The `cif2cell` manual

Torbjörn Björkman

November 6, 2013

Contents

Introduction – Of CIF's and Cells

An electronic structure program needs atomic positions and, at least in the case of band structure programs, unit cell vectors to calculate the electronic structure of a crystal. Experimental data on the other hand tends to be given as a space group and a set of irreducible, or wyckoff positions, and are often distributed in the form of CIF files. There is a wide range of sources of CIF files — it is in fact the standard means of communicating crystallographic data for virtually everyone outside of the electronic structure community! Yet far from all electronic structure programs can read the CIF format, and from this circumstance a small set of convenience scripts one day turned into the program `cif2cell`.

What the program does is simply to take the crystal structure information in a CIF file, typically space group information and a set of irreducible coordinates, and then set up a calculation cell which is then output to your favourite electronic structure program (ESP). As you are probably aware, the choice of calculation cell for a given crystal is arbitrary and infinitely many possibilities exist for any given crystal. Perhaps the simplest example is when a problem requires us to set up a supercell, which is just describing the same crystal in a different cell than the primitive one, but even the primitive cell can often be chosen in many different ways. `cif2cell` contains working solutions for choices of primitive cells for (almost) any possible crystal and will make a choice automatically. You may choose either the primitive cell for the given structure, such as the rhombohedral cell with one atom for fcc Cu, or the conventional cell, a cubic cell with four atoms for fcc Cu. These cells can then be used as building blocks for generating supercells by multiplying them up and/or inputting extra vacuum in the structure.

This manual is intended as a document for computational physicists and materials scientists. The problems that you solve when generating a computational cell from crystallographic data belongs to the field of crystallography and the program is not intended to replace a working knowledge of basic crystallography. Remember this, and refer to your textbook in crystallography or solid state physics/chemistry when you get confused and your life with `cif2cell` will be happy.

The code was published in Computer Physics Communications **182**, 1183–1186 (2011), please cite generously. Happy computing!

Torbjörn Björkman

1 Installation

1.1 Requirements

`cif2cell` requires Python 2.4 or higher and the PyCIFRW python package[1]. Note however that the output may be slightly different (but equivalent) with Python 2.4 than with later versions due to differences in the internal sorting routines of different Python distributions.

To install the program in your systems standard location, type:

```
python setup.py install
```

To choose a different location, add

```
--prefix=where/you/want/it
```

to the above line. For help and more options type

```
python setup.py --help
```

2 Getting started

This section is meant to give you a quick overview of how you generate cells to get you started as quickly as possible. Running `cif2cell` is quick, so the best way to learning to work with it is to simply run it over and over, testing different settings and options until you have what you want. The program comes with a small set of sample CIF files which can be found either in the source distribution or in the directory `[prefix]/lib/cif2cell/sample_cifs`, where `[prefix]` is the path where you installed `cif2cell`. The examples below assume that you stand in a directory containing these sample files. In the following, some arbitrary command line input will be specified within brackets `[like this]`. You are encouraged to run the tests and play around as you read.

2.1 Help!

The quickest way to get help in `cif2cell` is to type

```
cif2cell -h
```

This will list all available input options along with a description.

2.2 The first run

There is only one required piece of input, and that is the CIF file itself which is given either as an argument to the program:

```
cif2cell [someoptions] Si.cif [otheroptions]
```

[1] Available from `http://pycifrw.berlios.de`.

and if no output program is specified, the cell information is output to screen:

```
CIF2CELL 0.4.0
2011-03-25 16:35
Output for Si (Silicon)
CIF file exported from Inorganic Crystal Structure Database.
Database reference code: 51688.

 BIBLIOGRAPHIC INFORMATION
Toebbens, D.M. et al., Materials Science Forum 378, 288-293 (2001)

 SYMMETRY INFORMATION
Cubic crystal system.
Space group number     : 227
Hall symbol            : F 4d 2 3 -1d
Hermann-Mauguin symbol : Fd-3mS

 INPUT CELL INFORMATION
Lattice parameters:
          a           b           c
  5.4305300   5.4305300   5.4305300
      alpha        beta       gamma
 90.0000000  90.0000000  90.0000000
Representative sites :
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000

 OUTPUT CELL INFORMATION
Bravais lattice vectors :
  0.5000000   0.5000000   0.0000000
  0.5000000   0.0000000   0.5000000
  0.0000000   0.5000000   0.5000000
All sites, (lattice coordinates):
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000
Si      0.2500000   0.2500000   0.2500000
```

