Shell 4.3 users' guide

M B Taylor G D Barrera

April 3, 1998

RCS: Source: /home/mbt/shell/doc/RCS/shelluser.tex,v Revision: 2.1 Date: 1998/03/06 11:06:07

Contents

List of Figures

1 Legal and contact information

Shell is available free of charge to academic institutions and non-commercial establishments only. Copies should be obtained from the authors only and should not be distributed in any form by the user to a third party without the express permission of the authors. All rights for this code remain with the authors.

No claim is made that this code is free from errors and no liability will be accepted for any loss or damage that may result. The user is responsible for checking the validity of the results.

Copyright:	Neil Allan, Mark Taylor (Bristol, UK) Gustavo Barrera (Buenos Aires, Argentina)
Contact:	n.l.allan@bristol.ac.uk
Information:	`http://dougal.chm.bris.ac.uk/programs/shell/`

2 Introduction

This document is a guide for use of the lattice dynamics program Shell and also for some associated utility programs. Section 3 gives a detailed description of the input file format, section 4 explains the capabilities of the program, in the form of its behaviour in each of the available run modes, section 6 describes, mainly by way of example, the set of utility programs which are designed to make Shell easier and more productive to use, and section 7 describes how to install the program and associated files.

Detailed discussion of the theory used by this code exists elsewhere (see section 8).

2.1 About this document

This document exists in postscript and HTML form, generated from the original LaTeX by the excellent TtH. The hypertext form can be much more useful because of the extensive crosslinking of keywords although some of the typesetting is a bit inferior.

The mathematical parts of the text are typeset in the HTML version using HTML 3.2, which is supported on most relatively recent browsers. If some of the mathematics looks quite garbled using Netscape under X however you can fix it fairly painlessly as described in `http://hutchinson.belmont.ma.us/tth/Xfonts.html`.

2.2 Note for the impatient

It is not *necessary* to read all of this document. This section (2.2) contains the minimum you can get away with knowing.

2.2.1 Installing Shell

Source distribution (file `shell.tar`)
: To install Shell from source, you *could* try just unpacking the tar file (`tar xvf shell.tar`) in a directory called, say, `shell`, and running `make` in the `shell/src` subdirectory, consulting section 7 if you run into problems. You are however *advised* to read through that section anyway. It's not very long.

Binary distribution (file `binshell.tar`)
: You can just unpack the tar file (`tar xvf binshell.tar`) in a directory called, say, `shell`. You are *advised* to read through section 7 also.

2.2.2 Running Shell

You are advised to add the directory `shell/bin` to your `PATH`.

Source distribution
: If you have the source distribution then after compilation the Shell executable will probably be called `shell` in directory `shell/bin`.

Binary distribution
: If you have the binary distribution there should be executables in `shell/bin` called `shell.`n, where n is the maximum number of particles per unit cell for which it was compiled. Use the executable which corresponds to the smallest n which is greater than or equal to the number of particles (cores+shells) in your unit cell.

The quickest way to get started is to use the example file `examp.shl` (in directory `shell/doc`, or figure 1 of this document) modifying it to your own requirements by reference to section 3 and possibly 4. Using figure 1 of the hypertext version of this document is a pretty painless way to see what the file format means. Skimming through sections 3 and 4 before making much use of the program would be quite a good idea however. Shell is run by feeding a data file to standard input, e.g. `shell < shell/doc/examp.shl`.

It is not *necessary* to use the utility programs, and hence to read section 6, but it is advised for those who will make more than casual use of the program, especially within shell scripts, to look at least at sections 6.1 (`shed`) and 6.7 (`funcof`). The usefulness of the other utilities described in section 6 will vary from "no use at all" to "extremely useful" according to what you want to do.

3 Shell input file format

Shell does calculations as instructed by an input file, of which the format is described here.

3.1 Example input file

In figure 1 is shown an example input file `examp.shl` for a full optimisation of MgF_2 at finite temperature and zero pressure. Figure 2 shows the alternative format for specifying the `INTERNAL` strains. Examining and modifying this file, which is supplied in the Shell distribution as `shell/doc/examp.shl` should be an easy way to generate and understand Shell input files.

A more formal definition of the input file format is given in the next two sections: section 3.2 describes the general rules for parsing the input, and section 3.3 gives a detailed explanation of each part of the input file format.

3.2 General points about input file parsing

- The input file is line-formatted. On the whole, each line starts with a keyword which is followed by some values or other keywords. Keywords and other values are separated by one or more spaces. Some keywords however introduce a further set of lines which do not start with keywords.
- In general, each line can be up to 200 characters, and each word or value can be up to 30 characters. These values can be changed by modifying the header file `paramete.h`.
- Only the first four characters of *keywords* are significant.
- Blank lines are ignored.
- Any text after a '`#`' character on a line is ignored.

Figure 1: File `examp.shl`: a sample Shell input file

```
TITLE Magnesium Fluorite rutile structure with shells
LATTICE TP  4.66 3.19
PARTICLES
     F        0          -1.13757
    cF       18.99984     0.23357
     Mg      24.305       1.808
BASIS
     Mg   0      0      0
     Mg   0.5    0.5    0.5
    cF    0      0      0
    cF    1      1      0
    cF    0.5    0.5    0.5
    cF    0.5    0.5    0.5
     F    0      0      0
     F    1      1      0
     F    0.5    0.5    0.5
     F    0.5    0.5    0.5
POTENTIALS       #  Potentials of Barrera et al.
    EXPO     F   Mg      5971.7    0.2127      0.2   8.0
    EXPO     F   F       22400.    0.1917      0.2   8.0
    INVERSE  F   Mg     -8.9596    6.          0.2   8.0
    INVERSE  F   F       24.804    6.          0.2   8.0
    SPRING   F  cF      17.496                 0.    0.2
CUTSHELL 0.3
OPTION
MESH 3
TEMP 1000
PRESS 0
RUN OPTIMISE CELL INTERN
INTERNAL  0.3057
     3       1    1    0
     4      -1   -1    0
     5      -1    1    0
     6       1   -1    0
     7       1    1    0
     8      -1   -1    0
     9      -1    1    0
    10       1   -1    0
INTERNAL  0.0004
     7       1    1    0
     8      -1   -1    0
     9      -1    1    0
    10       1   -1    0
END
```

Figure 2: Alternative format for specifying INTERNAL strains

```
INTERNAL  0.3057
      0    0    0
      0    0    0
      1    1    0
     -1   -1    0
     -1    1    0
      1   -1    0
      1    1    0
     -1   -1    0
     -1    1    0
      1   -1    0
INTERNAL  0.0004
      0    0    0
      0    0    0
      0    0    0
      0    0    0
      0    0    0
      0    0    0
      1    1    0
     -1   -1    0
     -1    1    0
      1   -1    0
```

- Leading and trailing space on a line is ignored.
- Any text after the END keyword is ignored.
- Keywords are not case sensitive (though names of particles are).
- The order of the lines is important
- Floating point numbers may be entered in the normal way (with or without a sign and/or an exponent); additionally fractions like '1/3' (or even, say, '+1.0d+00/3.') may be entered, which can be handy for specifying atomic positions.
- Some validation of input is done. Shell attempts to detect invalid input and terminate with a helpful error message; however, this detection is not perfect.
- Shell reads its input from the standard input channel; therefore the "input file" may in fact not be a file but, e.g., the output of a UNIX pipe (see section 6.1 for examples), or text typed in from the keyboard. Typically however (under UNIX) Shell will be run using a command like

  ```
  % shell < examp.shl
  ```

3.3 Detailed description of input file

The lines in the input file are described here one by one. Unless otherwise noted, all lines are mandatory and must appear in the order shown here. The following typographical conventions are used:

KEYWORD

variable-element

[optional element or line]

(line to be repeated *num* times) ∗ *num*

may-occur-one-or-more-times . . .

{ one or | the other }

The format is as follows:

`TITLE` *title-text*
The *title-text* serves to label the file. It has no effect on the calculation but will be included in the output file.

{ `LATTICE` | `SLAB` } *lat-type lat-param* . . .
Normally the `LATTICE` keyword will be used, indicating a structure which is periodic in three dimensions. However if the keyword `SLAB` is used instead the structure is assumed to have a slablike geometry, i.e. is not periodic in the direction of the third (vertical) unit vector. The difference is principally in how the lattice sums and Brillouin zone summations are performed. For `SLAB` geometries the external strain derivatives (and that functionality of the program which depends on them) don't work. If an attempt is made to use the external strain derivatives with `SLAB` geometry the program will terminate with an error message.
The number and meaning of the *lat-param* values which must be specified varies according to the value of *lat-type* as follows:

lat-type	latparam . . .						structure
`CP`	a					–	simple cubic
`CF`	a					–	face centred cubic
`CI`	a					–	body centred cubic
`TP`	a	c				–	primitive tetragonal
`TI`	a	c				–	body centred tetragonal
`OP`	a	b	c			–	primitive orthorhombic
`OI`	a	b	c			–	body centred orthorhombic
`OC`	a	b	c			–	base centred orthorhombic
`OF`	a	b	c			–	face centred orthorhombic
`HP`	a	c				–	hexagonal
`HR`	a	α				–	rhombohedral
`MP`	a	b	c	α		–	primitive monoclinic
`MC`	a	b	c	α		–	base centred monoclinic
`AP`	a	b	c	α	β γ	–	triclinic

where a, b and c are lengths in Ångstrom of the unit cell vectors and α, β, γ are the angles between them in degrees.
The *lat-type* names follow the usage in the International Tables of X-Ray Crystallography.

