Part II Chemistry Programming Practical Option

1 Introduction

1.1 Preface

This course has been developed in response to requests for a more structured programming practical course for chemistry. In particular, as this is only the third year of running it, you may find some errors, bugs, and inconsistencies in the material, and it would be very helpful if you could let me know of these so they can be fixed for the future.

The Corporate Associates scheme has very generously funded the writing of this course, and I am also very grateful to Mark Driver who has proofread and written sections of these notes and test-driven the exercises. Any remaining errors should be entirely regarded as my own fault. I would also like to thank Dr Adrian Nickson provided inspiration for and explanation of the protein folding kinetics in Exercise 3.

I very much welcome any ongoing feedback as to the usefulness, difficulty, and relevance of the material in these notes to help refine the course for later years.

Alex Thom, October 2017

1.2 Scope of the Course

This course is designed to cater for students with a wide-ranging ability level and background in programming, so very little prior knowledge is assumed. The successful outcome of the course will be that you have the ability to write programs of moderate complexity, including using data structures, and libraries for plotting graphs and numerical methods, and by the end, successfully automate tasks or data analysis or implement new algorithms. However, the aim of the course is not to provide a comprehensive step-by-step tutorial in programming — many such guides already exist, and a number of existing resources will be referred to throughout the course. Instead this course is designed to provide a framework and motivation for a sufficient level of self-teaching, aided by chemically relevant exercises. Primarily, support will be by means of weekly demonstrating sessions, and assessment by submission of the code written and relevant output files.

One of the benefits of being able to start a course afresh is to use modern languages and programming techniques, and so the course has been designed around the Python 3 language and its relevant libraries. This is primarily because of the ready availability of the language, copious Internet-accessible documentation, vastly powerful and easy-to-use existing libraries, and the relatively low start-up hurdles to using an interpreted language. The techniques learnt should, however, allow you to broaden your knowledge of programming to other languages (both ancient and modern) without much additional effort. In addition, I hope to introduce you to a modern style of programming and tools which will make the transition to larger-scale coding considerably less painful.

1.3 Requirements

The basic requirements of this course are available on the PWF workstations (in G30) when booted into Linux, but you are free to use any computer with the following software.

Python 3 (any version)
The scipy libraries:
`matplotlib`
`numpy`

VMD is also useful for the last exercise. This is installed on the PWF. If you have a working `ssh` program (with X-forwarding), you can also ssh (using your PWF password) by `ssh -X abc12@linux.pwf.cam.ac.uk`, where abc12 is your CRSID.[1].

1.4 Assessment

Assessment of the exercises will be entirely electronic, and you should submit your source code and any related files and output files as an attachment to an email (preferably as a zipped tarball) or as a git pull request to Dr Alex Thom (`ajwt3@cam.ac.uk`), including instructions on how to execute the code to perform the tasks specified. You must ensure that your code runs on the PWF machines in Python 3.

Once you have submitted a solution to each exercise, you should signup for a mini-viva (on the termly doodle poll), which occur weekly, and where you will be asked to talk through your code and algorithm, and given feedback about its style and function. There is no deadline for individual exercises though an exercise is unlikely to be marked with only 24 hours notice, and it is recommended that you wait for feedback before moving on to the next exercise.

Successful submission of Exercises 1 and 2 will gain 1 credit for Advenced practicals, and that of Exercises 3 and 4 will gain a further 1 credit. All exercises must be submitted by the Monday of Week 8 of Lent Term (12th March 2018).

[1] This means you could work from your room without having to install Python and all the libraries

2 Scientific Programming

2.1 Course Map

This is a rough guide to the concepts that you'll need and when you should be able to complete the exercises (the concepts come before each exercise). Though you may be able to complete the exercises earlier, waiting until you have mastered the techniques required will make the process significantly simpler. Items with an asterisk are covered in the techniques part of the exercises, but it wouldn't hurt to look at them in other references.

- Simple Variables
- Loops
- Inputting Data from the keyboard
- Outputting Data (`print`)
- *Tuples and Lists
- *Manipulating Lists
- *`numpy` libraries and structures: vectors, matrices, diagonalizers.

  Exercise 1

- *File input and parsing
- *The `matplotlib` library
- *`numpy` data interpolation

  Exercise 2

- Dictionaries
- *`numpy.loadtxt`
- *Classes

  Exercise 3

- *Modularisation and objects.
- *Operator overloading.
- *Interfaces.

  Exercise 4

2.2 Learning to program

Computer programs are seldom written in the abstract, and are usually designed and written to be used. This course follows from the philosophy that programming can only really be learnt when applied.

You should start by following one of the online courses in the section **Python 3: The basics** below, which will give you an introduction into the basics of coding in Python. Other sections deal with some of the specifics of the local computers, and give guidance as to how to log in, edit programs and run Python. When following an online course, it is advisable to actually type (or paste) the examples and run them (either interactively using IPython, or Python itself, or by typing them into a file, and running the file through Python) and playing around with them to understand them[1]. Once you have a sufficient familiarity with Python itself, you can try to tackle the Exercises below, which list their various requirements. You will most likely have to look up documentation and techniques as you go along, and potentially revisit sections of the courses, so do not feel you have to be an expert before starting to write your own programs.

2.3 The Structure of Exercises

Section 3 consists of four exercises which will incrementally guide you into the key concepts in programming, introducing concepts as they are needed. You should therefore complete them in order. Each exercise can generally be completed by writing a single program file, or a core set of routines which can be called from a short other program file. Each exercise begins with a list of concepts assumed and concepts learnt, and consists of the following sections:

- **Specification** — A concise statement of the capabilities of the core program you should write. If you were writing a program for somebody else this is typically what you would receive. A detailed specification would also contain details of how data is to be input and output (especially the formatting), but this is often left out.
- **Tasks** — A list of specific tasks which your program should perform to demonstrate that it actually works. You might write a separate control program to perform each task, or have one which selects which task to perform and calls the main body of the program.
- **Techniques** — A selection of example snippets of code which demonstrate the techniques which are relevant to the exercise. You should ensure you understand why each line does as is commented, and can ask your demonstrator to explain.
- **Discussion** — A discussion of the ideas underlying the problem and some suggestions as to how you might break the problem down into parts to build up your eventual program. Possibilities for structures, and warnings of potential pitfalls are also included here.
- **Theory** — A description (hopefully a reminder) as to the theory behind a problem.

The examples in this guide are designed to expand on key concepts that you will have learnt from an existing Python course (either one mentioned above or one you have found). They should help you to grasp what the exercises require and what you need to be considering when designing a

[1] This is akin to actually doing the exercises in a textbook or the examples sheets for a lecture course.

program to perform these tasks. The practical demonstrating classes are for you to ask questions rather than be told how to complete the exercises.

The discussion section of each exercise suggests a number of ideas to think about when designing the program, and may suggest routes to use. You are, of course, free to ignore these suggestions if you have ideas about simpler or alternative algorithms.

2.4 Python Introduction

2.4.1 Python 2.x or Python 3.x

The Python language was created in the 1990s[2], and only really became popular from the year 2000[3] with Python 2.0 (which has since transitioned to Python 2.7), but with increased uptake it became clear that some design flaws in the original language needed to be remedied, and Python 3.0 was released in 2008 and was deliberately not backward compatible (in slight, but fundamental ways). The two languages have coexisted since then, and features (and libraries) in 3.x have been back-ported to 2.x Whilst large amounts of code still exists in Python 2.x, Python 2.7 is expected to only be supported until 2020, and so new users are encouraged to start with Python 3.x (currently 3.5). You will easily be able to switch between the two languages as the differences are minor.

2.4.2 Python 3: learning the basics

There are several resources for learning Python 2.7 and Python 3, searching for online courses yields millions of results. Unfortuantely the programming transition from Python 2.7 to Python 3 is a slow one (as it requires rewriting code), so you will find a lot of code written in Python 2.7. The primary difference for beginners is the change in printing:

Python 2.7	Python 3
`print "Hello World!"`	`print("Hello World!")`

so you should be able to use the majority of Python 2.7 courses.

The UCS has run courses on Python 3. A copy of the lecture notes of the course for absolute beginners can be found at `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/PythonAB`. If you have already got some programming experience you could look at `http://bit.ly/1Lt9T3I` .

There are also several websites that may prove useful. Codecademy provides an interactive course. It can be found at `http://www.codecademy.com/en/tracks/python`[4] . Another possible

[2]Python's creator, Guido van Rossum, writes

> in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).
>
> *van Rossum, Guido (1996). Foreword for "Programming Python" (1st ed.)*

[3]which gives it the classification as a 'modern' programming language

[4]This uses Python 2.7, but gives a very good overview of Python which can be completed in about a day and a half of focused study.

website is `http://www.learnpython.org/` where the "Learn the Basics" section should be sufficient (again python 2.7). There is also a Google class as well `https://developers.google.com/edu/python/`.

The Non-Programmer's Tutorial for Python 3 on wikibooks (`https://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_3`) is a gentle introduction. The Python Tutorial as part of the documentation (`https://docs.python.org/3/tutorial/`) is probably not suitable unless you are already a programmer.

These are just a selection of the resources already out there to learn the basics of Python. Another important source of information is the Python documentation. This is incredibly useful and contains information about several standard libraries that could be useful for future programming projects. There are also several resources that describe the differences between Python 2.7 and Python 3.

