Paul A. Nakroshis

Computational Physics
Physics 261

Course handouts

Fall 2011

October 6, 2011

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 sliderules of yesteryear, and at their maximum, provide a full-fledged computing environment. 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. Python is ideally suited to such work[5, 6]; it can be used in an interactive manner (IPython) similar to Matlab, and it can be used in a purely procedural fashion. At a more advanced level, one can use Python as a fully object-oriented manner similar to Java or C++.

My assumptions are that students have used either a computer running either the Mac, Windows, or Linux operating system, and have basic familiarity with creating folders and files, and have installed Python 2.7 along with SciPy, and MatplotLib, all of which are available for free for any of the three platforms. All of my development for this book has occurred on a 2011 Macbook Pro with 8GB ram, running Mac OS X 10.6.8. All python code used in this book should work equally well on Windows or Linux.

Paul A. Nakroshis

Portland, Maine
September 2011

Contents

1 Brief introduction to LaTeX

Reading:
Langtangen, Chapter 1
Any good online LaTeX tutorial

1.1 Basic idea of LaTeX

LaTeX is not a word processor like Microsoft Office, OpenOffice, 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 an the same. Rather, when you use LaTeX you use a text editor to create the content and the instructions for what to do with the content, and then you invoke LaTeX to process this text 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) 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 this semester—you only need to know enough to get by, which fortunately, is not so hard, and I'll provide you with a template file to use for your reports. Also, since I do have a lot of experience with it, I can likely help you if you get stuck.

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 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.

1.2 A simple LaTeX document

The basic LaTeX 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:

```
\documentclass[12pt]{article}
\begin{document}
Hello World!
\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 dealing 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.

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)

The rest of your document is bounded by the

```
\begin{document}

\end{document}
```

environment, and this is where you put all of your content. For the most part, LaTeX commands are relatively straightforward, and you'll pick up on things

pretty quickly; nonetheless, it's useful, for instance to look at some of the references that appear on http://www.latex-project.org/guides/—I recommend in particular:
The (not so) short introduction to LaTeX $2\mathcal{E}$ 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/.

1.3 LaTeX editor recommendations

1.3.1 Getting up and running

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 More advanced use: LaTeX and python development

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 (not free, but the best $50 I've ever spent on a computer program). I'd also email the developer and ask about an educational discount, which I think he gives. I also have a license for the computers in class. Although this program has not had a large update in several years, it still works quite well (Aug 2011, OS X 10.7) 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[1], 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.

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, 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 Non-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:

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

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] 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.

1.4.3 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 `~\ref{eq:sillyIntegral}` and LaTeX *automatically* takes care of the numbering for me.

1.5 How to include Python code in your LaTeX document

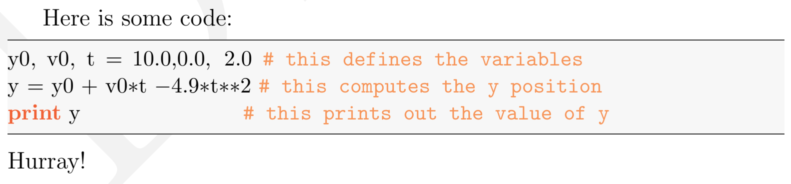
Now, suppose you want to include some Python code (or for that matter, code in practically any computer language) in your report. A nice way to do this is to use the `listings` package (see the WikiBooks site at http://en.wikibooks.org/wiki/LaTeX/Packages/Listings).

1.5.1 Short code snippet

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

```
\documentclass[12pt]{article}
% load the listings package:
\usepackage{listings}
% load a color package to allow us beautify the python code.
\usepackage[usenames,dvipsnames]{color}
% define a light gray color
\definecolor{light-gray}{gray}{0.97}
\begin{document}
%
% The following command defines some options to nicely color
% python code; feel free to use it.
% (as you can see, a % sign makes anything a comment in LaTeX)
\lstset{
language=Python,
basicstyle=\color{black}\small,
frame=lines,
keywordstyle=\color{RedOrange}\bfseries,
stringstyle=\normalfont\color{blue},
showstringspaces=false,
commentstyle=\color{Peach}\ttfamily,
columns=flexible,
backgroundcolor=\color{light-gray}}
%%%
Here is some code:
\begin{lstlisting}
y0, v0, t = 10.0,0.0, 2.0 # this defines the variables
y = y0 + v0*t -4.9*t**2 # this computes the y position
print y # this prints out the value of y
\end{lstlisting}
Hurray!
\end{document}
```

You can see the output produced by running this file through the LaTeX engine in Figure 1.1.



Here is some code:

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

Hurray!

Fig. 1.1. The output produced by the LaTeX code above.

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 python code directly. To accomplish this, here is a simple example:

```
\lstinputlisting[caption={Direct inclusion of a bit of python code
from a file. Notice that I also provide a description of the code,
a habit you should get into!}, label=plot, firstnumber=0]
{Code/Assignment_01/plot.py}
```

produces the following output (Listing 1.1):

Listing 1.1. Direct inclusion of a bit of python code from a file. Notice that I also provide a description of the code, a habit you should get into!

