THE UNIVERSITY of York

Third Year Theoretical Physics
Advanced Computational Physics

Version 1.15 — April 2018 — Phil Hasnip and Yvette Hancock

Contents

Module overview

Aims

The aims of this project are:

- To investigate the mutual solubility of two materials with compatible crystal structures using an appropriate computational model.
- To use appropriate computer programming languages and tools to set up, develop, analyse and visualise the results of computational simulations.
- To make appropriate use of program development and collaborative software 'good practice' and tools.
- To communicate simulation methodology and results effectively to a professional physics audience.

Learning outcomes

On completing this project you should have:

- Become familiar with the application of computational models to real materials problems.
- Experienced working effectively in a group, utilising collaborative software tools.
- Developed your programming skills further.
- Written a short report on the work that is suitable for a technical audience.

Assessment

At the end of the project the final mark will be assigned based on the following components:

- 30% Lab book and documentation
- 30% Software quality (see 10)
- 40% Write-up

Group Project

Introduction

This script provides the physical problem to be addressed, an outline of background material and suggested activities to be carried out; it does not provide you with a detailed set of instructions of what to do. It is for you, within your groups, to discuss and plan how to proceed each week. There are many strands to this project, you will need to organise yourselves as you see fit for the aims.

This is a 5 credit component, so you should expect to spend around 50 hours on it. Only 8 of those hours are in lab sessions, so use the sessions wisely! There are three lab sessions set aside for the computational work, and one lab session for the preparation of a short paper summarising the work.

The single most important skill in team-working is communication (listening to others, as well as communicating your own ideas clearly). To this end, it is suggested that your group has a short meeting at least once a week to review progress, plan and distribute tasks amongst the members of the group. The tasks have been designed to be largely independent of each other, so you should not generally be dependent on other team members finishing another task in order to finish your own; however your group should generate a cohesive software solution, so you must find a way of working together so that the tasks fit together in the final package. Note that this does not necessarily mean a single computer program must accomplish all tasks, though this may be the case.

Lab books

You must keep, as usual, a proper record of your work in a laboratory notebook – however in order to facilitate the group working, this notebook will take the form of a shared online document using e-log. Each of you should keep this document up to date with the work you have done. It should also include the plan which has been agreed each week.

This lab script

Because this is an open-ended research project, this document is not a conventional lab script which tells you exactly what to do; rather, it guides you through the background and opening parts of the project, suggests some possible avenues of research, and then gives a selection of reference information you may (or may not) find helpful. It is not recommended that you to read this cover-to-cover!

This document is organised as follows: In section 3.1, the background to the project is introduced, with outlines of the underlying theories and their context. Section 3 contains the tasks your group is expected to undertake, with suggestions for further open-ended tasks. Section 11 contains a description of the procedures you should follow in the computational laboratory, and an outline of how the project will be assessed.

Summary

- Talk to each other!
  Plan what you're going to do and who's going to do it
- Be prepared and spend your time wisely
  Expect to spend about 50 hours each on this project (including write-up). Think about how you can get the most out of the 4 lab sessions.
- Keep your Group lab book up-to-date
- Don't read the whole lab script at once
  Look at what you need to do, and read the parts which are useful.

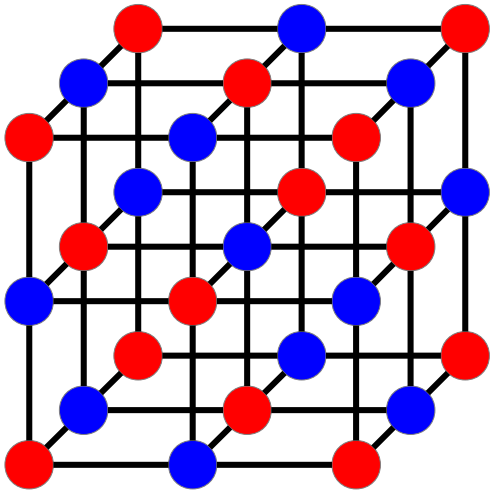

Figure 1: *Illustration of the rocksalt crystal structure.*

The project

Background

Most electronic devices are constructed from layers of different materials, each material having been selected because of a particular set of properties. Unfortunately the scale of these devices has now decreased so far that it is no longer possible to ignore the atomic and quantum properties of the materials. One common problem is that when a material is grown on top of another material, the properties of the resulting structure are not simply a combination of the properties of the two materials.

The example materials we will use are MgO and CaO, both of which have the rocksalt crystal structure. Both MgO and CaO are ionic, meaning that they consist of charged positive ions (Mg and Ca) and negative ions (O), and are held together principally by the electrostatic attraction between the two (simple chemical theories suggest each Mg and Ca ion should have a charge of approximately 2+, and each O should have a charge of approximately 2-).

Inter-diffusion

In order to grow material A on top of material B perfectly, it is necessary to first obtain a pure, ordered sample of material B with a clean, smooth surface. Material A is then deposited on top of B, often at temperatures of around 1000 K. At these temperatures the atoms can be fairly mobile, and so the atoms of the materials sometimes mix across the interface causing inter-diffusion.