`PARTICLES`
This line is required to introduce the definitions of the different particles which appear in the following lines:

(*type-name mass charge*) ∗ *num-types*
For each type of particle which will appear in the unit cell its name, mass and charge must be specified. *type-name* (which is case-sensitive, and can be up to 20 characters long) will typically be a chemical symbol, perhaps with some modification to indicate whether it's a shell or a core; if atoms of the same species with different effective charges are to be represented in the unit cell they should be given different labels here. *mass* is the mass in atomic mass units and *charge* is in units of proton charge.

`BASIS`

This line is required to introduce the positions of the particles in the unit cell which appear in the following lines:

(*type-name x y z*) $*$ *num-particles*

One line for each particle in the basis gives the particle's type *type-name* as defined in the `PARTICLES` stanza, and x, y, z coordinates in unit cell space (i.e. fractional coordinates).

The x y z coordinates may be given as any number, but will normally be mapped to the interval 0–1 for the purposes of the calculation and output. This remapping is not performed in the third (vertical) direction for the case of `SLAB` geometry however, for which reason it's not a good idea to specify z coordinates way outside this range for slabs.

Important: All the core particles (*mass* $\neq 0$) *must* be specified before all the shell particles (*mass* $= 0$). Otherwise, the program will terminate with an error message.

If *num-particles* (the number of lines given) is greater than the `MAXNSL` value compiled into the code (see section 7) then the program terminates with an error to the effect that it must be recompiled.

`POTENTIALS`

This line is required to introduce the definitions of the inter-particle potentials which appear in the following lines:

(*pot-type type-name* ... *param* ... *cutoff* ...) $*$ *num-pots*

One line for each potential specifies an interaction between the particle types whose *type-name*s are given. Zero, one or many potential lines may be listed for each set of *type-name*s. Currently Shell handles two- and three-body potential types, which have different numbers of *type-name* and *cutoff* parameters:

Two-body

pot-type type-name type-name param ... *short-cut long-cut*

The two *type-name*s define the particles participating in the interaction, and *short-cut* and *long-cut* are the shortest and longest distances over which the interaction operates (outside this range it is taken to be zero). The available two-body interaction types, and the *param*s which must be specified for them, are as follows:

pot-type	*param* ...			$\phi(r)$
`EXPO`	A (eV)	ρ (Å)		$Ae^{-r/\rho}$
`MORSE`	A (eV)	B (Å^{-1})	s (Å)	$A\left(1-e^{-B(r-s)}\right)^2 - A$
`INVERSE`	C (eVÅ^n)	n		Cr^{-n}
`SPRING`	k (eVÅ^{-2})			$kr^2/2$
`SPROFF`	k (eVÅ^{-2})	s (Å)		$k(r-s)^2/2$

Three-body

pot-type type-name type-name type-name param ...
low-theta high-theta short-1 long-1 short-2 long-2

The three *type-name*s define the participating particles, the first listed being the central one, so that if the vector between the first and second listed particles is $\mathbf{r}_1$ and between the first and third is $\mathbf{r}_2$ then the angle between them is $\theta = \cos^{-1}(\mathbf{r}_1.\mathbf{r}_2/r_1r_2)$.

Three (low,high) ranges must be supplied, corresponding to θ, r_1 and r_2 respectively. If any of the values is outside the range given the interaction is taken to be zero. Parameters for the available types of three-body interaction are as follows:

pot-type *param ...*	$\zeta(\theta)$
`HAR3` $k(\mathrm{eV\,Rad}^{-2})$ $\theta_0(°)$	$k(\theta-\theta_0)^2/2$
`HEX3` $k(\mathrm{eV\,Rad}^{-2})$ $\theta_0(°)$ $\rho_1(\text{Å})$ $\rho_2(\text{Å})$	$ke^{-r_1/\rho_1-r_2/\rho_2}(\theta-\theta_0)^2/2$

So for example the line

```
HAR3   Si O O   2.09 109.5   80 150   0.5 1.8   0.5 1.8
```

specifies a potential harmonic in the O–Si–O angle, with parameters $k = 2.09\,\mathrm{eV\,Rad}^{-2}$ and $\theta = 109.5°$, which applies to triples in which $80° \le \theta \le 150°$, $0.5\,\text{Å} \le r_1 \le 1.8\,\text{Å}$ and $0.5\,\text{Å} \le r_2 \le 1.8\,\text{Å}$.

`CUTSHELL` *shell-cutoff*

shell-cutoff, given in Ångstrom, is the maximum separation between two particles for them to be considered as shell and core of the same ion. It is used only by the part of the code which calculates Coulombic lattice sums (every pair of particles interacts with a Coulombic potential except if they are shell and core of the same ion). It must therefore be chosen to be greater than the largest possible displacement of a shell from its core, and smaller than the closest possible approach of two different atoms. 0.3 Ångstrom is a pretty safe choice.

[`OPTION` [*opt-key* [*opt-param*]] ...]

This line is optional. It may also exist with no parameters.
Here one can specify various options which modify the way the calculations are done. If the line is missing, or there are no *opt-key* arguments, sensible defaults will be assumed.
Some options have a parameter, and some don't. Possible values for *opt-key* and *opt-key opt-param* are as follows:

`ETAFAC` *eta-factor*:

A dimensionless factor *eta-factor* is applied to the parameter* η of the Ewald summation which determines how the summation is split between direct and reciprocal space. It is set to the value *eta-factor* $\times\ \pi^{1/2}(\mathrm{Volume})^{-1/3}$ (or *eta-factor* $\times\ \pi^{1/2}(\mathrm{Area})^{-1/2}$ for `SLAB` geometry).
If the line is omitted, *eta-factor* defaults to unity.

`CLASSICAL`:

If this option is specified the free energy is calculated in the classical limit (mode contributions to F are $kT\ln\beta\hbar\omega$) rather than using normal phonon (Bose-Einstein) statistics. Other thermodynamic quantities are calculated accordingly. This may be useful e.g. for comparison with Monte Carlo simulations.

`NOZPE`:

If this option is specified the free energy is calculated omitting the zero point energy (mode contributions to F are $kT\ln[1-e^{-\beta\hbar\omega}]$) rather than using normal phonon (Bose-Einstein) statistics. Other thermodynamic quantities are calculated accordingly.

`USER` *user-var*:

This is a debugging aid. It sets a variable `USER` which is available via a common block in the program. If you're not going to be mucking about with the source code this is of no use; but if you are it can be very useful for testing modifications to the source without having to keep recompiling.

`CONVERGE` *opt-tol*:

Controls the accuracy of convergence in optimization modes `VMM` (default), `NEWT`, `NNEWT`, and `1NNEWT`. The structure is considered to have

*This parameter is sometimes called G.

optimised when the magnitude (root mean squared) of the gradient does not exceed this value. Default value is *opt-tol* = 0.01.

`IMAGINARY` *num-imag*:
Normally if the dynamical calculations generate any imaginary vibrational frequencies the program is terminated. If *num-imag* is set to a non-zero value however, that many imaginary frequencies will be tolerated before the program gives up. If *num-imag* is set to a negative number it will carry on however many imaginary frequencies are encountered.
Tolerated imaginary frequencies are dealt with by effectively multiplying them by $\sqrt{-1}$. This option can be useful because occasionally an `OPTIMISE` run can recover from running into a region of configuration space which generates a few imaginary frequencies.

`MESH` *x-mesh* { [*y-mesh z-mesh*] | [*y-mesh*] }
This determines how many reciprocal lattice vectors are used for the summation over the Brillouin zone. Normally either one or three values must be supplied, but for `SLAB` geometry either one or two must be supplied. If only one is given the others are set to the same value. A rectangular grid of (*x-mesh* × *y-mesh* [× *z-mesh*]) wave vectors with the given number of nodes in each direction is used. The Γ-point is not on a node.
For some symmetries (currently only `CP` and `TP`) the symmetry of the Brillouin zone is exploited to reduce the number of calculations needed.
The wave vectors actually used can be seen by, e.g. `grep Q freq.out` after a `FREQ` run.
If *x-mesh* = 0 then all calculations done by the program are static only (frequencies are not calculated), and the temperature *temp* is ignored.
This line is ignored in run-modes `DISPERS` and `GRUNEIS`.

`TEMP` *temp*
The temperature *temp* is given in Kelvin. This line is ignored if *x-mesh* = 0.

