Level 3 PHYS3561 Computing Projects

Introductory Booklet

2019/20

Organiser: Dr Cristina Zambon
e-mail: cristina.zambon@durham.ac.uk
room: Ph210

Contents

Part I

The Course

Chapter 1

Course Overview

1.1 Where to find information

You should be able to find everything you need on DUO, course PHYS3561: look for "Computing Projects". Information is in the "Course Documents/Computing Projects" folder. This includes:

- The Project Booklet, giving you all the project information you need to get started.
- Staff contact information.
- Computing tutorials: time and rooms.
- Tips for presenting posters.

For help on Python programming, look on the web at
https://www.dur.ac.uk/physics/students/labs/skills/computing/

1.2 Important deadlines

check the project deadlines given here
https://www.dur.ac.uk/physics/students/deadlines/level3/

Guidlelines and page limits for your report are given here
https://www.dur.ac.uk/physics/students/assessment/reportwriting/

1.3 What is the Aim of the Course?

At levels 1 and 2, you have learned how to write simple programs using the python programming language and how to code up simple computer algorithms. Your computing project gives you a chance to apply these skills to undertake some research in physics. You will be able to investigate aspects of physics that

go well beyond most undergraduate text books, questioning whether the results presented are correct, and extending the ideas into regimes that cannot be addressed with analytic mathematics. The computer is a powerful tool – if you can write down a problem in mathematical form, then the computer will be able to solve it. You must however pay attention to the accuracy of your result and understand the errors that arise from the way in which you have discretized the problem. An ideal report should be a min-research paper, perhaps re-examining old work with new methodologies.

1.4 What to attend

After the introductory lecture (which everyone attends), you will attend tutorial style workshop (and perhaps the drop-in programming help sessions) *every two weeks*. You should check the schedule to see which day and which week you are to attend, and check the departmental webpage giving the official deadlines for your work. It is up to you to make the best use of this tutorial time, so you should come prepared, having made progress on your research in your own time. I would suggest that you keep a notebook in which you record ideas and questions that have occurred to you over the course of the two weeks between the tutorial sessions. It is very important that you take responsibility for your work and for making these tutorial sessions useful. It is up to you to bring up topics for discussion in the tutorial.

It is extremely important that you realize that most of your project work needs to be undertaken in your own time.

The tutorials are not intended to solve problems with computer programs. In order to get help with technical problems, you should seek help from the post-grad demonstrators during the optional drop-in sessions. These take place in Ph216, and run in parallel with the tutorials. We anticipate that you will gain most by attending this in the alternate weeks. In periods of high demand, the demonstrators will prioritise the students according to their tutorial time.

1.5 When to attend it

The project allocations have been published on DUO (and on the L3 noticeboard). You need to note the short name of the project that you have been allocated.

When/where does my group meet? This information is available on DUO (see e-mail from Adrian Skelton)

1.6 List of Projects

These are the projects being offered this year. The short name of the project (used in the timetable) is given in square brackets.

1. Silicon Band Structure [band structure]
2. Transmission of Electrons through a Semi-Conductor Barrier [semiC barrier]
3. Simulation of a Phase Transition [phase transitions]

4. Coloidal Fluids: simulations of hard disks and spheres [coloidal fluids]
5. Light-Matter Interactions in atomic physics [light-matter]
6. Quantum optimization and applied algorithms [Q-optim]
7. Qubit Dynamics and Quantum Computing [Q-comp]
8. Solitons [solitons]
9. The Black Hole Accretion Disk Spectrum [accretion disk]
10. Supernova Cosmology [superN cosmo]
11. Gravitional Collpase: simulating interactions between many bodies [Grav Collpase]
12. The Restricted 3-Body Problem [rockets]
13. White Dwarves and Neutron Stars [neutron stars]
14. Resonant Scattering [res.Scattering]
15. Quarkonium [quarkonium]
16. 3-Particle Quantum Mechanics [3-particle QM]
17. Feynman Path Integral [F-path]

You will need to look in the projects booklet (available on DUO) to find the detailed instructions for your project. This includes background information, an explanation of the "milestone" problem, and some suggestions for further research. The projects booklet is rather long - I recommend that you only print the relevant section.

1.7 What you have to do

The initial description of your project is intended only as a guide to get you started. You will need to research your topic in much more depth in order to present a good report. There are five tasks that you will need to undertake: the dates below are only indicative: make sure you check the official deadline on department's page .

- Design and create a "milestone" program. This is a simple program that will get you started on your research. All the programs have well defined plots that you need to make and numbers that you need to calculate. You should not rush into this program, but spend some time designing it so that it can easily be extended to undertake your research goals. The marks for this program are "formative" – we will check your program works and give you suggestions to help you be a better programmer. *You are expected to demonstrate good progress on your milestone program in tutorial session 2.* The formal deadline is session 4 (teaching weeks 8 or 9), so that everyone should have a working program before the end of Michaelmas term.

- A formative poster. In session 3 (teaching weeks 6 and 7) you will present a poster on your milestone and background information. This will be a warming up exercise in preparation for the summative poster in epiphany term.
- A 10 min Presentation in epiphany term. Once you have a satisfactory milestone program, you will research your topic, and present your ideas for further exploration in a 10 min talk, along with some preliminary results. You will give your talk in session 5 (teaching weeks 11 and 12). The oral talk will be summatively assessed.
- A summative Poster. In epiphany term, you will create a poster drawing on your research and research papers you have found on the web. Your poster is aimed at demonstrating the excitement of your project to a general physics audience. The poster will be presented in teaching week 15 or 16.
- Your Report. The majority of your marks are awarded on the basis of your report. This is a document similar to a research paper. Check the department pages for the page limits and other details . The highest marks will be awarded to reports that show an innovative approach. Reports are to be submitted in teaching week 19 or 20.