2.4.3 Running Python 3

If using Linux (see Section 2.5), the Python 3 interface can be loaded directly in your terminal using the command `python3`. This will load a session of the Python 3 interpreter. Note if you just use `python`, you will load Python 2.7.6, a legacy version of Python, which although it is similar, there have been some significant changes between Python 2.7 and Python 3. In this course you are expected to use Python 3 when programming. It is worth noting that most programs written in Python 2 will run in Python 3 and vice versa.

You can run your scripts from the command line by `python3 myscript.py`. This will run myscript.py (note the file extension is .py) then terminate and return to the command line.

2.4.4 IPython3 and other IDEs

An Interactive Development Environment (IDE) is another way to program. These often contain an interpreter as well as an editor for source code and sometimes debugging tools. There are several available for Python 3.

IPython3 is one such IDE, and is installed on the PWF machines. This is loaded from the terminal using `ipython3`. IPython3 is different to other IDEs because it runs in the terminal[5]. It passes commands to the terminal if the built in interpreter does not recognise it. It also has a useful debugging mode. More information can be found on the IPython website at `http://ipython.org/`.

2.4.5 Coding Practices

The code segments you see in this guide should all be compliant with PEP8 and PEP257. These are coding style guidelines which are designed to ensure any Python programmer can read someone else's Python scripts and understand them. This includes standard Python libraries. These can be found at `http://legacy.python.org/dev/peps/pep-0008/` and at `http://legacy.python.org/dev/peps/pep-0257/` . To aid debugging, use of these style guides is recommended. Poorly organised and documented code is hard to understand by everyone (including you if you do not use the code for an extended period of time). This could seriously affect your mark for the exercises if it isn't easily understandable. It is also important to get into good habits early. Any future coding

[5]This means it can be run over an ssh connection, but does not have such luxuries as an editor — you might think of it more as an Interactive Debugging Environment, albeit one in which is quite convenient to write code.

projects you undertake could be much larger, and written by several people, making readability a key feature.

2.4.6 Commenting

Comments do not pass to the interpreter when run. They are purely for the benefit of the programmers using the script. They should help you (and anyone reading your program) understand what is happening. Too many comments can be bad and confusing to the reader (including you) and make it harder to spot bugs. A simple rule is if it can be written more clearly in code than text, leave it as code; it doesn't need a comment to be added.

Comments are very useful when describing what functions, classes and their methods do. They can also be useful when you are doing data manipulation or plotting for a specific system. For example, earlier in your code you have created general functions and/or classes to hold and manipulate data, now you are using them to extract information about a specific data set or system (e.g. Exercise 3 — the protein folding and Oregenator equilibria can be evaluated with very similar or identical classes and functions, but with different input values). Commenting that you are now evaluating a specific system can also lead to faster error location. If a function works for one input, but not another either there is an issue with the format of your input or an underlying fault in the function that was not found on the first use.

2.4.7 Source Code Management

If you are writing programs sensibly in stages, you will reach a point where you have tested something and it works, and you might like to save the state of your program. One option would be to copy the file to back it up, but that will create an unwieldy series of old files (often strangely named) which will be basically unintelligable to anybody else.

If you are working on a larger project, a good habit to get into is that of Source Code Management. My preferred SCM sofware is `git`[6]. `git` is not entirely intuitive to the users of other SCM systems, but for simple usage it a workflow is

```
git init        #Create a repository

touch myfile    #Create a file - you might put something in it
git add myfile  #Add myfile to be 'staged' for commit - it's not yet committed.
touch myfile2   #Create a file - you might put something in it
git add myfile2 #Add myfile2 to be 'staged' for commit - it's not yet committed.

git commit -m "A useful message"
                #This commits both files to the repository.

vi myfile       #Edit it again
...
git add myfile  #Stage the changes to myfile for commit.
git commit -m "Fixed bug X"
```

[6] `git` was initially written by Linus Torvalds, the creator of Linux, and he has quipped it is the second piece of software he has named after himself.

```
                #This adds the changes to myfile to the repository
...
git commit -am "Lots of changes to implement X"
                #This add all changed files and commits.

git checkout HEAD^
                # Go back at the previous version
git difftool
                #What changes have I made since the last commit?
```

You can find information as to how to look back at past versions in a git tutorial online.

2.4.8 Tips

A couple of useful tips for programming:

- Test your scripts frequently to check for bugs and other errors in your code. This will reduce the time you spend looking for bugs — checking a small section of a program as it is written is easier than trying to check the entire program in one go. The first draft of any program will contain bugs, no matter how experienced you are, knowing how to deal with them becomes easier with experience.

- Modular code helps understanding and also reduces errors. Splitting a problem (e.g. one of the exercises) up into a series of smaller problems is a good start. For each of these smaller problems you should be able to solve them by short code blocks (a few lines). Appropriate use of functions and classes is the key to short but effective programs.

- Planning your program before you start writing code is important. The discussion sections should help you think about this. You should have a framework of what your program needs to do. This helps you think about when to create a function or class in your code — it must exist before something can call it. This links in with modularisation of your code.

2.4.9 More Advanced Concepts

You may wish to delve into these when exploring Python in the later exercises, and knowledge of them is very helpful when working on larger Python projects.

Logging

When debugging, it is often helpful to log what is going on in your program rather than printing it out. Python has a logging library which makes this relatively easy with some tutorials. See the following sites:

- `https://docs.python.org/3/library/logging.html`
- `https://fangpenlin.com/posts/2012/08/26/good-logging-practice-in-python/`
- `http://docs.python-guide.org/en/latest/writing/logging/`

Unit Testing

When your problems are large, splitting them into small manageable chunks is a valuable process. If what they need to do is sufficiently well specified, then having a unit-testing system in place makes modifying code significantly less bug-prone. There's a Python framework to do this at `https://docs.python.org/3/library/unittest.html`.

Anaconda Python environments

Sometimes the computer you are running Python on is not fully under your control, and you would like the ability to install and manage extra packaged. The Anaconda environment is ideal for this.

- `http://conda.pydata.org/docs/intro.html`
- `http://conda.pydata.org/docs/test-drive.html`

2.5 A Linux Survival Guide

2.5.1 Booting into Linux

To boot the PWF Workstations into Linux do the following:

- Press Ctrl-Alt-Del. Press OK on the CS rules screen.
- Don't log in, but press "Shutdown". Select "Shutdown and restart" and press OK.
- The manufacturer's splash screen appears after a minute or two. When the OS Loader screen appears follow the instructions on the screen to select PWF Linux.
- After a few minutes, RedHat Linux will start to load and this takes a few minutes.
- When the Login screen appears, type your usual PWF user identifier and password.

2.5.2 Using the terminal

In this course guide we shall focus on writing and running code using the terminal, a command line interface. The terminal is a simple but powerful interface. Typed commands are used rather than 'point and click', that most of you are probably familiar with.

You open a terminal by clicking on the terminal icon in the task bar (left hand side of the workspace). It is supposed to look like a black window containing a white prompt at which commands will be typed.

With your terminal you can now run UNIX commands. For example if you type:

```
whoami
```

into the terminal and then hit the ENTER key, it should tell you the user-id you used to log in, or you can type:

```
pwd
```

and hit ENTER and it will tell you the directory[7] you are in. To get help on a file command some people use google, but a faster way can often be to use the `man` command (`man` is short for "manual" as in "hand book".

To find out what the `cd` (change directory) command can do, and what options you can give it, type:

```
man cd
```

Press PageUp and PageDown to navigate, and q to quit.

There is some useful information about using the command line, and some of the commands on the Ubuntu (the version of Unix on the PWF) website: `https://help.ubuntu.com/community/UsingTheTerminal` .

Care should be taken when using the `rm` (delete) command. As long as you don't use `rm -rf *`, which deletes all files in the current working directory and any directories contained within it, or `rm *` which deletes all files in the current directory. As long as you don't accidentally type these commands you can experiment to your heart's content.

2.5.3 Navigation using the terminal

Navigating your directories will be important if you want to know were all your scripts are, as well as any data files that you want to read from, and where any output files and images you create will go. As we have already seen the command `pwd` returns your current working directory, (i.e. the location where any files you create will be put). Putting all your work directly into your home directory is not advised and will confuse you. Creating a folder for all of the work for this course, and a separate directory within that for each exercise would be a good way to make it easy to find the file you are looking for.

To create a new directory you use the `mkdir directoryname` command (this is much faster than using the files program). Using the command line you can then change your working directory simply by using the command `cd directoryname` (then hit ENTER). You should now be in the `/home/username/directoryname` directory. The `cd` command allows you to move several levels up or down the directory hierarchy with only one use. To move towards the root directory[8] you use `../` as the directory name to move up one level. To find out what is in a directory you can use the `ls` command. This lists all files and directories that are inside the directory specified (or current directory if no name is given, see `man` page for more information).

The `cp` (copy file) and `mv` (move file) are 2 other very useful commands you should become familiar using.

2.5.4 Text Editors

From the terminal you can use a text editor to create and edit text files. Text editors do not have built in spell checkers, like word processors. This is useful when programming so variable names (among other things) are not renamed. There are several text editors that you could use (`vi`, `emacs`, `nano`, `gedit` are a few examples) to write your code in. Of these text editors vi

[7] In Unix a "directory" is equivalent to a "folder" in Windows. Note that the directories in Unix are separated by forward slashes, while backward slashes are used in Windows

[8] The root directory is the `/` directory

is recommended. Notes for an old UCS course can be found at `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/earlier/vi` . This has all the commands that you need to know to be able to use vi. There is also a course guide by the UCS for emacs, found at `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/earlier/Emacs`.

From the terminal, to open a text file with vi, you simply enter `vi filename` . For your Python scripts you should make sure the file extension is `.py` (your filename ends with `.py`). You are now ready to begin programming.

