *Start* (or zero if *Start* is omitted) and ending just before *Stop*, incrementing by *Step* along the way.

Example 1.3.7
Create a list of 100 numbers for a graph axis.

```
axis = range(100)
print axis
```

⟶ [0, 1, 2, ... 98, 99]

Create a list of even numbers from 6 up to 17.

```
Evens = range(6,17,2)
```

List Comprehensions

The range() function only creates integers, and the spacing between the integers is always exactly the same. We often need something more flexible than that: what if we needed a list of squares of the first 100 integers, or a range that went up by 0.1 each step?

Python provides one very useful trick for doing just this thing: List Comprehensions. Again, this is most easily shown by example:

Example 1.3.8
Create a list of 100 evenly-spaced numbers for a graph axis which goes from a minimum of zero to a maximum of 2.

```
Axis = [0.02 * i for i in range(100)]
```

Let's look at that bit by bit. The square brackets indicate that the result is a list. The formuala (0.02 * i) is how it calculates each item in the list. The "for i in range(100)" tells Python to take each number in the list "range(100)", call that number "i", and apply the formula given.

List comprehensions work for *any* list, not just range(). So if you have a list of experimental measurements that were taken in inches, and you needed to convert them all to centimeters, then the line

```
metric = [2.54*measure for measure in ListOfMeasurements]
```

would do what you need.

1.4 Mathematical Operators

In Python, the mathematical operators $+ - * ()$ all work as one would expect on numbers and numeric variables. As mentioned previously, the + operator doesn't do matrix addition on lists — it just strings them together instead. The * operator (multiplication) also does something unexpected on lists: if you multiply a list by n, the result will be n copies of the list, strung together. As long as you stick with numeric types, though, addition, subtraction, and multiplication do what you want.

Division (/) is slightly different. It works perfectly on floats, but on integers it does "third-grade math": the result is always the integer portion of the actual answer.

Example 1.4.1

```
print 10/4
```

$\longrightarrow 2$

```
print 1/2
```

$\longrightarrow 0$

```
print 1./2.
```

$\longrightarrow 0.5$

That second case in example 1.4.1 is particularly bothersome: it causes more program bugs than you'd expect, so keep an eye out for it.

If you *want* "third-grade math" with floats, use the "floor division" operator, //.

Exponentiation in Python is done with the ** operator, as in

```
Sixteen = 2**4
```

The modulo operator (AKA "remainder") is %, so $10\%4 = 2$ and $11\%4 = 3$.

The precedence rules in Python are exactly what they should be in any mathematics system: (); then **; then *, /, %, // in left-to-right order; and finally +, − in left-to-right order. Although those are the rules, and they work, the *best* way to do things is to use the "cheap rule": "* before +, otherwise use ()."

Shortcut Operators

The statement x = x + y and other similar statements are so common in programming that many languages (including Python) allow shortcut operators for these statements. The most common of these is +=, which means "Take what's to the right and add to it whatever is on the left". In other words, x += 1 is exactly equivalent to x = x + 1. Similarly, − =, ∗ =, and / = do the same thing for subtraction, multiplication, and division.

These shortcut operators do not make your program run faster, and they do make the code harder for humans to read. Use them sparingly.

1.5 Lines in Python

Python is somewhat unique among programming languages in that it is whitespace-delimited. In other words, the Python interpreter actually cares about blank space before commands on a line.[4] Lines are actually grouped by how much whitespace preceeds them, which forces one to write well-indented code!

When one speaks of "lines" of Python code, there are actually two types of line. A *physical* line is a line that takes up one line on the editor screen. A *logical* line is more important — it's what the Python interpreter regards as one line. Let's look at some examples:

Example 1.5.1

```
print "this line is a physical line and a logical line."

x = ["this", "line", "is", "both", "also"]

x = ["this", "line", "is", "multiple",
     "physical", "lines", "but", "is",
     "just", "one", "logical", "line"]
```

Indentation like this helps make programs clear and easier for humans to read. Python ignores extra whitespace *inside* a logical line, so it's not a problem to put it there.

---

[4] Whether this is one of the best or worst features of Python is a matter of some debate.

1.6 Control Structures

Control statements are statements that allow a program to do different things depending on what happens. "If you are hungry, eat lunch." is a control statement of sorts. "While you are in Hawaii, enjoy the beach." is another. Of course control statements in a computer language are a bit more specific than that, but they have the same basic structure. There is the statement itself: "if". There is the "conditional", which is a statement that evaluates to either true or false: "you are hungry". And there is the action: "eat lunch".

Conditionals

A conditional is anything that can be evaluated as either true or false. In Python, the following things are always false:

- The word False.
- 0, 0L, or 0.0
- "" or '' (an empty string)
- (), [], {} (The empty tuple, list, or dictionary)

Just about everything else is true.

- 1, 3.14, 42 (True, because they are numbers that aren't zero)
- The word True.
- "False" (This is true, because it's a string that is not empty. Python doesn't look inside the string to see what it's about!)
- "0", for the same reason as "false".
- [0, False, (), ""] (This is true, even though it's a list of false things, because it is a non-empty list, and if a list is not empty then it is true.)

The comparisons are

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

== Equal to

!= Not equal to

Note that "=" is an assignment, and "==" is a conditional. FavoriteColor = ''Blue'' assigns the string "Blue" to the variable FavoriteColor, but FavoriteColor == ''Blue'' is a conditional that evaluates to either true or false depending on the contents of the variable FavoriteColor. *Using = instead of == is one of the most common and hard-to-find bugs in Python programs!*

There are also the boolean operators **and**, **or**, and **not**.

Example 1.6.1

```
if (Animal == ''Parrot'') and (not IsAlive(Animal)):
    Complain()
```

The precedence of **and**, **or**, and **not** is the lowest of anything in Python, so the parentheses are not actually necessary. Those parentheses make the code more readable, though, and as such are highly recommended.

One final boolean is the **in** command, which is used to test whether an item is in a list.

```
Cast = ('John', 'Eric', 'Terry', 'Graham', 'Terry', 'Michael')
if Name in Cast:
    print 'Yes', Name, "is a member of Monty Python's Flying Circus"
```

If . . . Elif . . . Else

The most basic control statement in Python, or any other computer language, is the **if** statement. It allows you to tell the computer what to do if some condition is met. The syntax is as follows:

```
if Conditional: # The : is needed at the end of this line.
    DoThis()    # This line must be indented.
    AndThis()   # Any other lines that are part of the if
                # statement must also be indented the same
                # amount. (comments are ignored, of course!)
    ThisAlso()
AlwaysDoThis()  # This line is not indented, so it is not
                # part of the if statement's action.
```

Note the indenting. In other computer languages, indentation like this makes the code easier to read; but in Python *the indentation is what defines the group of commands.* The indentation is *not* optional.

The **elif** and **else** keywords extend the **if** statement even further. **elif**, short for "else if" adds another if that is tested only if the first if is false. **else** is done if none of the previous **elif** statements, or the initial **if**, are true.

```
If TestOne:
    Do_A()          # Do_A() is done only if TestOne is true.
elif TestTwo:       # TestTwo is tested only if TestOne is false.
    Do_B()          # Do_B() is done if TestOne is false and
                    # TestTwo is true.
elif TestThree:
    Do_C()          # You can have as many elif's as you want,
                    # or none at all. They're optional.
else:
    Do_D()          # The 'else' is what happens if nothing else is true.

AlwaysDoThis()      # This statement is back at the left margin again.
                    # That means it's after the end of the whole if
                    # construct, and it is done regardless.
```

While

The **while** statement is used to repeat a block of commands until a condition is met.

```
while Condition:        # The : is required, again.
    DoThis()            # The block of things that should be done
    DoThat()            # is indented, as always.
    DoTheOther()
    UpdateCondition()
                        # If nothing happens to change the condition,
                        # the while loop will go forever!
DoThisAfterwards()      # This statement is done after Condition
                        # becomes false.
```

There are a few extra keywords that can be used with the **while** loop.

pass The **pass** keyword does exactly nothing. Its sole purpose is to create an indented line if you want the program to do nothing but there's a structural need for an indented line. I know this seems crazy, but I have actually found a situation in which the **pass** command was the simplest way to make something work!

continue The **continue** keyword moves program execution to the top of the while block without finishing the portion of the block following the **continue** statement.

break The **break** keyword moves execution of the loop directly to the line following the **while** block. It "breaks out" of the loop, in other words.

else An **else** command at the end of a **while** block is used to delineate a block of code that is done after the **while** block executes *normally*. The code in this block is *not* executed if the **while** block is exited via a **break** command.

Confusing enough for you? This is probably a good time for an example that uses these features.

Example 1.6.2
You need to write a program that tests whether a number is prime or not. The program should ask for the integer to test, then print a message giving either the first factor found or stating that the number is prime.

```
#!/usr/bin/env python
"""
    This Python program determines whether a number
    is prime or not. It is NOT the most efficient
    way of determining whether a large integer is
    prime!
"""

# Start by getting the number that might be prime.
Number = input("What integer do you want to check? ")

TestNumber = 2
# Main loop to test each number
while TestNumber < Number:
    if Number % TestNumber == 0:
        # The remainder is zero, so TestNumber is a factor
        # of Number and Number is not prime.
        print Number, "is divisible by", TestNumber, "."
        # There is no need to test further factors.
        break
    else:
        # The remainder is NOT zero, so increment
        # TestNumber to check the next possible factor.
```

```
            TestNumber += 1
    else:
        # We got here without finding a factor, so
        # Number is prime.
        print Number, "is prime."
```

For

The **for** loop iterates over items in a sequence, repeating the loop block once per item. The most basic syntax is as follows:

```
for Item in Sequence:    # The : is required
    DoThis()             # The block of commands that should
    DoThat()             # be done repeatedly is indented.
```

Each time through the loop, the value of Item will be the value of the next element in the Sequence. There is nothing special about the names Item and Sequence, they can be whatever variable names you want to use. In the case of Sequence, you can even use something that *generates* a sequence rather than a sequence, such as range().

Example 1.6.3
You need a program to greet the cast of a humorous skit.

```
Cast = ('John', 'Eric', 'Terry', 'Graham', 'Terry', 'Michael')
for Member in Cast:
    print 'Hello', member            # Each time through the loop
                                     # one cast member is greeted.
print 'Thank you for coming today!' # This line is outside the loop
                                     # so it is done once, after
                                     # the loop.
```

The output of this program will be

```
Hello John
Hello Eric
Hello Terry
Hello Graham
Hello Terry
Hello Michael
Thank you for coming today!
```

In numeric work, it's more common to use the **for** command over a numeric range:

```
for j in range(N):
    etc()
```

Since the range() function returns a list, the **for** command is perfectly happy with that arrangement.

The **for** command also allows the same extras as **while**: **continue** goes straight to the next iteration of the **for** loop, **break** causes Python to abandon the loop, **else** at the end of the loop marks code that is done *only* if the **for** loop exits normally, and **pass** does nothing at all.

