Paul A. Nakroshis

Introductory Computational Physics Using Python

January 2017

University of Southern Maine
Department of Physics

Preface

Since the advent of quantum mechanics in the 1920's, the subject matter of most of undergraduate physics hasn't changed significantly[1, 2]. Students still start with basic Newtonian physics, thermal physics, move on to study electricity, magnetism, and optics, and then take a standard sequence of more advanced courses: Modern Physics, Mechanics, E&M, Quantum Mechanics, and typically an upper level laboratory. Most physics departments also have added a minimal computing requirement that is not really integrated into the physics curriculum.

Although the subject matter we teach hasn't changed significantly, there have been many efforts to change the *manner* in which we teach. These changes are the result of research how people learn, and in large part, have roots in constructivist theories of learning. The upshot of this is that we now know that as human beings, we carry about mental models of how we *think* the world works. Many of these ideas are actually false, and until we confront these misconceptions, we are doomed to hold onto them. Hence, open—ended hands-on learning activities (like non-cookbook laboratory experiments) are excellent tools to facilitate real learning when carefully designed to force students to confront common misconceptions.

Along with changes brought by physics education research, another tool in that is finally starting to take hold in several physics departments (perhaps most noteably at Oregon State University, under the direction of Rubin Landau[2, 3, 4]), is the clear integration of computers as tools for learning about physics. Slide rules were abandoned in the late 1970's with the advent of pocket calculators, and scientists have been using computers for many decades now, but because computing power has been growing rapidly (at a pace slightly below that predicted by Moore's Law), a common modern laptop computer has the computing power that dwarfs that of mainframe computers of the past.

I believe that as physicists, we have not been coming close to using computers effectively in the college classroom, and we should be taking advantage of them as learning tools. Computers provide us with a multifaceted tool that is extremely useful.

First, programs such as Mathematica or Maple, provide, at minimum, a toolset that makes graphing calculators appear as the slide rules of yesteryear, and at their highest levels provide a full-fledged computing envronment. Once one learns even a small piece of such programs, tables of integrals become obsolete, and whole new easily utilized capabilities become easily accessible. We should be familiarizing physics students with these tools so that they may use them throughout their careers.

Second, computers have become indispensable tools for the simulation of complex physical systems that do not admit analytic solution. One does not need to look far to see examples of physical systems that have non-analytic solutions. In mechanics, the three body problem is a famous example; in the study of granular materials, a simple ball bouncing on a vibrating plate is a classic example of chaotic motion.

There are many more examples, but the relevant point is this: even though we have the computing power to simulate many interesting physical systems that are accessible to undergraduate physics majors, we persist in teaching physics majors as if the only interesting problems are those with closed-form analytic solutions. Of course, I am not advocating that we cease studying the classic analytically soluble problems, but rather, we shouldn't constrain our curriculum to only these problems.

This book is an attempt to help change this paradigm. The goal of this text is to provide a true *introductory* text on computational physics that provides students with sufficient tools to be able to simulate interesting physical systems. Because this is an introductory course taken by all our physics majors (and even our universitiy's chemistry program requires computational physics), I want students to learn several tools that will be useful throughout their career.

First, since composing research papers is important for scientists, students in my course work on simulations and write up their results using the Jupyter Notebook, which allows them to mix markdown text, and to include LaTeX the defacto typesetting program for physicists and mathematicians. The students' reports receive feedback on the physics content, code, and the quality of their writing—this course serves as their first introduction to the genre of scientific journal writing. This text starts, therefore, with a brief introduction to LaTeX —but because LaTeX is a huge package, my intent is to get them using it at a basic level, not to make it the focus of the course.

Second, I want students to enjoy programming *and* to walk away from this course with a toolbelt of skills that they can take to other laboratory and even theoretical classes. Thus, before diving into the details of programming, I begin with a few exercises that get their feet wet reading and plotting data files, skills that are immediately useful in their other coursework, and, of course, vital to this course.

Python is ideally suited to such work[5, 6]; it now has a mature interactive notebook–based interface similar to Matlab and Mathematica, and it can be used in a purely procedural fashion if needed. At a more advanced level, one can use Python as a fully object-oriented manner similar to Java or C++, though I do not emphasize the object oriented features in this course.

My assumptions are that students have used a computer running either the Mac, Windows, or Linux operating systems, and have basic familiarity with creating folders and files, and have installed Python 3.x along with SciPy, and MatplotLib, all of which are available for free for any of the three platforms. In addition, I require my students to gain familiarity with LaTeX and thus, all students need to have a functioning LaTeX distribution. It is assumed that the faculty member teaching using this text already has such familiarity and can assist students with the setup of their laptops.

When I first began teachin this course, most students did not have their own laptops, and I needed a university computer lab to teach in; nowadays, the situation is reversed, and most students have either a PC or a Mac. Nonetheless, I find Linux to be a far better environment to program in, and I strongly urge all students to run either Ubuntu Linux or Linux Mint. Getting all students on the same page with a functioning Linux OS typically takes the entire first day, during which I show students how to install programs, use the bash shell, and generally work their way around the Linux OS.

All of my development for this book has occurred on various Macintosh computers with 16GB ram, running Mac OS X and running Ubuntu Linux 16.10 as a virtual machine in IBM's VirtualBox. All python code used in this book should work equally well on Windows or Linux.

Paul A. Nakroshis

Portland, Maine
Jan 2017

Contents

1 Brief introduction to LaTeX

- basics of LaTeX as a typesetting language
- LaTeX resources and links
- how to include python code in a LaTeX document

1.1 Basic idea of LaTeX

LaTeX is not a word processor like Microsoft Office, LibreOffice, or Apple's Pages, which are all WYSIWYG (What You See Is What You Get) word processors where the editing window and the output are one and the same. Rather, when you use LaTeX you use a *text editor* (not a word processor) to create the content and the *instructions* for what to do with the content, and then you invoke LaTeX to process this text/instruction file into a printable output. This processing happens quite quickly, and produces a pdf file which can then be viewed on screen and will look exactly as it will print. Since this portable document format is pretty ubiquitous, almost anyone or any device (even smart phones and tablets) can read and display the file.

To become an expert at LaTeX is a lifelong task; I've been using it for almost 30 years and still do not know all the features available. So don't worry about figuring it all out in a single semester—you only need to know enough to get by, which fortunately, is not so hard. LaTeX is also extensible by numerous packages that allow one to easily add functionality, so there's no limit to what you can do.[1]

Of course, where LaTeX really shines is in its ability to typeset mathematics in an elegant and efficient manner. You'll even see that the iPython Notebook includes the ability to typeset mathematics—and it uses LaTeX to do so. Thus, by introducing you to LaTeX , you'll be learning a tool that almost all mathematicians and physicists use, and has the depth to serve your techical writing needs for the rest of your life. LaTeX produces beautifully formatted output that is without peer, so let's get started with a basic introduction.

[1] In my course, I provide students with a suitable template file for creating their simulation reports; the goal is to introduce the use of LaTeX as a tool, not make it the *focus* of the course.

1.2 A simple LaTeX document

The basic LaTeX text-editor created document consists of a *preamble* and the *Body*. The preamble defines the type of document you want to create—for example: article, letter, report, presentation—and the body is the content. Here is the most simple bare bones LaTeX document possible:

```latex
\documentclass[12pt]{article}
% preamble
\begin{document} % body
Hello World! % this is the standard first program for any language
\end{document}
```

The `\documentclass[12pt]{article}` command sets up the document as an article in 12pt type. There are many other document types, some of which are listed in Table 1.1. The preamble may also contain other formatting commands deal-

Table 1.1: Some popular LaTeX document types for the documentclass declaration.

Document Type	description
article	for journal articles, short reports, program documentation, etc.
report	for longer reports containing several chapters, small books, thesis.
book	for real books
slides	for slides. The class uses big sans serif letters.
letter	for writing letters.
beamer	for writing presentations (i.e. powerpoint/keynote replacement)

ing with headers, footers, margins, and even loading other LaTeX packages, and setting up custom commands, but at minimum, you have to have the `\documentclass` command; if you want more information about the options for this command, see http://en.wikibooks.org/wiki/LaTeX/Basics.

The rest of your document is bounded by the

```latex
\begin{document}

\end{document}
```

environment, and this is where you put all of your content. Here's a short example of a complete LaTeX file (on the left), and it's output after typesetting; notice that in order to boldface some text, you have to invoke a command:

.tex file:

```
\begin{document}
Here is where you put some
text. To create a new
paragraph, simply put in a
blank line, and \LaTeX\ takes
care of the rest.

If you want to make text
boldface, then you
use \textbf{this command}.
\end{document}
```

LaTeX output:

Here is where you put some text. To create a new paragraph, simply put in a blank line, and LaTeX takes care of the rest.

If you want to make text boldface, then you use **this command**.

Notice that when you are entering straight text, then you do not need to worry about the length of the lines, as LaTeX will format the page for you. You just worry about the content. To create a new paragraph, just put in a blank line; there's no need to indent, for that will also be taken care of when you typeset the document.

LaTeX also contains specialized environments for making bulleted lists, enumerated lists, captioned figures, footnoting, and anything you can conceivable desire to do. For the most part, LaTeX commands are relatively straightforward, and you'll typically learn environments on a need-to-know basis; nonetheless, it's useful to get an overview of some of the features of LaTeX by reading some of the references that appear on http://www.latex-project.org/guides/—I recommend in particular the three sites listed in Table 1.2.

The (not so) short introduction to LaTeX 2ε at http://ctan.tug.org/tex-archive/info/lshort/english/lshort.pdf,
Getting to grips with LaTeX at http://www.andy-roberts.net/writing/latex and,
as a general reference online, the WikiBooks LaTeX site at http://en.wikibooks.org/wiki/LaTeX/.

Table 1.2: Several good LaTeX references.

1.3 LaTeX editor recommendations

1.3.1 LaTeX only editors

If you are running Linux (or, for that matter Windows, or OS X) there is a cross-platform open source editor called TeXworks http://www.tug.org/texworks/ that is quite good at writing LaTeX documents, and is what I'd urge you use at the outset.

On the Mac, the program TeXShop http://pages.uoregon.edu/koch/texshop/ is singlehandedly responsible for the resurgence of LaTeX on this platform—this open-source editor replaced an extremely expensive commercial alternative, and there was no program like it on Linux or Windows. The TeXworks program was spawned in order to make a cross-platform version of TeXShop. In any case, if you're on the Mac, both of these programs come with the MacTeX distribution which is available at http://www.tug.org/mactex/.

1.3.2 LaTeX and Python editors

When you are writing a document which includes LaTeX and python code, it's useful to be able to work on both things within one unified environment. Here are my recommendations for Linux and MacOS.

For Linux, after much agonizing testing, and looking for something that works pretty much out of the box, I recommend the program GEANY (get it through the Synaptic Package Manager)—make sure to download all the extensions too. When it is installed, you'll want to enable all the plugins. gEdit is also a good program, but I could not get it to correctly compile this document properly, whereas GEANY worked perfectly.

For Mac OS X, I recommend TextMate, which is the editor I used to compose this text. With TextMate, I could see a file directory of all the LaTeX chapters, python code, and process the LaTeX files and run the python code independently. This is by far the best all-purpose editor for this purpose on the Mac platform.[2] and all my development work for this course is done in this editor which you can get at http://macromates.com/.

Exercise 1.1. Open up a LaTeX aware editor, create a simple LaTeX document with a few lines of text, save the file,[3], and typeset it. You should get a nice output with the text you typed, and a page number 1 at the bottom of the page. There's nothing to hand in here, this is just a test to make sure your LaTeX installation is working. You might start with the "Hello World!" example in section 1.2.

---

[2] VIM and Emacs users would beg to differ!

[3] One oddity with LaTeX is that you cannot have a filename with a space in it; instead use a dash or an underscore if you must.

1.4 How to include mathematics into LaTeX

Incorporating mathematical equations in a document is reason enough to use LaTeX over *any* other authoring program. It's notation is simple, powerful, and results is gorgeous typeset equations. There are several ways to include mathematics.

1.4.1 Inline equations

An inline equation is an equation that occurs right in the course of the text; for instance, if I had a sudden urge to write $\sin \pi = 0$, it's easy to do in LaTeX; all I have to do is type `$\sin \pi = 0$` and it will be typeset right in place. Inline equations are typeset in math mode and are demarcated by a beginning and ending dollar sign (therefore, should you actually need a dollar sign symbol to discuss money, you have to use `\$`).

Inline equations work okay for simple quations, but something more complicated like $\int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}$ doesn't look so good as an inline equatiuon. For this, we want to use the `\displaymath` environment.

1.4.2 Centered and un-numbered equations

Suppose we want the previous equation to look more readable, but didn't care to number it; then we use two dollar signs and write

```
$$\int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}.$$
```

which gives us a nicer looking result centered on its own line (in LaTeX parlance, this is called the `\displaymath` environment):

$$\int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}.$$

Notice that this equation ended a sentence, so I put a period at the end of the equation. Punctuation is important! There is an equivalent way to get the above equation, which is to use

```
\[ \int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}.\]
```

1.4.3 Subscripts & superscripts

A note about subscripts and superscripts: notice that the lower limit of the definite integral was preceded by an underscore character, and the superscript with an up-caret. For a single character sub or superscript, it is sufficient to use the character immediately after the underscore or up-caret; however, if there is more than one character (like T_0), then the sub or superscript must be enlosed by braces.

1.4.4 Numbered equations

If you have a formula that you want to be able to refer back to in the text, then you want to number it (LaTeX will do this automatically for you) and give it a label so that you can refer back to it. For instance, suppose I want to refer to Equation 1.1:

$$\int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}. \tag{1.1}$$

Here is what I typed:

```
...suppose I want to refer to Equation~\ref{eq:sillyIntegral}:
\begin{equation}\label{eq:sillyIntegral}
\int_0^{T_0} x^2\;dx = \frac{T_0^3}{3}.
\end{equation}
```

Use the `\begin{equation} ... \end{equation}` to enter math mode and this tells LaTeX that I want to number the equation. I then gave the equation a descriptive label. I've evolved a strategy to always use a label format that indicates what the item is—i.e. `eq:` for equations, `fig:` for figure labels, etc. You are free to just use `sillyIntegral` if you like. Then to refer to this equation, I type `Equation~\ref{eq:sillyIntegral}` and LaTeX *automatically* takes care of the numbering for me.

1.5 How to include Python code in your LaTeX document

Now, suppose you want to include some Python code (or for that matter, code in practically any computer language) in your report. A nice way to do this is to use the `listings` package
(see the WikiBooks site at http://en.wikibooks.org/wiki/LaTeX/Packages/Listings).

1.5.1 Short code snippet

If you have a short *python* code section or *snippet*, your *entire* LaTeX code might look like this:

```
\documentclass[12pt]{article}
\usepackage{listings} % loads the listings package
\usepackage[usenames,dvipsnames]{color} % load a color package
% define some colors
\definecolor{light-gray}{gray}{0.97}
\definecolor{lightGreenPython}{RGB}{240,255,240}
\definecolor{wheat}{RGB}{252,247,234}
%
%%% The following command defines some options in the listings
%%% package to nicely format python code; feel free to use it.
```

```
   %%%
   \lstdefinestyle{pythonSnippet}{
   language=python,
15 backgroundcolor=\color{wheat},
   frame=leftline,
   framerule=0.5pt,
   rulecolor=\color{RoyalPurple}
   }
20
   %%%
   \begin{document}
   %
   \noindent Here is a code snippet:
25 \begin{lstlisting}[style = pythonSnippet]
   y0, v0, t = 10.0, 0.0, 2.0 # defines the variables
   y = y0 + v0 * t -4.9 * t**2 # computes the y position
   print y                  # prints out the value of y
   \end\{lstlisting}
30 \end{document}
```

Listing 1.1: Including a short python snippet

You can see the output produced by running this file through the LaTeX engine in here:[4]

```
y0, v0, t = 10.0, 0.0, 2.0 # defines the  variables
y = y0 + v0 * t -4.9 * t**2 # computes the y position
print y                  #  prints  out  the  value  of  y
```

1.5.2 Extended Section of Code

If you have an entire python script, it's much more convenient to **not** have to cut and paste your code into the LaTeX file since it's often the case that you find a small error in your code and then you need to re-paste the code or edit it directly in the LaTeX file itself. A much nicer way to do this is to use the ability of the `listings` package to directly include the file containing the python code. What follows is the complete LaTeX code that will allow you to accomplish this.

```
\documentclass[12pt]{article}
\usepackage{listings} % loads the listings package
\usepackage[usenames,dvipsnames]{color} % load a color package
% define some colors
```

[4] Note that due to the difficulty of having Listing 1.1 include an example of the *listings* package within the *listings* environment there was an unavoidable backslash in line 29 before lstlisting which you should remove in an actual LaTeX file.

```
\definecolor{light-gray}{gray}{0.97}
\definecolor{lightGreenPython}{RGB}{240,255,240}
\definecolor{wheat}{RGB}{252,247,234}
%
%%% The following command defines some options in the listings
%%% package to nicely format python code; feel free to use it.

%%%
\lstdefinestyle{pythonSnippet}{
language=python,
backgroundcolor=\color{wheat},
frame=leftline,
framerule=0.5pt,
rulecolor=\color{RoyalPurple}
}
\lstdefinestyle{MyPythonStyle}{
language=python,
frame=lines,
framerule=0.5pt,
rulecolor=\color{RoyalPurple},
numbers=left,
stepnumber=2,
numberstyle=\tiny,
numbersep=5pt
}
\begin{document}
\lstinputlisting[label=plot,
caption=\LaTeX\ output by direct reference to a python file.
]
{Code/Assignment_01/plot.py}
\end{document}
```

The above LaTeX code produces the output shown in (Listing 1.2). Notice that the python file is well commented—a habit you should get into. Good code is well-documented, and easily understood by anyone familiar with the language.

```
"""
plot.py
Plots a simple function. This method of invoking matplotlib
automatically invokes matplotlib.pyplot and numpy in a manner
which is not advisable for long programs, but when I want to
quickly plot something, this is what I use at a terminal prompt.
"""

from pylab import *     # some people discourage this
t=linspace(0.0,2*pi,100) # 100 point list from 0 to 2*pi
plot(t,sin(t))          # format: plot(xaxis,yaxis)
```

```
show()                      # display plot
```

Listing 1.2: LaTeX output by direct reference to a python file.

Now you can see the advantage of this method, right? You can work on the LaTeX code for your report, and at the same time, you can be working on the python code and the *Listings* package takes care of including the final version of the code automatically since it merely links to this file and then formats the text file nicely including syntax highlighting. Good luck trying to do that seamlessly in any other word processor!

Also notice that at the beginning of the python file, I placed a description of the python script within triple quotes. This is the format python uses for documentation strings. This triple-quoted string provides a good description of what the python code does, and can be accessed interactively from a python interpretter, as we will see soon. As always, you can read more about this by performing a web search for "Python docstrings".

1.6 Final Words on LaTeX

LaTeX is a huge package and it takes years to learn. That's both good news and bad. The bad, of course, is that it takes some getting used to using, and when you make an error, the feedback that LaTeX gives you is not always transparent.

On the other hand, when you do get an error, a little googling of the error message will often set you straight. And, you likely have a few experts on LaTeX in your friendly neighborhood physics or mathematics departments that can help you. In addition, there is lots of room to grow in LaTeX ; you can design entire books, all typeset in gorgeous manner.

A good idea at this point is to sit down with a good LaTeX tutorial and practice your new LaTeX skills with the following exercises.

Exercise 1.2. When you create a LaTeX document, it should have the extension .tex, and after you typeset it, several other files will appear (depending on the document); thus it's useful to keep a LaTeX document in its own folder. Go through the following steps, designed to get you in a good organizational file management habit.

1. Create a folder called `ComputationalPhysics` (no spaces) and inside that, create a folder called `LaTeXPractice` and with that, a new LaTeX document whose `documentclass` is `article` and is called `exercise1.tex`. You may use any LaTeX editor you like, but for simplicity, I recommend that you use `Texworks` as it is available on all three major computing platforms. You can create this file from the `File > New from Template` menu option, and choose the `article` document class. Add the following to your document:
2. Add title, author, and date information, and make a title appear at the top of the page.

3. Add a paragraph or two, and include some **boldface text**, some *italic text*, and the famous equation $E = mc^2$, both as an inline formula and as a formula on it's own line like this:

$$E = mc^2.$$

4. Add an enumerated list.
5. Add a two column `Table` like this (hint: google latex booktabs): Fill in the second

Category	Favorite thing
food	
song	
exercise	

Table 1.3: Some of my favorite things.

column.
6. Download a full resolution picture of Albert Einstein from *Wikipedia*, place it in a `Figures` folder within the `LaTeXPractice` folder. Display the .jpg figure in your paper using the LaTeX `figure` environment as shown in Figure 1.1. Make the image 6 cm wide. Notice that LaTeX puts the image wherever it pleases. Don't worry about that right now.

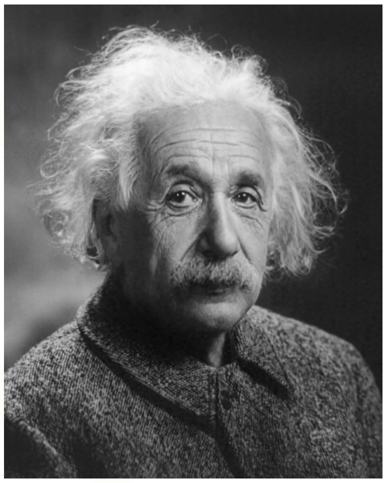


Fig. 1.1: Albert Einstein, image in the public domain; from *Wikipedia*.

