Modeling materials using density functional theory

John Kitchin

2012-07-11 Wed

Contents

1 Introduction to this book

This book serves two purposes: 1) to provide worked examples of using DFT to model materials properties, and 2) to provide references to more advanced treatments of these topics in the literature. It is not a definitive reference on density functional theory. Along the way to learning how to perform the calculations, you will learn how to analyze the data, make plots, and how to interpret the results. This book is very much "recipe" oriented, with the intention of giving you enough information and knowledge to start your research. In that sense, many of the computations are not publication quality with respect to convergence of calculation parameters.

You will read a lot of python code in this book. That is by design. I believe that computational work should always be scripted. That leaves a record of everything you did, so that you can reproduce it later, or report exactly what you did.

This book makes heavy use of many computational tools including:

- Python

  - Module index

- Atomic Simulation Environment (ase)

- numpy

- scipy

- matplotlib

- emacs

  - org-mode This book is written in org-mode, and is best read in emacs in org-mode. This format provides clickable links, easy navigation, syntax highlighting, as well as the ability to interact with the tables and code. The book is also available in pdf.

- git This book is available at https://github.com/jkitchin/dft-book

- jasp

jasp is a sophisticated wrapper to the :mod:`ase.calculators.vasp` python interface to the VASP calculator. It was written by me to facilitate writing this book, and to develop the best possible way to run DFT calculations. The best way to learn to use jasp is from this book. jasp is currently available at https://bitbucket.org/jkitchin/jasp.

It is not critical that you use jasp. You should be able to copy the actual VASP input files (which were generated by jasp) from a directory and simply run vasp. jasp is mostly compatible with :mod:`ase.calculators.vasp`, so you can replace this kind of code:

```
with jasp('directory', lotsofkeywords, atoms=atoms) as calc:
    #do stuff
```

with this code:

```
CWD = os.getcwd()
os.chdir('directory')
calc=Vasp(lotsofkeywords)
atoms.set_calculator(calc)

try:
    #do stuff
finally:
    os.chdir(CWD)
```

The DFT code used primarily in this book is Vasp.

- Vasp wiki

- Vasp Manual

Similar code would be used for other calculators, e.g. GPAW, Jacapo, etc. . . you would just have to import the python modules for those codes, and replace the code that defines the calculator.

2 Introduction to DFT

All calculations were performed using VASP [17, 18, 20, 19], the Perdew-Burke-Ernzerhof (PBE) [28] exchange correlation functional with projector augmented wave (PAW) potentials provided in VASP [3, 21].

Monkhorst-Pack k-points [24]

2.1 Recommended reading

Original papers [13, 16]
   Kohn's Nobel Lecture [15], Pople's Nobel Lecture [29]
   [12] Hoffman
   PAW in GPAW [25]

3 Molecules

In this chapter we consider how to construct models of molecules, how to manipulate them, and how to calculate many properties of molecules.

3.1 Defining and visualizing molecules

We start by learning how to define a molecule and visualize it.

3.1.1 Predefined molecules

ASE defines a number of molecular geometries in the molecules database. Data for the G2 database are from Raghavachari, Redfern, and Pople, J. Chem. Phys. Vol. 106, 1063 (1997). See http://chemistry.anl.gov/compmat/comptherm.htm for the original files.

All numbers are experimental values, except for coordinates, which are MP2(full)/6-31G(d) optimized geometries. Here is a list of all the species available in mod:ase.data.g2

```
1   from ase.data import g2
2   print g2.data.keys()
```

Some other databases include the mod:ase.data.s22 for weakly interacting dimers and complexes, and mod:ase.data.extra_molecules which has a few extras like biphenyl and C60.

Here is an example of getting the geometry of an acetonitrile molecule and writing an image to a file. Note that the default unit cell is a $1\mathring{A}\times1\mathring{A}\times1\mathring{A}$ cubic cell. That is too small to use if your calculator uses periodic boundary conditions. We center the atoms in the unit cell and add vacuum on each side. We will add 6 $\mathring{A}$ of vacuum on each side. In the write command we use the option show$_{\text{unitcell}}$=2 to draw the unit cell boundaries

```
1   from ase import Atoms
2   from ase.data.molecules import molecule
```

```
3   from ase.io import write
4
5   c60 = molecule('CH3CN')
6
7   c60.center(vacuum=6)
8   print c60.get_cell()
9
10  write('molecules/ch3cn.png', c60, show_unit_cell=2)
```
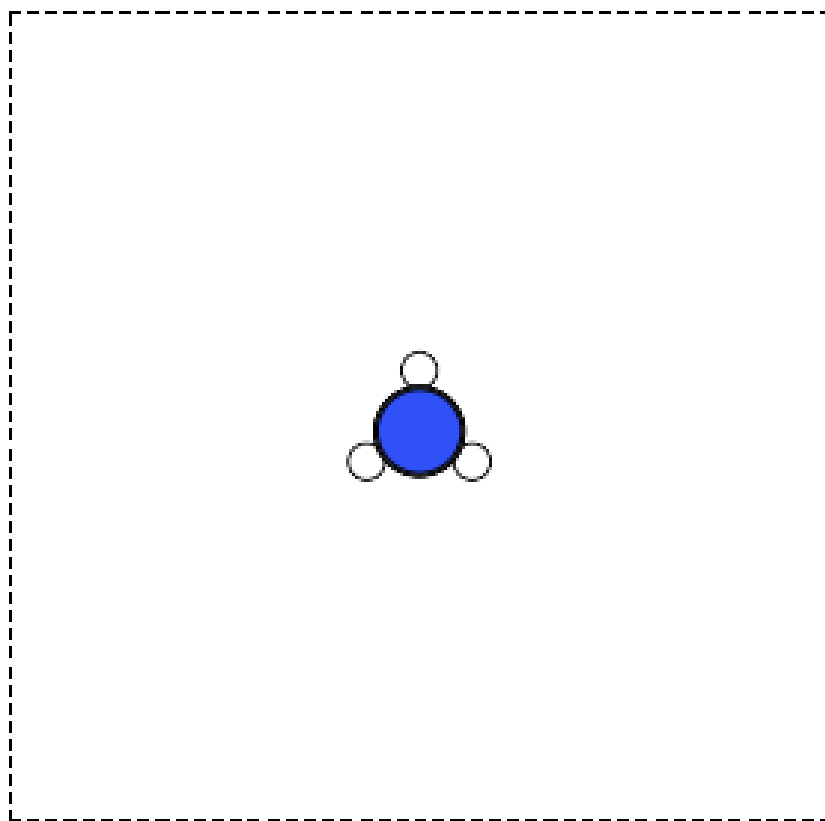

It is possible to rotate the atoms with func:ase.io.write if you wanted to see pictures from another angle. In the next example we rotate 45 degrees about the x-axis, then 45 degrees about the y-axis. Note that this only affects the image, not the actual coordinates

```
1   from ase import Atoms
2   from ase.data.molecules import molecule
3   from ase.io import write
4
5   atoms = molecule('CH3CN')
6
```

```
 7     atoms.center(vacuum=6)
 8     print atoms.get_cell()
 9
10     write('molecules/ch3cn-rotated.png', atoms,
11           show_unit_cell=2,rotation='45x,45y,0z')
```
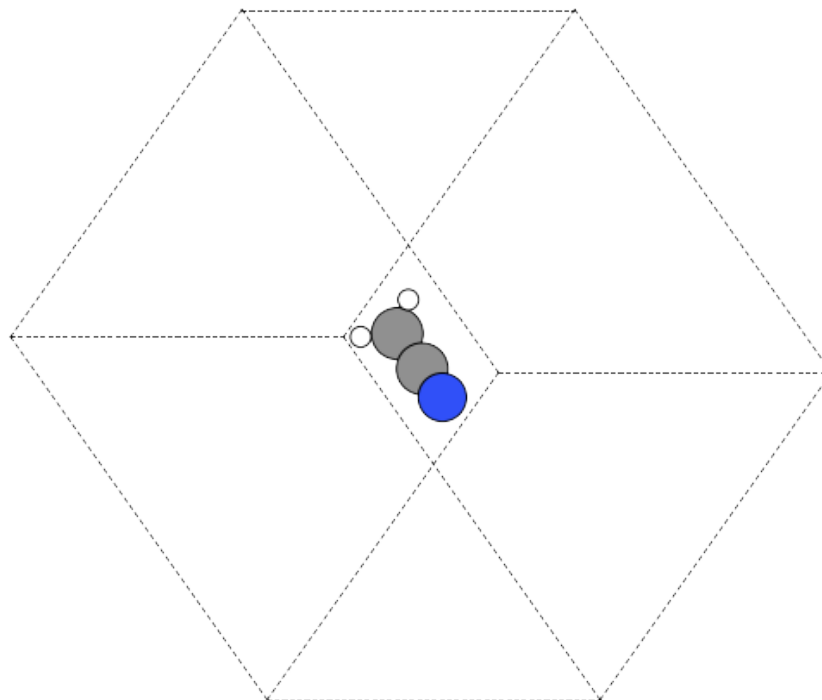