One caution about **for** loops: it's quite possible to change the list that is controlling the loop during the loop! This is not recommended, as the results may be unpredictable.

1.7 Functions

A function is a bit of code that is given its own name so that it may be used repeatedly by various parts of a program. A function might be a bit of mathematical calculation, such as sin() or sqrt(). It might also be code to do something, such as draw a graph or save a list of numbers.

Functions are defined with the **def** command. The function name must start with a leter, and may contain letters, numbers, and the underscore character just like any other Python variable name. In parentheses after the function name should be a list of variables that should be passed to the function. The **def** line should end with a colon, and the indented block after the **def** should contain the function code.

Generally, a function should **return** some value, although this is not required. For mathematical functions, the return value should be the result of the calculation. For functions that don't calculate a mathematical value, the return should be True or False depending on whether the function managed to do what it was supposed to do or not.

Example 1.7.1
Write a function that calculates the factoral of a positive integer.

```
def factoral(n):
    """ factoral(n)
        This function calculates n! by the simplest and
```

```
        most direct method imaginable.
    """
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Once this definition has been made, any time you need to know the value of a factorial, you can just use factoral(x).

```
print '%10s %10s' % ('n', 'n!')
for j in range(10):
    print '%10d %10d' % (j, factoral(j))
```

Functions can also be used to break the code up into more understandable chunks. Your morning ritual might look something like this, in Python code:

```
if (Time >= Morning):
    GetUp()
    GetDressed()
    EatBreakfast(Spam, eggs, Spam, Spam, Spam, Spam, bacon)
else:
    ContinueSleeping()
```

The functions GetDressed() and EatBreakfast() may entail quite a bit of code; but writing them as separate functions allows one to bury the details (socks first, then shoes) elsewhere in the program so as to make this code more readable. Writing the program as a set of functions also allows you to change the program easily, if for example you needed to eat breakfast before getting dressed.

The variables that are passed to the function exist for the duration of that function only. They may (or may not) have the same name(s) as other variables elsewhere in the program.

Example 1.7.2

```
def sq(x):
    # Returns the square of a number x
    x = x*x # Note that sq() changes the local value of x here!
    return x

# Here's the main program:
x = 3
```

```
print sq(x) # prints 9,
print x     # prints 3.
```

Note that the value of x is changed within the function sq(), but that change doesn't "stick". The reason for this is that sq() does not receive x, but instead receives some value which it then calls x for the duration of the function. The x within the function is not the same as the x in the main program.

Functions can have default values built-in, which is often handy. This is done by putting the value directly into the definition line, like this:

```
def answer(A = 42):
    # Put your function here
    # etc.

# main program
answer(6)    # For this call to answer(), A will be 6.
answer()     # But this time, A will be the default, 42.
```

Global variables

If a Python function can't find the value of some variable, it looks outside the function. This is handy: you can define π once at the beginning of the program and then use it inside any functions in the program. Values used throughout the program like this are called global variables.

If you re-define the value of a variable inside your function, though, then that new value is valid only within the function. To change the value of a global variable and make it stick outside the function, refer to that variable in the function as a **global**. The following example may help clarify this.

Example 1.7.3

```
a = 4
b = 5
c = 6

def fn(a):
    d = a           # d is a new local variable, and has the
                    # value of whatever was passed to fn().
    a = b           # 'fn' does not know 'b', so Python looks
                    # outside fn and finds b=5. So the local
```

```
                    # value of 'a' is 5 now.
        global c = 9 # Instead of making a new local 'c', this
                    # line changes the value of the global
                    # variable 'c'.

    print a,b,c     # --> 4 5 6
    fn(b)           # The value of d inside fn() will be 5.
                    # The value of a inside fn() will also be 5.
    print a,b,c     # --> 4 5 9
                    # The values outside fn() didn't change,
                    # other than c.
    print d         # --> ERROR! d is only defined inside fn().
```

Passing functions

Python treats functions just like any other variable. This means that you can store functions in other variables or sequences, and even pass those functions to other functions. The following program is somewhat contrived, but it serves to give a good example of how this can be useful. The output is shown in figure 1.1.

```
#!/usr/bin/env python
''' passtrig.py
    Demonstrates Python's ability to store functions an variables and
    pass those functions to other functions.
'''

from pylab import *

def plottrig(f):
    # this function takes one argument, which must be a function
    # to be plotted on the range -pi..pi.
    xvalues = linspace(-pi, pi, 100)
    plot(xvalues, f(xvalues))   # y values are computed depending on f.
    xlim(-pi, pi)               # set x limits to x range
    ylim(-2, 2)                 # set the y limits so that tan(x) doesn't
                                # ruin the vertical scale.

trigfunctions = (sin, cos, tan) # trigfunctions is now a tuple that holds
                                # these three functions.

for function in trigfunctions:
    # trigfunctions is a list of functions, so this for loop does things
```

```
    # with each function in the list.
    print function(pi/6.0)        # returns this function value.
    plottrig(function)            # passes this function to be plotted.

show()
```

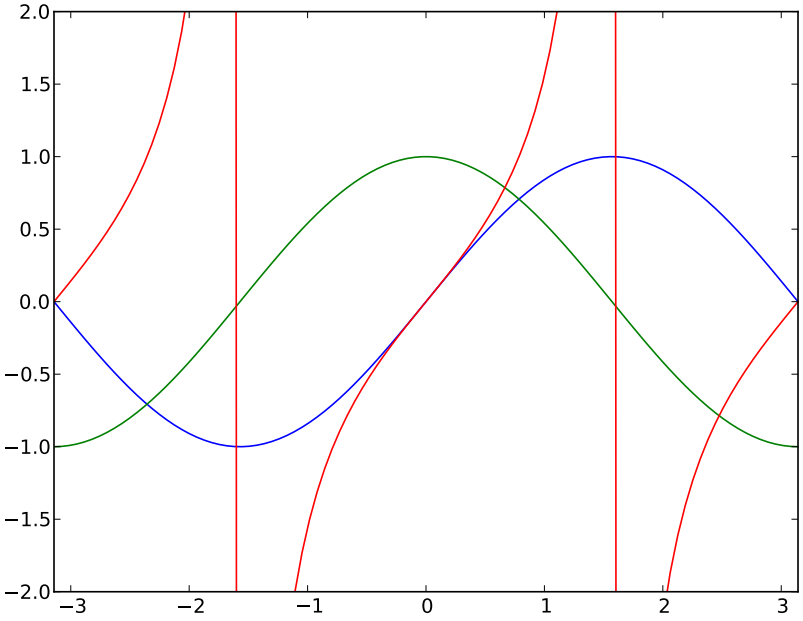

Figure 1.1: Output of function-passing program "passtrig.py"

As you can see, functions here are put into lists, referred to as elements in lists, and passed to other functions. This is particularly useful in the next chapter, where we will develop (among other things) ways of finding roots of equations. We can write functions that find roots, then pass functions to be solved *to* those root-finding functions. This allows us to use one general-purpose root-finding function for just about any function for which we need to find a root.

1.8 Files

More often than not in computational physics, the inputs and outputs of a project are large sets of data. Rather than re-enter these large data sets each time we run the program, we load and save the data in the form of text files.

When working with files, we start by "opening" the file. the open() function tells the computer operating system what file we will be working on, and what we want to do with the file. The function requires two parameters: the file name and the desired mode.

```
FileHandle = open(FileName, Mode)
```

FileName should be a string describing the location and name of the file, including any directory information if the file is not in the working directory for the Python program. The mode can be one of three things:

'r' Read mode allows you to read the file. You can't change it, only read it.

'w' Write mode will create the file if it does not exist. If the file *does* exist, opening it in 'w' mode will re-write it, destroying the current contents.

'a' Append mode allows you to write onto the end of a previously-existing file without destroying what was there already.

Once the file is open for reading, we can read it by one of several methods. We can read the entire text file into one string:

```
string = FileHandle.read()
```

The resulting string can be inconveniently huge, though, depending on the file! We could alternately read one line of the file at a time:

```
Line = FileHandle.readline()
```

Each time you invoke this command, the variable Line will become a string containing the next line of the file indicated by FileHandle. We could also read all of the lines at once into a list:

```
Lines = FileHandle.readlines()
```

After this command, Lines[0] will be a string containing the first line of the file, Lines[1] will contain the second line, and so on.

When we're done reading the file, it's best to close the file.

```
FileHandle.close()
```

If you don't close the file, it will close automatically when the program is done, but it's still good practice to close it yourself.

Notice that all of these methods of reading a file result in strings of characters. This is to be expected, since the file itself is a string of characters on the drive. Python makes no effort to try figuring out what those characters mean, so if you want to change the strings to numeric values you must convert them yourself by specifying what type of numbers you expect them to be. The float() and int() functions are the standard way of doing this. For example if the string S = '3.1415', then x=float(S) takes that string and changes it to the appropriate floating-point value.

If you wish to write to a file, you must indicate so when you open the file.

```
FileHandle = open(FileName, 'w')
```

After the file is open for writing, the write operation saves string information to the file:

```
FileHandle.write(string)
```

The write() operation is much more literal than the **print** command: it sends to the file *exactly* the contents of the string. If the string does not have a newline character at the end, then there will be no newline character written to the file, so if you want to write a single line be sure to include '\n' as the last character in the string. The write() operation can use any of the string formatting techniques described earlier.

```
FileHandle.write("pi = %6.4f\te=%6.4f\n" % (pi, e))
```

When you are done writing to the file, it is very important to close the file.

```
FileHandle.close()
```

Closing the file flushes the write buffer and ensures that all of the written matter is actually on the hard drive. The file will be closed automatically at the end of the program, but it's better to close it yourself.

1.9 Expanding Python

One of the nicest things about Python is how easy it is to add further functionality. In the mathematical department, for example, Python is somewhat limited. It does not have built-in trigonometric functions, or know the values of π and e. But this can be easily added into Python, as needed, using the **import** command.

```
import math
```

import commands are generally placed at the beginning of the program, although that's just for convenience to the reader. Once the **import** command has been run by the program, all of the functions in the "math" package are available as "math.(function-name)".

```
import math
x = math.pi / 3.0
print x, math.sin(x), math.cos(x), math.tan(x)
```

There are many other functions and constants in the math package.[5] It is often useful to just import individual elements of a package rather than the entire package, so that we could refer to "sin(x)" rather than math.sin(x).

```
from math import sin
```

You can also import everything from a package, each with its own name:

```
from math import *
x = pi / 3.0
print x, sin(x), cos(x), tan(x)
```

There are numerous other packages that add useful functionality to Python. In this book we'll be using the "scipy", "pylab", "matplotlib", and "numpy" packages extensively. These will be discussed later, but one package that deserves mention here is the "sys" package, which allows access to some of the system underpinnings of a unix-based system. The part of the sys package we use most often is the variable sys.argv. When you invoke a Python program from the command line, it is possible to enter parameters directly at time of invocation.