First there is some information about the program itself and a description of the compound derived from information in the CIF file. If the file comes from a databases known to `cif2cell`, information about that is also printed. Then follows any bibliographic information found in the file. Then starts the actual crystal information with data about the space group. If the `-v` or `--print-symmetry-operations` flags are given, all symmetry operations will also be printed here. Next comes the lattice parameters and representative sites (occupied wyckoff positions), with the the chemical elements and, in case of an alloy, another column with the site occupancies. Then follows the things we actually want – the Bravais lattice vectors and all positions of the atoms. In the case of Si in the diamond structure the result is probably familiar: the standard fcc lattice vectors and two atoms in the basis, one at $(0, 0, 0)$ and another in $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4})$.

Having generated the basic cell we want to export that to our favourite ESP. This is done by simply giving the option `-p espname` or `--program=espname`. The possible choices for `espname` can be found in Table 1, where you also find what files will be generated. Let's try this out for someones favourite program VASP. Adding the option `-p vasp` you will get the above output, properly formatted, to a file called POSCAR:

```
Generated by cif2cell 0.4.6 from [...] Species order: Si
   5.430530
  0.500000000000000   0.500000000000000   0.000000000000000
  0.500000000000000   0.000000000000000   0.500000000000000
  0.000000000000000   0.500000000000000   0.500000000000000
   2
Direct
  0.000000000000000   0.000000000000000   0.000000000000000
  0.250000000000000   0.250000000000000   0.250000000000000
```

Notice that the very long first line containing a lot of information about where the data came from is shortened here (marked by `[...]`). The important thing to see is that we get a correct output file, and that `cif2cell` helps you keep track of where the data came from, to the extent that this can be inferred from the CIF file. The last bit of the line, "Species order: Si", is important. It tells you which the atomic species are that are part of the file, since, in the particular case of VASP, this information is not given here, but in a different file, POTCAR, which was not generated by `cif2cell`. To enable the user to set up the POTCAR file properly, `cif2cell` stored the information needed here, but other ways of handling this information are also available, study the ESP specific output options to find out how.

3 Getting further

Here we go on discussing more cell generation options and some technical aspects of the cell generation.

3.1 Cell generation

By default, `cif2cell` will reduce the cell to the primitive cell, anticipating that the user wants to calculate something as small as possible, but we can also get the conventional cell. The diamond structure is a cubic system, and the primitive cell has only two atoms. Now try to also give the program the option `--no-reduce`. You should now get 8 atoms and cubic lattice vectors. Note that the conventional and primitive cell are the same in many systems.[2] For the fcc lattice the choice of primitive lattice vectors is straightforward, but in many systems of lower symmetry, in particular monoclinic systems, the choice is not always as simple. `cif2cell` has default choices for the Bravais lattice vectors that were selected to produce calculation cells similar to those listed at the Naval Research Laboratory site `http://cst-www.nrl.navy.mil/lattice/`.

3.2 Output

In section 2.2, output for VASP was generated to the default file POSCAR. You may also decide the output file name yourself by giving the option `-o outputfilename` or `--outputfile=outputfilename`. If you wish to append the output from `cif2cell` at the end of some existing file, give the option `-a` or `--append`, but note that this requires that you also specify the output file name. Appending to existing files is useful for example if you have a template file with your magic settings and just wish to add the cell with the atoms. You will notice that for nearly all ESP's, `cif2cell` generates *only* the geometric part of the setup, unless you specify the `--setup-all` flag, which not supported for many ESP's right now.