For some materials, a mixed state of A and B is in fact the ground state; this means that inter-diffusion is actually favourable and it is extremely likely to occur. The result is, at best, a broad ‘interface region’ over which the concentration of A increases from 0% to

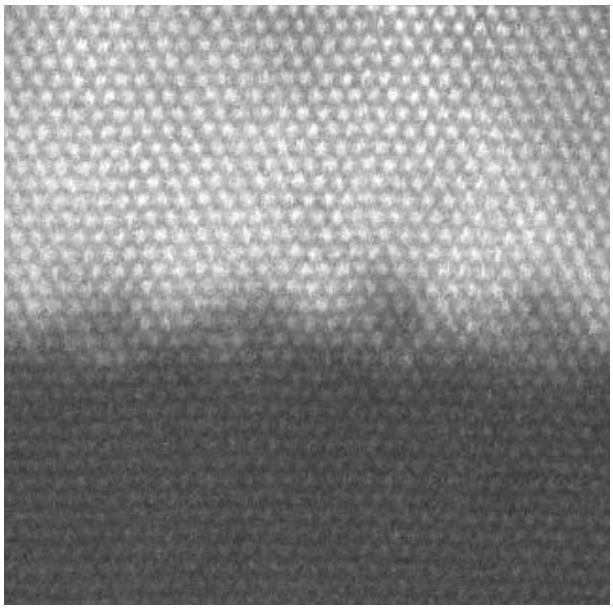

Figure 2: *Electron microscope image of NiO grown on MgO. Both oxides have a rocksalt structure, but the interface is not sharp and contains peculiar triangular features.*

100%; at worst the entire sample is the new AB-mixed crystal, which may not share any properties with either material A or B.

Objectives

Your task is to devise and create a software solution to determine the ground state atomic structure and properties of a system comprised of these two materials (MgO and CaO).

The overall software objectives are:

- Simulation construction
  Automate the construction of appropriate simulation geometries (see section 5).
- Computer modelling
  Model the simulation geometries computationally (see section 6).
- Analyse and visualise the results
  Produce aids to analyse and display the key simulation data (see section 7).

Possible extensions to the project are outlined in section 9.

Do MgO and CaO mix?

For our first task we will attempt to answer the question: 'Is mixing between CaO and MgO energetically favourable?'

Log onto the VLE, go to "Advanced Computational Physics Y3", then "Adv. Comp. Phys. T3" (on the left-hand pane); download the CASTEP input files for MgO.

MgO.param contains parameters telling CASTEP what simulation to do, which approximations to use, and how accurately to determine the ground state.

MgO.cell contains the material's crystal structure, including the lattice and the locations and types of the atoms; in this case, 4 Mg and 4 O atoms in a rocksalt structure.

- Run CASTEP using the MgO input files
  You can run CASTEP by typing:
  `mpirun -np 1 castep.mpi MgO &`
  at a command prompt. '-np 1' means 'use 1 processor core'.
- Examine the CASTEP output file MgO.castep and find the groundstate energy CASTEP has computed for MgO
- Vary the stoichiometry (Mg:Ca ratio)
  Edit MgO.cell to replace one of the Mg atoms with Ca, and compute its groundstate energy with CASTEP. Repeat for 2 Ca, 3 Ca and 4 Ca (pure CaO).
- Plot the computed groundstate energy as a function of the percentage of Ca
- Plot the *formation energy per atom* as a function of the percentage of Ca
  You may wish to ask for guidance for this part.

Simulation construction

How can we determine the groundstate configuration of a set of atoms? We could place our atoms randomly into a simulation cell, use an appropriate computer model (see 6) to compute the energy, compare to the lowest energy found so far and if its lower then store this energy and structure, then repeat. After many many attempts we might assume that the lowest energy we've found is the groundstate energy, and the corresponding structure is the optimal one.

For the particular example of MgO and CaO, both have the same crystal structure and so it is reasonable to assume that any mixed compound will also share this structure. In fact the oxygen sub-lattice is exactly the same in both materials so we should be able to leave the oxygen atoms alone and concentrate on the locations of the Mg and Ca atoms.

For a small simulation cell it is conceivable that we could try every single possible arrangement of Mg and Ca; however as we consider larger and larger systems this would become impractical rather quickly! A more sensible approach for large systems would be some kind of Monte Carlo algorithm. One possible method would be to consider all possible swaps of neighbouring Mg and Ca atoms, and to perform the swap which lowers the energy most (if none lower the energy then the algoruithm terminates). Now consider all possible swaps of neighbouring Mg and Ca atoms for this new geometry, perform the one which lowers the energy most, and repeat until you cannot lower the energy any more.

Given a lattice, possible atomic sites and a stoichiometry (particular numbers of atoms for each element), implement an algorithm to generate possible configurations.

Once we've generated a set of candidate structures, all we need to do is compute their energies...