1.8 Help on Python

Python is a good beginners' language because you can develop programs very quickly. This means you can spend more time focused on the physics of the problem you are tackling.

For help on Python programming, look on the web at

https://www.dur.ac.uk/physics/students/labs/skills/computing/python/

The very useful "Introduction to Programming in Python" booklet from the Level 2 course can also be found there.

Also, make sure you read the chapter "Common Traps and Pitfalls" at the end of this document, where we've collected together some common errors that are often made.

Finally, if you find any errors in any of the documentation, please tell us so that we can update it for next year's students.

Part II

Revision and Program Design

Chapter 2

Python Revision

2.1 Introduction

This tutorial is in the form of a set of exercises. Each exercise requires you to write a program that is only a few lines long. As a Python expert, you should be able to complete the initial programs quickly and without making syntax errors!

If you can't tackle one of the exercises, you need help. You could try: looking at your "Introduction to Python" notes from last year; reading Magnus Hetland's book "Beginning Python"; You could also try typing something suitable into Google. The last option is definitely quickest!

If you want to start from square 0, reading Magnus Hetland's quick and short introduction to programming (in general) and Python is definitely recommended: "Instant Hacking",

http://hetland.org/writing/instant-hacking.html

If you really can't solve the problem for yourself in 5 mins, ask a demonstrator. Don't ask the demonstrator immediately, or you'll never figure out how to solve problems for yourself!

Remember that you can start an interactive Python command line by just typing "python" (giving no file name). You can then type in a few lines of Python to check how the syntax works or exactly what some of the obscure numpy functions do. This is a major advantage of Python — use it!

The later exercises remind you how to use the "numpy" maths module and the graphics module "pyplot". For help on these modules (rather than general Python language problems), look at:

- numpy pages on the web (for example, http://www.scipy.org/Numpy_Example_List or the tutorial pages http://www.scipy.org/Tentative_NumPy_Tutorial)

- pyplot pages on the web. For example the tutorial, http://matplotlib.sourceforge.net/users/pyplot_tutorial.html

2.2 Python Programming

2.2.1 "Hello World"

Create a file called "hello_world.py" that prints "Hello World" on the screen when you run it by typing "python hello_world.py" on the command line.

Hint: you can use any editor you like to create the file. Personally, I like to use "emacs" with the fancy Python-friendly colouring enabled.

You should solve the problems below in the same way (ie., create a file containing Python code, run it by typing something like "python mycode.py": of course you can choose a more meaningful file name!).

2.2.2 A simple loop

Write a program to print out the integers 10 - 20 and their squares.

Hint: remember that Python loops start at 0. Don't forget that indentation is important! Don't miss out that all important ":".

2.2.3 Area of a rectangle

Write a program that asks the user for two numbers and multiplies them to compute the area of a rectangle. Print out the result in a nice "%g" format.

Hint: how do you make sure the input numbers are floating point type and positive?

2.2.4 Using functions

Modify the program above (2.2.2) to compute $f(x, y)$ rather than just squaring the numbers. Use a simple function to separate out the code that evaluates $f(x, y)$, and then set $f(x, y) = \sqrt{(}xy)$.

Hint: make sure your function does something sensible even if x or y are negative.

Separating out the piece of code that evaluates $f(x, y)$ is an important part of the Python style for several reasons: it's easy to change the function; if the code to evaluate $f(x, y)$ is very long then it doesn't obscure the working of the main program; you can use the command line to evaluate e.g. $f(2., 1.)$ and so check it works correctly. All of these features make it easier to find errors in your code.

Hint: if you find that a function is becoming longer than 10 or so lines, you should seriously think about splitting it up into smaller functions.

Note that the variable names in the function definition can be different from those used when you call it. This is a very important concept. Change your program so that the variables used to define the function are called u and v.

2.3 Using Numpy

Now you know how to write programs, let's explore some of the powerful features (don't worry about neatly formatting output etc.). These "programs" are only a few lines long.

2.3.1 Loading the module

Write a program that uses numpy to compute the sum $s = \Sigma_i \log_{10}(x_i)$ for $x_i = 10, 11, 12...20$. Do this using "np.arange" to create an array of numbers, and a loop over x_i to evaluate the logarithm and add them up.

Hint: it's good style to use the form "import numpy as np" to load the module. This makes it clear that "np.log10(x)" is a function from the numpy module and ensures it doesn't get confused with a function that you've defined yourself.

2.3.2 Using arrays

A very powerful aspect of numpy is that it defines data types for arrays, matrices and even tensors, and provides functions (or "methods") for multiplying, adding them etc. This makes it possible to write very compact powerful programs.

Re-write the program above to define x as an array containing the values 10 - 20 so that you can compute the sum s using something like y=np.log10(x) and np.sum(y).

To make sure you understand how this works, compute $\Sigma_i x_i^2$.

Note: be careful when you assign one array to another. The statement y=x is not the same as y=x.copy() or y=x[]. Can you explain the difference?

2.3.3 Accessing individual array elements

Given the array $x_i = 90, 91, 92...99$, replace even numbers with their negative values.

