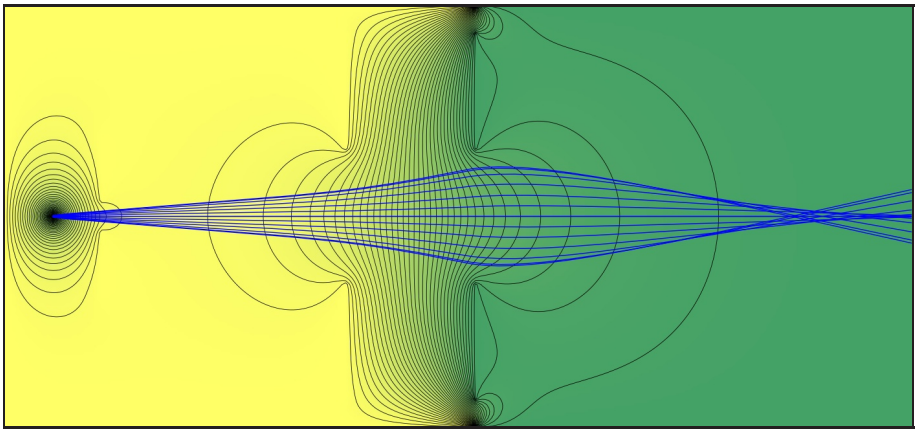
PHYS2022

Physics From Evidence I

Computing module 2020-21



Cover Image

This is a contour plot of the electrostatic potential in an electron gun. The superimposed blue lines are the tracks of electrons emitted from the cathode at the left hand side. The potential was determined by the same method used for a grid of resistors in section 7. The tracks were found by solving the equations of motion for electrons emitted from the cathode. Everything was computed and displayed using Python with the numpy and matplotlib.pyplot modules.

COMP PY 20200930 DKW
D. K. Whiter, based heavily on the original by Jonathan Flynn.
Department of Physics & Astronomy
University of Southampton

Typeset using LaTeX

Contents

Computational Physics — Introduction

In addition to everyday usage, physicists use computers for:

> controlling apparatus and taking readings automatically,
> processing data to extract conclusions,
> calculations to predict the results of experiments,
> simulating systems where analytic solutions are impossible or unknown.

This course introduces all these applications except the first. It is not primarily a course in programming and you should be able to manage just fine even if this is your first experience. The most important thing for you to learn is the idea of taking a problem and devising an *algorithm* for its numerical solution, that is, a recipe which a (dumb) computer can follow.

Once you have an algorithm you can implement it using *any* programming system, ranging from a spreadsheet to a traditional scientific language like FORTRAN, or perhaps C++ or Java. Of course you have to choose *some* language and for this module we have chosen Python. Although not originally designed for scientific use, it is perfectly adequate for the tasks you will undertake here. Python's high-level language features mean that programs will typically be much shorter than equivalent ones written in FORTRAN, C or C++ and that you should quickly start to be able to calculate useful results. As you acquire some familiarity you may well find that Python is great for the small data-processing and calculational tasks that you will meet throughout your studies for a physics degree.

Once you have a grounding you can extend your knowledge as you wish. For example, you could learn object-orientation using Python. Keep clear, however, the distinction between an algorithm or method of solution and its incarnation in a particular language. All programming languages implement the same basic constructs, loops, decision-making and so on. Once you have seen them in one language it will be much quicker to pick up a different language later if you need to (as you almost certainly will). Do not get too hung up about the choice of language.

We will introduce you to the Scientific PYthon Development EnviRonment (Spyder). This allows you to use Python interactively, in what's called *shell mode*, somewhat like

a very powerful programmable calculator: you type instructions and the interpreter processes them immediately. An alternative is to develop a script (program) for the entire solution which you save in a file and then ask the Python interpreter to execute in one go. You could create the file with any text editor, but Spyder provides convenient editing facilities to help you with this. Later on you may find that you will develop your own Python functions which you then call from a script or in an interactive shell-mode session in order to calculate your numerical answers.

Note that Python has evolved over time, and there are currently two major versions: Python 2 (recently deprecated) and Python 3. Code written for Python 3 is not necessarily backwards-compatible with Python 2. **In this course we will use Python 3.** If you have learnt Python 2 you should have no problems adapting to Python 3.

How Marks Are Awarded

This course uses fully electronic coursework submission, via the ECS "Handin" system. There are numbered questions throughout these notes, for which you will receive marks. Some questions require you to write a sentence or few to explain a point, in which case you should record your answers electronically. You should write all of your answers to these questions in a single document (per section), which you should submit as a **.pdf** file. You may include screenshots in your electronic lab book, but **please do not include too many, as they will make your file too big for Handin**. You might want to use your notes and answers as a reference when you do your final year research project. When considering how to record your answers, ask yourself "will I be able to look at this in a year's time and understand what I did?"

Questions which require you to write the answers in your electronic lab book use black text on a light blue background, like this:

Q0.1 How many logs would a groundhog lob if a groundhog could lob logs? **[1]**

Other questions require you to write a Python script, in which case you should submit the **.py** file which you create (usually one per question). You may also need to submit other files, such as graphs created by your scripts. Questions which require you to submit a Python script and/or other files use white text on a teal background, like this:

Q0.2 Create an elegant Python script and submit it to Handin. **[3]**

The marks for each question are given in square brackets after the question, with a total of 20 for each section (see below).

You can submit your work as many times as you like up until the deadline, but only the last version which you submit will be marked. The Handin system will do some automatic checking of your Python code as soon as you submit it, and will email you with the results of these checks. If it detects a problem you will be able to modify your

code and resubmit your work, provided it is done before the deadline. The aim is to help you to learn from your mistakes, but be aware that the system cannot detect all errors or bugs, and is **not a substitute for your own bug hunting**.

Your work will be sent to the demonstrators automatically as soon as the deadline has passed. They will mark your work and also get you to explain your answers and your programs. **It is a good idea to keep copies of the programs that you write.**

There are eight numbered sections in these notes, and you should aim to complete one section in every practical session. Each section is worth 12.5% of the total marks for the module. It is better to complete *most* of a section and move on than to complete the *whole* section and fall behind. You are, of course, free to work on the exercises outside the scheduled teaching sessions. It is essential that you **come to your marking slot, on time (this year on Teams)**. The demonstrators will not be able to catch up on a backlog of marking at the end of the module.

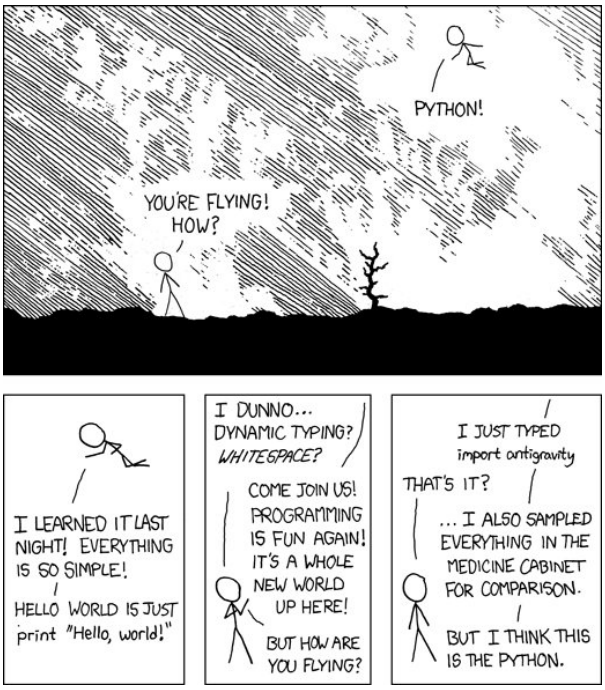
The mark awarded is based on your written answers and the understanding you display in describing how you found these answers. The division of the marks is as follows:

20 marks	Correct and detailed answers to all questions.
20 marks	Knowledge/understanding displayed in discussion with demonstrator.
5 marks	Presentation: clear code, clear notes, comments in code, etc.
5 marks	Bonus: extra investigations or code features.

The total mark awarded is out of 50.

The “understanding” marks are not linked to specific questions, but your mark will be capped if you do not attempt all of the questions in the notes - e.g. if you only try for half of the “question” marks, you can only get a maximum of 10 for understanding. You will not score full marks if you cannot explain what you have done.

The bonus marks are intended to reward you for extending your investigations or code above and beyond that required for the questions, and are there to stretch the best students. The demonstrators are instructed to award bonus marks sparingly, and only if the extra work is correct, logical, and beneficial, but you cannot lose marks in the other categories due to poor bonus work.



XKCD webcomic: http://xkcd.com/353/

Acknowledgement:

Thanks to Jonathan Flynn and to Geoff Daniell for producing the original physics computing module.

Section 1

Python Primer

In this section we'll take our first steps in using Python [1].

To begin with we'll use Python interactively in a Python Shell. In Spyder this is usually the window in the bottom right, with some messages at the top, ending in a line with a prompt >>>. The prompt is a signal for you to type something which will be **executed** (or **interpreted**) when you hit the "return" key.

1.1 Numbers

You can use Python as a calculator. Here are some examples:

```
>>> (1+1+2+3+5)*5-24/3
52.0
>>> 9**2 # this is a comment: use ** for exponentiation
81
```

In the second example we used a *comment*: the interpreter ignores everything from a # to the end of a line. Try some examples of your own.

Now compare the two examples above to the following:

```
>>> (1.0+1.0+2.0+3.0+5.0)*5.0-24.0/3.0
52.0
>>> 9.0**2
81.0
```

[1] This tutorial section borrows from the Python tutorial, accessible on the web at http://docs.python.org/tutorial/.

The answers are the same. So far so good. But now try:

```
>>> 12.0/5.0
2.4
>>> 12//5
2
```

The lesson is that computers distinguish between exact integer arithmetic and real number (or floating point) arithmetic. In particular, when using integer arthmetic and dividing two numbers does not result in an integer, the result is truncated (rounded down). Many programming languages assume you want integer arithmetic unless you define the numbers as floats, but Python 3 assumes floating point arithmetic and provides the `//` operator for integer division. If you mix integer and floating point numbers, they are all converted to floating point:

```
>>> 34+1.0
35.0
```

You can also convert from integer to float and back again, but note that conversion to integers is done by *truncating towards zero*

```
>>> float(1)
1.0
>>> int(2.71828)
2
>>> int(-3.14159)
-3
```

1.2 Assignments, strings and types

You can **assign** values to **variables**, but now you don't see a result before the next prompt is displayed.

```
>>> x=3
>>> y=2
>>> z=7
>>> x*y*z
42
```

Note that `x=3` should not be read as a normal mathematical equation. Instead it means "assign the value 3 to the variable called `x`". If you write `y=y*y` the computer first evaluates the right hand side using the current value of `y` and then assigns this as the new value of `y` (whereas in mathematics, solving the equation $y = y^2$ would tell you that $y = 0$ or $y = 1$).

As well as numbers, you can use **strings**, which can be joined (**concatenated**) by adding them:

```
>>> 'spam'
'spam'
>>> 'spam,' + ' ' + 'eggs and ham'
'spam, eggs and ham'
```

Strings can also be duplicated by multiplying them by a number, and they can be enclosed in single or double quotes:

```
>>> "spam, "*4 + "eggs and ham"
'spam, spam, spam, spam, eggs and ham'
```

You can ask what **type** a variable is, and you can print its value:

```
>>> a=1; b=1.0; c='1.0'
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
>>> print(a, b, c)
1 1.0 1.0
```

1.3 Complex numbers

Python supports complex numbers. They are written in the form `3+5j`, corresponding to $3+5i$ in usual mathematical notation. Here are some simple examples using complex numbers.

```
>>> d=3+5j
>>> type(d)
<class 'complex'>
>>> d.real
3.0
>>> d.imag
5.0
>>> abs(d) # absolute value = magnitude = modulus
5.830951894845301
>>> e=1+1j
>>> print(d*e)
(-2+8j)
```

1.4 Getting help

Python has many ways to get help. From Spyder you can access Python documentation from the "Help" menu item. Within the shell, try:

```
>>> help()
```

Type quit to exit help. Don't forget that you can use `type()` to find the type of an object and `print()` to display it. There is also `dir()`, which may be more useful when you know more Python (use `help(dir)` to find out more).

1.5 Lists

You can create a **list** of objects. The objects in the list don't have to be of the same type, and could even be other lists:

```
>>> a=[3,54,26,90]
>>> b=[3,54,'llama',a]
>>> print(b)
[3, 54, 'llama', [3, 54, 26, 90]]
```

The elements of a list are numbered from zero and you get an error if you ask for an element that's out of range. It's worth repeating that *the elements of a list are numbered from zero*, so a list with 4 elements, like `a` above for example, has elements numbered 0, 1, 2 and 3. Here are some examples of accessing list elements:

```
>>> a=[3,54,26,90]
>>> b=[3,54,'llama',a]
>>> print(b)
[3, 54, 'llama', [3, 54, 26, 90]]
>>> a[0]
3
>>> b[3]
[3, 54, 26, 90]
>>> b[3][2] # interpret as (b[3])[2]
26
>>> b[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can also access the list elements from the right: the index -1 refers to the last element in the list. Python can tell you the length of a list (and of a string) using `len()`. Finally, you can change elements of a list (but not of a string). We illustrate these things here:

```
>>> a=[3,54,26,90]
>>> b=[3,54,'llama',a]
>>> a[-1]
90
>>> len(b)
4
>>> len(b[2]) # length of string 'llama'
5
>>> a[2]='gecko'
>>> a
[3, 54, 'gecko', 90]
```

Q1.1 Using `a=[3,54,26,90]` and `b=[3,54,'llama',a,3.5]`, what are `b[-3][3]` and `a[-0]`? **[1]**

1.6 Tuples

There is another sequence data type in Python called the **tuple** which comprises a number of values, separated by commas. The code below shows how to create a tuple. On output, tuples are enclosed by parentheses, and you may have to use them for tuple input if you need to resolve ambiguities. Tuples are **immutable**, which means that you can't make assignments to individual elements of a tuple, as shown below.

```
>>> t=123,'trout',12.5,'mouse'
>>> type(t)
<class 'tuple'>
>>> t
(123, 'trout', 12.5, 'mouse')
>>> t[3]
'mouse'
>>> t[2]=34
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

We will use tuples from time to time in this course.

1.7 Towards programming

Often you may want to do something a fixed number of times, or perhaps you will perform some calculations as long as a particular condition is satisfied. You may also want to decide what to do based on some test. All programming languages contain standard structures to do these operations. Here's how Python implements them.

1.7.1 Iteration or "for loops"

Python can iterate over the items in a list (or characters in a string).

```
>>> animals=['lemur','tapir','shark','dingo']
>>> for a in animals:
        print('I have a pet ' + a)


I have a pet lemur
I have a pet tapir
I have a pet shark
I have a pet dingo
>>> for a in 'sloth':
        print(a)


s
l
o
t
h
```

Note that the `for` statement ends with a colon (`:`). Following this, the statements to executed (here just one) must be indented. This is how the interpreter can tell what to do. In the Spyder interactive shell, the indentation following a colon is automatic, and continues for subsequent lines. Hence, when you enter a for loop interactively you have to type return twice before your loop is executed: hitting return once gives a new indentation; hitting return again shows that you have finished your input.