Computer modelling

Computer modelling can aid tremendously in materials design, provided the computer simulation is sufficiently accurate. In a computer model, the positions and nature of all atoms in the material can be known and the properties calculated. Furthermore 'what if...?' experiments can be performed and the effect on the materials' properties computed, which can provide useful guidance to the experimentalists.

In this laboratory script we will focus on two classes of models: atomistic models (6.1) based on empirical inter-atomic forcefields, and electronic structure calculations (6.2) based on quantum mechanics.

Atomistic models

The simplest models are pairwise potentials; in these models each pair of atoms i and j has an interaction energy E_{ij}, and the total energy of the material is just the sum of these interactions $E = \sum_i \sum_{j<i} E_{ij}$. Evaluation of these potentials is quick and so large material samples may be simulated, provided of course that the potentials are accurate. To avoid confusion with models for nuclear power, these models are often called 'atomistic' rather than 'atomic'.

In general, atomistic simulations are:

- Empirical
  There are many different forcefields and each one has a set of parameters which can be tuned to match a particular materials' properties. You need to decide on a good forcefield and set of parameters, and consider how to test their accuracy.
- Quick
  You should be able to simulate 100s of atoms easily, and many more on a computer cluster.

One example of a pairwise potential is the Lennard-Jones potential, often used to simulate argon and other inert gases:

$$E_{ij} = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right]$$

where ϵ and σ are parameters of the potential, and $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between atom i (at position $\mathbf{r}_i$) and atom j (at position $\mathbf{r}_j$).

The Lennard-Jones potential consists of a long-ranged attractive potential ($\sim \frac{1}{r^6}$) due to the van der Waals interaction (a fluctuating dipole-dipole interaction) and a short-ranged repulsive potential which keeps the atoms apart. This short-ranged repulsion arises from the electron clouds surrounding each atom, but its actual form ($\sim \frac{1}{r^{12}}$) does not have physical significance in the Lennard-Jones potential; in fact Buckingham later replaced it with a harder, exponential repulsive term and this is common in other atomistic potentials (e.g. Born-Mayer potentials).

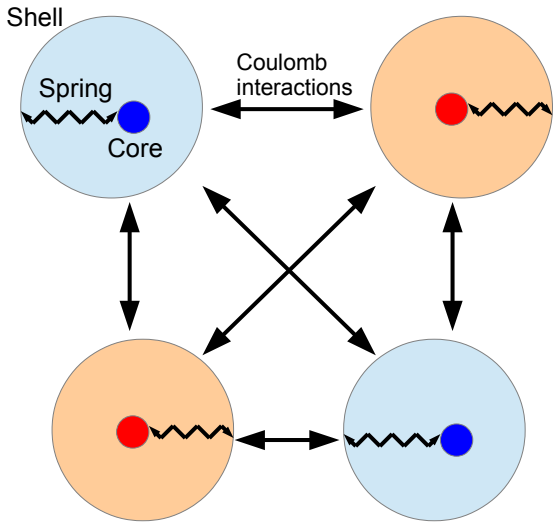

Figure 3: *2D illustration of a shell model for a rocksalt-like crystal. Each shell interacts with its particular core via a spring potential, but in addition all cores and shells interact via a Coulomb potential.*

When more than one atomic species is present, the number of interaction parameters increases. In the case of Lennard-Jones, ϵ and σ become (symmetric) 2nd order tensors ϵ_{nm} and σ_{nm} where n and m are elemental indices so that the interaction between each pair of atoms may vary according to their elements.

For an ionic material the attractive potential is largely due to the Coulomb attraction, not a dipole-dipole interaction, which suggests an interaction energy of the form:

$$E = \sum_i \sum_{j<i} \left(\frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} + A_{ij} e^{-\frac{r_{ij}}{\rho_{ij}}} \right)$$

where q_i, q_j, A_{ij} and ρ_{ij} are all parameters of the potential.

In fact one of the most successful atomistic modelling approaches for ionic systems is the *shell model*, where each atom is replaced by a positive point-like core and a negative (spherical) shell. A spring force connects the shell and the core and keeps them close together, but allows some movement in response to external fields.

Regardless of the precise form of the forcefield, the key to atomistic modelling is to ensure:

- The form of the potential captures the essential physics;
- The parameters of the potential capture the character of your material(s).

You may choose to write your own forcefield simulation program, or you might like to investigate existing programs. The following programs are free for UK academic use:

- GULP (http://projects.ivec.org/gulp/)
- DL_POLY_CLASSIC (http://www.ccp5.ac.uk/DL_POLY_CLASSIC/)

- LAMMPS (http://lammps.sandia.gov/)

Electronic structure calculations

It can be difficult to construct accurate atomistic models without detailed understanding of the behaviour of the materials involved. Furthermore even if the atomistic models are accurate, they can only give structural and atomic-level information; they cannot predict electronic properties. For both of these reasons it is often desirable to employ a more sophisticated simulation method either for the entire investigation, or in order to determine a suitable atomistic model. One such class of methods involve electronic structure calculations.

For a more detailed discussion of electronic structure methods see the Computational Quantum Mechanics course; in this project we will concentrate on density functional theory (DFT), as implemented in the CASTEP program.