`PRESS` *press*
The hydrostatic pressure *press* is given in GPa. Non-zero pressures are not defined for `SLAB` geometries.

`RUN` *mode* [*param* ...]
Different parameters *param* must be given according to the type of run specified by *mode*, as follows:

run-mode	[*param* ...]
`THERMO`	
`FREQ`	
`GRADIENT`	*var-set*
`OPTIMISE`	*var-set* [*opt-method*]
`DISPERS`	*qx qy qz num-points*
`GRUNEIS`	*qx qy qz num-points*
`ELASTIC`	*elast-options*
`TEST`	*var-set*
`BENCH`	[*var-set*]

var-set is a combination of {`CELL`, `BASIS` , `INTERN`, `ZSISA`} (see table 1).
opt-method is optionally one of {`VMM`, `NEWT`, `1NNEWT`, `NAG`, `NUM`, `OLD`}; if omitted `VMM` is assumed (see section 4.4.1).
num-points is the number of (equally spaced) points between the origin and the reciprocal lattice vector (*qx*, *qy*, *qz*) (units of Ångstrom^{-1}) at which to make calculations.

More details about the meanings of these run modes and options are given in section 4.

[(`INTERNAL` *strain-val*
([*index*] *x-mult y-mult z-mult*) $*$ *num*) $*$ *num*]
Optionally, a set of symmetric internal strains (offsets from the basis positions) may be specified in this way. Each is introduced by an `INTERNAL` line followed by the dimensionless magnitude *strain-val* of the strain.
This is followed by

either *num-particles* lines of the form: *x-mult y-mult z-mult*
referring to the particles in the same order as listed in the `BASIS` (as in figure 2)

or one or more lines of the form: *index x-mult y-mult z-mult*
where *index* gives the row number of the particle in the `BASIS` listing (as in figure 1). Particles for which an *index* is not given are assumed to have (x-*mult*, y-*mult*, z-*mult*) $= (0, 0, 0)$.

The position of each particle is then given by the (x,y,z) coordinates in the `BASIS` plus *strain-val* times the appropriate (x-*mult*,y-*mult*,z-*mult*) line from each `INTERNAL` block.
These internal blocks both modify the particle positions (if some of the *strain-val* $\neq 0$) and define a set of `INTERN` strains which can be used in some of the run modes as per table 1.

`END`
This line must be present. When it is read subsequent lines are ignored and calculation begins.

4 Program functionality

This section describes the calculations made by the program, and the output files produced, when run in its various *run-modes*.

4.1 `THERMO` mode

Format:	`RUN THERMO`
Output files:	`shell.out` `timing.out`

The static energy U is calculated. If *x-mesh* > 0 then then the dynamical properties at temperature *temp* are also calculated. If any of the eigenvalues of the dynamical matrix are negative (implying imaginary vibrational frequencies) the program is normally terminated, unless the `OPTION IMAGINARY` option has been specified.

Output is written to file `shell.out` The format of `shell.out` is quite self-explanatory: first a set of named results is printed with units, then the two- and three-body short ranged potentials used, then the lattice parameters *lat-param* and internal strains *strain-val* (if any) in the same units as in the input file, then the unit cell primitive vectors row-wise in Ångstrom, then the x, y, z basis coordinates in a format suitable for splicing into a Shell input file. Note that the x, y, z coordinates *include* the effect of any of the internal *strain-val* strains. An example `shell.out` file is shown in figure 3. The `Mesh =` line has three components normally, or two components for `SLAB` geometry.

A file `timing.out` which contains information about how many seconds of CPU time were spent in different parts of the code, is also written.

Table 1: All allowed values of *var-set* used for selecting gradient types. $\epsilon^{\rm ext}$ are the external lattice parameters *lat-param*, u_i^α are the $3\times$ *num-particles* strains x, y, z for each particle, w_m are the *strain-val* strains, and $G_{\rm stat}$ is the static part of the free energy G.

var-set	derivatives calculated
CELL	$\partial G/\partial\epsilon^{\rm ext}$
BASIS	$\partial G/\partial u_i^\alpha$
INTERN	$\partial G/\partial w_m$
CELL BASIS	$\partial G/\partial\epsilon^{\rm ext}$, $\partial G/\partial u_i^\alpha$
CELL INTERN	$\partial G/\partial\epsilon^{\rm ext}$, $\partial G/\partial w_m$
CELL BASIS ZSISA	$\partial G/\partial\epsilon^{\rm ext}$, $\partial G_{\rm stat}/\partial u_i^\alpha$
CELL INTERN ZSISA	$\partial G/\partial\epsilon^{\rm ext}$, $\partial G_{\rm stat}/\partial w_m$

These calculations are made, and the files `shell.out` and `timing.out` produced, in all run modes. Where a run mode has not calculated a quantity which appears in `shell.out`, the quantity is written to the file as zero.

4.2 FREQ mode

Format: `RUN FREQ`
Output files: `shell.out` `freq.out` `timing.out`

This is the same as `THERMO` mode, but additionally writes a file `freq.out` containing each wave vector used for the Brillouin Zone integration (in Ångstrom^{-1}) with the weighting of that point for the integration, followed a list of all the freqencies at that wave vector (in THz). For dense wave vector meshes this file can be quite long, which is why `THERMO` mode doesn't write `freq.out` by default.

4.3 GRADIENT mode

Format: `RUN GRADIENT` *var-set*
Output files: `shell.out` `grad.out` `timing.out`

This option calculates the gradient of Gibbs free energy analytically. *var-set* is a set of one or more keywords which specify which gradient elements are to be calculated as follows

var-set keyword	derivatives calculated with respect to
CELL	*lat-param* from the `LATTICE` line
BASIS	x, y, z from the `BASIS` block
INTERN	*strain-val* from the `INTERNAL` line(s)

At least one of these three strain types must be given, and `BASIS` and `INTERN` cannot be used together. If `CELL` and one of `BASIS` or `INTERN` is given, then an additional keyword `ZSISA` may be used. This indicates that the `CELL` gradient is to be derivatives of the full dynamic free energy, while the `BASIS` /`INTERN` gradient is derivatives of only the static part (if *x-mesh* $= 0$ however then `ZSISA` will have no effect).

The meanings of all the allowed *var-set* combinations are given explicitly in table 1. The keywords in *var-set* may be given in any order.

The selected derivatives are written to file `grad.out`. They are written `CELL` first, then `BASIS` /`INTERN`, each in the order in which they are defined in the

Figure 3: `shell.out` for command `shed -rTHERMO <examp.shl | shell`

```
Magnesium Fluorite rutile structure with shells
Temperature =  1000.000000    K
Pressure = 0.0000E+00 GPa
Mesh =   3   3   3
Heat Capacity =  147.3455370     J/(mol K)
Static energy = -4908.324615     kJ/mol
Zero point energy =  30.57291422     kJ/mol
Vibrational Energy =  151.9978001     kJ/mol
Vibrational Free Energy = -143.3949528     kJ/mol
Internal Energy = -4756.326815     kJ/mol
Helmholtz Energy = -5051.719568     kJ/mol
Gibbs Energy = -5051.719568     kJ/mol
Entropy =  295.3927529     J/(mol K)
Volume = 0.4171700544E-04 m**3/mol
Thermal Gruneisen parameters =   0.000000   0.000000   0.000000

Potentials
    EXPO  F    F       22400.000        0.192    0.20    8.00
    INVE  F    F          24.804        6        0.20    8.00
    SPRI  F    cF         17.496                 0.00    0.20
    EXPO  F    Mg       5971.700        0.213    0.20    8.00
    INVE  F    Mg         -8.960        6        0.20    8.00

Lattice parameters
    a =   4.660000000     A
    c =   3.190000000     A

Internal degrees
    0.3057000000    0.0004000000

Primitive vectors
    4.66000000    0.00000000    0.00000000
    0.00000000    4.66000000    0.00000000
    0.00000000    0.00000000    3.19000000

Particles
    F                0.000000       -1.137570
    cF              18.999840        0.233570
    Mg              24.305000        1.808000

Crystallographic basis
    Mg          0.0000000000  0.0000000000  0.0000000000
    Mg          0.5000000000  0.5000000000  0.5000000000
    cF          0.3057000000  0.3057000000  0.0000000000
    cF          0.6943000000  0.6943000000  0.0000000000
    cF          0.1943000000  0.8057000000  0.5000000000
    cF          0.8057000000  0.1943000000  0.5000000000
    F           0.3061000000  0.3061000000  0.0000000000
    F           0.6939000000  0.6939000000  0.0000000000
    F           0.1939000000  0.8061000000  0.5000000000
    F           0.8061000000  0.1939000000  0.5000000000
```

input file. They are written three to a line, except that the `BASIS /INTERN` ones always start on a new line.