"For loops" are used when you want to repeat something a fixed number of times. In scientific calculations we commonly want to loop over a list of numbers. Moreover, these numbers will often be a set of indices. Python's `range` function is very useful for this. For example, `range(10)` will give a set of ten integers from 0 to 9. Here are some more examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(5,20,2))
[5, 7, 9, 11, 13, 15, 17, 19]
```

Type `help(range)` for help.

1.7.2 List comprehensions

Often you will want to create a new list by doing something to the elements of an existing list. You can do this using a for loop as in the following example. Here

we create a list `a`. Next we create an empty list with `b=[]` and then run through the elements of `a`, successively adding (appending) the exponential of each element to `b`:

```
>>> from math import exp
>>> a=[0.0,1.0,2.0]
>>> b=[]
>>> for x in a:
        b.append(exp(x))
>>> b
[1.0, 2.718281828459045, 7.38905609893065]
```

However, Python provides a convenient way to create lists from lists using a **list comprehension**, which we illustrate by continuing the example above:

```
>>> c=[exp(x) for x in a]
>>> c
[1.0, 2.718281828459045, 7.38905609893065]
>>> b==c
True
```

The list comprehension is quite readable and short, but the first method above might be preferable if you wanted to do some complicated calculation before appending each new value to `b`.

Note too how we used `==` to test for equality (see also section 1.7.4).

1.7.3 "While loops"

In a "while loop", you keep performing an instruction or sequence of instructions, called the *body* of the loop, but always checking *beforehand* that some condition is true. Typically the condition is true when you first meet the loop, but after going round the loop some number of times, the condition becomes false. The interpreter then skips the instructions in the loop and moves on. If the condition is false when you first reach the while loop, the instructions in the body are *never* executed.

We illustrate a while loop by calculating the factorials of the first 9 positive integers.

```
>>> fac=1;i=1
>>> while i<10:
        fac=fac*i
        print(i, fac)
        i=i+1
```

Note that we combined two statements on one line, using a semicolon (`;`) to separate them. The condition for this loop is whether `i<10`, while the body comprises the three indented lines following the `while` line.

Q1.2 Run this code and write a short explanation of how it works. **[3]**

The factorial example above could also have been done using a for loop, since we knew in advance how many factorials to compute. While loops are really useful when some instructions need to be repeated but we don't know in advance how many times to repeat them. We will meet examples later in the course.

1.7.4 "If" statements

These are illustrated by an example. Read the text following the code for some explanation.

```
>>> cheese=input("Enter a cheese name: ")
Enter a cheese name: Wensleydale
>>> if cheese=='Brie':
        print("It's a bit runny, Sir.")
elif cheese=='Limburger':
        print("It's really **very** runny, Sir.")
elif cheese=='Wensleydale':
        print("Cracking cheese, eh Gromit?")
else:
        print("Is that a cheese?")


Cracking cheese, eh Gromit?
```

The `if` construction can have zero or more `elif` parts, while the `else` part is optional. If you have a single statement to execute, you can write it after the colon, on the same line, like this:

```
if ilikecheese==True: print('Yippee!')
```

The example illustrates some more things.

- First `input` shows how to read user input. This produces a string. If you wanted a number you would use `float()` or `int()` to convert the string to a floating point number or an integer.

  ```
  x=float(input("Please enter a number: "))
  ```

  Do NOT include `input` in any code you submit to Handin - the system will run your code, but there is nobody there to type anything for you!

- When testing for equality, use the == operator, not = which is assignment.

- Typing a long `if... elif... else...` sequence is hard work. Once you have more than a few lines of code, it begins to make sense to save your code in a file which you can ask Python to read and interpret for you, as we'll discuss in section 1.10.4.

1.8 Functions

If you are going to evaluate the same expression or perform the same sequence of operations many times in your program, it would be tedious and error-prone to keep typing in the same code. It is more efficient to define a **function**. Functions can also help make your code better structured, which means it's easier to write and understand and therefore more likely to be correct.

Here's how to define a function to evaluate the polynomial $x^3 - 7x^2 + 14x - 5$ in Python.

```
def f(x):
    return x*x*x-7.0*x*x+14.0*x-5.0
```

Note that calling the variable `x` in this definition has no special significance. In particular, it will not conflict with a variable `x` used *outside* the function definition. We could also have used any other allowed variable name instead of `x`.

The above is about the simplest function definition you could make. To learn more, a good place to start is the Python tutorial, accessible through the "Python documentation" Help menu item in Spyder. One immediate improvement is to add some documentation to the function:

```
def f(x):
    """Evaluate polynomial x^3 - 7 x^2 + 14 x - 5
    """
    return x*x*x-7.0*x*x+14.0*x-5.0
```

Now if you define this function and type `help(f)` in the interactive shell in Spyder, a message should appear containing the comment added above. If you come back to a function after some days, months, or years, such comments can prove really helpful.

1.9 A note on names

Variable and function names in Python can use upper and lowercase letters, digits and underscores. However, some names can't be used. You will have realised that you cannot use names like `print`, `else` and so on. Python has around 30 **keywords** which cannot be used as identifiers (you can find the list in the Python reference manual).

Names with leading or trailing underscores have special meanings, so you should avoid names of this form (at least until you learn more).

Finally, don't use names which are already used elsewhere for functions (or variables). This may be possible, inside your own function definitions for example, but is potentially confusing.

If a word changes colour when you type it in Spyder, that is a sure sign that you should not use it as a variable or function name yourself.

1.10 Modules, scripts and importing

When you quit the Python interpreter you will lose any definitions you've made. If you have a substantial amount of input, you are better off preparing it in a text editor and saving it as a file. You can then run the interpreter with that file as input. This is called creating a **script**, the equivalent of writing a program in a language like FORTRAN, C or C++. If you discover any mistakes you can go back to edit the file and try again. Another possibility is that you have defined a very useful function which you'd like to use in many programs without copying it into each program: this is called **importing**.

In the Python world, a file containing definitions and/or executable statements for use as a script either in its own right or to be loaded into an interactive session, is called a **module**. The file name is the module name with the suffix **.py** added. I will tend to use *script* for saved Python code intended to be used as a complete program, whose output you want to know when it's run through the interpreter. On the other hand, I will tend to use *module* for saved function definitions which can be imported into other scripts or into an interactive session.

We'll look first at importing from modules provided with Python before discussing creating our own scripts and modules.

1.10.1 Standard modules

Python comes with a library of standard modules. These give access to operations that are not part of the core Python language. Some of them are already built-in to the interpreter, but others have to be called for explicitly by **importing** them. Likewise you can import modules you have written yourself, or otherwise acquired. It's also possible for modules to import other modules.

1.10.2 Importing mathematical functions

Standard mathematical functions, like sin, log, sqrt and so on are not part of the core language, but they are provided as part of the standard library in the **math** module. We'll show several ways to access a module, using **math** as an example.

- Import the module. You can then use objects from the module:

```
import math
print(math.sin(0.5))
```

- Use some abbreviation:

```
import math as m
print(m.sin(0.5))
print(m.pi)
```

This example also shows that the **math** module contains an alias for π, referred to as `pi` in code.

- Import the objects you need explicitly:

```
from math import sin, pi
print(sin(0.5))
print(pi)
```

- You can import all objects from a module using `*`:

```
from math import *
print(cos(pi/3), log(54.3), sqrt(6))
```

The `import *` form is convenient but has a potential downside. It is possible that you might import functions with the same names but different effects from different modules. The last definition imported will take precedence. For this reason, it's good practice to use the other forms of `import` shown above. The second method using abbreviation is nice because you can tell from which module a function is imported, but you save some typing.

When illustrating Python code we will sometimes show an `import` statement to indicate a module that we depend on.

1.10.3 Modules for PHYS2022

Some useful functions have been defined in modules for this course. They are in the files called **minimise.py** and **neutrons.py** (both available on Blackboard). You will see these being used later in sections 3.3 and 8.

1.10.4 Making scripts

As a simple example of a script, consider the long `if` statement on cheese choices used earlier. If you are using Spyder, the "File → New file..." menu option opens up a new text editor window. You can type or paste your code into this window. Put the following code into an editor window and save it, remembering to be careful about indentation:

```
cheese=input("Enter a cheese name: ")
if cheese=='Brie':
    print("It's a bit runny, Sir.")
elif cheese=='Limburger':
    print("It's really **very** runny, Sir.")
elif cheese=='Wensleydale':
    print("Cracking cheese, eh Gromit?")
else:
    print("Is that a cheese?")
```

Now press the play button (green triangle) on the toolbar, or press F5, to get Python to run your script. If all is well (and you are using the script above), you should see the prompt `Enter a cheese name:` in the Python Shell window.

Q1.3 Write a short script which uses a `for` loop iterating over a `range` to produce a simple table of the first ten positive integers, their squares and cubes, in 3 columns. Save your script as a **.py** file and submit it with your answers to this section. **[4]**

1.10.5 Your own module

To illustrate making a module, we'll use the polynomial function defined earlier.

If you are using Spyder, the "File → New file..." menu option opens up a new text editor window. You can type or paste your code into this window. Then save the file, giving it a **.py** suffix. Put the following into an editor window

```
def f(x):
    """Evaluate polynomial x^3 - 7 x^2 + 14 x - 5
    """
    return x*x*x-7.0*x*x+14.0*x-5.0
```

and save it as **cubicpoly.py**.

If this is to be used as a module, then Python has to be able to find it. Python has a list of directories (folders), called the **module search path** where it looks for files to import. If you keep all your own Python code in the same directory (folder), then you can add this directory to the search path, as described in appendix A.3.

Now, in the Python Shell window, you should be able to import the module and use your function with

```
import cubicpoly
cubicpoly.f(3)
```

or

```
from cubicpoly import f
f(3)
```

or

```
import cubicpoly as cp
cp.f(3)
```

Q1.4 Write a short script which uses the **cubicpoly.py** module you have created to print the value of $f(\pi)$, and submit your short script for marking. Note you do **not** need to submit **cubicpoly.py**. **[3]**

1.11 Pitfalls of numerical computing

You should be aware that you can get incorrect results from a computer calculation, even when your program runs correctly. This occurs because of the finite precision with which numbers are stored in a computer, often referred to as *rounding errors*, and because mathematical operations involving limiting processes (such as differentiation, integration and summing infinite series) have to be approximated by a finite number of operations, referred to as *truncation* errors. Obviously we want to minimise the effect of such errors.

1.11.1 Machine accuracy

Here is a function to compute square roots:

```
def square_root(x):
    y=x
    while y*y!=x:
        y=0.5*(y+x/y)
    return y
```

You need not understand why it works but verify that it calculates the square roots of 4, 9 and 100 correctly.

Note that a new variable `y` has been defined and used inside the function. This `y` is **local** to the function definition and will not conflict with things named `y` used elsewhere. Note too that the combination `!=` denotes *not equal*.

Use the `square_root` function to try computing the square root of 2. The computer will apparently do nothing and you will not get an answer. To get out of situations like this in python, type Ctrl-c, which should interrupt the calculation.

When things like this happen you need to examine the values of variables like `y` during the calculation. Insert a statement `print(y, y*y)` inside the `while` loop in the function definition. Look at the output when you compute `square_root(4)` and then try `square_root(2)` again.

Q1.5 Describe as precisely as you can *what* is happening and *why* you think this is happening. How might you improve the `square_root()` function? **[4]**

1.11.2 Special cases causing problems

Try out the following:

```
from math import sin
x=0.0
while x<3.0:
    print(x, sin(x)/x)
    x=x+0.25
```

You will find that you get a "division by zero" error. Although $\sin(x)/x$ has a well-defined limit as $x \to 0$, it requires deeper mathematics than the computer's simple arithmetic to deal with this.

If your problem required you to evaluate $\sin(x)/x$ with the argument x likely to take the value 0, then you could define a function which checked for this possibility:

```
def sinexoverx(x):
    if x==0:
        return 1
    else:
        return sin(x)/x
```

1.11.3 Good and bad function definitions

Here we will see how mathematically equivalent expressions can behave differently when evaluated numerically with finite precision. Define the functions:

```
from math import sqrt

def a(x):
    return x-sqrt(x*x-1)

def b(x):
    return 1/(x+sqrt(x*x-1))
```

If you do the algebra, you can show that $a(x)/b(x) = 1$ so the functions should be the same. Check this numerically by getting the computer to print the values of `a(2)` and `b(2)`.

Q1.6 Now tabulate $a(x)$ and $b(x)$ for x between 150000 and 150009 in steps of 1.0, using a `for`-loop, and record the table in your electronic lab book. **[2]**

Notice that the two values are different! Which one, if either, is right?

To understand this further you need to know how the computer stores numbers. Imagine the number is written in the form $\alpha 2^{\beta}$ where α is between a half and one and β is a positive or negative integer. For example 3 is 0.75×2^2 and 0.3125 is 0.625×2^{-1}. The computer stores the number α, *truncated* if necessary after a number of digits, and the integer β. By storing α and β in this way a wide range of numbers can be stored in a small number of binary digits. Clearly there is a limit to the precision available (from α) and also to the range (from β). Numbers stored in this way are called **real** or **floating point** numbers.

In Python, the precision of floating point numbers is about 1 part in 10^{16}.

Q1.7 Go through the calculation of $a(100)$ and $b(100)$ step by step, using a calculator, and at each stage truncate the numbers to 4 decimal digits (that is retain a total of only four digits, not four places of decimals). Which of `a(x)` and `b(x)` is the more accurate? Explain the reasoning behind your choice. **[3]**

1.12 Computer languages, programs

[This subsection borrows freely from the book *Think Python: an introduction to software design* by Allen Downey. This is an early version of Downey's book *Python for software design*. The early version is freely available online at www.greenteapress.com/thinkpython/thinkpython.html — for which many thanks!]

This is a good place to emphasise the difference between using a computer language, in this case Python, and a natural language like English or Chinese. Natural languages are often ambiguous, with meaning enforced by context or extra information. Computer languages are examples of *formal* languages intended for specific applications, in this case describing computations.

Computer languages are designed to be unambiguous (every statement has a unique meaning), concise and literal (a statement means exactly what it says). Understanding a few lines of code, especially in a high-level language like Python, can take much longer than understanding a few sentences in English.

Formal languages have a precise **syntax** which you must follow (see appendix D for some common syntax errors in Python). The Python interpreter will stop when it encounters syntax errors, so these are the first **bugs** that you have to remove from any code you write.

When solving a problem you should first decide on the **algorithm** to use. This is like a recipe or set of instructions for the solution. Once you have an algorithm you can translate it into a sequence of statements in your chosen computer language: the sequence of statements is your **program** or **script**. Typically you will want to get some data from the keyboard, a file or simply by defining it with statements in the program. Then you will perform some operations and transformations on the data (in our case usually mathematical), perhaps using tests to determine the appropriate actions. Finally you will want to print out numerical answers, plot graphs or save new output data to a file (perhaps a file of numerical values or image data).

When the Python interpreter executes (processes) your script or program, execution begins at the first statement. The statements are interpreted one by one in order until the end of the program is reached. When a function definition is encountered, the statements in the definition (**body** of the function) are not executed right away, but are stored. Then when the function is used (**called**) later in the script the interpreter executes the body before carrying on with the statements following the function call. One consequence of this is that functions must be `def`ined before they are used.

Section 2

Arrays and Graphs

In scientific research a lot of time is spent analysing data, be it experimental results or the output of complex computer simulations. You will often want to perform checks on the output, to look for obvious mistakes in your data-collection or calculations. Perhaps most important, you will want to look at your data by plotting it. We will use the the **matplotlib.pyplot** module for this.

Before this we will consider dealing with arrays of numbers (for example to represent vectors, matrices, image data and so on) which crop up all the time in scientific computation.

2.1 Lists and arrays

We often need to manipulate sets of numbers all of which are related in some way. A simple example would be a vector given as a list of its components. Mathematically we might label them $y_0, y_1, y_2, \ldots, y_{n-1}$. In Python, you can do this with a list, for example `g = [0,3,5,8,12,2]`, with elements `g[0]=0`, `g[1]=3` and so on. The numbers `[0]`, `[1]`, etc are called **subscripts** by analogy with the mathematical notation.

2.1.1 Numpy for numerical arrays

Numpy[1] is a Python module which interfaces to high performance linear algebra libraries for efficient vector and matrix manipulation. It provides roughly the same functionality as Matlab. **Numpy** provides an "array" data structure: this is the first thing we will take advantage of.

Python lists can contain any Python object, but for scientific use, we often want them to contain numbers only. **Numpy** arrays, which can be mutidimensional, are lists of

[1] http://www.numpy.org/

objects of the same type and take advantage of speedups you can get if you know this. Here is how to create an `array`:

```
>>> from numpy import *
>>> a=array([0,1,3,6,9])
>>> type(a)
<class 'numpy.ndarray'>
```

Note that `array()` takes a list as its argument. Numpy also provides `arange()`, in analogy to `range()`, which gives another way to create arrays. Below we use it to create two arrays. Note that arrays can be added, subtracted, multiplied and divided: the operations are performed elementwise as the following example should make clear. You can also `print` an array.

```
>>> b=arange(5) # array of integers
>>> c=arange(0.1,1.0,0.2) # array of floats
>>> b
array([0, 1, 2, 3, 4])
>>> c
array([ 0.1,  0.3,  0.5,  0.7,  0.9])
>>> b+c
array([ 0.1,  1.3,  2.5,  3.7,  4.9])
>>> b*c
array([ 0. ,  0.3,  1. ,  2.1,  3.6])
>>> print(b*c)
[ 0.   0.3  1.   2.1  3.6]
```

Numpy redefines arithmetic operations and some functions, such as `sin`, so that they act elementwise when applied to an array:

```
>>> a=array([0,1,3,6])
>>> a+a
array([ 0,  2,  6, 12])
>>> a*a
array([ 0,  1,  9, 36])
>>> sin(a)
array([ 0.,  0.84147098,  0.14112001, -0.2794155 ])
```

There are handy functions to create arrays of zeros or ones:

```
>>> zeros(7,int)
array([0, 0, 0, 0, 0, 0, 0])
>>> ones(5,int)
array([1, 1, 1, 1, 1])
>>> ones(5,float)
array([ 1.,  1.,  1.,  1.,  1.])
```

To create a two-dimensional array, you could use, for example:

```
>>> a=zeros((3,5),int)
>>> print(a)
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

Observe that the printed form of a two-dimensional array looks like a matrix.

In the following example we create a 3×5 array, called a, and compare it with a list of 3 length-5 lists, called b. Note how the elements of the array are accessed, and compare this to the list-of-lists case. The last input line of the example shows that you can convert b into an array using array(b).

```
>>> a=zeros((3,5),int)
>>> b=[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0]]
>>> for i in range(3):
        for j in range(5):
                a[i,j]=i+j     # note the
                b[i][j]=i+j    # difference


>>> print(a)
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]]
>>> print(b)
[[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]
>>> print(array(b))
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]]
```

Numpy also has functions for finding the maximum and minimum values in an array, as well as the mean, median, and standard deviation:

```
>>> from numpy import *
>>> a=array([1,4,2,9,8,6,3,7,3,5,10,4])
>>> max(a)
10
>>> min(a)
1
>>> mean(a)
5.166666666666667
>>> median(a)
4.5
>>> std(a)
2.733536577809454
```

2.2 Drawing graphs: plotting with matplotlib

The preferred plotting library for scientific users of python is **matplotlib**[2]. It is most frequently used by importing the **matplotlib.pyplot** module, which provides simple commands for rapid viewing of plots, but can also be used to make publication-quality plots if needed. We will stick to simple plots in this course.

We provide two scripts (on Blackboard) to show you how to use pyplot to make 2D plots. You can load these into the Spyder editor window and then press play or press F5 to execute them. The first script, called **pyplot-show.py**, illustrates the pyplot `plot()` command. Given two lists or arrays of x- and y-coordinates, called, say, `x` and `y` respectively, the command `plot(x,y)` makes a 2-D plot. If you use a single list, say `plot(y)`, this is taken as the list of y-values and the x-values default to integers starting from 0. By using several `plot` commands, you can put several curves on the plot. The script also shows pyplot commands for axis-labelling and titling of the plot. Note that the `show()` command at the end of the script is what causes the plot to be drawn and displayed. Spyder may show you the plot even without `show()`, but not all python editors will.

```
import matplotlib.pyplot as plt
from numpy import *

x=arange(0,8*pi,0.05*pi)
plt.plot(x,cos(x))
plt.plot(x,cos(x)*exp(-x/20.0))
plt.xlabel('x')
plt.title('Oscillation and damped oscillation')
plt.show()
```

You can use the 'save' button in the plot window to open a standard file-saving dialogue. If you want to include your plot in a word-processor document, I recommend saving it as a **png** file (denoting portable network graphics), using file-extension **.png**. As an alternative, **pyplot-save.py** generates the same plot, this time saving it directly:

```
import matplotlib.pyplot as plt
from numpy import *

x=arange(0,8*pi,0.05*pi)
plt.plot(x,cos(x))
plt.plot(x,cos(x)*exp(-x/20.0))
plt.xlabel('x')
plt.title('Oscillation and damped oscillation')
plt.savefig('pyplotplot.png')

import os
curdir=os.getcwd()
print('Plot saved in:', curdir)
```

[2] http://matplotlib.org/

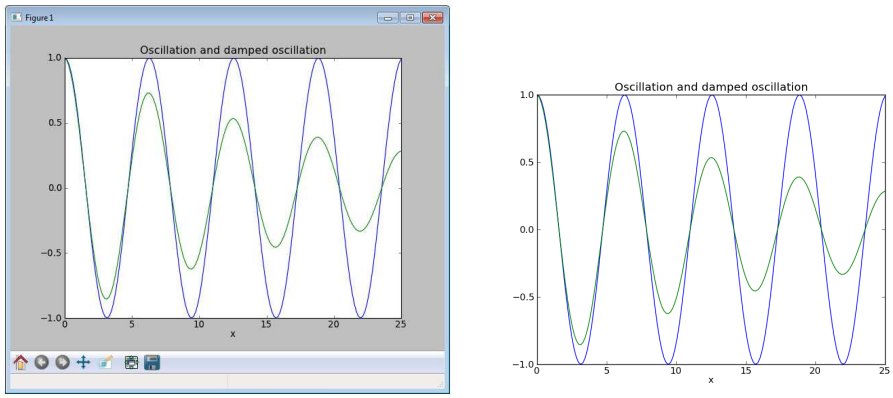


Figure 2.1 Pyplot plot window and corresponding saved plot.

At the end of this script, I show how to use the standard Python **os** module ('os' denotes operating system) to find out the 'current working directory' (cwd) where the plot is saved. Figure 2.1 shows the pyplot plot window and the corresponding saved plot.

Note that you should NOT use the `show()` or `ion()` commands in any code submitted to the Handin system.

2.2.1 Practice plotting

For practice, try making your own plot showing the curves $\sin(x)$ and $\cos(x/2)$ for $0 \leq x \leq 5\pi$. Recall that the **math** module (and **numpy**) defines `pi` as an alias for the constant 3.1415926535897931.

As an example combining series and plotting, look at the function $g(x)$ defined by an *infinite* series whose first few terms are:

$$g(x) = \frac{\cos x}{1^2} + \frac{\cos 3x}{3^2} + \frac{\cos 5x}{5^2} + \frac{\cos 7x}{7^2} + \cdots$$

If you wanted to evaluate this by hand you would successively calculate terms in the series, accumulating their sum as you went. Eventually the terms would become so small that the sum would stop changing (within your desired precision) and that would be your result. Below is a function which gets the computer to follow these steps to evaluate $g(x)$ from the series definition.

```
from math import cos, fabs, pi
def g(x):
    n=1; total=term=cos(x)  # First term
    while fabs(term)>(1.0e-7*fabs(total) + 1.0e-13):
        n+=2  # Advance to next term
        term=cos(n*x)/(n*n)
        total+=term  # Add term to total
    return total
```

Study the code and the following comments, noting how the code relates to the series and our description of evaluating the sum of terms.

- Two statements have been written on one line, each separated by a semicolon. This is sensible where the statements are short and closely related, as in this case, where they set the conditions at the start of the calculation. Use this if you think it makes the code more comprehensible.
- The initial values of `total` and `term` have been set simultaneously (this is called multiple assignment).
- Inside the loop the shorthand `+=` has been used: `n+=2` is the same as `n=n+2`. This construction also works with `-`, `*` and `/`.
- The notation `1.0e-7` denotes the number 1.0×10^{-7}.
- The `fabs(x)` function returns the absolute value (or magnitude or modulus) of a floating point number `x` (the `f` in `fabs` indicates that `x` has to have type 'float'). There is also a built-in `abs()` function which can give the absolute value of any number type.
- It would be tempting to use a variable name `sum` instead of `total`, but `sum()` is a built-in function in Python.
- The `while` condition makes the programme stop adding terms once they get very small. Since the computed function can be zero (or tiny compared to its maximum magnitude), we control both the relative (`1.0e-7*fabs(total)`) and absolute (`1.0e-13`) precision in our stopping condition.

Q2.1 What would go wrong if `fabs` were omitted? **[2]**

Q2.2 Why do you think the number 1.0e−7 is chosen for the stopping condition, at least in comparison to a much larger or much smaller number, say 1e−3, or 1e−26? **[2]**

Q2.3 Create your own script and copy the `g(x)` function into it. Your script should then make suitable arrays of values of x and $g(x)$ and plot a graph of $g(x)$ for $0 \leq x \leq 4\pi$. Make sure your script saves the plot to a png file (do not use `show()`). **[7]**

Series like this one are called Fourier series and are very important in physics because it turns out that any periodic function can be represented in this form. Try inventing your own similar one, for example take alternating signs, even numbered terms, even and odd numbered terms, different reciprocal powers, and so on.

2.2.2 A note on variables

We have already noted that a variable, say `x`, defined and used inside a function definition is distinct from `x` used elsewhere. This also applies to the other variables `n`, `total` and `term` used in `g()`. This means that you can use a variable `n` elsewhere without its value being altered every time you calculate `g(x)`.

The variable `n` used in the definition of `g(x)` is said to be **local** to the function. You can use a function definition like `g(x)` in combination with other function definitions or pieces of Python code and there will not be any side effects from using the same variable names in both. The opposite of **local** is **global** which means that a name like `n` refers to the same storage location in the computer memory wherever it appears. This can cause problems if one piece of code unexpectedly changes a global variable that is used elsewhere, so it's a good idea to avoid global variables.

The rules on global and local variables vary from language to language.

2.3 A real physics problem

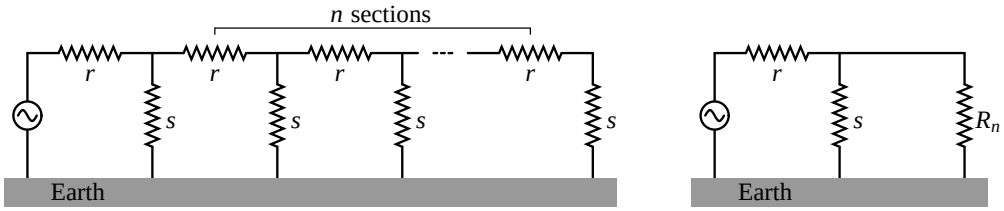


Figure 2.2 Electricity transmission cable. On the left is a cable comprising $n+1$ sections in total. On the right this is viewed as n sections with resistance R_n in parallel with resistance s, both in series with resitance r.

A long overhead electricity transmission cable is supported on poles at regular intervals. It is a single wire; the Earth is used as the return, and the resistance of the Earth can be neglected. The poles are not perfect insulators but each has a resistance s ohms to Earth. The wire between each pair of poles has a resistance r ohms and the generator

is connected to the first pole by a cable also of resistance r ohms. There is no load at the end remote from the generator. Let the resistance seen from the generator of such a cable consisting of n poles be R_n ohms. By regarding a cable of $n+1$ sections as n sections in parallel with a resistance s ohms, both in series with a resistance r ohms, as shown in figure 2.2, you can see that

$$R_{n+1} = r + \frac{sR_n}{s+R_n}$$

This formula enables us to calculate R_{n+1} given R_n. Since R_1 is clearly $r+s$ we can get all the values. To gain some understanding of how R_n varies with n we should plot a graph. Here is a Python script to do that, where we set the two resistances to be $r = 100$ and $s = 10000$:

```
r=100.0
s=10000.0
n=50
a=[r+s]
for i in range(n-1):
    a.append(r+s*a[-1]/(s+a[-1]))
import matplotlib.pyplot as plt
plt.plot(a)
plt.show()
```

Note the use of symbols `r` and `s` for the resistances. If we saved this code, perhaps to reuse it with different resistances, it would be easy to change the values in just one place. Moreover, the expression in the code looks as similar as possible to the expression in the theory; this reduces the chance of making an error when typing. Note also how the list of values is built up in the list `a`. First we make `a` contain a single value, `r+s` (so that `a[0]=r+s`). Then we successively append new elements to `a`. In calculating the new element, we always use the value of the last element found so far, which is `a[-1]` (remember this is Python's way of letting you access the last element of a list without you having to know how long the list is).

A final point is that we might wish to change the number of sections of cable we consider. We made this easy by setting `n=50`. To change the number of sections we only have to change `n` in one place. For a large program with many occurrences of a particular number, this is much easier and less error-prone than trying to edit the number in many places.