7. when you are done, submit your work by printing out the .tex file and the .pdf file that results from running LaTeX on your code.

2 Python Basics

- The Terminal, iPython Notebook, & IDE's
- Loading libraries, creating functions and scripts
- Data primitives: Strings, Numbers, Booleans
- Data structures: Lists, Dictionaries, Numpy arrays
- Flow Control: If, while, list comprehensions

This chapter will assume that you have a computer running either the Mac or Linux operating systems, and have installed a working version of Python 3.x and LaTeX .

Briefly, the easiest way to install python is to use the Anaconda Python Distribution (free for academic use). Google this, download the installer for your operating system, and you should be good to begin. Installing a working LaTeX distribution is more involed; on MacOS, go to http://www.tug.org/mactex/mactex-download.html, and install the MacTeX distribution (about 2 Gb!). On Linux, I recommend installing the synaptic package manager, and then installing `texlive` as well as many of the optional LaTeX packages — here it will be useful to have the help of an experienced LaTeX user to help install all the needed LaTeX packages. To minimize the hassle, I provide students in my class a pre-packaged Ubuntu virtual machine that contains all needed software installs.

2.1 General Overview; Ways of interacting with Python

In order to use Python, you have to type commands, and to do so you have many options; I've grouped them into four categories, with the applications I most often use in boldface:

1. Terminal–based applications
   a) **Terminal** or Terminator (linux)
   b) **Terminal** (Mac; in *Applications/Utilities* folder)
   c) ipython qtconsole
2. Notebook interface (linux & Mac)

   a) **jupyter notebook**
   b) jupyterlab (in deveolpment)
3. Python-aware text editors
   a) **Geany** (linux)
   b) **Atom** (linux, Mac)
   c) **TextMate** (Mac)
   d) vim (linux, Mac)
4. Integrated development environments
   a) **Spyder**
   b) Eclipse
   c) Eric
   d) PyCharm

Of these four categories, the first two (Terminal–based and notebook interfaces) are typically used for developing and testing code interactively, and the last two (Text-editors and Integrated Development Environments) are typically used for larger, more involved programs (although, many development environments include a python terminal or an interactive qtconsole as part of the environment).

We'll begin by discussing the `Terminal`, the `jupyter qtconsole` and the `jupyter notebook` variations first, as they provide the simplest way to begin to use Python. The jupyter notebook (see http://jupyter.org[1]) has become the most popular way to interact with and develop python code, as it allows the integration of LaTeX markdown, graphics and code all in one browser-based interface very similar to packages such as *Mathematica*. In addition, one there are now the beginnings of Gitbub (version control) integration (see https://github.com/jupyter/nbdime) for jupyter notebooks; a feature very useful for collaborative development. The Jupyter project name is an acronym stemming for the main languages that the project is designed to integrate: **Ju**lia, **Py**thon, and **R**. It is also a nod to Galileo's publication about the discovery of the moons of Jupiter, one of the first scientific publications that included the data along with the analysis. Enabling scientific reproducability is one of the main thrusts of the Jupyter project.

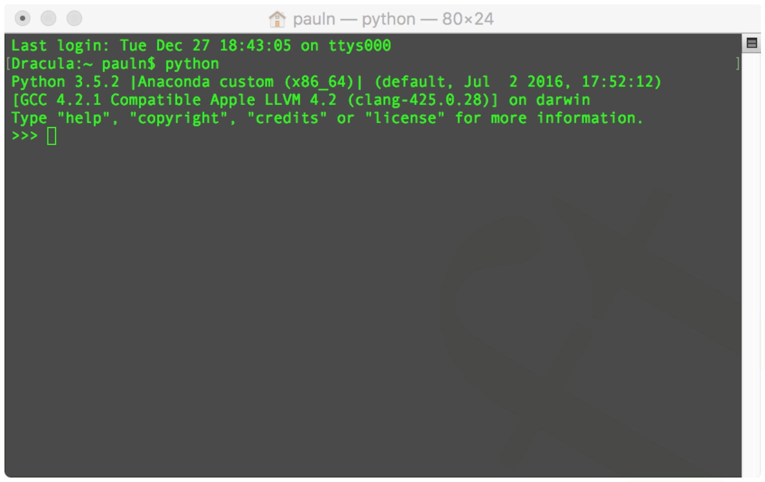
2.1.1 Terminal–based Python

Both the Mac and Linux operating systems come with a `Terminal` application, which you should find and open. You'll be presented with the terminal window (likely running a Bash shell) as shown in Figure 2.1: Once opening the terminal window, you see I've typed

```
python
```

which then shows the user the version of python being used and displays >>> indicating that Python is ready to accept commands, and is an interactive mode. This interactive mode in the terminal is limited to displaying textual output in the terminal

[1] for information about the logo and the project goals see the page https://github.com/jupyter/design/wiki/Jupyter-Logo



```
Last login: Tue Dec 27 18:43:05 on ttys000
Dracula:~ pauln$ python
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Fig. 2.1: Opening a terminal window and starting Python

(but can create a separate plot window), so the terminal is the most primitive way to interact with python. To exit the Python interpreter, type:

```
>>> quit()
```

and you will be returned to the Bash shell window. As the most primitive manner of interacting with Python, you might (rightly) ask why this is worth using.

First, the terminal window (or shell) allows one to interact with the file system (this has nothing to do with Python, but everything to do with viewing, moving, listing, and manipulating items on your computer); some of the common commands you'll likely use are shown in Table 2.1. Second, the terminal window is used to invoke the `jupyter notebook`, and is often integrated into a text editor (like Geany) or used by a text editor to run an extended python script, so it's worth learning as tool in it's own right, and is necessary to get things done on Linux and the Mac.

2.1.2 Jupyter qtconsole

A second way to interact with Python is through an jupyter qtconsole. To start it, you open a terminal window, and type *jupyter qtconsole*; when you do this, a separate python terminal will launch, as shown in Figure 2.2:

Notice that the jupyter qtconsole actually invokes iPython (Interactive Python) and has so much more capability than the simple Bash terminal, that it's usually a much better choice for interacting with python. Many of the Bash terminal commands are able to be used within an ipython terminal via so-called "magic" commands as shown in Table 2.2. For example, `ipython` knows if you type `ls` that you want to print a listing of the current directory, or `pwd` means that you want to see what the current directory is. Simply typing `magic` in an `ipython` terminal will give you more information about magic functions.

Table 2.1: A partial list of commands in the Bash Shell. Many more can be found by performing a web search for "Linux Bash Shell Cheat Sheet" or by going to http://freeworld.posterous.com/.

command	Description
`man <command>`	interactive help about `<command>`
`ls`	list files & folders in current directory
`ls -a`	list all files & folders, even invisible files
`ls <folderName>`	list files in `<folderName>`
`cd <folderName>`	change directory (cd) to `<folderName>`
`cd ~`	change to user's home folder
`cd /`	change to root directory
`cd ..`	cd by going up in the folder hierarchy
`pwd`	print working directory
`python`	invoke python within the Bash shell
`jupyter notebook`	jupyter notebook w/inline graphics
`jupyter lab`	invokes new customizable notebook interface (alpha)

A particularly very useful magic command is the python `timeit` command which is useful to checking the speed of a line or section of code. For example, in an ipython terminal it is easy to see that for individual evaluations of the cosine (presumably for the other trig functions too?) the math library's cosine function is roughly seven times faster than the cosine function in the NumPy library. However, instead of an individual evaluation, if you had a list of say 1000 values, the numerical python (NumPy) library can caclulate the cosine roughly seven times faster than bare python code using the math library. This is because numpy is optimized to be extremely fast at dealing with array objects.

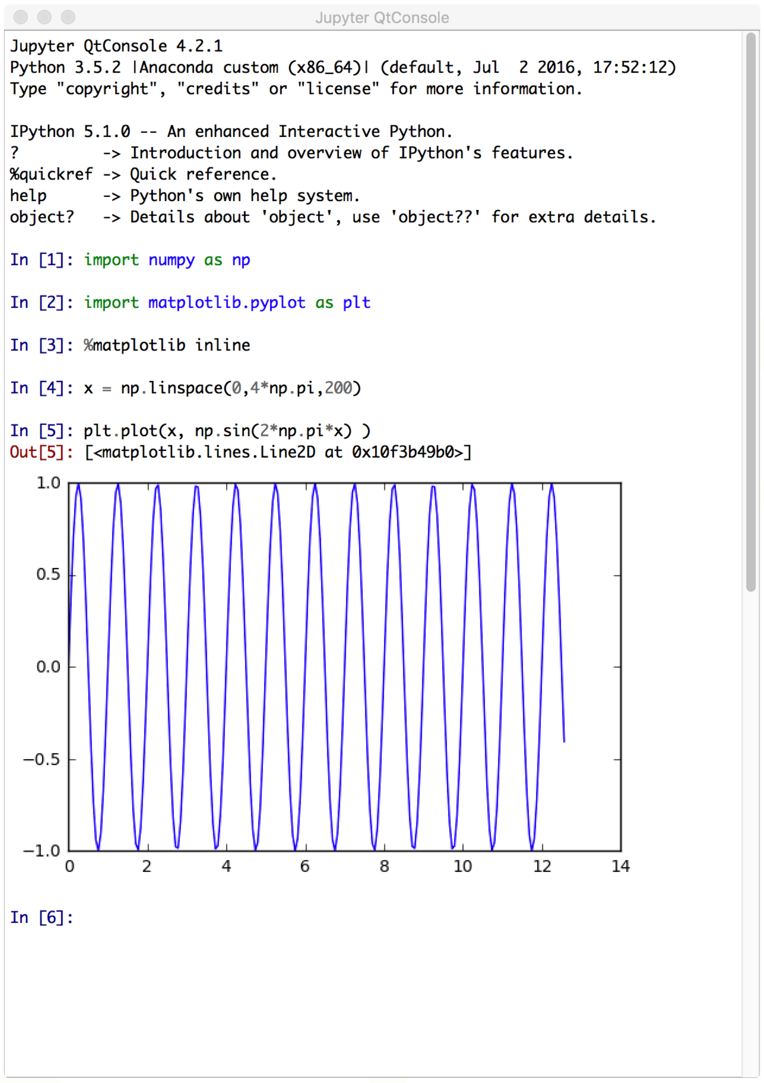
```
In [1]: import math as m

In [2]: import numpy as np

In [3]: %timeit m.cos(3.4)
10000000 loops, best of 3: 80.2 ns per loop #(MacOS 10.12.2)
10000000 loops, best of 3: 75.5 ns per loop #(Ubuntu Linux 16.10)

In [4]: %timeit np.cos(3.4)
1000000 loops, best of 3: 534 ns per loop #(MacOS 10.12.2)
1000000 loops, best of 3: 450 ns per loop #(Ubuntu Linux 16.10)
```

For purposes of a beginning course, it is good to know *how* to time a snippet of code, but for the most part, it pays to first exert efforts to make clean, *understandable and well-commented* code. In many circumstances, one simply doesn't care about obtaining the most efficient code; you're simply interested in obtaining an answer, and waiting a few more nanoseconds isn't so critical. It is only in more serious multiparticle simulations where this becomes more of an issue, and here could live an entire course devoted to algorithms to speedily execute a task.



```
Jupyter QtConsole 4.2.1
Python 3.5.2 |Anaconda custom (x86_64)| (default, Jul  2 2016, 17:52:12)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt

In [3]: %matplotlib inline

In [4]: x = np.linspace(0,4*np.pi,200)

In [5]: plt.plot(x, np.sin(2*np.pi*x) )
Out[5]: [<matplotlib.lines.Line2D at 0x10f3b49b0>]
```

```
In [6]:
```

Fig. 2.2: The `jupyter qtconsole` interface. Notice that it's easy to make a plot, and the *%matplotlib inline* places the graphics within the terminal window.

It is interesting to note that in the above code snippet, the tasks were actually faster running in a Linux virtual machine than natively on the MacOS. So, don't think that running Linux in a virtual machine means for slower execution.

Table 2.2: A partial list of the iPython "magic functions" commands. There are two types of "magic" commands, line-oriented and cell-oriented. Line oriented magic functions are preceded by a single % character and work like standard terminal commands where the argument of the command is the rest of the line following the command. By default, "automagic" is turned on, and you do not need to use the % sign for single line magic commands. Cell magic functions are preceded by a double %% and use as their argument the rest of the lines in the cell. See http://ipython.org/ipython-doc/stable/interactive/tutorial.html for more information.

command	Description
`magic`	get help on magic functions; use "q" key to exit
`lsmagic`	lists all the iPython magic commands
`%run <scripy.py>`	run python script `<script.py>`
`%timeit <python command>`	time a python command
`%%python3`	run cell body using Python 3
`%load <webAddress>`	load a python script from a web address
`ls`	list files & folders in current directory
`pwd`	print working directory
`cd <folderName>`	change directory (cd) to `<folderName>`

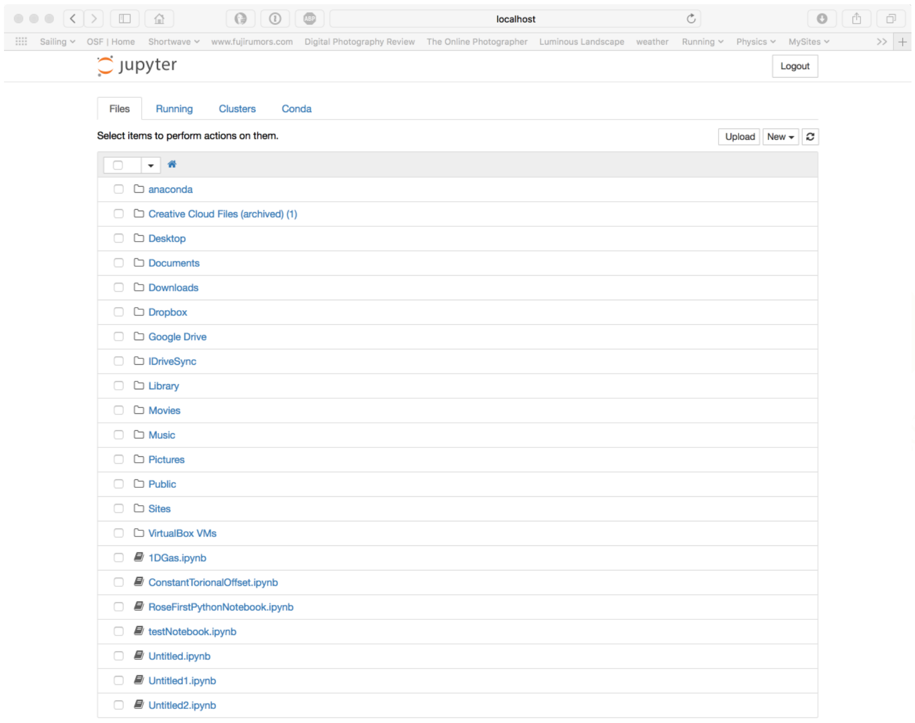
2.1.3 Jupyter notebook

The third way of interacting with python is the `Jupyter notebook`. This is one of the newest additions to python, and executing the command

```
jupyter notebook
```

in a Bash terminal will open up a new notebook in your default browswer and display a Dashboard allowing you to create a new notebook or open a pre-existing notebook file as in Figure 2.3. Here's an example of the Jupyter notebook with a simple plot of a sine wave (see Figure 2.4): Now, instead of entering python commands line-by-line, one can enter entire scripts within one cell and have the freedom to edit them in place. Cells with python code are executed by typing Shift-Enter (just like Mathematica). Because the Jupyter notebook runs in a browswer, not only can it execute python cells, but one can add straight textual descriptions, multimarkdown text (including LaTeX equations!), as well as python cells which can can incorporate graphics, images, youTube movies, and even an entire web page within an notebook cell.

This relatively new interface is highly useful, as the notebook can essentially contain everything (and more) than a traditional paper book can. The exception is that text formatting is somewhat primative (for instance, you cannot change the default width of a text cell), but this is a known deficiency. As a project in active development, these features will likely improve. The notebook interface provides a useful way to distribute text, code, and output in an all-inclusive `.ipynb` format. In fact, there are



Select items to perform actions on them.

anaconda
Creative Cloud Files (archived) (1)
Desktop
Documents
Downloads
Dropbox
Google Drive
IDriveSync
Library
Movies
Music
Pictures
Public
Sites
VirtualBox VMs
1DGas.ipynb
ConstantTorionalOffset.ipynb
RoseFirstPythonNotebook.ipynb
testNotebook.ipynb
Untitled.ipynb
Untitled1.ipynb
Untitled2.ipynb

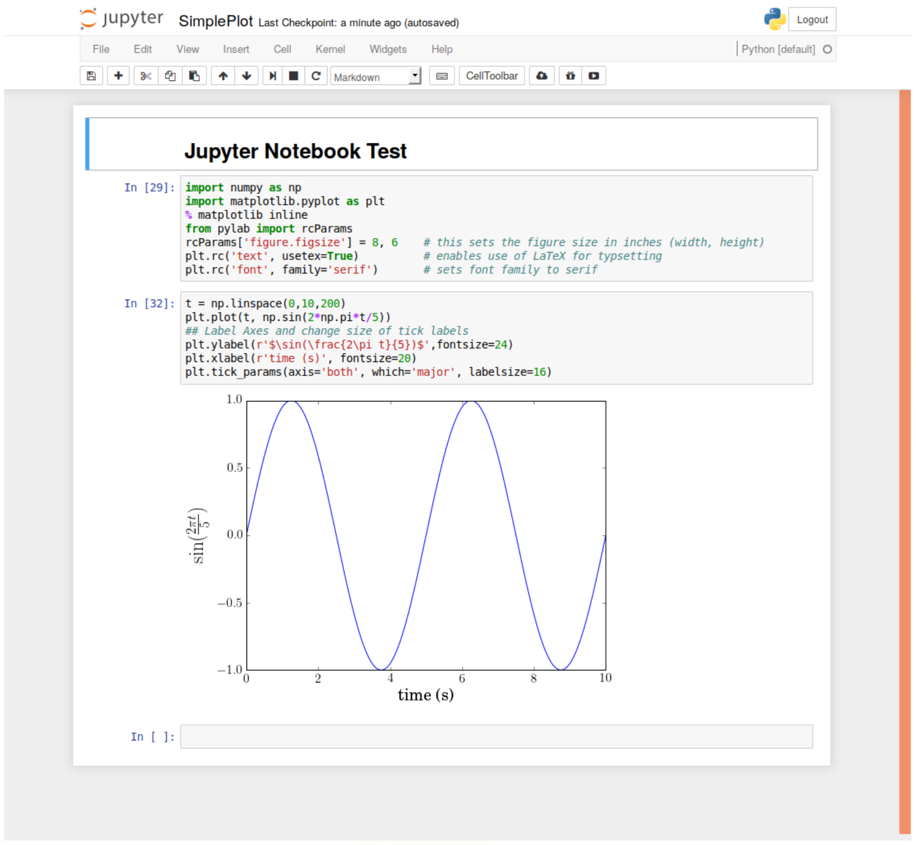
Fig. 2.3: The Jupyter notebook Dashboard.

already utilities to convert `.ipynb` files to html, pdf, and there has been even success at using an ipython notebook to automatically post to a blog.

Here's a link to and ipython notebook that shows some of the capabilities of this format: iPythonNotebookIntro.ipynb

2.1.4 Integrated Development Environments

When working on longer, multi file, and more involved pieces of code, it's convenient to use an integrated development environment (IDE). Such an environment is well suited to this task as it incorporates a file browser, a code viewer/editor, a debugger, a python console, and an ipython terminal. The inclusion of the python console and the ipython terminal mean that one can run the code or even write small snippets to test before including them in the main program files.



Jupyter Notebook Test

```
In [29]: import numpy as np
         import matplotlib.pyplot as plt
         % matplotlib inline
         from pylab import rcParams
         rcParams['figure.figsize'] = 8, 6    # this sets the figure size in inches (width, height)
         plt.rc('text', usetex=True)          # enables use of LaTeX for typsetting
         plt.rc('font', family='serif')       # sets font family to serif

In [32]: t = np.linspace(0,10,200)
         plt.plot(t, np.sin(2*np.pi*t/5))
         ## Label Axes and change size of tick labels
         plt.ylabel(r'$\sin(\frac{2\pi t}{5})$',fontsize=24)
         plt.xlabel(r'time (s)', fontsize=20)
         plt.tick_params(axis='both', which='major', labelsize=16)

In [ ]:
```

Fig. 2.4: The Jupyter notebook interface. An individual cell can contain essentially an unlimited number of python commands. Cells are executed with a SHIFT-ENTER.

For scientific purposes, I think the best open-source option for a full-featured IDE is Spyder, and this is primarily because of the integrated iPython window, and its object inspector that automatically shows docstrings as you type commands in the ipython window. `Spyder` also includes an interactive debugger and can handle large projects with multiple files.

There are also many other python IDEs that are summarized at wiki.python.org; with PyDevelop being a popular Eclipse plugin, and another good IDE (not shown on this page for some reason) is Ninja-IDE.

However, due to the wonderful flexibility of the new Jupyter notebook, I now find myself using Spyder less frequently, as it's easy to prototype in the Jupyter notebook. Programming is really a personal activity, and the key is to find a working environment that you work well with, and this environment can depend on the size of the project you are working on. It's possible to prototype and refine an entire python program in a Jupyter notebook, but it's unlikely that you will complete an extremely complex project in one—for that, you'll likely use an IDE.

2.2 Using Python

2.2.1 Python as an Interactive Calculator

One of the wonderful features of Python is the ability to use it in an interactive fashion, and also, the simplicity of its syntax. For example, suppose you want to perform a simple calculation such as adding two numbers. Open up a terminal window type `python` and type `print 2+5`; you will see the following:

```
>>> 2+5
7
>>>
```

Python simply prints the result of the operation, and the interpreter shows a new prompt indicating it is ready for further input. Notice that we simply have added two integers without having to declare them as such. Python makes an intelligent decision based on whether we input integers or floating point numbers; it assumes that if you do not enter a decimal point, that the number is an integer, and otherwise (with the exception of complex numbers) the number is a floating point (called a `float` in Python parlance). However, python 3.x now executes integer division by assuming you desire a floating point (i.e. Real) result (which, in scientific useage is usually what you want):

```
>>> 2/5  # dividing two integers gives a floating point number
0.4
>>> 5/2  # same thing here.
2.5
```

Notice that we can put a comment within a line of Python input. A # sign starts a comment; anything after this character is ignored. All versions of Python prior to version 3.x assumed that integer division truncates to an integer; we're now free of this in python 3.x.

2.2.2 Higher Mathematics: the `math` library

In physics, we often have more complicated arithmetic, and need to use trigonometric, hyperbolic, exponential, and other functions. To do this, we need to load python's math library (see Table 2.3 for a partial list of math library functions). For example, suppose we want to evaluate the expression $\pi \times 10^7 \cdot \sqrt{90.1}$. This is accomplished as follows:

```
>>> from math import *
>>> pi*10**7*sqrt(90.1)
298203178.477 049 65
```

In order to use the value of π, we have to first load the math library; the statement `from math import *` loads all (the `*` wildcard character tells you that) of the functions from the math library, including trigometric and logarithmic functions. This will be a standard library to include in most of the Python programs in this book. Also notice that Python understands the order of operations, and we did not need parenthesis. The calculation we performed here involves both integers and floating point (i.e. numbers with decimal points) numbers.

Now, if you are being a skeptical student, you will have checked the previous calculation with your calculator, and verified that it appears to agree (actually, your calculator will likely only give you an answer to the 4th decimal place). However, if you perform the calculation in Mathematica 6.0, and C++, you will find that the answers are

```
298,203,178. 477 049 591 064 453 125 (Mathematica 6.0)
298 203 178. 477 049 589 157 104 492  (C++, gnu compiler)
```

Notice that these disagree with the Python calculation in the 16th significant digit. The point here is that Python (and Mathematica, C++, Fortran, etc) have inherent numerical limitations (typically about 16 digits of accuracy for double precision floating point numbers). We have to be careful to not perform calculations where this limitation is significant. Interestingly, there are many physical systems where even this small discrepency in initial conditions (say in the position), will—after not "too long"—result in very different macroscopic outcomes. This subject leads one to the study of chaotic systems, which we will attend to later.

If you want to get more significant figures, one can ask Mathematica to do so, and if you install the python mpmath package (Multi-Precision MATHematics), you can also get arbitrary precision floating point answers. If you do so, you'll find

```
298,203,178. 477 049 639 169 375...  (Mathematica)
298,203,178. 477 049 639 169 375...  (Python mpmath)
```

For this book, we will not be using this arbitrary precision capability, but it is important to be aware that you are limited to about 16 significant figures unless you specifically load the mpmath package. Also, as a high level language with a expanding user base, it is often the case that specialized capabilities exist is some custom package. We will be learing the python packages most relevant to the sciences: numpy (NUMerical PYthon), scipy (SCIentific PYthon), matplotlib (plotting library), and pandas (Python Data Analysis Library).

2.2.3 Python libraries: Loading and getting help

As we have already seen, the core of the Python language is easily extensible by loading libraries (most of which are freely available) There are libraries for mathematics, graphing functions, 3d visualization, and many others. Here, we start by looking more closely at the math library: how to load the library, how to find out about the available functions, and how to use the functions.

Table 2.3 lists some of the functions available in the Python math library; in addition, one can always get complete information about the functions available in a given library by using the `help()` command:

Table 2.3: A partial list of constants and functions in the Python math library.

Function	Description
sqrt(x)	Returns the square root of x
exp(x)	Returns exp(x)
log(x)	Returns the natural log, i.e. ln(x)
log10(x)	Returns the log to the base 10 of x
degrees(x)	converts angle x from radians to degrees
radians(x)	converts angle x from degrees to radians
pow(x,y)	Return x^y; you can also use x**y
hypot(x,y)	Return the Euclidean distance, $\sqrt{x^2+y^2}$
sin(x)	Returns the sine of x
cos(x)	Return the cosine of x
tan(x)	Returns the tangent of x
asin(x)	Return the arc sine of x
acos(x)	Return the arc cosine of x
atan(x)	Return the arc tangent of x
atan2()	Return the arc tangent of y/x.
fabs()	Return the absolute value, i.e. the modulus, of x
floor()	Rounds a floating point number down
ceil()	Rounds a floating point number up

Constant	Description
e	$e = 2.718...$
pi	$\pi = 3.14159...$

```
>>> import math
>>> help(math)
```

I haven't printed the output of the help command here to save space, but you should get familiar with this command, as it is a useful feature of Python that applies to other libraries too. Also notice that in order to get help on the elements of a library, one has to import the library (in this case `import math`) in a different manner than we used in to actually access the functions within the library. An exception to this is that in the ipython notebook invoked by the `--pylab=inline` option, one can simply type `help(math)` in the browser window. In addition, the notebook interface has a dedicated help menu with links to online information about the main python libraries.

2.2.4 Four methods of loading libraries

When importing libraries into Python, we have four alternatives (math library used as an example):

1. `from math import sin`

2. `from math import *`
3. `import math`
4. `import math as m`

Method (1) loads only the sine function, and method (2) loads all of the functions in the math library; the advantage of this method is that it allows us to call a function by its name in the particular library, for example, to calculate the sine of x, we simply type

```
sin(x)
```

Method (3) also loads the entire math library, but now to calculate the sine of x, we must type

```
math.sin(x)
```

This method, although it involves more typing, has the advantage of explicitly addressing the function sin() contained in the math library. For instance, you could also have your own function defined (more on functions soon) which was also called sin() and there would be no conflict between the two. Of course, it would be bad programming practice to define your own function with the same name as a known function in the math library, but if it was important to do so, this alternate method of loading the library would allow it. The funtion sin(x) in the *math* library is said to exist in the math *namespace*.

The fourth method, is simply allows one to have a shorthand method for addressing the math library; now we need only type

```
m.sin(x)
```

to calculate the sin(x) using the math library. Methods (3) and (4) are the preferred way to load libraries, because they remove all ambiguity as to what library a particular funtion belongs to.

However, in order to be able so get help on the contents of the library, we must import the library itself. This is outlined in section 2.2.3.
(a) Start a python session using a terminal window. Import the math library and type `help(math)` as outlined in section 2.2.3. (Do not type `from math import *`) If you are using a terminal window (as opposed to IDLE) you need to know the following: the space bar goes to the next page, and the `q` key exits. Read about the hypot(x,y) command.
(b) Now, to evaluate `hypot(3,4)`, you will have to type `math.hypot(3,4)`. Try it and verify that you get the correct answer.
So, what's the point of this method of importing the math library? It insures that when you want to use a function specific to that library, you have to expressly indicate so; if we type `>>>from math import *`, we have the advantage of being able to address the functions without the `math.func()` notation, but we run the risk (in a sufficiently complicated program) of defining our own function with the same name. Then, we could get unexpected results. A sufficiently cautious programmer would, I suppose, opt to use the safer `import math` style. Another reason that it is sometimes preferable to use this method, is that it explicitly indicates which library the function is from,

which aids in understanding the program or finding documentation (if you import with the * notation, you lose the ability to know which library the function belongs to). This book will mix the two methods, but you are free to use either method in your code.

Problem **??** will give you some more experience with using help(math) as well as exploring some of the functions in the math library. For a partial list of functions in the math library, see Table 2.3.

Other Libraries

There are several Python libraries that we will use in this text; however, since my goal is to get us thinking about physics as soon as possible, I will discuss their usage as we go. For complete documentation on the Python language, see the Python Library Reference. Click on the Library Reference link to see full documentation on each library.

Alternate Help via pydoc

Another way to get help—not within a Python session, but on the command line (i.e. a terminal window)—is with `pydoc`, which is included with Python. Typing `pydoc Z`, where `Z` is any function or module within Python's path, will reveal documentation on that item. For example, opening a terminal window and typing `pydoc math.sin` will reveal documentation on the sine function. When using this in a terminal window, the spacebar forwards one screen, and the `q` key quits.

2.3 Variable types and data structures in Python

There are several main *fundamental* types of *variables* in Python: numbers, strings, and booleans. In the numbers category, we have several sub-types: integers, floating point (i.e. real numbers), and complex numbers.

2.3.1 Assignment Statements; integers, floating points, & strings

Python is a dynamically typed language, which means that you do not need to declare a variable type before an assignment statement; Python will determine the type of variable when you make an assignment. The equal sign (=) is used to assign values to variables; the operand to the left of the = operator is the variable *name*, and the operand to the right of the = operator is the *value* stored in the variable. For example, here are some assignment statements

```
>>> i = 2                        # i is now an integer
>>> type(i)                      # the type statement confirms this
<type 'int'>
```

```
>>> myMass = 75.0                  # myMass is a  floating  point  number
>>> thisClass = 'physics'          #  single  quotes  define  a  string ,  or
>>> dayOfWeek = "Tuesday"          # double  quotes .  Makes no  difference
>>> myMass, thisClass, dayOfWeek
(75.0, 'physics', 'Tuesday')
>>> print myMass, thisClass, dayOfWeek # use of the print statement
75.0 physics Tuesday                     #  will  print  out  values

>>> type(myMass), type(thisClass), type(dayOfWeek)
<type 'float'> <type 'str'> <type 'str'>
```

Notice that when typing an assignment statement, it is good practice to

1. Leave a space on either side of the = sign (the assignment operator)
2. Use a meaningful, descriptive name for the variable (within reason of course)
3. Begin variable names with a lower case letter and start successive words within the variable name with capital letters.
4. If you don't like this naming convention, make your own, but first
5. Read the Style Guide for Python Code for a thorough discussion by the author of the the Python language.

2.3.2 Variable Naming

Variable names in Python can contain alphanumerical characters `a-z, A-Z, 0-9` and some special characters such as _. Normal variable names must start with a letter. By convension, variable names start with a lower-case letter, and Class names start with a capital letter. It is a good idea to use a consistent naming scheme; for instance, suppose I want to use the name `sillyWalk` to describe some quantity—I start the first word (silly) with a lowercase s and the second word (Walk) with a capital W. This naming scheme is called lowerCamelCase (or `mixedCase`), and is commonly used in Java; according to an entry in Wikipedia, the pythonic way is to use silly_Walk, but I think this is ugly and so does the Style Guide for Python Code, which, incidentally, is a good thing to read at your leisure.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

`and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield`

Be careful of the keyword `lambda`, which is a commonly used variable for an exponential decay constant in many physics problems. But being a keyword, it cannot be used as a variable name. Similar problems occur when importing the math library—for instance, it would be a bad idea to create a variable name of `sin`, as it would conflict with the math library's variable name. This is the reason behind using `import math as m`, so that to refer to the sine function, you would use `m.sin`, and this avoids a possible conflict with a user-defined variable of the name `sin`. It's better, of course, to avoid the *possibility* of conflict in the first place.

Python also has several other data types that we will use: Boolean, complex numbers, strings, and lists.

2.3.3 Boolean Integers

A boolean variable in Python is actually an integer; either False (0) or True (1), which you can see if you attempt to use them as in a numerical context. The following examples illustrate the use of booleans:

```
>>> b = 1<2
>>> b
True
>>> b + 1
2
>>> bool(b)
True
>>> bool(2<10)
True
>>> bool(20<=19)
False
>>> bool(2<=2)
True
```

2.3.4 Complex Numbers

Complex numbers in Python are created by one of two methods:

```
a = 1.0 + 2.0j  # you can also use uppercase J if you like
a = complex(1,2)
```

and the real and imaginary parts are represented internally as floating point numbers (even if you type them without a decimal point). You can extract the real and imaginary parts and obtain the modulus as follows:

```
>>> z = 3 + 4j
>>> z.real
3.0
>>> z.imag
4.0
>>> abs(z)
5.0
```

Of course, you can easily perform mathematical calculations with complex numbers:

```
>>> z = 3 + 4j
>>> w = 4 - 5j
>>> print(z * w)
(32+1j)
```

2.3.5 Strings

Strings are simply immutable sequences of alphanumeric characters, and in Python, can be enclosed in single or double quotes. Being immutable, you cannot change a string, but you can create a new string by concatenating a string with another character. You can also refer to a specific character by its position in the sequence:

```
>>> x = 'Physics is fun'
>>> x
'Physics is fun'
>>> x[0]      # notice that the first element starts at zero, not one!
'P'
>>> x[5]
'c'
```

You can also easily extract a range of characters; the general format to do this as
`x[start at : stop before: step size ]`,
but notice that the format is assymetrical in that the `start` index is inclusive, and the `stop` index is exclusive. This is why I've described the format as `start at` and `stop before`. If you leave off the `step size`, it is assumed to be numerically equal to one.

```
>>> x[0:2]
'Ph'
>>> x[1:2]
'h'
>>> x[ : :1]
'Physics is fun'
>>> x[ : :2]
'Pyisi u'
```

You can also add a character to a string in a straightforward intuitive manner:

```
>>> y = x + '!!'    # creates a new string with added exclamation points
>>> y
'Physics is fun!!'
>>> z = y[ :-2] # creates a new string which is every character from
>>> z         # y except the last two characters.
'Physics is fun'
>>> x + '!'
'Physics is fun!'
>>> x         # notice that x is unchanged! Strings are immutable.
'Physics is fun'
>>>
```

Being able to add a character(s) to a string is especially convenient when writing a series of output files with slightly different names, for instance a sequence `plot01`, `plot02`, etc.

2.3.6 Lists

A list is a compound data type composed of several comma-separated values enclosed by square braces; the individual elements need not be of the same data type:

```
>>> misc=['silly', 8, 2.0, 3.0 + 4.0j]
>>> misc[0]          # extracts first element of misc
'silly'
>>> misc[1]*misc[2]  # you can multiply elements together if appropriate
16.0
>>> misc[1]*misc[3]  # even this is okay
(24+32j)
>>> misc[-1]  # displays last element
(3+4j)
>>> new=misc + ['walk', 3.14] # create new list
>>> new
['silly', 8, 2.0, (3+4j), 'walk', 3.1400000000000001]
>>> len(new)  # the number of elements in the list
4
```

You can also use format statements to control how variables and strings are displayed when printed (see Section 2.8.2 for an example, and Table 2.5 for a list of formatting specifiers):

```
>>> # you can use C-style format strings:
>>> value = 'The number = %.2f' % misc[2]
>>> print value
The number = 2.00
```

Starting with python 2.7, the recommended manner of formatting is to use the string.format() method; see for reference, Python 2.7 Common String operations.

```
>>> value= 'The number = {:.3f}'.format(misc[2])
>>> print value
The number = 2.000
```

This newer style differs from the C-format with the addition of the curly braces { } and with : used instead of %. For example, `%06.4f` can be translated as `{:06.4f}`.

Lists are important structures in Python; you can create lists with the

`range(start at, stop before, step)`

function, but each argument of `range` must be an integer. Hence, this function cannot produce a list with floating point values by itself (see Section 2.3.7 for how to convert a list of integers to floating point numbers):

```
>>> time = range(0,11,1)          # creates a list from 0 to 10
>>> time      # notice that there are 11 elements in this list!
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> type(time[0])
<type 'int'>
```

and there are funtions for determining the length of lists, the location of and number of occurances of a particular character or digit, etc. See Table 2.4 for a summary of these operations.

Table 2.4: This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, s and t are sequences of the same type; n, i and j are integers. From the Python Standard Library Reference.

Function	Description
`x in s`	True if an item of s is equal to x, else False
`x not in s`	False if an item of s is equal to x, else True
`s + t`	the concatenation of s and t
`s * n, n * s`	n shallow copies of s concatenated
`s[i]`	ith item of s, origin 0
`s[i:j]`	slice of s from i to j
`s[i:j:k]`	slice of s from i to j with step k
`len(s)`	length of s
`min(s)`	smallest item of s
`max(s)`	largest item of s
`s.index(i)`	index of the first occurence of i in s
`s.count(i)`	total number of occurences of i in s

2.3.7 List Comprehensions

I find the term *list comprehension* to be an unfortunately vague descriptor; I'd rather see it called a*list generator*, but we're stuck with this term, as python adopted its use from other programming languages (and there are python objects called generators). I've read discussions of this on Stack Overflow that point to its origination as being derived from the mathematics of set comprehensions, but I like to just think of a list comprehension as a compact notational way to *generate* a list. For instance, suppose that you want to create a list of the first 10 odd numbers. One way to do this is with a simple for loop:

```
>>> odd = []
>>> for x in range(10):
...    odd.append(2*x+1)
...
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In a list comprehention (ahem...*generator*), this is a one-liner:

```
>> odd = [2*x + 1 for x in range(10)]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

If you wanted to make a list of the first 10 odd integers but convert them to floating point, you could accomplish this as

```
>>> oddFloat = [2.0*x + 1 for x in range(10)]
>>> oddFloat
[1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0]
```

List comprehensions are a little confusing at first, but they do provide a lot of power in a single line of code. In fact, it is possible to use list comprehensions to write very difficult to understand code, and one must be careful to balance conciseness with clarity. Furthemore, when we work with floating point lists we will generally use the NumPy (NUMerical PYthon) library, as they are designed to be very efficient and powerful ways to work with single and multi-dimensional lists (i.e. arrays).

2.3.8 Tuples

A `tuple` is essentially identical to a list, except that it's elements are immutable; once you create the tuple, you cannot change the values. You can define a tuple in two ways:

```
>>> tuple1 = ('physics', 'fun', 3e8, 3.141); # semicolon supresses output
>>> tuple2 = "a", "b", "c", "d";
>>> print tuple1, '\n', tuple2    # \n character creates new line
('physics', 'fun', 300000000.0, 3.141)
('a', 'b', 'c', 'd')
```

To access individual values in a tuple or a slice of values, you can use the same notation as in lists. For example:

```
>>> print 'Tuple1[0]: ', tuple1[0]
Tuple1[0]: physics
>>> print 'Tuple2[1:5] : ', tuple2[1:5]
Tuple2[1:5] : (10, 15, 20, 25)
```

However, you may not change individual elements of a tuple—remember, a tuple is immutable:

```
>>> tuple1[0] = 'chemistry' # this is not allowed for a tuple!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

2.3.9 Dictionaries

A `dictionary` is like a `list`, but instead of an implicit index number (from 0 to N), each element in the list is given a keyword. You can think of a `dictionary` as an unordered set of key & value pairs. Within a given dictionary, the keys must be unique. To create an empty dictionary, we use a simple set of open/close braces:

```
elevation = {}
```

Then we could add elements as

```
>> elevation = {}
>>> elevation = {'Washington' : 1917, 'Katahdin' : 1606, 'Schoodic' : 319.1}
>>> elevation
{'Katahdin': 1606.0, 'Schoodic': 319.1, 'Washington': 1917.0}
>>> elevation['Washington']
1917.0
```

Suppose you wanted to create a dictionary that contained the first 10 odd numbers, with the key being the ranking (1,2,3,etc…) and the value being the odd number itself. You can use the `dict()` constructor with a list comprehension to write:

```
>>> oddNums = dict( [(i+1,2*i+1) for i in range(10)] )
>>> print oddNums
{1: 1, 2: 3, 3: 5, 4: 7, 5: 9, 6: 11, 7: 13, 8: 15, 9: 17, 10: 19}
>>> for key in oddNums:
       print oddNums[key]
```

2.4 Flow Control: if, while, for

There are three main ways to control the flow of program execution in Python. We will look briefly at each.

2.4.1 if Statements

The if-statement has the general form

```
       if <expression is true> :
              then execute
              each indented line
       otherwise continue on to next unindented line
```

Here is a simple example:

```
i=10
if i <= 100:  # note colon at end of line
       i=i+1
print i
```

Running the above will print out a result of 11 for i.

Often, a single `if` statement is not sufficient, so Python provides for `if`…`else` and `if`…`elif`…`elif` …structures. The `else` portion is optional, and `elif` is short for `else if`. The logic is fairly straightforward, as this simple example shows:

```
i = 100       #note that one equals sign assigns the value 100 to i.
if i < 100:
       print 'i<100'
elif i==100:  # two equals signs are needed to test for equality
```

```
        print 'i=100'
elif i>100:
        print 'i>100'
else:
        print 'it is not possible to get here!'
```

2.4.2 while Statements

`while` statements are used to iterate over a range of values. The extent of the loop is controlled by indentation, and the loop executes repeatedly until the condition is no longer true. `while` loops have the general format

```
        while <expression is true> :
                execute each indented line
                return to the beginning
                of the while loop to retest
                the condition. When the test fails,
        exit the loop to the next un-indented line
```

Here is a simple example that sums the integers from 0 to 100:

```
        i,sum = 0,0   # we can assign values to i and sum simultaneously
        while i <= 100:
                sum = sum+i
                i = i+1
        print sum
```

This code properly prints out the sum as 5050, which is obviously correct, since there are 50 sets of 101 (1+100, 2+99, 3+98, …).