If you actually want to rotate the coordinates, there is a nice way to do that to, with the func:ase.Atoms.rotate method. Actually there are some subtelties in rotation. One rotates the molecule an angle (in radians) around a vector, but you have to choose whether the center of mass should be fixed or not. You also must decide whether the unit cell should be rotated or not. In the next example you can see the coordinates have changed due to the rotations.

```
1     from ase import Atoms
2     from ase.data.molecules import molecule
3     from ase.io import write
4     from numpy import pi
5
6     # ammonia
7     atoms = molecule('CH3CN')
8     p1 = atoms.get_positions()
9
```

```
10    atoms.rotate('x', pi/4, center='COM', rotate_cell=False)
11    atoms.rotate('y', pi/4, center='COM', rotate_cell=False)
12
13    write('molecules/ch3cn-rotated-2.png',atoms,show_unit_cell=2)
14    p2 = atoms.get_positions()
15    print p2 - p1
```
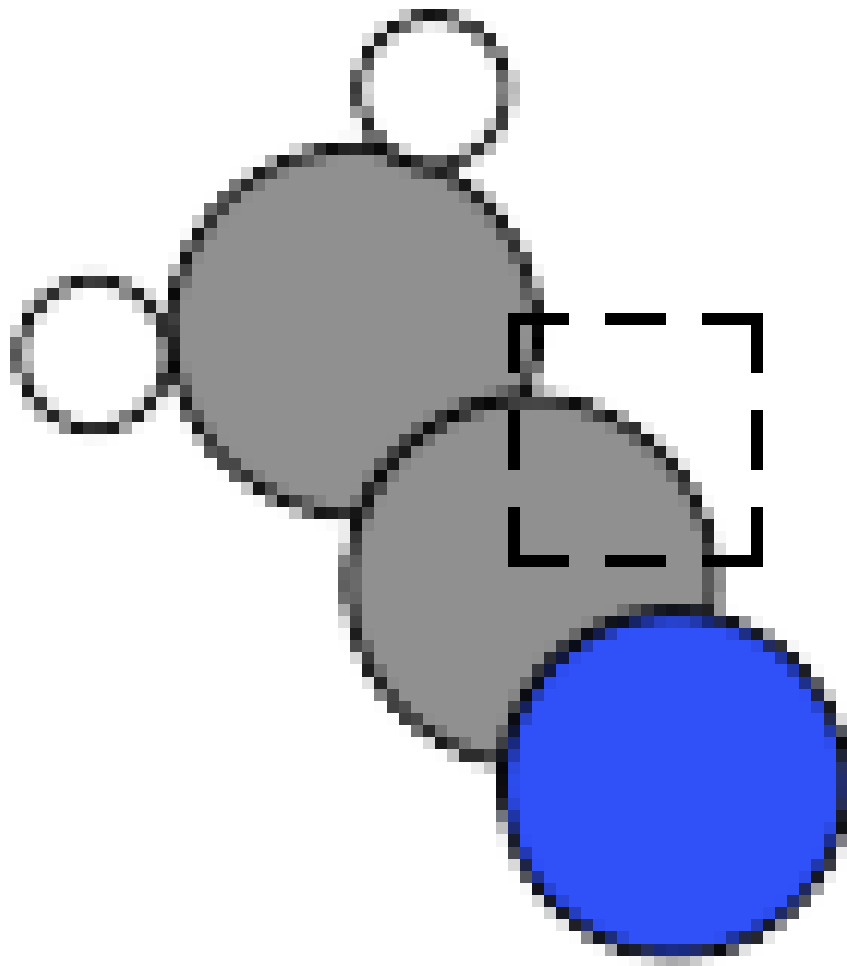

Note in this last case the unit cell is oriented differently than the previous example, since we chose not to rotate the unit cell.

3.1.2   Reading other data formats in

You can read xyz-coordinate files to create mod:ase.Atoms objects. Here is what an xyz file might look like:

Note that the xyz format does not have unit cell information in it, so you will have to figure out a way to provide it. In this example, we center the atoms in a box with vacuum on all sides.

```
1    from ase.io.xyz import *
2    from ase.io import write
3    import numpy as np
4
5    atoms = read_xyz('molecules/isobutane.xyz')
6    atoms.center(vacuum=5)
7    write('molecules/isobutane_xyz.png', atoms, show_unit_cell=2)
```

file:molecules/isobutane_xyz.png

3.1.3   From scratch

When there is no data file for the molecule you want, or no database to get it from, you have to define your atoms geometry by hand. Here is how that is done.

```
1    from ase import Atoms, Atom
2    from ase.io import write
3
4    #method 1 - simple cubic unit cell
5    atoms = Atoms([Atom('C',[0., 0.,0.]),
6                  Atom('O',[1.1,0.,0.])],
7                  cell=(10,10,10))
8
9    print 'V = %1.0f Ang^3' % atoms.get_volume()
10
11   write('molecules/simple-cubic-cell.png',atoms,show_unit_cell=2)
```

11
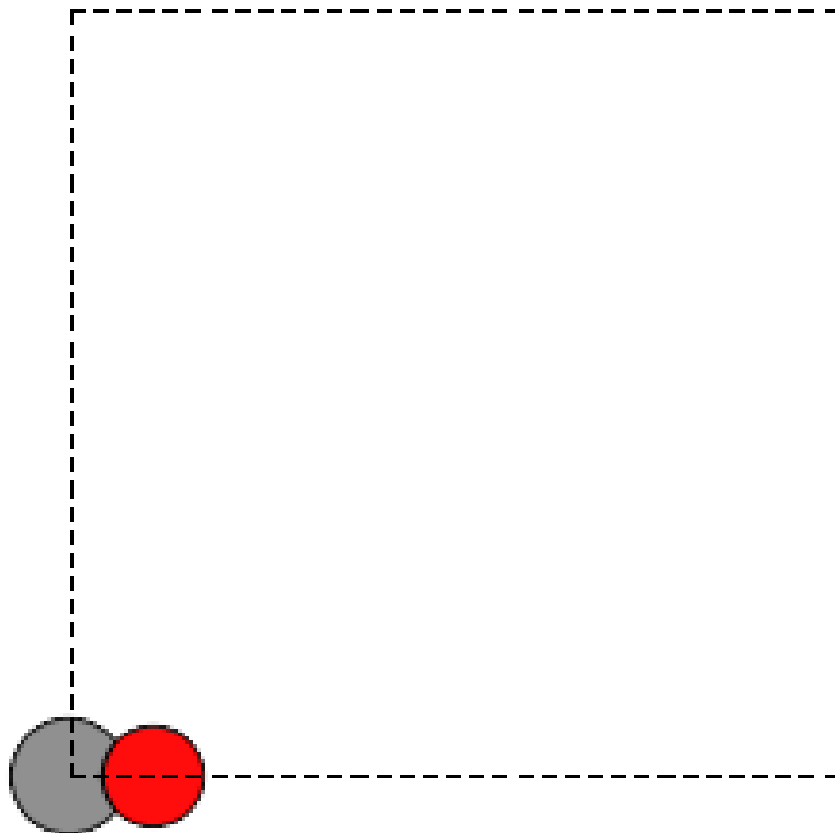
There are two inconvenient features of the simple cubic cell:

1. Since the CO atom is at the corner, its electron density is spread over the 8 corners of the box, which is not convenient for visualization later.

2. Due to the geometry of the cube, you need fairly large cubes to decouple the molecule from its images. Here, the CO molecule has 6 images due to periodic boundary conditions that are 10 angstroms away. The volume of the unit cell is 1000 $\mathring{A}^3$.