2.3.1 Using the program

You will notice that the resistance R_n tends to a constant for a very long line. Call this limiting value $R_\infty = \lim_{n\to\infty} R_n$.

Q2.4 Write a program to compute R_∞ and print the result. It may be helpful to consider how we used a `while` loop in the `g(x)` function earlier. **[5]**

Now suppose we were interested in how close the resistance got to R_∞ when the line was long but not infinitely long. That is, consider $R_n - R_\infty$ for large n. Write a program to plot a graph of $\log(R_n - R_\infty)$ against n.

Q2.5 From your graph explain why when n is large R_n is given approximately by:

$$R_n = R_\infty + Ae^{-Bn}$$

[4]

2.4 String formatting

While not technically related to plotting, being able to format strings (text) correctly can be useful when creating axis labels, and so is briefly introduced here. In Python, variables can be formatted as strings using the `"{}".format(x)` method. Some examples are provided below, and you can find further information and examples at http://pyformat.info.

```
>>> a="a string"
>>> b=10 # An integer
>>> c=5.6 # A float
>>>
>>> # Pad string with spaces to 10 characters,
>>> # and append an integer:
>>> print("{:10s} {:d}".format(a,b))
a string   10
>>> # Pad integer with zeros to 6 characters:
>>> print("{:s} {:06d}".format(a,b))
a string 000010
>>> # Show float with 3 decimal places:
>>> print("{:.3f}".format(c))
5.600
>>> # Show float with 2 decimal places, in a 10-character
>>> # long string:
>>> print("{:10.2f}".format(c))
      5.60
>>> # Pad float with zeros to a total width of 5 characters:
>>> print("{:05.1f}".format(c))
005.6
>>> # Align a string to the right:
>>> print("{:>20s}".format(a))
            a string
>>> # Include other constant (i.e. not a variable) text:
>>> print("Current = {:.3f}A, Voltage = {:+d}V".format(c,b))
Current = 5.600A, Voltage = +10V
>>> # Use format as part of a string assignment operation:
>>> x="You have {:d} apples.".format(b)
>>> print(x)
You have 10 apples.
>>> # Always include a sign (+ or -):
>>> print("{:+.3f}".format(c*-10))
-56.000
```

If you have trouble understanding what the above examples are showing ask a demonstrator for help.

Section 3

Data Processing

One of the most important uses of computers is for automatic processing of experimental data to test hypotheses or determine physical constants. A proper introduction to this subject requires the systematic treatment of experimental errors. We won't do that here but will look at some aspects of fitting a theoretical model to data.

3.1 Fitting a theoretical curve to data points

The data we discuss concerns the lifetime of an unstable fundamental particle called the delta resonance of the proton. The experiment consists of firing a beam of pions at the protons in a liquid hydrogen target, varying the energy E of the incident pions and counting the number scattered into a detector. Theory shows that this number $n(E)$ should be given by the formula

$$n(E) = \frac{1.467 \times 10^7}{w^2/4 + (E - 1232)^2}$$

where E and w are in units of MeV. The lifetime is given by h/wc^2, where h is Planck's constant and c is the speed of light. We count $n(E)$ for various values of E and the problem is to extract a value for w.

The data have been stored in files, **data1.txt**, **data2.txt** and **data3.txt**. Choose any one of these to analyse (note that **data1.txt** has the most "well-behaved" data, but that doesn't mean you shouldn't also look at the others). Each file consists of a number of lines, with each line containing three numbers: the energy E, the corresponding number of scattered pions $n(E)$ and the error in this number, $\delta n(E)$. The energy values are evenly-spaced. Some corrections have been made for detector efficiency so the values of $n(E)$ and $\delta n(E)$ are not whole numbers.

You should view the contents of one or more of these files to see what the numbers look like. The code below shows one way to put the data into lists using Python's file-handling operations. Once we have the lists we convert them into arrays.

```
from numpy import array
f=open('data1.txt','r')
ea=[]; na=[]; dn=[]
for line in f:
    estr,nstr,errstr=line.split()
    ea.append(float(estr))
    na.append(float(nstr))
    dn.append(float(errstr))
f.close()
ea=array(ea)
na=array(na)
dn=array(dn)
```

In computer languages, files have to be opened for reading and writing. Here, we create a file object `f`, opening the file **data1.txt** for reading (`'r'`). We create empty lists for the energies and counts. Then we loop through the lines of the file. Each line is read as a string, which, for our data files, contains number strings separated by white space. Using `line.split()` we pull out the number strings. The line `ea.append(float(estr))` converts the string for the energy into a floating-point number and appends it to the list of energies. Once we have read all the lines we `close` the file.

Note that for the above code to work the **data1.txt** file must be in Python's current working directory, or the same place as the Python script doing the file reading (if on a University computer, then probably somewhere in your home filestore, `H:\`.)

Q3.1 Produce a graph of the values of $n(E)$ against E, including the errorbars, using the pyplot command `errorbar(ea,na,dn)`, which will put an errorbar at each data point and join the points with straight line segments. Label the axes appropriately. Submit your graph to Handin (not your code) along with your answers to this section. **[4]**

Q3.2 Summarise the general features of the data values. Imagine you are trying to convey the shape of the graph to a colleague over the telephone. **[2]**

Now create another array containing the values for $n(E)$ given by the *theoretical formula*, by defining a function `theory(e, w)` to calculate $n(E)$ using the formula given at the start of this section. If this new array is called `th`, say, then the element `th[i]` should be the theoretical count corresponding to energy `ea[i]`. You will need a value for w when evaluating the formula. To get this, substitute a data value from near the middle of the dataset into the formula and solve for w. Superpose a graph of the theoretical curve over the experimental one.

By adjusting the value of w try to improve the fit between the two curves. Find *very* roughly the value of w that produces a curve similar to the experimental one.

Q3.3 Describe qualitatively the effect of changing w on the theoretical curve and record the value of w in your electronic lab book. **[2]**

3.2 A quantitative method of fitting

As a rough principle for data processing, the best value to take for a physical quantity is the value that produces the smallest discrepancy between the observed readings and the numbers predicted by a theoretical analysis of the experiment. The word discrepancy is deliberately vague; without more knowledge of the sources of experimental error we cannot be more precise. However a commonly used procedure is as follows.

Consider the **residuals**, r_i, that is the differences between the observed values $n(E_i)$ and the theoretical predictions given in the equation above:

$$r_i = n(E_i) - \frac{1.467 \times 10^7}{w^2/4 + (E_i - 1232)^2}$$

Next, square these residuals, giving positive numbers. If we add together all the squared residuals then the resulting number is a convenient measure of the discrepancy between the data and the theory. However, this takes no account of the errors in the data. We expect that a data point with a larger error should be less influential, so we divide each squared residual by the corresponding squared error before summing. Hence our measure of the discrepancy is

$$\sum_i \left(\frac{r_i}{\delta n(E_i)} \right)^2$$

We now want to see how this discrepancy varies as w changes. The obvious thing to do is to define a function `discrepancy(w)` which calculates the sum of the squares of the residuals for any value of w. You then want to choose w to minimise this function.

There is one issue to think about. If you are working interactively in the Python shell and have already defined the arrays they will be available to any function definitions you enter in the shell. Alternatively, if you are building up a script (or module) and create the arrays `ea` and `na` *outside* (and before) all function definitions in the script, they will be **globally** available in any functions you write in the script. In either case, this means you could write a function `discrepancy(w)` and simply refer to `ea` and `na` within the function. If the Python interpreter does not find definitions of objects inside a function, it will look for global definitions.

In general, using global objects is frowned upon, because it's easy to make mistakes where one piece of code changes a global object that some other piece of code assumes *not* to have changed. Here, our code is simple enough and short enough that it should not be a problem.

Q3.4 Write a function called `discrepancy`, to calculate the sum described above. Save this function in a **.py** module. **[3]**

Q3.5 Use your function and a manual trial-and-error method to find an approximate value for the w which minimises the discrepancy. **[1]**

3.3 Finding a minimum automatically

It makes sense to automate the procedure of finding the minimum. The minimisation algorithm, known as the Golden Section search, coded in the function `gmin(f,a,c)` in **minimise.py** (available on Blackboard) is very suitable for this particular problem. It finds the minimum of a function $f(x)$ which is known to lie between $x = a$ and $x = c$ using an intelligent search. You do not need to understand in detail how it works. When you call this function it returns a pair of values, the position $x = b$ of the minimum and the value $f(b)$ of the function at the minimum. You can import this function in the usual way:

```
from minimise import gmin
```

Q3.6 Use the `gmin` function to find the minimum of the function $\exp(x) + 1/x$. **[2]**

Q3.7 Try minimising the function `discrepancy` to find an accurate value for w, and record this in your electronic lab book. Make a plot showing the fitted data and the best-fit theoretical curve, and submit your plot to Handin. **[4]**

This value can be regarded as the "best" value obtainable in the experiment. Note that just because one value of w gives the lowest discrepancy does not mean that we should reject all nearby values. We need some measure of how big an increase above the minimum we should tolerate before we decide that the value of w is unacceptable. In other words we want a value for w and a number indicating the uncertainty in this value. Getting this requires more detailed consideration of the experimental errors and a knowledge of statistics.

3.4 Fitting with SciPy

SciPy is a Python library giving you easy access to high level mathematical algorithms and convenience functions[1]. It contains some useful functions to do fitting for you, which can also provide error estimates on the fitted parameters. One of the most useful ones is `curve_fit`, which you can import from `scipy.optimization`. This fits a function (which you can define) to a set of data points, by minimising the sum of the squares of the residuals between the model (theoretical function) and the data, as

[1] E Jones, T Oliphant, P Peterson et al, *SciPy: Open source scientific tools for Python* (2001–), http://www.scipy.org/.

we did using the `discrepancy()` and `gmin()` functions. It also returns a "covariance matrix", which can be used to estimate the uncertainty on the fitted parameters (see PHYS1201!)

The function `curve_fit` takes 5 input parameters:
1. The name of the theoretical function to fit to the data.
2. An array of x data values.
3. An array of y (measured) data values.
4. Initial estimates for the fit parameters (optional - can put 'None').
5. Uncertainties on the y data values (optional - can put 'None').

This example shows how `curve_fit` can be used to fit a straight line to some data:

```
>>> from numpy import array, arange
>>> from scipy.optimize import curve_fit
>>>
>>> def straightline(x, m, c):
        y = (m * x) + c
        return y

>>> # x-values, 1-9
>>> x = arange(1.0,10.0)
>>> # measured y-values
>>> y = array([ 0.1, 0.4,0.2, 0.3, 0.6,0.5, 0.8, 0.9, 0.8])
>>>
>>> # error on y-values
>>> dy = array([0.05,0.2,0.09,0.12,0.1,0.09,0.14,0.18,0.19])
>>>
>>> popt, pcov = curve_fit(straightline, x, y, None, dy)
>>>
>>> # popt contains the optimal values for the parameters
>>> print(popt)
[ 0.09898443 -0.00838688]
>>>
>>> # pcov contains the covariance matrix
>>> print(pcov)
[[ 0.00015424 -0.00051777]
 [-0.00051777  0.00265769]]
```

This tells us that the best-fit parameters for our `straightline` function are $m = 0.09898443$ and $c = -0.00838688$. The variances on these are in the diagonal of the covariance matrix: 0.00015424 for m and 0.00265769 for c. The one standard deviation error on the parameters is the square root of these variances (although see PHYS1201 for caveats, etc!)

```
>>> from numpy import sqrt, diag
>>> print(pcov)
[[ 0.00015424 -0.00051777]
 [-0.00051777  0.00265769]]
>>> print(sqrt(diag(pcov)))
[ 0.01241928  0.05155281]
```

This tells us that $m = 0.09898443 \pm 0.01241928$ and $c = -0.00838688 \pm 0.05155281$.

Q3.8 Use `curve_fit` to fit your `theory` function to the **data1.txt** data used earlier. What value of w do you obtain using this method, and what is it's uncertainty? Make some notes about how you used `curve_fit` in your electronic lab book. **[2]**

3.5 Using what you know

Now you have acquired a powerful set of programming skills and python knowledge. You can read data into a program, plot it, and fit a theoretical function to it. You can also code mathematical expressions to calculate derived parameters from measured data. Writing your own programs gives you a lot more flexibility than if you were to perform the same tasks using commerical spreadsheet software. The remaining sections of this computing module will help you to develop your skills further, and will teach you techniques and algorithms for implementing other mathematical methods in your data analysis. However, I strongly encourage you to start using your programming skills as part of your experimental labs already, as well as in your future research projects. By far the best way to improve your skills is to use them!

Section 4

Numerical Integration

Physics calculations often require the evaluation of an integral. Some integrals can be done analytically but often the physicist has to resort to numerical integration. In this section we'll look at how you can get a computer to perform numerical integrals for you, by developing our own simple integration method. At the end of the section we'll show how you can use a more robust integrator imported from the **scipy** module.

4.1 The trapezium rule

A simple formula you have probably met is the *trapezium rule*, which calculates the area under a curve by approximating it as a series of trapeziums, and is illustrated in figure 4.1. The formula is:

$$\begin{aligned}\int_a^b f(x)\,dx &= h\left[\frac{1}{2}f(a)+f(a+h)+f(a+2h)+\cdots+f(b-h)+\frac{1}{2}f(b)\right] \\ &= \frac{h}{2}\left[f(a)+f(b)\right]+h\sum_{i=1}^{n-1}f(a+ih)\end{aligned}$$

This comes from summing the areas of the trapeziums underneath the curve; the area of the first one is the width h times the average height $[f(a)+f(a+h)]/2$. If you have not met this get a demonstrator to explain it in more detail. Although the trapezium rule is simple it forms the basis for more sophisticated methods, such as Simpson's rule, for example.

We will begin by writing a program to evaluate the following integral that is difficult to do analytically:

$$\int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,dx \tag{4.1}$$

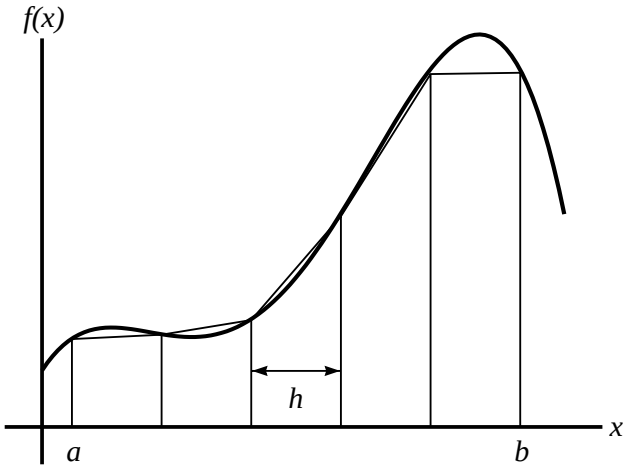


Figure 4.1 Trapezium rule for integration.