2.4.3 for Statements

The `for` statement in Python iterates over all of the items in a sequence (which can be a list of numerical values, or even a list of string variables). Typically, for numerical programming, we will make use of the `range()` function as discussed in Section 2.8.2. Here is an example that sums the integers from 0 to 100 using a `for` statement:

```
        i,sum = 0,0
        for i in range(101): # note that range(101) consists
                sum = sum+i   # of integers from 0 to 100
                i = i+1
        print sum
```

Notice that the `range()` function creates a `list`, and that `i` takes on, sequentially, all the values contained in the list. Alternately, one can create a list and loop over the elements in the following way:

```
>>> words = ['the', 'ministry', 'of', 'silly', 'walks']
>>> for i in words:
...    print i
...
the
ministry
of
silly
walks
```

2.5 Other Methods of iterating in Python

The methods for iterating over items in a list described in Section 2.4 are typical methods that look similar in other languages. However, Python provides several other methods to iterate over elements in a list, a dictionary, a character, a file, or ANY suitable item.

For example, suppose that you have a list of grades and what to print out each value; one way to accomplish this is to do the following: (in an iPython Qtconsole window):

```
In[1]: grades = [95, 100, 88, 77, 91]
In [2]: for i in range(len(grades)):
          print grades[i]
 95
100
 88
 77
 91
```

The method above works fine, but is inneficient (why create an unneeded list of index values?) and not as clear as the more pythonic syntax below:

```
In[2]: for x in grades:
         print x
95
100
88
77
91
```

If you want to print the index for each value, then you can either (not great)

```
for i in range(len(grades)):
   print i, grades[i]
```

or, use the enumerate() function to return an iterator that when called returns the successive values of the list along with the index number (better!):

```
for i, v in enumerate(grades):
   print i, v
```

You may also have a character variable, and then one can iterate over the individual characters in a similar way:

```
In [3]: name = 'Curie'

In [4]: for x in name:
          print x
C
u
r
i
e
```

Although iterating over lists and character variables yield output values in the order you expect, the same is not true when iterating over the key in the (key, value) pairs in a dictionary. Dictionaries have no inherent internal ordering of the (key, value) pairs.

```
In [8]: mountains = {'Schoodic Mtn': 319, 'Cadillac Mtn': 466, \
              'Mt. Washington': 1917, 'South Baldface':1087}
In [9]: for mtn in mountains:
   ...:    print mtn
   ...:
Mt. Washington
Cadillac Mtn
South Baldface
Schoodic Mtn
```

What if you want access to the elevation value associate with each (key) mountain? Then python provides an `iteritems` function that returns an iterator over the (key, value) pairs from a dictionary:

```
In [10]: for mtn,height in mountains.iteritems():
    ...:    print '%s is %d (m) high' % (mtn, height)
    ...:
Mt. Washington is 1917 (m) high
Cadillac Mtn is 466 (m) high
South Baldface is 1087 (m) high
Schoodic Mtn is 319 (m) high
```

Notice that when we iterate over a {list, dictionary, etc} that we have no control over accessing a particular item; we simply grab the first value, then the second, etc (and in dictionaries, the user doesn't even know the order used) until the iteration reaches the last element.

2.6 Numpy

In Python, a list is a general purpose container where each element can be of any type. These lists are dynamically typed (i.e. python determines the type of each element of the list at run time) and thus, python lists cannot support the mathematical operations such as dot product or matrix multiplication. To remedy this, the **num**erical **py**thon (numpy) package allows one to work with arbitrary sized arrays. These arrays must contain elements of all one type (integer, floating point, or complex; declared upon creation). Mathematical operations on these arrays are very fast, as the underlying code is written in C.

2.6.1 Numpy vectors

Assuming you've imported numpy in the recommended manner: `import numpy as np`, you can create a vector in a straightforward manner using `np.array()` with a python list of numbers specifying the components of the vector. Here I also print out the type of the object—you can see it's different than a python list:

```
In [1]: v = np.array([1,3,5,7,9])
        print v
        type(v)
Out[1]: [1 3 5 7 9]
        numpy.ndarray
```

We can also find out the size and shape of the array:

```
In [2]: print v.size
       print v.shape
Out[2]: 5
       (5,)
```

In addition, numpy provides a library of statistical functions. For instance, you can easily calculate the minimum or maximum elements, or any statistical info you might care for; here I compute the mean, population standard deviation,

$$S_N = \left(\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

and the sample standard deviation

$$S_{N-1} = \left(\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

(the latter uses the same numpy function (np.std()) but decreases the number of degrees of freedom by one via setting the obscurely named parameter ddof=1 (decreased degrees of freedom); see numpy reference for more info.

```
print "statistics for v:"
print "(max, min): ", np.max(v), "\t", np.min(v)
print "(mean, pop std, sample std): ", np.mean(v), "\t", np.std(v), "\t", np.std(v, ddof=1)
Out[]:
statistics for v:
(max, min): 9  1
(mean, pop std, sample std): 5.0  2.82842712475  3.16227766017
```

Let's create another array:

```
In [3]: w = np.array([2,4,6,8,10])
        print "v : ", v # just to remind you of v's values
        print "w : ", w
Out[3]: v :[1 3 5 7 9]
        w :  [ 2  4  6  8 10]
```

If you multiply two vectors together, numpy multiplies like elements, i.e.

$$v * w = [v_0 * w_0, v_1 * w_1, ..., v_{N-1} * w_{N-1}]$$

and returns another list (notice that this is NOT the dot product):

```
In [5]: print v*w
Out[5]: [ 2 12 30 56 90]
```

to do the dot product, use numpy.dot():

```
In [6]: print np.dot(v,w)
Out[6]: 190
```

2.6.2 Refering to subsets of numpy vectors: slicing

For a numpy vector, we can refer to a particular element by refering to its index number in the vector. For example, in the vector v, the zeroth element is 1:

```
In [7]: v[0]
Out[7]: 1
```

For a vector, we can extract a sub-vector by the `v[lower:upper:step]` notation, where one has to remember that the indices are inclusive at the lower end, but not at the upper end; i.e. [lower,upper). The step default value is 1, but can set to be any integer. One can omit any of the three parameters in the slice. Make sure you understand the next few cells:

```
# another way to think of this: return the first 2 elements of v
In [8]: v[ :2]
Out[8]: array([1, 3])
# absent the lower and upper assumes whole array limits
In [9]: v[ : :2]
Out[9]: array([1, 5, 9])
```

You can assign values to a slice of a vector:

```
In [10]: v[ : :2] = [0, 13, 19]
         print v
         [ 0  3 13  7 19]
```

A negative index counts backward from the first element of the array; i.e. since v[0] = 0, counting leftward 1 indec wraps around to the end of the array and returns 19.

```
In [11]: print v[-1] # last element in v
        print v[-2: ] # last 2 elements in v
Out[11]: 19
        [ 7 19]
```

As a last check, since the vector v now at equal to [0, 3, 13, 7, 19] what would `np.sum(v[1: :3])` be[2]?

2.6.3 Other ways to create vectors

More often, one is faced with the task of creating large arrays with uniformly spaced values. For this purpose, there are two commonly used numpy methods: `np.arange()` and `np.linspace()`. The `arange()` function returns equally spaced values within the half-open interval [start, stop). For integer values of start, stop, step, this function is equivalent to the stock range() funtion built-in to python. But, when using a non-integer step, it is possible that the spacing will not be consistent (but I haven't yet run into this yet).

```
# with only one arg (stop), it is assumed that start=0 and step = 1:
In [12]: np.arange(5)
Out[12]: array([0, 1, 2, 3, 4])

# next line: since stop = 5.0, all array elements will be floating point
In [13]: np.arange(5.0)
Out[13]: array([ 0., 1., 2., 3., 4.])

# if you specify a step size, you must give a starting value:
In [14]: np.arange(0.9, 6.0, 0.10)
Out[14]: array([ 0.9, 1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9,
       2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. ,
       3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4. , 4.1,
       4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. , 5.1, 5.2,
       5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9])
```

Many times it's more convenient to use the `np.linspace()` command, which produces equally spaced points in the closed interval [start, stop] with N points:

```
In [15]: np.linspace(4.0, 20.0, 17)
Out[15]:
array([ 4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
       15., 16., 17., 18., 19., 20.])
```

2.6.4 Random number generation

Now, just for fun, and to use some of numpy's other functionality, let's create a 10,000 element array of Gaussian random numbers with mean = 0, standard deviation = 0.5.

[2] Since the first index is 1, and we read only every 3rd after that, we would sum 3 and 19 to get 22.

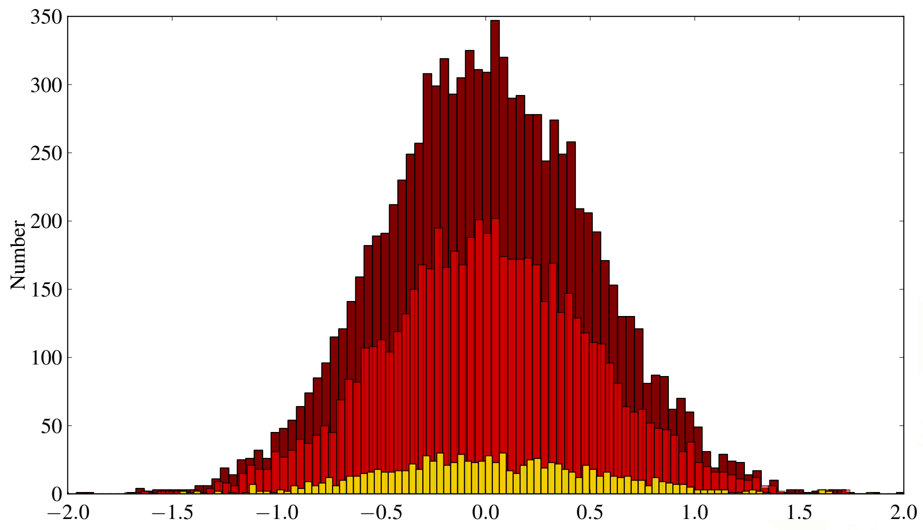


Fig. 2.5: A histogram of 10,000 random numbers picked from a Gaussian distribution. Also shown is a histogram of the middle 6000 points, and a histrogram of the first 1000 points.

We'll then plot the histogram of the values using 100 bins (use matplotlib's hist() plotting function) (see the documentation at scipy.org for more)

```
mean, sigma = 0, 0.5 # mean and standard deviation
s = np.random.normal(mean, sigma, 10000)
print "Average, StdDev = ", np.average(s), "\t", np.std(s)
# now plot a histogram of the full array,
# as well as two histograms of some slices of the array:
plt.figure(figsize=(12,8))
plt.hist(s, 100, color = 'maroon'); # plot the histogram of values
plt.hist(s[2000:8000], 100, alpha = 0.6, color = 'red');
plt.hist(s[4000:5000], 100, alpha = 0.8, color = 'yellow');
plt.xlim(-2,2)
plt.ylim(0,350)
plt.ylabel('Number', fontsize=24)
plt.savefig('histogram.pdf')
plt.show()
Out [16]: Average, StdDev = -0.00311611268417  0.496531757529
```

2.6.5 2d arrays in numpy

Of course, vectors are only one possibility; very often we have to contend with 2 or 3 (or higher) dimensional arrays. Numpy refers to array elements as [row : column]. For example, here is a 4 row, 3 column array:

```
In [17]: test = np.array([[1, 2, 3],[4,5,6], [7,8,9], [10, 11, 12]])
          print test
Out[17]: array([[ 1, 2, 3],
               [ 4, 5,  6],
               [ 7, 8,  9],
               [10, 11, 12]])
```

You can see the shape of the array is indeed (4,3):

```
In [18]: test.shape
Out[18]: (4, 3)
```

2.6.6 Slicing and Masking to select sub-arrays

If we want to know a particular element, then we simply use `arrayName[row,column]` and remember that in Python, the index starts at zero. The following command therefore gives matrix element in the second row of the third column. Bloody confusing to keep this straight in your head, but that's the way it is in Python:

```
In [20]: test[1,2]
Out[20]: 6
```

Here's one way to refer to the whole third row:

```
In [21] : thirdRow = test[2]
           print thirdRow
Out [22]:[7 8 9]
```

But I prefer this notation :

```
In [22] :thirdRow = test[2, : ]
          print thirdRow
Out [22]: [7 8 9]
In [23]: test[ : ] # shows the whole array
Out[23]: array([[ 1, 2, 3],
                [ 4, 5,  6],
                [ 7, 8,  9],
                [10, 11, 12]])
In [24]: firstRow = test[0, : ] # returns first row
         print firstRow
         type(firstRow) # notice it returns another array
Out[24]: [1 2 3]
         numpy.ndarray
In [25]: secondRow = test[1, : ]
         print secondRow
Out [25]: [4 5 6]
In [26]: firstColumn = test[ : ,0]
         print firstColumn
Out[26]: [ 1 4 7 10]
In [27]: chunk = test[1:3] # prints 2nd & 3rd rows (but NOT i=3)
          print chunk
Out [27]: [[4 5 6] [7 8 9]]
```

You can even select a subrange:

```
In [28]: subset = test[1:4, 1:3]
         print subset
Out [28]: [[ 5 6] [ 8 9] [11 12]]
```

Suppose you only want to know what values in the array test are between 5 and 9; you can create a boolean mask and use that to find the values that satisfy this condition:

```
In [29]: mask = (test >5) * (test < 9)
In [30]: test[mask]
Out[30]: array([6, 7, 8])
In [31]: print mask
Out[31]: [[False False False]
         [False False True]
         [ True True False]
         [False False False]]
```

You can find out what the position indices are in the mask using the where() function:

```
In [32]: indices = where(mask)
          print indices
Out[32]: (array([1, 2, 2]), array([2, 0, 1]))
 In [33]: test[indices]
Out[33]: array([6, 7, 8])
```

The diagonal elements can be extracted simply; notice that this operation works even on our (4,3) array:

```
In [34]: print test
          diag(test)
Out [34]:[[ 1 2 3]
         [ 4  5  6]
         [ 7  8  9]
         [10 11 12]]
         array([1, 5, 9])
In [35]: rows=[0,3]; cols = [1,2]
In [36]: test[rows] # extracts only first and forth rows
Out[36]: array([[ 1, 2, 3],
              [10, 11, 12]])
In [37]: test[rows,cols] # but this extracts (0,1) and (3,2)
Out[37]: array([ 2, 12])
```

2.6.7 3d surface plots

Suppose that you want to plot a function such as

$$z = \sin(x)e^{-(x^2+y^2)}$$

over the domain $x : [-2, 2]$ and $y : [-2, 2]$. Here are the steps we need to go through: 1) create a 2d array of values over the given domain 2) evaluate the function over this grid and plot The numpy function that achieves this is `np.meshgrid([x,y])`. What this function does it to take each 1d array and turn it into a 2d array where the value at each array position is the x coordinate.

See Figure 2.6 for the figure that results from the code below:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
##
######  create two 1d arrays that span the x, y limits for plotting.
##
x = np.linspace(-2,2,100)
y = np.linspace(-2,2,100)
###
###### meshgrid does its magic, turns x and y into 2d arrays.
x,y = np.meshgrid(x, y)
z = np.sin(x)*np.exp(-x**2 -y**2)
###
### Plot the Surface and a Contour Plot:
###
fig = plt.figure(figsize=(14,10))
ax = fig.gca(projection='3d')
 #
cset = ax.contourf(x, y, z, zdir='z', offset=-0.4, cmap=cm.coolwarm)
ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.coolwarm,
       linewidth=0.03, antialiased=True, alpha = .85)
 #
ax.set_xlabel(r'$\mathrm{x-axis}$', fontsize=24)
ax.set_xlim(-2.1,2.1)
ax.set_ylabel(r'$\mathrm{y-axis}$', fontsize=24)
ax.set_ylim(-2.1, 2.1)
ax.set_zlabel(r'$\mathrm{z-axis}$', fontsize=24)
ax.set_zlim(-0.4, 0.4)
ax.set_title(r'$z(x,y) = \sin x \cdot e^{-(x^2 + y^2)}$', fontsize=20)
plt.savefig('3dSurface.pdf')
plt.show()
```

2.6.8 Reading and writing data files

Numpy also provides utilities (np.loadtxt(), np.genfromtxt(), and np.savetxt()) for reading and writing data files that are in either comma separated value (CSV) format, or space or tab separated value files (see Numpy input/output for documentation). Numpy also has it's own native data format for saving arrays (.npy) and has routines (np.load() and np.save()) for reading and writing such files.

For CSV, SSV, or TSV files with no missing values, it's easiest to use np.loadtxt(), but for more difficult files that may contain missing values, np.genfromtxt() provides option to deal with missing points.

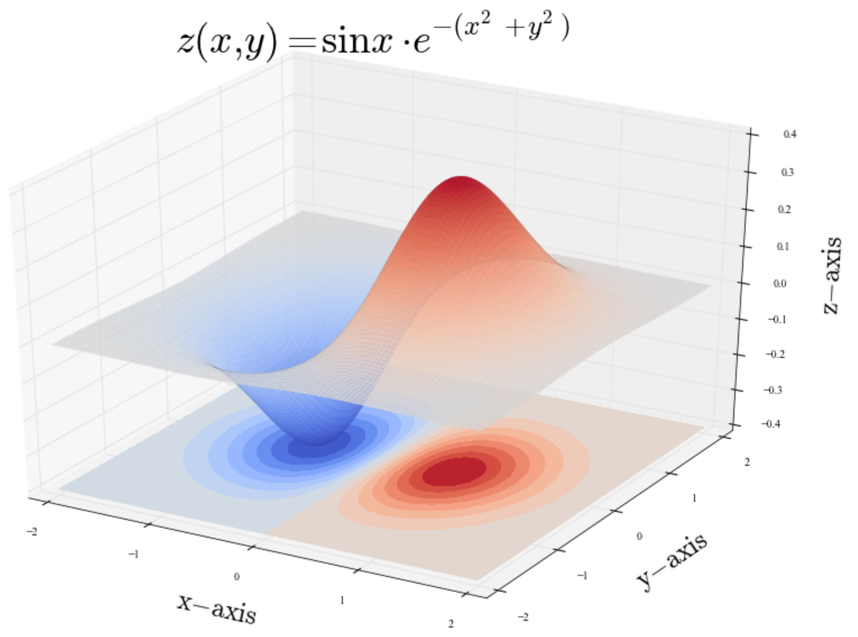


Fig. 2.6: A plot of the function $z = \sin(x)e^{-(x^2+y^2)}$, showing a contour plot below the surface itself.

There is quite a bit more to numpy, and we haven't even touched on some of the features in the SciPy package that builds on numpy. What you should come away with from this is some beginning knowledge of how to work with 1 and 2d arrays, and the idea that numpy and SciPy are powerful packages for scientific computing. If there's some function that deals with arrays or lists of many numbers, it's a good bet that perusing numpy/SciPy is a good place to start.

2.7 First Python Program

Let's get started by writing our first Python program that, while not terribly useful, illustrates how to read input and write output to and from the screen. Typically, this is called a "hello world" program, but we'll make it a little more interesting by having if actually do something a little more complicated.

Our first program will simply calculate the vertical position of a ball thrown vertically upward from the surface of Earth at a time t after launch. For simplicity, we make the standard assumptions of constant acceleration due to gravity and no air resistance. The physics of this motion is simple. The ball's vertical position is given by

$$y(t) = v_0 t - \frac{1}{2} g t^2. \tag{2.1}$$

Each program should be proceeded by an outline (or in more complicated programs, a full–blown flowchart illustrating all of the logical steps) that lays out the steps needed. Here is a simple outline for our program:

1. Read input velocity, v_0 and the time t
2. Compute the position at time t
3. Print out the result

Here is the Python code needed:

```
#!/usr/bin/env python
import sys
from math import *

try:
       v0 = float(sys.argv[1]) # initial velocity
       t = float(sys.argv[2]) # time
except:
       print "Usage:", sys.argv[0], "v0 time"
       sys.exit(1)

y = v0*t-4.9*t**2    #  position
print "ball's position is", y, "meters at t=", t, "seconds"
```

Listing 2.1: Our First Python Program

To create a python program with this code, it is helpful to use a Python aware text editor, so that the editor will perform syntax highlighting and formatting specific to Python. On the Mac, I recommend TextMate, Vi, or Emacs (all open source). Having chosen a suitable editor, create a new file and give it a .py extension, or download the file program1.py from the text website. Decide where you are going to keep your programs, and start off by thinking about a logical scheme. I suggest you create a folder called MyPrograms and place it in your home user folder. Then, since you will likely have many programs in here, it makes sense to have each program in its own subfolder, so that any files or images generated stay organized and associated with the original Python program.

To run the program, open a terminal window (on the Mac, this is in the Applications/Utilities/ folder). Then, before running the program, you have to change your directory (unix command=cd) to your current program folder. To do this, you have two options on the Macintosh:

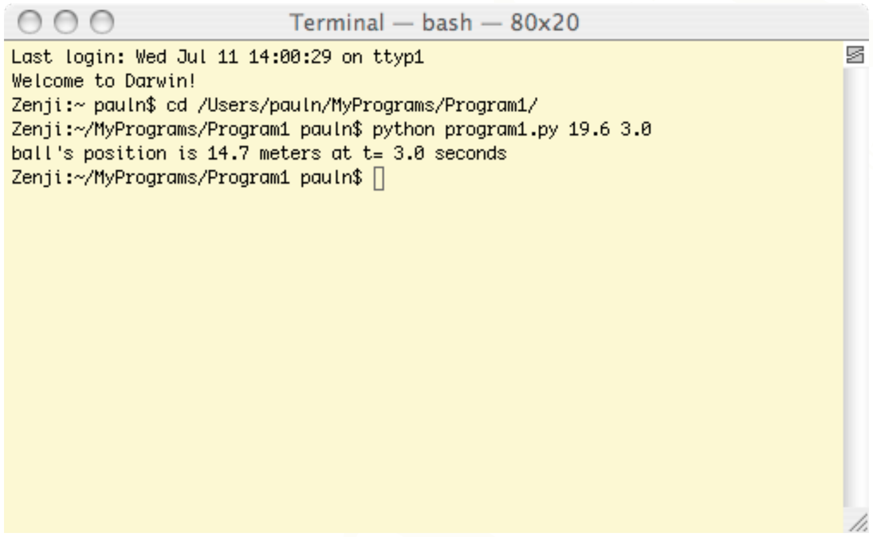
1. With the terminal window open, type `cd MyPrograms/Program1/` followed by the RETURN key.
2. With the terminal open, type `cd ` (with a space after cd) and then drag the folder Program1 (or whatever you have called it) to the terminal window. The Mac OS

will copy the address of the folder to the terminal. Then click on the terminal window and press RETURN.

To run the program with an initial upward velocity of 19.6 m/s and a time 3.0 seconds, type

```
python program1.py 19.6 3.0
```

followed by RETURN. Python returns the answer of 14.7 meters, as seen in Figure 2.7.