4.4 `OPTIMISE` mode

Format:	`RUN OPTIMISE` *var-set opt-method*
Output files:	`shell.out timing.out`

An iterative procedure is used, starting from the geometry given in the input file, to minimise the free energy G (or G_{stat} if *x-mesh* $= 0$) with respect to the N_ϵ geometric variables specified in *var-set*, i.e. locating (locally) optimal unit cell geometry. The convergence condition is that the root mean squared of the derivatives calculated (as shown in table 1) is within a certain tolerance *opt-tol* of zero. *opt-tol* defaults to 0.01 but can be set using the `CONVERGE OPTION`.

The optimisation possibilities are therefore quite flexible. If a completely relaxed solution at a given applied pressure *press* is required then either '`CELL BASIS` ' or '`CELL INTERN`' should be specified, while if optimised internal strains at the constant volume determined by the *lat-param* variables is required then just `BASIS` or `INTERN` (though note this is not strictly a constant-volume optimisation except for cubic symmetries, since the shape as well as the volume of the unit cell is held constant).

Giving the keyword `ZSISA` in addition to `CELL` and `BASIS` or `INTERN` as shown in table 1 causes a full optimisation within the Zero Static Internal Stress Approximation to be carried out. Lacking analytic free energy derivatives, ZSISA is the approximation within which most existing lattice dynamical structure optimisation programs work by default. Note however that because of the way Shell works, this approximation is not any cheaper than a fully dynamic optimisation, so it will only be useful for comparison with other methods.

The even more extreme Constant Internal Strain Parameter Approximation (CISPA) can be used by doing a '`CELL INTERN`' optimisation at *x-mesh* $= 0$, then setting the *strain-val* parameters in the input file to their thus-optimised values, and doing `CELL`-only optimisations. An example of this is given in section 6.1, example 5.

If `BASIS` optimisation is chosen then the last listed particle specified in the `BASIS` section of the input file is held fixed.

4.4.1 Optimisation methods

By default a variable metric method algorithm with an initial static approximation to the hessian is used to perform the optimisation and this usually provides the fastest convergence. However, alternative algorithms may be selected by specifying the *opt-method* variable.

The various methods work by making use of the free energy G and its derivatives with respect to whatever set *var-set* of strains $\{\epsilon_i\}$ has been specified. The program can make direct analytic evaluations of the dynamic free energy G, the dynamic free energy gradient (first strain derivative vector) $\partial G/\partial\epsilon_i$, and the static part of the free energy hessian (second strain derivative matrix) $\partial^2 G_{\mathrm{stat}}/\partial\epsilon_i\partial\epsilon_j$. Since the static contribution to the free energy derivatives is usually the most important part, the static hessian can be used as an approximation to the full (dynamic) hessian, i.e.

$$\frac{\partial^2 G_{\mathrm{stat}}}{\partial\epsilon_i\partial\epsilon_j} \approx \frac{\partial^2 G}{\partial\epsilon_i\partial\epsilon_j}$$

If the *x-mesh* specified in the input file is zero, then this is no longer an approximation, but is exact.

If the first and second derivatives of G are available then a Newton method can be used to iterate from a given strain vector ϵ to an improved (closer to the free energy minimum) vector ϵ^+ as follows:

$$\epsilon_i^+ = \epsilon_i - \sum_j \frac{\partial^2 G}{\partial \epsilon_i \partial \epsilon_j} \frac{\partial G}{\partial \epsilon_j}$$

If only the first derivative is available then a conjugate gradient method can be used, in which one dimensional minimisations are carried out in a series of directions conjugate to those of previous iterations, to approach the minimum. The methods described below are on the whole variants of these two approaches. More information can be found, e.g., in *Numerical Recipes*.

Explanations of the various *opt-method* options are given in approximate order of usefulness, brief comments being given on applicability. Some are of mainly historical interest.

`VMM`: Variable Metric Method
: Initially the static hessian is calculated, and this is used as an approximation to the dynamic Hessian. On subsequent iterations the dynamic gradient is calculated and used both to update the approximate Hessian, and to iterate using a Newton algorithm towards the minimum.
This method is currently the default, and usually gives the most rapid convergence, but if the dynamic contributions to the hessian are large (e.g. at high temperatures) the initial step can sometimes be into a region distant from the minimum, leading to failure of the optimisation.

`NAG`: Numerical Algorithms Group conjugate gradient routine
: The routine `E04DGF` from the Numerical Algorithms Group FORTRAN 77 library is used for the minimisation. This is a pre-conditioned, limited memory quasi-Newton conjugate gradient method. It calculates a number of gradients (typically 1 or 2, but up to 10 if the optimisation is not going well) per iteration. A partial approximate Hessian is built up from scratch.
This method seems fairly robust and can be useful if the `VMM` method gets off to a bad start.
This option is only available on machines with the NAG library installed (see section 7).

`NEWT`: Newton method (static)
: At each iteration the static hessian and the dynamic gradient are calculated and a Newton step is made.
If *mesh-val* $= 0$ this is simply the Newton method, otherwise it is a sort of approximation to that method, and is unlikely to work well unless the dynamic contributions to the hessian are very small.

`1NNEWT`: One-variable numeric Newton method
: At each iteration three calculations of the free energy itself are calculated, which is enough to generate numerical approximations to the gradient and hessian if there is only one strain.
Can be fast because no derivatives are required, but convergence is sometimes poor because of numeric errors.

`NNEWT`: Semi-numeric Newton method
: At each iteration the dynamic hessian is calculated semi-numerically (by N_ϵ evaluations of the gradient) and a Newton step is made.
This will be very slow for large N_ϵ, and may converge poorly because of errors introduced in the numeric evaluation of the hessian, especially those due to potential cutoff.

`OLD`: Basic conjugate gradient method

At every iteration the analytic gradient is calculated and used to generate a new direction along which to minimise. This line minimisation is then done using only free energy evaluations, not the gradient.
This method is often very slow.

`NUM`: Numeric conjugate gradient method

At every iteration the gradient is calculated numerically and used to generate a new direction along which to minimuse. This line minimisation is then done using only free energy evaluations.
This method is very slow and has nothing to recommend it unless you want to avoid using the analytic derivative calculations for some reason.

`NAPACK`: NAPACK conjugate gradient routine

An implementation of the conjugate gradient method using the routine `NAPCG` from NETLIB's NAPACK library. At every iteration the analytic gradient is calculated and used to generate a new direction along which to minimise. This line minimisation is done using further analytic gradient evaluations.
Source is included with Shell and does not need to be installed separately.

Output to the screen (that is UNIX standard output) varies according to the *opt-method* chosen, but it is intended to give some indication of the progress of the optimisation. At least every iteration, something will be printed. For the methods `VMM`, `NEWT`, `1NNEWT` and `NNEWT` an initial message is printed describing the method to be used and at each iteration a summary line is written. The screen output from the file `examp.shl` is:

```
 PROGRAM SHELL 4.3
 VMM: Variable metric method using analytic static Hessian and dynamic gradient
 VMM: N =    4
   it         normX             normG                        func   minutes
    1      0.000000      237.68555463        -5051.719567773502      0.03
    2      0.034998       19.32592149        -5052.573602450151      0.06
    3      0.038911        0.54372692        -5052.608214875611      0.09
    4      0.038742        0.11647919        -5052.608233106312      0.12
    5      0.038742        0.00770701        -5052.608233156117      0.15
Optimisation successful in   5 iterations
```

The meaning of the columns is as follows:

`it` is the number of the iteration

`normX` is $\sqrt{\sum(\epsilon_i - \epsilon_i^0)^2}$, where ϵ_i^0 are the starting values

`normG` is $\sqrt{\sum(\partial G/\partial\epsilon_i)^2}$

`func` is G (the function being minimised)

`minutes` is the cumulative CPU time used

Most of the methods have a maximum number of iterations: if convergence is not reached after that many then the program is terminated.

If imaginary frequencies are encountered during the optimisation, the program will normally terminate, unless the `IMAGINARY` option has been specified. It can be worth doing this, since sometimes (though not very often) the optimisation is able to return from such unfavourable regions of the configuration space to achieve a physically realistic free energy minimum.

4.5 DISPERS mode

Format:	`RUN DISPERS` *qx qy qz num-points*
Output files:	`disper.out motion.out charge.out shell.out timing.out`

Note that the only (non-zero) data in `shell.out` will be the geometry and potentials.

The values *qx qy qz* specify a reciprocal lattice vector in reciprocal Cartesian space in units of Ångstrom^{-1}. At each of *num-points* equally spaced points along this vector (the Γ-point is excluded) the frequencies $\nu_{\mathbf{q}s}$ in units of THz are calculated and output to file `disper.out`. If any of the eigenvalues $(2\pi\nu_{\mathbf{q}s})^2$ is negative, then the corresponding (imaginary) frequency multiplied by $\surd$-1 is output, i.e. a negative value.

Additionally the eigenvectors in both unmodified form and multiplied by their corresponding charges (the latter useful for identifying IR active normal modes) are output in the files `motion.out` and `charge.out` respectively. The first column in each of these files gives the reduced wave vector, e.g. 0.25, 0.50, 0.75, 1.00 if *num-points* = 4.