Some things to think about before writing any code are:

- How many trapeziums should be used? Too small a value will give the wrong answer and too large a value will waste time. Some experimentation is clearly required, so we should write the code for n trapeziums as in the formula and input the value of n from the keyboard.
- We can compute h from n and the integration limits.
- We start with the term $[f(a)+f(b)]/2$ and successively add the remaining terms to it using a loop.

To get the number of trapeziums from the keyboard we use `input()` to read a string and then convert the string to an integer:

```
n=int(input("Enter number of trapeziums: "))
```

Do NOT include `input` in any code you submit to Handin - the system will run your code, but there is nobody there to type anything for you! Before reading on, think how you would translate the description above into code. Then consider the following:

```
while True:
    n=int(input("Enter number of trapeziums: "))
    h=1.0/n
    intgrl=0.0
    for i in range(1,n):
        intgrl=intgrl+h*(i*h)**4*(1-i*h)**4/(1+i*h*i*h)
    print(n, intgrl)
```

There are several things to remark here.

- First, recall that Python lets you use `**` to denote exponentiation.
- Since `True` is *always* true, the while loop will always repeat. Use Ctrl-c to interrupt it when you have finished trying different values for n.

More fundamental is that the program can be considerably improved with some thought. It has been written with the integration limits 0 and 1 built into the code, and we might want to change them. It is also written to integrate one particular function, but once you have code for integration you may want to keep it and use it again for other problems. An improvement would be to define a function to do the integration:

```
def trap0(f,a,b,n):
    """Basic trapezium rule. Integrate f(x) over the
    interval from a to b using n strips."""
    h=float(b-a)/n
    s=0.5*(f(a)+f(b))
    for i in range(1,n):
        s=s+f(a+i*h)
    return s*h
```

Here the function to be integrated, `f`, and the integration limits `a` and `b` are given as arguments to the function `trap0` so they are easy to change. Likewise the number of trapeziums is given as the argument `n`. To use this, you will need to give a definition of the function you wish to integrate, either by writing your own or by importing a suitable module which defines the function (for example the sin function from the **math** module).

Q4.1 Write a Python function for the integra**nd** (the thing being integrated) in equation (4.1). **[2]**

Use `trap0` to estimate the integral, increasing the number of trapeziums until the estimate hardly changes.

Q4.2 What is the value of the integral in equation (4.1) at the beginning of this section? **[3]**

Q4.3 How many trapeziums are needed to get a fractional (relative) error of less than 1 part in 10^6 (this is 6 significant figures)? **[3]**

4.2 Getting a specified accuracy

There is still an unsatisfactory feature about our integration routine. It is inconvenient to investigate the number of trapeziums to use for every integral; computers can do that just as well. It is the accuracy of the answer that is much more relevant.

We can never be absolutely certain of the right answer in a numerical integration, but if we double the number of trapeziums and still get the same answer then that answer is probably correct. So we will be satisfied if

$$|I(n) - I(2n)| < \delta \times |I(2n)|$$

where $I(n)$ is the answer for n trapeziums and δ is a small number specified by you.

Here is an integration function which can increase the number of trapeziums:

```
def trap1(f,a,b,delta,maxtraps=512):
  """Improved trapezium rule. Integrate f(x) over
  interval from a to b, trying to get relative accuracy
  delta. Optional last argument is maximum allowed
  number of trapezia."""
  n=8
  inew=trap0(f,a,b,n)
  iold=-inew # make sure we do not terminate immediately
  while (fabs(inew-iold)>delta*fabs(inew)):
    iold=inew
    n=2*n
    if n>maxtraps:
      print("Cannot reach requested accuracy with", \
            maxtraps, "trapezia")
      return
    inew=trap0(f,a,b,n)
  return inew
```

The backslash (\) at the end of the `print` line marks a continuation, telling the interpreter that the `print` statement continues on the next line.

Note how the new function uses `trap0`. The new function expects the function to be integrated, the integration limits and the desired accuracy to be given as arguments. A new feature is that the last argument is given in the form `maxtraps=512`. In Python, this makes the argument *optional*. You can use `trap1(f,0,1,1.0e-7)` for example. In this case `maxtraps` takes its default value of 512. If 512 trapeziums are not enough to get your desired accuracy, you can raise the number to 1024 by using `trap1(f,0,1,1.0e-7,1024)` or `trap1(f,0,1,1.0e-7,maxtraps=1024)`.

In the code above, every time you double the number of trapeziums you have already evaluated the integrand at half the points you need. A further improvement (which you are not asked to make) would be to modify the code so that you do not evaluate the integrand more than once at the same point.

4.3 Code development

Consider how we arrived at the integration function `trap1`. We first developed a piece of code which worked, then thought about it some more to improve it. Once we had something useful we encapsulated it in functions which we could save for future use. This is a typical way to develop programs.

The Python shell (interactive interpreter) can be very helpful here. First you can try a line or two of code in the shell. Once you have something useful you can begin to construct a script or function definition in the editor window. This allows you to build up a script or complicated function in stages, checking as you go along that everything works.

4.4 Another integral

As a further example consider the integral:

$$\int_{-\infty}^{\infty} e^{-x^2}\,dx$$

One cannot use the trapezium rule without modification because the range of integration is infinite. However if you plot the integrand you will see that it rapidly becomes negligible if x is large, so you can cut off the range at some finite points. Think what happens when x is negative.

Q4.4 Try a few experiments over different ranges and try to get an accurate value for this important integral. Make some notes about what you have done in your (electronic) lab book. **[3]**

4.5 A real physics problem — the simple pendulum

The period of swing of a pendulum of length l in which the string reaches an angle α to the vertical is given by

$$T = T_0 \frac{\sqrt{2}}{\pi} \int_0^{\alpha} \frac{d\theta}{(\cos\theta - \cos\alpha)^{1/2}}$$

where $T_0 = 2\pi\sqrt{l/g}$ is the period for small amplitudes. The easiest way to derive this is by conservation of energy; ask a demonstrator if you cannot do it. We will see that the period increases with the amplitude α (note that all trigonometric functions in computer languages expect arguments in radians).

If you try to use your integration program to evaluate T/T_0 for a pendulum with amplitude $\alpha = 45°$, you will find that it fails! This is a typical discovery; your new program, which worked for the test problem, fails when used in the first real problem. The golden rule when doing numerical integration is to *plot the integrand*; remember you are calculating the area under this curve.

Q4.5 Plot the integrand for the integral above and submit your graph to Handin. **[4]**

Q4.6 You will see that any numerical evaluation of this integral is going to fail: explain why. **[1]**

The trick is to change the variable. If we write $\sin(\theta/2) = \sin(\alpha/2)\sin\phi$ we can get

$$T = \frac{2T_0}{\pi}\int_0^{\pi/2} \frac{d\phi}{\left(1-\sin^2(\alpha/2)\sin^2\phi\right)^{1/2}}$$

Plot this new integrand and note that it is well behaved.

Q4.7 Write a script to tabulate the ratio T/T_0 for a range of amplitudes. Be sure to `def`ine all functions which you use inside your script, including `trap1` and `trap0`. **[3]**

Q4.8 Obtain as accurate a value as you can of T/T_0 for an amplitude of 90°. **[1]**

4.6 Integration with SciPy

SciPy is a Python library giving you easy access to high level mathematical algorithms and convenience functions[1]. Amongst the things it provides is numerical integration, using routines much more sophisticated than the simple trapezium rule we've investigated.

Here we'll use **SciPy**'s integration routine to evaluate

$$\int_0^2 x^4 \log\left(x+\sqrt{x^2+1}\right)dx$$

Our simple routine has to work hard:

```
>>> from math import log, sqrt
>>> def f(x):
        return x**4*log(x+sqrt(x*x+1))

>>> trap1(f,0,2,1.0e-7)
Cannot reach requested accuracy with 512 trapezia
>>> trap1(f,0,2,1.0e-7,maxtraps=8192)
8.1533643848121962
```

[1] E Jones, T Oliphant, P Peterson et al, *SciPy: Open source scientific tools for Python* (2001–), http://www.scipy.org/.

Using **SciPy** we get both the integral and an error estimate:

```
>>> from scipy.integrate import quad
>>> quad(f,0,2)
(8.153364119811167, 9.0520525739669813e-014)
```

The "quad" is short for "quadrature", an older name for integration.

A plot of the integrand shows that there is a substantial region of integration at small x where the integrand hardly varies relative to its value near $x = 2$. Our simple trapezium rule, with equally-spaced function evaluations wastes a lot of resources on this region. The `integrate.quad()` routine is **adaptive** and so can adjust the function evaluations to concentrate on the more important region.

4.7 Some notes on programming style

The purpose of a good style is to decrease the chance of making errors, particularly errors that cause the program to give plausible but wrong answers. Here are some things to think about.

- Use spaces and blank lines freely to make the structure of your program clearer.
- Do not write `A/B/C` or `A/B*C`, they are ambiguous to humans[2].
- Use meaningful names for variables and functions. Where you cannot make your meaning clear by choice of names put explanatory remarks in comments.
- If you are coding a page of algebra, use the same symbols in the program as on the written page. Do not do a mental translation: you will make mistakes.
- Add clear and helpful comments to your programs, so that it is easy for somebody else to understand your code, or for you to understand it some years later.
- Many programs by physicists are written to solve one problem, are used over a period of a few days by the person who wrote them and are then finished with. There is no point in writing elaborate user interfaces for such programs or including extensive annotation. However physicists often re-use parts of programs, a routine for integration, for example: these should be annotated and carefully documented.

[2]Computer languages specify the *precedence* of mathematical operations, so the expressions given should not cause the interpreter or compiler to complain. The danger is that you may not remember the precedence rules correctly.

Section 5

Random Numbers

Many physics experiments are not exactly reproducible and physicists have to learn to live with experimental errors. In this section we study the counting and display of random data. For convenience the generation of the data is not done by a piece of apparatus but is simulated in the computer. Computer-generated random numbers are also used extensively in simulations.

5.1 Random numbers on computers

Python (like all languages) incorporates a "random" number generator. Of course such numbers cannot be truly random, indeed the modern definition of a sequence of random numbers is one that could not be produced by a shorter computer program than one that simply says "type the following numbers". A "pseudo-random" sequence is usually obtained by multiplying the last number by a constant and using some of the middle digits of the product as the new random number.

Python's built-in random number generator generates real-valued random numbers, x, uniformly distributed in the range $0 \leq x < 1$. However, we will want to use some other random distributions as well so we will make use of the random number module from numpy: `from numpy import random`.

To begin with we'll look at random integers. After using the import statement above, the function `random.randint(n)` produces a random integer between 0 and $n-1$ with all numbers equally likely. Try printing a few values.

5.2 Binning data

Let's check that each value of `random.randint(10)` is equally likely. To do this we generate a large number of random numbers and count how many are equal to 0, how many are equal to 1 and so on. Think how you might do this by hand; one way would

be to create columns labelled 0 to 9 and go through the list of numbers putting a tick in the appropriate column. At the end you count the ticks in each column.

An elegant method to do this on the computer is to accumulate the counts in an array. If the array is called `m` then the value of `m[3]` would tell you how many times 3 occurred and so on. The elements of the `m` array are called **bins** in physicists' jargon. The algorithm described here is like tossing the numbers into labelled boxes or bins and then counting how many there are in each bin.

Here is a code to implement the binning method, for 1000 random numbers.

```
from numpy import zeros, random
m=zeros(10,int)
for i in range(1000):
    n=random.randint(10)
    m[n]=m[n]+1
print(m)
```

While you are *testing* a piece of code like this you may want to keep the maximum count small, say 100, to save time, but set it to say 10000 to get accurate results.

Q5.1 Write a brief description of how the binning code works and what is printed. **[2]**

Q5.2 Why are so many numbers needed? **[1]**

5.2.1 Displaying a bar chart

It is easier to look at results of this sort pictorially, so let's draw a **bar chart** of the counts as in figure 5.1. This has bars corresponding to the bins and the height of each bar is proportional to the count in the corresponding bin.

To make a bar chart of the counts in the 10-element array `m` above, we use the pyplot `bar` function:

```
import matplotlib.pyplot as plt
plt.bar(range(10),m,width=1,align='center')
```

The first argument is a list of the positions of the bars and the second argument is the list of counts. We have used the optional `width` argument to set the width of the bars such that they touch, and the align argument to centre the bars (note the spelling `'center'`). We can then use `xticks` to make x-axis labels which are more appropriate for our bar chart than the defaults.

```
plt.xticks(range(10))
```

Q5.3 Try out the binning code above, and create a bar chart of the result. Save your plot as a png file, and submit this **image file** to Handin. **[3]**

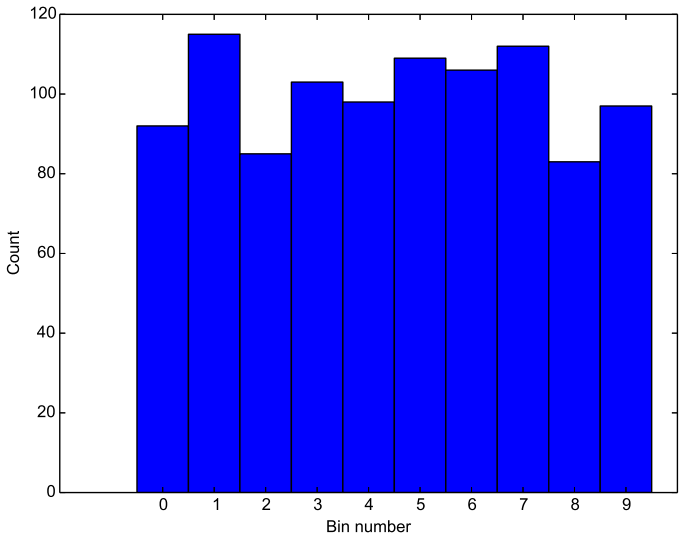


Figure 5.1 A bar chart of 1000 random integers.

5.3 A continuous variable

The numpy function `random.random()` produces a random number x uniformly distributed with $0 \leq x < 1$. Let's check that the numbers do appear uniformly distributed. When examining a continuous random variable like this you have to decide how many bins to take and where to put their boundaries. The bins don't have to have the same width, but it's simpler if they do. In our case, we first decide how many equal-width sections the range 0 to 1 should be split into. For 10 bins, a simple way to assign a number `x` to a bin would be:

```
if 0.0<=x and x<0.1: m[0]=m[0]+1
if 0.1<=x and x<0.2: m[1]=m[1]+1
```

and so on. Note that `and` is used for the **logical and** operator (in this example, Python will also let you write `0.0 <= x < 0.1`). However, it is better to *compute* the bin number as `k=int(10*x)`, remembering that the function `int(x)` gives the integer below `x`. Then increase the count in the appropriate bin: `m[k]=m[k]+1`. Go through a couple of calculations by hand and convince yourself that this gives the same results as using the `if` tests above.

For a graphical display of the frequency distribution of a continous random variable, we use a **histogram**. This has bars whose widths correspond to the intervals defining the bins. Since the upper limit of one bin is the lower limit of the next, the bars touch. The *area* of each bar is proportional to the number of counts in the corresponding bin.