```
"""
plot.py

Plots a simple function. This method of invoking matplotlib
automatically invokes matplotlib.pyplot and numpy. Some people
discourage this method for various reasons dealing with
namespaces....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
```

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 file, I placed a description of the file within triple quotes. This is the format python uses for documentation strings. You can read more about this on page 83 of Langtangen's book.

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. Google away!

Exercise 1.2. Write and save python scripts which solve Langtangen's Exercises 1.2, and 1.3, and 1.8 (1.9 if you have the second edition) and write your answers in a LaTeX file. Use the listings package to include the python files. Make sure for your discussion of Langtangen's exercise 1.8 that you discuss all of the (deliberate) errors in his code.

2

Making simple plots with Matplotlib

Reading:
Langtangen, Chapter 2
Matplotlib Web Site

2.1 Plotting a function

Sometimes you need a quick plot of a function. In Listing 1.1, I have a simple example of this. Let's assume that you want to plot a more complicated function; say

$$y = \sin(t^2) \tag{2.1}$$

Exercise 2.1. Go ahead and write a python script called `plotFunction.py` that displays this Equation 2.1. Is your plot reasonable? Why or why not? If it's not reasonable, fix the code so that it gives a reasonable plot!

2.2 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 2.1)

Table 2.1. Imaginary data from your lab notebook; assume that the time uncertainties are all equal to 0.2 s.

time (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

Exercise 2.2. Using the “data” in Table 2.1, use numpy and matplotlib to plot the data, complete with errorbars and axes lables. Your result should look like Figure 2.1. Your report should include a duplicate of the Table 2.1, your python code, and the plot created by matplotlib.

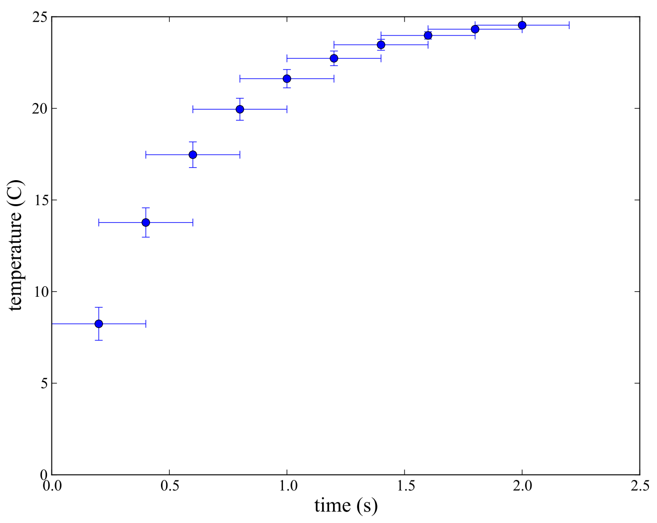


Fig. 2.1. A plot of the data shown in Table 2.1.

3

Extracting the data you want from a text file

Reading:
Langtangen, Chapter 2
Matplotlib Web Site

3.1 Statement of the problem

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.

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.

3.2 An example

Let's make this more concrete with an example. Table 3.1 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 3.1. 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 3.1 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 3.2), *Matplotlib*.

3.3 Details for this Assignment

1. Read the file PeriodData.dat, 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). I've posted one way (not the most elegant, but it works) to do this at http://people.usm.maine.edu/pauln/261downloads.html in file filter.py

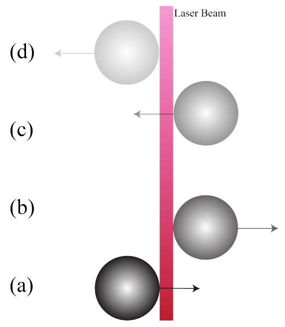


Fig. 3.1. The idea behind an optical switch—each time the pendulum enters or leaves the beam, a digital timing signal is recorded. In the figure, the pendulum is drawn a four different times; on the 5th crossing (not shown), one will have enough information to calculate the period.

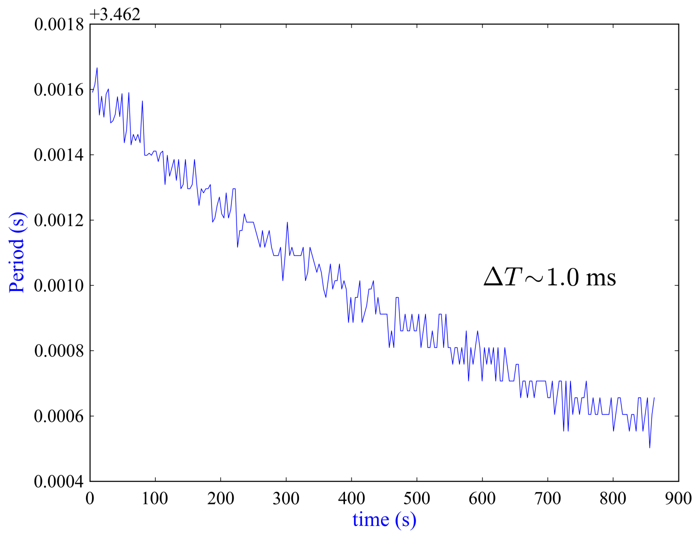


Fig. 3.2. A plot (using Matplotlib) of the period vs time data from the full data set. Notice that Matplotlib automatically pulled out 3.462 s from the period axis on the left, so that the vertical tics are space 0.4 ms apart, and over the course of 850 seconds, the period of the pendulum only changed by about 1 mS.

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.
4. As a part two to this exercise, still using the data from periodData.txt (at http://people.usm.maine.edu/pauln/261downloads.html) 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. You do not have to use the code I posted to do part 1—if you have a better way, please feel free to ignore my code, and I'll include a handout of all the different methods people used when I hand back your assignment submissions.

4

Python Basics

4.1 General Overview

This chapter will assume that you have installed a working version of Python 2.7 on your computer. In order to use Python, you have to type commands, and to do so you have several options. First, your installation of Python likely comes with a Python interpreter (on the Enthought Python installation, it is called IDLE). Double-clicking on this application will start a python shell window. Second, you can open (in Linux or OS X) a terminal window and type