*Hint: the obvious approach is to loop through each element of x multiplying values by -1 if x_i is divisible by 2 (eg., "np.mod(i,2)==0"). But you could alternatively define a vector "ieven = np.arange(0,10,2)" and then use "x[ieven] = -1.0 * x[ieven]." This is called "masking" an array. Experiment with both approaches.*

2.3.4 Sorting an array

Write a program to sort some numbers (eg., [17., 9., 2., 3., 20.]) into the right order.

A simple way to do this is a "bubble sort". You loop through the array comparing adjacent numbers and swapping them over if they have the wrong order. The following code snippet gives you the idea:

```
for i in range(len(x)):
    for j in range(len(x)-1):
        if (x[j] > x[j+1]):
```

```
tmp = x[j]
x[j] = x[j+1]
x[j+1] = tmp
```

Once you understand why it works, turn this into a working program. Can you make it more efficient?

Hint: add an extra variable to record if a swap was made, and then replace one of the "for" statements with a "while" statement.

Of course, numpy contains its own sorting routines "sort" and "argsort". Use the numpy documentation to find out what they do. It is important that you figure out how to get help on routines, so ask a demonstrator if you are finding it hard to work out how these routines work.

2.3.5 Numpy's array operations

Numpy has many built-in array operations. Arrays can be multidimensional too, so that it is simple to take dot and matrix products for example. Write a simple program to illustrate this: if $\mathbf{x} = [1., 2., 3.]$ and $\mathbf{y} = [2., 2., 1.]$ and $z = [[1., 2., 1.], [0., 3., 4.], [8., 7, 9.]]$ (where $\mathbf{z}$ is a 3×3 array), calculate:

- $\mathbf{x} \cdot \mathbf{y}$ (dot product of two vectors, "np.dot")
- $\mathbf{x} \times \mathbf{y}$ (cross product of two vectors, "np.cross")
- $\mathbf{z} \cdot \mathbf{x}$ (a matrix operation on a vector, "np.dot", the order matters!)

There are many more operations that are built into numpy. The relevant ones are pointed out in the project descriptions.

Hint: be very careful to check that an operation really does what you intend. It is easy to check by typing into an interactive Python command line. A few seconds checking will save you a lot of time later!

What happens if you do a straight-forward multiplication of $\mathbf{x}$ and $\mathbf{y}$ (ie., "a = x * y")?

2.4 Plotting graphs

The matplotlib.pyplot module provides a powerful graph plotting package. You can use the module to quickly make graphs on the command line, but (most powerfully) you can include the graph plotting commands in your programs and write the graphs to a file. You can then use these plots in your report.

A good starting point is

http://matplotlib.sourceforge.net/users/pyplot_tutorial.html

Note that you make also come across pylab. This is similar to matplotlib.pyplot, but includes some of the functionality of numpy.

These programs illustrate just a few of the more obvious possibilities. Look at the pyplot web page (http://matplotlib.sourceforge.net/gallery.html) to see a gallery of the kinds of beautiful plots you can make. *Most of the examples are far too complicated for what you will need!!!*

2.4.1 Plot a simple function

Make a plot of the function $f(x) = \sin(x)/x$ (the "sinc" function) over the range $-\pi < x < \pi$. Start by writing your output to the screen. When it looks good, write it to a file.

I would do something like:

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

...some code to define x and y ...

plt.figure()
plt.plot(x,y,'r')               # 'r' means plot the line in red
plt.xlabel('x')
plt.ylabel('sinc(x)')
plt.savefig('myplot.png')     # delete this line unless want to save a copy in a file.
plt.show()
```

If you just want to just generate an image file to put in your report, you don't need the interactive window, so you can delete "plt.show".

2.4.2 Plotting a graph line by line *[Advanced]*

Sometimes it is useful to be able to see a graph built up line by line. Use Python's command line to make your plot appear line by line. **For obscure reasons it seems tricky to make this work reliably on the CIS X windows system**.

You can only use "show" once at the end of your script, but you can use "ion" to update a figure as it emerges step by step. Once the program gets to the "show" command the interactive allows you to use zoom the graph etc. Kill the window to return to Python (sometimes it gets stuck (if you forget "ioff"): typing cntrl-C will get you back, but rather ungracefully). For example:

```
plt.ion()                 # switch on interactive mode:  don't use for complicated drawings
plt.figure()
for ii in np.arange(len(x)):
  plt.plot(x[ii],y[ii],'r.')         # 'r.' means plot with a red dot
  plt.draw()             # forces one point at a time and rescales the axes
plt.xlabel('x')
plt.ylabel('sinc(x)')
plt.ioff()               # switch off interactive mode.
```

If you want to do this a lot, look at "ipython" on the web.

2.4.3 Making prettier plots *[Advanced]*

Make your plot a bit prettier. Change the range of the y axis so that it is 20% larger than the range of the data.

Hint: There are commands that let you change the range of the axes (e.g., "plt.xlim(xmin,xmax)"), plot additional lines ("plt.over"), plot multiple plots on a single page ("plt.subplot"), histograms ("plt.hist"), and many more.

Hint: as you've probably guessed, the numpy functions np.min(y) and np.max(y) return the minimum and maximum of y.

2.4.4 Making a contour plot *[Advanced]*

Make a contour plot of the function $f(x, y) = exp(-(x^2 + y^2 - 0.5xy))$ over the range -3 to +3 in x and y.

Hint: Make an array of x values and an array of y values and then use the trick "xgrid, ygrid = np.meshgrid(x,y)" to expand your values into two 2-dimensional arrays giving x and y at every point. To make a contour plot use "plt.contour" (you guessed it!). You can make your graph really fancy by superposing an image of the function. Take a look at

http://matplotlib.sourceforge.net/examples/pylab_examples/contour_demo.html

Chapter 3

Designing a Program

3.1 Introduction

This chapter tells you how to plan a program. It gives an example and an exercise for you to complete. You will then be required to design the program that you use in your project. This will be part of the summative assessment for this course.

The text draws heavily on Magnus Hetland's excellent programming tutorial "Instant Hacking"
http://hetland.org/writing/instant-hacking.html
and on Naomi Nishimura's discussion of "pseudo-code"
(http://www.cs.cornell.edu/Courses/cs482/2003su/handouts/pseudocode.pdf).

There are really 3 parts to the process of designing your program:

- Figure out the algorithm you will use. This is best done by writing "pseudo code".
- Decide which bits are best made into separate functions (or methods).
- Decide how to store the data and variables. These are called "data structures". They may be simple arrays, but they might be much more complicated.

The order in which you do these things is not fixed. You'd usually start by writing a pseudo-code description of what your program does. This will naturally suggest some helpful functions and lead you to think about the variables that will be required. As you describe these, you might realise that there are better ways to approach the algorithm etc.

3.2 Why plan a program?

Most beginners simply launch in to writing their Python program, by opening the editor and then typing a few commands. This is OK for simple things (such as the revision exercises), but is a bad idea for something complex like your computing project. Before you start creating any Python code, you need to sit back and think about the problem.

Some of the advantages:

- Separate thinking about what you want the computer to do from how to code it.
- Check that your algorithm really does what you want.
- Make sure your design can be extended easily.
- Make sure your design is structured and makes good use of functions.

Trust me - these things are really important. The last thing you want to do is to write 100s of lines of code only to discover that the algorithm has a fundamental misconception, or that you can't extend you program to do the next part of the problem because of the way the data is stored.

The process of planning your program should be something like this:

- First think about the problem you want to solve. Do *you* understand what you are going to ask the computer to do?
- Sketch out the algorithm that you're going to use. By writing "pseudo-code" you can write an English description of what your program will do.
- You don't have to worry about the syntax of Python or exact command names, so it is very quick. You can easily see what your design does, and write it down without being distracted by the details.
- You can now sit back and check that your algorithm really does what you want.
- You can identify which bits of code are self-contained - these should be functions.
- Does your design repeat similar calculations several times - write a function so that the core code for the calculation is only written and debugged once. *Believe me this saves a lot of time later!*
- What variables does your program need to store? Are these arrays, matrices or just lists? How many elements will they have? Are the variables real, complex or integer? Think up sensible names.
- Will your program write numbers to a file or draw a graph?
- What are the instabilities in your algorithm? What might cause it to crash? How can you avoid this happening?
- How can you check your code works. Now is a good time to think about a test problem (or two) where you know what the answer should be.

Once you've written your pseudo-code, its easy to translate this into Python. Start at the bottom, writing the low-level functions. As you write them, check that they work. Checking as you go along will also save you lots of time.

3.3 Writing pseudo-code

The idea of pseudo-code is to generate a compact and informal description of your algorithm. You lay it out using code-like constructs (eg., if...then; for...) but there's no strict syntax and you can describe the actions in English. You need to strike a balance between being compact and being sufficiently precise about how the algorithm works.

Here are some guidelines (from Naomi Nishimura). There's an example of some pseudo code below:

- Mimic good code and good English. Using aspects of both systems means adhering to the style rules of both to some degree. It is still important that variable names be mnemonic, comments be included where useful, and English phrases be comprehensible (but full sentences are usually not necessary).
- Ignore unnecessary details. If you are worrying about the placement of commas, you are using too much detail. It is a good idea to use some convention to group statements (begin/end, brackets, or whatever else is clear), but you shouldn't obsess about syntax.
- Don't belabour the obvious. In many cases, the type of a variable is clear from context; unless it is critical that it is specified to be an integer or real, it is often unnecessary to make it explicit.
- Take advantage of programming shorthands. Using if-then-else or looping structures is more concise than writing out the equivalent in English; general constructs that are not peculiar to a small number of languages are good candidates for use in pseudo-code.
- Consider the context. If you are writing an algorithm to sort data, the statement "use quicksort to sort the values" is hiding too much detail; if it's just a small part of the algorithm, the statement would be appropriate to use. You can define this function later.
- Separate out self-contained (or repeated) pieces of the algorithm as functions. Specifying the input/output as well as giving the function as name.
- Don't lose sight of the underlying algorithm. It should be possible to see through your pseudo-code to the algorithm below; if not (that is, you are not easily able to check the algorithm works), it is written at too high a level.
- Check for balance. If the pseudo-code is hard for a person to read or difficult to translate into working computer code (or worse yet, both!), then something is wrong with the level of detail you have chosen to use.

3.4 Some Pseudo-Code Examples

3.4.1 Fiesta Spam Salad

I've borrowed this example from Magnus Heltand's "Instant Hacking". Imagine you need to write a computer code to make Fiesta Spam Salad. Here's the recipe (from Hormel Foods Digital Recipe Book, http://www.hormel.com/):

```
Fiesta SPAM Salad

Ingredients:

Marinade:
1/4 cup lime juice
1/4 cup low-sodium soy sauce
1/4 cup water
1 tablespoon vegetable oil
3/4 teaspoon cumin
1/2 teaspoon oregano
1/4 teaspoon hot pepper sauce
2 cloves garlic, minced