In general, density functional theory simulations are:

- Non-empirical
  They solve the Schrödinger equation (under some approximations) and there are no tunable parameters. This means they are usually accurate, and they do not need to be changed for different materials.
- Slow
  Solving the quantum mechanical equations is time-consuming, and you will only be able to simulate 10s of atoms easily.

The CASTEP program

CASTEP is a computer program based on DFT and co-developed at York. The Born-Oppenheimer approximation is used to decouple the nuclear and electronic degrees of freedom and treat the nuclei semi-classically, thus concentrating the quantum mechanical effort on the electrons (see Computational Quantum Mechanics for more details).

CASTEP is designed to study periodic systems, though aperiodic systems may be studied using a supercell method, whereby the aperiodic system is surrounded by a large isolating region (perhaps vacuum or bulk material) so that it is decoupled effectively from its periodic images.

CASTEP takes two input files, of the form seedname.cell and seedname.param, where "seedname" is just a file prefix; thus for an 8-atom silicon unit cell you might call them Si8.cell and Si8.param. The cell file contains the physical information about the simulation cell, such as the lattice vectors and the positions and elements of each atom. The param file contains information about the values of the parameters you want to use for the calculation, such as the task you want CASTEP to perform, the exchange-correlation functional etc. There are useful introductory notes on using CASTEP in Computational Quantum Mechanics practical 3 (see the VLE) and a summary of key concepts in section 12. More detailed documentation may be found on the CASTEP website (http://www.castep.org).

Some of the most relevant parameters for this project are:

- task : singlepoint or geometryoptimisation
  Tells CASTEP to perform either a single energy calculation for this input structure ("singlepoint") or optimise the simulation geometry to minimise the energy ("geometryoptimisation").
- xc_functional : LDA or PBE
  Tells CASTEP to use the LDA (local density approximation) or PBE for the exchange-correlation functional. PBE is a Generalised Gradient Approximation (GGA) developed by Perdew, Burke and Ernzerhof (hence the acronym) and is widely regarded as the most accurate GGA at present. There are many other choices...
- cut_off_energy : 400 eV
  Tells CASTEP to use a plane-wave cut-off energy of 400 eV. You will need to investigate what cut-off energy is appropriate for your simulations.

Some of the relevant cell file keywords are:

- kpoint_MP_grid or kpoint_MP_spacing
  These set the Brillouin zone sampling grid, either directly (e.g. kpoint_MP_grid 3 5 2 gives a 3×5×2 sampling grid) or by specifying the maximum spacing of the sampling points (e.g. kpoint_MP_spacing 0.04 gives sampling points no more than $0.04 \times \frac{2\pi}{\mathring{A}}$ apart).
- fix_all_ions : true or false
  If true, then when performing a geometry optimisation keep the ions (the nuclei + core electrons) in the same *fractional* coordinates.
- fix_all_cell : true or false
  If true, then when performing a geometry optimisation keep the lattice parameters the same.
- cell_constraints
  This isn't actually a single keyword but a block, and is used when you only want to optimise certain latice parameters (see section 12 for more details).

Starting CASTEP tasks

Log onto the VLE and go to the "Advanced Computational Physics Y3" module page, then the "Adv. Comp. Phys. T3" link on the left-hand content pane. You should find CASTEP input files there for magnesium oxide (MgO) and calcium oxide (CaO), which you should download.

Run a series of CASTEP calculations for each system to determine:
- An appropriate cut-off energy
- An appropriate **k**-point sampling

Does your answer change if you consider the convergence of total energy or pressure?

With these in mind, investigate:
- What the optimal lattice constants are for MgO and CaO
- How the results change with exchange-correlation functional

Analysis and visualisation

If all goes according to plan, your software will generate and run a lot of simulations! There will probably be too much data to go through all of it manually, so you need to automate at least some of the analysis and visualisation. You could use "grep" to pick out the final energy from a CASTEP run, for example, but you also need to record the structure and, ideally, visualise it.

- Decide what the key information you need from the simulation is
- Decide how to summarise and/or display it

Programming languages

Now that you know the task at hand, it's a good time to think about the programming language(s) you want to use. There are many different programming languages in the world, and every so often the question arises 'which language is best?' These questions usually stem from a misunderstanding of the purpose of a programming language. All programming languages are a compromise between various competing desires; for example the desire to make life easy for the computer, so it produces an extremely fast program, and the desire to make life easy for the programmer so you can actually finish the programming in a reasonable amount of time!