```
python
```

You will then see something like this:

```
Enthought Python Distribution -- www.enthought.com
Version: 7.1-2 (64-bit)

Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 27 2011, 14:50:45)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "packages", "demo" or "enthought" for more information.
>>>
```

The notation `>>>` indicates that Python is ready to accept commands, and is an interactive mode. These first two options are almost identical in their function, so I'll leave it to the reader to choose which is most convenient. The third option is to use a text editor to write Python code and then compile the code either with the terminal, or, if the text editor is powerful enough, from within the text editor itself. On OS X, there is an excellent editor called TextMate, which excels at this. In fact, TextMate was also the editor used to write the LaTeX code for this book. A cross platform and open source option is Geany. The last option is to use an integrated development environment; there are many programs in this category, but I opt not to use this route because of the overhead needed, and my bias that a closer contact to the code and compilation is better for understanding what is happening when learning to program.

This textbook will use the interactive mode (through either the terminal or IDLE) and compiling text files containing Python code. For short programs, or for developing code, it is convenient to use the interactive mode, as you can get immediate feedback. More on this soon. For the bulk of our work in this text, we will assume you are using a text editor, and write files which are then run by the Python interpreter. Whenever you see the notation `>>>`, this is an indication that Python is being used interactively, and it's a good idea to try out the examples.

This chapter will explore the interactive mode, extending Python with external libraries, how to write and compile Python code, and generally provide you with a baseline of tools needed to get started programming so that we can get to the physics as quickly as possible.

4.2 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). Notice that we have to be careful when dividing as the following examples show (comments are mine):

```
>>> 2/5   # dividing two integers gives the lowest integer
0
>>> 5/2   # same thing here.
2
>>> 5./2   # here we divide a floating point number by an integer
2.5
>>> 5/2.   # same 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.

In physics, we often have more complicated arithmetic, such as 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 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. In this case, I would likely trust the Mathematica result, as it is specifically designed for high precision calculations. However, the point here, is that Python (and 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.

4.3 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 look a little more closely at the math library and how to find out about the available functions.

Table 4.1 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:

```
>>> 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.

Table 4.1. 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	$\pi = 3.14159...$

4.3.1 Two methods of loading libraries

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

1. **from** math **import** *
2. **import** math

This text will usually use method (1) which 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 (2) 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.

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 4.3.
(a) Start a python session using a terminal window or using IDLE. Import the math library and type `help(math)` as outlined in section 4.3. (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 4.2 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 4.1.

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.

4.4 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

$$y(t) = v_0 t - \frac{1}{2} g t^2. \tag{4.1}$$

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:

Listing 4.1. Our First Python Program