```
$ add.py 35 7
```

In the case shown above, the variable sys.argv will be a list containing the program name and whatever came after on the line:

```
sys.argv = ['add.py', '35', '7']
```

So we can use this to enter values directly into a program.

Example 1.9.1

```
import sys

x = float(sys.argv[1])   # Note that the items in argv[]
y = float(sys.argv[2])   # are strings!

print "%0.3f + %0.3f = %0.3f" % (x, y, (x+y))
```

[5] Give the command "pydoc math" in a terminal window for more information.

If this program were saved as "add.py", a user command of
`add.py 35 7`
would result in an output of
`35.000 + 7.000 = 42.000`

Note that the components of sys.argv are strings, so it's necessary to convert them to floating-point numbers with the float() function so that they add as numbers.

You can also build your own packages. This is easier than you might imagine, since *every* Python program is a package already! Any Python program ending with a '.py' suffix on its filename (and they all should end with .py) can be imported by any other Python program. Just use **from** <filename> **import** <function(s)>.

Example 1.9.2
You've written a program called "area.py" that includes an integration routine "integrate()", and you'd like to use that routine in your next program.

```
from area import integrate
```

It's a good idea, as you work through the exercises in this course, to put any useful functions you may develop into a "tools.py" file. Then all of those functions are easily accessible from new programs with the command

```
from tools import *
```

One thing to consider, though: any time you import a package, it overwrites anything with the same name. This can cause problems. The pylab package has trig functions, for example, that are capable of operating on entire arrays at once. The math package trig functions have much more basic capabilities. So if you use this code:

```
from pylab import *
from math import *
x = sin([0.1, 0.2, 0.3, 0.4, 0.5])
```

then the basic trig functions from math.py overwrite the pylab.py trig functions, and the sin() function will crash when you give it the array.

A better way to do it, if you *must* have both pylab and math packages, would be to import the packages with their "full names" intact.

```
import pylab
import math
x = pylab.sin([0.1, 0.2, 0.3, 0.4, 0.5])
y = math.sin(pi/2)
```

There's another thing to consider when dealing with Python packages: *any* Python program can be imported as a package into any other Python program. A Python program can even import itself, which will cause nothing but trouble. Problems arise if you happen to name your program "math.py" and them import math, for example. Python will happily import the first "math.py" it finds, which will not be the one you expect and nothing will work right and the error messages you get will be completely unhelpful.

1.10 Where to go from Here

This chapter has barely scratched the surface of the Python language, but it hopefully gives you enough of a foundation in Python that you'll be able to get started on solving some interesting physics problems.

There's a lot more to learn. One excellent resource for learning more Python is the book *Learning Python* by Mark Lutz and David Ascher.[6] Another is *Computational Physics* by Mark Newman.[9] The official Python website is at `http://www.python.org/`, and it always has up-to-date documentation about the latest version of Python. Google and other search engines are also excellent resources: there are numerous pages with helpful descriptions about how to do things in Python.

One other thing to be aware of: This chapter (this entire book, actually) is based on Python 2.7. Python 3.0 was released in November of 2008, and 3.0 is not entirely reverse-compatible with 2.7! Although 2.7 is not the latest-and-greatest, not all of the packages we use in the rest of this text are available for 3.x at this time. It is my hope that the matplotlib, visual, and numpy packages will be updated to work with 3.x soon, and until then 2.7 is the preferred version of Python for numerical work.

1.11 Problems

1-0 Use a list comprehension to create a list of squares of the numbers between 10 and 20, including the endpoints.

1-1 Write a Python program to print out the first N numbers in the Fibonacci sequence. The program should ask the user for N, and should require that N be greater than 2.

1-2 For the model used in introductory physics courses, a projectile thrown vertically at some initial velocity v_i has position $y(t) = y_i + v_i t - \frac{1}{2}gt^2$, where $g = 9.8$ m/s^2. Write a Python program that creates two lists, one containing time data (50 data points over 5 seconds) and the other containing the corresponding vertical position data for this projectile. The program should ask the user for the initial height y_i and initial velocity v_i, and should print a nicely-formatted table of the list values after it has calculated them.

1-3 The energy levels for a quantum particle in a three-dimensional rectangular box of dimensions $\{L_1, L_2, \text{and } L_3\}$ are given by

$$E_{n_1,n_2,n_3} = \frac{\hbar^2\pi^2}{2m}\left[\frac{n_1^2}{L_1^2} + \frac{n_2^2}{L_2^2} + \frac{n_3^2}{L_3^2}\right]$$

where the n's are integers greater than or equal to one. Write a program that will calculate, and list in order of increasing energy, the values of the n's for the 10 lowest *different* energy levels, given a box for which $L_2 = 2L_1$ and $L_3 = 4L_1$.

1-4 Write a function for

$$\text{sinc}(x) \equiv \frac{\sin x}{x}$$

Make sure that your function handles $x = 0$ correctly.

1-5 Write a function that calculates the value of the n^{th} triangular number. Triangular numbers are formed by adding a series of integers from 1 to n: i.e. $1 + 2 + 3 + 4 = 10$ so 10 is a triangular number, and 15 is the next triangular number after 10.

1-6 Write a function called `isprime()` that determines whether a number is prime or not, and returns either `True` or `False` accordingly. Redo Example 1.6.2 using this function. Try to make your program more efficient than the example, which is pretty dreadful in that regard!

1-7 Write a Python program to make an $N \times N$ multiplication table and write this table to a file. Each row in the table should be a single line of the file, and the numbers in the row should be tab-delimited. Write the program so that it accepts two parameters when invoked: the size N of the multiplication table and the filename of the desired output file. In other words, the command to make a 5×5 table saved in file table5.txt would be

```
timestable.py 5 table5.txt
```

1-8 Write a program that takes a multi-digit integer and prints out its digits in English. Bonus point if the program takes care of ones/ tens/hundreds/thousands properly.

Chapter 2

Basic Numerical Tools

2.0 Numeric Solution

A problem commonly given in first-semester physics lab is to calculate the launch angle for a projectle launcher so that the projectile hits a desired target. This is an easy enough problem, if the launch point and the target are at the same elevation: you just take the range equation

$$R = \frac{v^2 \sin(2\theta)}{g}$$

and solve for θ.

Complications arise, though, when the launcher is on the table and the target is on the floor. (See figure 2.0.)

To solve the problem in this case, we have to back up a bit and start with our basic kinematics equations. The horizontal distance traveled by the projectile in time t is

$$x = v_x t \tag{2.1}$$

where v_x is the horizontal component of the initial velocity, $v_x = v_o \cos\theta$. The time t at which the projectile hits the ground (and the target) is found by solving

$$0 = y_o + v_{iy} t - \frac{1}{2} g t^2 \tag{2.2}$$

where $v_{iy} = v_o \sin\theta$ and $g = 9.8$ m/s^2. The solution to equation 2.2 is given by the quadratic equation, so

$$t = \frac{1}{-g}\left[-v_o \sin\theta \pm \sqrt{v_o^2 \sin^2\theta + 2 y_o g}\right]$$

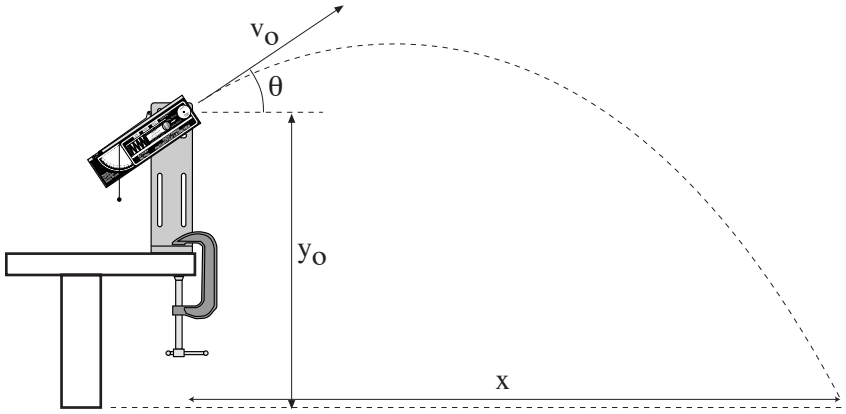

Figure 2.0: Introductory physics laboratory problem

and we want the positive answer, so

$$t = \frac{1}{g}\left[v_o \sin\theta + \sqrt{v_o^2 \sin^2\theta + 2y_o g}\right] \tag{2.3}$$

Plugging equation 2.3 into equation 2.1 gives us our general range equation:

$$x(\theta) = \frac{v_o \cos\theta}{g}\left[v_o \sin\theta + \sqrt{v_o^2 \sin^2\theta + 2y_o g}\right] \tag{2.4}$$

Now we have a problem. It's easy enough to calculate the range x, given θ, but the problem is to find θ to reach a given range x. It is possible to find an algebraic expression for θ from equation 2.4, but it's not easy, or pretty.

One way of solving equation 2.4 is to graph the right-hand side versus t and see where that graph reaches the desired value of x. Or more generally, graph the right-hand side minus the left-hand side, and see where the result crosses the θ axis. (See figure 2.1.) We're going to introduce three methods of solving equations numerically. You may not think of graphing as the most precise method of finding numeric solutions, but it is helpful to keep the graphical model in mind when considering how these numeric methods actually work.

One other factor to keep in mind is that these methods will give you an *approximately* correct answer in all but some special cases. The level of approximation depends on how much computer time you're willing to

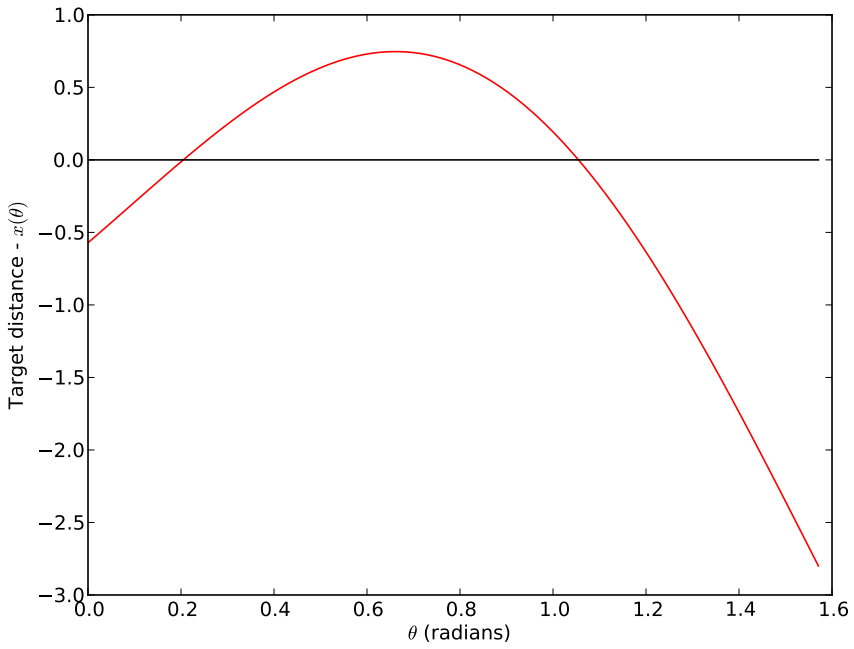

Figure 2.1: Graphical solution for θ, given parameters typical of introductory physics lab.

devote to the problem. You, as the computer user, must decide what level of approximation is good enough.

Bisection Method

To use the bisection method of finding a root, start with two guesses on either side of the root. They don't have to be exact guesses, or even particularly good guesses, but they have to be on either side of the root. We'll call the guess to the left of the root a and the guess to the right of the root b, as shown in figure 2.2.

Next, find the value of the function at the midpoint x between a and b. Compare the signs of $f(x)$ and $f(a)$: if the signs are different, then the root must be between a and x, so let $b = x$. If the signs are the same, then the root must be between x and b, so let $a = x$. Now a and b are such that the solution is still between the two, but they're half as far apart! Repeat this

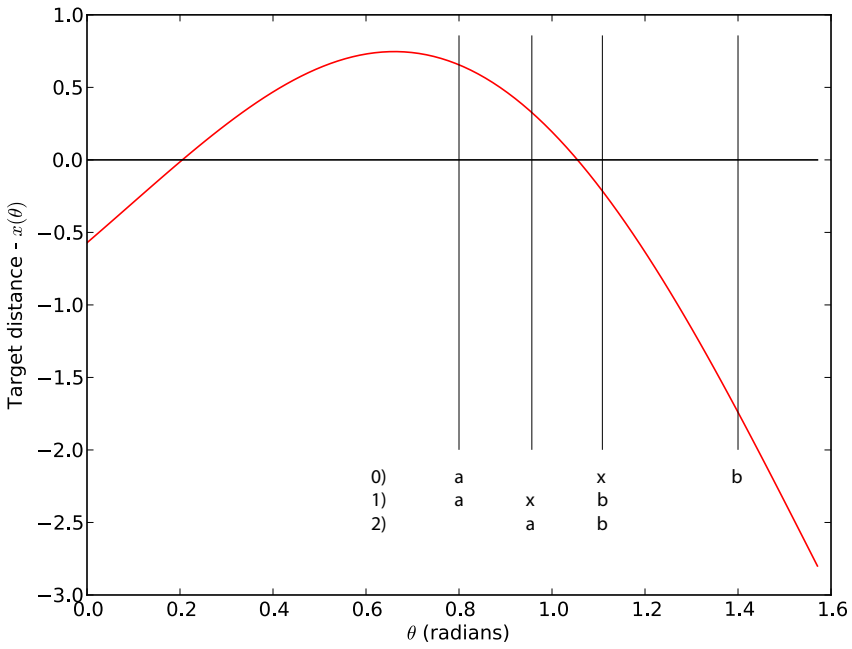

Figure 2.2: Bisection method of finding a root. At step 0, a and b are initial guesses, with a root between them. x is the initial halfway point: since the root is between a and b the new value of b is x. These new values of a and b are used in step 1, and so on.

process until the distance between a and b is less than the desired tolerance for your solution.

Here's a program to solve $f(x) = 0$, in Python code:

Example 2.0.1