Generation of cells for alloy theory calculations is possible for ESP's that implement such features. For other ESP's, you may use the `--force-alloy` option to generate as much of the geometric input as possible.

Warning! Output files generated with `--force-alloy` *always* need further editing. The partially occupied sites will get some placeholder string where the element should be specified, no attempt is made by `cif2cell` to guess what you want to do.

The supported ESP's can be found in Table 1, where you also find what files will be generated and whether they support full output generation and alloy theory.

4 Supercell generation

`cif2cell` has a powerful supercell generator which enables you to set up virtually any possible larger system from the original CIF file. As usual, "powerful" to some extent also means "hard to understand and messy to use", but if you don't panic and take it a bit at a time it is not as bad as all that. We will just go through how to generate

[2] In all systems which has a space group, or Hermann-Mauguin (H-M), symbol that starts with "P" (for "primitive"). See the example for fcc Si above, where the H-M symbol is 'F', (from German "Flächenzentriert", meaning face-centered).

Table 1: Supported electronic structure programs (ESP) and the files output by `cif2cell`. Also shown is whether the ESP implements some alloy theory and whether `cif2cell` currently supports full output.

ESP	Alloy support	Full setup	Output file(s)
ABINIT	no	no	[compoundname].in
CASTEP	no	no	[compoundname].cell
cellgen	no	no	cellgen.inp
CPMD	no	no	[compoundname].inp
Crystal09	no	no	[compoundname].d12
Fleur	no	no	inp_[compoundname]
RSPt	no	no	symt.inp
Elk	no	no	GEOMETRY.OUT
EMTO	yes	no	[spacegroupname/compoundname].dat for kstr, bmdl, shape, kgrn and kfcd in separate directories.
Exciting	no	no	input.xml
ncol	no	no	[spacegroupname/compoundname].dat for bstr and ncol.
Siesta	no	no	[compoundname].fdf
Spacegroup	no	no	spacegroup.in
SPRKKR	yes	no	[compoundname].sys (via XBAND)
VASP	no	yes	POSCAR
XBAND	yes	no	[compoundname].sys

a number of supercells of Si to show the capabilities. If you didn't already do so, now is a good time to locate the example CIF's that comes with `cif2cell` and to do the examples as you read.

4.1 Basic supercell generation

The simplest way of getting a supercell was already mentioned, just use the `--no-reduce` flag to produce the conventional cell instead of the primitive cell. This will of course only produce a supercell for systems where the primitive cell and conventional cell are the same. Doing this for Si, we get an output that is similar to what was shown in Section 2.2, but the output cell is now:

```
 OUTPUT CELL INFORMATION
Bravais lattice vectors :
  1.0000000   0.0000000   0.0000000
  0.0000000   1.0000000   0.0000000
  0.0000000   0.0000000   1.0000000
All sites, (lattice coordinates):
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000
Si      0.5000000   0.0000000   0.5000000
Si      0.0000000   0.5000000   0.5000000
Si      0.7500000   0.2500000   0.7500000
Si      0.2500000   0.2500000   0.2500000
Si      0.2500000   0.7500000   0.7500000
Si      0.5000000   0.5000000   0.0000000
Si      0.7500000   0.7500000   0.2500000
```

You can see that we got a simple cubic lattice with 4 times as many atoms as in the primitive cell. This cell is probably what you want as your starting point for generating larger supercells for doing something like a defect calculation. It is very often easier to work with the conventional cell as a starting point for supercells, since they tend to better reflect the relevant symmetries at larger length scales, but there are exceptions to this rule which is why we sometimes need more advanced options. A typical example is generation of a (111) surface of a cubic system, which will be covered in Section 4.2.