Editing

Once you have opened a file with your preferred text editor you can then start writing or editing a file. For most of these text editors you will note that you cannot use your mouse to move the cursor, this can only be done by the keyboard.

2.5.5 Tips

The following bits of advice may be useful when using the command line and text editors:

- Organise your files into directories and give them descriptive names. This will help you know which things to hand in.
- There is a lot of documentation for commands and the text editors mentioned. Use them to help you.
- Other commands that may be useful but have not been mentioned already include `cat`, `chmod`, `grep`, `find`.
- The keyboard interrupt for Linux is Ctrl+c.

3 Exercises

3.1 Exercise 1: A general Hückel solver

Concepts Assumed

- Simple Variables
- Inputting Data from the keyboard
- Loops
- Outputting Data (`print`)

Concepts Learnt

- Tuples and Lists
- Manipulating Lists
- `numpy` libraries and structures: vectors, matrices, diagonalizers.

From the IB Chemistry A Symmetry and Bonding course, and this year's A4 Theoretical Techniques (or alternatively the A6 Concepts in physical chemistry course), you should have some familiarity with the basics of Hückel Theory. This exercise will develop your theoretical knowledge and practical experience into a program to calculate energies of simple π-systems.

3.1.1 Specification

For an arbitrary molecule, specified by its connectivity between adjacent atoms, calculate and print the Hückel energies and degeneracies of its π-system. You may assume that the only relevant atoms are all of the same element, are sp^2 hybridized, and that adjacent atoms are all the same distance from each other.

3.1.2 Tasks

Determine the Hückel π-energies and degeneracies for:

1. A linear polyene with n carbons;
2. A cyclic polyene with n carbons;
3. The sp^2-hybridized Platonic solids (i.e. 3D structures with faces of a single regular polygon, with carbons at each vertex and only 3 edges per vertex): tetrahedron, cube, and dodecahedron. (*If you wish to suspend your chemical disbelief you might also try the octahedron and icosahedron*);
4. (*Optional*) Buckminsterfullerene.

3.1.3 Techniques

Numbers, Strings, Variables, Tuples and Lists

Simple numbers (like `int`, `float`) and strings (`str`) are known as *literals*, and can be thought of as everyday objects which you can create, assign to variables and use, but not change:

```
i = 7               # Create the value 7, and set i to have this value
i = "hippopotamus"  # Create the string "hippopotamus"
                    # and set i to this (losing its previous value 7)

7 = "hippopotamus"  # SyntaxError: can't assign to literal
                    # i.e. 7 is a literal not a variable, so we can't change its value.

i = 7
j = 8
k = i + j           # Looks up the values of variables i and j, evaluate them, add them,
                    # and put the result in k

v1 = "hippo"        # meaning horse in ancient Greek
v2 = "potamus"     # meaning river in ancient Greek

animal = v1 + v2    # Adds strings next to each other, so should have value "hippopotamus"
```

If you wish to manipulate groups of values, it would be a pain to have to assign them all to a variable, so there are two simple concepts to collect them. The `tuple` is an object which collects values together, but cannot be changed (like a literal). The `list` is an object which, like a `tuple` has a number of elements, but these can be changed. To access the elements of a list or tuple, square brackets are used with a zero-based integer index.

```
t = (1, 2, 3)       # A tuple with three elements
print(t[0])         # print the first element, 1

l = [4, 5, 6]       # A list with three elements
print(l[0])         # print the first element, 4

l[1] = "camel"      # change the second element of l
print(l)            # print the whole list to see

t[0] = 17           # error as you can't change a tuple

l[1] = t[2]         # Access the third element of t (6) and put it in the second position in l.
print(l)
```

An important feature of lists (and in general other objects you will start to create) is that two variables can contain the same list, so replacing an element of it might have strange effects.

```
l = [1, 2, 3]       # make a list and assign it l
m = l               # m now contains the same list (not a copy)
print(m)            # [1, 2, 3]

l[1]=7              # change an element of l
```

```
print(m)            # [1, 7, 3].  m has changed as well

m = [1, 2, 3]       # make a new list and assign it to m.
n = list(m)         # This creates a new list, but sets its elements to be the same as those of m.

m[2] = 7            # Replace m's third element with 7
print(n)            # [1, 2, 3].  n's elements haven't changed

l = [1,[2]]         # a list with a list in its second element
m = list(l)         # a new list with the same elements as l
l[0] = 7            # Change l's first element
print(m)            # As expected, m's first element hasn't changed
l[1][0] = 9         # Get l's 2nd element (a list), and set its first element to be 9
print(m)            # [1, [9]]
```

This last result may take you by surprise. Importantly, when we made `m` with elements which are the same as `l`, its second element was the **same** one-element list as in `l` (not a copy). We then changed the only element in that list to 9. Because `m`'s second element is the same list, when we printed it, we saw the change.

The `numpy` library

The `numpy` library provides very many common linear algebra functions in a convenient wrapper. The most important object in the `numpy` library is the `array`, which you can think of as a general container for sets of numbers, like a vector or a matrix. The following Python program should be sufficient to demonstrate how to manipulate arrays.

```
import numpy as np

v = np.ndarray(3)       # Make a 3d vector.  NB the elements are uninitialised
v = np.zeros(3)         # Make a 3d vector of zeros.
v = np.array([1, 2, 3]) # Make the 3d vector (1, 2, 3)

v *= 2                  # Scale the vector by factor 2
print(v)                # prints [2 4 6]

# Note that vectors are objects, so the assigning them doesn't make another copy:

w = np.array((1, 2, 3)) # You can initialise with a tuple as well as a list
x = w                   # x is the SAME vector
y = np.array(w)         # y is a copy.
w[0] = 0                # set the first element in w to be zero.
print(w)
print(x)                # x has changed as well
print(y)                # y hasn't changed.
```

Now vectors have been mastered, matrices are very simple. You can either use a 2D array or the `numpy.matrix` class. The `linalg` subpackage of `numpy` contains routines to find eigenvalues.

```
import numpy as np
```

```
M = np.ndarray((3, 3))      # A 3x3 matrix
MI = np.identity(4)         # A 4x4 identity matrix
print(MI)

Mz = np.zeros((4, 4))       # A 4x4 zero matrix

print(Mz.shape)             # Get the size of a matrix (a,b) means a x b
lc = Mz.shape[0]            # Number of columns in Mz
for i in range(lc):
    Mz[i, lc-i-1] = 1       # Max a matrix whose backwards diagonal elements are 1

M2 = Mz + MI                # Add matrices together
evals, evecs = np.linalg.eig(M2)  # get the eigenvalues and vectors

print(sorted(evals))        # print a sorted list of eigenvalues
```

For more information see the `numpy` documentation `http://docs.scipy.org/doc/numpy/reference/`.

3.1.4 Discussion

The specification given is quite short and relatively specific as to the task, though extremely unspecific as to exactly how to specify a molecule, and how the results are to be output. This is quite common in scientific applications, and you are therefore relatively free to choose whatever method is most convenient to get the data in and out. This is aided by looking in more detail at what more specific tasks the program will need to perform — this information is in the Tasks section. There are also some assumptions which should make the procedure simpler (though if you decided to write a very generic solution they should also be easy to implement).

The first thing to do is to sketch out how you would like the eventual program to work. This is often best done schematically on paper, but you could formalise more in pseudo-code if you know more specifics. The core calculation will require a Hückel matrix to be built, and diagonalized to give the eigenvalues, and so a key element is the construction of this matrix, which might need to be done differently for each task. In this case we need to work out some specifications for input which fit in with the Tasks. The poly-enes only require n to specify them, but the larger molecules will be more tricky. Once the matrix is constructed, it can be passed to the same routine for each Task, which can go on to produce the eigenvalues and degeneracies.

An important principle in software design is to avoid reinventing the wheel wherever possible. Quite often one of the steps you require will be common to some other process, or be well-used in some possibly quite distant field. In such cases, there will usually exist a software library of routines to perform these common tasks. The more benevolent amongst the community may even maintain these and distribute them for free. In general, if such a library exists (assuming you can find it), it is a good idea to use it. The writer will usually have spent considerable time designing and testing it to be error free and (hopefully) easy to use. Here a very convenient set of libraries are the `numpy` libraries, which are designed for numerical analysis, and in particular include a diagonalizer which will produce the eigenvalues of an arbitrary matrix. The Techniques section contains examples of how to use such a library, though in general the library's documentation should be sufficient.

The final challenge is to determine degeneracies. Here we should warn that in all computer implementations there are usually numerical round-off errors, and so one cannot just compare to see if two eigenvalues are identical to check degeneracy.

With these three stages in mind we can devise a pathway to writing the complete program. It's usually best to make this in sufficiently small steps such that you can test each part along the way. If such a step is sufficiently self-containable, it might be worth encapsulating it in a function. In particular, as soon as you start to copy and paste code or write multiple pieces of code doing the same thing, you should consider putting it in a function and calling the function.