4.6 GRUNEIS mode

Format:	`RUN GRUNEIS` *qx qy qz num-points*
Output files:	`gruneis.out disper.out motion.out charge.out shell.out timing.out`

This mode operates in exactly the same way as the `DISPERS` mode, but it additionally calculates the six macroscopic mode Grüneisen parameters

$$\gamma^{\lambda}_{\mathbf{q}s} = -\frac{\partial \ln \omega_{\mathbf{q}s}}{\partial \eta_\lambda}$$

for each mode s and reciprocal space vector $\mathbf{q} = (qx, qy, qz)$. These are written with the corresponding frequencies to the file `gruneis.out`.

This requires the calculation of the eigenvalue strain derivatives and so is slower than the simple `DISPERS` mode.

In the case that some states are degenerate, correct calculation of the *mode* Grüneisen parameters requires more careful handling of the perturbation theory than Shell currently performs. Currently the code simply prints a warning message to the effect that there may be errors in some of the mode Gruneisen parameters. Specifying slightly different *qx*, *qy*, *qz* is usually a satisfactory workaround. Note that this limitation does not affect any of the other quantities calculated by the program.

4.7 ELASTIC mode

Format:	`RUN ELASTIC` [*elast-options*]
Output files:	`elasti.out shell.out timing.out`

The available *elast-options* are

`ISOTHERMAL` If this is included then isothermal constants are calculated, otherwise adiabatic constants are calculated.

`NUMERIC / SEMIANALYTIC` If the `NUMERIC` option is given then the elastic constants are calculated by a double numerical differentiation of the free energy, which is very slow.
If the `SEMIANALYTIC` option is given then the elastic constants are calculated by single numerical differentiation of the analytic free energy gradient. This is much faster, but there is currently an error in this method; it seems to give results which are correct to about 2 significant figures for examples we've tried, but this can't be guaranteed.
If *x-mesh* = 0 then the static derivatives are in any case calculated entirely analytically, which is fast and accurate.

Shell will say what it's going to do (isothermal or adiabatic, and which method it's using) before it prints the output.

A generalised stiffness matrix $\mathcal{S}^T_{uv}$ (Hessian of the Helmholtz free energy with respect to strains) and compliance matrix $\mathcal{C}^T_{uv}$ (its inverse) are calculated

$$\mathcal{S}^T_{uv} = \left(\frac{\partial^2 F}{\partial \epsilon_u \partial \epsilon_v}\right)_T$$

$$\mathcal{C}^T_{uv} \quad = \quad \left(\mathcal{S}^T_{uv}\right)^{-1}$$

where the derivatives are over the whole set of general external strains and Cartesian ion position vectors $\epsilon_u = (\eta_\lambda, r_i^\alpha)$, with the exception of position vectors of one ion to account for the translational symmetry of the crystal.

Adiabatic elastic constants are generated using the relationship

$$\mathcal{S}^S_{uv} = \mathcal{S}^T_{uv} + \frac{TC_v}{V}\gamma_u\gamma_v$$

where γ_u is the generalised Grüneisen parameter

$$\gamma_u = \frac{1}{C_v}\frac{\partial S}{\partial \epsilon_u}$$

By default the adiabatic elastic constants are printed out, but by using the `ISOTHERMAL` option the isothermal ones can be printed instead.

The first 6×6 elements of $\mathcal{C}_{uv}$ are then inverted to give the macroscopic stiffness matrix, $c_{nn'}$.

4.8 TEST mode

Format: `RUN TEST` *var-set*
Output files: `shell.out timing.out`

In this mode the program calculates the gradients both by numerical differentiation of the free energy and directly using the analytic method. The numeric values are calculated as

$$\frac{\partial G}{\partial \epsilon_i} \approx \frac{G(\epsilon_i + h) - G(\epsilon_i - h)}{2h}$$

with a range of values of h so that convergence can be seen. *var-set* may be chosen as in table 1.

Output is to the screen (standard output); an example is as follows:

```
% shed -r 'TEST CELL INTERN' < examp.shl | shell
 PROGRAM SHELL 4.3
Gradient number    1
       27.04837017           28.20934211
      -23.70344873          -233.2491974

 Testing derivatives numerically

             27.04807926       (numeric, h = 0.1E-02)
             27.04836719       (numeric, h = 0.1E-03)
             27.04836975       (numeric, h = 0.1E-04)
             27.04837016       (numeric, h = 0.1E-05)
             27.04832241       (numeric, h = 0.1E-06)
             27.04832696       (numeric, h = 0.1E-07)
----------------------------------------------------
       1     27.04837017       (analytic)
----------------------------------------------------
             28.20898461       (numeric, h = 0.1E-02)
             28.20933851       (numeric, h = 0.1E-03)
             28.20934260       (numeric, h = 0.1E-04)
             28.20933605       (numeric, h = 0.1E-05)
             28.20935606       (numeric, h = 0.1E-06)
             28.20906957       (numeric, h = 0.1E-07)
----------------------------------------------------
       2     28.20934211       (analytic)
----------------------------------------------------
            -23.37487765       (numeric, h = 0.1E-02)
            -23.70016310       (numeric, h = 0.1E-03)
            -23.70341522       (numeric, h = 0.1E-04)
```

```
            -23.70344737       (numeric, h = 0.1E-05)
            -23.70347829       (numeric, h = 0.1E-06)
            -23.70384209       (numeric, h = 0.1E-07)
-------------------------------------------------------
       3    -23.70344873       (analytic)
-------------------------------------------------------
            -232.7846775       (numeric, h = 0.1E-02)
            -233.2445520       (numeric, h = 0.1E-03)
            -233.2491506       (numeric, h = 0.1E-04)
            -233.2491931       (numeric, h = 0.1E-05)
            -233.2492340       (numeric, h = 0.1E-06)
            -233.2492841       (numeric, h = 0.1E-07)
-------------------------------------------------------
       4    -233.2491974       (analytic)
-------------------------------------------------------
 Timings:
     Routine AGIBBS:  Total   35.630,  Per call    0.727
     Routine DAGIBS:  Total    1.836,  Per call    1.836
```

4.9 BENCH mode

Format: RUN BENCH [*var-set*]
Output files: shell.out timing.out

This mode is provided for timing ('benchmarking') the performance of the code in each of its principal types of calculation. It presents an easy to understand summary of the time the code takes to calculate the static and dynamic free energy and their derivatives for a given structure. The data presented is less detailed however than that given in the file timing.out. It may be useful to give an idea of how performance scales for different sized systems, or between different machines, or when varying parameters of the codes operation like *eta-factor*. Output is fairly self-explanatory: the final line is a summary giving all the times as well as recording *x-mesh* (MESH), *num-particles* (NSL) and the number of the particles which are shells (Nsh), these being the principal values which affect the timings.

If the optional *var-set* list is omitted then a default value of 'CELL BASIS' is assumed.

An example output (run on a 133MHz Pentium PC running Linux, compiled using g77) is as follows:

```
% shed -rBENCH <examp.shl | shell
 PROGRAM SHELL 4.3
            Static energy     0.00  minutes
          Static gradient     0.01  minutes
           Static hessian     0.03  minutes
           Dynamic energy     0.10  minutes
         Dynamic gradient     0.29  minutes

 NSL Nsh Mesh       U_st  Grad_st  Hess_st     F_dy  Grad_dy       Total
  10   4    3        0 %      1 %      7 %     23 %     65 %       100 %
  10   4    3       0.00     0.01     0.03     0.10     0.29        0.43
```

5 Notes and tips

5.1 Analysis of results

Symmetry of dynamical results The algorithm for choosing the grid of reciprocal lattice vectors for the Brillouin zone integration does not in general take account of the symmetry of the unit cell. For this reason dynamical results can fail to observe the symmetry relations which would be expected: for instance a BASIS optimisation with *x-mesh* = 2 can give a different geometry from an

INTERN optimisation which constrains the symmetry. At $x\text{-}mesh = 0$, and as $x\text{-}mesh \to \infty$, the two will give the same results, as expected.

5.2 Three-body interactions

There are a couple of peculiarities about the evaluation of the three body contributions to the free energy and its derivatives.

Collinear triples If the angle θ is equal or very close to 180° or 0° then the code will try to perform a divide by zero operation during evaluation of the vibrational contribution to the free energy, or any derivative of the static energy, when using the HAR3 or HEX3 interactions. This will probably result in termination of the program, although this is platform-dependent and may simply result in garbage results. This is not due to the functions in question becoming physically unstable, but is an artifact of the method of evaluation of the derivatives of the three-body potentials. In the current version of Shell there is no way to avoid this problem except by not defining three-body interactions for triples which are (or may become) collinear.