The two languages we teach are Python and Fortran:

- Python
  - High-level – many operations available either natively or via standard modules
  - Interpreted – easy to debug (the program will often stop on the problematic line)
  - Interpreted – slow
  - Good file and string handling – easy to read, write and manipulate data files
  - Easy to write graphics (especially Visual Python)
- Fortran
  - Low-level – only simple operations available, though many libraries if you look around
  - Compiled – can be hard to debug (source-code is not executed line-by-line)

  - Compiled – fast (possibly the fastest language)
  - Poor file and string handling – only basic operations supported directly
  - Difficult to write graphics – need external libraries and usually C/C++ support
- Shell scripts
  - Wait, what? No-one told us about shell scripts.
  - Executable text files in Linux/Unix (chmod u+x "filename")
  - Contain lists of commands to be carried out
  - Have rudimentary programming features: loops, if statements etc.
  - Easy access to any Linux/Unix command or program, including your own
  - You've already used one – the script you use on Jorvik for CASTEP calculations is a shell script
  - Two main flavours, called "bash" and "c-shell" (csh). Use bash!
  - First line specifies the interpreter to use, e.g. "#!/bin/bash" means "use bash"

Whilst a compromise choice is clearly possible, it may be that a mixed-language solution is best whereby the sub-tasks are implemented in different languages. At its most basic this may involve separate programs that are loosely linked together, for example one program reads as input the other program's output file, but it is often more elegant to use one language to write a library or module that may be called from the other. One such example is the `f2py` utility, which can convert a Fortran program into a Python module, enabling fast, optimised subroutines written in Fortran to be used from within a Python program.

Extensions

This project has many possible extensions, and in many ways it's up to you where you want to take it. You can get full marks on the project without doing any extensions at all, but there are 'bonus marks' available for going beyond the basic problem.

This section contains some suggestions for possible extensions, but it is not exhaustive and you should feel free to investigate any relevant issues or phenomena that interest you.

Geometry optimisation

Atomistic simulation programs can compute the atomic forces as well as the total energy of an atomic configuration. These forces can be used to move the atoms in order to determine the *local* ground state of the atomic configuration, and this is called a *geometry optimisation*. Performing a geometry optimisation ensures that the final energy reported is the lowest one in the neighbourhood of the starting configuration.

Properties of interface planes

If you cut material A and material B in two and stick the halves together, there is an interface between them. Cutting the two materials in different directions will give different interfaces. How do the stability and properties vary with different interfaces?

Point defects

Real crystals have flaws, and for many materials the most common are point defects such as *vacancies* and *interstitials*. Vacancies are gaps in the crystal: lattice sites which should have an atom, but the atom is missing. Vacancies provide a natural mechanism for atoms to move around within the crystal. Interstitials are where extra atoms have been squeezed into the crystal where they don't belong. These defects can have a profound effect on the structure and electronic properties of the material.

Temperature (Kinetic Monte Carlo)

Real crystal growth usually takes place at elevated temperature, where atoms have high kinetic energy. The structure obtained from a geometry optimisation ignores the effects of this thermal energy–essentially it is at 0K–and so the structure and interfaces observed in real growth may not necessarily be the lowest energy ones from a standard calculation.

To model the actual growth, we could modify the Monte Carlo algorithm in 5 to create a *Kinetic Monte Carlo* model. In this method if a proposed swap leads to a higher energy then it is not rejected, but rather it is accepted with a certain probability which is proportional to $e^{-\frac{\Delta E}{k_B T}}$, where T is the temperature of the system, and an input parameter for the simulation. This allows the calculation of thermodynamically stable structures at finite temperature, not just the ones at T=0K.

Alternative models

How do the results depend on the model you have chosen, either the particular atomistic model (section 6.1) or the exchange-correlation in a first-principles model (section 6.2).

Other materials

MgO and CaO are not the only polar oxides with a rocksalt structure. How do the results differ if you consider ZnO, NiO or FeO? For NiO and FeO you may also need to consider the magnetic ground states...

Computational practice

In the past, many computer programs arising from academic research groups were extremely difficult to use and understand. Programs were started by one student, then

continued by another, added to by a third, and so on. The resulting program was a mess, with many bugs and different versions in different places (which may do different things). There was also a tendency to have minimal comments, and no documentation.

Part of the aim of this computational laboratory is to make you aware of good programming techniques, what is often called software "best practice". Some of the techniques you must do yourself, for others there are software tools to help you. Further details may be found in the following subsections, but in essence:

- Structured programming
  Don't start at the beginning and spit out line upon line of code. Design your program neatly, using subroutines and functions where appropriate. In general subroutines should have one entry point and one exit point (though exceptions are usually made in the case where an error is detected).

- Data encapsulation
  Don't pass enormous argument lists between subroutines, wrap the data up in sensible ways. In Python and Fortran you can define your own types of variables, called *derived types*, that can be used as containers for lots of individual bits of data and ensure that it is all held in one place. It is good practice not to allow high-level subroutines to modify the individual data, but to provide special routines to do this.

- Comments
  Comment your code! It is considered good practice to have about 1 comment line for every 2-3 lines of code. In particular you should explain general methodology and procedure, what key variables are for, what subroutines and functions do (including what theor arguments are) and any other operations that are not clear and obvious.

- Documentation
  There are two basic types of documentation: user documentation; and developer documentation. The first kind is focussed on a person who wants to run your code. The documentation needs to be easy to find, and tell the user *as a minimum* how to run the code and interpret the output. Developer documentation is for people who want to extend your program, or perhaps even investigate and fix bugs, and needs to cover what the key subroutines and functions do. A developer will have access to the source code, and hence your comments, whereas a user may not and even if they do they should not need to read it.