If the bins are all of equal width, as will be the case in this course, then the heights of the bars are proportional to the counts.

The pyplot module contains a function `hist` for plotting histograms with equal-width bins. This function even does the binning for you, but it's not very convenient if you want to make your own choice of bins. Therefore, we'll follow a two-step procedure: first choosing and filling the bins, and subsequently drawing the histogram using the pyplot `bar` function. The left-hand edges of the bins can be specified using `arange` from the numpy module. For 10 equal-width bins in the unit interval with counts stored in a list or array `m`, the appropriate command is:

```
bar(arange(0,1,0.1),m,width=0.1)
```

Q5.4 Write a script to create a histogram showing the distribution of the numbers obtained from `random.random()`. Use the binning method with a sensible number of bins and a sensible number of random numbers. The script should save the histogram as a `png` file. **[3]**

5.4 Averages of random numbers

Next we will examine averages of random numbers. The exercise is to display histograms of the distributions of a single uniform random number, the average of two uniform random numbers, `0.5*(random.random()+random.random())`, and the averages of 3 and 4 such numbers.

Think before you start how to break this problem into its separate parts: generating a particular sort of number, doing the binning, and displaying the result. Use the same horizontal scale and bin sizes for all histograms. The bins will need to cover the range 0 to 1. Use enough random numbers so that you can see the shape of each distribution clearly. We can show several histograms (or any other type of plot) in a grid by using the pyplot `subplot` function. For example, here is a script to make a 2×1 grid of plots, with output shown in figure 5.2:

```
from math import exp
from numpy import arange
import matplotlib.pyplot as plt
x=arange(0,10.1,0.1)
y=[2*xi*exp(-xi) for xi in x]
z=[xi*xi*exp(-xi) for xi in x]
plt.subplot(2,1,1)
plt.plot(x,y)
plt.subplot(2,1,2)
plt.plot(x,z)
plt.show()
```

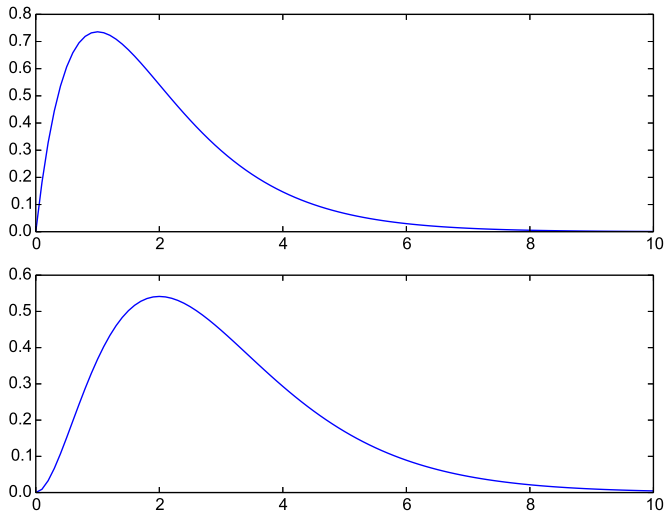


Figure 5.2 Just an example of using `subplot`. You should put axis labels on your plots!

The first two arguments to `subplot` are the numbers of rows and columns in the grid, while the last argument is the subplot number, counting across rows first and then down.

You can optionally use `plt.subplots_adjust()` to adjust the spacing between subplots. This function should be called immediately before `plt.show()`, after the `plt.plot()` commands. It takes up to 6 parameters (left, bottom, right, top, wspace, hspace) which control the amount of white space as a fraction of the total width or height of the whole figure. The following example leaves 20% space (height) between the subplots, with a 30% bottom border:

```
plt.subplot(2,1,1)
plt.plot(x,y)
plt.subplot(2,1,2)
plt.plot(x,z)
plt.subplots_adjust(hspace=0.2, bottom=0.3)
plt.show()
```

Q5.5 Write a script to make histograms showing the distributions of the averages of 1, 2, 3 and 4 uniform random numbers. Choose the sample size and bin width to make the differences between the distributions clear. The script should save the histograms in a single `png` file. **[3]**

Q5.6 Write a short description of what you have discovered about averages of random numbers, and why it happens. **[1]**

5.5 Other types of random number

5.5.1 Gaussian random numbers

A very important type of random number has the so-called Gaussian or normal distribution. These numbers may take any value between *plus and minus infinity* and the probability that one lies in the range between x and $x+dx$ is:

$$\frac{1}{\sqrt{2\pi}} e^{-x^2/2}\, dx$$

The function `random.normal()` in numpy's random module calculates a number of this type. Replace the random number generator in your program by this new one and examine the shapes of histograms of Gaussian numbers and averages of them. When binning these numbers, remember that they can be negative or positive. Remember also that sooner or later you will get a number which lies outside the interval covered by your bins: your code should check for this possibility.

Q5.7 Compute the probability that x lies outside the range $-2 < x < 2$ if it has a Gaussian distribution. Explain how you did this. **[2]**

An important related distribution is that for the quantity $(x^2+y^2+z^2)^{1/2}$ where x, y and z all have Gaussian distributions. This is called the Maxwell distribution and describes the distribution of velocities of molecules in a gas.

5.5.2 Poisson random numbers

Random numbers with a Poisson distribution occur when events happen randomly, but with a constant probability per unit time of occurring. If you count events for a fixed time interval, the number of events follows a Poisson distribution. Examples are:

- Using a Geiger counter to count the number of radioactive decays occurring in a given time interval, say one second (for something with a half-life much longer than one second).
- The number of calls arriving each minute at a busy call centre.

Numpy's random module contains a Poisson random number generator. The function `random.poisson(mean)` produces integer values which represent the number of counts in a fixed time interval. The argument `mean` is the average number expected in that time (which, in the examples above, will vary with the strength of the radioactive source, or the average rate of call arrival).

Q5.8 Write a short script to compute the average of the numbers produced by the poisson generator `random.poisson(mean)` for a mean of 15 and verify that the answer is close to 15. **[2]**

Q5.9 Write a script to draw histograms of the distributions of `random.poisson()` for means of 2, 4, 6 and 8. Do these with the same number of bins so they are directly comparable. You will need about 25 bins. A warning: there is no upper limit to the numbers produced, although very large values are rare. When you bin these numbers, you will eventually find one that is beyond the bin intervals you have chosen, so you should test the numbers and ignore any that are too big. Your script should save the histograms in a single png file (i.e. use `plt.subplot`). **[2]**

Q5.10 Write a brief description of how the histograms change as the mean changes. This corresponds to changes in the strength of the radioactive source, or the rate of call arrivals. **[1]**

Section 6

Differential Equations

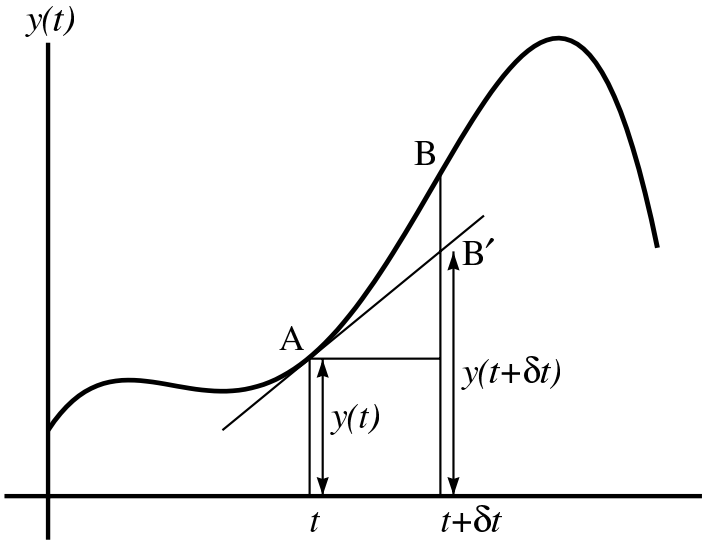


Figure 6.1 Simple numerical integration of a differential equation.

Many physical phenomena are described by differential equations. For example Newton's second law leads to a second order differential equation for most mechanics problems. Many differential equations can be solved analytically but numerical methods are often necessary. Any first order equation (one containing dy/dt but no higher derivatives) can be rearranged into the form:

$$\frac{dy}{dt} = f(y,t)$$

This says that if we are at a point such as A in figure 6.1 on the curve $y(t)$ then we can calculate the slope as $f(y,t)$. We can then get y at $t+\delta t$, that is the point B, by

extrapolating along the tangent, so that:

$$y(t+\delta t) = y(t) + \frac{dy}{dt}\,\delta t \tag{6.1}$$

This involves an approximation in that we obtain the point B′ not B, but the approximation can be reduced by making δt small enough so that B′ is very close to B. We can then repeat the calculation, using the slope at B calculated from the differential equation, to get a third point $y(t+2\delta t)$ etc. So the solution of a differential equation proceeds by stepping in t and successively computing the corresponding y values. We need one value of y to get started and this corresponds to the fact that the solution to a differential equation needs a boundary condition to fix an unknown constant. The method used in equation (6.1) is known as *Euler's method*.

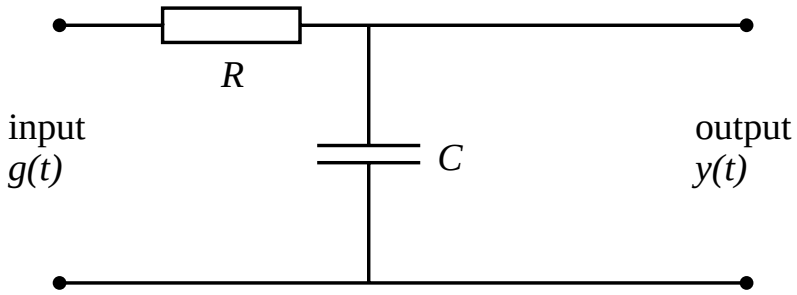


Figure 6.2 RC circuit with input $g(t)$.

A simple example (which could be solved without a computer) describes the RC circuit shown in figure 6.2 with an arbitrary signal $g(t)$ applied to the input. It can be shown that the output voltage $y(t)$ is related to the input $g(t)$ by

$$\frac{dy}{dt} + \frac{y}{RC} = \frac{1}{RC}\,g(t)$$

The product RC has dimensions of time and if we measure t in units of RC, we can take $RC = 1$. Notice that it is usually advisable to arrange that the variables in a computer calculation are *dimensionless*. To get this into the standard form above we write it as

$$\frac{dy}{dt} = -y + g(t)$$

6.1 The computer solution

We now consider how to create a program to solve the differential equation. Before writing any code you should think how to break down the problem into parts.

You should separate the *specification* of the particular differential equation from the *algorithm* used for the solution. The equation is specified by the value of dy/dt; in

the general case we wrote $f(y,t)$ and in our particular case (the electrical circuit) it is $-y+g(t)$. This is a number that depends on y and t and so can conveniently be made a function, `f(y,t)` giving the slope for any values of y and t.

Next consider the algorithm used to solve the equation. The building block of the solution is the calculation of $y(t+\delta t)$ from $y(t)$. This sort of operation is conveniently done by a `function`. The top level of the program is the repeated application of this operation together with incrementing t. The top level also needs to fix the starting value of y and a small enough increment δt.

As a simple input signal suppose $g(t)$ is a constant voltage that is switched on at $t=0$, so that $g(t)=1$ for $t\geq 0$. Since there is no input or output for $t<0$ we need not consider these times and we will take the initial condition $y=0$ at $t=0$, corresponding to the capacitor being uncharged.

The following code illustrates the approach:

```
def f(y,t):
    return -y + 1.0

def odestep(f,y,t,dt):
    return y+dt*f(y,t)

t=0; y=0; dt=0.2
tf=2.0; nsteps=int(tf/dt)
print(t, y)
for i in range(nsteps):
    y=odestep(f,y,t,dt)
    t=(i+1)*dt
    print(t, y)
```

First we defined the function $f(y,t)$ giving dy/dt for our particular problem. Then we defined the function `odestep` which advances the solution by a time step `dt`. Observe that we give the function `f` as one of the arguments to `odestep`. Finally we set the initial conditons $y=0$ at $t=0$, set the step size, δt and then used a loop to perform the integration.

As written, this prints the approximate output of the circuit at time intervals of 0.2 from $t=0$ to 2 for a constant input voltage switched on at $t=0$ with the capacitor uncharged.

We emphasize again that we have separated the problem into three parts:

1. specifying the particular equation in `f(y,t)`
2. specifying the actual algorithm for the solution in `odestep()`
3. performing the integration and printing the results

This is typical for a case where we might want to modify our code to handle many different differential equation problems. Make sure you understand how this works.

Q6.1 Use the code above to determine the value of the voltage when $t = 1$ (that is $t = RC$ in physical units) and $t = 2$. **[2]**

Q6.2 Reduce the value of δt to 0.1, 0.02, and 0.01 and again determine the voltages at $t = 1$ and $t = 2$. **[1]**

6.2 An improved algorithm

The limitation of the above method is the very small value of δt necessary to get reasonable accuracy. This is because the slope is not constant. A considerable improvement would be to use the average of the slopes at A and B in figure 6.1. Unfortunately we cannot compute the slope at B because we don't know where B is, since that is what we are trying to find! However, we use the following improved method:

1. Find the slope at A.
2. Use the slope at A, which we write as $(dy/dt)_{\mathrm{A}}$, to find the coordinates of the point B$'$.
3. Use the values of y and t at B$'$ to compute the slope at B$'$, using the differential equation. Call this $(dy/dt)_{\mathrm{B}'}$. This will not be the correct slope at B but using it will be better than ignoring the change of slope between A and B.
4. Now throw away the value of y corresponding to B$'$ and compute a new *corrected* value for B using the average of the two slopes:

$$y(t+\delta t) = y(t) + \frac{1}{2}\left[\left(\frac{dy}{dt}\right)_{\mathrm{A}} + \left(\frac{dy}{dt}\right)_{\mathrm{B}'}\right]\delta t$$

5. Return $y(t+\delta t)$ in the usual way.

Q6.3 Make this modification to your code, noting that you only need to modify the algorithm for the solution, that is, modify `odestep`. **[4]**

Q6.4 Using your updated program, calculate the values of voltage when $t = 1$ and $t = 2$ with $\delta t = 0.2$. **[1]**

Q6.5 Reduce δt until the voltages at $t = 1$ and $t = 2$ cease to change significantly, and record accurate values of the voltage at the two times. Explain how you did this in your lab book. **[3]**

We will now try a different input to the circuit by altering $g(t)$ to be a cosine wave.