Neighbour lists When it starts up, for reasons of efficiency, the code sets up a list of triples on which three-body potentials will operate. This list is not regenerated later in the run, even if the geometry has changed (i.e. on subsequent iterations of an optimization run). Thus it is possible for an interaction to continue to operate on a triple even though the angle or distance ranges specified in the input file are no longer satisfied, if the triple has moved outside of the specified range as a result of the change of geometry brought about by the optimization (the opposite effect, of a triple moving into range without the potential being used, can also occur). This is unlikely to be a serious problem, and will in fact improve the performance of the optimization, since if particles are moving into and out of the ranges of interactions there is probably trouble with the potentials anyway, but it may lead to unexpected behaviour.

6 Utility programs

To make Shell more productive and easier to use, a few external utility programs have been written to manipulate the Shell input and output files. These are on the whole written in Perl 5, a freely available scripting language. Some of them might work for earlier versions of Perl, either with or without modification, but no guarantees. The philosophy of these programs is to provide useful facilities which are more easily done in a scripting language than in FORTRAN, so as to increase the functionality without cluttering up the source code of Shell.

In the following subsections each utility is described and examples are given of its use. The descriptions and examples assume a UNIX environment and understanding of terms like *standard input*, *standard output* and *pipe*. If you don't understand these terms (and you are using UNIX) then if you just look at the examples you can probably get the idea.

These utilities are self-documenting, in that giving the `-h` option will cause them to print a usage message and exit. This usage message is shown for reference in the following subsections.

Most of the utilities take one-letter command line flags. Where these are followed by an argument, whitespace between the flag and the argument is optional. These flags cannot be combined behind a single minus sign, so that, e.g. `bond -ts` is NOT equivalent to `bond -t -s`.

Note: These programs are *reasonably* robust, but were rather more sloppily written, tested and documented than Shell itself, so there may be anomalies in their operation. In particular, strange combinations of flags, or feeding them unexpected input, may cause them to misbehave.

6.1 shed: input file manipulator

Running `shed -h` gives this usage message:

```
Usage:  shed [options] < old-shell-input-file > new-shell-input-file
  options:
    -h                  print this message
    -s <shell-out-file>  use external and internal strains from file
    -e <shell-out-file>  use external strains (lattice params) from file
    -i <shell-out-file>  use internal strains from file
    -b <shell-out-file>  use basis strains from file
    -E <lattice-params>  set lattice params to <lat-params> (space separated)
    -I <int-strains>     set internal strains to <int-strains> (spc separated)
    -z                  set all internal strains to zero
    -m <mesh>           use <mesh> mesh points
    -t <temp>           set temperature to <temp>
    -p <press>          set pressure to <press>
    -r <run-type>       do a run of type <run-type>
    -o <opt-line>       use options <opt-line>
    -x <t1>:<t2>        exchange atoms type <t1> -> <t2> in basis
    -X <s1>:<s2>        exchange string <s1> -> <s2> throughout
    -v                  print all changes as made (verbose output)
```

`shed` (Shell file editor) is designed for manipulating Shell input files so that it is not necessary to enter an editor every time some modification to the run needs to be done. It's a bit like the UNIX `sed` command, but tailored to the needs of Shell input files. This should make running Shell both from the command line and from shell scripts much less fiddly.

The operation of `shed` is as follows: it reads a Shell input file line by line on standard input, and for each line examines the flags supplied on the command line. According to these flags, it may make modifications to the line. It then writes the modified or unmodified line to standard output. If the `-v` (verbose) flag is supplied then a copy of every modified line will be written to standard error as well. Using the the `-v` flag is therefore a good idea to check that `shed` is doing what you think it is.

Useful things to do with `shed` are therefore

- construct a new input file from an old one:
  `shed -t 500 <examp.shl >newexamp.shl`
- run Shell using an existing input file with some modifications:
  `shed -t 500 <examp.shl | shell`

and some useless ones would be

- use it as a null filter:
  `cat examp.shl | shed | lpr`
- make a long pipeline of them:
  `cat examp.shl | shed -t500 | shed -m1 | shed -p15 >newexamp.shl`
  (since that could be done with a single command combining all the options).

Each option flag (except `-h`, `-z`, `-v`, which do not take arguments) is followed by a single command line argument. If the argument has spaces in it therefore, it must be enclosed in single or double quotes to prevent the UNIX shell from breaking it into several words.

The meaning of the flags should be reasonably self-explanatory from the output of `shed -h`. Where it is not, playing around with it (especially using the `-v` flag as in example 6.1 below) should make things clear. Instead of explaining each option in detail, we give some examples.

1. **Run Shell using a slightly modified input file**

   ```
   shed -m5 -t1500 -r'TEST CELL' <examp.shl | shell
   ```

This command line runs Shell according to the input file `examp.shl` except that the lines `MESH` 5, `TEMP` 1500 and `RUN TEST CELL` are substituted for the original `MESH`, `TEMP` and `RUN` lines.
The quotes in the `-r'TEST CELL'` part are required to prevent the UNIX shell from interpreting the space between the two keywords as breaking them into two arguments of `shed`.

2. See what changes a `shed` command would make to a file

```
shed -v -s shell.out -p50 <examp.shl >/dev/null
```

This command goes through the file `examp.shl`, making the changes indicated by the supplied flags (taking the *lat-param* and *strain-val* geometry values from `shell.out` and setting *press* to 50 GPa), and discards the resulting shell input file (writes it to the device `/dev/null`). However, the `-v` flag causes it to write the changes it has made to the screen. Routine use of the `-v` flag is a good idea to check that `shed` is doing what you think it is.
A more direct, though less readable, way of seeing the changes made by `shed` would be

```
shed -s shell.out -p50 <examp.shl | diff examp.shl -
```

3. Run Shell at a series of temperatures

```
#!/bin/sh
for temp in 500 1000 1500
do
  shed -r THERMO -t $temp <examp.shl | shell
  cat shell.out >> examp.log
done
```

This is a short Bourne shell script to run shell for each of three temperatures and append the output file of each to a log file.

4. Optimisation in steps

```
shed -r'OPTIM CELL INTERN' -m0 <examp.shl | shell
shed -r'OPTIM CELL INTERN' -m2 -s shell.out <examp.shl | shell
shed -r'OPTIM CELL INTERN' -m4 -s shell.out <examp.shl | shell
```

In the first line a static optimisation is done, which is very fast, saving the statically optimised structure in file `shell.out`. In the second line the *strain-val* and *lat-param* (internal and external strain) values are extracted from `shell.out` and used as a starting point for a slower, but more accurate finite temperature optimisation with *x-mesh* $= 2$. Finally that optimised structure is used as a starting point for a much slower, but even more accurate, optimisation with *x-mesh* $= 4$.
Since starting near the optimised value usually reduces the number of iterations required, this multiple step approach can be an efficient way of reaching a dynamically optimised state. Two steps (one static and one dynamic) are often enough.
In a loop of optimisations it is usually a good idea to use the previous state as a starting point for the next run, e.g.:

```
#!/bin/sh
shed -v -m0 -rTHERMO <examp.shl | shell
for press in 0 5 10 15
do
  shed -v -p $press -m2 -r'OPTI CELL INTER' -s shell.out \
     <examp.shl | shell
  cat shell.out >>examp_pressure.log
done
```

5. CISPA approximation

```
#!/bin/sh
shed -v -r'OPTIM CELL INTERN' -m0 -t0 <examp.shl | shell
cp shell.out examp_static.out
```

Figure 4: Output of `toev <shell.out` for the file in figure 3

```
Magnesium Fluorite rutile structure with shells
Temperature  = 1000.000000 K
Pressure  = 0.0000E+00 GPa
Mesh =   3   3   3
Heat Capacity  = 0.00152712924427472 eV/K
Static energy  = -50.871212067726 eV
Zero point energy  = 0.316866003128853 eV
Vibrational Energy  = 1.57534656511607 eV
Vibrational Free Energy  = -1.48618431450878 eV
Internal Energy  = -49.2958655036463 eV
Helmholtz Energy  = -52.3573963843076 eV
Gibbs Energy  = -52.3573963843076 eV
Entropy  = 0.00306153087962485 eV/K
Volume  = 69.272764000857 A**3
Thermal Gruneisen parameters =   0.000000   0.000000   0.000000
```

```
for temp in 500 600 700 800
do
  shed -v -r'OPTIM CELL' -m3 -t $temp \
       -i examp_static.out -e shell.out \
           <examp.shl | shell
  cat shell.out >>examp_cispa.log
done
```

Here a static optimisation is done, and the optimised geometry saved in file `examp_static.out`. Then for each of a set of temperatures an optimisation is done in which the lattice parameters *lat-param* are varied but the internal strains *strain-val* are held fixed to their starting values, which are taken from the statically optimised geometry in `examp_static.out`. For efficiency, as in example 6.1 the initial *lat-param* values are taken from the previous calculation.

6.2 `toev`: output file unit convertor

Running `toev -h` gives this usage message:

```
Usage:  toev [-f] [ <shell-output-file> ]
  -f    full file not just <quant> = <value> lines
```

`toev` is a very simple utility to convert the quantities output in the `shell.out` file from kJ-mol units into eV-Å units. It simply reads the Shell output specified (or from standard input if none is given on the command line) and writes to standard output the block of calculated quantities, but in different units. It doesn't repeat the geometrical information unless the `-f` flag is used.