- Version control
  A method of tracking versions of your project which records a log of changes and why they were made (e.g. added functionality or fixed a bug). This also allows you to revert to an earlier version if you suddenly discover your brand new all-singing all-dancing algorithm has broken everything and you can't quite remember all the things you changed...

- Unit testing
  Don't just write hundreds (thousands?) of lines of code, put it together, run it and find you get the wrong answer; think about ways to test each particular unit of your program.

Structured Programming

Computer scientists have put much effort into producing languages and standards for formal specification of programs (e.g. Z), but this is beyond what is needed for the present task and you need not be burdened by it. Nevertheless a little forethought and planning can reduce dramatically the amount of time needed to produce a well-structured, error-free computer program and can aid collaborative programming substantially.

The key is to consider how to break your overall task into small, manageable sub-tasks and then to further outline the operations that will be needed to perform each sub-task, and link the operations and sub-tasks together. What will you require as the input to your model, and what should be the output? What steps are required to turn input into output? How will you detect (and reject) nonsensical input? What kinds of data are important in the various tasks, and can these be grouped together sensibly?

Don't be afraid to use modules and/or more than one source file. If a group of subroutines naturally fit together with some data, then it often makes sense to put both in a module.

Data encapsulation

Many computer models of physical systems have a large number of associated parameters and variables, and many operations to be performed on them. The sheer number and variety of these variables can be unwieldy, and so it is sensible to group them together around common themes. For example in an atomic simulation each atom may be assigned a unique number, and there may be an element name (e.g. Si) per atom, some bonding parameters or model for each atom, a position vector and so on. It is good programming practice to keep all of this data together, and to support this modern programming languages allow you to define your own *derived type* to encapsulate the data, e.g.

- Python

```
class atom:
  """ A single atom of the simulation """

  element_name = 'Mg'
  atomic_number = 12
  position = (0.0, 0.0, 0.0)
```

  Which you can then use in your program using "." to select the various component data, e.g.

```
simulation_atom = atom()

""" Put a silicon atom at the origin """
simulation_atom.element_name  = 'Si'
simulation_atom.atomic_number = '14'
simulation_atom.position      = (0.0,0.0,0.0)
```

- Fortran 90 and later

```
type :: atom
  ! A single atom of the simulation
```

```
  character(len=3)                :: element_name
  integer                         :: atomic_number
  real(kind=dp),    dimension(3)  :: position

end type atom
```

Which you can then use in your program using "%" to select the various component data, e.g.

```
type(atom)  :: simulation_atom

! Put a silicon atom at the origin
simulation_atom%element_name  = 'Si'
simulation_atom%atomic_number = '14'
simulation_atom%position(1:3) = (0.0_dp,0.0_dp,0.0_dp)
```

(where we have assumed a double precision real kind has been defined, called 'dp'.)

You can even create arrays of such types, e.g.

```
type(atom), dimension(:), allocatable  :: simulation_atom

allocate(simulation_atom(num_atoms),stat=status)
if(status/=0) stop 'Error! Cannot allocate memory to array simulation_atom'

! Make the 1st atom a silicon atom at the origin
simulation_atom(1)%element_name  = 'Si'
simulation_atom(1)%atomic_number = '14'
simulation_atom(1)%position(1:3) = (0.0_dp,0.0_dp,0.0_dp)
```

Comments

For your program to be useful to anyone else, or indeed yourself in the future, it is vital that the program should be both commented and documented. Comments are used in the source code to explain what the program is doing; they should be concise but informative, assuming the reader has a reasonable knowledge of the programming language but not necessarily the method you are implementing. A good rule of thumb is one comment line for every three lines of code. Both Fortran and Python allow 'in-line' comments, and these can be useful for short, line-specific comments, whereas longer comments are more useful for explaining the overall operations, e.g.

```
x_old  = 0.0_dp    ! initialise x to zero
lambda = 0.1_dp    ! set the default step length

! Start the main iteration loop to solve the equations
! This uses a conjugate gradient minimisation method to generate the proposed change
! in x (stored in delta_x) and then
iter_loop: do iteration=1,max_iterations
  call compute_delta_x(x_old,delta_x)
```

```
  x_new = x_old + lambda*delta_x
end do iter_loop
```

Documentation

In general some entry-level documentation should be present, and obvious. Many programs have a file called README in the top-level directory which has a brief description of what the software does, how you compile/install/run it, and where more detailed documentation can be found. Because people will often be reading the documentation on a text terminal, it's best to stick to plain text files rather than Word or OpenOffice documents though there's nothing stopping you having a prettier alternative (though a PDF is probably friendlier than a .docx).