```
Last login: Wed Jul 11 14:00:29 on ttyp1
Welcome to Darwin!
Zenji:~ pauln$ cd /Users/pauln/MyPrograms/Program1/
Zenji:~/MyPrograms/Program1 pauln$ python program1.py 19.6 3.0
ball's position is 14.7 meters at t= 3.0 seconds
Zenji:~/MyPrograms/Program1 pauln$ 
```

Fig. 2.7: Running our first Python program

2.7.1 Discussion of Code

The first line imports the `sys` (system) library, and which we use in this program to be able to pass input values for the initial velocity and the time to the program. The second line imports all the functions of the math library (which, in this case we don't need, but I include anyway since most programs will need the math library).

The next section is a `try...except:` block specific to Python which is the standard method for dealing with potential errors. The system library contains a text array called `argv[]`; sys.argv[0] contains the name of the python program (sometimes called a script instead), sys.argv[1] and sys.argv[2] refer to other parameters passed to the program (in our case, v0 and t). When Python encounters a `try...except` block, it attempts to execute the elements in the `try:` block, and, if successful, passes control to what follows the `except:` block.

So, in our case, the program reads the three parameters that you entered when running the program in the terminal: sys.argv[0] is the name of the program (in this case program1.py), sys.argv[1] is defined to be v0, and sys.argv[2] is defined to be t. If an error occurs—for instance, you forget to enter any values for v0 and t when you call the program—then the `except:` block is executed.

In the case of an error the `except:` block prints out a reminder to the user to call the program with two arguments (v0 and t), and then calls another system routine which exits the program.

The last portion of the program is only executed when there are no errors, and that consists of a straightforward calculation of the height of the projectile and a simple print statement. In Python, it is acceptable to use either single or double quotes; this example uses double quotes.

Although this first program is very simple, the `try:...except:` block is a significant chunk of the script, and the program could be made considerably shorter without it; the script would then be written simply as

```
import sys
from math import *

v0 = 19.6           # initial  velocity
t = 3.0             # time
y = v0*t-4.9*t**2   #  position
print "ball's position is", y, "meters at t=", t, "seconds"
```

In this case, the code is clearly more readable, but now, to run the script with different values for the initial velocity and time, the code would have to be edited and saved, and then you would have to execute the script from the terminal once again. Reading the input parameters from the command line using the `try:...except:` block gives one the ability to re-run the code without going through this extra step.

Two other comments about Python that we will encounter as we move forward: in Python, we do not have to end lines with semicolons (as in C and C++), and do not have to use braces to demarcate the extent of structured environments. For instance, in this example, the extent of the `try` and of the `except:` blocks are completely demarcated by the indentation. So, it is critical to be very careful with indentation in Python. It makes for very easy to read code, but forces the programmer to be mindful of indentation when coding. The second comment is that you should get in the habit of using comments to explain your code and make it readable to other users (and yourself). Comments in Python are preceded by a # sign; anything on the line after this character is ignored. Commenting your code achieves several things; it makes you explain your code (which often catches errors in reasoning), and it makes your code easier to decipher, especially when others may not understand your choice of variables.

2.8 Second Python Program

Now that we have written a simple Python program, we are ready to add more to our toolbag. Lets see how to incorporate graphics into our output; we'll modify our program to plot the vertical position of the ball as a function of time. In doing so, we will also learn about loop structures, reading and writing data to files, and the MatplotLib plotting library. Here is the Python script that accomplishes this.

```
#!/usr/bin/env python
# Program 2
"""
Usage:  python program2.py outfile  v0 t dt
This program computes and plots the position vs time for a projectile
launched upward from the surface of Earth. Assumptions: zero air
 resistance; small initial velocity so that the variation
of g with height can be neglected. This program uses the simple 1d
kinematic equations to calculate the height as a function of time.
"""
import sys
import numpy as np
import matplotlib.pyplot as plt

g=9.8          # acceleration at Earth's surface in m/s^2
t=0.0          # initialize time to 0.0
#
#       Now read the input from the terminal:
#       Format: python program2.py 'outfile' v0 tmax dt
#
try:
       outfilename = sys.argv[1]
       v0 = float(sys.argv[2])     # initial velocity (+=up)
       tmax = float(sys.argv[3])   # stop time
       dt = float(sys.argv[4])     # time step
except:
       print "Usage:", sys.argv[0], "outfile v0 t dt"
       sys.exit(1)

outfile = open(outfilename, 'w')  # open file for writing

def height(v0,t):
       if t==0 and v0<0.0:
              print "initial velocity must be >= 0"
              sys.exit(1)
       elif (v0*t-0.5*g*t**2) >=0.0:
              return v0*t-0.5*g*t**2
       else:
              print "ball has hit the ground"
              return 0.0
```

```
# Write Header line as first row in data file
outfile.write('time (s) \t height (m) \n')

# Main calculation loop
imax = int(tmax/dt)
for i in range(imax+1):
       t=i*dt
       y = height(v0,t)
       outfile.write('%g \t %g\n' % (t,y))
outfile.close()

# read in the datafile , & plot it with MatPlotLib:
data = np.loadtxt(outfilename, skiprows=1) # (numpy function)
xaxis = data[ : , 0 ]                # first column
yaxis = data[ : , 1 ]                # second column
plt.plot( xaxis, yaxis, marker='o', linestyle='None')
plt.xlabel('time (s)')
plt.ylabel('Height (m)')
plt.title('Vertically Launched Ball')
plt.show()
```

Listing 2.2: Our second Python program adds graphical output using MatPlotLib.

2.8.1 Running the script

Once you've created a file with the above code, save the file as `program2.py` and then open a terminal window. Change your directory to the folder containing your script; let's assume you place the script in a folder `Program2` which is a subfolder of the `MyPrograms` folder in your home directory. Then type

```
cd MyPrograms/Program2/
```

and then, assuming that you want to create a data file called `trajectory.dat`, and you want to launch the ball upward at 19.6 m/s, plot the vertical position for 4 seconds, and plot points every 0.1 seconds, you would type

```
python program2.py 'trajectory.dat' 19.6 4.0 0.1
```

Python will create the data file, and display the graphic shown in Figure 2.8.

2.8.2 Discussion of the Script

Our second Python script starts by an extended comment (the triple quotes demarcate a special comment called a docstring, which is useful in documenting this piece of Python code. It's a good idea to include a useage statement to show the user how to utilize the script; the docstring will automatically be displayed in an ipython terminal, and can be accessed using help(). Then the script procedes by loading the libraries we will need. The system (sys) library for reading from the terminal and writing data,

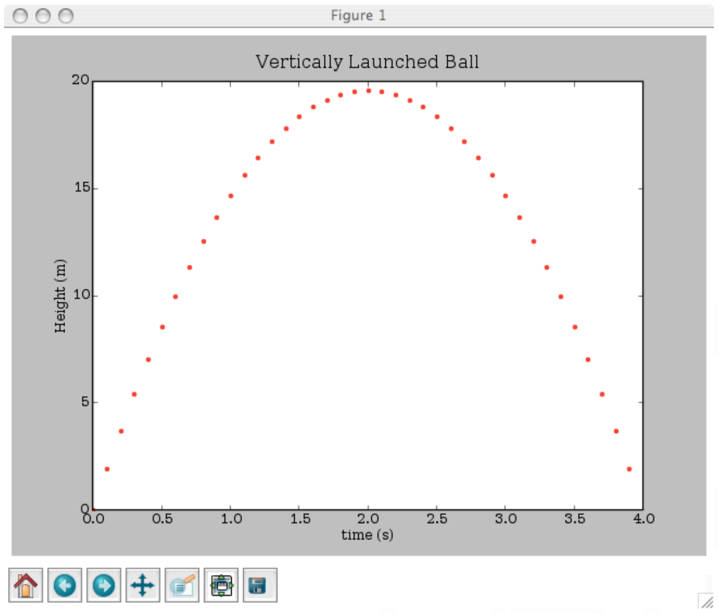


Fig. 2.8: MatplotLib output from our second Python script. Clicking the bottom right-most disk icon at the bottom of the plot will save the figure as a .png file. To return control to the terminal, simply close the window. Explore the other buttons to see some of the interactive features provided with every Matplotlib figure.

and the numpy library for reading the data file, and matplotlib.pyplot for plotting the results.

The next block of code uses a `try:...except:` block to read in the output file name, the initial velocity, the length of time to follow the ball, and the time step.

After reading in these parameters from the terminal, we define `outfile` to open an output file to write data to:

```
outfile = open(outfilename, 'w')
```

where `open` is a system library function that opens creates the file, and `outfile` is an arbitrary user create variable. If there is a need to write multiple output files, then one must create a variable to point to each file. The `'w'` indicates that the file is prepared for being written to; if we wanted to open a file for reading, we would use `'r'` instead of `'w'`. We then define the acceleration due to gravity, and set the start time to 0 seconds.

Now, we introduce a new structure called a function. We define a function called `height(v0,t)` which has two inputs, the initial velocity, and the time. Within this function, we have a decision structure called an `if...else` statement. In our example, if the initial velocity is greater than zero, then the function `height(v0,t) returns` the height of the ball at time t using the standard kinematic result. If the ball's initial velocity is less than or equal to zero, then a statement to this effect is printed to the terminal, and the program exits.

Notice that the extent of the body of the function is demarcated by the indentation; the blank line after `sys.exit(1)` is purely for a visual readability of the code. In addition, indentation also governs the extent of the `if...else` statement. Note that Python executes code sequentially, so that the function `height()` must be defined *before* it is used; for example, it won't do to place this function at the end of the script—Python will give an error if this occurs.

The next line writes the column labels, `time (s)` and `height (m)`, as the first line of `outfile`. Between these column headings is a tab character, represented as `\t`, and at the end is a newline character, `\n`.

The physics in this program is in the main calculation loop. First I calculate the number of steps needed to iterate over given a time of `tmax` and a time step of `dt`.

The main calculation is a loop that uses a `for` statement. This statement tests a condition, and if true, executes the body of the loop (demarcated by indentation, of course). In our case, we use Python's `range()` function, which has three possible forms:

1. range(n) : returns a list of integers from 0 to n-1
2. range(a,b): returns a list of integers from a to b-1
3. range(a,b,dn): returns a list of integers from a to (b-dn) in incements of dn.

For example:

```
>>> range(3)
[0, 1, 2]
>>> range(1,3)
[1, 2]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
```

So, the `for` statement starts with i=0, evaluates the next three lines, then reads the next value of i, executes the three lines again, reads the next value of i, etc. Execution of the loop ceases after evaluating the loop for the i=imax-1, then the output file is closed. So, in our code, in order to plot points from `t=0` to `t=tmax`, we must have our for statement read

```
    for i in range(imax+1)
```

otherwise, we will end up one time step short of the maximum. This (at least to me) is a slight annoyance of Python (an C/C++ too), but we are stuck with this fact that indexing starts from zero in Python. The rest of the `for` loop calculates the time, the height (using our defined function `height()`), and then writes out the time and height to our output data file, one line at a time. Notice that the write statement

```
    outfile.write('%g \t %g\n' % (t,y))
```

consists of two pieces. The first

```
'%g \t %g\n'
```

is called a format string, and it defines that two numbers are to be written to the output file. The first number is a floating point (%g), followed by a tab character (`\t`), another floating point number, and finally, a newline character (`\n`). This format string is inherited from the C programming language; at its most basic level, a format string has the form

```
%<width>.<precision><type-character>
```

where the width and precision are optional arguments, and not all formats (shown in Table 2.5) can accept width and precision arguments. For example, if x=1234.5678 the format string

```
%10.4f
```

indicates that a floating point number with 10 digits will be written with 4 decimal places shown. Since x has 9 characters (the decimal point counts as one character), the above print statement will pad the output with one blank space at the left. On the other hand, the %g format is a *general* format specifier that defaults to a precision (read:# significant figures) of 6. To specify the number of significant figures with the %g format, the width argument is irrelevant, and the precision argument specifies the number of significant figures. So, if x=1234.5678, a Python terminal session will produce the following:

```
>>>print '%10.4f \n' % x
 1234.5678 # note the space at the left
>>>print '%9.4f \n' % x
1234.5678
>>>print '%g \n' % x  # %g defaults to 6 signif. figs
1234.567
>>>print '%.6g \n' % x  # same as default!
1234.567
>>>print '%.7g \n' % x
1234.568
>>>print '%.8g \n' % x  # now the full number is shown
1234.5678
>>>print '%.8f \n' % x  # this will show 8 decimal places
1234.56780000
```

If you ever want to see the result of a particular formatting statement, you can always see the results in an interactive terminal session—one of the benefits of Python over compiled languages. For reference, Table 2.5 shows a list of common format specifiers.

The last portion of the program uses pylab/matplotlib to read the data and plot it to a new window on the computer The `loadtxt` command is from the numerical python (numpy) library; remember, you can get help on this command by opening a terminal window, and typing

```
$ python
>>> import numpy as np
```

Table 2.5: A partial list of format specifiers in Python. For more information, see the Python Documentation, click on Library Reference, and search for String Formatting Operations.

Specifier	Description
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format
E	Same as `%e` except an upper case E is used for exponent.
f	Floating point decimal format.
g	Floating point format. Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise.
G	Same as `%g` except an upper case E is used for the exponent.
c	Single character (accepts integer or single character string).
r	String (converts any python object using repr()).
s	String (converts any python object using str()).

```
>>> help(np.loadtxt) .
```

in a terminal window. In the ipython notebook or an ipython terminal, the help will automatically pop-up as you type the command

```
In[1]: np.loadtxt()
```

The lines

```
xaxis = data[ : , 0 ]              # first column
yaxis = data[ : , 1 ]              # second column
```

define two lists `xaxis` and `yaxis` to be the first and second columns of the array `data`. Note that because python starts arrays with index zero, the first column is column 0. The remaining commands use `Matplotlib` to plot these two lists. Notice that the plot produced comes with a toolbar along the bottom of the display. Your should experiment with them to see what options they present (one of them is to save a copy of the plot to disk). To exit the plot and return control to the terminal, you have to close the window.

There are many more features of Matplotlib; if you are eager to see more, you can see the `Matplotlib` tutorial, and for a complete reference, see the User's Guide at the `Matplotlib` home page. We will learn to use other features of this plotting library as we progress forward.

2.9 Saving Functions as Modules

Although our second program (2.2) is not terribly complicated, as we develop more involved codes, it is good practice to modularize our code. There are two primary ways to accomplish this; one is to use the object–oriented features of Python and another is to split off functions into separate pieces of Python code called modules. For example, we can split the function `height(vo,t)` from our main script, and save it as a separate file; however, it is a good idea to make a small change to the `height` routine by adding an option for the acceleration due to gravity, with g=9.8 m/s^2 being a default value. Here is the modified `height()` routine, saved as `analytic.py` (named both to remind us that this is the analytic solution for the height, and to avoid an awkward function call):

```
def height(v0,t,g=9.8):
       if t==0 and v0<0.0:
              print "initial velocity must be >= 0"
              sys.exit(1)
       elif (v0*t-0.5*g*t**2) >=0.0:
              return v0*t-0.5*g*t**2
       else:
              print "ball has hit the ground"
              return 0.0
```

Listing 2.3: Sections of code can be saved as reusable functions

If we wanted to call the height function from our main program, we have to make sure to place `analytic.py` in the same folder as `program2.py`, and make sure to import it either by
`import analytic`
or
`from analytic import height` (or `from analytic import *`).
Then, to call the function, we have to use either analytic.height(v0,t), or height(v0,t), respectively. Notice that due to the inclusion of g=9.8 in the definition of the function, we do not need to pass the acceleration due to gravity; however, if we wanted to, we could alter the value of g in the function call by, for instance, `height(v0,t,g=4.9)`. Here is the code of Listing 2.2 modified to use our function `height()` which is included in the file `analytic.py`:

```
#!/usr/bin/env python
# Program 2 Modified
"""
This program computes and plots the position vs time for a projectile
launched upward from the surface of Earth. Assumptions: zero air
 resistance; small initial velocity so that the variation of g with
height can be neglected.
Modifications:
07/30/2007: Modified code to have the analytic height calculation
performed in a separate file called analytic.py.
```

```
”””
import sys, analytic
import numpy as np
import matplotlib.pyplot as plt

t=0.0          #  initialize  time  to  0.0
#
#        Now read the  input  from the  terminal :
#        Format:  python program2.py ’ outfile ’  v0 tmax dt
#
try:
       outfilename = sys.argv[1]
       v0 = float(sys.argv[2])     #  initial   velocity  (+=up)
       tmax = float(sys.argv[3])   # stop  time
       dt = float(sys.argv[4])     # time  step
except:
       print "Usage:", sys.argv[0], "outfile v0 t dt"
       sys.exit(1)

outfile = open(outfilename, 'w')   # open  file  for  writing
# Write Header line  as  first  row in  data  file
outfile.write('time (s) \t height (m) \n')

# Main calculation  loop
imax = int(tmax/dt)
for i in range(imax+1):
       t=i*dt
       y = analytic.height(v0,t)
       outfile.write('%g \t %g\n' % (t,y))
outfile.close()

# read  in  the  datafile ,  & plot  it  with MatPlotLib:
data = np.loadtxt(outfilename, skiprows=1) # (pylab function)
xaxis = data[ : , 0 ]              #  first  column
yaxis = data[ : , 1 ]              # second column
plt.plot( xaxis, yaxis, marker='o', linestyle='None')
plt.xlabel('time (s)')
plt.ylabel('Height (m)')
plt.title('Vertically Launched Ball')
plt.show()
```

Listing 2.4: Our first program made modular.

2.9.1 A further change; all looping done using numpy

Now we will make several further changes:

1. Use numpy to handle the calculation of the times and heights (thereby completely eliminating the need for the height() function).

2. Use numpy to save the data to a file (we already used it to read the file.)

Here's the new code:

```
#!/usr/bin/env python
# Program 2np
"""
Useage:   python program2np.py outfilename v0 tmax dt

This program computes and plots the position vs time for a projectile
launched upward from the surface of Earth. Assumptions: zero air
 resistance; small initial velocity so that the variation
of g with height can be neglected. This code uses numerical python
to calculate the vertical height, instead of using python to loop over
 points one at a time using a for loop.
"""
import sys
import numpy as np
import matplotlib.pyplot as plt

g=9.8          # acceleration at Earth's surface in m/s^2
#
#       Now read the input from the terminal:
#       Format: python program2.py 'outfile' v0 tmax dt
#
try:
       outfilename = sys.argv[1]
       v0 = float(sys.argv[2])     # initial velocity (+=up)
       tmax = float(sys.argv[3])   # stop time
       dt = float(sys.argv[4])     # time step
except:
       print "Usage:", sys.argv[0], "outfile v0 t dt"
       sys.exit(1)

# Main calculation using numpy arrays
t = np.arange(0.0, tmax, dt)
y = v0*t-0.5*g*t**2

# write data arrays to a file using numpy:
# (np.c_ forces arrays to be written as columns)

np.savetxt(outfilename, np.c_[t,y], fmt = '%.3f', delimiter='\t' )