```
#!/usr/bin/env python

"""
This program uses the bisection method to find the root
of f(x) = exp(x)*ln(x) -x*x = 0.
"""

from math import *                    # math functions and constants
```

```
tolerance = 1.0e-6          # solution tolerance.

def f(x):                   # function definition: this is the function
                            # to which we are finding a (the) root.
    f = exp(x)*log(x) - x*x
    return f

# Get the initial guesses
a,b = input("Enter two guesses, separated by commas: ")

dx = abs(b-a)               # initial value of dx

while dx > tolerance:       # Repeat until dx < tolerance
    x = (a+b)/2.0
    if (f(a)*f(x)) < 0:     # root is in left half
        b = x
    else:                   # root is in right half
        a = x
    dx = abs(b-a)           # update uncertainty in root location

print 'Found f(x) = 0 at x = %.8f +/- %0.8f ' % (x, tolerance)
```

Note that in the bisection program above, the function could be *any* function that returns a numeric value. In the example it's a relatively simple numeric function, but it doesn't have to be a numeric function in the mathematical sense. You could just as well have a function that — for example — runs a complete simulation and returns the final result. If you can write a function that returns a continuous numeric result, you can use that function with the bisection method.

With this in mind, it would be very handy to have a bisection-method root-finding function that could be used in any general case. Something that you could call like this would be ideal:

```
answer = root_bisection(Equation, FirstGuess, SecondGuess)
```

Here's how to do just that:

Example 2.0.2

```
def root_bisection(f, a, b, tolerance=1.0e-6):
    """
    Uses the bisection method to find a value x between a and b
    for which f(x) = 0, to within the tolerance given.
```

```
    Default tolerance is 1.0e-6, if no tolerance is specified in
    the function call.
    """
    dx = abs(b-a)                   # initial value of dx
    while dx > tolerance:           # Repeat until dx < tolerance
        x = (a+b)/2.0
        if (f(a)*f(x)) < 0:         # root is in left half
            b = x
        else:                       # root is in right half
            a = x
        dx = abs(b-a)               # update uncertainty in root location

    return x                        # answer is x +/- Tolerance
```

You can then include this function in any Python program, and use it to find the root of any function you define. Just pass the *name* of that function to root_bisection().

Example 2.0.3
You want to find the value of θ for which $\cos\theta = 0$.

```
from math import *
from root_bisection import *

theta_0 = root_bisection(cos, 0, pi)
```

The bisection method is simple and very robust. If there is a root between the two initial guesses, bisection *will* find it. Bisection is not particularly fast, though. Each step improves the accuracy only by a factor of two, so if you start with a and b separated by something on the order of 1, and you have a desired tolerance on the order of 10^{-6}, it will take roughly 20 steps. There are faster methods for finding roots.

Newton's Method

Newton's method of rootfinding requires that one know both the function $f(x)$ and its derivative $f'(x) = \frac{df(x)}{dx}$.

Start with a guess a, and calculate the values of $f(a)$ and $f'(a)$. (see figure 2.3.) Use the point-slope form of a line to find the new point b at

which the slope from $f(a)$ intersects the line $y = 0$.

$$f(a) = f'(a)(a - b) + 0 \Longrightarrow b = a - \frac{f(a)}{f'(a)}$$

Repeat the process, starting from point b, and continue repeating until the change from one step to the next is less than the maximum permissible error.

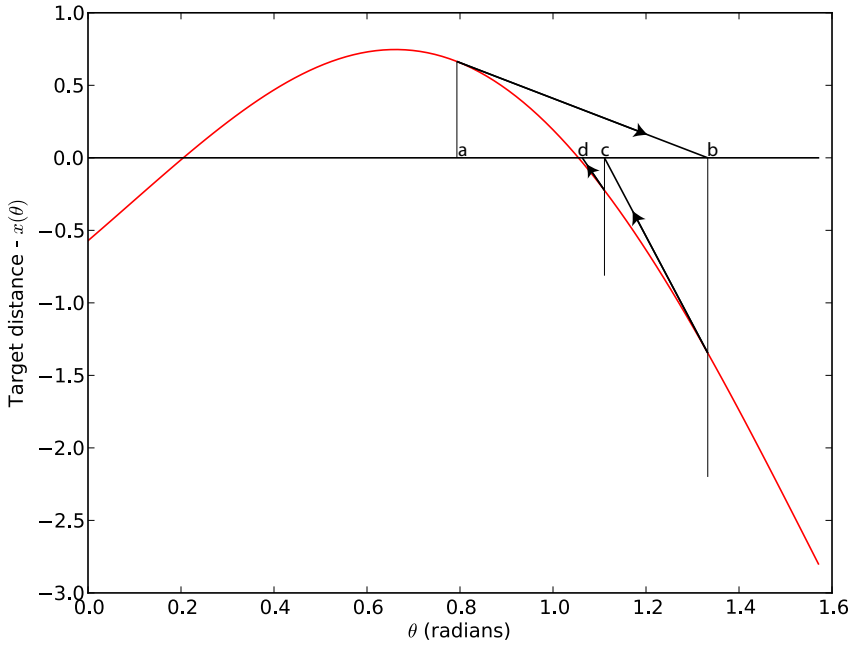

Figure 2.3: Newton's method of finding a root. a is the initial guess. The value of the function at a and the slope of the function at a leads us to the next approximate solution at b. The function value and slope at b lead to the approximate solution at c, and so on until the change between two successive approximations is less than the desired tolerance.

Example 2.0.4

```
def root_newton(f, df, guess, tolerance=1.0e-6):
    """
    Uses Newton's method to find a value x near "guess"
    for which f(x) = 0, to within the tolerance given.
```

```
    Default tolerance is 1.0e-6, if no tolerance is
    specified in the function call.

    It is required to pass this function both f(x) and f'(x).
    """

    dx = 2*tolerance          # initial dx > delta
    while dx > tolerance:     # main loop -- loop until
                              # dx < tolerance
        x1 = x - f(x)/df(x)   # Point-slope form of line
        dx = abs(x-x1)        # how much have things changed?
        x = x1                # Here's the new value