You can also get more creative:

- Have some built-in help
  Try typing "tail –help".
- Write a "man" page
  If you've never used "man", try "man tail". Use "q" to quit.
- Write a web page

For more detailed documentation you might like to investigate automated documentation generation tools. Python actually has one built-in, but there's also a program called "doxygen" which can be used with both Python and Fortran (and C, C++, Java...). You need to create a doxygen configuration file first; it's probably easiest to get doxygen to create a default configuration file:

```
doxygen -g
```

which produces a file called "Doxyfile" – you can edit this file to tailor it to your aims. You probably want to edit PROJECT_NAME and change EXTRACT_ALL to "yes" and OPTIMIZE_FOR_FORTRAN to "yes" at the very least.

There is a very simple Fortran example "doxygen_example.f90" on the VLE, plus a Doxyfile. Download both, and then run "doxygen Doxyfile" to generate the documentation. The html documentation is put in a subdirectory called "html", and you will need to view it in a web browser – the top-level html file is "index.html". Detailed instructions and documentation can be found at http://www.doxygen.org.

Version control

"Version control" or "Revision control" is the name given to a system of keeping track of previous versions of your project. Typically they allow you to store any version of the project – not just one file, but all files in all subdirectories, including documentation – and recover any previous version. This is very useful when you find and fix bugs, as you can easily record what the bug was and how you fixed it in the log message. It's also useful when something that used to work becomes broken: you can go back through earlier

versions until you find one that works, and then compare this to the current version to find out which change broke the functionality.

There are many tools to perform version control, and the choice is largely a matter of taste. Two popular modern version control systems are mercurial (Hg) and git. Each of these works by creating a software repository which contains information about all versions of the code. When you have a version of your program you want to put in the repository, this is called "committing" or "checking in" the file. There's a brief guide to using Mercurial on the VLE.

Unit Testing

Most useful programs are quite long and complex, so if they don't give the answer you're expecting it can be difficult to know whether the problem lies with the code or with your expectations! It is good practice to test each unit of your program independently, and this is called *unit testing*. For example if you implement a pair potential you could test it by giving it two atoms at known coordinates (perhaps one at the origin) and comparing the answer to one you worked out yourself. If you reimplement something, does it give the same answer as the previous method? Testing is important!

General computational laboratory procedures

This section contains general computational laboratory advice that is relevant for the group project. It is in part a summary of the full laboratory notes, which are available on the departmental intranet, and you may also find that document useful as a reference point.

Disk back-up

You cannot guarantee that your data or program left on the hard disk of a machine will remain uncorrupted, nor that you will necessarily use the same machine each session. It is therefore essential that you make a copy of all your files on to a USB stick or a different machine *in a different location* at the end of every session. You are responsible for the safe keeping of your work.

Laboratory notebooks

You are required to keep a laboratory notebook in which you enter the full details of your experiments. Even though for this project it takes the form of a Google Doc, the maintenance of a laboratory book remains an essential skill for a practicing scientist and the ability to write a concise account of what has been achieved is often necessary. Your lab book should be kept in an accepted style so that if necessary another scientist could continue your work.

You may wish to place rough notes, calculations or sketches in an appendix and keep your neat tables of results and other comments in the main document. Under no circumstances

should you be writing "lab book data" on bits of paper or into other files, and the demonstrators are instructed to tell you not to do this.

Laboratory books should be neat and well organised and should be a contemporaneous record of what you have done. A laboratory notebook does not necessarily contain the same elements as are in a formal report. The laboratory script also forms part of your laboratory record and hence there is no information in the script that you need copy out into your book. In each session, as part of the team planning meeting, you and the rest of your team should reread the relevant section of the script and plan the code either as a flowchart or as pseudocode.

CASTEP

---

k-point sampling

Various quantities such as the energy or charge density are the result of integrations over the first Brillouin zone of the simulation cell. For example we know from solid-state physics than the energy of a band varies across the Brillouin zone, so that the the total energy E_b of the band b is:

$$E_b = \iiint_{\mathbf{k}} E_b(\mathbf{k}) d^3\mathbf{k}$$

This integral cannot be performed analytically in general, since we don't know what the analytic form of $E_b(\mathbf{k})$ is! In CASTEP these integrals are approximated by a summation over discrete sampling points, called **k**-points:

$$E_b \approx \sum_{\mathbf{k}} E_{b\mathbf{k}}$$

where the quality of the approximation is controlled by the density of sampling points. The actual sampling points are usually evenly spaced on a regular 3D 'Monkhorst-Pack' grid (i.e. chosen according to a scheme proposed by Monkhorst and Pack).

The summation may over- or under-estimate the integral, and this may change as the density of **k**-points is increased so that it is common for the value to oscillate as more and more sampling points are used. Clearly the results of any simulation must be converged with respect to these points in order for it to be meaningful.

The **k**-point sampling is specified in CASTEP's cell file using either:

- kpoint_MP_grid p q r
  Tells CASTEP to use a $p \times q \times r$ **k**-point mesh. In general you want the same quality of sampling in each direction, so if your simulation cell is cubic you should probably choose $p = q = r$.
- kpoint_MP_spacing α
  Tells CASTEP to generate a **k**-point mesh for you such that the **k**-points are no more than α apart (in units of $2\pi/\text{Å}$).