The first problem is easy to solve by centering the atoms in the unit cell. The second problem can be solved by using an fcc lattice. Below we show the result, where we have guessed values for b until the CO molecules are on average 10 angstroms apart. Note the final volume is only about 715 $\mathring{A}^3$, which is smaller than the cube. This will result in less computational time to compute properties.

```
1    from ase import Atoms, Atom
2    from ase.io import write
3
4    b = 7.1
5    atoms2 = Atoms([Atom('C',[0., 0.,0.]),
6                   Atom('O',[1.1,0.,0.])],
7                  cell=[[b, b, 0.],
8                        [b, 0., b],
9                        [0., b, b]])
10   print 'V = %1.1f Ang^3' % atoms2.get_volume()
11
12   atoms2.center() #translate atoms to center of unit cell
13   write('molecules/fcc-cell.png',atoms2,show_unit_cell=2)
```
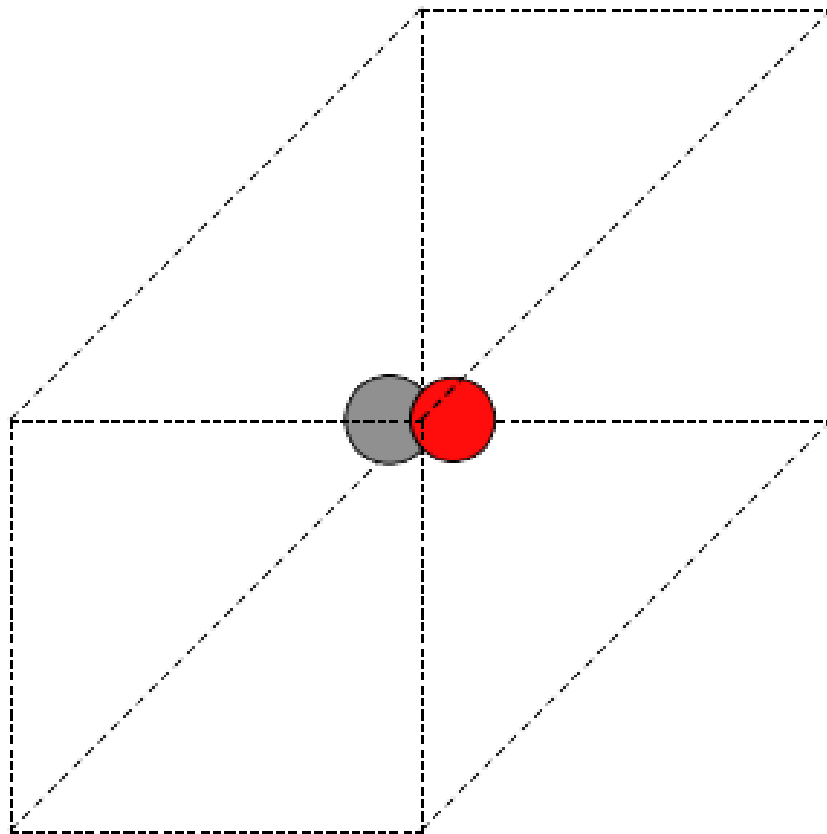

At this point you might ask "How do you know the distance to the neighboring image?" The `ag` viewer lets you compute this graphically, but we can use code to determine this too. All we have to do is figure out the length of each lattice vector. We use the numpy module to compute the distance of a vector as the square root of the sum of squared elements.

13

```
1    from ase import Atoms, Atom
2    import numpy as np
3
4    b = 7.1
5    atoms2 = Atoms([Atom('C',[0., 0.,0.]),
6                    Atom('O',[1.1,0.,0.])],
7                   cell=[[b, b, 0.],
8                         [b, 0., b],
9                         [0., b, b]])
10
11   # get unit cell vectors and their lengths
12   (a1, a2, a3) = atoms2.get_cell()
13   print '|a1| = %1.2f Ang' % np.sum(a1**2)**0.5
14   print '|a2| = %1.2f Ang' % np.sum(a2**2)**0.5
15   print '|a3| = %1.2f Ang' % np.sum(a3**2)**0.5
```

3.2

3.2.1 Combining Atoms objects

It is frequently useful to combine two Atoms objects, e.g. for computing reaction barriers, or other types of interactions. in ase, we simply add two Atoms objects together. Here is an example of getting an ammonia and oxygen molecule in the same unit cell.

```
1    from ase import Atoms
2    from ase.data.molecules import molecule
3    from ase.io import write
4
5    atoms1 = molecule('NH3')
6
7    atoms2 = molecule('O2')
8    atoms2.translate([3,0,0])
9
10   bothatoms = atoms1 + atoms2
11   bothatoms.center(5)
12
13   write('molecules/bothatoms.png', bothatoms, show_unit_cell=2, rotation='90x')
```
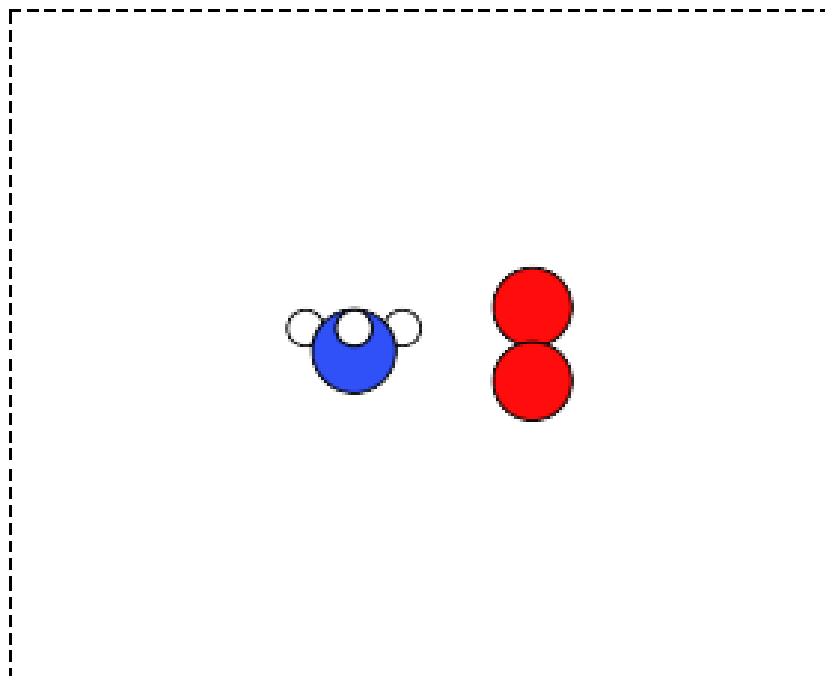
3.3 Simple properties

Simple properties do not require a DFT calculation. They are typically only functions of the atom types and geometries.

3.3.1 Getting cartesian positions

If you want the x,y,z coordinates of the atoms, use the `func:ase.Atoms.get_positions`. If you are interested in the fractional coordinates, use `func:ase.Atoms.get_scaled_positions`.

```python
from ase import Atoms
from ase.data.molecules import molecule
from ase.io import write

atoms = molecule('C6H6')

# access properties on each atom
print ' # sym   p_x     p_y     p_z'
print '----------------------------'
for i,atom in enumerate(atoms):
  print '%3i%3s%8.2f%8.2f%8.2f' % (i,atom.symbol,atom.x,atom.y,atom.z)

# get all properties in arrays
sym = atoms.get_chemical_symbols()
```

```
15    pos = atoms.get_positions()
16    num = atoms.get_atomic_numbers()
17
18    atom_indices = range(len(atoms))
19
20    print
21    print ' # sym   at#     p_x      p_y      p_z'
22    print '-----------------------------------'
23    for i,s,n,p in zip(atom_indices,sym,num,pos):
24      px,py,pz = p
25      print '%3i%3s%6i%10.2f%8.2f%8.2f' % (i,s,n,px,py,pz)
```

```
 #  sym   p_x      p_y      p_z
----------------------------
 0   C    0.00     1.40     0.00
 1   C    1.21     0.70     0.00
 2   C    1.21    -0.70     0.00
 3   C    0.00    -1.40     0.00
 4   C   -1.21    -0.70     0.00
 5   C   -1.21     0.70     0.00
 6   H    0.00     2.48     0.00
 7   H    2.15     1.24     0.00
 8   H    2.15    -1.24     0.00
 9   H    0.00    -2.48     0.00
10   H   -2.15    -1.24     0.00
11   H   -2.15     1.24     0.00


 #  sym   at#    p_x      p_y      p_z
------------------------------------
 0   C     6    0.00     1.40     0.00
 1   C     6    1.21     0.70     0.00
 2   C     6    1.21    -0.70     0.00
 3   C     6    0.00    -1.40     0.00
 4   C     6   -1.21    -0.70     0.00
 5   C     6   -1.21     0.70     0.00
 6   H     1    0.00     2.48     0.00
 7   H     1    2.15     1.24     0.00
 8   H     1    2.15    -1.24     0.00
 9   H     1    0.00    -2.48     0.00
10   H     1   -2.15    -1.24     0.00
11   H     1   -2.15     1.24     0.00
```

3.3.2 Molecular weight and molecular formula

We can quickly compute the molecular weight of a molecule with this little recipe.

```
1   from ase import Atoms
2   from ase.data.molecules import molecule
3
4   atoms = molecule('C6H6')
5   masses = atoms.get_masses()
6
7   molecular_weight = sum(masses)
8
9   print 'The molecular weight of %s is %f' % (atoms.get_chemical_symbols(reduce=True),
10                                               molecular_weight)
```

```
The molecular weight of C6H6 is 78.113640
```

3.3.3 Center of mass

The center of mass is a helpful quantity to have for a variety of computations, including translating atoms for a rotation, etc... Here is an example of getting the center of mass from an Atoms object using func:ase.Atoms.get_center_of_mass.

$$\mathrm{COM} = \frac{\sum m_i \cdot r_i}{\sum m_i}$$

```
1   from ase import Atoms
2   from ase.structure import molecule
3   from ase.io import write
4
5   # ammonia
6   atoms = molecule('NH3')
7
8   print atoms.get_center_of_mass()   # cartesian coordinates
9
10  # compute the center of mass by hand
11  from ase.data import atomic_masses
12  import numpy as np
13  pos = atoms.positions
14  masses = atoms.get_masses()
15
16  COM = np.array([0., 0., 0.])
17  for m,p in zip(masses, pos):
18      COM += m*p
19  COM /= sum(masses)
20
21  print COM
22
23  # one-line linear algebra definition of COM
24  print np.dot(masses, pos)/np.sum(masses)
```

```
[  0.00000000e+00    5.91843349e-08    4.75457009e-02]
[  0.00000000e+00    5.91843349e-08    4.75457009e-02]
[  0.00000000e+00    5.91843349e-08    4.75457009e-02]
```