    return x                  # Best value so far
```

Newton's method is generally much faster than the bisection method. It is not as robust, though. For best results, your initial guess should be near the actual solution, and should not be separated from the solution by any local extremes. In addition, Newton's method does not work well near discontinuities in f. The biggest disadvantage to Newton's method, though, is that it requires knowledge of $f'(x)$. If you don't know, or can't obtain, a function that describes the derivative, then Newton's method is not an option.

Secant Method

The secant method is a modification of Newton's method which has the advantage of not needing the derivative function.

Start with *two* guesses, a and b. These should be near the desired solution, as with Newton's method, but they don't have to bracket the solution like they do with the bisection method. Use the values of $f(a)$ and $f(b)$ to approximate the slope of the curve, instead of using a function $f'(x)$ to find the slope exactly, as was done in Newton's method. (see figure 2.4.)

Find the value of x at which the line through points $(a, f(a))$ and $(b, f(b))$ hits the x axis. Next, repeat the process with the new point and the initial guess that is closest to the new point. Keep repeating until the change from one iteration to the next is less than the desired tolerance.

Like Newton's method, the secant method is not as robust as the bisection method. It can be unpredictable when used on discontinuous functions, and the initial guesses should be relatively close to the desired root. It is nearly as fast as Newton's method, though, and it has the advantage of not

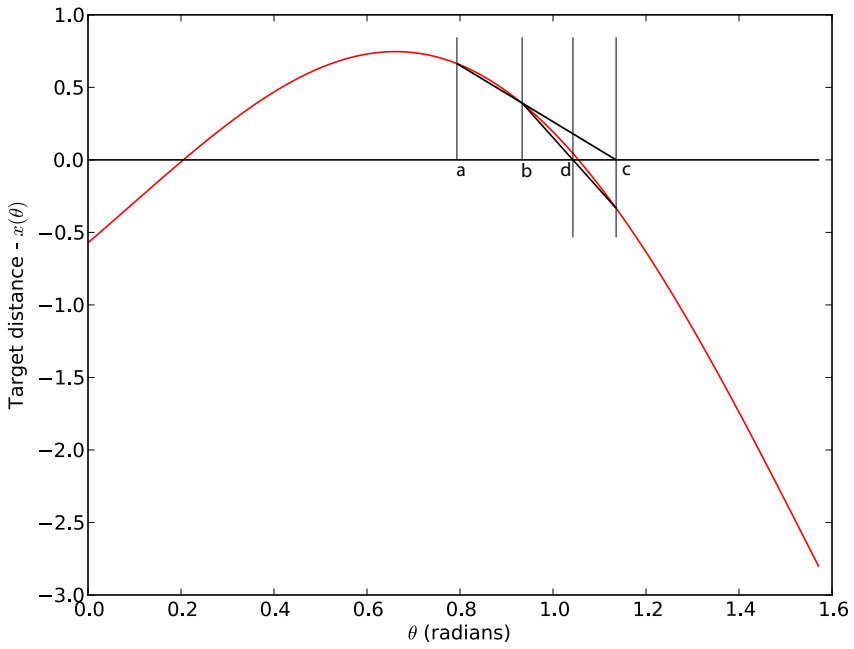

Figure 2.4: Secant method of finding a root. a and b are the initial guesses. The line through $(a, f(a))$ and $(b, f(b))$ leads us to the next approximate solution at c. The process is repeated until the change between two successive approximations is less than the desired tolerance.

needing a derivative. The secant method is a good choice for non-oscillatory smooth functions, such as usually appear in physics problems.

2.0.1 Python Libraries

Finding roots is a common-enough problem that people have written some good libraries to take care of it for you. One of those libraries is scipy.optimize, a subset of the "scientific Python" (scipy) library. Scipy includes implementations of the bisection method (`bisect()`), Newton's method (`newton()`), and several others including the `brentq()` routine. For a complete description of the Brent algorithm, see chapter 9 of [13]: the short version is that by combining root-bracketing, bisection, and inverse quadratic interpolation the routine can find roots about as fast as the secant method while *guaranteeing*

to find an existing solution between the given endpoints. Details on how to use brentq() are given in the scipy reference guide on the web: an example follows.

Example 2.0.5

```
from pylab import *
from scipy.optimize import brentq
x = brentq(sin, 2, 4)    # will find value of x between 2 and 4
                         # for which sin(x) = 0
print x, x-pi
```

$$\Longrightarrow 3.141592653589793, 0.0$$

It's important to know how to "roll your own"; to be able to write routines for basic tasks such as root-finding. It is also important to know when to use existing code. The brentq() routine is a good example of when to use existing code. Unless you are finding roots of a pathologically difficult function, brentq() will find the solution quickly and accurately and save you a lot of work.

2.1 Numeric Integration

Integration is an area in which numerical methods can be of considerable help. Take the function shown in figure 2.5. Depending on the functional form of $f(x)$, it may not be possible to calculate the integral of f from a to b analytically.

The integral of $f(x)$ from a to b is the area under the curve, though, which leads to the earliest numeric method of estimating that integral: carefully graph the function onto cardboard of known density and cut the shaded area out. From the weight of the cut section and the density of the cardboard, one can calculate the area and thus the value of the integral.

There are, of course, more precise methods of estimating integrals. Bear in mind, though, that all of the methods given here are methods of *estimation*. Do not blindly assume that because you use a given method that your answer is correct.

Simple Method

The simplest method of estimating the integral shown in figure 2.5 is to divide the integration range into N slices of width dx. Calculate the value

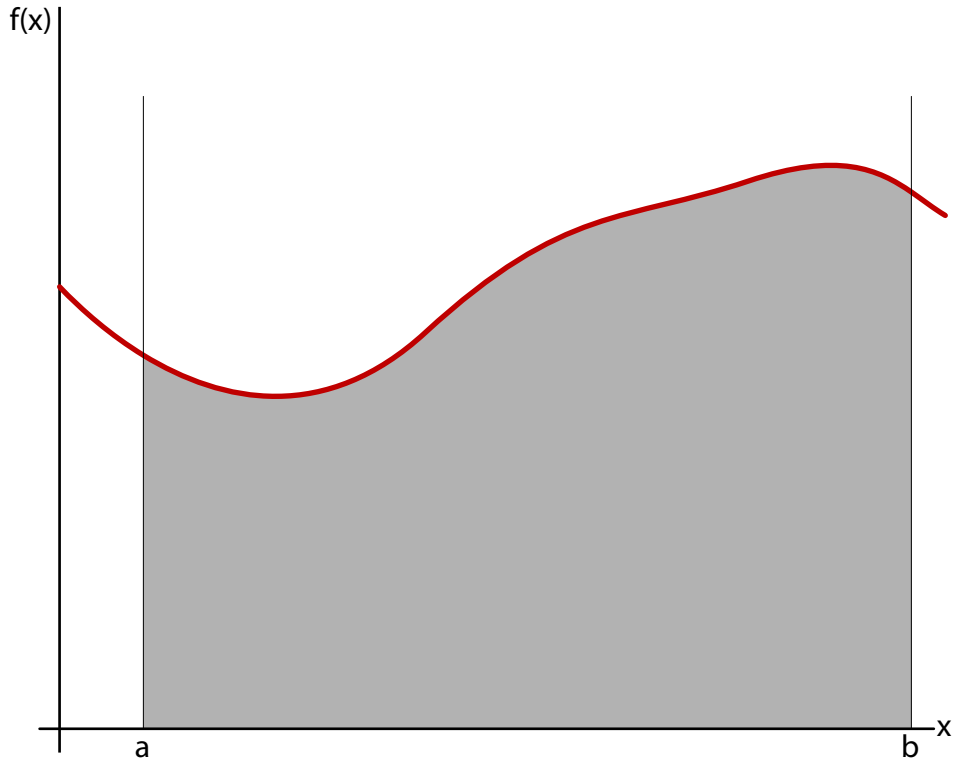

Figure 2.5: Some function $f(x)$, integrated from a to b. The exact value of the integral is the shaded region.

of $f(x_i)$ at some point on each slice, and find the area $A_i = f(x_i)\Delta x$ for each slice, as shown in figure 2.6. The integral is then approximately

$$\int_a^b f(x)\,dx \approx \sum_i f(x_i)\Delta x$$

Example 2.1.1
A python implimentation of the simple integration method could look something like this:

```
def int_simple(fn, a, b, N):
    """
    A routine to do a simple a rectangular-slice approximation
    of an integral.
```

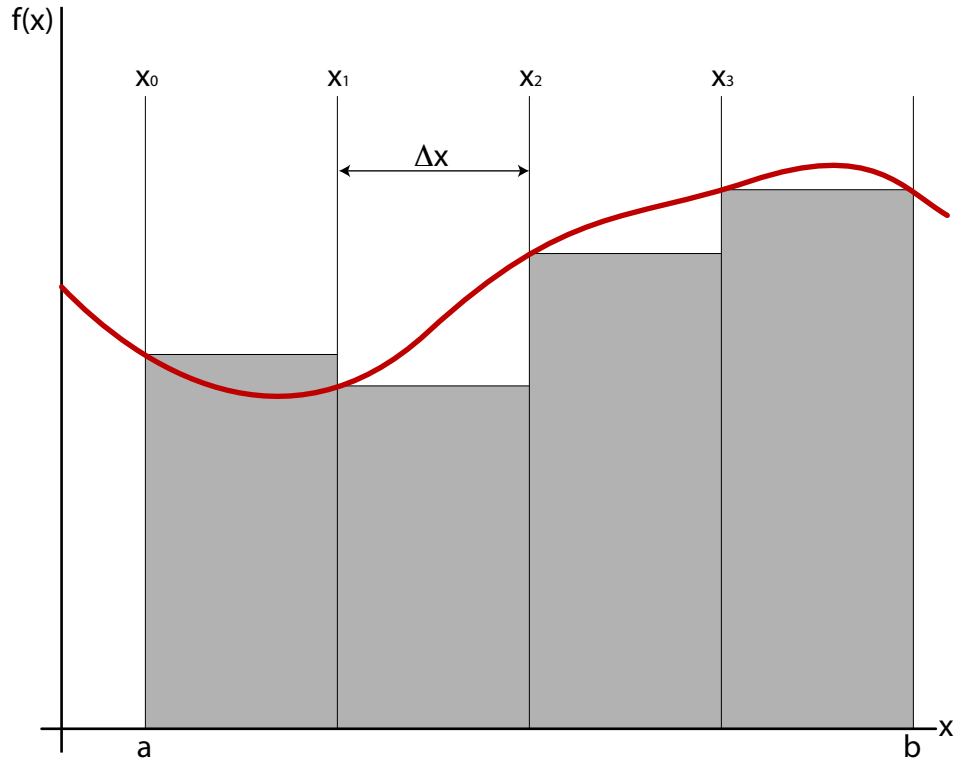

Figure 2.6: Simplest method of numerical integration. Here the value of $f(x)$ is calculated at the left edge of each slice: the final point $f(b)$ is not used in this calculation.

```
    fn: the function to integrate
    a,b: limits of integration
    N: number of slices to take
"""
I = 0                   # The value of the integral
dx = (b-a)/float(N)     # the stepsize, or width of slice
for j in range(N):
    x = a + dx * j      # The x value for this slice
    I = I + fn(x)*dx    # add the area of this slice
return I
```

This method is very simple to use and to understand, but as you can see from figure 2.6 it's not particularly good. If you increase the number of

slices, the estimate becomes better: in fact the exact value of the integral is

$$\int f(x)\,dx = \lim_{\Delta x \to 0} \sum_i f(x_i)\Delta x$$

so if you use a slice width of zero, the answer is exact. The problem, of course, is summing the infinite number of slices that accompany a slice width of zero!

There is a limited return on investment with an increasing number of slices. The error in each slice is the approximately triangular piece at the top. If you use twice as many slices, each of half the width, of the original slice, then that triangle becomes two triangles, each one roughly half the linear size of the original. The area of the smaller triangles is each 1/4 the area of the original, but there are two of them, so the area that is missed is cut by half overall.

There are more precise methods of doing that rough error calculation, but the results are the same: The precision of the result for the simple method goes as N. Doubling the number of slices takes you twice as long, since you must calculate $f(x)$ for twice as many points, and the uncertainty in your result is half what it was.

There are better ways of approximating the integral, but before we go on to them let's take a moment or two to improve the way we call an integrating function. In example 2.1.1, there were actually two different things that took place: calculation of the function to obtain the function values at regular intervals, and summing of those values to calculate the approximate value of the integral. Although these two seem to be parts of the same process, there are two significant advantages to treating them separately. First, not all integrations are integrations of functions. It is often necessary to integrate data, in which case there is no function to call to find the next value of $f(x)$. Instead, the most likely form of $f(x)$ is that of a list of data values. Second, the simple integration method shown in example 2.1.1 is somewhat unusual in that it uses the value of the function at any point only once. The more precise methods described in the next few pages use the function values more than once.

Because of these two reasons, it's best to design our integration routines to accept a list of function values and the spacing Δx between those values. This way the routine can be used more generally. For data sets, the calling program passes the list of values to the integration routine. For functions, the calling program calculates a list of function values once, then passes that list to the same integration routine.

Example 2.1.2

```
def int_simple(f,dx):
    """
    Simplest integration possible, using uniform rectangular
    slices. Gives a rather poor result, with error on the
    order of dx.

    f[] should be a list of function values at x values
    separated by the interval dx. The limits of integration
    are x[0] -> x[0]+dx*len(f).