As our first supercell example, let's double the conventional cell above along the x-direction. This is done by typing exactly like this:

```
cif2cell Si.cif --no-reduce --supercell=[2,1,1]
```

The argument to the `--supercell` option (the stuff in brackets after the "=" sign) just means "take two unit cells along the first lattice vectors, one unit cell along the second lattice vector and one unit cell along the third lattice vector". The resulting output is the same as before, but now we also get the supercell information at the end:

```
 SUPERCELL INFORMATION
Bravais lattice vectors :
  2.0000000   0.0000000   0.0000000
  0.0000000   1.0000000   0.0000000
  0.0000000   0.0000000   1.0000000
All sites, (lattice coordinates):
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000
Si      0.2500000   0.0000000   0.5000000
Si      0.0000000   0.5000000   0.5000000
Si      0.3750000   0.2500000   0.7500000
Si      0.1250000   0.2500000   0.2500000
Si      0.1250000   0.7500000   0.7500000
Si      0.2500000   0.5000000   0.0000000
Si      0.3750000   0.7500000   0.2500000
Si      0.5000000   0.0000000   0.0000000
Si      0.7500000   0.0000000   0.5000000
Si      0.5000000   0.5000000   0.5000000
Si      0.8750000   0.2500000   0.7500000
Si      0.6250000   0.2500000   0.2500000
Si      0.6250000   0.7500000   0.7500000
Si      0.7500000   0.5000000   0.0000000
Si      0.8750000   0.7500000   0.2500000
```

Here we clearly got another cell along the x-direction (actually along the direction of the first lattice vector, which happens to be along the x axis in this case) and there are twice as many atoms. We can put any integers we like with `--supercell`, so we can get as large a cell as we like. Such a basic supercell based on either the primitive or the conventional cell is often all you need.

4.2 Advanced supercell generation

The previous section demonstrated how to generate supercells that are simply multiples of the original lattice vectors, but `cif2cell` also allows construction of general supercells, where a the original lattice vectors can be remapped to *any* other lattice vector. We have already encountered one example of this, namely the mapping from the primitive cell to the conventional cell, but now we will see how it is done with a supercell map for a couple of examples.

4.2.1 Cubic cell from fcc cell

Getting the conventional, cubic unit cell for Si is of course most easily done by just giving the `--no-reduce` flag, as was shown earlier for Si, but we could have generated the same cell by instead specifying a supercell map matrix that takes the primitive Bravais lattice

vectors of the fcc lattice to the unit matrix. For `cif2cell`'s choice of primitive lattice vectors, this map is given by

$$\begin{pmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}, \tag{1}$$

and to feed this into the program, just type

```
cif2cell Si.cif --supercell=[[1,1,-1],[1,-1,1],[-1,1,1]]
```

Try this and verify that you get the same cell as you got with `--no-reduce` and which is also illustrated in Figure 1.

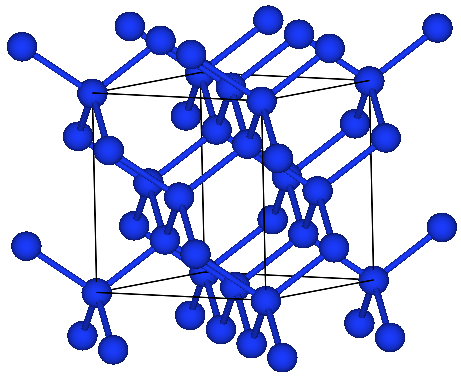

Figure 1: Cubic cell of fcc Si in standard orientation with one of the fourfold rotation axes along the z direction.

Two technical remark is in order here. 1) What is being entered after the "=" sign in the `--supercell` argument is Python syntax for a matrix. Python uses row-major order, so it should be read as "three columns of row vectors". When we used the "simple" supercell mode in Section 4.1, we entered a vector, which the program converts to a diagonal matrix. 2) The supercell map is operating from left to right onto bravais lattice vectors represented as *row* vectors. This is important to get things right when we go on to building up a more complicated map in steps.