# read in the datafile, & plot it with MatPlotLib:
data = np.loadtxt(outfilename, skiprows=1) # (pylab function)
xaxis = data[ : , 0 ]               # first column
yaxis = data[ : , 1 ]               # second column
plt.plot( xaxis, yaxis, marker='o', color = 'r', linestyle='None')
plt.xlabel('time (s)')
plt.ylabel('Height (m)')
plt.title('Vertically Launched Ball')
plt.show()
```

Listing 2.5: Using numpy to handle calculating, saving, and reading the data.

Notice now that only 2 lines of code are needed to create and fill the time and height arrays, and 1 line to write out both arrays to a file. Furthermore, numpy is much faster than straight python code, so for big data sets, this last method will save considerable time. Also, when you become familiar with the numpy commands, you can quickly interactively open a data file and perform plots in a simple ipython terminal.

2.10 General Guidelines for Programming

Writing a Python script or program is necessarily an individualistic endeavor; those of you just learning the language will clearly write different programs than those who have previous experience. However, There are several guidelines that are good to follow:

- Start each program with a pen and paper outline of its structure. For simple programs, this can be a short bit of pseudo code (just a brief outline of the logical steps the script needs to accomplish); for more complicated programs, you will need to actually create a flowchart that explicitly outlines the many logical steps needed.
- When it comes to writing code, get in the habit of using a logical format; here is a structure suggested by Wesley J. Chun in his book *Core Python Programming*[7]:
  1. Startup line (Unix; `#!/usr/bin/env python`)
  2. module documentation (this is what appears between the triple quotes)
  3. module imports (import statements)
  4. variable declarations
  5. class declarations (we'll get to this later)
  6. function declarations
  7. main body of program
- Comment your code as you write. Ideally, your comments should be sufficient for someone else (assumed to be proficient in Python) to understand your code.
- Strive for clarity in your code. Especially as you are first learning to program, there is a temptation to include fancy programming techniques. **Don't**. After you are sure your code produces reasonable results (see the next item!), then you can (if it is worth the time and effort) optimize your code for speed and add new features.
- Always be skeptical of your program's output and check it by testing it for trivial cases where you know an analytical result. For instance, in our second program, even though we were simply computing a known analytic solution for a vertically launched projectile, notice the values I input were an initial velocity of 19.6 m/s and a run time of 4.0 seconds; a quick calculation reveals that the ball should hit the ground at t=4 seconds, and this is reflected in Figure 2.8. Checking your

program's validity is one of the most important steps in computational physics and a considerable effort should be made to insure that it is working properly before you move on to apply the code to regions that do not admit of analytical results.

- Modularize your code and/or use object oriented programming when possible. Modularization improves your code's clarity as well as providing code that can be used by other programs. As we have seen, separating off functions into modules is very easy in Python. Object oriented programming is also easy to implement in Python, but we leave this to a later chapter.

What chapter?

2.11 First two Python programs in the iPython Notebook

Although Section 2.7 and Section 2.8 appear more complex than necessary for such simple tasks, I began with these examples because they illustrate how to:

1. read input from the command line,
2. write to and read from a data file,
3. save a function as a reusable section of code (important for larger projects),
4. plot the columns from a data file using matplotlib.

With the advent of the ipython notebook, however, things can look substantially simpler. In this section, I will re-work the code in Section 2.7 and in Section 2.8 using an ipython notebook. One of the primary differences with the ipython notebook is that one does not need to interact with a terminal window (other than to invoke it via `{ipython notebook --pylab=inline}` or—if you do not want inline plots—`{ipython notebook --pylab}`); hence input values are merely editted within a notebook cell. A notebook cell can be either text (with varying styles), markdown, or code. To execute a code cell, one uses shift-enter (or shift-return). You can see the first and second programs reworked in an ipython notebook here: firstAndSecondPythonPrograms.ipynb

2.12 Python References

For Python, I recommend that everyone have a copy of Guido van Rossum's book[8] An Introduction to Python handy; this book is available for purchase as a standard paperback, a downloadable pdf file, or is available for reading online. Many more details about Python are clearly covered in his introduction. Guido is the author of the Python language, and is its BDFL (Benevolent Dictator for Life). If you need more detail, see his complete documentation for Python at the Python web site. Keep in mind that although I have only discussed the very basics of the language, Python is a very rich programming language, and if there is something you wish you could do, it's probably possible.

Two other introductory books can are by John Zelle[9] and an excellent introduction and reference by Wesley Chun[7]. At a more advanced level, but very geared toward computational physics is Hans Petter Langtangen's *Python Scripting for Computational Science*[10]. At the writing of this book, the book is was in its third edition. Highly recommended.

These days, however, with a little searching using google, you can usually find the answer to your python question relatively quickly. Often times google finds an answer at the site http://stackoverflow.com/, an excellent Q/A site. Do not underestimate the power of a well-worded google search.

Problems

2.1. Python as a calculator

Use Python interactively to evaluate the following mathematical expressions, and compare to what you would calculate exactly by paper and pencil:
(a) $7.5 + \frac{5}{2}$ (b) $2.0 * (3.0 \times 10^8)^2$ (c) $tan(\frac{\pi}{4})$ (d) $3 \times 10^{-7}\log(1000)$
(e) $\sin(90^o)$ (f) $\cos(\frac{\pi}{2})$ (g) $\ln(e)$

2.2. Matplotlib plots

Sometimes you need a quick plot of a function. In Listing 1.2, I have a simple example of this. Create an iPython Notebook to make the following plots and make sure to use the notebook format's markdown capablilities to discuss your thinking.
(a) Make a plot of $\cos(2\pi t)$ from t=0 to t=4π. Hint: look at the Matplotlib web page and see the screenshots link for examples complete with code.
(b) Let's assume that you want to plot a more complicated function; say

$$y = \sin(t^2); \tag{2.2}$$

use matplotlib to plot Equation 2.2 from $t = 0$ to $t = 2\pi$ with 50 linearly spaced points. Is your plot reasonable? Why or why not? If it's not reasonable, fix the code so that it gives a reasonable plot! Why does one have to be careful when plotting $y = \sin(t^2)$? (c) Now modify your work in (b) to create a new plot which contains two side-by-side plots (using matplotlib's subplot feature); the leftmost plot being the 50 point version, and the rightmost plot being your beautified version.

2.3. Practice with loops, writing to a file, and Matplotlib

(a) Write a simple Python program to print out the Fibonacci series up to some specified maximum integer, N. You may use a python terminal, and ipython terminal, or an ipython notebook to write this.
(b) Now alter the program so that the maximum number N is read from the command line and the Fibonacci numbers are printed out to a file and plotted with Matplotlib. In this case, I want you to write a python script using a `try...except` block (akin to that shown in Section 2.8) and be able to run the code from a python console.

2.4. Deciding if a number is in the Mandelbrot set

Consider a complex number $c = x_0 + iy_0$ and a complex number $z = 0$. Now let

$$z_n = z_n^2 + c$$

and consider the sequence of iterates

$$z_0, z_1, z_2, \ldots.$$

The Mandelbrot Set $\mathcal{M}$ is that set of complex numbers $\mathbb{C}$ such that the magnitude of z_n does not escape to infinity; it can be shown that this is equivalent to requiring that no iterate exceed 2 in magnitude:

$$|z_n| < 2$$

Write a function (called mandelTest(c, nMax) that determines if a complex number c is a member of the Mandelbrot Set after nMax iterations. Your mandelTest(c, nMax) should return a boolean (True or False) and the number of iterations reached (either upon escape or inclusion). For example, if some point does not exceed 2 after nMax=200 iterations, then mandelTest should return True, 200. On the other hand, if the point does escape (say after 50 iterations), then mandelTest should return False, 50.

2.5. Integration of a function

Suppose you need to integrate a function $f(x)$ between $x = a$ and $x = b$. A simple way to do this is to break up the interval into N pieces, and approximate the area under the curve by inscribed trapezoids.

(a) Write a python script to find the area under the curve

$$f(x) = e^{-\frac{x}{5}} \cos(\frac{\pi x}{5})$$

between $x = 0$ and $x = 10$.

(b) Make a plot of the area as a function of the number of inscribed trapezoids. Use a minimum of 5 and a maximum of 100 inscribed trapezoids.

2.6. Plotting data from a text file

Imagine you're in a lab and you've recorded some (x,y) data points in your laboratory notebook and you want to make a plot of this data. Because you are a scientist, you have uncertainties associated with each of these values and you want your plot to include error bars. You can of course, use some scientific plotting program to do this, or you can use python. Suppose you have entered data into a text file and you have the following data (Table 2.6)

Using the "data" in Table 2.6, use numpy and matplotlib to plot the data, complete with errorbars and axes lables. Your result should look like Figure 2.9. Your report should include a duplicate of the Table 2.6, your python code, and the plot created by matplotlib.

2.7. Extracting the data you want from a text file

Here's a problem that you frequently encounter as a scientist: you use a data acquisition system to measure some phenomenon, and although the data that gets written to disk is merely alphanumeric text (technical term is ascii—American Standard Code for Information Interchange for those of you who are interested), the format of this ascii file is not of the form that scientific graphics tools can easily plot.

What do you do?

One option is to import it into a spreadsheet, and use the functionality of a spreadsheet to extract the data points you want. I won't do this, because I don't know enough about spreadsheet functionality—I suspect most physicists are in the same boat on this.

Table 2.6: Imaginary data from your lab notebook; the time uncertainties are all equal to 0.2 s.

time (± 0.2 s)	Temp (C)	ΔT (C)
0.2	8.24	0.9
0.4	13.76	0.8
0.6	17.47	0.7
0.8	19.95	0.6
1.0	21.62	0.5
1.2	22.73	0.4
1.4	23.48	0.3
1.6	23.98	0.2
1.8	24.32	0.1
2.0	24.54	0.1

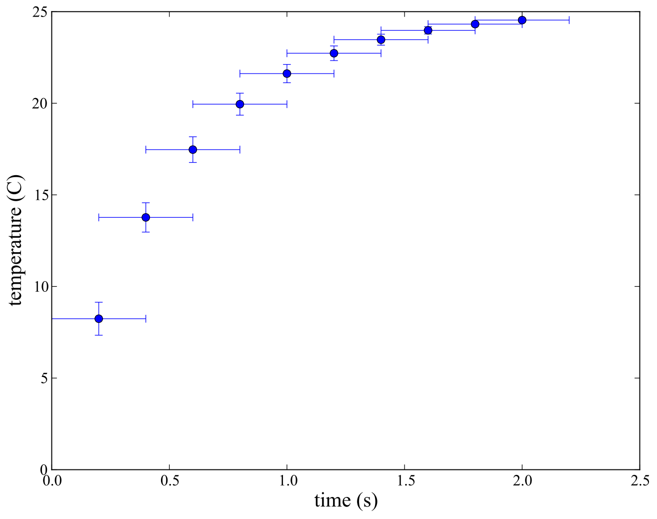


Fig. 2.9: A plot of the data shown in Table 2.6.

Another option is to manually read the data file and type it by hand. This is fine for a small data file, but introduces the inevitable typo(s), and is a totally absurd approach to a data file with millions of data points.

What we need is a way to read in a data file, extract what we need and write out a new data file in a more convenient format; and this method should work equally well for a small file or a data set with millions of points.

An example

Let's make this more concrete with an example. Table 2.7 shows a short section of a 1000 line data file. The data is from an optical switch used to measure the period of a pendulum.

Table 2.7: A short sample from the data set.

time (s)	state	period (s)
0.5389116	1	—
0.6551832	0	
2.2663992	1	
2.3827892	0	
4.0025032	1	3.4635916
4.118984	0	
5.7299832	1	
5.8465832	0	
7.4661176	1	3.4636144
7.5827832	0	

A laser beam is sent through the air to photodiode, and a digital signal is recorded each time the pendulum enters or leaves a laser beam. Figure 2.10 schematically shows the pendulum breaking the laser beam at time t_a, leaving the beam at t_b, breaking it at t_c, and leaving the beam at t_d. The next breaking of the beam at time t_e (not shown) will then allow one to calculate the period as $T = t_e - t_a$.

The problem here is that the format of this data file is not readily readable by many plotting programs. What we'd like to do is filter through this data only extracting the data where we have a period measurement. When you do so, you'll then have a two column data set with time and period values only. Then, it is a simple matter to read the data file and plot it with (for example) a tool like *gnuplot* or (as in Figure 2.11), *Matplotlib*.

Details for this Assignment

1. Read the file periodData.txt, and extract only lines with actual period values. Create an output file called Filtered.dat (which will go in your data folder—see Appendix B for submission guidelines).
2. Plot this data from within your script using Matplotlib. I suggest you go to the Matplotlib gallery, find a simple x-y plot similar to what you want, and examine the code needed to create the plot.
3. Once you create the plot on the screen, you can click the disk icon on the plot to save it to disk (in your `LaTeX/Figures` folder) as a .png or a .pdf file. Alterna-

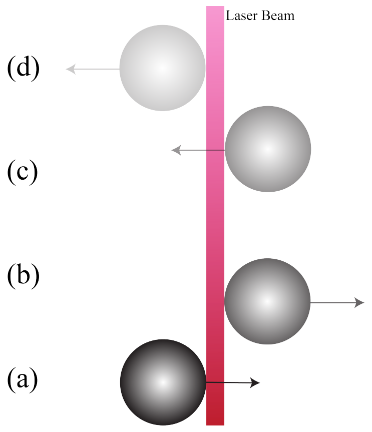


Fig. 2.10. The idea behind an optical switch—each time the pendulum enters or leaves the beam, a digital timing signal is recorded. In the figure, the pendulum is drawn a four different times; on the 5th crossing (not shown), one will have enough information to calculate the period.

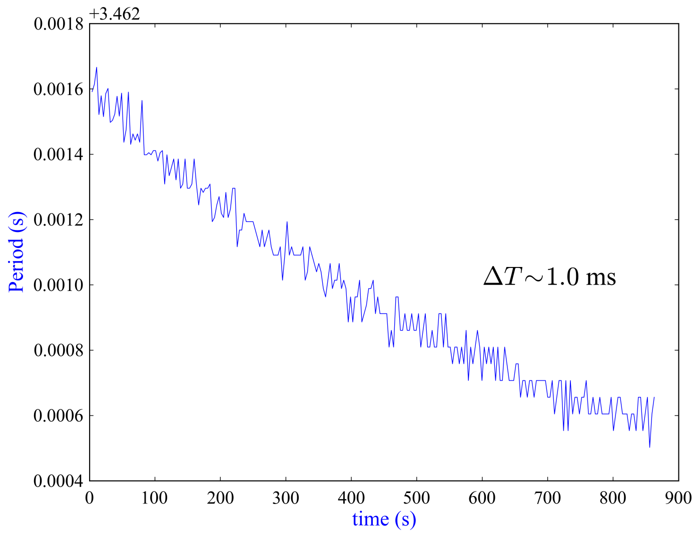


Fig. 2.11: A plot (using Matplotlib) of the period vs time data from the full data set. Notice that Matplotlib automatically pulled out 3.462 s from the period axis on the left, so that the vertical ticks are spaced 0.4 ms apart, and over the course of 850 seconds, the period of the pendulum only changed by about 1 mS.

tively, you can use the Matplotlib savefig() command to save a plot to disk (as usual, google for help on this).

4. As a part two to this exercise, still using the data from periodData.txt (at this link) create a data file with all possible periods extracted; i.e. you can calculate the period as the difference in time between every 4th crossing:

$$\text{Period}_i = t_{i+4} - t_i$$

In this scheme, you'll end up with roughly three times as many period measurements. Create a new output file called `tripleFiltered.dat`, and plot this data file as you did with `filtered.dat`. Keep in mind that you will have to modify your program to produce this data file.

5. Now write a short LaTeX report about what you did. This is **not** a formal report, but simply an exercise to get your Linux/Python/LaTeX feet wet. When you're done, you'll have had experience with the three main tools we'll work with all semester, so the rest of the term will polish and deepen your familiarity with these tools.
6. Don't forget to submit your completed assignment according to the format specified in Appendix B. Your python scripts for part 1 and part 2 should of course be in your code folder.

3

Kinematics in One and Two Dimensions

In almost all introductory physics courses, we begin with kinematics in one and two dimensions. We will begin our study of computational physics similarly, as we can easily check our code in the limiting case of no air resistance and constant vertical acceleration. We will also explore realms that are generally not discussed: motion with linear and non-linear air resistance, and motion with non-constant vertical acceleration.

3.1 Motion in one dimension: Linear Air Resistance

Consider the simplest case of a ball *dropped from rest* from close to the surface of Earth. Assuming that the ball is sufficiently dense, so that we can ignore buoyancy, the main forces on the ball during its downward flight are gravitation and air resistance. If we choose positive y upward, and y=0 at the ground, then Newton's second law tells us that

$$-F_g + F_d = -ma_y,$$

where F_g and F_d are the magnitudes of the gravitational force and the drag force, respectively, and a_y is the *magnitude* of the acceleration of the ball. The free-body diagram is shown in Figure 3.1.

3.1.1 Theoretical Picture

The drag force F_d is actually a complicated force that depends on the shape of the object, its speed, and the density of the air and only in simple situations can we **analytically** calculate its exact form. We begin by working out one such case; we assume that the drag force is proportional to the speed of the ball (this is only good for very small velocities; realistic projectiles dropped from appreciable heights are better modeled by air resistance that is proportional to the square of the speed). Then we can write the equation of motion for the ball as

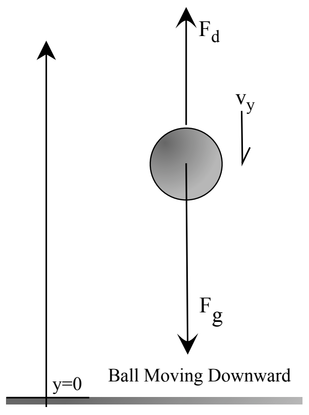


Fig. 3.1. Free body diagram for an object released from rest a short distance above the surface of Earth. Note that the velocity will always be negative while the ball is falling, so that the drag force will always be in the positive direction.

$$-mg + bv_y = m(a_y),$$

where a_y is the acceleration of the ball as it falls. Now, multiplying by $-1/m$,

$$g - \frac{b}{m}v_y = -a_y.$$

Since the acceleration is the derivative of the velocity, and the velocity is increasing in the negative directions, $a_y = -\frac{dv}{dt}$, and we can write this as

$$g - \frac{b}{m}v_y = \frac{dv_y}{dt} \tag{3.1}$$

Then, multiplying by dt and dividing by $g - \frac{b}{m}v_y$, we have

$$\frac{dv_y}{g - \frac{b}{m}v_y} = dt$$

This equation can be easily integrated to yield

$$-\frac{m}{b}\ln\left(g - \frac{b}{m}v_y\right) = t + const$$

or, using the properties of the logarithm, the speed of the ball is

$$v_y(t) = \frac{mg}{b}\left(1 - \frac{A}{g}e^{-\frac{b}{m}t}\right) \tag{3.2}$$

where A is some constant. We determine the constant A by the initial condition $v_y(0) = 0$, and this implies that $A = g$; hence Equation 3.2 simplifies to

$$v_y(t) = \frac{mg}{b}\left(1 - e^{-\frac{b}{m}t}\right) \tag{3.3}$$

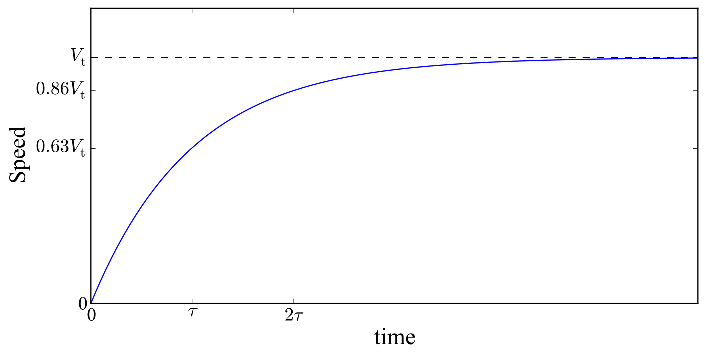


Fig. 3.2: Speed vs time for an object falling in a linear drag regime. Notice that if we define a time constant $\tau = v_t/g$, the object reaches 63% of its terminal velocity after τ seconds. After several time constants have elapsed, the object is very close to its terminal velocity.

If the object falling falls for a sufficiently long time, then the air resistance will continue to increase as its speed increases, and at some point, the air resistance will be equal in size to the gravitational pull on the object; at this point, the net force will be zero, and the object will fall at a constant rate referred to as its terminal velocity, v_t.

Looking at Equation 3.3 in the limit $t \to \infty$, we find that

$$v_t = \frac{mg}{b} \tag{3.4}$$

and hence, the speed of the falling object as a function of time may also be written as

$$v_y(t) = v_t \left(1 - e^{-\frac{gt}{v_t}}\right) \tag{3.5}$$

Notice that if we define a time constant $\tau = v_t/g$, we can write this as

$$v_y(t) = v_t \left(1 - e^{-\frac{t}{\tau}}\right) \tag{3.6}$$

and then, when $t = \tau$, the speed will be $v_y(t = \tau) = v_t \left(1 - e^{-1}\right) \approx 0.63v_t$. After two time constants, the speed will be at $\approx 0.86v_t$. After four time constants, the speed will be within 2% of its terminal value. This can be seen in Figure 3.2

3.1.2 Simulation of Linear Drag

Now, suppose we want to simulate the free fall motion of the ball that we've just analytically discussed. To do so, we begin with Newton's equation of motion for the ball (Equation 3.1)

$$\frac{dv_y}{dt} = g - \frac{b}{m}v_y, \tag{3.7}$$

and multiply both sides by dt:

$$dv_y = \left(g - \frac{b}{m}v_y\right) dt.$$

then, we **approximate** the change in v_y by

$$\Delta v_y \approx \left(g - \frac{b}{m}v_y\right) \Delta t.$$

A more convenient form to write this in is to use the definition of the terminal velocity (Equation 3.4) to write this as

$$\Delta v_y \approx g\left(1 - \frac{v_y}{v_t}\right) \Delta t. \tag{3.8}$$

This is the form we will use to numerically integrate the equation of motion. If we are given the initial velocity in the vertical direction as $v_y(0)$, then a time Δt later, the velocity is

$$v_y(\Delta t) \approx v_y(0) + \Delta v_y,$$

or

$$v_y(\Delta t) \approx v_y(0) + g\left(1 - \frac{v_y(0)}{v_t}\right) \Delta t$$

notice that we are using the value of the known initial velocity to determine the velocity at the next time step. In a similar fashion, the velocity after one more time step is

$$v_y(2\Delta t) \approx v_y(\Delta t) + g\left(1 - \frac{v_y(\Delta t)}{v_t}\right) \Delta t.$$

The value of the velocity at the next time step is the value at the previous step, plus the dervivative $\frac{dv}{dt}$ evaluated at this previous time step times Δt. This is called the **Euler Method**, and is the simplest method for numerically solving a differential equation. Soon, we will see its origins and its limitations; for now, here is a piece of Python code to solve for the speed of the dropped ball using linear air drag:

```
#!/usr/bin/env python
# Program FreeFall_V.py
"""
This program uses the Euler method to solve for the motion of a ball
dropped from rest close to the surface of Earth. The program depends
on the function VelocityLinearDrag (contained in the file EulerFreeFall)
to execute the Euler method.
"""
import sys, EulerFreeFall
import numpy as np
import matplotlib.pyplot as plt

#t=0.0          # initialize time to 0.0
g=9.8           # define acceleration due to gravity
#
#       Now read the input from the terminal:
#       Format: python FreeFallLinearDrag 'outfile' v0 tmax dt
#
try:
       outfilename = sys.argv[1]
       v = float(sys.argv[2])   # initial velocity (+=down)
       vterminal=float(sys.argv[3]) # terminal velocity
       tmax = float(sys.argv[4])   # stop time
       dt = float(sys.argv[5])     # time step
except:
       print "Usage:", sys.argv[0], "outfile vinitial vterminal tmax dt
           "
       sys.exit(1)

outfile = open(outfilename, 'w')  # open file for writing

# Main calculation loop
t = np.arange(0.0, tmax + dt, dt)
vel = np.zeros(t.size) # t.size returns the length of the numpy t array
vel[0] = v
for i in range(1, t.size) :
    vel[i] = EulerFreeFall.VelocityLinearDrag(vel[i-1], dt, 9.8,
        vterminal)

# write data arrays to a file using numpy:
# (np.c_ forces arrays to be written as columns)

np.savetxt(outfilename, np.c_[t,vel], fmt = '%10.3f %10.3f', delimiter=
    '\t' , header = 'time (s) \t speed (m/s)')

# create array for plot of analytic function for comparison:
analytic = vterminal*(1-np.exp(-g*t/vterminal))
plt.plot(t, analytic,'k--')
```

```
# read in the datafile , & plot it with MatPlotLib:
data = np.loadtxt(outfilename, skiprows=0) # (numpy function)
xaxis = data[ : , 0 ]                # first column
yaxis = data[ : , 1 ]                # second column
#plt.ylim(0.0, 0.21)
plt.plot( xaxis, yaxis, marker='.', markersize=5, linestyle='None')
plt.legend(('Analytic Solution', 'Euler Method'), loc='best') # writes
    legend
plt.xlabel('time (s)', fontsize = 18)
plt.ylabel('velocity (m/s)', fontsize = 18)
plt.title('Vertically Dropped Ball', fontsize = 18)
plt.show()
```

Listing 3.1: This program uses the Euler method to solve for the velocity of a ball falling under the influence of a drag force proportional to the speed of fall. It also plots the analytic solution for comparison purposes.

Here is the file EulerFreeFall.py, which contains the functions that implement the Euler method:

```
# File EulerFreeFall.py
"""
This file defines functions needed to implement the Euler method for
one dimensional free fall with air resistance .
"""
def VelocityLinearDrag(v,dt,g=9.8, vt=30.0):
       return (v + g*(1-v/vt)*dt)

def PositionLinearDrag(x, v, dt):
       return (x + v*dt)

def newVelocityNoDrag(v, dt=0.01, g=9.8):
       return ( v - g*dt)

def newPositionNoDrag(y, v, dt):
       return (y + v*dt)
```

Listing 3.2: The contents of the file EulerFreeFall.

For a ball bearing of diameter 1.25 cm falling in glycerin, the terminal velocity is roughly 0.2 m/s (see [11]). If we run the script in Listing 3.1 with

- vinitial = 0.0
- vterminal = 0.2
- tmax=0.2
- dt=0.01

we obtain the output shown in Figure 3.3.

Notice that Figure 3.3 shows the exact (analytic) solution, as well as the simulated solution via the Euler Method. In running the simulation, the time step of 0.01

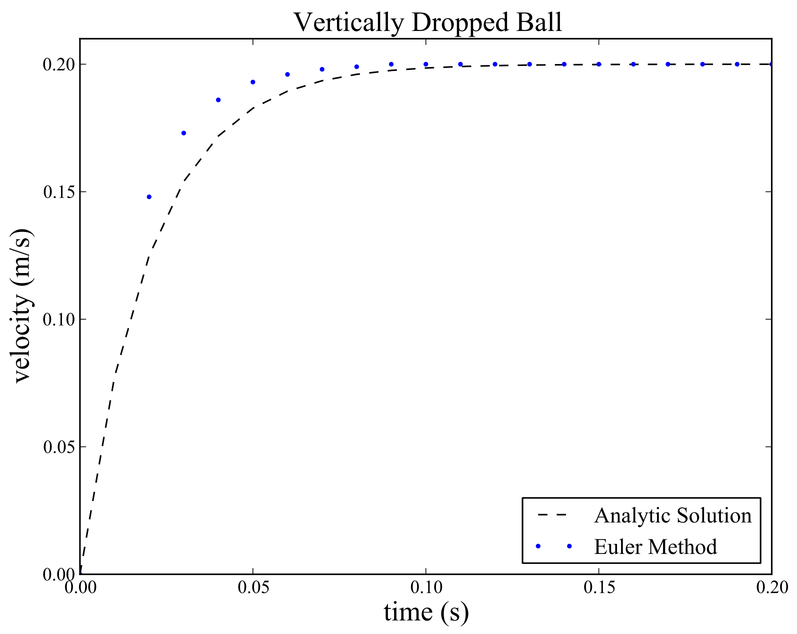


Fig. 3.3: Speed vs time for a 1.25 cm diameter ball bearing falling through glycerin at room temperature. Note that the time step is clearly too large, since the simulation is clearly not a good fit to the analytic solution.

was clearly too large. This is evidenced by the poor disagreement with the analytic solution. In a situation such as this one, where the analytic solution is available, it is easy to make a comparison; however, we typically employ a computer simulation to solve a problem that is **not** analytically solvable. How do we decide on an appropriate time step then?

There are two answers to this question. First, we should always check our simulation's reasonableness by inputing parameters that are analytically solvable. For instance, in the previous problem, we can set the terminal velocity to be very large (ideally infinite, but a large number will do) and check to see that we recover the behavior of a ball falling freely in a gravitational field. Second, we can run the simulation with smaller and smaller time steps until the solutions converge upon one another.

3.1.3 Simulation of Linear Drag: argparse package

For example, here is Listing 3.1 modified in the following manner:

- We use the `argparse` (argument parsing) package to parse (i.e. read in) the options passed from the command line.

- We add the ability to specify several different time steps (in fact, as many as you want!), and the code runs once for each one and plots all of them out, complete with a labeled legend.
- Among the input options is the ability to automatically save a pdf or png of the resulting plot.

The `argparse` package is now part of the python standard library, and is the recommended method of parsing input parameters from the command line.

```
#!/usr/bin/env python
# Program Falling Ball
"""
This program uses the Euler method to solve for the motion of a
 ball dropped from rest close to the surface of Earth. The
program depends on the function VelocityLinearDrag to execute
the Euler method.
"""
import sys, EulerFreeFall
import numpy as np
import matplotlib.pyplot as plt
import argparse
import datetime
t=0.0         # initialize time to 0.0
g=9.8         # define acceleration due to gravity
#
###### READ IN RUN PARAMETERS ###################
###### and Define variables needed   ###################

parser = argparse.ArgumentParser()

# define all options:
parser.add_argument('--filename', dest='filename', \
action="store", help='output filename', default="junk.dat")

parser.add_argument('--v_initial', dest='v', type=float, \
default=0.0, help='initial velocity')

parser.add_argument('--v_terminal', dest='vterminal', type=float, \
default=2.0, help='terminal velocity')

parser.add_argument('--tmax', dest='tmax', type=float, \
default=0.2, help='maximum simulation time')

parser.add_argument('--dt', dest='dt', help='time steps', \
action='append')     # the append action makes a list.

parser.add_argument("--savePlot", dest='savePlot', action="store",\
default='none', help='Save a hardcopy of plot? (specify .pdf or png)' )
```

```
#now read from command line
input = parser.parse_args()
# define the variables we need
filename=input.filename
# set initial velocity :
v=input.v
#save initial velocity for later use (for multiple runs):
vinitial = v
vterminal = input.vterminal
tmax = input.tmax
savePlot = input.savePlot
# saves dt values as a string list called timeSteps:
timeSteps = input.dt
print "output File = ",filename
print "intitial V = ", v
print "terminal V = ", vterminal
print "tmax = ", tmax
print "timeSteps = ", timeSteps
print "savePlot = ", savePlot
##
#####
###################################################
#####

# Main Loop:
for n in timeSteps:
       dt=float(n)# need to convert string into a floating point.
       fname=filename+str(n)+'.dat'
       outfile=open(fname, 'w')    # open file for writing
       # Write Header line as first row in data file
       outfile.write('time (s) \t speed \n')
       outfile.write('%g \t %g\n' % (t,v))
       # Main calculation loop
       imax = int(tmax/dt)
       for i in range(imax+1):
              t=i*dt
              v = EulerFreeFall.VelocityLinearDrag(v, dt, 9.8,
                  vterminal)
              outfile.write('%g \t %g\n' % (t,v))

       outfile.close()
       data = np.loadtxt(fname, skiprows=1) # (numpy function)
       xaxis = data[ : , 0 ]       # first column
       yaxis = data[ : , 1 ]       # second column
       plt.plot( xaxis, yaxis, marker='.', markersize=7, linestyle='
           None')
       t=0.0
       i=0
       v=vinitial
```

```python
# set up plot parameters and plot data

legendstr=[]  #the following four lines set up the legend
for timestep in timeSteps:
       legendstr.append('dt = ' + timestep)
legendstring=str(legendstr[:])
legendstring=legendstring.strip("[ ]") + ', Analytic Solution'
#print legendstring
plt.legend(legendstring.split(', '), loc='best') # writes legend
plt.xlabel('time (s)', fontsize = 18)
plt.ylabel('velocity (m/s)', fontsize = 18)
plt.title('Vertically Dropped Ball', fontsize = 18)
plt.ylim(0, .21)
# if user specified a hardcopy of the file, then save a uniquely named
# copy as either a pdf of png. Then display plot to screen (in all cases).
if savePlot == 'pdf':
    now = datetime.datetime.now()
    fname = str(now.year) + str(now.month) + str(now.day) + "-" +\
      str(now.hour) + str(now.minute) + str(now.second) + ".pdf"
    plt.savefig(fname, dpi=600)
elif savePlot == 'png':
    now = datetime.datetime.now()
    fname = str(now.year) + str(now.month) + str(now.day) + "-" +\
      str(now.hour) + str(now.minute) + str(now.second) + ".png"
    plt.savefig(fname, dpi=600)
elif savePlot == 'none':
    pass

# create array for plot of analytic function for comparison:
a = np.arange(0,tmax,dt)
c = vterminal*(1-np.exp(-g*a/vterminal))
plt.plot(a,c,'k--')
plt.show()
```

Listing 3.3: This program uses the Euler method to solve for the velocity of a ball falling under the influence of a drag force proportional to the speed of fall. You may enter any number of different time steps, and the program will run once for each one. The program also uses the `argparse` package to parse input parameters.

The `argparse` package is designed to parse the input parameters from the command line; We use the previous input parameters to run the program:

- vinitial = 0.0
- vterminal = 0.2
- tmax=0.2
- dt=0.01

and then add two more time steps (dt=0.001, and 0.0001), and save the plot to a pdf file; then we run this program as follows:

```
python FreeFallArgparse.py --filename junk --v_initial 0.0\
--v_terminal 0.2 --tmax 0.2 --dt 0.01 \
--dt 0.005 --dt 0.001 --savePlot pdf
```

and this produces the output shown (I had to add a line ylim(0, 0.21) manually for aesthetic reasons) in Figure 3.4 The convergence of the simulation to the analytical result in this figure is very clear; in fact, a plot of the analytical result lies almost directly on top of the trial with dt=0.001 seconds (at dt = 0.0001 seconds, the analytic and Euler method are essentially identical; try it and see!).

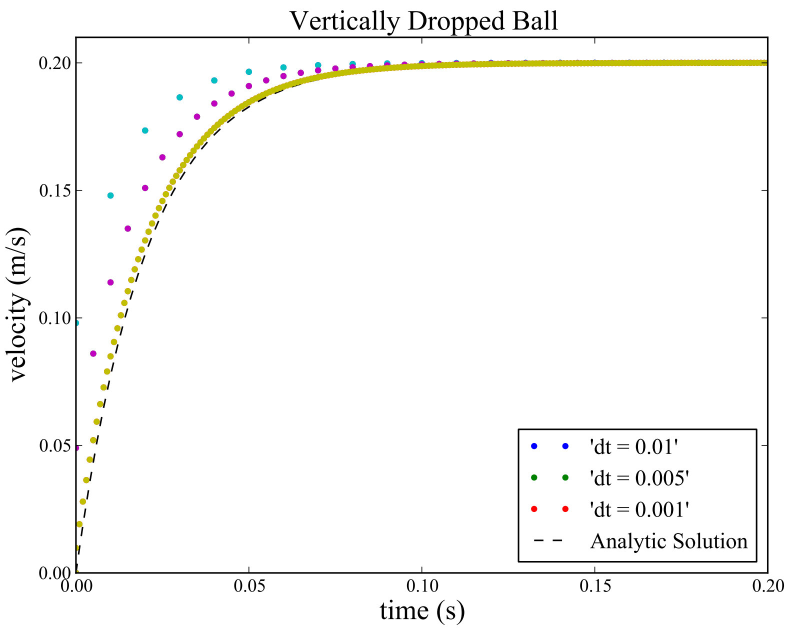


Fig. 3.4: Speed vs time for a 1.25 cm diameter ball bearing falling through glycerin at room temperature; three different time steps are shown.

3.2 Projectile Motion in Two Dimensions

Now that we have solved a few one-dimensional problems, let's work out how to numerically integrate Newton's 2nd law for motion in two dimensions. Building on our work modeling air resistance, let's reason out the physics for projectile motion with quadratic drag.

At high velocities (technically, when the Reynolds number, $\mathrm{R}_e > 1000$), we can write the drag force, F_D on an object moving through a fluid of density ρ as roughly

$$\mathbf{F}_\mathrm{D} = -\frac{1}{2}\rho C_d A v^2\ \hat{v} \tag{3.9}$$

where C_d is the drag coefficient (depends on velocity; 0.07 to 0.5 for a sphere, for example, ≈ 0.04 for a plane, 0.25 to 0.45 for a car), A is the cross-sectional area, and v is the speed for the speed of the object through the fluid.

Notice, of course, that the drag force is always opposite the velocity, so we can draw the free-body diagram on (for example) a sphere as in Figure 3.5:

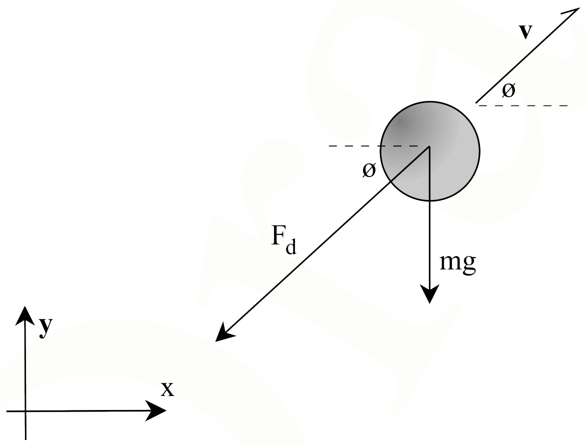


Fig. 3.5: A sphere moving in two dimensions with air drag.

Now, let's consider a sphere traveling through the air with a drag force quadratic in the velocity and of the general form

$$\mathbf{F}_\mathrm{D} = -Bv^2\ \hat{v}.$$

Applying Newton's second law (using a Cartesian coordinate system) and keeping in mind that

$$\hat{v} = \frac{\mathbf{v}}{v} = \frac{v_x\,\hat{\imath} + v_y\,\hat{\jmath}}{\sqrt{v_x^2 + v_y^2}}$$

we see that the drag force is

$$\mathbf{F}_\mathrm{D} = -Bv\ \left(v_x\,\hat{\imath} + v_y\,\hat{\jmath}\right).$$

and therefore Newton's second law in the x and y directions is

$$m\frac{dv_x}{dt} = -Bvv_x$$

and

$$m\frac{dv_y}{dt} = -Bvv_y - mg.$$

We can now write down the governing equations to simulate the motion using the Euler method:

$$x_{i+1} = x_i + v_{x_i}\Delta t$$
$$y_{i+1} = y_i + v_{y_i}\Delta t$$
$$v_{x_{i+1}} = v_{x_i} - \frac{Bv_i v_{x_i}}{m}\Delta t$$
$$v_{y_{i+1}} = v_{y_i} - \frac{Bv_i v_{y_i}}{m}\Delta t - g\Delta t$$

where the speed, v_i is

$$v_i = \sqrt{v_{x_i}^2 + v_{y_i}^2}$$

Notice that to update the x and y velocities, we need to know the value of B/m; so let's calculate this value assuming the simplest case of a sphere of density ρ_s moving through air with density ρ_a. Notice that we can write equation 3.9 as

$$\mathbf{F}_\mathrm{D} = -\frac{1}{2}\rho\, C_d A v^2\ \hat{v} = -B\, v^2\ \hat{v} \tag{3.10}$$

where $B = \frac{1}{2}\rho\, C_d A$ and therefore a sphere with cross-sectional area $A = \pi R^2$, has

$$\frac{B}{m} = \frac{\frac{1}{2}\rho_a C_d \pi R^2}{\frac{4}{3}\pi R^3 \rho_s} \tag{3.11}$$

where ρ_a and ρ_s are the densities of the air and the sphere, respectively. If we make the assumption that $C_d = \frac{1}{2}$ for the drag coefficient, then we have

$$\frac{B}{m} = \frac{3}{16}\frac{\rho_a}{\rho_s}\frac{1}{R} \tag{3.12}$$

and therefore, our set of governing equations to numerically integrate Newton's laws for the sphere using the Euler method are summarized in Equation 3.13. To implement the method, you *first calculate the new positions*, and *then you update the velocities*.

$$
\begin{aligned}
x_{i+1} &= x_i + v_{x_i}\Delta t \\
y_{i+1} &= y_i + v_{y_i}\Delta t \\
v_{x_{i+1}} &= v_{x_i} - \frac{B v_i v_{x_i}}{m}\Delta t \\
v_{y_{i+1}} &= v_{y_i} - \frac{B v_i v_{y_i}}{m}\Delta t - g\Delta t
\end{aligned}
$$

where

$$
\begin{aligned}
v_i &= \sqrt{v_{x_i}^2 + v_{y_i}^2} \\
\frac{B}{m} &= \frac{3}{16}\frac{\rho_a}{\rho_s}\frac{1}{R}
\end{aligned}
\tag{3.13}
$$

3.2.1 The Euler–Cromer Method

The Euler method (pronounced—by the way—as "oil-er") calculates the new positions based on the *old* positions and velocities, and calculates the new velocities based on the *old* values of the position and velocities.

We will find that for periodic motion (this includes planetary motion, and any oscillatory system), the Euler method does not conserve energy, and the simplest modification that conserves energy (over the course of one oscillation) is the Euler–Cromer method. The Euler–Cromer method differs from the Euler method in that the new velocities are calculated first, and then the new positions are calculated with the *new* velocities and the old positions. Hence, the equations and ordering needed to implement the Euler–Cromer method for our problem would look like:

$$
\begin{aligned}
v_{x_{i+1}} &= v_{x_i} - \frac{B v_i v_{x_i}}{m}\Delta t \\
v_{y_{i+1}} &= v_{y_i} - \frac{B v_i v_{y_i}}{m}\Delta t - g\Delta t \\
x_{i+1} &= x_i + v_{x_{i+1}}\Delta t \\
y_{i+1} &= y_i + v_{y_{i+1}}\Delta t
\end{aligned}
$$

where

$$
\begin{aligned}
v_i &= \sqrt{v_{x_i}^2 + v_{y_i}^2} \\
\frac{B}{m} &= \frac{3}{16}\frac{\rho_a}{\rho_s}\frac{1}{R}
\end{aligned}
\tag{3.14}
$$

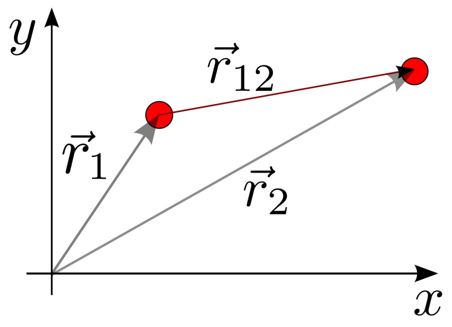


Fig. 3.6. The Cartesion coordinate system we'll use to begin simulating gravitational interaction. The locations of the masses m_1 and m_2 are given relative to some arbitrary origin, and the vector $\mathbf{r}_{12} = \mathbf{r_2} - \mathbf{r_1}$ points from mass 1 toward mass 2.

3.3 Planetary Motion in Two Dimensions

For another aspect of two dimensional motion, we consider the motion of two massive bodies moving in a plane under the influence of Newton's Law of Gravitation, which gives the magnitude of the mutually attractive gravitational force between two masses m_1 and m_2 as

$$F = \mathrm{G}\frac{m_1 m_2}{r^2}$$

where, in the case of spherical masses, r is the distance between the centers of the two objects. In reality, for two arbitarily shaped objects, we need to perform an integration over the volume of each object, a complication that we will not entertain in this chapter.

Figure 3.6 shows the Cartesian coordinate system we will use; we see that we can write the gravitational force *on* mass 1 *due to* mass 2 as

$$\mathbf{F}_{12} = G\frac{m_1 m_2}{r_{12}^2}\hat{r}_{12} \tag{3.15}$$

where

$$\mathbf{r}_{12} = \mathbf{r}_2 - \mathbf{r}_1 = (x_2 - x_1)\hat{x} + (y_2 - y_1)\hat{y} \tag{3.16}$$

and

$$\hat{r}_{12} = \frac{\mathbf{r}_{12}}{r_{12}} = \frac{(x_2 - x_1)\hat{x} + (y_2 - y_1)\hat{y}}{r_{12}} \tag{3.17}$$

thus, putting this all together, we can write (using Newton's 2nd Law: $\sum F_{12} = m_1\frac{dv_1}{dt}$)

$$m_1\frac{dv_1}{dt} = G\frac{m_1 m_2}{r_{12}^3}\left\{(x_2 - x_1)\hat{x} + (y_2 - y_1)\hat{y}\right\}$$

and therefore (cancelling m_1 and using $v_1 = v_{1x}\hat{x} + v_{1y}\hat{y}$) we have

$$dv_{1x} = G\frac{m_2}{\left\{(x_2 - x_1)^2 + (y_2 - y_1)^2\right\}^{\frac{3}{2}}}(x_2 - x_1)dt \tag{3.18}$$

$$dv_{1y} = G\frac{m_2}{\left\{(x_2 - x_1)^2 + (y_2 - y_1)^2\right\}^{\frac{3}{2}}}(y_2 - y_1)dt. \tag{3.19}$$

(3.20)

With the identifications $\Delta v_{1x} \approx dv_{1x}$ and $\Delta t \approx dt$, we rewrite these as

$$\Delta v_{1x} = G \frac{m_2}{\{(x_2 - x_1)^2 + (y_2 - y_1)^2\}^{\frac{3}{2}}} (x_2 - x_1)\Delta t \tag{3.21}$$
$$\Delta v_{1y} = G \frac{m_2}{\{(x_2 - x_1)^2 + (y_2 - y_1)^2\}^{\frac{3}{2}}} (y_2 - y_1)\Delta t. \tag{3.22}$$
$$\tag{3.23}$$

If we go through the same logic for mass m_2, we notice two things: (1) Newton's 3rd Law tells us that the force on m_2 is equal and opposite to the force on m_1 (so $\mathbf{F}_{21} = -\mathbf{F}_{12}$), and (2) when writing down Newton's 2nd Law, the factor m_2 will cancel (instead of m_1) Hence, with these ideas, we can write down the Euler-Cromer method for numerically integrating a two-body gravitational system in 2d.

3.3.1 Euler-Cromer equations for 2 body gravitating system

Here are the equations for mass 1:

$$\begin{aligned} v_{1x}[i+1] &= v_{1x}[i] + G \frac{m_2}{\{(x_2[i] - x_1[i])^2 + (y_2[i] - y_1[i])^2\}^{\frac{3}{2}}} (x_2[i] - x_1[i])\,\Delta t \\ v_{1y}[i+1] &= v_{1y}[i] + G \frac{m_2}{\{(x_2[i] - x_1[i])^2 + (y_2[i] - y_1[i])^2\}^{\frac{3}{2}}} (y_2[i] - y_1[i])\,\Delta t. \end{aligned}$$

and

$$\begin{aligned} x_1[i+1] &= x_1[i] + v_{1x}[i+1]\,\Delta t \\ y_1[i+1] &= y_1[i] + v_{1y}[i+1]\,\Delta t \end{aligned} \tag{3.24}$$

And here are the equations for mass 2:

$$\begin{aligned} v_{2x}[i+1] &= v_{2x}[i] - G \frac{m_1}{\{(x_2[i] - x_1[i])^2 + (y_2[i] - y_1[i])^2\}^{\frac{3}{2}}} (x_2[i] - x_1[i])\,\Delta t \\ v_{2y}[i+1] &= v_{2y}[i] - G \frac{m_1}{\{(x_2[i] - x_1[i])^2 + (y_2[i] - y_1[i])^2\}^{\frac{3}{2}}} (y_2[i] - y_1[i])\,\Delta t. \end{aligned}$$

and

$$\begin{aligned} x_2[i+1] &= x_2[i] + v_{2x}[i+1]\,\Delta t \\ y_2[i+1] &= y_2[i] + v_{2y}[i+1]\,\Delta t \end{aligned} \tag{3.25}$$

Euler-Cromer equations in vector form

With an eye toward implementing the above, we can anticipate the use of a vector for the position and velocity of each mass; with

$$|r_{12}^3| = \left((x_2 - x_1)^2 + (y_2 - y_1)^2\right)^{\left(\frac{3}{2}\right)},$$

we write Equations 3.24 and 3.25 more simply in vector form as

$$\begin{aligned}
\mathbf{v}_1[i+1] &= \mathbf{v}_1[i] + G\frac{m_2}{|r_{12}|^3}\left(\mathbf{r}_2[i] - \mathbf{r}_1[i]\right)\,\Delta t \\
\mathbf{r}_1[i+1] &= \mathbf{r}_1[i] + \mathbf{v}_1[i+1]\,\Delta t \\
&\text{and} \\
\mathbf{v}_2[i+1] &= \mathbf{v}_2[i] - G\frac{m_1}{|r_{12}|^3}\left(\mathbf{r}_2[i] - \mathbf{r}_1[i]\right)\,\Delta t \\
\mathbf{r}_2[i+1] &= \mathbf{r}_2[i] + \mathbf{v}_2[i+1]\,\Delta t
\end{aligned} \tag{3.26}$$

3.3.2 Natural units for the gravitating system

If one would like to begin numerically integrating Equations 3.24 and 3.25 (or, equivalently, Equation 3.26), and you're comfortable with using SI units, then we're really good to go. Then, before plotting the motion, it might be convenient to scale all distances to astronomical units (1 Astronomical Unit (AU) = mean Earth-Sun distance = 1.496×10^{11} m).

Another option—and it's really not necessary, only convenient—is to change all the variable to dimensionless form in the following manner (well use human/solar system – centric units where masses are in solar masses, distances are in AU, time in years, and velocities in AU/yr.); then we cast each variable into dimensionless form as follows:

$$\begin{aligned}
m &= \frac{m}{m_s}\cdot m_s \equiv \tilde{m}\cdot(m_s) \\
r &= \frac{r}{1\,\mathrm{AU}}\cdot(1\,\mathrm{AU}) \equiv \tilde{r}\cdot(1\,\mathrm{AU}) \\
v &= \frac{v}{1\,\mathrm{AU/yr}}\cdot(1\,\mathrm{AU/yr}) \equiv \tilde{v}\cdot(1\,\mathrm{AU/yr}) \\
t &= \frac{t}{1\,\mathrm{yr}}\cdot(1\,\mathrm{yr}) \equiv \tilde{t}\cdot(1\,\mathrm{yr})
\end{aligned} \tag{3.27}$$

I'll leave it as an exercise to the reader (as well as for a problem at the end of this chapter) to substitute these equations into Equation 3.26; then, after simplifying and collecting terms, one finds that the only change is to replace the existing variables with their dimensionless $\widetilde{variable}$ form and that the constant G is replaced by

$$\frac{G\cdot m_s\cdot(1\,\mathrm{yr})^2}{(1\,\mathrm{AU})^3} = 4\pi^2$$

and then we have the following equations with all variables in dimensionless form:

$$\tilde{\mathbf{v}}_1[i+1] = \tilde{\mathbf{v}}_1[i] + 4\pi^2 \frac{\tilde{m}_2}{|\tilde{r}_{12}|^3} \left(\tilde{\mathbf{r}}_2[i] - \tilde{\mathbf{r}}_1[i]\right) \Delta\tilde{t}$$

$$\tilde{\mathbf{r}}_1[i+1] = \tilde{\mathbf{r}}_1[i] + \tilde{\mathbf{v}}_1[i+1]\,\Delta\tilde{t}$$

and

$$\tilde{\mathbf{v}}_2[i+1] = \tilde{\mathbf{v}}_2[i] - 4\pi^2 \frac{\tilde{m}_1}{|\tilde{r}_{12}|^3} \left(\tilde{\mathbf{r}}_2[i] - \tilde{\mathbf{r}}_1[i]\right) \Delta\tilde{t}$$

$$\tilde{\mathbf{r}}_2[i+1] = \tilde{\mathbf{r}}_2[i] + \tilde{\mathbf{v}}_2[i+1]\,\Delta\tilde{t} \tag{3.28}$$

3.3.3 Creating our simulation

To simulate our system, we will save ourselves considerable time by using the capabilities of the numerical python package; in particular, we will define the postion and velocity of each particle as numpy arrays, and will use the linear algebra package's *norm* package to fine the length of $r_2 - r_1$.