Here's an example of a set of steps which might be sensible when developing the Hückel program:

1. Hard code a specific 3×3 matrix and print out its eigenvalues.
2. Write a function (e.g. get_evals) which takes a matrix (of any dimension) and returns a list containing its eigenvalues. Test it by passing it a specific matrix and printing out what it returns.
3. Write a function to make an $n \times n$ Hückel matrix for a length n poly-ene. Check the matrix looks right for $n = 4$. Pass it to get_evals and check that the returned eigenvalues are right. Now automate the test to check the returned eigenvalues are correct for $n = 100$.
4. Write a function to make an $n \times n$ Hückel matrix for a length n cyclic poly-ene. Check the eigenvalues for $n = 6$.
5. Write a function to work out the degeneracy of each eigenvalue in a list of eigenvalues, and print it out in a nice manner. Check the degeneracies for the $n = 10$ cyclic poly-ene.
6. Write a function to work out (or just return hard-coded) the Hückel matrix for the sp^2-hybridized Platonic solids. Check the degeneracies are appropriate given the symmetries of the solids.
7. Consider how you might interface these together in one program where the user specifies what system to calculate: direct user input, command-line arguments, or an input file are all possibilities.

3.1.5 Theoretical Background

Hückel Theory can give reasonable predictions of orbital energies and degeneracies for systems of sp^2 carbons. Its basic assumptions are that the molecular orbitals of a π-system can be formed from a linear combination of atomic p-orbitals. For a molecule of N atoms, the MOs are, $\phi_i = \sum_{j=1}^{N} c_j^{(i)} p_j$, where p_j is a p-orbital on atom j. The coefficients $c_j^{(i)}$ are obtained (via the Variational Theorem) as eigenvectors of the Hückel matrix, $H_{ij} = \langle p_i | \hat{H} | p_j \rangle$, and each has a corresponding eigenvalue $\epsilon^{(i)}$.

The Hückel matrix is formed from the assumptions

$$H_{ij} = \begin{cases} \alpha & \text{if } i = j, \\ \beta & \text{if } i \text{ is adjacent to } j, \\ 0 & \text{otherwise.} \end{cases}$$

You have previously used symmetry to simplify the problem $\boldsymbol{H}\boldsymbol{c}^{(i)} = \epsilon^{(i)}\boldsymbol{c}^{(i)}$ to manageable proportions, but this eigenvalue problem is so commonly encountered in linear algebra that there are sophisticated libraries which will solve it for matrices up to many thousands of columns.

To make the problem amenable to numerical implementation, we may set $\alpha = 0$ and $\beta = -1$. If actual values are known, the eigenvalues can be formed by scaling by β and shifting by α.

3.2 Exercise 2: Data processing and plotting

Concepts Assumed

- Simple Variables
- Loops
- `printing`
- `numpy` arrays

Concepts Learnt

- File input and parsing
- The `matplotlib` library
- `numpy` data interpolation

3.2.1 Specification

The program must take a directory pathname as an input. Within the directory are a set of Gaussian output files for a triatomic at different geometries (corresponding to the symmetric stretch and change in bond angle). From each file, the program should extract the geometry of the molecule and the energy at that geometry and store it. It should then plot the energy surface as a function of the two degrees of freedom, and determine the vibrational frequencies of both the symmetric stretch and the bending mode of the molecule.

3.2.2 Tasks

Two sets of output files are provided on Moodle — one for H_2O and another for H_2S. Your program should plot the energy surface and determine the vibrational frequencies when given either of the directories containing the files. An example surface is given in Fig. 3.1, though you should feel free to plot out the surface in any way that you think clear. From experience, you will want to set the limits on the energy to make the minimum look sufficiently clear.

Getting the output files

The output files required for both parts can be found on moodle. They are in separate compressed tar archives: H2Ooutfiles.tar.bz2 and H2Soutfiles.tar.bz2. To get the output files into a readable format you must:

1. Download the files to your machine.
2. Decompress the file. The files were compressed using bzip2[1]. Decompression can be done by `bunzip2 filename` or `bzip2 -d filename`.

[1] Please read the man page or documentation for more information about it

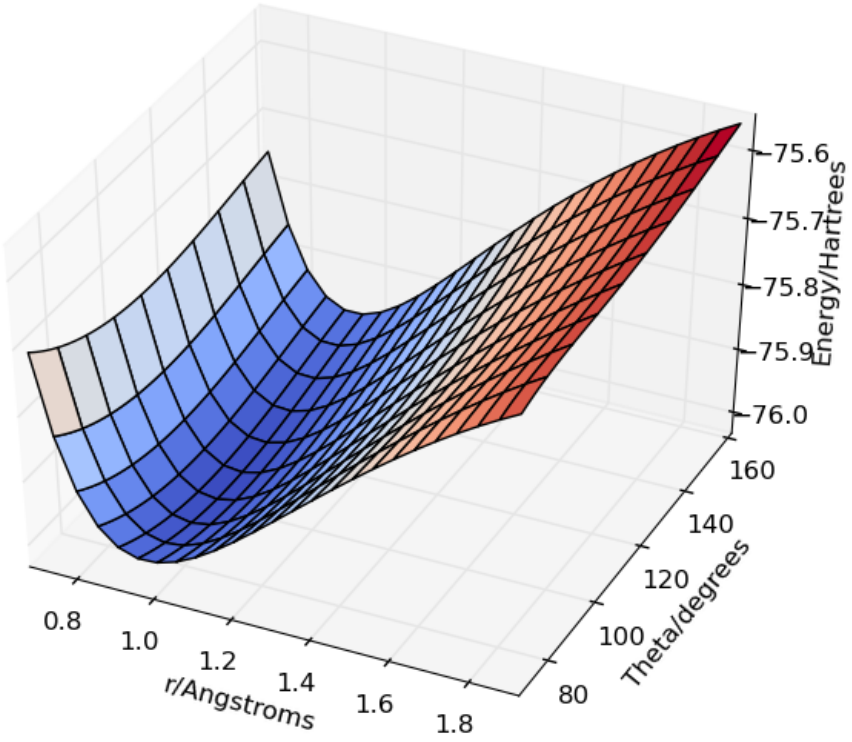

Figure 3.1: The Potential Energy Surface for H_2O.

3. You must now extract from the tar archive[2]. This is done with `tar -xvf filename`

4. You should have successfully extracted a copy of the files you need into a file. You can now read them using a Python script (recommended) or manually [3].

3.2.3 Techniques

File manipulation

Files are quite easy to open and close in Python, and you can read from them line-by-line.

```
f = open("myfile", "w")          # Open a file for writing
for i in range(10, 20):
    f.write(str(i) + "\n")       # Write a number followed by a newline
f.close()
```

[2]Documentation is useful here to understand the commands

[3]There are over 2000 text files for both H_2O and H_2S. It would probably take days to extract the information by hand. The documents all have the same format, so a simple parser can extract the required data in a minute or less.

```
f = open("myfile", "r")         # Open the same file for reading
total = 0
for line in f:
    value = int(line)
    total += value
print("The total is ", total)
f.close()
```

The `"w"` attribute will overwrite an existing file, so be careful using it.

Data Extraction

Parsing is the activity of taking some text and extracting information from it. Manipulating text in Python is also quite simple, and in this example, we can `split` strings and convert parts of them to numbers and use the result. We'll generate the data using a random number.

```
from random import randint

f = open("myfile", "w")          # Open a file for writing
for i in range(100):             # We'll write 100 lines
    s= "I saw "                  # Construct a string
    s += str(randint(1, 20))     # Pick a random number between 1 and 20
    switch = randint(0, 2)       # Use another random number to work out colour
    if switch is 0:
        s += " yellow lorries.\n"
    elif switch is 1:
        s += " red lorries.\n"
    else:
        s += " green cars.\n"
    f.write(s)                   # Write the string to the file

f.close()

# Now we can read in the file and count the number of different colours

f = open("myfile", "r")          # Open a file for writing
d = {}                           # We're going to include the information in a dictionary
for line in f:
    if 'lorries' in line:        # We only care about lorries
        l = line.split()         # this splits the line up by its spaces making a list.
        colour = l[-2]           # Extract the penultimate element
        number = int(l[-3])      # and the one before, turning it into an integer.
        if not colour in d:      # see if we've seen the colour before
            d[colour] = number   # if not, we create a new key in the dictionary
        else:
            d[colour] += number # if we have, we just add our count in
f.close()

for colour in d:                 # Go through the colours we found and print out the information
    print("Overall we saw", d[colour], colour, "lorries.")
```

Plotting

When confronted with large amounts of data, one of the most convenient ways to representing it is a graph. In times of yore, I used to paste data into Excel and get it to plot graphs. However, Python has some extremely powerful plotting libraries which avoid you having to do so much work. It is particularly useful in the `ipython` environment, but also very easy to use in scripts. `matplotlib` is part of the scipy suite and enables you to make professional looking plots.

```
from numpy import *          # This means we can use numpy functions without a prefix.
import matplotlib.pyplot as plt

x = arange(0, 2*pi, pi/10)  # Make an array of x points spaced pi/10 apart
y = sin(x)                   # This applies sin to each element of the array.
                             # NB it uses numpy.sin not math.sin

plt.plot(x, y)               # Plot using the list x as the abscissa and y as the ordinate.

y2 = cos(x)
                             # We can choose colours and markers etc..  This goes on the same axes
plt.plot(x, y2, color="red", marker="x")

plt.show()                   # If you've got X enabled this will work

plt.savefig("myplot.png")    # Or you can save to a file.
```

The `matplotlib` documentation is very extensive, and allows almost every conceivable type of plot. The example above is very rudimentary, and you would be advised to label axes etc.

Fitting

Fitting data to a model is an extremely common task for the scientist. Once again `numpy` makes this very easy.

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

lim = pi
x = arange(-lim, lim + .01, 0.1)    # Make an array of points spaced 0.1 apart
y = cos(x)                          # Let's take the cosine of this.