The map here is just the inverse of the Bravais lattice vector matrix, $\mathbf{B}_{fcc}$. The reason for that is that we want the unit matrix in the end, in general to get the map, $\mathbf{M}$, that produces the new lattice vectors, $\mathbf{B}_{new}$, from the old lattice vectors, $\mathbf{B}_{old}$, we have

$$\mathbf{M}\mathbf{B}_{old} = \mathbf{B}_{new} \quad \Rightarrow \quad \mathbf{M} = \mathbf{B}_{new}\mathbf{B}_{old}^{-1}. \tag{2}$$

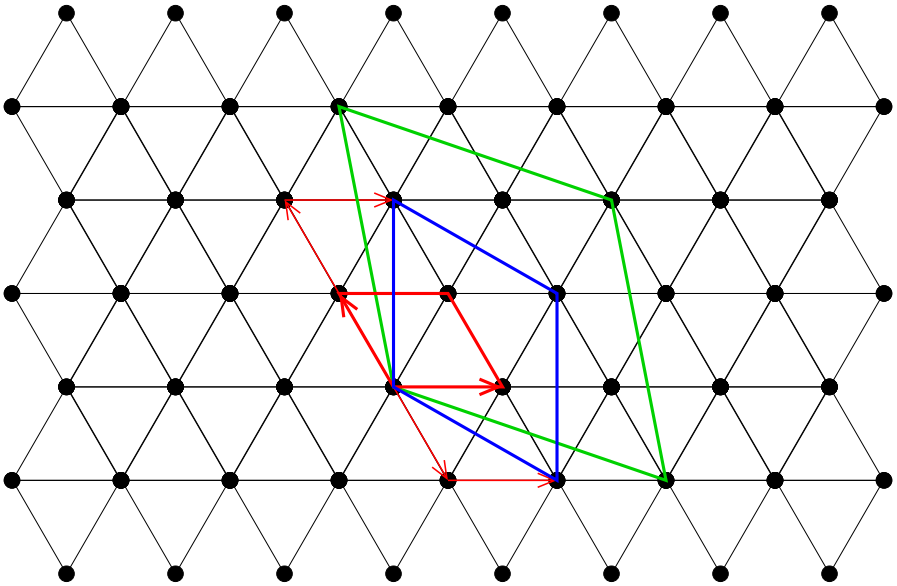

Figure 2: Examples of non-trivial hexagonal surface supercells. The red cell is a primitive cell with lattice vectors as generated by `cif2cell` shown with arrows. The blue cell is a $(\sqrt{3} \times \sqrt{3})R30°$ cell and the green cell is a $(\sqrt{7} \times \sqrt{7})R19°$ structure. Thin red arrows illustrate the construction of the $\sqrt{3}$ supercell map.

4.2.2 Hexagonal surface supercell

In many cases the most straightforward way is to just figure out the new lattice vectors directly in terms of the old vectors. If we look at Figure 2 of a hexagonal surface, the primitive surface cell and a $(\sqrt{3} \times \sqrt{3})R30°$ cell are shown in red and blue, respectively. Thinner red lines demonstrate that to get one of the new supercell lattice vectors (blue) we take 2 of the first primitive lattice vector and 1 of the second. This gives the first row of the matrix below. To get the second supercell lattice vector, we take -1 of the first primitive vector and 1 of the second, and this gives the second row of the matrix below. Leaving the out-of-plane axis unchanged, this gives the following supercell map:

$$\begin{pmatrix} 2 & 1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3}$$

It is a good idea to not go on to the next section before you have understood how this works.

4.2.3 Hexing the cube: Si(111) surface supercell

Generating a (111) surface of a cubic compound is a common problem for computational scientists that drives you crazy the first time you work it out. When viewed from any of the main diagonals (such as the [111] direction), a cubic system looks hexagonal, since

the main body diagonals of a cube have threefold rotation symmetry. If that does not sound familiar, this is a good point to sit down with pen and paper and draw a cube in perspective, or turn something cubic around a couple of times so that you see how it works.

We will now produce a Si(111) surface supercell with the surface perpendicular to the z axis, the standard way of orienting a surface. This is done in steps, gradually building the cell from the primitive fcc cell. It is recommended that you study the intermediate cells that are created along the way to see that everything works correctly. This is most easily done by choosing `cif` as your output program. This will produce an output file with the name `[inputfilename]_allatoms.cif`, which contains the cell with all its atoms for rendering in some visualization program. As we go, we will see different cells that represent very different cuts of the crystal, so the atoms may appear to be organized differently. Remember that you are always looking at exactly the same Si in the diamond structure as in Fig. 1. Work out the local environment for each atom if you need to convince yourself.