3.3.4   Moments of inertia

`func:ase.Atoms.get_moments_of_inertia`

```python
from ase import *
from ase.structure import molecule
from ase.units import *

print 'linear rotors: I = [Ia Ia 0]'
atoms = molecule('CO2')
print '  CO2 moments of inertia: ',atoms.get_moments_of_inertia()
print

print 'symmetric rotors (IA = Ib) < Ic'
atoms = molecule('NH3')
print '  NH3 moments of inertia: ' ,atoms.get_moments_of_inertia()

atoms = molecule('C6H6')
print '  C6H6 moments of inertia: ' ,atoms.get_moments_of_inertia()
print

print 'symmetric rotors (IA = Ib) > Ic'
atoms = molecule('CH3Cl')
print 'CHCl3 moments of inertia: ',atoms.get_moments_of_inertia()
print

print 'spherical rotors Ia = Ib = Ic'
atoms = molecule('CH4')
print '  CH4 moments of inertia: ' ,atoms.get_moments_of_inertia()
print

print 'unsymmetric rotors Ia != Ib != Ic'
atoms = molecule('C3H7Cl')
print '  C3H7Cl moments of inertia: ' ,atoms.get_moments_of_inertia()
```

```
linear rotors: I = [Ia Ia 0]
  CO2 moments of inertia:  [  0.          44.45384271  44.45384271]

symmetric rotors (IA = Ib) < Ic
  NH3 moments of inertia:  [ 1.71012426  1.71012548  2.67031768]
  C6H6 moments of inertia:  [  88.77914641   88.77916799  177.5583144 ]

symmetric rotors (IA = Ib) > Ic
CHCl3 moments of inertia:  [  3.20372189  37.97009644  37.97009837]
```

18

```
spherical rotors Ia = Ib = Ic
  CH4 moments of inertia:  [ 3.19145621  3.19145621  3.19145621]

unsymmetric rotors Ia != Ib != Ic
  CHFClBr moments of inertia:  [  19.41351508  213.18961963  223.16255537]

    mod:ase.structure.molecule
```

3.4 Simple properties that require computations

3.4.1 Computing bond lengths and angles

It is a common to compute bond lengths. The Atoms object contains a
func:ase.Atoms.get_distance method to make this easy. You have to
specify the indices of the two atoms you want the distance between. Don't
forget the indices start at 0.

```
1    from ase import Atoms
2    from ase.structure import molecule
3    from ase.io import write
4
5    # ammonia
6    atoms = molecule('NH3')
7
8    for i, atom in enumerate(atoms):
9        print '%2i %3s' % (i,atom.symbol)
10
11   # N-H bond length
12   print atoms.get_distance(0,1)
```

```
 0   N
 1   H
 2   H
 3   H
1.01679344636
```

Bond angles are a little trickier. We can use some simple trigonometry:
$a \cdot b = |a||b|\cos(\theta)$, so we can calculate the angle as $\theta = \arccos\left(\frac{a \cdot b}{|a||b|}\right)$, we
just have to define our two vectors a and b. We compute these vectors as
the difference in positions of two atoms. For example, here we compute
the angle H-N-H in an ammonia molecule. This is the angle between N-H1
and N-H2. In the next example, we utilize functions in numpy to perform
the calculations, specifically the arccos function, the dot function, and norm
functions.

19

```
1   from ase import Atoms
2   from ase.structure import molecule
3   from ase.io import write
4
5   # ammonia
6   atoms = molecule('NH3')
7
8   for i, atom in enumerate(atoms):
9       print '%2i %3s' % (i,atom.symbol)
10
11  a = atoms.positions[0] - atoms.positions[1]
12  b = atoms.positions[0] - atoms.positions[2]
13
14  from numpy import arccos, dot, pi
15  from numpy.linalg import norm
16
17  theta_rad = arccos(dot(a,b)/(norm(a)*norm(b))) # in radians
18
19  print 'theta = %1.1f degrees' % (theta_rad*180./pi)
```

```
 0   N
 1   H
 2   H
 3   H
theta = 106.3 degrees
```

Alternatively you could use the `func:ase.Atoms.get_angle`. Note we want the angle between atoms 1-0-2 to get the H-N-H angle.

```
1   from ase import Atoms
2   from ase.data.molecules import molecule
3   from numpy import pi
4   # ammonia
5   atoms = molecule('NH3')
6
7   print atoms.get_angle([1,0,2])*180./pi
```

```
106.334624232
```

Dihedral angles There is support in ase for computing dihedral angles. Lets illustrate that for ethane. We will compute the dihedral angle between atoms 5, 1, 0, and 4. That is a H-C-C-H dihedral angle, and one can visually see (although not here) that these atoms have dihedral angle of 60 ˆ.

```
1   from ase import Atoms
2   from ase.structure import molecule
```

```python
3    from ase.visualize import view
4    import numpy as np
5
6    atoms = molecule('C2H6')
7
8    for i, atom in enumerate(atoms):
9        print '%2i %3s' % (i,atom.symbol)
10
11    da = atoms.get_dihedral([5,1,0,4])*180./np.pi
12    print 'dihedral angle = %1.2f degrees' % da
```

```
 0   C
 1   C
 2   H
 3   H
 4   H
 5   H
 6   H
 7   H
dihedral angle = 60.00 degrees
```

3.4.2 Energy and forces

Two of the most important quantities we are interested in is the total energy and the forces on the atoms. To get these, we have to define a calculator and attach it to an `mod:ase.Atoms` object so that it knows how to get the data, and then a DFT calculation must be run.

Here is an example of getting the energy and forces from a CO molecule. The forces in this case are very high,indicating that this geometry is not close to the ground state geometry. Note that the forces are only along the x-axis, which is along the molecular axis.

```python
1    #!/usr/bin/env python
2    import torque
3    from ase import Atoms, Atom
4    from ase.calculators.jacapo import Jacapo
5    import numpy as np
6
7    # this makes array printing a little more readable
8    np.set_printoptions(precision=3,suppress=True)
9
10   co = Atoms([Atom('C',[0,   0, 0]),
11              Atom('O',[1.2, 0, 0])],
12              cell=(6.,6.,6.))
13
14   calc = Jacapo('molecules/co.nc',   #output filename
15                  nbands=6,
```

```
16                pw=350,
17                ft=0.01,
18                atoms=co)
19
20   print 'Total energy = ',co.get_potential_energy()
21   print 'Forces = \n', co.get_forces()
```

```
 Total energy =  -591.081446318
 Forces =
 [[ 4.466 -0.     0.    ]
  [-4.5   -0.    -0.    ]]
```

It does not look very different in Vasp. The total energies, and forces differ, because different pseudopotentials were used and different functionals were used (default functional for Dacapo is PW91).

```
1    #!/usr/bin/env python
2    from ase import Atoms,Atom
3    from jasp import *
4    import numpy as np
5    np.set_printoptions(precision=3,suppress=True)
6
7    co = Atoms([Atom('C',[0,   0, 0]),
8               Atom('O',[1.2, 0, 0])],
9               cell=(6., 6., 6.))
10
11   with jasp('molecules/simple-co', #output dir
12           xc='PBE',
13           nbands=6,
14           encut=350,
15           ismear=1,
16           sigma=0.01,
17           atoms=co):
18
19       print co.get_potential_energy()
20       print co.get_forces()
```

```
 -14.687906
 [[ 5.095  0.     0.    ]
  [-5.095  0.     0.    ]]
```

3.4.3   The density of states

```
1    #!/usr/bin/env python
2    from ase.dft.dos import *
3    from jasp import *
4    from pylab import *
5
```

```
6   with jasp('molecules/co-qn') as calc:
7       dos = DOS(calc)
8       plot(dos.get_energies(),dos.get_dos())
9       xlabel('Energy - $E_f$ (eV)')
10      ylabel('DOS')
11  savefig('images/co-dos.png')
```
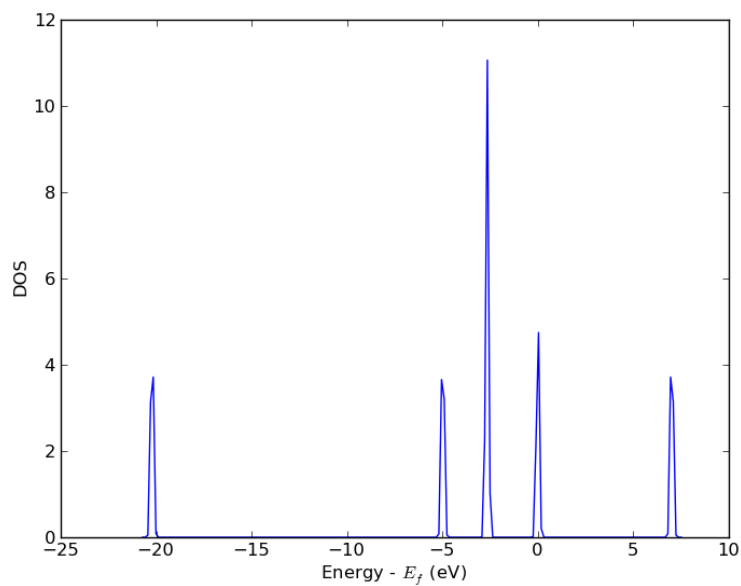