# We'll try to fit this cosine to a quartic polynomial:
# p[0] x**4 + p[1] * x**3 + p[2] * x**2 + p[3] * x + p[4]

p=polyfit(x, y, 4)
print(p)

xfit = arange(-lim, lim+.01, 0.01)  # Let's have a finer spaced array for plot
yfit=polyval(p, xfit)
                                    # Original data in red x's
```

```
plt.scatter(x, y, color="red", marker="x")
plt.plot(xfit, yfit)                # Plot a line for the fit
plt.show()                          # If you've got X enabled this will work
plt.savefig("myplot.png")           # Or you can save to a file.
```

In general you will probably want to fit a subsection of an array rather than the whole data set. This could be performed by making a new array and putting into it only values which are in the region you would like to fit. There are also more advanced curve fitting algorithms with error bars available if you are feeling adventurous.

3.2.4 Discussion

In this exercise we have been given a quite specific problem to solve. The specification tells you what you need to input but you are given a lot of freedom about the implementation of the solution. There is a lot of data which needs to be extracted from the Gaussian output files and stored in a variable to be used by the program, and the extraction of the information about the geometry and the energy is a task well suited to modularisation in a function. It's important to note that the two molecules to analyse are both triatomics, so there is no need to make a very general code to deal with more complicated systems. The data are also given in terms of X−H bond length and H−X−H angle and so you only need to deal with these two degrees of freedom.

Plotting data is an excellent way to check that it has been extracted correctly and that whatever source the data has come from has indeed produced a correct answer. `matplotlib` is an incredibly flexible library which will allow you to plot data in whatever way you think most appropriate.

To determine the normal mode frequencies, it will be necessary to determine the equilibrium geometry (the one with the lowest energy). This process should be automated as it needs to be applied to two different molecules.

Armed with the equilibrium geometry, one can calculate the vibrational frequencies in a number of ways:

- A crude finite difference estimate of each frequency.
- Fitting a curve through the data plotted along each normal mode and using the parameters of the curve.
- Determining the Hessian matrix at the equilibrium geometry and diagonalising it. The frequencies can be determined from the eigenvalues.

The middle option appears to have the best compromise of accuracy and effort.

3.2.5 Theoretical Background

Electronic Structure Calculations

For a given molecular geometry (i.e. co-ordinates of all the atoms in a molecule), the electronic energy of the molecule can be routinely calculated by many different software packages (e.g. HyperChem in Part IB practicals). Here a series of calculations have been run for you using the Gaussian 09 package (sparing you the logistics of running them yourself). Each calculation has been run a specific geometry of the triatomic H_2X, and the output saved to a file. As with many such pieces of

software, there is far more information output that you need, so you will need to locate and extract the relevant data.

We have chosen to use Density Functional Theory[4] which can be thought of as a black box technique taking in a geometry and spitting out an energy[5].

If you inspect a file (e.g. `H2O.r1.8theta92.0.out`), you can find a line detailing the calculated energy:

```
SCF Done:  E(RB3LYP) =  -76.1206432590     A.U. after   13 cycles
```

This indicates the the energy is -76.1206432590 Hartree.

Vibrational Frequencies

You have some experience of normal modes, being the vibrations of a molecule seen in IR and Raman spectroscopy. Given a potential energy surface (the electronic energy of a molecule as a function of its geometry), the equilibrium geometry will correspond to the lowest point on this surface. In this example we are considering just two degrees of freedom — the H−X bond length (both are set to be the same), r, and the H−X−H bond angle, θ — which means that the energies really can be plotted as a surface. Around this minimum (where $r = \bar{r}$ and $\theta = \bar{\theta}$), the energy increases, and can in general be fitted to the form:

$$E = E_0 + \frac{1}{2}k_r(r - \bar{r})^2 + \frac{1}{2}k_\theta(\theta - \bar{\theta})^2.$$

Following the A4 course (with some approximations), you can calculate the vibrational frequencies from k_r and k_θ,

$$\nu_1 = \frac{1}{2\pi}\sqrt{\frac{k_r}{\mu_1}} \qquad \text{and} \tag{3.1}$$

$$\nu_2 = \frac{1}{2\pi}\sqrt{\frac{k_\theta}{\bar{r}^2\mu_2}}. \tag{3.2}$$

where $\mu_1 \approx 2\,m_u$ and $\mu_2 \approx 0.5\,m_u$ given this coordinate system.

[4]For the purposes of this exercise you don't need to know any of the details of the calculation method, but they are given for completeness. The B3LYP functional was used as it is commonly considered to give very good energies for most simple molecules. A 6-31G** basis set provides an adequate description without significant computational time. For more information about the methods, see the C6 course.

[5]For more complicated molecules and extreme geometries of simple molecules this is not entirely true.

3.3 Exercise 3: Kinetics Simulation

Concepts Assumed

- `matplotlib` plotting
- Lists
- Loops
- Dictionaries

Concepts Learnt

- `numpy.loadtxt`
- Classes

3.3.1 Specification

For a given set of equilibria, specified by elementary reactions (with forward and backwards rate constants) between a set of substances, determine the time-evolution of the system from a given set of initial conditions.

3.3.2 Tasks

1. **Protein Folding Simulation**

   A two-domain protein (i.e. one which can be thought of as two separate proteins joined together) has been found to fold cooperatively (i.e. the foldedness of one domain affects the folding rates of the other). Here is a two-domain protein's folding pathway from Ref 1:

   Here R15 and R16 are the two different domains which can be both unfolded (D), R15 folded and R16 not (I), and both folded (N). Protein folding is commonly studied by adding a concentration of a denaturant (e.g. urea), which destabilises the folded protein and promotes unfolding, and so the rate constants become exponentially dependent on the concentration of denaturant,

$$k_f^{R15}([\text{urea}]) = k_f^{R15}(0\,\text{M})\, e^{-(1.68\,\text{M}^{-1})[\text{urea}]} \tag{3.3}$$

$$k_u^{R15}([\text{urea}]) = k_u^{R15}(0\,\text{M})\, e^{+(0.95\,\text{M}^{-1})[\text{urea}]} \tag{3.4}$$

$$k_f^{R16}([\text{urea}]) = k_f^{R16}(0\,\text{M})\, e^{-(1.72\,\text{M}^{-1})[\text{urea}]} \tag{3.5}$$

$$k_u^{R16}([\text{urea}]) = k_u^{R16}(0\,\text{M})\, e^{+(1.20\,\text{M}^{-1})[\text{urea}]} \tag{3.6}$$

   For varying concentrations of urea, the equilibrium concentrations of each species were found as in this figure. Show that the kinetic model fits these equilibrium data.

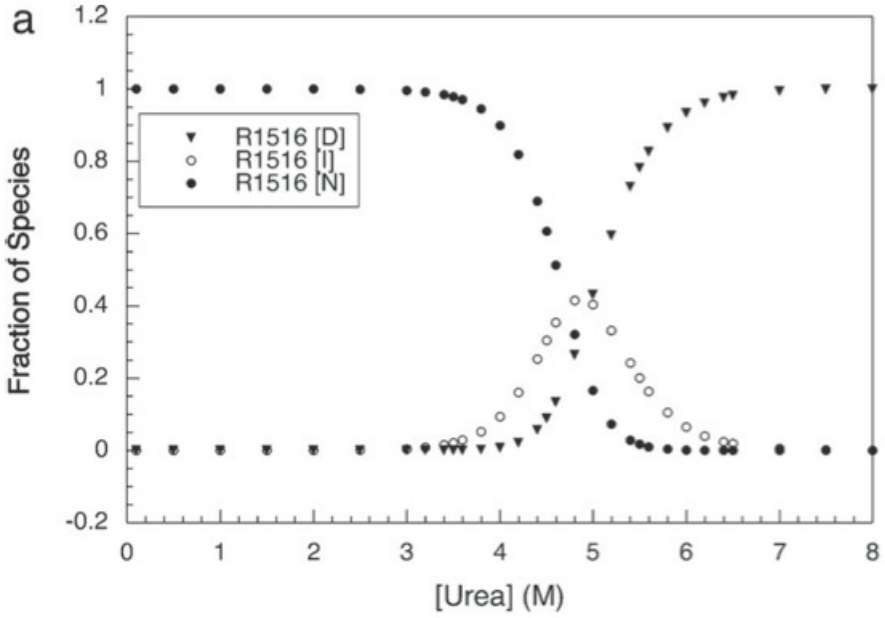

Figure 3.2: The measured fraction of each species at equilibrium for different denaturant concentrations from Ref 1.

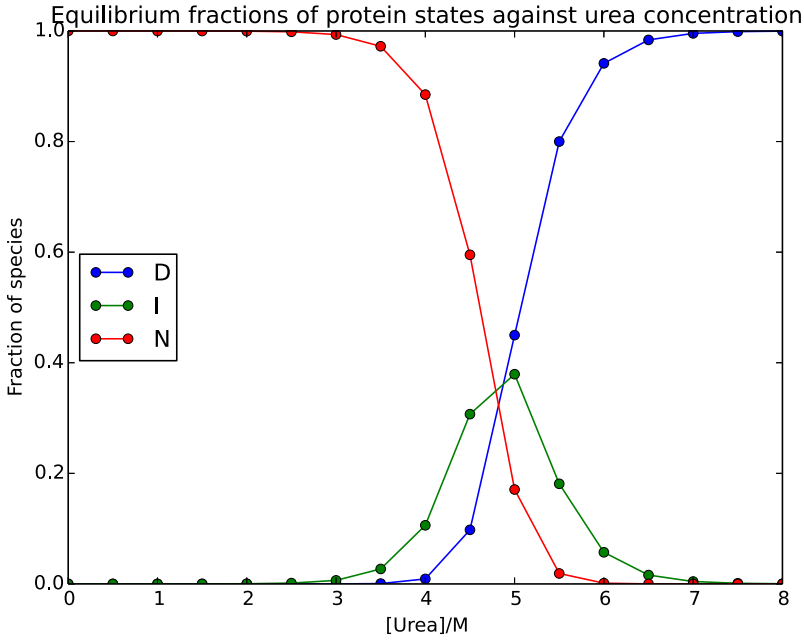

Figure 3.3: The results of a Python kinetics simulation. These are probably too coarse-grained.

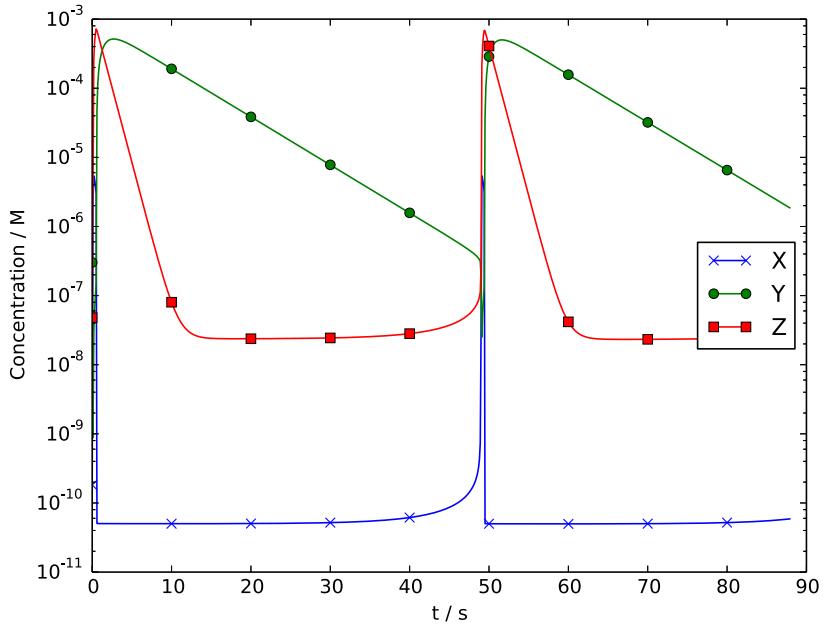

Figure 3.4: Concentrations of the relevant species in the Oregonator over time. The vertical axis is logarithmic!

2. The Oregonator

Inspired by the delightful oscillatory character of the Belousov-Zhabotinsky reaction, the Oregonator[2] was developed as a simple theoretical model of the more complicated BZ reaction which still shows oscillatory behaviour.

$$\begin{aligned}
\mathrm{A} + \mathrm{Y} &\xrightarrow{k_1} \mathrm{X} + \mathrm{P} & k_1 &= 1.34\,\mathrm{M^{-1}s^{-1}} \\
\mathrm{X} + \mathrm{Y} &\xrightarrow{k_2} \mathrm{P} & k_2 &= 1.6 \times 10^9\,\mathrm{M^{-1}s^{-1}} \\
\mathrm{B} + \mathrm{X} &\xrightarrow{k_3} 2\,\mathrm{X} + \mathrm{Z} & k_3 &= 8 \times 10^3\,\mathrm{M^{-1}s^{-1}} \\
2\,\mathrm{X} &\xrightarrow{k_4} \mathrm{Q} & k_4 &= 4 \times 10^7\,\mathrm{M^{-1}s^{-1}} \\
\mathrm{Z} &\xrightarrow{k_5} \mathrm{Y} & k_5 &= 1\,\mathrm{s^{-1}}
\end{aligned}$$

The state is highly dependent upon the initial concentrations, and Figure 3.4 shows an example with the following initial state[6]: $[\mathrm{A}] = 0.06\,\mathrm{M}, [\mathrm{B}] = 0.06\,\mathrm{M}, [\mathrm{P}] = 0\,\mathrm{M}, [\mathrm{Q}] = 0\,\mathrm{M}, [\mathrm{X}] = 10^{-9.8}\,\mathrm{M}, [\mathrm{Y}] = 10^{-6.52}\,\mathrm{M}, [\mathrm{Z}] = 10^{-7.32}\,\mathrm{M}$.

A stable integration time-step for this system proved to be $10^{-6}\,\mathrm{s}$. If your system is behaving oddly, you might use a smaller time-step.

a) Reproduce the dynamic spikes seen in Figure 3.4 by integrating the Oregonator's kinetic equations.

[6] Starting from different conditions may produce strange results

b) One interesting effect of this reaction is that if one of the species is coloured, a well-mixed reaction will produce timed colour changes. If the reactants are instead allowed to diffuse (say along a 1-d line), these colour changes will manifest themselves as spatially-resolved patterns which move over time.

Sketch a design of a program which will allow a spatially-resolved (say in 1- or higher-dimensions) simulation. [If time permits, you might even write the code and plot out the results.]

3.3.3 Techniques

Data Management

As your programs become bigger you will need to handle increasing amounts of data, and the way the data is stored can have a great effect on the speed and readability of a program. Let's consider the simple example of a St Cedric's College Wine Cellar, which the Wine Steward is deciding to computerise. Typically his ledgers record the name of the wine, the country of origin, its year, and the number of bottles left. One way of organising such information is in the form of a series of lists