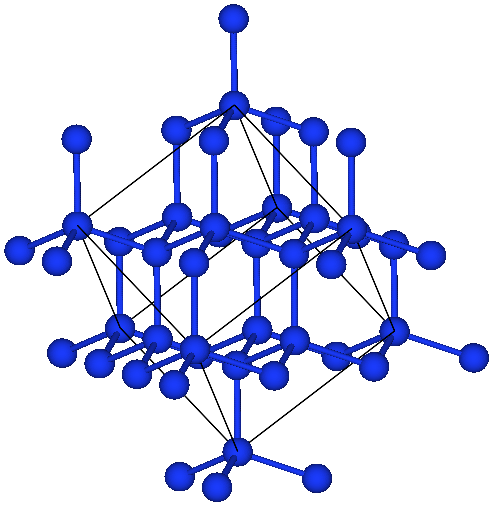

Figure 3: Cubic cell of fcc Si with the [111] direction along the z axis.

Since we want the (111) planes to lie along the z axis, we first need to rotate the fcc lattice vectors so that the main diagonal of the cube ends up along the [001] direction. There is a command for performing rotations of the lattice vectors, `--transform-cell`, by explicitly giving a rotation matrix to be applied from the right to the lattice vector matrix (since the lattice vectors are *row* vectors). For the particular case of aligning the main diagonal of a cube to the z axis, however, there is a ready defined command, `--cubic-diagonal-z`. If we apply this:

```
cif2cell Si.cif --cubic-diagonal-z
```

we produce a cell as shown in Fig. 4.

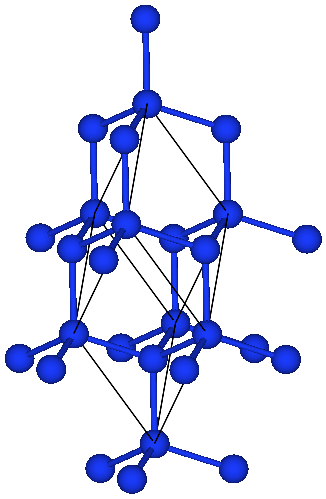

Figure 4: Rhombohedral cell of fcc Si with the [111] direction along the z axis.

The corresponding output from `cif2cell`is the same as for the primitive cell, but we get a new section describing the transformed cell:

```
 TRANSFORMED CELL
Bravais lattice vectors :
  0.5773503   0.0000000   0.8164966
 -0.2886751   0.5000000   0.8164966
 -0.2886751  -0.5000000   0.8164966
All sites, (lattice coordinates):
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000
Si      0.2500000   0.2500000   0.2500000
```

The next step here is to map the rhombohedral cell to its corresponding hexagonal cell, which is done by the supercell map matrix

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}, \tag{4}$$

and so we add to the input: `--supercell=[[1,-1,0],[0,1,-1],[1,1,1]]`. This adds a supercell section to the output:

```
 SUPERCELL INFORMATION
Bravais lattice vectors :
  0.8660254  -0.5000000   0.0000000
 -0.0000000   1.0000000   0.0000000
 -0.0000000   0.0000000   2.4494897
```

```
All sites, (lattice coordinates):
Atom           a1          a2          a3
Si      0.0000000   0.0000000   0.0000000
Si      0.0000000   0.0000000   0.2500000
Si      0.6666667   0.3333333   0.3333333
Si      0.3333333   0.6666667   0.6666667
Si      0.6666667   0.3333333   0.5833333
Si      0.3333333   0.6666667   0.9166667
```

and we get the new unit cell as shown in Figure 5.