    Note that this algorithm does not use the last point in f!
    """
    return dx*sum(f[0:-1])  # The sum() function is built-in,
                            # and does # exactly what you would
                            # expect.

# What follows is a program to use the int_simple() function
# to calculate the integral of sin(x) from 0 to pi.

from math import *

N = 100                     # number of slices.
                            # Higher N gives better results.
a = 0.0
b = pi
interval = (b-a)/float(N)

# Calculate the values of x to use.
x = [a + interval * i for i in range(N+1)]
# notice that it takes N+1 function values to define N slices!

# Calculate the values of f(x).
FunctionValues = [sin(value) for value in x]

print "The value of the integral is approximately",
print int_simple(FunctionValues, interval)
```

Now we're ready to move on to better integration methods.

Trapezoid Method

We can greatly improve the efficiency of our integration by approximating the slices as trapezoids instead of as rectangles, as shown in figure 2.7.

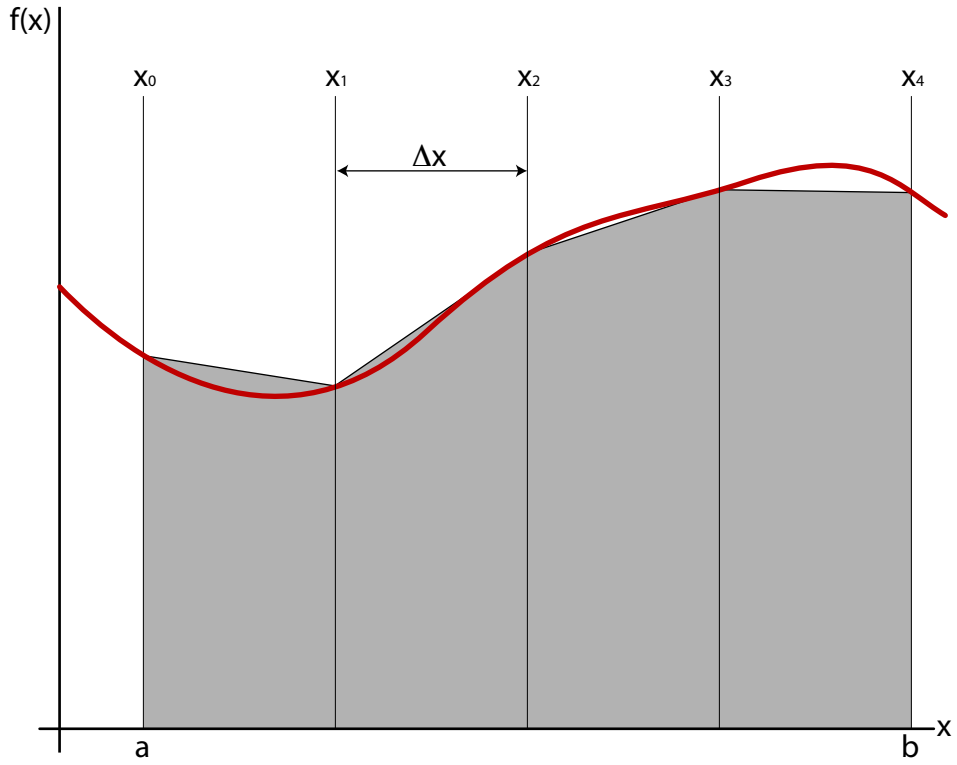

Figure 2.7: The trapezoid method of numerical integration. The value of $f(x)$ at both sides of each slice is used to calculate the area of each trapezoid.

The area each trapezoid is the width Δx times the average height of the slice sides:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} \frac{f(x_i) + f(x_{i-1})}{2} \Delta x \tag{2.5}$$

If you look at the right hand side of equation 2.5 closely, you can convince yourself that it works out to

$$\int_a^b f(x)\,dx \approx \left[\frac{f(x_0) + f(x_N)}{2} + \sum_{i=1}^{N-1} f(x_i) \right] \Delta x \tag{2.6}$$

This is probably the most efficient way to code the trapezoid method, although the actual function is left as an exercise.

Something to note about terminology, here: there's a difference between *points* and *slices*. In figure 2.7, there are four slices (the four grey trapezoids.) It takes five points (x_0–x_4) to define those four slices. Make sure in your code that it's clear whether N is the number of points or the number of slices, because if you're not careful either your value of dx or your range of integration will be incorrect! For large N, it makes a very small error, but the point is to avoid using large N.

Simpson's Method

The simple method approximates the function as a constant for the slices. The trapezoid method approximates the function as a linear equation for each slice. The next level of approximation, called Simpson's method after its inventor, is to approximate the function as a collection of parabolas, with each pair of slices providing the three points necessary to define the parabola for that region. (See figure 2.8.) The value of the integral is then approximately

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{\frac{N-1}{2}} \int_{x_{2i-1}}^{x_{2i+1}} g_i(x)\,dx \tag{2.7}$$

where the $g_i(x)$ are the parabolas through the sets of three points described previously.

The derivation of Simpson's method is somewhat complicated, but worth considering carefully. Start with the equation for a parabola which goes through the three points $(x_{k-1}, f(x_{k-1}))$, $(x_k, f(x_k))$, and $(x_{k+1}, f(x_{k+1}))$:

$$\begin{aligned} g_k(x) =& \frac{(x-x_k)(x-x_{k+1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} f(k-1) \\ &+ \frac{(x-x_{k-1})(x-x_{k+1})}{(x_k-x_{k-1})(x_k-x_{k+1})} f(k) \\ &+ \frac{(x-x_{k-1})(x-x_k)}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)} f(k+1) \end{aligned} \tag{2.8}$$

(Remember that $g_k(x)$ is the parabolic approximation of a section of the function $f(x)$, and $f(x)$ is what we're actually integrating.) We re-order

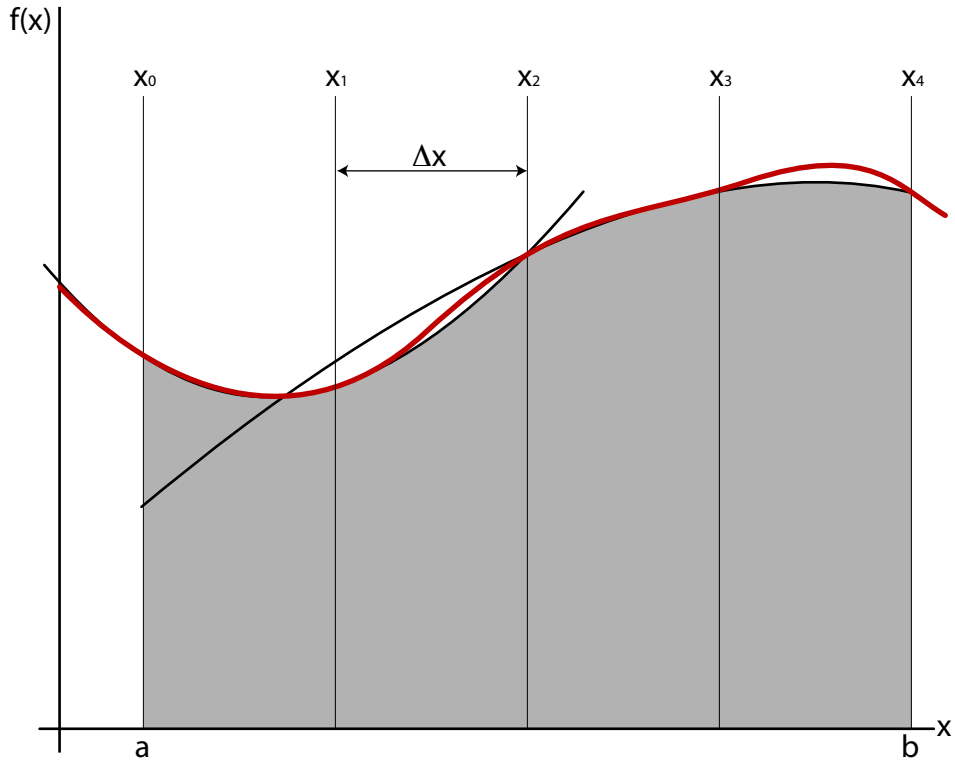

Figure 2.8: Simpson's method of integration. Each pair of slices is used to form a parabola which goes through the three function values associated with that pair. This example shows four slices, with two approximation parabolas.

equation 2.8 in terms of powers of the continuous variable x:

$$
\begin{aligned}
g_k(x) = x^2 & \left[\frac{f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& + x \left[\frac{-(x_k+x_{k+1})f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{-(x_{k-1}+x_{k+1})f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{-(x_{k-1}+x_k)f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& + \left[\frac{x_k x_{k+1} f(x_{k-1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})} + \frac{x_{k-1}x_{k+1}f(x_k)}{(x_k-x_{k-1})(x_k-x_{k+1})} + \frac{x_{k-1}x_k f(x_{k+1})}{(x_{k+1}-x_{k-1})(x_{k+11}-x_k)} \right] \\
& \equiv x^2[A] + x[B] + [C] \qquad (2.9)
\end{aligned}
$$

Each of the terms $[A]$, $[B]$, and $[C]$ depend only on the numeric values at the edges of the slices for the integrations.

The integral of $g_k(x)$ is just

$$\int_{k-1}^{k+1} g_k(x)\,dx = \frac{1}{3}[A](x_{k+1}^3 - x_{k-1}^3) + \frac{1}{2}[B](x_{k+1}^2 - x_{k-1}^2) + [C](x_{k+1} - x_{k-1}) \tag{2.10}$$

After a few pages of careful algebra, the combination of equations 2.7–2.10 gives us a final result:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3} \sum_{k=odd}^{N-1} (f(x_{k-1}) + 4f(x_k) + f(x_{k+1})) \tag{2.11}$$

The algorithm described above requires that there be an even number of slices. If there is an odd number of slices, it's necessary to add the area of the last slice separately. The derivation of that area is left as an exercise to the student, and the result is

$$A_{\text{last slice}} = \frac{\Delta x}{12}[5f(N) + 8f(N-1) - f(N-2)] \tag{2.12}$$

A Simpson's method integration function is shown in example 2.1.3.

Example 2.1.3

```
def int_simpson(f,dx):
    """
    Simpson's rule, using uniform slices.

    f[] should be a list of function values, separated on the
    x axis by the interval dx. The limits of integration are
    x[0] -> x[0]+dx*len(f).