```
Names = ["Chateau d'Yquem", "Le Cigare Volant", "Chateau Chasse-Spleen", "Goats do Roam in
    Villages", "Chateau d'Yquem"]
Countries = ["France", "USA", "France", "South Africa", "France"]
Years = [1975, 2008, 2005, 2009, 1990]
Bottles = [1, 25, 96, 36, 48]
```

Each wine has an index, and its data can be found in that position in each list. This is a somewhat unsatisfactory way to store the information — if a wine were to be removed it would have to be removed from each list, and unless great care is taken the lists might become inconsistent and the data mangled.

It would be better to group the data together such that each wine's data is in the same place. This could be a dictionary, with a key for each attribute[7]. The wine list would then be a list of dictionaries:

```
Wines = [ { "Name":"Chateau d'Yquem", "Country":"France", "Year":1975, "Bottles":1},
          { "Name":"Le Cigare Volant", "Country":"USA", "Year":2008, "Bottles":25},
          ...
        ]
```

This approach allows you to 'move' wines about with ease. You can swap elements of the list, and even move wines in bulk:

```
Wines[1], Wines[2] = Wines[2], Wines[1] #This works because the tuple on the right is evaluated
                                        #completely and then the individual assignments made.
MainCellar += OldCellar   #Append the wines in the OldCellar list to the MainCellar list
```

[7] A dictionary is significantly better than a simple list with, say, Name in position 0, Year in position 2 etc., for the simple reason that when a future self[8] looks at the code, `d["Year"]` is far more readable than `d[2]`.

[8] For more information see `http://xkcd.com/1421`.

This situation appears a lot cleaner, but it's still possible for there to be inconsistencies — there is no reason for each dictionary to have the same types of information. This could be needed when there are functions which manipulate the data, e.g.

```
def get_short_name(d):
    """Take a wine dictionary object and return a short name for it."""
    return d["Name"] + " " + str(d["Year"])

#Print a simple wine list
for w in Wines:
    print(get_short_name(w), ":", w["Bottles"], "bottles")
```

Functions like this will only work if there is (some of) the same data for each wine. If such uniformity is required, it might be worth creating a `class`.

Classes

When there is a need to manipulate data that is grouped together, classes are the easiest way to do this. A class stores both data, and the functions which operate on the data in the same place. Such functions are known as *methods*. A method is just a function belonging to a class, and will usually take `self` as its first argument, which is an *instance* of the class (like each dictionary above). There are a number of special methods usually pre- and post-fixed with __. One such is `__init__` which is used to initialise an instance of the class. It is convenient to think of instances of classes as objects (like `list`, `int`, `string` etc.).

```
class Wine:
    """Information about wine.

        Variables:
            name:       string  The name of the wine
            year:       int     The year of the wine or None for NV
            country:    string  The country of origin
            bottles:    int     The number of bottles

        Methods:
            get_short_name(self)    Return a string with a short name for the wine.
            is_ready(self)          Determine if a wine is ready to drink
    """
    def __init__(self, name, year, country, bottles):
        """Create a wine record."""
        self.name = name
        self.year = year
        self.country = country
        self.bottles = bottles

    def get_short_name(self):
        """Return a string with a short name for the wine. """
        s = self.name+" "
        if self.year:
            s += str(self.year)
        else
```

```
            s += "NV"
        return s

    def is_ready(self):
        """Return a Boolean indicating if the wine is ready to drink.