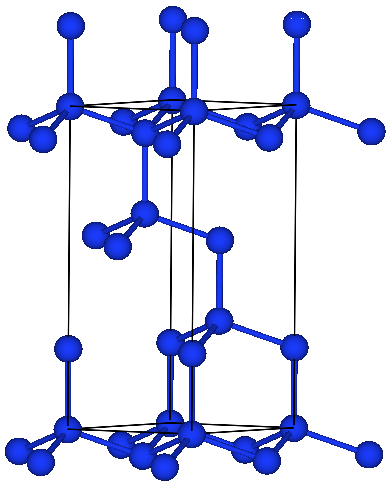

Figure 5: Hexagonal cell of fcc Si with the [111] direction along the z axis.

Next, we probably want to have a couple of extra layers in the cell, and this has to be achieved by applying a new supercell map after the previous one that gave us the hexagonal cell. We have three layers in the primitive cell, but suppose that we want to double that, then we need to multiply the first supercell map by the map that doubles the cell in the z direction, from the left:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 2 & 2 \end{pmatrix}. \tag{5}$$

In general, when building up a map in n steps, starting at step 1, the map is given by:

$$\mathbf{M} = \mathbf{M}_n \mathbf{M}_{n-1} \dots \mathbf{M}_1. \tag{6}$$

The Si(111) surface has an observed reconstruction which turns it into a 7×7 surface supercell. To generate a 7×7 surface supercell, we need to expand the previous map in

the ab plane:

$$\begin{pmatrix} 7 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 7 & -7 & 0 \\ 0 & 7 & -7 \\ 2 & 2 & 2 \end{pmatrix}. \tag{7}$$

You can admire the result of applying this matrix in Figure 6.

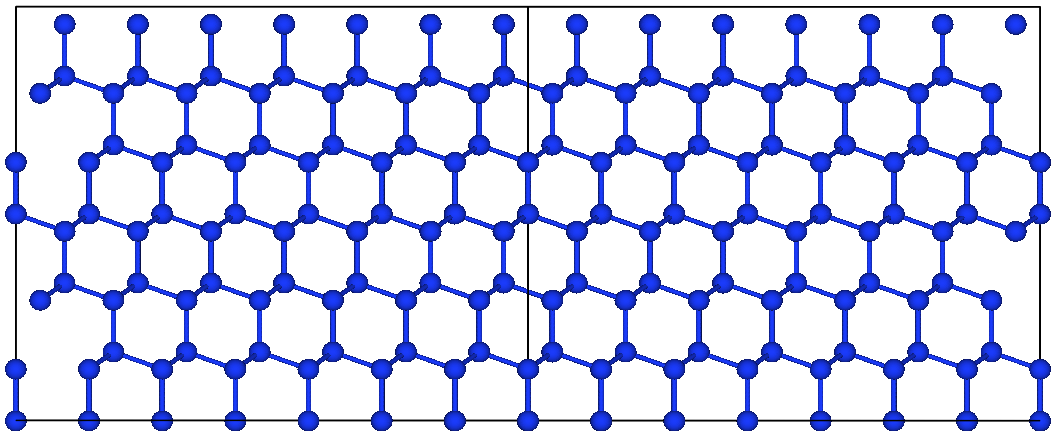

Figure 6: Hexagonal supercell of fcc Si with a 7×7 primitive cells in the ab plane.

Now, the cell that we created is not a surface yet, we need to add vacuum along the z direction. This is done with `--supercell-vacuum=[a,b,c]`, where a, b and c are numbers (not necessarily integers) that give the amount of vacuum in units of the three lattice vectors. This means that to make a surface supercell in this way, the surface *must* be aligned to the cell axes, which is why we need the `--transform-cell` command to align the cell correctly. The vacuum is added *right at the cell boundary* along the lattice vectors and *after the supercell map* has been applied. In the present case we wan to add the vacuum along the third lattice vector and we can for example choose to make the vacuum as thick as our Si slab. This is then is done by giving the command `--supercell-vacuum=[0,0,1]`. But wait! Let's look again at Figure 6. In the figure all atoms that fall outside of the cell has been cut out, and also the atoms in the uppermost plane, that belong to the next cell. Adding vacuum at the upper boundary in this figure would give us a surface that consists of a lot of single Si atoms sticking out (Figure 6 shows this clearly). This is not what we want in this case, we would instead like to make the cut just below the top layer of atoms (or just above the bottom layer). This can be done using the `--supercell-translation-vector` command, which allows