3.4.4   Visualizing electron density

```
1   #!/usr/bin/env python
2   from enthought.mayavi import mlab
3   from jasp import *
4
5   with jasp('molecules/co-qn') as calc:
6       x,y,z,cd = calc.get_charge_density()
7
8   mlab.contour3d(x,y,z,cd)
9
10  mlab.savefig('images/co-cd.png')
11
12  mlab.show()
```
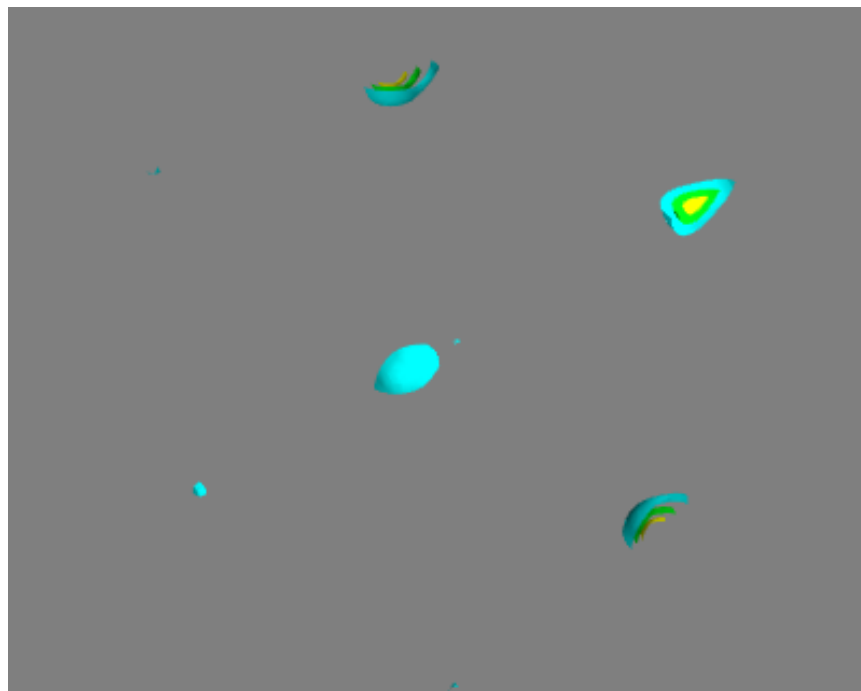
3.4.5 TODO Visualizing Kohn-Sham orbitals

need to read wavecar files!

3.4.6 TODO Bader analysis

3.4.7 Dipole moments

```
1   from jasp import *
2   from ase.calculators.vasp import *
3   from ase.units import Debye
4   from ase import Atom, Atoms
5
6   atoms = Atoms([Atom('C',[2.422,0,0]),
7                  Atom('O',[3.578,0,0])],
8                 cell=(10,10,10))
9
10  atoms.center()
11
12  with jasp('molecules/co-centered',
13            encut=350,
14            xc='PBE',
15            atoms=atoms) as calc:
16      atoms.get_potential_energy()
17
```

```
18        vcd = VaspChargeDensity()

19
20        cd = np.array(vcd.chg[0])
21        n0, n1, n2 = cd.shape

22
23        s0 = 1.0/n0
24        s1 = 1.0/n1
25        s2 = 1.0/n2

26
27        X, Y, Z = np.mgrid[0.0:1.0:s0,
28                           0.0:1.0:s1,
29                           0.0:1.0:s2]

30
31        C = np.column_stack([X.ravel(),
32                             Y.ravel(),
33                             Z.ravel()])

34
35        atoms = calc.get_atoms()
36        uc = atoms.get_cell()
37        real = np.dot(C, uc)

38
39        #now convert arrays back to unitcell shape
40        x = np.reshape(real[:, 0], (n0, n1, n2))
41        y = np.reshape(real[:, 1], (n0, n1, n2))
42        z = np.reshape(real[:, 2], (n0, n1, n2))

43
44        nelements = n0*n1*n2
45        voxel_volume = atoms.get_volume()/nelements
46        total_electron_charge = -cd.sum()*voxel_volume

47

48
49        electron_density_center = np.array([(cd*x).sum(),
50                                            (cd*y).sum(),
51                                            (cd*z).sum()])
52        electron_density_center *= voxel_volume
53        electron_density_center /= total_electron_charge

54
55        electron_dipole_moment = electron_density_center*total_electron_charge
56        electron_dipole_moment *= -1.0 #we need the - here so the two
57                                       #negatives don't cancel
58        # now the ion charge center

59
60        from jasp.POTCAR import get_ZVAL

61
62        LOP = calc.get_pseudopotentials()
63        ppp = os.environ['VASP_PP_PATH']

64
65        zval = {}
66        for sym, ppath, hash in LOP:
67            fullpath = os.path.join(ppp, ppath)
68            z = get_ZVAL(fullpath)
69            zval[sym] = z
70        ion_charge_center = np.array([0.0, 0.0, 0.0])
71        total_ion_charge = 0.0
72        for atom in atoms:
73            Z = zval[atom.symbol]
```

```
74         total_ion_charge += Z
75         pos = atom.position
76         ion_charge_center += Z*pos
77
78     ion_charge_center /= total_ion_charge
79     ion_dipole_moment = ion_charge_center*total_ion_charge
80
81     dipole_vector = (ion_dipole_moment + electron_dipole_moment)
82
83     dipole_moment = ((dipole_vector**2).sum())**0.5/Debye
84
85     print 'The dipole moment is {0:1.2f}'.format(dipole_moment)
```

Note that a convenience function using the code above exists in jasp:

```
1  from jasp import *
2  from ase.units import Debye
3
4  with jasp('molecules/co-centered') as calc:
5      dipole_vector = calc.get_dipole_moment()
6      dipole_moment = ((dipole_vector**2).sum())**0.5/Debye
7      print 'The dipole moment is {0:1.2f} Debye'.format(dipole_moment)
```

3.4.8   TODO Atom-projected density of states

3.5   Geometry optimization

3.5.1   Bond lengths

Manual determination   The equilibrium bond length of a CO molecule
is approximately the bond length that minimizes the total energy. We can
find that by computing the total energy as a function of bond length, and
noting where the minimum is. Here is an example in vasp. There are a few
features I would like to point out here. We want to compute 5 bond lengths,
and each calculation is independent of all the others. :mod:`jasp` is setup to
automatically handle jobs for you by submitting them to the queue.

```
1  #!/usr/bin/env python
2  from ase import *
3  from jasp import *
4  import numpy as np
5  np.set_printoptions(precision=3,suppress=True)
6
7  bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
8  energies = []
9
10 ready = True # flag for when all calculations are done.
11 for d in bond_lengths: #possible bond lengths
```

```
12
13      co = Atoms([Atom('C',[0,0,0]),
14                  Atom('O',[d,0,0])],
15                 cell=(6,6,6))
16
17      with jasp('molecules/co-{0}'.format(d), #output dir
18                xc='PBE',
19                nbands=6,
20                encut=350,
21                ismear=1,
22                sigma=0.01,
23                atoms=co):
24          try:
25              e = co.get_potential_energy()
26              energies.append(e)
27              print 'd = %1.2f ang' % d
28              print 'energy = %f eV' % e
29              print 'forces = (eV/ang)\n', co.get_forces()
30              print '' #blank line
31          except (VaspSubmitted, VaspQueued):
32              ready = False
33              pass
34
35  if ready:
36      import matplotlib.pyplot as plt
37      plt.plot(bond_lengths, energies, 'bo-')
38      plt.xlabel('Bond length ($\AA$)')
39      plt.ylabel('Total energy (eV)')
40      plt.savefig('molecules/co-bondlengths.png')
41      print('[[./molecules/co-bondlengths.png]]')
```

```
d = 1.05 ang
energy = -14.215189 eV
forces = (eV/ang)
[[-14.903   0.       0.    ]
 [ 14.903   0.       0.    ]]


d = 1.10 ang
energy = -14.719882 eV
forces = (eV/ang)
[[-5.8  0.    0. ]
 [ 5.8  0.    0. ]]


d = 1.15 ang
energy = -14.838448 eV
forces = (eV/ang)
[[ 0.645  0.       0.    ]
 [-0.645  0.       0.    ]]
```

```
d = 1.20 ang
energy = -14.687906 eV
forces = (eV/ang)
[[ 5.095  0.     0.    ]
 [-5.095  0.     0.    ]]

d = 1.25 ang
energy = -14.351675 eV
forces = (eV/ang)
[[ 8.141  0.     0.    ]
 [-8.141  0.     0.    ]]
```

[[./molecules/co-bondlengths.png]]