Q6.6 Modify your code to use $g(t) = \cos(t)$ and integrate ("run") over a total time corresponding to several cycles of the input voltage (i.e. set `tf` to be large enough). Plot curves of the output and input voltages using $g(t) = \cos(\omega t)$ as input, for $\omega = 0.5$, $\omega = 1$ and $\omega = 2$ (you will need to modify your code to allow you to plot `y` against `t`). Submit your **plot** as a single png file to Handin. **[4]**

Q6.7 Try different starting values for $y(t{=}0)$ and carefully describe the output for $g(t) = \cos(\omega t)$, up to times corresponding to several periods. **[3]**

Q6.8 Determine approximately the amplitude and phase (including its sign) of the output, relative to the input $g(t)$, for each value of ω. **[2]**

6.3 Using odeint from SciPy

In this section we have developed a crude method to integrate a first order ordinary differential equation. However, just as there are much more sophisticated methods for integration than the trapezium rule we used in section 4, so there are highly developed methods for integrating differential equations. Here we'll see how to use `odeint` from the **scipy** module. There are two advantages here: (i) `odeint` uses more intelligent integration algorithms with things like adaptive stepsize control to get the fastest solution for a given error, (ii) `odeint` is compiled from FORTRAN and saved as an object which can be called from Python, making it much faster to run than a pure Python code. A potential disadvantage is that if you treat `odeint` as a "black box" routine, you have to trust that its writers did a good job and you understand exactly what it produces. In this case `odeint` is a "wrapper" around the `lsoda` integrator, part of the ODEPACK library[1].

We will find solutions to the equation of motion of a damped harmonic oscillator (ignoring the fact that this can be done analytically...). The differential equation is

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0$$

where γ measures the strength of the damping and ω_0 is the natural (undamped) angular frequency.

To integrate this we first have to convert it to a pair of first-order differential equations. To do this, we let $u = dx/dt$ and find

$$\frac{dx}{dt} = u$$

[1] *ODEPACK, A Systemized Collection of ODE Solvers*, in RS Stepleman et al (eds), *Scientific Computing*, IMACS Transactions on Scientific Computation, Vol 1, North-Holland, Amsterdam (1983) pp 55–64.

$$\frac{du}{dt} = -\gamma u - \omega_0^2 x$$

We combine x and u into a two-component vector $\mathbf{y} = (x,u)$, with a corresponding vector of derivatives, $d\mathbf{y}/dt = (dx/dt, du/dt)$. In this vector notation, the differential equations are

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y},t).$$

For the damped oscillator,

$$\mathbf{f}(\mathbf{y},t) = \begin{pmatrix} u \\ -\gamma u - \omega_0^2 x \end{pmatrix}$$

To use `odeint`, we must tell it the function $\mathbf{f}(\mathbf{y},t)$ giving the derivatives, the initial conditions $\mathbf{y}_0 = (x_0, u_0)$, and a list of times for which we want the solution. The output of `odeint` is a list of (x,u) values for each of the times we asked for. In other words we get a matrix where the rows correspond to different times, while the first column is x and the second column is u.

Here is a python script which uses `odeint` to integrate the damped oscillator problem and plots the output $x(t)$. Try it out. Note that the parameters γ and ω are used as arguments to the function which calculates the derivatives and observe how the call to `odeint` distinguishes the parameter arguments from $\mathbf{y}$ and t. If your derivative function has no extra arguments for parameters, then you can leave out the `args=` argument to `odeint`.

```
from numpy import arange, array
from scipy.integrate import odeint

def f(y,t,g,w): # func for derivatives, y is array
    # g is damping, w is nat freq
    x,u=y   # position/velocity are 0th/1st cpts
    return array([u,-g*u-w*w*x]) # array of derivs

g=0.2; w=3.0

tl=arange(0.0,20.1,0.1) # give soln at these times
x0=1.0;u0=0.0           # initial values
y0=array([x0,u0])       # ... put in array
yl=odeint(f,y0,tl,args=(g,w)) # solve ODE's
xl=yl[:,0]              # extract x(t)
ul=yl[:,1]              # extract u(t)=dx(t)/dt

import matplotlib.pyplot as plt
plt.plot(tl,xl)       # plot x(t)
plt.show()
```

Section 7

A Lattice Problem

7.1 Theory

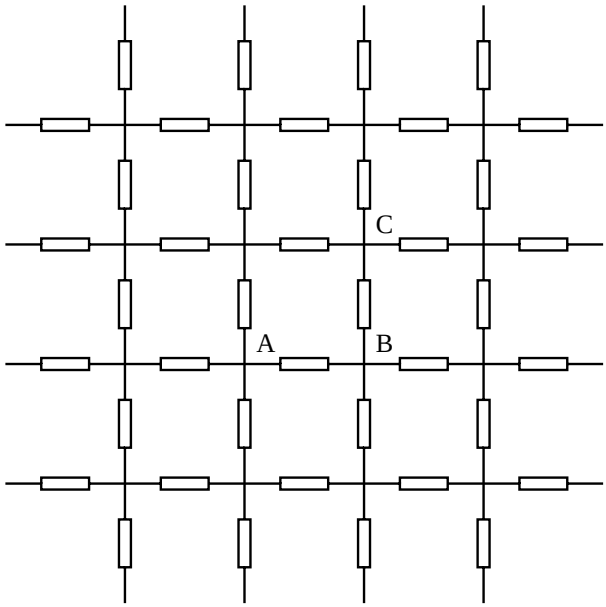


Figure 7.1 A network of resistors.

In this section we consider the following problem. A very large square network of identical resistors is arranged as shown in figure 7.1. You are to imagine that the figure shows only part of the network. It extends beyond the figure and eventually we will make it infinitely large. Suppose I measure the resistance between the nodes A and B across the edge of a square, or between A and C across the diagonal, what values do I get?

This problem is important, not for its own sake, but because the method of solution can be used for other more practical problems. If the measuring points are separated by many squares the fact that there are discrete resistors becomes relatively unimportant and the potentials approximate those for the problem of two electrodes attached to a uniform conducting sheet. The potential distribution in electrostatics and the temperature distribution in thermal conduction obey the same equations, so there are many applications.

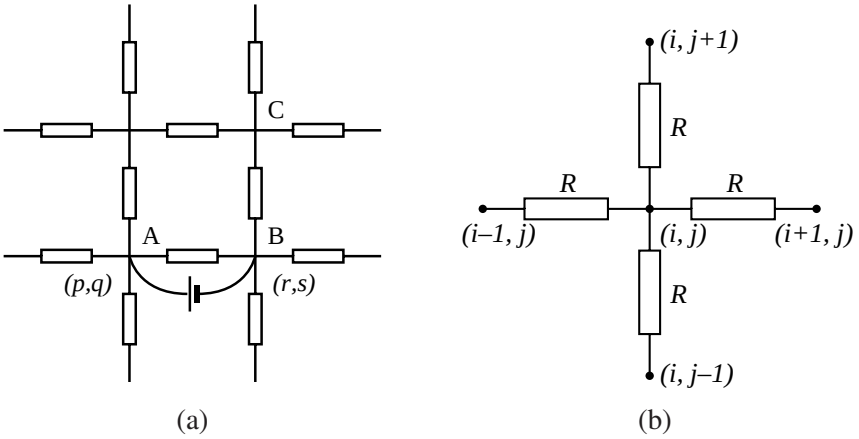


Figure 7.2 (a) A battery connected across two nodes of the network; (b) The four resistors and neighbouring nodes of a given network node.

Label the nodes (junctions) in the network by a pair of indices (i, j). We can choose any node $i = p$, $j = q$ and we can perform an experiment to determine the resistance between this node and another nearby node (r,s), as follows. Suppose we connect a battery between these nodes and arrange for a current I to flow into the network at (p,q) and out at (r,s) as shown in figure 7.2(a). Now imagine we measure, with a voltmeter, the potential differences between each node (i, j) and somewhere at a great distance near the edge of the network. Let these measurements be V_{ij}. The potential difference between the nodes (p,q) and (r,s) is then $V_{pq} - V_{rs}$ and so we can see from Ohm's law that the apparent resistance R_{pqrs} between the nodes where the battery is connected must be:

$$R_{pqrs} = \frac{V_{pq} - V_{rs}}{I}$$

The theoretical solution to this problem therefore hinges on calculating the potentials V_{ij}. Figure 7.2(b) shows the resistors, each with resistance $R = 1\,\text{ohm}$, and 4 nodes surrounding node (i, j). Suppose for the moment that (i, j) is *not* a node where an external battery connection has been made. The currents flowing in these resistors are $(V_{i+1,j} - V_{ij})/R$, $(V_{i-1,j} - V_{ij})/R$, $(V_{i,j+1} - V_{ij})/R$ and $(V_{i,j-1} - V_{ij})/R$ and the sum of these must be zero since no current enters or leaves the network at (i, j). Therefore

$$V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{ij} = 0 \tag{7.1}$$

For the two special nodes, (p,q) and (r,s), where the battery connections are made, the sum of the four currents equals (minus) the current flowing out of or into the battery. The right hand side of the equation is then replaced by $-IR$ or $+IR$. These equations encode all the physics of the problem and we have to solve them.

7.2 The computer solution

The code for solving this problem can be loaded from the file **network.py** (available on Blackboard). It is also listed below. Look at the code while reading the following explanation of how it works.

```
from numpy import zeros, arange, fabs

def sweep(v,p,q,r,s):
  for i in range(1,len(v)-1):
    for j in range(1,len(v)-1):
      c=0.0
      if i==p and j==q: c=1.0
      if i==r and j==s: c=-1.0
      v[i,j]=0.25*(v[i-1,j]+v[i+1,j]+v[i,j-1]+v[i,j+1]+c)

N=12
v=zeros((N,N),float)
p=q=int((len(v)-1)/2)
r=s=p+1

dv=1.0e10
lastdv=0
count=0
while (fabs(dv-lastdv)>1.0e-7*fabs(dv)):
  lastdv=dv
  sweep(v,p,q,r,s)
  dv=v[p,q]-v[r,s]
  count+=1
  print(count, dv)

import matplotlib.pyplot as plt
plt.figure(figsize=(8,8)) # square plot for square grid
plt.contour(v)
plt.show()
```

We need to store the potentials V_{ij} in the computer, using a two-dimensional array. Since we cannot calculate with an infinite network, we must fix some size and see how the results change as we increase the size. For reasons you will see later we will use an array of size $N \times N$ but only use $(N-2) \times (N-2)$ nodes. It is easy to create an array filled with floating-point zeros using the `zeros()` function from **numpy**.

```
N=12
v=zeros((N,N),float)
```

Note that we have used a variable `N` to store the size of the grid. We can then use `N` everywhere else in our code and it will be simple to change the size of the grid simply by changing the value of `N`.

7.2.1 The iterative solution of the equations

The algorithm that we are going to use to solve the equations is called *relaxation*. You can see from equation (7.1) on page 53 that at all the nodes, except those where the external connections are made, the potential is the average of the values at the four adjacent nodes. So we start with *any* values of the potentials, zero in our program, and keep replacing the potential at each node in turn by the average of the four adjacent values. At the two nodes with external connections the external current must be included. After a number of sweeps through the network the potentials "relax" to the solution of the above equations. If you think carefully you can see that *if* the potentials converge to constant values then these values satisfy the equations. It is not obvious that they will converge, or how fast.

The following lines replace each value of `v` by the average of four adjacent values:

```
for i in range(1,len(v)-1):
  for j in range(1,len(v)-1):
    v[i,j]=0.25*(v[i-1,j]+v[i+1,j]+v[i,j-1]+v[i,j+1])
```

However, we also need to take care of the external currents. If the current enters and leaves at nodes labelled (p,q) and (r,s) respectively, we modify the lines above to

```
for i in range(1,len(v)-1):
  for j in range(1,len(v)-1):
    c=0.0
    if i==p and j==q: c=1.0
    if i==r and j==s: c=-1.0
    v[i,j]=0.25*(v[i-1,j]+v[i+1,j]+v[i,j-1]+v[i,j+1]+c)
```

where `c` is the extra term arising from the current in the battery connections. It is a good idea to make this a function which is called `sweep(v,p,q,r,s)` in **network.py**. Using functions like this helps break up your problem into separate parts. This can make your code more understandable and reduce the chance of you making errors.

Note that in `sweep`, the array `v` that is passed to the function gets its values changed: in some languages a function which changes the values of some of its arguments is called a **procedure**.

If you look at the code lines above you will see that we change the values of `v` only at nodes whose labels are in the range 1 to $N-2$ (since `len(v)` is N) whereas we allocated space for nodes 0 to N. This is to take care of the edges of the network. If (i, j) is a node on the edge then some of $i \pm 1, j \pm 1$ are outside the network. You cannot have a subscript outside the range allocated so we have allowed an extra band of nodes all round the network and change only the elements numbered 1 to $N-2$. We have set the elements on the edges to zero and they are not changed during the calculation. The simplest thing is to leave the program like this, the potentials will be small anyway. What we have done is made a choice of *boundary conditions*: for a large enough lattice, the particular choice should make no difference to our final answer.

At the end of the script we use the function `contour(v)` from pyplot, which produces a contour plot of the potentials. We also use `figure(figsize=(8,8))` to make a square plot since we have a square grid.

The program has been written with the battery connected between the nodes `(p,q)` and `(r,s)` as discussed above. Take `p,q` near the centre, and, if you want to measure the resistance between A and C in figure 7.1, for example, set `r=p+1`, `s=q+1`.

7.3 Measurements using your program

Remember the aim of the calculation is to get the resistance between A and B and between A and C in figure 7.1 for an *infinite* network. As you increase N the program eventually gets too slow or uses too much memory, and the results for these finite networks are still changing significantly as N increases. This situation is common in theoretical calculations on, for example, crystals where we can handle only a small number of atoms and we would like results for the bulk material.

Q7.1 Create your own copy of the `network.py` module. Use your program to determine the resistance between adjacent nodes across the diagonal of a square (nodes A and C in figure 7.1) and the edge of a square (nodes A and B), using a network of 5×5, 10×10, 15×15 and 20×20 nodes. Note that to measure the resistance between A and B you need to change the battery connections to these points. **[5]**

Q7.2 Plot these resistances against $1/N^2$ (you can use Python for the plotting!) and extrapolate (by eye) to $N = \infty$ to estimate the values for an infinite network. Save your plot as a `png` image to submit to handin, and record your extrapolation estimate in your electronic lab book. **[5]**

7.3.1 Speeding up your program

Your program will get slow on larger networks. A significant improvement in speed comes from a modified algorithm. In two dimensions, if we write the equation

$$V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - \alpha V_{ij} = (4-\alpha)V_{ij}$$

we can use a similar iteration to the one used before but we are free to pick a value for the constant α. A judicious choice can greatly speed up convergence.

Q7.3 Create a modified version of your code to use the new method, including α. Try $\alpha = 1$ and count the number of iterations required for convergence in the old and new methods. Vary α to look for an optimal choice, reporting your findings. **[5]**

Another improvement is possible if we note that there are symmetries in the potential and it is sufficient to store a quarter, or even one eighth, of the network. In this way much larger problems are solvable. You are not expected to do this.

Q7.4 Make measurements of the resistance between various pairs of points (keep one fixed, you gain nothing by moving both) and plot a graph of how the resistance changes with the physical separation between the measuring points. Use a 20×20 mesh and include all your measurements on the same graph.**[5]**

Incidentally the problems of computing the resistances between neighbouring nodes can be solved theoretically and the answers are fairly simple numbers. The solution for neighbours across a diagonal is much more complicated.