If the file was generated from a `SLAB` geometry it will also work out the area of the unit cell in $Å^2$.

Figure 4 shows an example output.

6.3 `intern`: internal strain coordinate guesser

Running `intern -h` gives this usage message:

```
Usage:  intern [-q(uiet) | -v(erbose)] [ <shell-input-file> ]
```

`intern` is a utility designed to assist in constructing Shell input files with `INTERNAL` degrees of freedom. If one has a file (unit cell) containing many particles, but with considerable symmetry, it may be convenient to be able to specify the geometry (internal strains) in terms of a smaller number of *strain-val* values than the entire set of $3N$ (x, y, z) strains. This can be a good idea both in order to understand the geometry of the unit cell better, and to improve the efficiency of the code (optimising with a smaller number of strains may be more efficient than with a large number, although this isn't necessarily the case since certain optimisations can be made if the whole set of (x, y, z) strains are to be calculated). Constructing a file defining the `INTERNAL` strains by hand can be tedious and error-prone.

The command reads the Shell input file specified, (or from standard input if none is given on the command line) and writes a modified input file on standard output. By default it also writes some progress information to standard error, but this can be suppressed by the `-q` flag.

`intern` takes an existing input file and attempts to guess a set of `INTERNAL` strains which will fully characterise the geometrical configurations which are allowable by symmetry. It does this in rather a crude way: it calculates the static `BASIS` gradient for the supplied structure, and groups together elements of the gradient which are 'nearly' equal to each other. Each of these groups it assumes to correspond to a single `INTERNAL` strain and it constructs an `INTERNAL` *strain-val* stanza (with *strain-val* $= 0$) with lines corresponding to each of the grouped basis coordinates (each entry will be 0 or ± 1). Each of these `INTERNAL` stanzas is then appended to the input file. The process is then repeated with the newly generated input file, but with each of the *strain-val* values nudged slightly off zero; this repeated cycle may generate more strains if the initial state was in a specially symmetric configuration. This cycle is continued until an iteration does not reveal any more symmetric strains than the previous one.

Clearly, this procedure is not guaranteed to generate either a full, or a minimal, set of internal strains; in particular, its success will depend on the tolerance of 'nearness' below which two gradient elements are identified as the same, and the size of 'nudge' from symmetric positions between iterations (both these parameters can easily be changed by examining the script). It is a good idea to check the results, e.g. by a sequence of commands such as the following:

```
intern <old.shl >new.shl
shed -m0 -r'OPTIM CELL INTERN' <new.shl | shell
shed -m0 -r'GRAD CELL BASIS' -s shell.out <new.shl | shell
```

and examining the file `grad.out` to check that all the elements are (nearly) zero. If not, then the set of `INTERNAL` strains generated by `intern` was not sufficiently general.

To summarize: `intern` can be a useful tool, but should not be trusted implicitly. In practice however, we've found it to work pretty well.

6.4 `bond`: structure bond-length and angle calculator

Running `bond -h` gives this message:

```
Usage:  bond <options> [ <shell-out-file> ]
  options:
    -h               print this message
    -o <out-mode>    2 for lengths, 3 for angles, 23 for both (default 23)
    -b [<min>:]<max> bondlength range in Angstrom (default 0.2:2)
    -t <types>       s for shells, c for cores, sc for both (default c)
    -n               give particle numbers only, not types
    -x               exclude right angles
```

This utility summarises the structure of a crystal from the shell output file (or from standard input if none is given on the command line) in terms of bond lengths (i.e. distances between pairs of particles) and angles between triples of particles. A range of distances is specified using the `-b` flag, either by specifying `-b bmax` or `-b bmin:bmax`, where `bmax` and `bmin` are given in Ångstrom. If either part is omitted the defaults as shown in the usage message are used.

The other command line options should be fairly self explanatory. By way of example: output of the command

```
bond -b 2.1 -x shell.out
```

using the `shell.out` from figure 3 is as follows:

```
Bonds:
                 i           j                R_ij
                 -           -                ----
                 1 -  Mg     4 -  cF      2.0146349
                 1 -  Mg     3 -  cF      2.0146349
                 2 -  Mg     6 -  cF      2.0146349
                 2 -  Mg     5 -  cF      2.0146349
                 1 -  Mg     5 -  cF      2.0453999
                 2 -  Mg     4 -  cF      2.0453999
                 2 -  Mg     3 -  cF      2.0453999
                 1 -  Mg     6 -  cF      2.0453999

Angles:
   i           j           k          Angle j/i\k         R_ij          R_ik
   -           -           -          -----------         ----          ----
   2 -  Mg     3 -  cF     4 -  cF      77.515723      2.0453999     2.0453999
   1 -  Mg     5 -  cF     6 -  cF      77.515723      2.0453999     2.0453999
   3 -  cF     1 -  Mg     2 -  Mg     128.757862      2.0146349     2.0453999
   5 -  cF     1 -  Mg     2 -  Mg     128.757862      2.0453999     2.0146349
   6 -  cF     1 -  Mg     2 -  Mg     128.757862      2.0453999     2.0146349
   4 -  cF     1 -  Mg     2 -  Mg     128.757862      2.0146349     2.0453999
```

6.5 vrml: 3d graphics file generator

Running `vrml -h` gives this message

```
Usage:  vrml <options> [ <shell-out-file> ]
  options:
    -h                print this message
    -b [<min>:]<max>  draw bonds between atoms separated by range in Angstrom
    -i <factor>       surround UC with images up to <factor> lattice params
    -l <label>        either n[umber] or t[ype] text in sphere (default none)
    -t <types>        s for shells, c for cores, sc for both (default c)
    -s <transp>       spheres have transparency of <transp> (0-1)
    -a                print out coordinates (Cartesian, Angstrom) not vrml
```

It reads the Shell output file specified (or from standard input if none is given on the command line) and writes to standard output a VRML 1.0 (Virtual Reality Modelling Language) file giving a three-dimensional description of the unit cell. These files are conventionally named with a `.wrl` extension, and have a MIME type of x-world/x-vrml. They can be viewed with a specialised viewer† or with some versions of general purpose WWW browsers. Viewing a structure in this way is not essential, but can give a good feel for the geometry of a unit cell, particularly by using the `-l` option in conjunction with the `bond` utility described in section 6.4.

The unit cell is generated as a set of spheres bounded by a wire frame representing the shape of the unit cell. The colours and sizes of the particles are arbitrarily chosen, but characteristics of named particle types can easily be given by editing the `vrml` script. The meaning of the flags is as follows:

`-b [<min>:]<max>`
: If this parameter is specfied then a 'bond' (black line) is drawn between any pair of particles separated by a distance which is less than `<max>` Å and greater than `<min>` Å (`<min>` defaults to zero if not supplied). Otherwise the particles (spheres) only are plotted.

`-i <factor>`
: Normally each particle in the output file is plotted once. Using this flag however

†There is a list of available VRML viewers at `http://www.sdsc.edu/vrml/`.

it is possible to plot images of the unit cell around the outside. If `factor` is, say, 1.01 then a particle at the origin of the unit cell will be plotted at each corner, rather than only once. If `factor` is 3, then $3^3 = 27$ unit cells will be plotted, with the basic one in the middle.

The basic unit cell is marked in two ways: firstly it is bounded by a white wire-frame parallelepiped, and secondly only the particles in the basic unit cell are represented by brightly coloured spheres. The image particles are represented by spheres in similar colours, but of reduced intensity.

In the case of `SLAB` geometry images of the unit cell are only reproduced in the two horizontal directions.

Values of `factor` < 1 are not interpreted meaningfully.

`-l <label>`

If this flag is used then as well as the sphere itself, a label is written inside the sphere identifying the type of atom, giving either its order of appearance in the `BASIS` block of the input file (if `-l number` is used) or its *type-name* (if `-l type` is used). Because the label is written inside the sphere, it might be invisible if your VRML browser doesn't support transparent rendering. Using a wire-frame representation might help.

`-t <types>`

Normally only the cores are drawn, but shells also can be selected by specifying `-t sc` or shells only with `-t s`. Normallly the shell displacement is so small that it's pretty tricky to distinguish shell from core if both are drawn.

`-a`

This option doesn't generate aVRML vile at all, but just spits out the Cartesian coordinates of each particle in units of Angstrom. Note this is not the same as the values in the file `shell.out` since those are in unit cell space. The Cartesian coordinates can be useful, amongst other things, as inputs into visualisation packages such as RASMOL.

An example VRML file `examp.wrl`, the result of running

```
vrml -b2.1 -lt -i1.01
```

on the file in figure 3, is provided with the distribution.