Automatic geometry optimization with vasp Vasp has built in geometry optimization using the IBRION and NSW tags. Here we compute the bond length for a CO molecule, letting Vasp do the geometry optimization for us.

```python
#!/usr/bin/env python
from ase import *
from jasp import *
import numpy as np

co = Atoms([Atom('C',[0,0,0]),
            Atom('O',[1.2,0,0])],
            cell=(6,6,6))

with jasp('molecules/co-qn',
          xc='PBE',
          nbands=6,
          encut=350,
          ismear=1,
          sigma=0.01,
          ibrion=2,
          nsw=5,  # do at least 5 steps to relax
          atoms=co) as calc:

    print co.get_forces()

    pos = co.get_positions()
    d = ((pos[0] - pos[1])**2).sum()**0.5
    print 'Bondlength = %1.2f angstroms' % d
```

```
[[ 0.003307  0.        0.       ]
```

```
      [-0.003307  0.        0.        ]]
    Bondlength = 1.14 angstroms
```

3.6 Vibrational frequencies

3.6.1 Manual calculation of vibrational frequency

The principle idea in calculating vibrational frequencies is that we consider a molecular system as masses connected by springs. If the springs are Hookean, e.g. the force is proportional to the displacement, then we can readily solve the equations of motion and find that the vibrational frequencies are related to the force constants and the masses of the atoms. For example, in a simple molecule like CO where there is only one spring, the frequency is:

$\nu = \frac{1}{2\pi}\sqrt{k/\mu}$ where $\frac{1}{mu} = \frac{1}{m_C} + \frac{1}{m_O}$ and k is the spring constant. We will compute the value of k from DFT calculations as follows:

$k = \frac{\partial^2 E}{\partial x^2}$ at the equilibrium bond length. We actually already have the data to do this from section Manual determination. We only need to fit an equation to the energy vs. bond-length data, find the minimum energy bond-length, and then evaluate the second derivative of the fitted function at the minimum. We will use a cubic polynomial for demonstration here. Polynomials are numerically convenient because they are easy to fit, and it is trivial to get the roots and derivatives of the polynomials, as well as to evaluate them at other points.

```python
1   from jasp import *
2   from ase.units import *
3
4   bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
5   energies = []
6
7   for d in bond_lengths:
8       with jasp('molecules/co-{0}'.format(d)) as calc:
9           atoms = calc.get_atoms()
10          energies.append(atoms.get_potential_energy())
11
12  # fit the data
13  pars = np.polyfit(bond_lengths, energies, 3)
14  xfit = np.linspace(1.05, 1.25)
15  efit = np.polyval(pars, xfit)
16
17  # first derivative
18  dpars = np.polyder(pars)
19  # find where the minimum is. chose the second one because it is the
20  # minimum we need.
21  print 'roots of first derivative are {0}'.format(np.roots(dpars))
22
```

```
23   d_min = 1.14425395
24   # second derivative
25   ddpars = np.polyder(dpars)
26
27   #curvature at minimum = force constant
28   k = np.polyval(ddpars, d_min)/kg*s**2
29
30   # reduced mass
31   from ase.data import atomic_masses
32   C_mass = atomic_masses[6]/kg
33   O_mass = atomic_masses[8]/kg
34
35   mu = (C_mass*O_mass)/(C_mass + O_mass)
36
37   frequency = 1./(2.*np.pi)*np.sqrt(k/mu)
38   print 'The CO vibrational frequency is {0} cm^{{-1}}'.format(frequency/3e10)
39
40   import matplotlib.pyplot as plt
41   plt.plot(bond_lengths, energies, 'bo ')
42   plt.plot(xfit, efit, 'b-')
43   plt.xlabel('Bond length ($\AA$)')
44   plt.ylabel('Total energy (eV)')
45   plt.show()
```

This result is in good agreement with experiment. The procedure used above is how many vibrational calculations are done. With more atoms, you have to determine a force constant matrix and diagonalize it. For more details, see [6]. In practice, we usually allow a packaged code to automate this, which we cover in the next section.

A final note is that this analysis is in the "harmonic approximation". The frequency equation is the solution to a harmonic oscillator. If the spring is non-linear, then there are anharmonic effects that may become important, especially at higher temperatures.

3.6.2 Automated vibrational calculations

Vasp has built-in capability for performing vibrational calculations.

```
1    #adapted from http://cms.mpi.univie.ac.at/wiki/index.php/H2O_vibration
2    from ase import Atoms, Atom
3    from jasp import *
4
5    atoms = Atoms([Atom('H',[0.5960812,  -0.7677068,   0.0000000]),
6                   Atom('O',[0.0000000,   0.0000000,   0.0000000]),
7                   Atom('H',[0.5960812,   0.7677068,   0.0000000])],
8                   cell=(8,8,8))
9
10   with jasp('molecules/h2o_vib',
11             xc='PBE',
12             encut=400,
```

```
13            ismear=0,# Gaussian smearing
14            ibrion=6,# finite differences with symmetry
15            nfree=2, # central differences (default)
16            potim=0.015,# default as well
17            ediff=1e-8,
18            nsw=1,
19            atoms=atoms) as calc:
20
21      print atoms.get_forces()
22      print calc.get_vibrational_frequencies()
```

```
[[ 0.006474 -0.021573 -0.001509]
 [-0.012948  0.        0.003019]
 [ 0.006474  0.021573 -0.001509]]
[3836.1533119999999, 3722.659114, 1583.004226, 35.646208999999999, 21.37895100000000
```

Note we get 9 frequencies here. Water has 3 atoms, with three degrees of freedom each. Three of those degrees of freedom are translations, and three are rotations. That leaves 3N-6 = 3 degrees of vibrational freedom. The modes of water vibration are (with our calculated values in parentheses):

1. a symmetric stretch at 3657 cm^{-1} (3723)

2. an asymmetric stretch at 3756 cm^{-1} (3836)

3. and a bending mode at 1595 cm^{-1} (1583)

http://webbook.nist.gov/cgi/cbook.cgi?ID=C7732185&Mask=800#Electronic-Spec

The results are not too far off, and more accurate frequencies may be possible using tighter tolerance on POTIM, or by using IBRION=7 or 8.

Note: the calc.get$_{vibrationalfrequencies}$() is only available in jasp (7/12/2012).

3.7 Thermochemical properties of molecules

ase provides a thermochemistry module.

```
1  from ase.structure import molecule
2  from ase.thermochemistry import IdealGasThermo
3  from jasp import *
4
5  atoms = molecule('N2')
6  atoms.set_cell((10,10,10), scale_atoms=False)
7
8  # first we relax a molecule
9  with jasp('molecules/n2-relax',
```

```
10             xc='PBE',
11             encut=300,
12             ibrion=2,
13             nsw=5,
14             atoms=atoms) as calc:
15
16        electronicenergy = atoms.get_potential_energy()
17
18    # next, we get vibrational modes
19    with jasp('molecules/n2-vib',
20             xc='PBE',
21             encut=300,
22             ibrion=6,
23             nfree=2,
24             potim=0.15,
25             nsw=1,
26             atoms=atoms) as calc:
27        calc.calculate()
28        vib_freq = calc.get_vibrational_frequencies() # in cm^1
29
30        #convert wavenumbers to energy
31        h = 4.1356675e-15 # eV*s
32        c = 3.0e10 #cm/s
33        vib_energies = [h*c*nu for nu in vib_freq]
34        print vib_energies
35
36    # # now we can get some properties. Note we only need one vibrational
37    # energy since there is only one mode. This example does not work if
38    # you give all the energies because one energy is zero.
39    thermo = IdealGasThermo(vib_energies=vib_energies[0:0],
40                          electronicenergy=electronicenergy, atoms=atoms,
41                          geometry='linear', symmetrynumber=2, spin=0)
42
43    # temperature in K, pressure in Pa, G in eV
44    G = thermo.get_free_energy(temperature=298.15, pressure=101325.)
```

```
[0.29159234323953859, 0.016977524105896053, 0.016977524105896053, 2.8536105750000002e
Enthalpy components at T = 298.15 K:
==============================
E_elec                  -16.478 eV
E_ZPE                     0.000 eV
Cv_trans (0->T)           0.039 eV
Cv_rot (0->T)             0.026 eV
Cv_vib (0->T)             0.000 eV
(C_v -> C_p)              0.026 eV
------------------------------
H                       -16.388 eV
==============================


Entropy components at T = 298.15 K and P = 101325.0 Pa:
```

```
==================================================
                         S                T*S
S_trans (1 atm)     0.0015579 eV/K          0.464 eV
S_rot               0.0007870 eV/K          0.235 eV
S_elec              0.0000000 eV/K          0.000 eV
S_vib               0.0000000 eV/K          0.000 eV
S (1 atm -> P)     -0.0000000 eV/K         -0.000 eV
--------------------------------------------------
S                   0.0023449 eV/K          0.699 eV
==================================================


Free energy components at T = 298.15 K and P = 101325.0 Pa:
=======================
    H          -16.388 eV
 -T*S           -0.699 eV
-----------------------
    G          -17.087 eV
=======================
```

3.8 Molecular reaction energies

3.8.1 O2 dissociation

The first reaction we consider is a simple dissociation of oxygen molecule into two oxygen atoms: $O_2 \rightarrow 2O$. The dissociation energy is pretty straightforward to define: it is the energy of the products minus the energy of the reactant. $D = 2 * E_O - E_{O_2}$. It would appear that we simply calculate the energy of an oxygen atom, and an oxygen molecule and evaluate the formula. Let's do that.