```
#!/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"
```

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 (shareware) or Smultron, Vi, or Emacs (open source). Another option (which is also cross platform and 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.

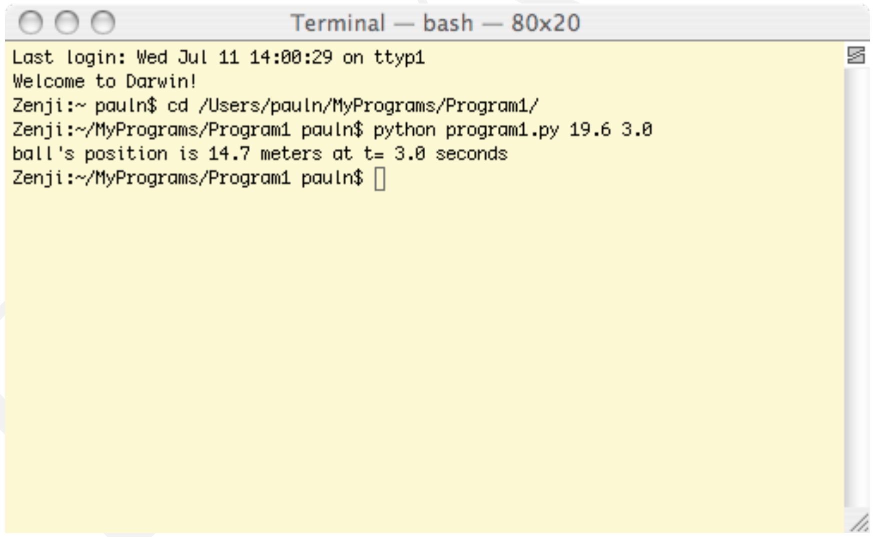
To run the program, open a terminal window (on the Mac, this is in the Applications/Utilities/ folder). Then, before running the program, you have to change your directory (unix command=cd) to your current program folder. To do this, you have two options on the Macintosh:

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 4.1.



```
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. 4.1. Running our first Python program

4.4.1 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

```python
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.

4.5 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.

Listing 4.2. Our second Python program adds graphical output using MatPlotLib.

```
#!/usr/bin/env python
# Program 2
"""
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.
"""
import sys
from pylab import *
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 = load(outfilename, skiprows=1)   # (pylab function)
xaxis = data[ : , 0 ]                  # first column
yaxis = data[ : , 1 ]                  # second column
plot( xaxis, yaxis, marker='o', linestyle='None')
xlabel('time (s)')
ylabel('Height (m)')
title(' Vertically Launched Ball')
show()
```

4.5.1 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 4.2.

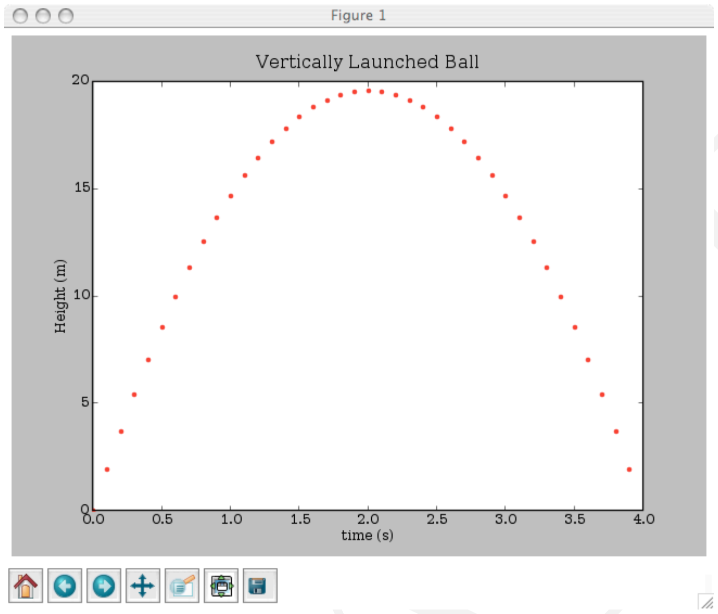


Fig. 4.2. MatplotLib output from our second Python script. Clicking the bottom rightmost disk icon at the bottom of the plot will save the figure as a .png file. To return control to the terminal, simply close the window. Explore the other buttons to see some of the interactive features provided with every Matplotlib figure.

4.5.2 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. More on this in a later chapter. Then the script procedes by loading the libraries we will need. The system (sys) library for reading and writing data, and the pylab library for accessing Matplotlib, which is a Matlab-like plotting library. The distinction between pylab and Matplotlib is actually a bit unclear to this author, as even the Matplotlib web site presents the two packages as synonymous, however, it will not work in this case to replace `from pylab import *` with `from matplotlib import *`.

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 4.2) 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 4.2 shows a list of common format specifiers.

Table 4.2. 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()).

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

```
$ python
>>> import python
>>> help(pylab.load) .
```

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.

4.6 Saving Functions as Modules

Although our second program (4.2) 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):

Listing 4.3. Sections of code can be saved as reusable functions

```
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
```

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 4.2 modified to use our function `height()` which is included in the file `analytic.py`:

Listing 4.4. Our first program made modular.