            This uses an extremely primitive algorithm which needs to be improved.
        """
        if self.country == "France":
            return self.year < 2005    # NV are True as None < 2005
        else:
            return True

Cellar = [  Wine("Chateau d'Yquem", "France", 1975, 1),
            Wine("Le Cigare Volant", "USA", 2008, 25),
            Wine("Chateau Chasse-Spleen", "France", 2005, 96),
            Wine("Goats do Roam in Villages", "South Africa", 2008, 36),
            Wine("Chateau d'Yquem", "France", 1990, 48)]

print("Wines for drinking now:")
for w in Cellar:
    if w.is_ready():
        print(w.get_short_name(), ":", w.Bottles, "bottles")
```

Importantly, classes allow you to keep the data and the code together logically, so you can manipulate the data by calling a method of the class.

Loading and Saving data

Quite often you will write a program which generates some data. You can of course use the data on the fly for a plot, but it's often much better to save any data you generate to a file (as well) just in case. Reading in such data is easily automated with `numpy.loadtxt`[9].

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

#First generate some data
x = np.arange(-np.pi, np.pi, 0.1)
y = np.sin(x)
z = np.cos(x)

#Now write it to a file
with open("datafile.dat", "w") as f:  # This opens the file and creates a 'context'
                                      # (i.e. indented section) and will close
                                      # the file automatically when the context ends.
    f.write("# x \t y \t z \n") #Write a header line so we know what the file's for.
    for xyz in zip(x, y, z):    #zip makes a tuple (x[i],y[i],[z[i]) for each element in the list.
                                #The following is an example of a perhaps too cryptic statement
```

[9] I would recommend using `loadtxt` and `savetxt` which manipulate text files rather than `load` and `save` which use binary files which are not human-readable, except when reading in the data takes considerable time. Having human-readable formats has been shown to make life considerably easier when debugging.

```
        f.write("\t".join([str(n) for n in xyz])+ "\n")
                                #Turn each data item into a string, join them with tabs,
                                #terminate with a newline and write to the file.

#Now get numpy to read it in
d = np.loadtxt("datafile.dat", unpack=True) # The True transposes the array, so that the first
                                            # index can be used to extract the different columns.

nx, ny, nz = d
plt.plot(nx, ny)
plt.plot(nx, nz)
plt.show()
```

Another option for saving data is to use the `pickle` library which can save nearly any type of object into a file.

```
import pickle
# We'll carry on from the context of the final wine example above with the Cellar object
with open("MyCellar", "w") as f:
    pickle.dump(f, Cellar)

with open("MyCellar", "r") as f:
    newCellar=pickle.load(f)

for w in newCellar:
    print(w.get_short_name())
```

Pickled data can only easily be read in by Python however, unlike the tab separated text above.

Optimizing Code

In general if a program runs sufficiently quickly then there is little point in optimizing it. However, in this exercise you will probably end up using a number of nested loops, and this can slow things up considerably.

Consider the example

$$S = \sum_i \sum_j a_i b_j \tag{3.7}$$

$$= \sum_i a_i \sum_j b_j \tag{3.8}$$

$$= (\sum_i a_i)(\sum_j b_j) \tag{3.9}$$

These three ways of writing this could corespond to the three methods below

```
import time
a = list(range(1000))
b = list(range(20000))
S = 0.0
t1 = time.time()
```

```
for i in range(len(a)):
    for j in range(len(b)):
        S += a[i] * b[j]
t2 = time.time()
print("S=",S)
print("Method 1 took ",t2-t1," seconds.")
S = 0.0
t1 = time.time()
for i in range(len(a)):
    bsum = 0.0
    for j in range(len(b)):
        bsum += b[j]
    S += a[i]*bsum
t2 = time.time()
print("S=",S)
print("Method 2 took ",t2-t1," seconds.")
S = 0.0
t1 = time.time()
asum=0.0
for i in range(len(a)):
    asum += a[i]
bsum = 0.0
for j in range(len(b)):
    bsum += b[j]
S = asum * bsum
t2 = time.time()
print("S=",S)
print("Method 3 took ",t2-t1," seconds.")
```

Hopefully Method 3 was faster than Methods 1 and 2. This example shows that, in many cases, you can make a code significantly faster by rearranging loops.

3.3.4 Discussion

The two tasks are very similar in nature, one looking at the long-time limit of evolving the kinetic equations, and the other looking at the time evolution itself. It would therefore be natural to have some form of function which takes a time-step. This function will be called repeatedly and the information output either at convergence, or after certain intervals.

The challenge in this task is to manage that data which the time-stepper uses: What are its inputs and outputs? How can they be most efficiently arranged? Note that the Oregonator task requires something of the order of a million time-steps, and the spatial extension probably considerably more.

We might consider the two types of data present: a list of the elementary reactions and, a concentration for each of the substances. The reactions are fixed at the beginning, whereas the concentrations change at each time-step. These data are however linked together, so it might be worth including them, and the functions that manipulate them in a single object.

Looking at the final part of the Oregonator the case for objectifying becomes even clearer as a canonical way to simulate a spatially extended system is to break it into cells, each of which can performs its own internal time evolution, as well as interact with its neighbours by diffusion.

We can imageine the concept of a Cell class which contains a list to store the reactions and some way of storing the concentrations. For both tasks here, one would probably just have a single "Cell" object (though the extension to a spatially resolves system might have a Cell for each position in space). Each Cell would have a method which takes a step forward in time within the Cell, and potentially another function which takes two next-door Cells and simulates diffusion between them. It is best to decide to define how it interacts with the outside world (i.e. the user), before worrying about the innards of how the data is stored. The specification is typically vague as to how to get data in and out, but the example tasks involve a list of reactions with different numbers of reactants and products. Clearly for each reaction you need to know the reactants and products, and for convenience it's probably worth referring to them by their names as strings, rather than giving each species a unique index number which would need looking up. Tempting though it might be to create a Species class to hold the information about concentrations of a given species, tis is probably a step too far as each class will just be a wrapper for the name and concentration of the species.

You might also consider the merits of specifying reactions as `"A+B+C+C->D+E"` against `[["A","B","C","C"],["D","E"]]` for example.

Looking more closely at the guts of the algorithm, the integration for each time-step requires you to determine a gradient, i.e. the change in the individual concentrations, at each time-step. Because each species might be used in multiple reactions, you cannot add the gradient directly to the concentrations, but need to form it separately first, and then take the step later. If you are feeling adventurous and wish to implement a more advanced integrator, this requires more data to be stored, and you may wish to consider this in your design of objects from the beginning.

You may also need to consider what it means for the integration to have converged. For small time-steps, the changes in concentrations will themselves be small, so this needs to be taken into account.

Lastly, the Oregonator takes a large number of timesteps, so rather than waiting for many hours for a calculation to complete, you will want to both check that it's doing exactly what you expect, and probably optimize the code by thinking about the loop structures.

3.3.5 Theoretical Background

Simple kinetics, as covered in the IA Chemistry course, can be formulated in terms of elementary reactions, in general as expressions like

$$\nu_{\mathrm{A}}\mathrm{A} + \nu_{\mathrm{B}}\mathrm{B} \xrightarrow{k} \nu_{\mathrm{P}}\mathrm{P} + \nu_{\mathrm{Q}}\mathrm{Q}, \tag{3.10}$$

for which there are elementary rate expressions like

$$\frac{\mathrm{d}[\mathrm{A}]}{\mathrm{d}t} = -\nu_A k [\mathrm{A}]^{\nu_A}[\mathrm{B}]^{\nu_B}, \qquad \text{and} \tag{3.11}$$

$$\frac{\mathrm{d}[\mathrm{Q}]}{\mathrm{d}t} = \nu_Q k [\mathrm{A}]^{\nu_A}[\mathrm{B}]^{\nu_B}. \tag{3.12}$$

When there are many such elementary reactions, the rate equations become coupled and are usually not possible to solve analytically. For such systems, we can rely on a numerical integration, which effectively splits time up into very small time-steps, Δt and propagates equations like

$$[\mathrm{A}](t+\Delta t) = [\mathrm{A}](t) + \Delta t \left.\frac{\mathrm{d}[\mathrm{A}]}{\mathrm{d}t}\right|_t. \tag{3.13}$$

This is the simplest form of integrator, and is by no means the most efficient; more sophisticated ones such as Runge–Kutta will allow for larger time-steps, but usually require more information to be stored.

3.4 Exercise 4: Exploring Potential Energy surfaces

Concepts Assumed

- functions
- loop control
- `numpy` arrays

Concepts Learnt

- Classes, Modularisation and objects.
- Operator overloading.
- Interfaces.

3.4.1 Specification

For a system of n identical particles, with interactions defined by an arbitrary pairwise interaction potential $u(r)$, determine the lowest energy configuration of the n-particle system. You should output a table of inter-particle distances and an XYZ format file.

XYZ format

The XYZ format is commonly used for molecular visualisation and is an ASCII text file with the following

```
NumberOfAtoms
A comment line
AtomicSymbol1  X1 Y1 Z1
...
AtomicSymbolN  XN YN ZN
```

where `NumberOfAtoms` is an integer, `AtomicSymbol`n is the atomic symbol of atom n and `X`n,`Y`n,`Y`n, are the Cartesian coordinates of atom n. A sample XYZ file for water is:

```
3
Water geometry from the CCCBDB
O   0.0000  0.0000  0.1173
H   0.0000  0.7572  -0.4692
H   0.0000  -0.7572 -0.4692
```

If you wish to visualise XYZ files, VMD[10] can be used. This is installed on the PWF. To view a molecule from a `.xyz` file you use `vmd filename` in the terminal.

[10] If you are using an ssh client, you can run VMD remotely if X is enabled

3.4.2 Tasks

1. Determine the lowest-energy conformation of the cluster of 7 particles using the Lennard-Jones potential.

2. Determine the lowest-energy conformation of the cluster of 7 particles using the Morse potential with $r_e/\sigma = 1$ and $r_e/\sigma = 2$.