Simple estimate of O2 dissociation energy

```
1   from jasp import *
2   from ase import Atom, Atoms
3
4   atoms = Atoms([Atom('O',[5,5,5])],
5                 cell=(10,10,10))
6
7   with jasp('molecules/O',
8             xc='PBE',
9             encut=400,
10            ismear=0,
11            atoms=atoms) as calc:
12        try:
```

```
13              E_O = atoms.get_potential_energy()
14        except (VaspSubmitted, VaspQueued):
15              E_O = None
16
17    # now relaxed O2 dimer
18
19    atoms = Atoms([Atom('O',[5,5,5]),
20                    Atom('O',[6.22, 5,5])],
21                   cell=(10,10,10))
22
23    with jasp('molecules/O2',
24              xc='PBE',
25              encut=400,
26              ismear=0,
27              ibrion=2, # make sure we relax the geometry
28              nsw=10,
29              atoms=atoms) as calc:
30        try:
31              E_O2 = atoms.get_potential_energy()
32        except (VaspSubmitted, VaspQueued):
33              E_O2 = None
34
35    if None not in (E_O, E_O2):
36        print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)
```

The answer we have obtained is way too high! Experimentally
We implicitly neglected spin-polarization in the example above. That could be a problem, since the O2 molecule can be in one of two spin states, a singlet or a triplet, and these should have different energies. Furthermore, the oxygen atom can be a singlet or a triplet, and these would have different energies. To account for spin polarization, we have to tell Vasp to use spin-polarization, and give initial guesses for the magnetic moments of the atoms. Let's try again with spin polarization.

Estimating O2 dissociation energy with spin polarization in triplet ground states To tell Vasp to use spin-polarization we use ISPIN=2, and we set initial guesses for magnetic moments on the atoms with the magmom keyword. In a triplet state there are two electrons with spins of the same sign.

```
1    from jasp import *
2    from ase import Atom, Atoms
3
4    atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
5                   cell=(10,10,10))
6
7    with jasp('molecules/O-sp-triplet',
8              xc='PBE',
9              encut=400,
```

```
10              ismear=0,
11              ispin=2,
12              atoms=atoms) as calc:
13      try:
14          E_O = atoms.get_potential_energy()
15      except (VaspSubmitted, VaspQueued):
16          E_O = None
17
18  print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20  # now relaxed O2 dimer
21  atoms = Atoms([Atom('O',[5,5,5],magmom=1),
22                Atom('O',[6.22, 5,5],magmom=1)],
23                cell=(10,10,10))
24
25  with jasp('molecules/O2-sp-triplet',
26           xc='PBE',
27           encut=400,
28           ismear=0,
29           ispin=2,   # turn spin-polarization on
30           ibrion=2, # make sure we relax the geometry
31           nsw=10,
32           atoms=atoms) as calc:
33      try:
34          E_O2 = atoms.get_potential_energy()
35      except (VaspSubmitted, VaspQueued):
36          E_O2 = None
37
38  # verify magnetic moment
39  print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
40
41  if None not in (E_O, E_O2):
42      print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)
```

This is much closer to accepted literature values for the DFT-GGA O2
dissociation energy. It is still more than 1 eV above an experimental value,
but most of that error is due to the GGA exchange correlation functional.
Some additional parameters that might need to be checked for convergence
are the SIGMA value (it is probably too high for a molecule), as well as the
cutoff energy. Oxygen is a "hard" atom that requires a high cutoff energy
to achieve high levels of convergence.

Looking at the two spin densities   In a spin-polarized calculation
there are actually two electron densities: one for spin-up and one for spin-
down. We will look at the differences in these two through the density of
states.

```
1  from jasp import *
2  from ase.dft.dos import *
```

```
3
4    with jasp('molecules/O2-sp-triplet') as calc:
5        dos = DOS(calc, width=0.2)
6        d_up = dos.get_dos(spin=0)
7        d_down = dos.get_dos(spin=1)
8        e = dos.get_energies()
9
10   ind = e <= 0.0
11   # integrate up to 0eV
12   print 'number of up states = {0}'.format(np.trapz(d_up[ind],e[ind]))
13   print 'number of down states = {0}'.format(np.trapz(d_down[ind],e[ind]))
14
15   import pylab as plt
16   plt.plot(e, d_up, e, -d_down)
17   plt.xlabel('energy [eV]')
18   plt.ylabel('DOS')
19   plt.legend(['up','down'])
20   plt.savefig('images/O2-sp-dos.png')
```
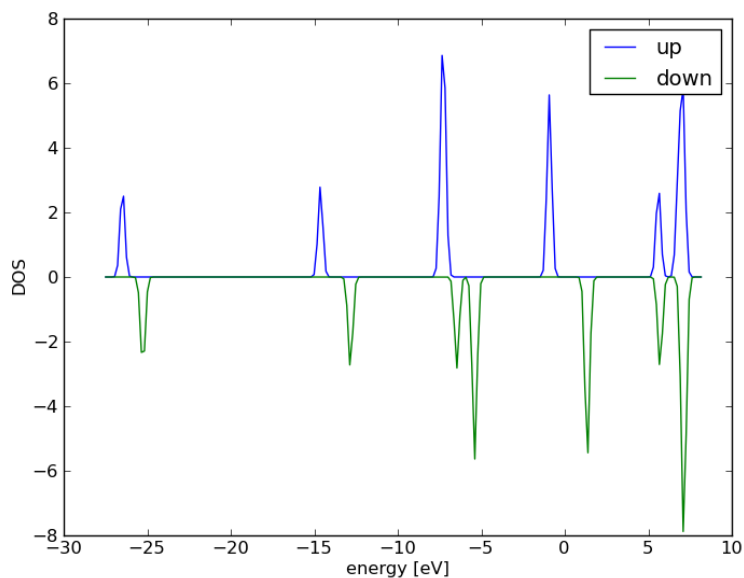

You can see there are two different densities of states for the two spins. One has 7 electrons in it, and the other has 5 electrons in it. The difference of two electrons leads to the magnetic moment of 2 which we calculated earlier.

Convergence study of the O2 dissociation energy

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '10:00:00'
3   from ase import Atom, Atoms
4   encuts = [250, 300, 350, 400, 450, 500, 550]
5
6   D = []
7   for encut in encuts:
8       atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
9                     cell=(10,10,10))
10
11      with jasp('molecules/O-sp-triplet-{0}'.format(encut),
12                xc='PBE',
13                encut=encut,
14                ismear=0,
15                ispin=2,
16                atoms=atoms) as calc:
17          try:
18              E_O = atoms.get_potential_energy()
19          except (VaspSubmitted, VaspQueued):
20              E_O = None
21
22      # now relaxed O2 dimer
23      atoms = Atoms([Atom('O',[5,5,5],magmom=1),
24                     Atom('O',[6.22, 5,5],magmom=1)],
25                cell=(10,10,10))
26
27      with jasp('molecules/O2-sp-triplet-{0}'.format(encut),
28                xc='PBE',
29                encut=encut,
30                ismear=0,
31                ispin=2,   # turn spin-polarization on
32                ibrion=2, # make sure we relax the geometry
33                nsw=10,
34                atoms=atoms) as calc:
35          try:
36              E_O2 = atoms.get_potential_energy()
37          except (VaspSubmitted, VaspQueued):
38              E_O2 = None
39
40      if None not in (E_O, E_O2):
41          d = 2*E_O - E_O2
42          D.append(d)
43          print 'O2 -> 2O encut = {0}  D = {1:1.3f} eV'.format(encut, d)
44
45  import matplotlib.pyplot as plt
46  plt.plot(encuts, D)
47  plt.xlabel('ENCUT (eV)')
48  plt.ylabel('O$_2$ dissociation energy (eV)')
49  plt.savefig('images/O2-dissociation-convergence.png')
```
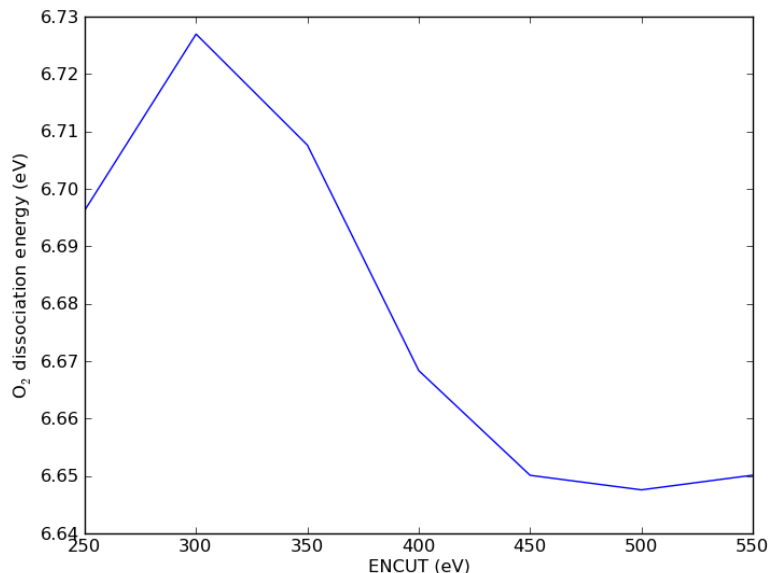
Based on these results, you could argue the dissociation energy is converged to about 2 meV at a planewave cutoff of 450 eV, and within 50 meV at 350 eV cutoff. You have to decide what an appropriate level of convergence is. Note that increasing the planewave cutoff significantly increases the computational time, so you are balancing level of convergence with computational speed.