```
#!/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
from pylab import *
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 = load(outfilename, skiprows=1)    # (pylab function)
xaxis = data[ : , 0 ]                   # first column
yaxis = data[ : , 1 ]                   # second column
plot( xaxis, yaxis, marker='o', linestyle='None')
xlabel('time (s)')
ylabel('Height (m)')
title('Vertically Launched Ball')
show()
```

4.7 Other Data Types in Python

Although we will primarily be using floating point numbers and integers, Python also has several other data types that we will use: Boolean, complex numbers, strings, and lists.

4.7.1 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(20<100)
True
>>> bool(20<=19)
False
>>> bool(20<=20)
True
```

4.7.2 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
```

4.7.3 Strings

Strings are simply sequences of alphanumeric characters, and in Python, can be enclosed in single or double quotes. You can also refer to a specific character by its position in the sequence, and can also easily extract a range of characters:

```
>>> x='Ministry of Silly Walks'
>>> x
'Ministry of Silly Walks'
>>> x[0]
'M'
>>> x[5]
't'
>>> x[0:8]
'Ministry'
```

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

```
>>> y=x+'!!'  # creates a new string with added exclamation points
>>> y
'Ministry of Silly Walks!!'
>>> z=y[:-2] # creates a new string which is every character from
>>> z        # y except the last two characters.
'Ministry of Silly Walks'
```

Being able to add a character(s) to a string is especially convenient when writing a series of output files with slightly different names.

4.7.4 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
```

There are many other features of lists, and we will introduce them as needed.

4.8 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.

4.8.1 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:   #note that 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!'
```

4.8.2 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, ...).

4.8.3 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 4.5.2. 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
```

4.9 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 4.2. 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?

4.10 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 second edition, with a third edition underway. Highly recommended.

Problems and Tutorials

4.1. Interactive Python

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) $\ln(e)$

4.2. Using Python Help

When importing libraries into Python, we will generally use the method described in section 4.2. 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). 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 4.3.

(a) Start a python session using a terminal window or using IDLE. Import the math library and type help(math) as outlined in section 4.3. (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.

(c) Read about the atan2() function using help. Evaluate the arc tangent of a vector with x and y components of -2 and +3 respectively. Why is this a useful function (compared to atan()?)

4.3. Matplotlib

Write a simple Python program to 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.

4.4. 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.

(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.

5

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.

5.1 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 5.1.

5.1.1 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

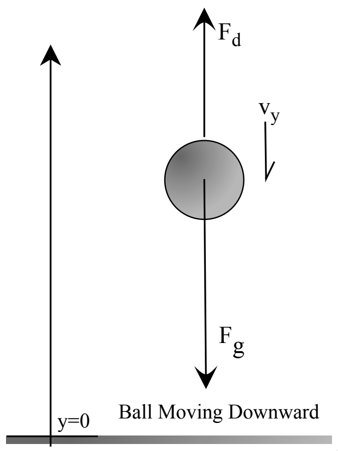


Fig. 5.1. Free body diagram for an object released from rest a short distance above the surface of Earth.

to the square of the speed). Then we can write the equation of motion for the ball as

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

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

$$g - \frac{b}{m}v_y = \frac{dv_y}{dt} \tag{5.1}$$

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

$$v_y(t) = \frac{mg}{b}\left(1 - \frac{A}{g}e^{-\frac{b}{m}t}\right) \tag{5.2}$$

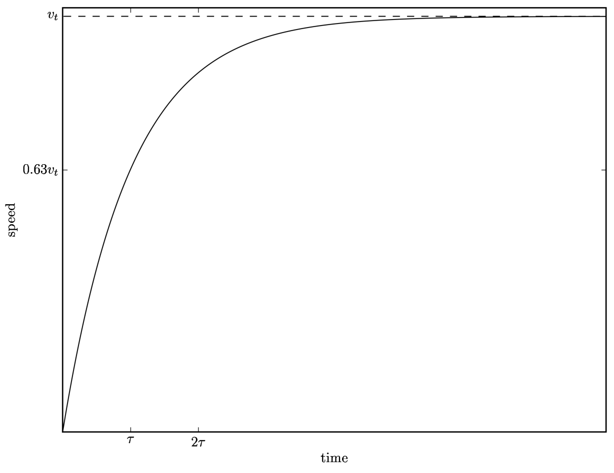


Fig. 5.2. Speed vs time for an object falling in a linear drag regime. Notice that if we define a time constant $\tau = v_t/g$, the object reaches 63% of its terminal velocity after τ seconds. After several time constants have elapsed, the object is very close to its terminal velocity.