you to give a vector that translates all the coordinates of the cell. The translation vector is applied before the vacuum is added given in units of the *supercell*. In our case we just need to move all atoms a little way up (or down) and so to get the surface supercell, in our case, we can for example choose the vector as `[0,0,0.084]`.

The complete command for generating the 7×7 Si(111) surface supercell is then:

```
cif2cell Si.cif --cubic-diagonal-z\
--supercell=[[7,-7,0],[0,7,-7],[2,2,2]]\
--supercell-translation-vector=[0,0,0.084]\
--supercell-vacuum=[0,0,1]
```

Note that the backslashes just indicate a newline that you should not have when giving the command in the terminal. The finished surface cell is shown in Figure 7.

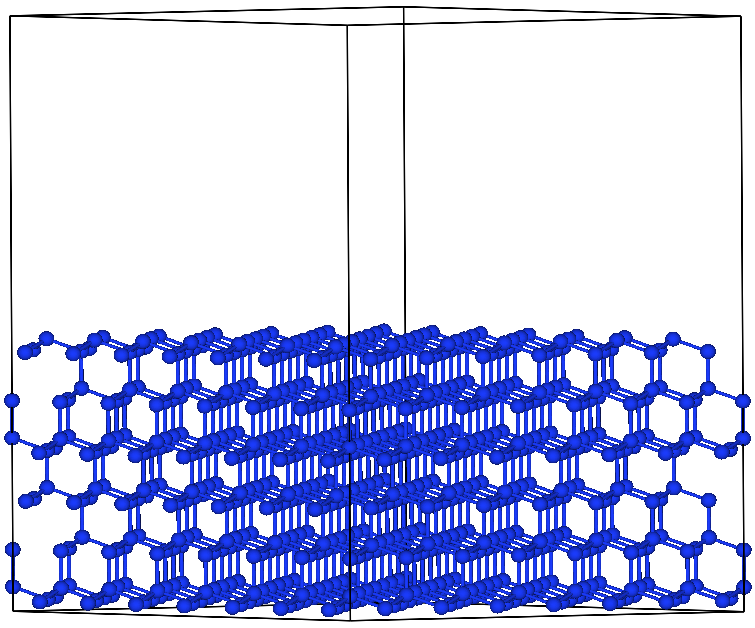

Figure 7: Si(111) surface supercell with a 7×7 primitive cells in the ab plane.

A Technicalities

A.1 Order in which things are done in the supercell generation

Note the order, and in particular the units in which the operations are done. The

1. Generation of primitive cell (or the conventional cell, if the flag `--no-reduce` is used).
2. Rotation/rescaling transformation of the lattice vectors with `--transform-cell`.
3. Applying supercell map with `--supercell=...`. The units are the previously generated primitive or conventional cell.
4. Moving atoms in the supercell by `--supercell-translation-vector` *or* `--supercell-prevacuum-translation` (in units of the supercell!).
5. Adding vacuum along the direction of one of the lattice vectors with `--supercell-vacuum` (in units of the supercell!).
6. Moving atoms in the supercell by `--supercell-postvacuum-translation` (in units of the supercell including vacuum!).

Table 2: Various useful transformations.

Transform this...	...into that...	...with this
Standard rhombohedral cell with threefold axis along the z direction	Standard hexagonal cell with sixfold axis along the z direction	$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$
Hexagonal unit cell	$(\sqrt{3} \times \sqrt{3})R30°$ hexagonal cell	$\begin{pmatrix} 2 & 1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Hexagonal unit cell	$(\sqrt{7} \times \sqrt{7})R19°$ hexagonal cell	$\begin{pmatrix} 3 & 1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$