Salad:
1 (12-ounce) can SPAM Less Sodium luncheon meat,
    cut into strips
1 onion, sliced
1 bell pepper, cut in strips
Lettuce
12 cherry tomatoes, halved

Instructions:

In jar with tight-fitting lid, combine all marinade ingredients;
shake well. Place SPAM strips in plastic bag. Pour marinade
over SPAM. Seal bag; marinate 30 minutes in refrigerator.
Remove SPAM from bag; reserve 2 tablespoons marinade. Heat
reserved marinade in large skillet. Add SPAM, onion, and
green pepper. Cook 3 to 4 minutes or until SPAM is heated.
Line 4 individual salad plates with lettuce. Spoon hot salad
mixture over lettuce. Garnish with tomato halves. Serves 4.
```

The idea of pseudo-code is to re-write this so that it looks more like a computer might understand it. However, its important that we don't lose sight of what's really going on. Our Psuedo-Code description might look something like this:

```
program make_Fiesta_SPAM_salad
   Marinade the SPAM     (function)
   Cook the SPAM         (function)
   serve the SPAM salad  (function)
finished!

function Maridade_the_SPAM
```

```
      input:  marinade ingredients, SPAM
      output: marinaded_SPAM, reserved_marinade
  collect all the ingredients for the marinade.
  put them in a bag
  shake well
  cut up the spam
  put the SPAM in the bag
  leave in the refrigerator for 30 min
  return:  SPAM,  two spoons of marinade

function Cook_the_SPAM
      input:  SPAM, marinade, onion, pepper
      output: hot salad mixture
   heat reserved marinade in pan
   add spam, onion, paper
   cook for 3 to 4 min
   return:  heated salad mixture

function serve_the_spam
      input: lettuce, hot salad mixture, salad plates, tomatoes
      output:  lunch
   line plates with lettuce
   spoon over salad mix
   add tomato halves
   return: lunch
```

Notice how we were able to split things up using functions so that the basic idea is very clear. However, the analogy is wearing a bit thin now. I'll leave you to think up some data structures. Let's get on with a more relevant example.

3.5 A proper example — projectile motion.

Let's tackle the problem of projectile motion, for example the motion of a cannon shell. We want to plot a graph of the position of the cannon shell as a function of time. If we ignore air resistance, this is a trivial problem, but we want our program to be capable of allowing for air resistance (however, we'll assume the density of air is constant).

We can solve this problem using "Euler's Method" (see "Computational Physics" by Giordiano & Nakanishi for example). In this method, we update the position at time t_i using the velocity at time t_i to get the position at time $t_{i+1} = t_i + \Delta t$.

$$\begin{aligned} x_{i+1} &= x_i + v_{x,i}\Delta t \\ y_{i+1} &= y_i + v_{y,i}\Delta t \end{aligned} \tag{3.1}$$

We also need to update the velocity allowing for the acceleration due to gravity and the effect of air

resistance.

$$\begin{aligned} v_{x,i+1} &= v_{x,i} + \frac{F_{\text{drag}} \cos\theta}{m} \Delta t \\ v_{y,i+1} &= v_{y,i} + \frac{F_{\text{drag}} \sin\theta}{m} \Delta t - g\Delta t \end{aligned} \tag{3.2}$$

where the drag force is given by $F_{\text{drag}} = -A\rho_{\text{air}} v^2$ (where A is constant and ρ_{air} is the density of air). Since v_x, v_y are the x and y components of v, we have $v_x = v\cos\theta$.

[Those of you who studied the Computational Physics module last year will be familiar with improvements to this algorithm, but now we're concentrating on implementation, so we'll stick to Euler's scheme.]

We want to start with an initial velocity and angle, then step along in time plotting the position of the shell until it strikes the ground.

In pseudo-code our algorithm looks something like this:

```
program shell_trajectory
   get initial values                    (function)
   create arrays x,y,vx,vy,t
   initialize variables vx[0] and vy[0]. Set x[0],y[0],t[0] = 0
   choose timestep  delta_t
   loop over time steps:              (Python: "for i in range(max_steps)")
      compute t[i+1], x[i+1] and y[i+1]
      compute air resistance             (function)
      compute vx[i+1], vy[i+1]
      if y < 0 exit loop
   plot graph of y vs x                  (function)

function get_initial_values
      input none
      output initial values of ...
    #  read from file or from keyboard?
    set initial x,y velocity,
    set density of air, constant A, mass of shell.
    return values

function air_resistance
      input: vx, vy, constant A, density of air
      output: Fx, Fy   (x and y components of drag force)
   compute speed        #   v = sqrt(vx**2 + vy**2)
   compute angle        #   theta = arctan(vy/vx)
   compute x and y components of drag force
   return drag force Fx, Fy

function plot_graph
       input: x, y
```

```
      output: none
   open graph window
   plot x, y
   add axis titles/units
   return