    This particular algorithm does not require that there be
    an even number of intervals (odd number of points): instead,
    it adds the last section separately if necessary.
    """

    # number of points
    N = len(f)
    # initial value of integral
    integral = 0.0

    # add up terms
```

```
    for i in range(1, N-1, 2):
        integral = integral + f[i-1] + 4.0*f[i] + f[i+1]
    # multiply by dx, and divide by 3
    integral = integral * dx / 3.0

    # if number of points is even (odd number of slices) then
    # add the last point separately.
    if (N % 2) == 0:
        integral = integral + dx * (5.0*f[-1] + 8.0*f[-2]
                   - f[-3])/12.0

    return integral
```

Other Methods

The error of the simple measure goes as Δx, as we showed previously, so doubling the number of slices decreases the error in the result at a cost of twice the computation time. The error of the trapezoid method goes as Δx^2, so doubling the number of slices decreases the error by a factor of four but only costs twice as much in terms of computation time. That's a nice gain, but Simpson's method is even better. The error in Simpson's method goes as Δx^4 [11], so doubling the number of slices decreases the error by a factor of sixteen. Doubling the slices doubles the time, of course, but Simpson's method is considerably more complicated than the trapezoid or simple method so there's an additional time cost to Simpson's method compared to others.

Intuition might tell you that next method in the series would approximate the integral as a cubic over each set of three slices, and that this method would give you an error that went as some even higher power of Δx. While in theory that intuitive answer is correct, in practice it's not practical to implement the method. The amount of calculation for this third-order method is so high that you can get better results for less work by increasing the number of points on Simpson's method rather than using a cubic method.

There are methods of numeric integration that are specialized for certain classes of functions, and if you are curious about these I encourage you to read more about Gaussian Quadrature (and other methods) in *Numerical Methods in C* [13] or some similarly advanced text. For general use, though, Simpson's method is usually a good place to start.

2.2 Differentiation

One of the tools commonly used in an introductory physics laboratory is the "sonic ranger". This uses pulses of sound to determine the distance between a sensor and some object, typically a dynamics cart. The sonic ranger measures distance only, but we would like to know velocity and acceleration. This is an example where numeric differentiation becomes useful.

One way of finding a numeric derivative is to start from the definition of a derivative:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + x_o) - f(x)}{\Delta x}$$

So we could approximate the derivative by taking the difference between two successive data points and dividing by the distance between them:

$$f'_i \approx \frac{f_{i+1} - f_i}{\Delta x} \tag{2.13}$$

where the notation $f_i \equiv f(x_i)$ is used to indicate the discrete nature of our data and simplify the expression. This simple approach works, but there are better methods.

To find some of these better methods, let's start with the Taylor expansion:

$$f(x) = f(x_o) + (x - x_o)f'(x) + \frac{(x - x_o)^2}{2!}f''(x) + \frac{(x - x_o)^3}{3!}f^{(3)}(x) + \cdots$$

or in discrete notation,

$$\begin{aligned} f_{i+1} &= f_i + (x_{i+1} - x_i)f'_i + \frac{(x_{i+1} - x_i)^2}{2!}f''_i + \frac{(x_{i+1} - x_i)^3}{3!}f^{(3)}_i + \cdots \\ &= f_i + \Delta x f'_i + \frac{\Delta x^2}{2!}f''_i + \frac{\Delta x^3}{3!}f^{(3)}_i + \cdots \end{aligned} \tag{2.14}$$

We can immediately see the simplest method of finding a first derivative from this equation: it comes directly from the first two terms on the right hand side of equation 2.14. We can also see that since the error in this approximation goes as the power of Δx in the last term we actually use, the error in this simplest approximation is Δx.

We could also use the Taylor expansion to approximate the function at f_{i-1}.

$$f_{i-1} = f_i - \Delta x f'_i + \frac{\Delta x^2}{2!}f''_i - \frac{\Delta x^3}{3!}f^{(3)}_i + \cdots \tag{2.15}$$

Subtracting equation 2.15 from 2.14 gives us

$$f_{i+1} - f_{i-1} = 2\Delta x f_i' + 2\frac{\Delta x^3}{6} f_i^{(3)} + \cdots \tag{2.16}$$

If we take the Δx^3 and smaller terms as negligible and solve for f_i', we obtain the "three-point derivative" approximation:

$$f_i' \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x} \tag{2.17}$$

The error in equation 2.17 goes as the power of Δx in the last term we didn't just ignore, which is Δx^2. Computationally, this gives us a much better result than the simplest method for very little extra computational cost.

We can obtain higher precision by taking the expansion of $f_{i\pm 2}$:

$$f_{i+2} - f_{i-2} = 4\Delta x f_i' + 2\frac{8\Delta x^3}{6} f_i^{(3)} + \cdots \tag{2.18}$$

Now multiply equation 2.16 by 8 and subtract equation 2.18 to obtain the "five-point derivative" formula:

$$f_i' = \frac{1}{12\Delta x}\left(f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}\right) \tag{2.19}$$

The error in this "five-point derivative" formula goes as Δx^4, since the 4^{th}-order term in the Taylor expansion cancels in each of equations 2.16 and 2.18, so our first neglected term is 5^{th} order.

We can continue the process to obtain higher-order approximations, at the cost of increased complexity. A disadvantage of higher-order approximations, in addition to complexity, is that derivatives near the edges of the initial data set are lost. The five-point derivative formula can only calculate derivatives at points 3 through $N - 2$.

Higher-order derivatives are available by the same process. For example, adding equations 2.14 and 2.15 gives us

$$f_{i+1} + f_{i-1} = 2f_i + \Delta x^2 f_i'' + \cdots \tag{2.20}$$

so

$$f_i'' \approx \frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} \tag{2.21}$$

where the error is on the order of Δx^3, since the third-derivative terms cancel perfectly in equation 2.20. This is the "three-point second-derivative

formula". The "Five-point second-derivative formula" is given —through a similar process— by

$$f_i'' \approx \frac{1}{12\Delta x^2}\left(-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}\right) \tag{2.22}$$

As you might expect, the scipy library has a derivative function also.[1] It's scipy.misc.derivative(), and it uses a central difference formula (as described above) to calculate the n^{th} derivative at a given point. It expects a function, though, rather than a list of data points. It should be called with the function name, the point at which one wants the derivative, and (optionally) the value of dx, the order n of the derivative, optional arguments to the function, and the number of points to use.

```
from scipy.misc import derivative
print derivative(sin, pi)
>>> -0.8414709848078965
```

You will notice that the value returned by the above code is not -1. By default, $dx = 1.0$ and the order is the three-point approximation. To get better results, use a smaller value of dx and at least a 5-point approximation:

```
print derivative(sin, pi, dx=0.1, order=5)
>>> -0.99999667063260755
```

[1] `http://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html`

2.3 Problems

2-0 Write a generalized function implementing the secant method of root-finding, similar to example 2.0.2.

2-1 Write a program that uses the trapezoid method to return the integral of a function over a given range, using a given number of sample points. The actual calculation should be a function of the form int_trap(f,dx), where f is a list of function values and dx is the slice width.

2-2 Compare the results of the simple integration method, the trapezoid integration method from problem 1, and Simpson's method of integration for the following integrals:

(a)
$$\int_0^{\pi/2} \cos x \, dx$$

(b)
$$\int_1^3 \frac{1}{x^2} \, dx$$

(c)
$$\int_2^4 x^2 + x + 1 \, dx$$

(d)
$$\int_0^{6.9} \cos\left(\frac{\pi}{2}x^2\right) \, dx$$

For each part, try it with more and with fewer slices to determine how many slices are required to give an 'acceptable' answer. (If you double the number of slices and still get the same answer, then try half as many, etc.) Parts (c) and (d) are particularly interesting in this regard. In your submitted work, describe roughly how many points were required, and explain.

Note: The function in (d) is the *Fresnel Cosine Integral*, used in optics. It may be helpful in understanding what's going on with your integration if you make a graph of the function. For more information on this function, see [13].

2-3 Show that equation 2.12 is correct.

2-4 Write a program that can calculate double integrals. Use the simple method, but instead of using rectangular slices use square prisms with volume $f(x, y)\Delta x\Delta y$ Check your program by using it to calculate the volume of a hemisphere.

2-5 The Simpson's method program developed in this chapter requires uniform slice width. It's sometimes convenient to integrate using slices of varying width, for example if one needs to integrate data sets that are taken at irregular time intervals. Write a Simpson's method routine that integrates over non-constant-width slices. The routine should take two arrays as its arguments: the first array should be an array of function values $f(x)$, and the second an array of values of x.

2-6 The Fermi-Dirac distribution describes the probability of finding a quantum particle with half-integer spin ($\frac{1}{2}, \frac{3}{2}, \cdots$) in energy state E:

$$f_{FD} = \frac{1}{e^{(E-\mu)/kT} + 1}$$

The μ in the Fermi-Dirac distribution is called the *Fermi energy*, and in this case we want to adjust μ so that the total probability of finding the particle *somewhere* is exactly one.

$$\int_{E_{min}}^{E_{max}} f_{FD}\, dE = 1$$

Imagine a room-temperature quantum system where for some reason the energy E is constrained to be between 0 and 2 eV. What is μ in this case? At room temperature, $kT \approx \frac{1}{40}$ eV. Feel free to use any of the integration and/or root-finding routines you've learned in this chapter.

2-7 Write a function that, given a list of function values f_i and the spacing dx between them, returns a list of values of the first derivative of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the function and $\cos(x)$.

2-8 Write a function that, given a list of function values f_i and the spacing dx between them, returns a list of values of the second derivative of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the function and $-\sin(x)$.

2-9 Derive equation 2.22.

Chapter 3

Numpy, Scipy, and MatPlotLib

3.0 Numpy

Python lists are quite powerful, but they have limitations. Take matrices, for example: a two-dimensional array can be expressed as a list of lists.

```
M = [ [1,2], [3,4] ]
```

The result is something that is very much like the matrix

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .$$

Addressing elements in this "matrix" is clumsy: the top-right element would be addressed as M[0][1] instead of the more intuitive M[0,1]. More bothersome to us in a computational physics course, though, is that although a list of lists looks something like a matrix it doesn't behave like a matrix mathematically!

```
print M*3
[[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]]
print M+M
[[1, 2], [3, 4], [1, 2], [3, 4]]
print M*M
TypeError: can't multiply sequence by non-int of type 'list'
```

There is, of course, a package to provide real matrices to Python. Numpy, short for "Numerical Python", provides everything we need and much more, in the form of numpy "arrays". Setting up an array is similar to what you'd do for a list of lists, but with the specification that the item is an array:

```
from numpy import *
M = array([ [1,2], [3,4] ])
print M
[[1  2]
 [3  4]]
```

Note that the array is automatically printed in an array format. Scalar multiplication now works as expected:

```
print M*3
[[ 3  6]
 [ 9 12]]
```

As does addition:

```
print M+M
[[2  4]
 [6  8]]
```

There are at least three different ways of multiplying matrices: the default is scalar multiplication of matrix elements:

```
print M*M
[[ 1  4]
 [ 9 16]]
```

But of course one can also do dot and cross product multiplications with numpy:

```
print dot(M,M)
[[ 7 10]
 [15 22]]
print cross(M,M)
[0 0]
```

In fact, most common matrix operations are built in to numpy:

```
print transpose(M)
[[1  3]
 [2  4]]
print inv(M)
[[-2.   1. ]
 [ 1.5 -0.5]]
print det(M)
-2.0
print eig(M)
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

Numpy uses the LINPACK linear algebra package, which is written in FORTRAN and is extremely fast. The fact that arrays are multiplied element-wise by default is a great advantage. It means that if you have large lists of numbers that need to be multiplied together, you can define each array of numbers as a numpy array and then just mulitply the arrays. Doing the multiplications this way is much faster than using a **for** loop, and generally easier as well. Similarly, functions that would normally calculate single values will usually calculate array values if given an array input.