6.6 `plotpot`: potential function plotter

Running `plotpot -h` gives this message:

```
Usage:  plotpot [ <options> ] <shell-input-files>
  options:
    -h                   prints this message
    -x <xmin>:<xmax>     xrange in units of Angstrom.  [1:3] is default.
    -y <ymin>:<ymax>     yrange in units of eV.
    -t <terminal-type>   as in gnuplot, eg. x11, postscript, table.
                               x11 is default
    -p <plot-type>       s for shortrange, c for coulomb
                               a for short+coulomb or a combination
                               like sc for both.  's' is default
```

This utility generates a gnuplot[‡] input file to plot the pairwise short-range and/or Coulomb potentials specified in one or more Shell input files. The utility isn't essential, but can be handy for visual comparison of alternative sets of short-ranged potentials.

The flags should be reasonably self-explanatory from the help message. It will typically be used in a pipeline, e.g.

```
plotpot -x1:2 pots1.shl pots2.shl | gnuplot
```

or

[‡]Gnuplot is a freely available command-driven plotting program available for many platforms. The official distribution site is `ftp://ftp.dartmouth.edu/pub/gnuplot/`.

```
plotpot -x1:2 -tpostscript pots1.shl pots2.shl | gnuplot >pots.ps
```

Three-body potentials are not handled at all.

6.7 `funcof`: print dependence of one variable on another

Funcof is a simple Bourne shell (`/bin/sh`) script which reads a Shell input file on standard input and runs Shell multiple times, modifying the input using `shed` each time, to display just one (Shell output) value as the function of another (Shell input). It would be useful for a preliminary look at a variable dependency, or a quick evaluation of some derivative-based quantity, but since it discards most of the data would not be appropriate for serious calculations.

Running `funcof -h` gives this message:

```
Usage: funcof [-v] [-t] <egrepstring> <paramtype> <value> ...
        -v       verbose output
        -t       filter shell.out through toev before egrep
```

Here `<grepstring>` is a regular expression identifying the file `shell.out` (filtered through `toev` if the `-t` flag is specified), `<paramtype>` is one of the flags to `shed` (see section 6.1) and `<value>` is a list of one or more values to supply to that flag.

For example, the following command shows the effect on the unit cell volume of changing the *x-mesh*:

```
% funcof Volume -m 0 1 2 3 4 5 6 <examp.shl
      -m 0  Volume = 0.3957030972E-04 m**3/mol
      -m 1  Volume = 0.4057376001E-04 m**3/mol
      -m 2  Volume = 0.4089172153E-04 m**3/mol
      -m 3  Volume = 0.4090018009E-04 m**3/mol
      -m 4  Volume = 0.4090162240E-04 m**3/mol
      -m 5  Volume = 0.4090185519E-04 m**3/mol
      -m 6  Volume = 0.4090200914E-04 m**3/mol
```

which gives an indication that if you want four significant figures in volume evaluations, the right *x-mesh* to use would be about 3. Note that the input file must be prepared appropriately for this to work: in this example it is important that the *run-mode* in `examp.shl` is `OPTIMISE`.

`funcof` is quite primitive, and provided more as an example of how to write a shell script using `shed` to run Shell than as a useful tool.

7 Installation

Shell is distributed as a tar file, and will be called something like `shell.tar` or `shell-4.3.tar`. If you have a binary distribution (no source code) it will be called `binshell.tar` or `binshell-4.3.tar`. If the file has the extension `.tar.gz` or `.tgz` then first uncompress it using `gzip -d`. How to install then depends on which distribution you have.

7.1 Source distribution

This contains the source code with a makefile, some documentation, and the utility programs described in section 6. Some effort has gone into writing the code in standard FORTRAN 77, and although this hasn't been perfectly observed we hope that compilation shouldn't present too many problems.

With luck only the following steps should be required:

1. **Untar the tar file**
   Put the tar file in the directory where you want the Shell files; the rest of this

documentation will assume that this is done in a directory called `shell`. Then run `tar` with extract flags:

```
mkdir shell
cp shell.tar shell
cd shell
tar xvf shell.tar
```

This will unpack the files into three directories: source code in `shell/src`, executables (Shell binary after compilation, and utility programs) in `shell/bin` and documentation in `shell/doc`.

2. **Edit the `makefile`**
Change directory to `shell/src` and edit `makefile`. Hopefully this should just be a matter of identifying a block of lines appropriate for the system in question and uncommenting them (removing the `#` characters from the start of the lines, as well as commenting out any uncommented block which does not apply). Changing the optimisation flags can obviously make a big difference to speed.
The code makes use of one NAG routine, for the `NAG` optimisation method, if the NAG FORTRAN 77 libraries are available. The makefile should be modified as per the comments according to whether this is the case or not. If the NAG libraries are not available it just means that this optimisation method cannot be used, which isn't a major disadvantage (see section 4.4.1).
It may be necessary to check the documentation for your local system to get the code working optimally.

3. **Edit header files**
Edit `paramete.h` in directory `shell/src` to set `MAXNSL` to the maximum number of particles (cores plus shells) in a unit cell which the code will be used to consider. This will allocate something like $1.6 \times$ $\texttt{MAXNSL}^2$ kilobytes of RAM in arrays. If a dynamic gradient is being calculated, most of this will need to be resident at the same time (or page thrashing will result). The effect of using a larger value of `MAXNSL` than required for a given run will depend on the details of the system in question.
In the event that you want to use values of physical constants other than the CODATA 1986 values, you can edit `physical.h` as well.

4. **Run `make`**
In `shell/src` run `make`. This will build the executable `shell/src/shell` and also a symbolic link to it from `shell/bin/shell` if that file does not already exist, so that adding `shell/bin` to your path will grant access to the executable as well as the utilities.

5. **Set `perl` path in utility files if neccesary**
The utility scripts in `shell/bin` all start with a line something like `#!/usr/bin/perl`. If that is the path to Perl 5 on your system then no further action is required (`perl -v` will tell you the version number). If you have Perl 5 but it exists in a different place, then each of these lines must be replaced by the correct path. You can do this by setting the `PERL5=` line in `shell/src/makefile` to the correct path and running `make utils` (or just run `make utils PERL5=<path>`).
If you don't have Perl 5 then either install it[§], or resign yourself to not using the scripts. In fact a full installation of Perl is not *required*, none of the external modules are used by these scripts, so simply having the Perl binary is sufficient.

The executable Shell can of course be remade at a later date, either with modified source code or with a different value of `MAXNSL` or the physical constants, simply by performing steps 3 and 4 again.

7.2 Binary distribution

The tar file contains documentation and Shell executables compiled for the target system. All that is required is to unpack the tar file in the directory where you want the Shell files; the rest of this documentation assumes that this is done in a directory called `shell`. Create and enter such a directory and run `tar` with extract flags:

[§] Source for Perl 5 can be found at `http://www.perl.com/CPAN/src/5.0/`.

```
mkdir shell
cp binshell.tar shell
cd shell
tar xvf shell.tar
```

This will unpack the files into two directories: executables (Shell binaries and utility programs) in `shell/bin` and documentation in `shell/doc`. In `shell/bin` are some executable files called `shell.`n; each is compiled with the parameter `MAXNSL` equal to n. `MAXNSL` is the largest number of particles (cores + shells) per unit cell which that instance of the code can consider. Runs should be done using the smallest value of n which is large enough for the unit cell under consideration (an error is generated if you try to use n too small).

If one system size is always used you might want to make a symbolic link

```
cd shell/bin
ln -s shell.32 ./shell
```

If none of the `shell.`n executables is appropriate then apply to whoever has the source distribution to compile you another one.

If the Perl scripts fail because Perl 5 is not installed in the right place you might need to perform step 5 in section 7.1.

8 References

This document is a manual for use of the program Shell and does not seek to give details of the theory used. The following documents provide some of this background.

Free energy derivatives and structure optimisation within quasiharmonic lattice dynamics
M.B.Taylor, G.D.Barrera, N.L.Allan and T.H.K.Barron, Phys. Rev. B 56, 14380 (1997)
This is the principal reference for the theory used in Shell. It presents the general theory for calculating the free energy and its derivatives, discusses methods of structure optimisation in general and the program's default (`VMM`) method in particular, and gives detailed expressions for lattice sums used in treating the short-ranged and Coulombic pairwise interactions.
An unpublished appendix `standard.ps` to this paper is supplied with the Shell distribution; it contains some discussion of theory not sufficiently novel to be published, but which may be a useful reference.

Quasiharmonic free energy and derivatives for three-body interactions
M.B.Taylor, N.L.Allan, J.A.O.Bruno and G.D.Barrera (submitted)
This paper extends the theory to treat three-body short ranged interactions. Detailed expressions for the lattice sums are presented, and results of applying the theory to α-quartz are given.

Shell: a code for lattice dynamics and structure optimisation of ionic crystals
M.B.Taylor, G.D.Barrera, N.L.Allan and T.H.K.Barron, Comp. Phys. Commun. (in press) (1998)
This is the primary (soon-to-be) published reference for the program itself. The program's capabilities and computational requirements, including scaling for large system sizes, are discussed, the method is put into context, and example results for rutile MgF_2 are given.

Other papers, including several using Shell for particular applications, can be found on the Shell WWW page (see section 1).