Data structures
   A, rho_air, vx0, vy0:   initial values.  (what units?)
   delta_t, max_steps:     parameter values.
   x,y,t,vx,vy:   arrays of real numbers. Dimensions 0 .. max_steps-1
```

That will do for a start. Doing this has forced me to think about exactly how I implement the algorithm, and there are some issues that need a little more work. For example,

- Am I confident that using arctan will give the correct sign for theta if vy is negative? (I'll make a note to check this).
- How am I going to read in the initial values: from the keyboard or a file. Maybe I should have a subroutine to do this so it doesn't clutter up the main code. I should check that the values are sensible.
- What units am I going to use for the calculation?
- How do I decide on a value for delta_t and max_steps? Maybe I need a subroutine to guess these based on the time of flight for the case without air resistance... or I could just ask for these as part of the input data.
- There's a dangerous bug in the example: I need the loop to go i=0 ... max_steps-2 or I'll try to assign a value to a array element x[max_steps], which doesn't exist.[1]
- If I exit the loop because $y[i] < 0$, what values get assigned to the remaining elements of x, y, vx, vy etc? I need to be careful that these are set to zero when I create the array.
- I should also think about how I might extend the code in future. For example: would it be easy to change the program to allow the density of air to vary with height? Could I run the program with several values of the constant A to show how this affects the path?

I should now revise my pseudo-code to take these points into account (best) or add them as comments (second best). I need to be careful not to add too much detail, or I'll obscure what's really going on. Expanding things as functions is a good way to go.

[1] Think carefully about how Python uses array indices and "range" statements: using "for i in range(max_steps)" I'll get "i=0,1,2...max_steps-1", which normally goes with defining the array as dimension max_steps (eg "x=N.zeros(max_steps)"). The trouble is that in this case I assign to "x[i+1]".

3.6 Summary

The steps you should take when you write a program are:

- Separate the overall task into a 3 or 4 parts (functions).
- Now describe how each of these functions work. This might involve adding further functions.
- Decide how the data will be stored and what units it will have.
- Sit back and look at your design. Is it clear how it works? Are there any bugs in it?
- Revise your design to make it clearer. Fix any bugs.

Now you can convert your design into a working program.

- Identify the lowest-level functions (the ones that come last in your design). These are the building-blocks of your program.
- Implement the first of these in Python.
- Check that the function works correctly. For example, you can import your program as a module and then test your function on the command line.
- Move onto the next low-level function. Code it. Test it...
- Now combine the building blocks to make higher-level functions.
- Code it. Test it.
- Finally, code your main program using the high-level functions you've created.

Note the way you work from the top-level down when designing, but from the bottom-level up when coding. It is the difference between an architect and a builder.

Part III

Other Useful Stuff

Chapter 4

Some Help on Scipy Functions

4.1 How to use scipy.integrate.odeint

result=scipy.integrate.odeint(func, solve, time, args=() ....)

Consider a second order differential equation e.g.

$$\frac{d^2x}{dt^2} = f(x) \tag{4.1}$$

By introducing a variable xp which is simply the differential of x this can be replaced by two coupled first order equations:

$$\dot{x} = \frac{dx}{dt} \tag{4.2}$$

$$\frac{d(\dot{x})}{dt} = \frac{d^2x}{dt^2} = f(x) \tag{4.3}$$

One call to **scipy.integrate.odeint** will simultaneously solve these to give x and $\dot{x}$ as a function of t. The parameters you need to supply are as follows:

func - this is the name of a user supplied function (i.e. you write it!) which must return $\frac{dx}{dt}$ (which is just the current $\dot{x}$) and $\frac{d\dot{x}}{dt}$. The inputs to **func** are (in order) **solve**, t, and any extra arguments passed by **args=(...)**. Note that **solve** will contain the current values (i.e. at time t) of x and $\dot{x}$. You should store the return values in the same length of vector as **solve**.

solve - For our simple example, this is a vector (1D array) of two elements containing the values of x and $\dot{x}$ at time $t = 0$ (i.e. the initial values). During the calculation, this is used to store the values of x and $\dot{x}$ at time t.

time - this is simply a 1D array containing all the values of t (in numerical order) at which we wish to calculate the values of x.

args=(...) - any variables inside the brackets (it is a tuple, so they need to be separated by commas) are passed to **func** as extra parameters after **solve** and t (so, for example, you would pass here any constants you need in the calculation of $f(x)$).

On exit, **odeint** will return an $N \times 2$ array, where N is the number of output times you specified (len(**time**)), containing the solved values of x and dx/dt at each time t.

There are other possible parameters which control the internal workings of **odeint** but you probably won't need to use these.

If you have more than two coupled equations then simply increase the length of **solve** to accommodate the extra values you wish to solve for. So if you had a three dimensional problem with coupled equations in x, y and z you would store x, y, z, $\dot{x}$, $\dot{y}$, $\dot{z}$ in **solve** and **func** would have to return $\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}, \frac{d\dot{x}}{dt}, \frac{d\dot{y}}{dt}, \frac{d\dot{z}}{dt}$.

An example

Let's solve the equation for simple harmonic motion: $\ddot{x} = -\omega^2 x$

```
"""this is an example of how to use odeint to solve 2nd order equation.
It stores variables as x[0]=position, x[1]=velocity."""

import numpy as np
import scipy.integrate as SP
mport matplotlib.pyplot as plt


def myfunc(x, t, k):       # must include t, even though it is not used
    """returns dx/dt and dxdot/dt for SHM. k is the spring constant"""
    return  (x[1],  -1.0 * k**2 * x[0])

def solve_it(t, k):
    """solves the equation and returns solution"""
    solve_me = np.array( [0., 1.])              # start position and velocity
    return SP.odeint( myfunc, solve_me, t, args=( k, ) )  # note "," after k

if __name__ == "__main__" :
    const = 2.0
    times = np.arange( 100.0 ) /10.
    answer = solve_it( times, const )
```

```
plt.plot( times, answer[:,0] )    # plot position vs time
plt.plot( times, np.sin(const*times)/const +0.05)  # analytic solution, shifted up
plt.show()
```

4.2 How to use scipy.linalg.eig

result= scipy.linalg.eig(M))

A Matrix **M** has associated eigen values, λ and eigen vectors **E**, that are defined by

$$\mathbf{M}.\mathbf{E} = \lambda\mathbf{E}.$$

Scipy provides routines to calculate λ and **E**. For example, if

$$\mathbf{M} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

We can compute the eigenvalues and vectors eig(M). This returns a list of eigenvalues and a list of eigenvectors, as the following example makes clear.

There is no particular order to the eigenvalues, you may need to use argsort to find the smallest one, or to put them in the right order.