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 5.2 simplifies to

$$v_y(t) = \frac{mg}{b}\left(1 - e^{-\frac{b}{m}t}\right) \tag{5.3}$$

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 5.3 in the limit $t \to \infty$, we find that

$$v_t = \frac{mg}{b} \tag{5.4}$$

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

$$v_y(t) = v_t\left(1 - e^{-\frac{gt}{v_t}}\right) \tag{5.5}$$

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

$$v_y(t) = v_t\left(1 - e^{-\frac{t}{\tau}}\right) \tag{5.6}$$

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 5.2

5.1.2 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 5.1)

$$\frac{dv_y}{dt} = g - \frac{b}{m} v_y, \tag{5.7}$$

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 5.4) to write this as

$$\Delta v_y \approx g \left(1 - \frac{v_y}{v_t} \right) \Delta t. \tag{5.8}$$

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:

Listing 5.1. 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.

```
#!/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 (contained in the file EulerFreeFall)
to execute the Euler method.
"""
import sys, EulerFreeFall
from pylab import *
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
# 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()
# create array for plot of analytic function for comparison:
a = arange(0,tmax,dt)
c = vterminal*(1-exp(-g*a/vterminal))
plot(a,c,'k--')
# read in the datafile, & plot it with MatPlotLib:
data = load(outfilename, skiprows=1)  # (pylab function)
xaxis = data[ : , 0 ]                 # first column
yaxis = data[ : , 1 ]                 # second column
```

```
plot( xaxis, yaxis, marker='.', markersize=5, linestyle='None')
legend(('Analytic Solution', 'Euler Method'), loc='best') # writes legend
xlabel('time (s)')
ylabel('velocity (m/s)')
title('Vertically Dropped Ball')
show()
```

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

Listing 5.2. The contents of the file EulerFreeFall.

```
# 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)
```

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 5.1 with

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

we obtain the output shown in Figure 5.3.

Notice that Figure 5.3 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 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

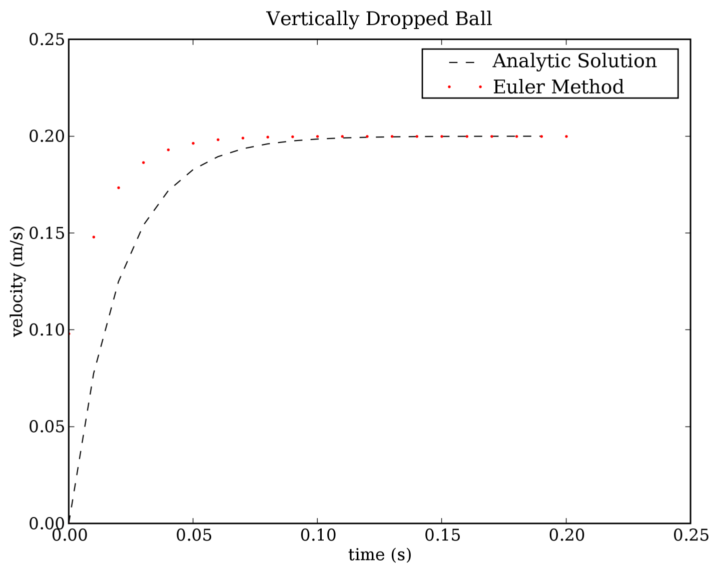


Fig. 5.3. Speed vs time for a 1.25 cm diameter ball bearing falling through glycerin at room temperature. Note that the time step is clearly too large, since the simulation is clearly not a good fit to the analytic solution.

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.

5.1.3 Simulation of Linear Drag: getopt package

For example, here is Listing 5.1 modified in the following manner:

- We use the getopt (get options) 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.