Cut-off energy

CASTEP exploits the periodicity of the simulation cell to expand the electronic wavefunction as a Fourier series. Because this is a 3D Fourier series the complex exponentials represent a plane-wave travelling in space, so this is often referred to as a plane-wave basis. Each band b of the wavefunction at each sampling point $\mathbf{k}$ in the first Brillouin zone is written in this basis as:

$$\psi_{bk}(\mathbf{r}) = \sum_{\mathbf{G}} c_{Gbk} e^{i\mathbf{G}\cdot\mathbf{r}}$$

where the sum is over all reciprocal lattice vectors $\mathbf{G}$; this is an infinite sum! In practice it is found that as $|\mathbf{G}| \to \infty$, $|c_{Gbk}| \to 0$ so that this sum can be truncated safely beyond a certain cut-off G_{cut}, so that only plane-waves with $|\mathbf{G}| \leq G_{cut}$ are included in the sum.

Since $|\mathbf{G}|^2$ is proportional to the kinetic-energy of the plane-wave it is common to define this cut-off in terms of a cut-off energy E_{cut} rather than a wavevector directly:

$$E_{cut} = \frac{\hbar^2}{2m}|\mathbf{G}|^2$$

It is necessary to determine a suitable cut-off energy for your work before any serious simulation work may be undertaken. The cut-off energy only depends on the property of interest and the elements involved, not the actual crystal structure.

The cut-off energy is specified in CASTEP's param file via:

- cut_off_energy α
  Tells CASTEP to use a cut-off energy of α (in units of eV). Cut-off energies are typically several hundred eV.

Pseudopotentials

The innermost electrons on each atom are so tightly bound to the nucleus that their behaviour is almost independent of the physical and chemical environment of the atom. Rather than spend computational effort computing these "core" states, CASTEP employs a further approximation called the "frozen core" approximation; the core states are precomputed for each element, and a modified potential is created that includes their effect on the outer "valence" electrons. This modified potential is called a "pseudopotential" so this method is also sometimes called the "pseudopotential approximation".

For the same reason that the core electrons may be assumed to be "frozen", the form of the wavefunction near the nucleus stays the same even for the valence electrons. Pseudopotentials can be constructed to exploit this by basically changing the potential near the nucleus and only computing the *change* in the valence wavefunctions near the nucleus.

Pseudopotentials are specified for each element in CASTEP's cell file, in a block called SPECIES_POT:

- %BLOCK SPECIES_POT
  Tells CASTEP this is the start of a section specifying the potentials to use for each species (element).

- symbol file
  Tells CASTEP that the element whose symbol is "symbol" has a corresponding pseudopotential in the file "file". E.g.
  Si Si_00.usp
  tells CASTEP to use the pseudopotential in the file "Si_00.usp" for elements with the symbol Si – i.e. silicon.
- %ENDBLOCK SPECIES_POT
  Tells CASTEP this is the end of the pseudopotential block.

Other CASTEP parameters

There are many other parameters and settings you might want to investigate, for more detailed documentation see the CASTEP website (http://www.castep.org). Some of the most relevant ones in the param file are:

- task : singlepoint or geometryoptimisation
  Tells CASTEP to perform either a single energy calculation for this input structure ("singlepoint") or optimise the simulation geometry to minimise the energy ("geometryoptimisation").
- xc_functional : LDA or PBE
  Tells CASTEP to use the LDA (local density approximation) or PBE for the exchange-correlation functional. PBE is a Generalised Gradient Approximation (GGA) developed by Perdew, Burke and Ernzerhof (hence the acronym) and is widely regarded as the most accurate GGA at present. There are many other choices...

Some relevant cell file keywords are:

- symmetry_generate
  Tells CASTEP to see whether the simulation cell has any symmetries, and to use these to reduce the number of **k**-points generated (symmetries in real-space have corresponding symmetries in reciprocal-space). CASTEP will always assume time-reversal symmetry, so that $\mathbf{k}$ and $-\mathbf{k}$ are equivalent; this is why even with no symmetry a $3 \times 3 \times 3$ **k**-point grid leads to 14 **k**-points rather than 27.
- fix_all_ions
  When performing a geometry optimisation, keep the ions (the nuclei + core electrons) in the same *fractional* coordinates.
- cell_constraints
  This isn't actually a single keyword but a block, like "SPECIES_POT". You can use it to specify constraints on the cell degrees of freedom. Its syntax is simplest to explain by example:

  ```
  %BLOCK CELL_CONSTRAINTS
  1 1 2
  0 0 0
  %ENDBLOCK CELL_CONSTRAINTS
  ```

  The 6 integers refer to the 6 lattice degrees of freedom a, b, c (the cell's lattice constants) and α, β, γ (the cell angles). If an integer is zero it means "do not optimise

this parameter"; if an integer is positive it means "optimise this parameter"; if two integers are the same, it means "these parameters should be optimised together", in other words they will be changed by exactly the same amount. Thus in the example above, the a and b lattice constants will be allowed to optimise, but enforcing $a = b$; the c lattice constant is also optimised; the angles α, β and γ will be fixed at their initial values.