Example 3.0.1
Calculate sine of $\{\pi/6, \pi/5, \pi/4, \pi/3, \pi/2\}$.

```
x = array([pi/6, pi/5, pi/4, pi/3, pi/2])
print sin(x)
[ 0.5        0.58778525  0.70710678  0.8660254   1.
]
```

The numpy package also includes *many* other functions of interest to physicists and mathematicians.

Example 3.0.2
You are given a system of linear equations as follows, and need to find the values of w, x, y, and z:

$$\begin{aligned} w + 3x - 5y + 2z &= 0 \\ 4x - 2y + z &= 6 \\ 2w - x + 3y - z &= 5 \\ w + x + y + z &= 10 \end{aligned}$$

The solution, using numpy, is to rewrite the system of equations as a matrix multiplication:

$$\begin{bmatrix} 1 & 3 & -5 & 2 \\ 0 & 4 & -2 & 1 \\ 2 & -1 & 3 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 5 \\ 10 \end{bmatrix}$$

or

$$Mx = b$$

So we define M and b and solve using the linalg.solve() function of numpy:

```
M = array([ [1,3,-5,2], [0,4,-2,1], [2,-1,3,-1], [1,1,1,1] ])
b = array([0,6,5,10])
x = linalg.solve(M,b)
print x
[ 1.  2.  3.  4.]
```

We will be using many numpy functions throughout the rest of the course, but there are several other tools in numpy that we use frequently enough to mention specifically here.

arange() Creates an "Array Range". This is just like the range() function built into python, except it returns a numpy array and *the step doesn't have to be an integer.*

```
print arange(0,1, 0.25)
[ 0.    0.25  0.5   0.75]
print arange(0,1.01, 0.25)
[ 0.    0.25  0.5   0.75  1.  ]
```

linspace() Creates a "linearly-spaced" array. This is similar to arange() except instead of the step value you tell it the number of points you want. The resulting array will contain both endpoints and evenly-spaced values between those points.

```
print linspace(0,1,5)
[ 0.    0.25  0.5   0.75  1.  ]
```

logspace() Just like linspace() but the values are spaced so that they are evenly distributed on a logarithmic scale. The stop and start values should be given as the powers of 10 that are desired.

```
print logspace(1, 3, 4)
[    1.    10.   100.  1000.]
```

zeros() Produces a numpy array filled with zeros of the specified type. This is useful for setting up an empty array that your program can then fill with calculated values. The "type" can be float or int or double (or others) and specifies what format should be used for storing values. The default type is float.

```
print zeros([2,3], int)
[[0 0 0]
 [0 0 0]]
```

ones() Just like zeros() but fills in ones instead.

Numpy arrays are the standard format for any numeric work in Python, and are the expected format for Scipy and MatPlotLib.

3.1 Scipy

As one might guess, "Scipy" is short for "Scientific Python" and it is a pack of extensions to Python that provides numerous scientific tools. Where numpy provides basic numeric tools such as the ability to do nice things with matrices, Scipy provides higher-level tools such as numeric integration, differential equation solvers, and so on.

In the previous chaptor we derived a method for calculating integrals using Simpson's method: Simpson's method is included in Scipy.

```
from scipy.integrate import simps
```

The documentation for this function shows that it has some extra features not discussed previously: pydoc scipy.integrate.simps to see these features.

Scipy also provides the ability to integrate functions (rather than lists of y values) using Gaussian Quadrature.

```
from scipy.integrate import quad
print quad(sin, 0, pi)
(2.0, 2.2204460492503131e-14)
```

The quad() function takes any function (sine, in this case) and integrates over the range given (0, pi). It returns a tuple containing the value of the integral (2.0) and the uncertainty in that value.

If you need to integrate to infinity, scipy provides for that also:

```
from scipy.integrate import inf
print quad(lambda x: exp(-x), 0, inf)
(1.0000000000000002, 5.8426067429060041e-11)
```

Scipy also provides Fourier transforms, differential equation solvers, and other useful tools that will be described later in this text.

3.2 MatPlotLib

"A picture is worth a thousand words," as the saying goes, and nowhere is that more true than in this field. For all but the most simple numeric problems, graphs are the best way of showing your results. The MatPlotLib

library provides a powerful and easy set of tools for two-dimensional graphing.

Example 3.2.1
Plot sine and cosine over the range $\{-\pi, \pi\}$.

```
from pylab import *
x = arange(-pi, pi, pi/100)
plot(x, sin(x), 'b-', label='sine')
plot(x, cos(x), 'g--', label='cosine')
xlabel('x value')
ylabel('trig function value')
xlim(-pi, pi)
ylim(-1,1)
legend(loc='upper left')
show()
```

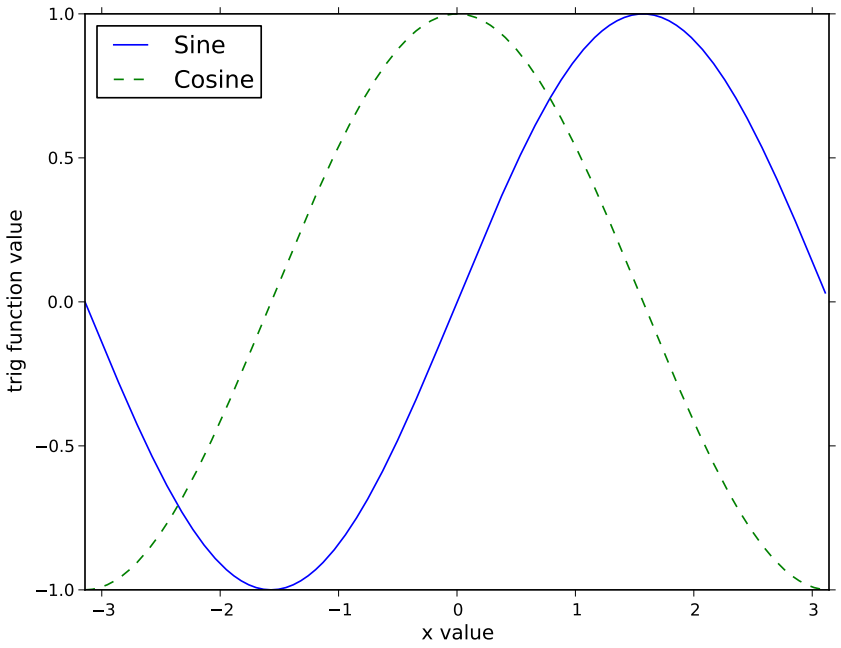

Figure 3.0: Figure produced in example 3.2.1.

Character	Line Type
'-'	solid line
'–'	dashed line
':'	dotted line
'-.'	dash-dot line

Table 3.1: MatPlotLib line types

Character	Datapoint indicator
'o'	circle
'^'	upward-pointing triangle
's'	square
'+'	plus
'x'	cross
'D'	diamond

Table 3.2: MatPlotLib datapoint types

Let's go over the details of example 3.2.1.

```
from pylab import *
```

The pylab package is a wrapper package that imports numpy, scipy, and matplotlib. It indirectly takes care of trig functions and π also, which makes it a nice start to everything we're doing.

```
x = arange(-pi, pi, pi/100)
```

This sets up an array of x values, 100 of them, from $-\pi$ to π.

```
plot(x, sin(x), 'b-', label='sine')
```

This sets up the plot of sin(x). The first parameter is the array of x-axis values, the second is the array of y-axis values. (Remember that numpy arrays can be calculated "all at once": since x is an array, $\sin(x)$ is an array containing $\sin(x_i)$ for each value x_i in the array x.)

The third item in the plot function call specifies the line style, in this case a blue (b) solid line (-).

The various characters in tables 3.1–3.3 can be combined, so for a green dotted line connecting circles at the datapoints one would specify 'go:'.

The label keyword sets what is to be displayed in the legend. The xlabel() and ylabel() set the labels on the x and y axes, respectively. The xlim()

Character	Color
'b'	blue
'c'	cyan
'g'	green
'k'	black
'm'	magenta
'r'	red
'w'	white
'y'	yellow

Table 3.3: MatPlotLib built-in colors

and ylim() function calls set the range displayed on the graphs. These are optional, and matplotlib is pretty good at selecting reasonable values if you don't specify any. The legend() call sets various characteristics of the legend — in this case it is used to set the location.

The final function call in the example is show(), which draws the graph on-screen and halts program execution until that window is closed. If instead you wish to save the figure to a file, use savefig('filename'), where the filename should end with the type of file you wish to save. The most common filetypes are .ps, .png, and .pdf. If you wish to save the figure *and* show the results, call savefig() before show() since show() halts program execution.

Where to learn more

Pylab is much more powerful than one would guess from the simplified description in this chapter. Pylab is also rapidly changing. Although the core functionality described here is unlikely to change significantly, more features are constantly being added. Google is your friend: searching for "matplotlib polar", for example, will tell you everything you need to know about plotting on non-Cartesian axes.

If you prefer to browse through books rather than websites, Shai Vaingast's book [17] is particularly useful.

3.3 Problems

3-0 Solve this system of equations for a through f.

$$\begin{aligned} a/5 + b + c + d - e - f &= 24.1312 \\ a + c - d + 5e + f &= 46.2798 \\ -a - 3b + 2c - d - e - f &= -61.8372 \\ 5b - c + d + 2e &= 31.1466 \\ a - 2b + 3c + d/2 &= 51.2106 \\ a/2 - 2b - 2c - d + e - 6f &= -5.7008 \end{aligned}$$

3-1 A ball is thrown upwards with initial velocity $v_o = 5$m/s and an initial height $y_o = 3$ m. Write a Python program to plot y(t) from $t = 0$ until the ball hits the ground.

3-2 A mass m is suspended from a spring of spring constant k. The mass is displaced from equilibrium by an initial distance y_o, then released. Write a Python program to plot $y(t)$ for some reasonable set of parameters.

3-3 For the previous problem, plot a *phase-space* plot. The horizontal axis should be position, and the vertical axis velocity.

3-4 Repeat the previous two problems, but add damping to the spring-mass system. In other words, the equation of motion for the system is

$$\ddot{y} = -\frac{k}{m}y - \beta\dot{y} \; .$$

Assume that $\beta < 2\sqrt{k/m}$.