```
pi = math.pi
G = 4*math.pi*math.pi
AU = 1.496e11
M_SUN = 1.989e30
M_EARTH = 5.974e24
## define dimensionless masses:
m1 = 1.0
m2 = M_EARTH/M_SUN
## define initial positions and velocities as vectors:
r1 = np.array([0.0, 0.0])
r2 = np.array([1.0, 0.0])
cm = ( m1*r1 + m2*r2 ) /(m1 + m2)
v1 = np.array([0.0, -2.0*pi*m2/m1]) ## this insures that ptotal = 0
v2 = np.array([0.0, 2.0*pi])
dt = 0.001
tmax = 1.0
```

Problems

3.1. Modify the program Falling Ball to correctly deal with air resistance that is proportional to the square of the velocity. This is the air resistance that is a better model for objects such as bowling balls falling from the tops of buildings. The magnitude of the air drag, F_d in this case is

$$F_d = b_2 v^2$$

where b_2 is a constant that (in general) must be determined empirically. If a ball is dropped and allowed to achieve terminal velocity, then the drag force will be equal to the pull of gravity, and we will have

$$mg = b_2 v_t^2$$

and therefore, the constant b_2 will be

$$b_2 = \frac{mg}{v_t^2},$$

which allows us to rewrite the drag force in terms of the terminal velocity:

$$F_d = mg\left(\frac{v}{v_t}\right)^2$$

Let's assume that we have a ball of radius 0.01 m, where the terminal velocity in air is found to be about 30 m/s. Now, following the reasoning in Section 3.1.2, modify the code in program Falling Ball to use quadratic air resistance (also referred to as inertial drag), and compute the speed at which this pebble reaches the ground if it is dropped from rest from a height of 100 m. Compare this speed to a a freely falling object with no air resistance.

3.2. Using an initial velocity of 100 m/s and a launch angle of 30^o, simulate the 2d motion of three different objects. Each scenario is assumed to be taking place at the surface of Earth:

1. A steel ball with density 8000 kg/m^3 traveling through an enormous vacuum chamber.
2. A 2 cm diameter steel ball with density 8000 kg/m^3 traveling through air of density 1.2 kg/m^3
3. A 2 cm balsa wood ball with density 160 kg/m^3 traveling through air of density 1.2 kg/m^3

a) Make sure to test that your simulation for the zero air resistance case agrees with basic kinematics (it goes without saying that you should do this!) and check to see that the other two situations give sensible results.
b) Make a plot of y .vs. x for the three scenarios.
c) Plot the total mechanical energy .vs. time for the three scenarios. Discuss this plot

in your report. Does the Euler method properly conserve energy for the zero air resistance case? Should it? What about the other cases?
Lastly,
(d) modify your program to find the launch angle that gives the maximum range for the steel ball and the balsa wood ball. Assume that the initial launch speed is 100 m/s. Write all this up using the LaTeX report template. Follow the guidelines closely; the quality of the writing in the report is important. I'll reject it and return it to you if it's poorly written, and you'll have to re-submit your report.

3.3. Prove that substituting Equation 3.27 into Equation 3.26 yields Equation 3.28. Then, using SI units show that

$$\frac{G \cdot m_s \cdot (1\,\mathrm{yr})^2}{(1\,\mathrm{AU})^2} = 4\pi^2$$

3.4. Extend the Earth-Sun simulation so that it calculates the
(a) kinetic, potential, and total energy, and the
(b) total angular momentum
of the system, and investigate how the time time step effects the conservation of energy and angular momentum. Create 3 plots:
(i) the orbit (show 5 complete orbits)
(ii) a plot of kinetic, potential and total energy (all three curves in one plot) vs time.
(iii) a plot of total angular momentum vs time.
Show that for time steps that are too large, the orbit is not re-entrant (i.e. it doesn't repeat). What is a reasonable time step to use for the Earth-Sun system?

3.5. This question has two parts:
(a) Calculate (theoretically) the minimum speed the Earth needs to escape from the sun.
(b) Now use your simulation to verify your answer in (a).

3.6. Suppose that you begin with the Earth-Sun simulation (which yields a very nearly circular orbit) and then, after one revolution, you give the Earth a radial kick in velocity of 6 km/s. The orbit will now become elliptical.
(a) Make a prediction (before creating the simulation) about the resulting orientation of the major axis of the ellipse: will the major axis be parallel or perpendicular to the kick direction?
(b) Create the simuation, and see if your intuition was correct.
(c) Repeat the simulation (starting from the circular orbit), but this time give a tangential kick after one year.

A

Guidelines for Reports

A.1 General Overview

The heart of this class is learning to use computers as tools to (a) solve problems and (b) simulate physical systems (typically ones that are too difficult to solve analytically).

When we start becoming familiar with python, we will have some shorter assignments that are geared toward solving some simple problem (say reading an inconveniently formatted data file and re-writing it in a form that graphing programs can understand); for these reports, I'll assume you'll do a sensible writeup that address the main concerns outlined in the statement of the problem. Keep in mind, however, that you should still follow the digital submission guidelines outlined in Appendix B.

For the simulations, I'll want a formal report, and the rest of this appendix will address the content and style requirements for a formal report.

A.2 Formal Reports

1. Introduction: The introduction should give an overview of the problem and an indication of where it fits into the subject of physics.
2. Physics & Numerical Method: Describe the background physics of the problem, and detail the algorithm that is used to solve the problem. This section should also list relevant snippets of your code to show how it is implemented. (A full listing of your code should always be attached as the last section of your report.)
3. Verification: Tell what you did to verify that the program gives correct results; this typically involves showing that your code gives reasonable results for simple cases where an analytic solution is known or obvious. Generally speaking there should be more than one test used to verify program integrity.
4. Results: Present the results of running the program to demonstrate the behavior of the system under different circumstances. Results might be presented in graphical form or as tables, as appropriate. Be sure that results that are presented are labeled properly, so that the reader can figure out what has been calculated and what is

being displayed. Make sure that all figures and tables should have descriptive captions.

5. Conclusion: Present a discussion of the physical behavior of the system based on your simulations and answer any special questions posed in the assignment.
6. Code: Always provide a full listing of your code at the end of the paper. In LaTeX, there is an excellent package called listings that does an wonderful job of formatting code.

B

Digital Submission of Reports

Each project you do this semester will involve several pieces: Code, Images, and LaTeX code. In order to make it easier for me to give timely feedback and to facilitate my record keeping, I am asking everyone to follow a very specific format for both the naming and organization of submitted projects. I will now describe this structure in detail (you'll see that the format is really quite simple); however, keep in mind that submitted projects that do not follow this structure will be returned and will need to be re-submitted and will result in a 10% penalty.

I will be asking each of you to submit your reports electronically as a single compressed file (in Linux or Mac OS, just right click on a folder to bring up a dialog option to compress the folder.)

B.1 Folder Structure

The folder structure is really quite simple, and Figure B.1 shows the general layout; there is a single top level folder, which contains four sub-folders which delineate the major pieces of each assignment. Of these four sub-folders, only the LaTeX folder has one more sub-folder.

After you understand the general scheme for structuring your assignment submission, you can see a specific example in Figure B.2. Please email your assignment to me at `pauln@maine.edu`, with subject line: `PHY261 Assignment 01` (for example).

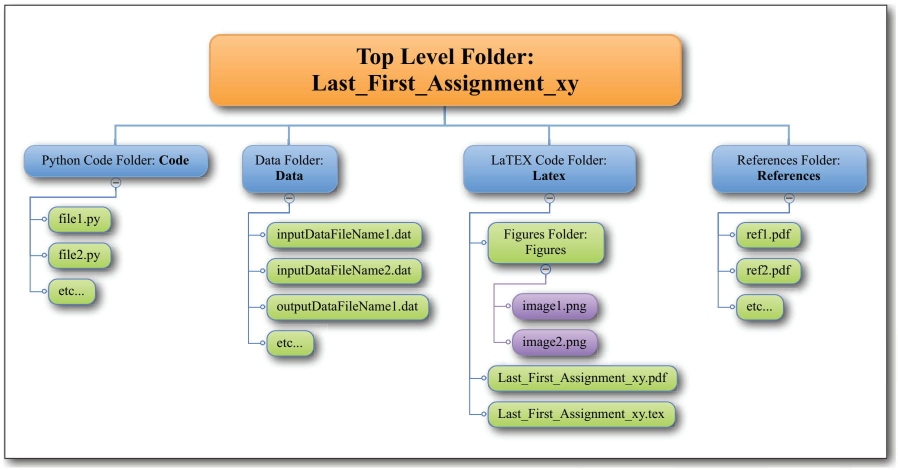


Fig. B.1: You will submit a top level folder with four immediate sub-folders. The LaTeX folder will also have one sub-folder containing all the figures generated by either Python code you wrote, or by a drawing program such as *Inkscape*.

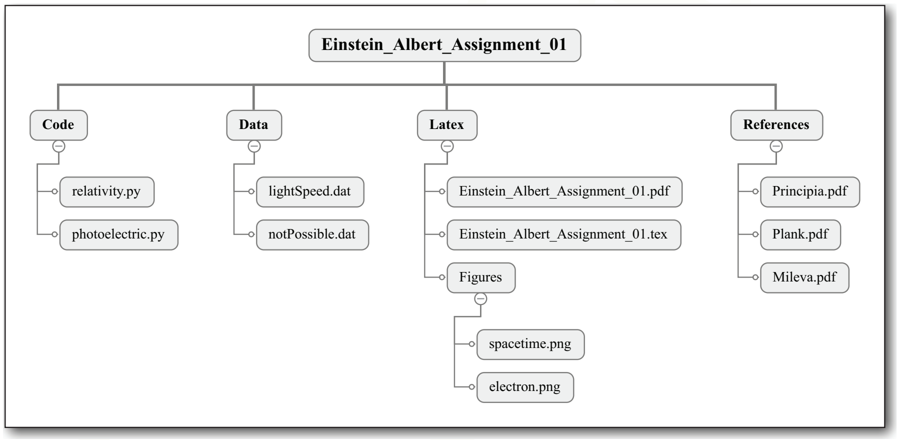


Fig. B.2: Here is a specific example of an assignment submission by a former student of physics. Note that said student would send me a compressed version of the top level folder.

C

General Guidelines for Programming

Writing a Python script or program is necessarily an individualistic endeavor; those of you just learning the language will clearly write different programs than those who have previous experience. However, There are several guidelines that are good to follow:

- Start each program with a pen and paper outline of its structure. For simple programs, this can be a short bit of pseudo code (just a brief outline of the logical steps the script needs to accomplish); for more complicated programs, you will need to actually create a flowchart that explicitly outlines the many logical steps needed.
- When it comes to writing code, get in the habit of using a logical format; here is a structure suggested by Wesley J. Chun in his book *Core Python Programming*[7]:
  1. Startup line (Unix; `#!/usr/bin/env python`)
  2. module documentation (this is what appears between the triple quotes)
  3. module imports (import statements)
  4. variable declarations
  5. class declarations (we'll get to this later)
  6. function declarations
  7. main body of program
- Comment your code as you write. Ideally, your comments should be sufficient for someone else (assumed to be proficient in Python) to understand your code, or equivalently, for you to understand the code a year later.
- Strive for clarity in your code. Especially as you are first learning to program, there is a temptation to include fancy programming techniques. **Don't**. After you are sure your code produces reasonable results (see the next item!), then you can (if it is worth the time and effort) optimize your code for speed and add new features.
- Always be *skeptical* of your program's output and check it by testing it for trivial cases where you know an analytical result. Checking your program's validity is one of the most important steps in computational physics and a considerable effort

should be made to insure that it is working properly before you move on to apply the code to regions that do not admit of analytical results.

- Modularize your code and/or use object oriented programming when possible. Modularization improves your code's clarity as well as providing code that can be used by other programs.

Another set of guidelines I find useful is by Tim Peters, and is called *The Zen of Python*. You can always see it by opening a terminal window, starting python and typing

```
>>> import this
```

and you'll see the following:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

References

1. Norman Chonacky. Has computing changed physics courses? *Computing in Science and Engineering*, pages 4–5, September/October 2006.
2. Rubin Landau. Computational physics: A better model for physics education? *Computing in Science and Engineering*, pages 22–30, September/October 2006.
3. Rubin H. Landau. *A First Course in Scientific Computing*. Princeton University Press, 2005.
4. Rubin H. Landau and Manuel Jose Páez. *Computational Physics: Problem Solving with Computers*. Wiley, 1997.
5. Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, pages 10–20, May/June 2007.
6. Fernando Perez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science and Engineering*, pages 21–29, May/June 2007.
7. Wesley J. Chun. *Core Python Programming*. Prentice Hall, 2nd edition, 2007.
8. Guido van Rossum. *An Introduction to Python*. Network Theory LTD, November 2006.
9. John Zelle. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates, 2004.
10. Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 3 edition, 2008.
11. Katharina Baamann, Cornelius Ejimofor, Alan Michaels, and Alec Muller. Viscous flow around metal spheres, December 2002.