Listing 5.3. 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 getopt package to parse input parameters.

```
#!/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, getopt, EulerFreeFall
from pylab import *
t=0.0          # initialize time to 0.0
g=9.8          # define acceleration due to gravity
#
#      READ IN RUN PARAMETERS (new method; GETOPT)
#
options, timeSteps = getopt.getopt(sys.argv[1:],
 'hf:v:t:m:',['help','filename=', 'v_initial=', 'v_terminal=', 'tmax='])
for option, value in options:
        if option in ('-h', '--help'):
                print "python sys.argv[0] --filename --v_initial \
                 --v_terminal --tmax dt0 dt1..."; sys.exit(0)
        elif option in ('-f', '--filename'):
                filename=value
        elif option in ('-v','--v_initial'):
                v=float(value)
                vinitial=v
        elif option in ('-t', '--v_terminal'):
                vterminal = float(value)
        elif option in ('-m','--tmax'):
                tmax = float(value)
print filename, v, vterminal, tmax
#Now loop over all values of dt...
for n in timeSteps:
        dt=float(n)      # change string to floating point number.
        print dt
        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()
# read in the datafile, & plot it with MatPlotLib:
        data = load(fname, skiprows=1) # (pylab function)
        xaxis = data[ : , 0 ]                   # first column
        yaxis = data[ : , 1 ]                   # second column
        plot( xaxis, yaxis, marker='.', markersize=7, linestyle='None')
        t=0.0
        i=0
        v=vinitial

# set up legend, label axes and show plot:
legendstr=[]    #the following four lines set up the legend
for j in range(len(timeSteps)):
        legendstr.append('dt=' + timeSteps[j])
legendstring=str(legendstr[:])
legendstring=legendstring.strip("[ ]")
print legendstring
legend(legendstring.split(', '), loc='best') # writes legend
xlabel('time (s)')
ylabel('velocity (m/s)')
title('Vertically Dropped Ball')
show()
```

The getopt 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), we run this program as follows:

```
python FreeFallGetOpt.py --filename junk --v_initial 0.0
--v_terminal 0.2 --tmax 0.2 0.01 0.001 0.0001
```

and this produces the output shown in Figure 5.4 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=0.0001 seconds.

5.1.4 Simulation of Linear Drag: argparse package

Listing 5.4. 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.

```
#!/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
from pylab import *
import argparse
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()
```

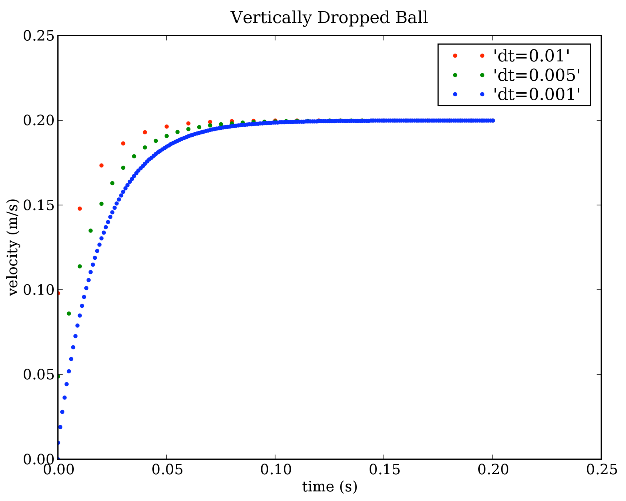


Fig. 5.4. Speed vs time for a 1.25 cm diameter ball bearing falling through glycerin at room temperature; three different time steps are shown.

```
# 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=10.0, help='terminal velocity')

parser.add_argument('--tmax', dest='tmax', type=float, \
default=20.0, help='maximum simulation time')

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

#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
# saves dt values as a string list called timeSteps:
timeSteps = input.dt
print filename, v, vterminal, tmax, timeSteps
#
# 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 = loadtxt(fname, skiprows=1)      # (pylab function)
        xaxis = data[ : , 0 ]                  # first column
        yaxis = data[ : , 1 ]                  # second column
        plot( xaxis, yaxis, marker='.', markersize=7, linestyle='None')
        t=0.0
        i=0
        v=vinitial

# set up plot parameters and plot data
legendstr=[]    #the following four lines set up the legend
for j in range(len(timeSteps)):
        legendstr.append('dt=' + timeSteps[j])
legendstring=str(legendstr [:])
legendstring=legendstring.strip("[ ]")
print legendstring
legend(legendstring. split (', '), loc='best') # writes legend
xlabel('time (s)')
ylabel(' velocity (m/s)')
 title (' Vertically Dropped Ball')
show()
```

5.2 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

$$\mathbf{F}_\mathrm{D} = -\frac{1}{2}\rho C_d A v^2\ \hat{v} \tag{5.9}$$

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 5.5:

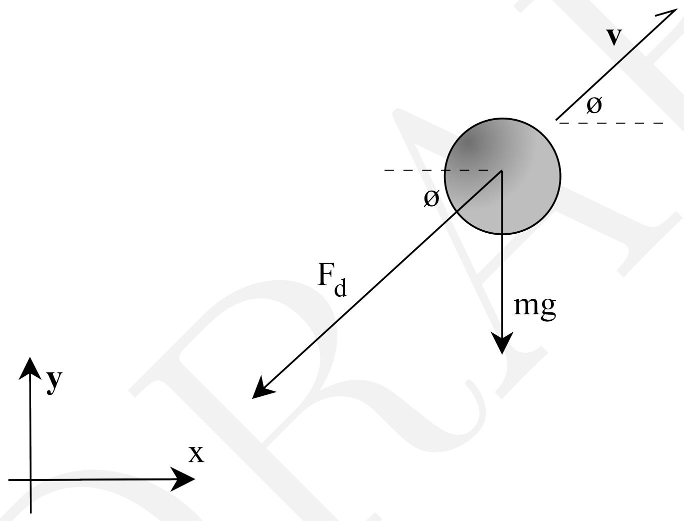


Fig. 5.5. A sphere moving in two dimensions with air drag.