```
""" example of how to solve eigen values and vectors from set of
linear equations."""

import numpy as np
import scipy.linalg as la

M = np.array( [ [1., 2.], [3., 4.] ] )

(eval, evec) = la.eig(M)

# note that the returned eigen value/vector may be complex.
# in this case, it is real, so uncomment if you wish...
##  eval = np.real(eval)

isort=np.argsort( np.abs(eval) )     # sort by magnitude of eigen value

i = isort[0]
print "smallest eigen value and vector", eval[i], evec[:,i]

# check that it really works.
check = np.dot(M, evec[:,i]) - eval[i] * evec[:,i]
```

```
print "    solution accuracy = ", check

i = isort[1]
print "largest eigen value and vector", eval[i], evec[:,i]
# check that it really works.
check = np.dot(M, evec[:,i]) - eval[i] * evec[:,i]
print "    solution accuracy = ", check
```

Some things to note. The eigen values/vectors may be complex. Use eigh if your matrix is symmetric etc. If you do not want to compute the vectors, just the values it may be simpler to use the eigvals or eigvalsh functions.

Numpy includes routines to manipulate matrices. For example numpy.dot(M, N) will multiply matrices M and N. We can use this to check the eigen values/vectors really work. There are other important matrix functions too: inv takes the inverse of a matrix, det computes the determinant, etc. This is very powerful: matrix equations are solved very easily. More help is available at

http://docs.scipy.org/doc/scipy/reference/linalg.html

4.3 How to use Spherical Bessel Functions

Scipy provides many different special functions, including routines to calculate Bessel functions and their derivatives. These are briefly described at

http://www.scipy.org/SciPyPackages/Special

In the resonant scattering project, you need to be able to calculate the Spherical Bessel functions and their derivatives. The recommended way to do this is to use the sph_jnyn function. Here is an example to show you how to do this.

```
"""Example of how to compute the spherical Bessel functions and
derivatives"""

import numpy as np
import scipy.special as ss

order = 5                       # order must be an integer
radius = 1.5                    # radius at which to evaluate

# sph_jnjy return a list of arrays, I use numpy to turn it into a
# 2d array. It returns all the orders up to 5, you have to select
# the one you want.

jnyn = np.array( ss.sph_jnyn( order, radius ) )
print "this creates an array of shape ", np.shape(jnyn)
```

```
print "Computed Bessel functions of order 5 at radius %5g" % radius
print  "value       j5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 0, order ] )
print  "derivative j'5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 1, order ] )
print  "value       y5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 2, order ] )
print  "derivative y'5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 3, order ] )
```

Chapter 5

Some Common Traps and Pitfalls in Python

5.1 Using Unix

1. **You will need some basic unix commands.** Use the CIS help sheet to find out the basic UNIX commands. For example "ls" lists the files in your directory; "cd" changes directories.
2. **Use the & symbol.** If you put an & symbol at the end of a line (eg., "idle -n &"), it will run in a background mode and you will be able to type other commands in to you terminal window.
3. **Unix command history.** Unix remembers the commands that you type in, you can use the "up-arrow" key to go back to old commands. This saves a lot of typing. Unix can also try to guess a command you type: use the "tab" key.

5.2 Using Python

1. **How do I exit from the Python command line?** `quit()` will quit! but it needs the `()`! Ctrl-D should also work.
2. **Obscure looking error messages.** The error messages generated by Python can look a bit obscure. However, they tell you the line numbers in each module (some of which may be from numpy or pyplot) where the problem occurred. Start by looking for your module/function name. Find the line number and go to that line in your editor. Then read the error message carefully. For more help on what the message means, and how to debug your code, look in the Level 2 "Introduction to Programming in Python" guide.
3. **Using IDLE.** IDLE is helpful but not essential. The editor is great, but the development environment sometimes does things strangely. It's also possible to cut and paste your program one line at a time into a command line terminal. *However, beware of blank lines in loops when you cut and paste:* these are taken to be the end of the loop. It is best to avoid them when doing this. You should also be careful how you use comment strings `""" ..... """` which can span several lines. Once code

developed in this way works, paste it into a file and run it by typing `python my-file`. **Important: using pyplot/pylab from within IDLE may cause a crash on the CIS system.** Run you program from the command line instead.

4. **When importing a module.** When you `import` some routines, they will not be imported again if you type `import ...` a second time. This causes problems if you use IDLE: if there's an error in the routines you import that you then fix. The fix is ignored! So, be sure to exit IDLE and start again.

5. **Avoid giving a file the same name as a module.** Python looks for files in your directory before it searches the rest of the computer. So if you create your own file called "numpy.py" you will not be able to import the standard module!

6. **Getting help.** There's lots of help on the web. Try typing
`python myErrorMessage` into Google to figure out what an obscure error message means. Typing `help('someFunctionName')` at the command line will give you useful help too. Try `help('numpy')` for example.

7. **Python is case sensitive** so `a` and `A` are not the same variable. But never use `a` and `A` in the same program. Stick to a consistent way of using lower and upper case letters.

8. **Keep a backup!** Make sure you keep a back-up of old versions of your code. This way, you can always hand in your last working version at the milestone interview.

5.3 Using Pyplot

1. **Using pyplot/pylab from within IDLE sometimes causes a crash on the CIS system.** Run you program from the command line instead.

2. **Interactive Plotting** using p.ion() and p.ioff() doesn't always seem to work on the CIS X-windows system.

5.4 Programming Python

1. **You must define functions before you use them.** The best strategy is to put your function definitions into a module file and then import them from your main program.

2. **When assigning values from function,** you need the `()` or else it generates a link to the method. `a = random` is not what you meant!

3. **Note the standard structure of loops in Python**: you need to use `range(n)`. Note that indices go from 0 to n-1. This is not so much of a problem in itself, but could be confusing compared to other languages. Get used to the python way and use `for i in range(n)`, NOT `for i in range(1,n+1)`! This way of doing things is consistent with the way Python uses arrays.

4. **Take care with array slices.** If you want to access part of an array, remember that `A[3:5]=0.0` will only set `A[3]` and `A[4]` to zero. This is consistent with the way python loops work (to see this, realise that `A[0:n]` means the first n elements of the array).

5. **Some variable names unexpectedly conflict with reserved functions/keywords.** For example `map`. When this happens, the error just says "Syntax error" so it's not terribly obvious what's going wrong!

6. **What's the difference between "xxx" and 'xxx' ?** There isn't any. But the quotes do need to match. `"""` allows a comment to run over several lines. This is useful for doc strings.

7. **The two forms of import can be confusing.** We strongly encourage use of `import numpy as np`. This is much better than `from numpy import *`. The first version makes it obvious which functions are external to your program.

8. **Reading numbers from a file.** This is a bit long-winded if you write this yourself. However, numpy provides lots of routines to read files in various formats. The most useful one is `np.loadtxt` which reads from a text file. It will put all your numbers in a 2d array. If you really want/need a "diy" approach, the trick is to read the line of the file as a string and then convert it to a list of numbers using `string_to_float`, which I defined as