7.4 A final comment

Your code can take a noticeable time to run for very large networks. This is a disadvantage of using an interpreted language like Python to loop explicitly over the elements of large arrays. By using the **numpy** module's faster vector and matrix handling, you would be able to speed things up considerably.

For many programs, the bulk of the processing time is spent in a minority of the code. It's possible to write the CPU-intensive part of the code in say C, C++ or FORTRAN and call this from a Python program. This combines the ease of use of Python with the speed of compiled code for the most demanding part of the calculation.

Alternatively, you may have a pre-existing program written in C, C++ or FORTRAN. Here you could write a Python script to control the existing program, feeding it input and collecting the output for further analysis.

Section 8

Monte Carlo Simulation

Some physical systems are so complicated that it is either too difficult to write down the equations describing them, or these equations may be too difficult to solve. In these circumstances a computer method called *Monte-Carlo simulation* may be a way of making predictions or confirming that a theory adequately accounts for an observed phenomenon. Programs for this are often short and simple to write, but they can consume very large quantities of computer time and very accurate results are difficult to achieve.

8.1 Estimation of the critical mass of uranium 235

The aim of this section is to estimate the critical mass of uranium 235.

The chain reaction in a nuclear reactor or nuclear bomb goes as follows. A uranium atom is hit by a stray primary neutron and undergoes fission, with the release of energy, and the production of several more neutrons. These neutrons may escape from the piece of uranium or they may hit other nuclei and initiate further fissions. In a small piece of uranium most neutrons will escape but above some critical volume a self-sustaining chain reaction will occur.

Rather than try to follow the fate of all the neutrons produced in all the subsequent fissions, we will track just those neutrons produced in the first fission, and count how many secondary fissions they initiate. Since this number will fluctuate, we will have to repeat our simulation for some large number of initial fissions to find the average number of secondary fissions per initial fission.

8.2 A one-dimensional model

To begin with we will simplify to a one-dimensional model. Suppose the uranium extends along a line from 0 to L. The initial fission may occur anywhere along the

line, so we generate a random number uniformly-distributed in the interval from 0 to L using `L*random.random()` (after importing `random` from numpy). The number of extra neutrons produced in each fission varies; on *average* it is 2.5, but for the moment we will simply take it to be 2 in every case.

These neutrons move along the line, colliding with uranium nuclei. In most cases they simply bounce off, but there is a chance of initiating another fission before they diffuse out of the end of the line. It can be shown that the average (rms) distance that a neutron diffuses away from its starting point before causing a fission is $R = (2ab)^{1/2}$ where a is the mean free path between the neutron hitting another nucleus and bouncing off, and b the mean free path between the neutron hitting a nucleus and causing fission. Experimental values are $a = 1.7\,\text{cm}$ and $b = 21\,\text{cm}$. For the moment assume that all the neutrons travel exactly the average distance.

The procedure is now to generate some number, say 100, of initial fissions. In your code you will need an outer loop counting these 100 initial fissions. Within the outer loop you will need to deal with each of the two secondary neutrons and the easiest way is to use a loop counting up to 2. For the moment assume that all the neutrons travel exactly the average distance. They may be travelling to the right or the left (remember we are in one dimension) and we ignore the possibility that their directions might be correlated. So for each of the two neutrons generate a random number `random.random()`. If it is greater than 0.5 take the neutron moving to the right, otherwise to the left. Add or subtract the average diffusion length from the initial position (according to whether the neutron is going right or left) to find the possible positions for the next two fission events. If a calculated position is outside the range 0 to L then the neutron will escape without causing further fission. Use a variable to count the secondary fissions as they occur. Increase it by one for each neutron that causes a secondary fission inside the bar.

Q8.1 Write a program to model the one-dimensional Uranium, according to the above description. **[10]**

Notice that the number of secondary fissions fluctuates if you run the program several times.

Before attempting accurate measurements on the critical size of the uranium we should allow for the fluctuations in the number of secondary neutrons. The file `neutrons.py` (available on Blackboard) contains the function `neutrons()`, designed to give integer values with average 2.5 and with a distribution resembling that observed experimentally. `neutrons.py` also contains another function, `diffusion()`, which you will need later.

Q8.2 Create an updated version of your program using the `neutrons()` function. **[1]**

Q8.3 Start with $L = 0.1\,\text{m}$ and vary this to determine the critical value. Decide whether you need to use more than 100 initial neutrons. **[2]**

8.3 Extension to three dimensions

One dimensional models cannot be taken very seriously but are often used to get an order of magnitude estimate when the full three dimensional calculation is beyond the capacity of the computer available. In this case, a three dimensional simulation is feasible.

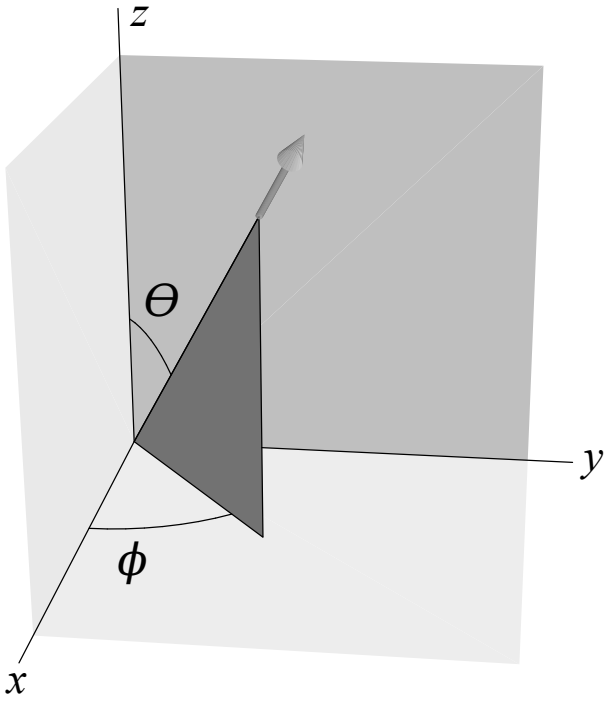


Figure 8.1 Specifying a direction in three dimensions with angles θ and ϕ.

Consider a cube of uranium of side L. The initial fission requires 3 coordinates (x, y, z) chosen uniformly between 0 and L. The secondary neutrons are emitted in random directions. Think of the fission occurring at the centre of a sphere and the direction of an emitted neutron being specified by the latitude and longitude where its track crosses the sphere. We conventionally use the *co-latitude*, θ, that is the angle from the North pole, *not* from the equator, as shown in figure 8.1. All angles of longitude ϕ are obviously equally likely so we can generate a random value for ϕ with

```
phi = 2.0*pi*random.random()
```

In contrast, values of θ are not equally likely; we want the neutron paths to be uniform over the surface of the sphere, and a small range of θ near the pole corresponds to a much smaller area than the same range near the equator. It can be shown that $\cos\theta$ is uniformly distributed, so we can generate θ from

```
theta = acos(2.0*random.random()-1.0)
```

The model for the diffusion of the neutrons can be improved. It can be shown that the probability that the neutron diffuses a distance s before causing fission is proportional to $s^2 \exp(-3s^2/2R^2)$. The function `diffusion()` (in `neutrons.py`) generates a random number s with this distribution. These numbers have average value of unity so need to be scaled (multiplied) by $R = (2ab)^{1/2}$ before use. The coordinates of the next fission are then given by:

$$\begin{aligned} x' &= x + s\sin\theta\cos\phi \\ y' &= y + s\sin\theta\sin\phi \\ z' &= z + s\cos\theta \end{aligned}$$

Remember that in your program you will need to test whether this point is inside the cube.

Q8.4 Update your program for the 3-dimensional model, as described above. **[5]**

Q8.5 Vary the value of L and find the critical value and hence the critical mass. The density of uranium is $18.7\,\mathrm{Mg\,m^{-3}}$. **[2]**

Appendix A – Configuration

Here we give some help for configuration of Python.

A.1 Files for PHYS2022

The data files and modules for this course are located on the PHYS2022 Blackboard page.

A.2 The current directory

If you try to open a file `lemur.dat` using `f=open('lemur.dat','r')` for example, then Python will look in the "current directory" to find it. Likewise, if you want to save a plot using `plt.savefig('llama.png')`, the file will be written to the current directory. If you are running a script, then the current directory is the one containing the script. However, if you are working interactively in the shell, it may not be clear what the current directory is. You can find out by using the `getcwd` command from the `os` module:

```
import os
os.getcwd()
```

If this is not the directory you want, you can change it using the `os.chdir` command. For example, in Windows, if my username is `palin`, then I could set the current directory to my Documents with a command like (note the use of forward slashes):

```
os.chdir('C:/Users/palin/Documents')
```

Setting the current directory like this may be more convenient than specifying a full path to a file every time you want to read and write data or save plots.

Be careful not to use `os.chdir()` in code submitted to the Handin system - the directory you try to change to will almost certainly not exist on the Handin server!

A.3 The module search path

When you import a module, for example `import gecko`, Python has to be able to locate a file `gecko.py`. To do this it searches first in the current directory and then in a list of directories called the search path. Standard modules like numpy or matplotlib should be installed so that they are in the search path. However, local modules, such as minimise and neutrons used in this course, may or may not be in the path.

If you are running a script, the first entry in the path should be the directory containing the script, so it should be simple to import modules from the same directory as the script. If you are running interactively, it may not be so clear what the search path is. You can find out by using:

```
import sys
print(sys.path)
```

If it does not contain the directory you want, there are a number of things you can do.

- You can set the PYTHONPATH environment variable to include directories you would like on the path. For example, you could put all your python code and modules for this course in the same directory and add this directory to the PYTHONPATH.
- Since `sys.path` is a Python list, you can append a string containing a directory name to this list before trying to import a module from that directory. For example, to add the Windows H drive to your path, use:

  ```
  sys.path.append('H:\\')
  ```

- Another alternative is to copy a Python startup icon to your desktop. Right-click on the icon, select "Properties" and look for the "Start in:" field in the "Shortcut" tab. Edit this field to contain a directory (folder) name. You should find that this directory is on the path of a Python shell launched using the edited desktop icon.

Appendix B – Basic Pyplot Plotting Commands

Pyplot was designed to make it easy to produce basic plots, but allow full customisation if you need it. Using pyplot in Python scripts should not be problematic: just remember to use a show() command to make your plot actually appear in a window, or savefig() command to save your plot in an image file.

Here we give some of the commands used in this course:

`plot(x,y)`	`x` and `y` are lists or arrays of x and y coordinates
`plot(y)`	`y` is a list/array of y coordinates
`errorbar(x,y,dy)`	`x` and `y` are lists/arrays of x and y coordinates, with `dy` a list of errors. Each data point is $y_i \pm dy_i$ at x_i. To plot points as dots with a simple line for the errorbar use `errorbar(x,y,dy,fmt='o',capsize=0)`
`bar(bins,cts,width=w)`	draw bars with left-hand edges given by the list/array `bins`, heights specified in the list/array `cts` and each bar of width `w`
`contour(v)`	contour plot using values from a two-dimensional array `v`
`xlabel('text')`	labels x-axis with text; similarly for `ylabel`
`title('text')`	title for the plot
`axis(xmin=0.0)`	set lower limit of x-axis; similarly for `xmax`, `ymin` and `ymax`
`axis([xl,xu,yl,yu])`	set lower and upper limits of x and y axes to (xl,xu) and (yl,yu) respectively
`subplot(r,c,n)`	`n`'th subplot in a grid of `r` rows by `c` columns
`subplots_adjust(left=0.1)`	set left border to 10% of the figure width; similarly for `top`, `bottom`, `right`, `wspace`, `hspace`
`show()`	draw the plot in a window
`clf()`	clear the current figure
`cla()`	clear the current axes
`savefig('tapir.png')`	save a plot (in png format)

Note that you should NOT use the `show()` or `ion()` commands in any code submitted to the Handin system.

Appendix C – String Formatting

In Python, variables can be formatted as strings using the `"{}".format(x)` method. Some examples are provided below, and you can find further information and examples at http://pyformat.info.

```
>>> a="a string"
>>> b=10 # An integer
>>> c=5.6 # A float
>>>
>>> # Pad string with spaces to 10 characters,
>>> # and append an integer:
>>> print("{:10s} {:d}".format(a,b))
a string   10
>>> # Pad integer with zeros to 6 characters:
>>> print("{:s} {:06d}".format(a,b))
a string 000010
>>> # Show float with 3 decimal places:
>>> print("{:.3f}".format(c))
5.600
>>> # Show float with 2 decimal places, in a 10-character
>>> # long string:
>>> print("{:10.2f}".format(c))
      5.60
>>> # Pad float with zeros to a total width of 5 characters:
>>> print("{:05.1f}".format(c))
005.6
>>> # Align a string to the right:
>>> print("{:>20s}".format(a))
            a string
>>> # Include other constant (i.e. not a variable) text:
>>> print("Current = {:.3f}A, Voltage = {:+d}V".format(c,b))
Current = 5.600A, Voltage = +10V
>>> # Use format as part of a string assignment operation:
>>> x="You have {:d} apples.".format(b)
>>> print(x)
You have 10 apples.
>>> # Always include a sign (+ or -):
>>> print("{:+.3f}".format(c*-10))
-56.000
```

If you have trouble understanding what the above examples are showing ask a demonstrator for help.

Appendix D – Common Python Errors

Here are some common syntax errors:

- Using a Python keyword or builtin function name for a variable name. For example `class` is a keyword and `sum` is a builtin function.
- Forgetting the colon in a `def`, `for`, `while` or `if` statement.
- Forgetting to match quotation marks around strings.
- Mismatching parentheses.
- Using assignment, =, when you mean an equality test, ==.
- Inconsistent indentation of statements.

Another common error is to forget the `return` statement at the end of a function definition.

Appendix E – Some Notes on Programming Style

The purpose of a good style is to decrease the chance of making errors, particularly errors that cause the program to give plausible but wrong answers. Here are some things to think about.

- Use spaces and blank lines freely to make the structure of your program clearer.
- Do not write `A/B/C` or `A/B*C`, they are ambiguous to humans[1].
- Use meaningful names for variables and functions. Where you cannot make your meaning clear by choice of names put explanatory remarks in comments.
- If you are coding a page of algebra, use the same symbols in the program as on the written page. Do not do a mental translation: you will make mistakes.
- Add clear and helpful comments to your programs, so that it is easy for somebody else to understand your code, or for you to understand it some years later.
- Many programs by physicists are written to solve one problem, are used over a period of a few days by the person who wrote them and are then finished with. There is no point in writing elaborate user interfaces for such programs or including extensive annotation. However physicists often re-use parts of programs, a routine for integration, for example: these should be annotated and carefully documented.

[1] Computer languages specify the *precedence* of mathematical operations, so the expressions given should not cause the interpreter or compiler to complain. The danger is that you may not remember the precedence rules correctly.