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

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

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 5.9 as

$$\mathbf{F}_{\mathrm{D}} = -\frac{1}{2}\rho\, C_d A v^2\ \hat{v} = -B\, v^2\ \hat{v} \tag{5.10}$$

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

$$\frac{B}{m} = \frac{\frac{1}{2}\rho_a C_d \pi R^2}{\frac{4}{3}\pi R^3 \rho_s} \tag{5.11}$$

If we make the assumption that $C_d = \frac{1}{2}$ for the drag coefficient, then we have

$$\frac{B}{m} = \frac{3}{16}\frac{\rho_a}{\rho_s}\frac{1}{R} \tag{5.12}$$

Problems

5.1. 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 5.1.2, 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.

5.2. Simulate the motion of three objects:

1. A steel ball with density 8000 kg/m^3 traveling through vacuum.
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.
Using an initial velocity of 100 m/s and a launch angle of 30^o:
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?

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.

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).

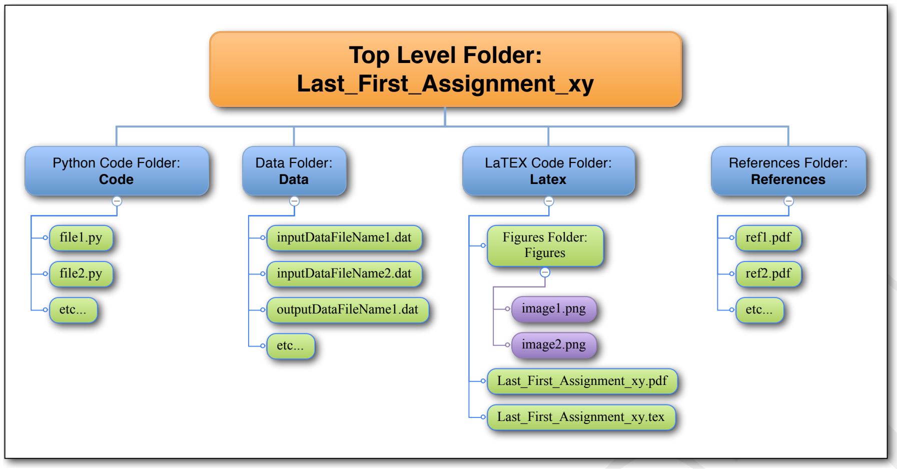


Fig. B.1. You will submit a top level folder with four immediate sub-folders. The LaTeX folder will also have one sub-folder containing all the figures generated by either Python code you wrote, or by a drawing program such as *Inkscape*.

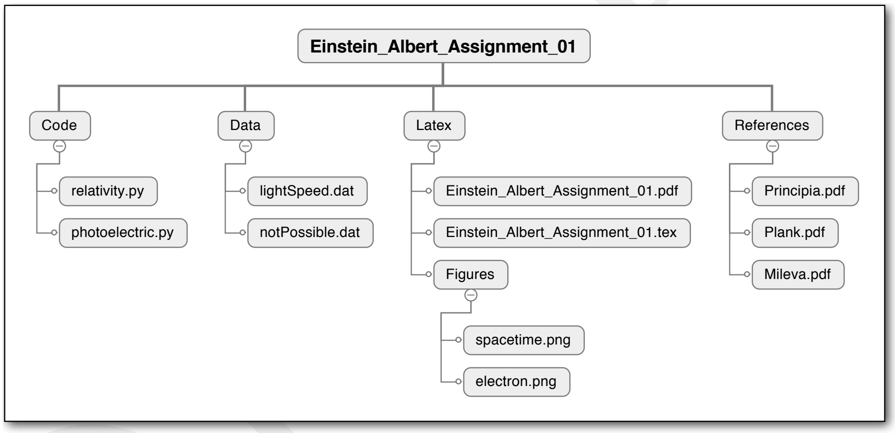


Fig. B.2. Here is a specific example of an assignment submission by a former student of physics. Note that said student would send me a compressed version of the top level folder.

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.
- 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.

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, 2nd edition, 2006.
11. Katharina Baamann, Cornelius Ejimofor, Alan Michaels, and Alec Muller. Viscous flow around metal spheres, December 2002.