```
def string_to_float(line):
    """convert comma separated string to list of numbers"""
    return map(float, line.split(","))
```

9. **It is dangerous to compare two floating point numbers to see if they are equal**. Because floats have limited accuracy, they may be very slightly different. You really want to see if they are equal within some tolerance.

10. **Be careful to type `a==b` when comparing**, not `a=b`. The later will give a syntax error if you try to use it in an `if` statement. (Python is a lot safer than C in this respect!)

11. **Beware of integer division.** Remember that Python version 2 gives an integer result when you divide two integers, thus "3/2" gives 1, not 1.5. You will force the answer to be a float if either of the numbers is a float. Thus "3/2." gives the expected answer. To avoid this trap, enter constants with an explicit decimal point (unless you really mean them to be integers). Thus a cosmology program will contain lots of statements like `a = 1./(1.+z)`.

    The alternative is to declare

    from __future__ import division

    at the top of your code. This will force all divisions to return the correct floating point result.

12. **Be careful with global variables.** You can safely use scalar values: they can be read inside a function, but cannot be written to. This can be a good way to define things like the gravitational constant G **!!! BUT !!!!** if you try to use global arrays (or other structures), you can inadvertently alter the contents.

13. **Be careful of how you copy arrays.** If you want a copy of array A, use `B = A.copy()`. I suggest you experiment with what happens if you use `B=A` instead. For example, try `A[0]=-99`, and then `print B`. Now you will remember to be very careful!!! It is safe to use things like `C = A + B`, but *not* (surprisingly!) to do `B = A[2:4]`.

14. **Use module and function doc strings.** These are very, very useful. For example, you can print out a description of the function in a module by typing `help('myModule')` or `print myModule.__doc__`.

15. **Keep functions short.** Use functions to break your code into small chunks which have a clear purpose.

16. **Make your program a module.** By using `if __name__ == '__main__'` you can include some test code at the end (for example, to solve your milestone problem).

5.5 Debugging a program

1. **The easiest bugs to fix are those that cause a syntax error.** Find out which line it happens on. *Read the error message carefully.* See the Level 2 "Introduction to Programming in Python" document for more help and examples.

2. **The next easiest bugs** to fix are ones that cause your program to stop with an exception (for example, I'll get `ZeroDivisionError: float division` if I try to divide a number by zero). Find out which line it happens on. Add a print statement before it to check the values of relevant variables.

3. **The hardest bugs to fix** are those where your program runs but gives the wrong answer. This is really bad (but quite normal!). It's easier to find these bugs if you break your program into short functions. You can then test each function one-by-one on the command line, making sure it does exactly what it's supposed to. In this way you slowly build up the full program. If you design your program before you start typing, it is much easier to test as you go.

4. **Never, ever write the same piece of code twice.** If you have two snippets of code that do the same thing (or almost the same thing) you will fix the bug in one, but forget to fix it in the other. [I know].

5.6 Python Philosophy

1. **build your program out of short functions.** Test them as you write them. Create modules from collections of related functions. Import your module(s) into your main program.

2. **Examples should use `spam` and `eggs`**, not `foo` and `bar`. (if you are explaining how your program works).

3. **Always look on the bright side of life.**