3.4.3 Techniques

Classes and Object Orientation

When manipulating coordinates, vectors and other similar objects it is very sensible to write them as classes. The vector will be an instance of the class passed around and manipulated as an object. This is an example of a common but fairly modern paradigm known as *Object Oriented Programming*, where the data inside the class is primarily manipulated by the class's methods. Here is a simple example for a 2d vector.

```
import math

# Define the class.
class vec2d:
    """ A 2D cartesian vector."""

    # The __init__ method initializes an instance of the class (i.e. a vec2d object).
    # The method is passed the vec2d to initialize as 'self'
    # You can initialize it with a list of two co-ordinates but if you leave
    # them out it will be initialized to (0, 0).  The "= (0, 0)" indicates a default parameter
        value
    def __init__(self, lst = (0, 0)):
        """Initialize vector.

            lst is a tuple or list with two elements (default (0, 0)).
        """

            # Here we create a member variable r and initialize it to a list.
        self.r= list(lst)    # we take a copy of the list.

    # length is a method which acts on the data in the object (which is called self)
    def length(self):
        """Returns the length of the vector."""
        return math.sqrt(self.r[0]*self.r[0] + self.r[1]*self.r[1])


v0 = vec2d()        # Create a vec2d with the default (0, 0) value
print(v0.length())  # Call its length method.

v1 = vec2d((3, 4))   # Create a vec2d initialized with (3, 4).
print(v1.length())
```

One of the tenets of object orientation is to **only** use methods of the class to manipulate data inside the class. In that way, the internal details of how the methods are implemented (and even how the data are stored) are hidden from the user of the class, but the methods retain the same calling structure. The advantage of this is that if the internal details of the class change, any code which uses it should still function without change. This is called *data encapsulation*.

Operator Overloading

One useful feature of classes is that in addition to creating new methods, we can also define what will happen when we use the in-built operators from Python (e.g. `+, -, /, **=` etc.). This ability is called *operator overloading*, and is simply implemented as an additional method.

```
import math

#Define the class.
class vec2d:
    """ A 2D cartesian vector."""

    # The __init__ method initializes an instance of the class (i.e. a vec2d object).
    # The method is passed the vec2d to initialize as 'self'
    # You can initialize it with a list of two co-ordinates but if you leave
    # them out it will be initialized to (0, 0).  The "= (0, 0)" indicates a default parameter
        value
    def __init__(self, lst = (0, 0)):
        """Initialize vector.

            lst is a tuple or list with two elements (default (0, 0)).
        """

            # Here we create a member variable r and initialize it to a list.
        self.r=list(lst)    # we take a copy of the list.

    # length is a method which acts on the data in the object (which is called self)
    def length(self):
        """Returns the length of the vector."""
        return math.sqrt(self.r[0]*self.r[0] + self.r[1]*self.r[1])

    def __add__(self, other):
        """Add self to other and return the result"""
        ret=vec2d()
        for i in range(2):
            ret.r[i]= self.r[i] + other.r[i]
        return ret

v0 = vec2d((1, 1))
v1 = vec2d((2, 3))
v2 = v0 + v1        # Add these together which should produce [3,4]

print(v2.r)         # This is a naughty statement which violates data encapsulation.
                    # It is however convenient for this example.
```

Inheritance

We would ideally like our `vec2d` object to act just like a vector, with addition and subtraction. We could implement these features yourself (see Operator Overloading), but it turns out they have already been implemented for us, and the the `numpy.ndarray` has all of these features. Instead of making a method which calls each one in turn, we will use an extremely useful feature of classes, called *inheritance*. Inheritance allows a class to extend another class (called a *base class*[11]) by automatically calling the base class's method if one has not been defined for this class. In this example the inheritance is a little complicated as the `ndarray` is (unusually) not initialized in `__init__`, but in a method `__new__`. This is a special *class method* as does not take an instance of the class (which would be passed in `__self__`), but instead takes the class itself (passed in as `cls`) and uses it to create and return an instance of the class.

```
import numpy as np

class vec2d(np.ndarray):
    """ A 2D cartesian vector."""
    def __new__(cls, lst = (0, 0)):
        """Create a new instance of the vec2d by calling the ndarray __new__."""
        # First turn the lst into an array
        ar = np.array(lst, dtype=float)
        # The cls here is used to tell ndarray to make a vec2d object.
        # The data passed to the parameter buffer (i.e. ar) is used to initialize the array.
        x=np.ndarray.__new__(cls, shape=(2,), dtype=float, buffer=ar)
        # Why do you not just do x=np.array(lst)?, they cry.
        # That would create a numpy ndarray object, rather than a vec2d object

        return x
    def length(self):
        """Returns the length of the vector."""
        return np.sqrt(self*self)

v0=vec2d((1, 3))
print(v0)
v1=vec2d((3, 4))
print(v1)
print(v1 - v0)
```

Interfaces and Functions

With the advent of inheritance, it is quite easy to create objects all of which have the same methods which take the same parameters, but act differently (as they act on different types of objects). The specification of such a set of functions is known as an *interface*, and is an important part of object oriented programming, as it enables one class to be used as a drop in replacement for another without any changes in the way the code manipulates it. In fact it is possible to have routines which manipulate objects of very different types, but which all *expose* the same interface, without

[11] In fact it is possible to have multiple base classes, but such a situation is beyond this simple guide.

ever needing to know about the innards of the objects. Lastly, it is worth noting that it is possible to pass functions as arguments.

```
# Let's make some functions with an interface we'll call f, which takes two numbers and returns a
    single one
def fadd(a, b):
    return a + b

def fsub(a, b):
    return a - b

def digest_lists(lista, listb, f):
    """Take two lists and apply f to each corresponding pair of elements.

    lista, listb are lists of numbers
    f is a function which takes two numbers and resturns a number.
    """
    return [ f(a, b) for a, b in zip(lista, listb) ]

a = range(10)
b = list(reversed(a))

print(digest_lists(a, b, fadd))
print(digest_lists(a, b, fsub))
```

3.4.4 Discussion

The two tasks appear to be basically the same, but use a different potential. This means you should structure your program to be able to specify a general potential. The remainder of the program looks suspiciously similar to the previous exercise, as it involves calculating a gradient and taking a step in that direction. The gritty details of calculating the gradient involve more maths than previously, but essentially require you to manipulate a system of co-ordinates of particles to generate energies. This makes a System of n-particles a tempting candidate for a class.

You should consider how you might avoid repeating code, and allow for System objects to take different potentials without needing to duplicate any other functionality. Two possibilities spring to mind: a System could be a class with a specific interface for calculating potentials, inheriting most other methods from another class; or a System might have a function as a parameter passed into it at initialization.

3.4.5 Theoretical Background

Inter-particle potentials

A pairwise interaction potential is the simplest way of describing the interaction between particles (which could be rare gas atoms or even more complicated molecules). It is specified as a function of the distance between the particles, r. Here is, for example, the Lennard–Jones potential,

$$u_{LJ}(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right).$$

The potential is a function of r, but has two additional parameters, ϵ and σ, which have units of energy and distance respectively. You should use these as your units of energy (i.e. you might quote an energy as 15.9ϵ or a distance as $\sqrt{2}\sigma$.

In general, a system will consist of more than two particles, and so the total potential energy can be made up from the sum of the individual pairwise interactions:

$$U(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} u(|\boldsymbol{r}_i - \boldsymbol{r}_j|),$$

where $\boldsymbol{r}_i$ is the (vector) position of particle i.

Another potential that is commonly used is the Morse potential, given by

$$u_M(r) = D_e \left(1 - e^{-\frac{r-r_e}{\sigma}}\right)^2,$$

which has energy unit D_e and length σ. There is an additional parameter here, r_e/σ which dictates the strength of the potential.

While the energy U is a function of all the co-ordinates, the gradient $\frac{\partial U}{\partial \boldsymbol{r}_i}$ is a vector with $3n$ elements (3 co-ordinates for each particle). Importantly for optimising, this simple form of potential makes the gradient very easy to calculate. Denoting $\boldsymbol{r}_{ij} = \boldsymbol{r}_j - \boldsymbol{r}_i$, the (vector) derivative with respect to the coordinates of particle i is

$$\frac{\partial U}{\partial \boldsymbol{r}_i} = \sum_{j \neq i}^{n} \left.\frac{\partial u}{\partial r}\right|_{r=|\boldsymbol{r}_{ij}|} \frac{\boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|}.$$

An alternative to coding up the derivative is to get the computer to do the work for you using the potential function you have already coded up. For, e.g. co-ordinate $(\boldsymbol{r}_i)_x$, the gradient component can be calculated by a finite difference:

$$\left(\frac{\partial U}{\partial \boldsymbol{r}_i}\right)_x \approx \frac{U(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_i + \boldsymbol{\delta}, \ldots, \boldsymbol{r}_n) - U(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_i - \boldsymbol{\delta}, \ldots, \boldsymbol{r}_n)}{2|\boldsymbol{\delta}|},$$

where $\boldsymbol{\delta}$ is a very small displacement in the x-direction. This can be done for all $3n$ coordinates, avoiding the need for a separately coded gradient of the potential (which might be buggy and inconsistent with the potential).

Optimisation

The field of optimisation is a deep and fascinating one, and is covered in the Mathematical Methods course this year and in more depth in the M4 Energy landscapes and soft materials section in Part III. The highlights of these courses are that:

1. There are many potential local minima for a complex surface,
2. A simple way to head towards a minimum is to take a small step downhill, i.e. $\Delta \boldsymbol{r}_i = -\lambda \frac{\partial U}{\partial \boldsymbol{r}_i}$, where λ is a small number controlling the step size.

When the optimisation has found a minimum, the gradient will be (close to) zero and the optimisation will have converged.

Bibliography

[1] S. Batey and J. Clarke, Proc. Natl. Acad. Sci. USA, 2006, **103**, 18113–18118.

[2] R. J. Field and R. M. Noyes, J. Chem. Phys., 1974, **60**, 1877–1884.