```
1   encuts = [250, 300, 350, 400, 450, 500, 550]
2
3   for encut in encuts:
4       OUTCAR = 'molecules/O2-sp-triplet-{0}/OUTCAR'.format(encut)
5       f = open(OUTCAR, 'r')
6       for line in f:
7           if 'Total CPU time used (sec)' in line:
8               print '{0}: {1}'.format(encut, line)
9       f.close()
```

Estimating triplet oxygen dissociation energy with low symmetry
It has been suggested that breaking spherical symmetry of the atom can result in lower energy of the atom. The symmetry is broken by putting the atom off-center in a box.

```
1   from jasp import *
2   from ase import Atom, Atoms
```

```
3
4   atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
5                 cell=(8,9,10))
6
7   with jasp('molecules/O-sp-triplet-lowsym',
8             xc='PBE',
9             encut=400,
10            ismear=0,
11            ispin=2,
12            atoms=atoms) as calc:
13      try:
14          E_O = atoms.get_potential_energy()
15      except (VaspSubmitted, VaspQueued):
16          E_O = None
17
18  print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20  # now relaxed O2 dimer
21  atoms = Atoms([Atom('O',[5,5,5],magmom=1),
22                 Atom('O',[6.22, 5,5],magmom=1)],
23                 cell=(10,10,10))
24
25  with jasp('molecules/O2-sp-triplet',
26            xc='PBE',
27            encut=400,
28            ismear=0,
29            ispin=2,   # turn spin-polarization on
30            ibrion=2, # make sure we relax the geometry
31            nsw=10,
32            atoms=atoms) as calc:
33      try:
34          E_O2 = atoms.get_potential_energy()
35      except (VaspSubmitted, VaspQueued):
36          E_O2 = None
37
38  # verify magnetic moment
39  print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
40
41  if None not in (E_O, E_O2):
42      print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)
```

The energy difference between this 6.668 and 6.657 eV is only 11 meV!

Estimating singlet oxygen dissociation energy Finally, let us consider the case where each species is in the singlet state.

```
1   from jasp import *
2   from ase import Atom, Atoms
3
4   atoms = Atoms([Atom('O',[5,5,5],magmom=0)],
5                 cell=(10,10,10))
6
7   with jasp('molecules/O-sp-singlet',
```

```
 8              xc='PBE',
 9              encut=400,
10              ismear=0,
11              ispin=2,
12              atoms=atoms) as calc:
13       try:
14           E_O = atoms.get_potential_energy()
15       except (VaspSubmitted, VaspQueued):
16           E_O = None
17
18  print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20  # now relaxed O2 dimer
21  atoms = Atoms([Atom('O',[5,5,5],magmom=1),
22                Atom('O',[6.22, 5,5],magmom=-1)],
23                cell=(10,10,10))
24
25  with jasp('molecules/O2-sp-singlet',
26            xc='PBE',
27            encut=400,
28            ismear=0,
29            ispin=2,  # turn spin-polarization on
30            ibrion=2, # make sure we relax the geometry
31            nsw=10,
32            atoms=atoms) as calc:
33       try:
34           E_O2 = atoms.get_potential_energy()
35       except (VaspSubmitted, VaspQueued):
36           E_O2 = None
37
38  # verify magnetic moment
39  print atoms.get_magnetic_moment()
40
41  if None not in (E_O, E_O2):
42      print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)
```

Interestingly, Vasp still found a triplet spin state on the oxygen atom, even though we guessed an initial magnetic moment of 0. This highlights a difficulty in computing magnetic moments: you provide an initial guess and a solution is found. The magnetic moment of a singlet state is zero, so the molecule is correct. Also interesting is that the dissociation energy is almost equal to the experimental value. This is probably a coincidence, and may reflect the fact that the singlet oxygen state is less stable than the triplet state. Let us directly compare their total energies:

```
1  from jasp import *
2
3  with jasp('molecules/O2-sp-singlet') as calc:
4      print calc.get_atoms().get_potential_energy()
5
6  with jasp('molecules/O2-sp-triplet') as calc:
7      print calc.get_atoms().get_potential_energy()
```

You can see here the triplet state has an energy that is 1 eV more stable than the singlet state.

3.8.2 Water gas shift example

We consider calculating the reaction energy of the water-gas shift reaction in this example.

$CO + H_2O \rightleftharpoons CO_2 + H_2$
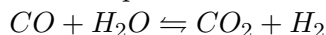
We define the reaction energy as the difference in energy between the products and reactants.

$\Delta E = E_{CO_2} + E_{H_2} - E_{CO} - E_{H_2O}$

For now, we compute this energy simply as the difference in DFT energies. In the next section we will add zero-point energies and compute the energy difference as a function of temperature. For now, we simply need to compute the total energy of each molecule in its equilibrium geometry.

```
1   from ase.data.molecules import molecule
2   from jasp import *
3   JASPRC['queue.walltime'] = '1:00:00'
4
5   # first we define our molecules. These will automatically be at the coordinates from the G2 database.
6
7   CO =  molecule('CO')
8   CO.set_cell([8,8,8], scale_atoms=False)
9
10  H2O = molecule('H2O')
11  H2O.set_cell([8,8,8], scale_atoms=False)
12
13  CO2 =  molecule('CO2')
14  CO2.set_cell([8,8,8], scale_atoms=False)
15
16  H2 = molecule('H2')
17  H2.set_cell([8,8,8], scale_atoms=False)
18
19  # now the calculators to get the energies
20  with jasp('molecules/wgs/CO',
21            xc='PBE',
22            encut=350,
23            ismear=0,
24            ibrion=2,
25            nsw=10,
26            atoms=CO) as calc:
27      try:
28          eCO = CO.get_potential_energy()
29      except (VaspSubmitted, VaspQueued):
30          eCO = None
31
```

```
32  with jasp('molecules/wgs/CO2',
33          xc='PBE',
34          encut=350,
35          ismear=0,
36          ibrion=2,
37          nsw=10,
38          atoms=CO2) as calc:
39      try:
40          eCO2 = CO2.get_potential_energy()
41      except (VaspSubmitted, VaspQueued):
42          eCO2 = None
43
44  with jasp('molecules/wgs/H2',
45          xc='PBE',
46          encut=350,
47          ismear=0,
48          ibrion=2,
49          nsw=10,
50          atoms=H2) as calc:
51      try:
52          eH2 = H2.get_potential_energy()
53      except (VaspSubmitted, VaspQueued):
54          eH2 = None
55
56  with jasp('molecules/wgs/H2O',
57          xc='PBE',
58          encut=350,
59          ismear=0,
60          ibrion=2,
61          nsw=10,
62          atoms=H2O) as calc:
63      try:
64          eH2O = H2O.get_potential_energy()
65      except (VaspSubmitted, VaspQueued):
66          eH2O = None
67
68  if None in (eCO2, eH2, eCO, eH2O):
69      pass
70  else:
71      dE = eCO2 + eH2 - eCO - eH2O
72      print '\Delta E = {0:1.3f} eV'.format(dE)
73      print '\Delta E = {0:1.3f} kcal/mol'.format(dE*23.06035)
74      print '\Delta E = {0:1.3f} kJ/mol'.format(dE*96.485)
```

We estimated the enthalpy of this reaction at standard conditions to be -41 kJ/mol, which is a fair bet lower than we estimated. In the next section we will examine whether additional corrections are needed, such as zero-point and temperature corrections.

It is a good idea to verify your calculations and structures are what you expected. Let us print them here. Inspection of these results shows the geometries were all relaxed.

```
1   from jasp import *
2
3   print '**** calculation summaries'
4   print '***** CO'
5   with jasp('molecules/wgs/CO') as calc:
6       print '#+begin_example'
7       print calc
8       print '#+end_example'
9
10  print '***** CO2'
11  with jasp('molecules/wgs/CO2') as calc:
12      print '#+begin_example'
13      print calc
14      print '#+end_example'
15
16  print '***** H2'
17  with jasp('molecules/wgs/H2') as calc:
18      print '#+begin_example'
19      print calc
20      print '#+end_example'
21
22  print '***** H2O'
23  with jasp('molecules/wgs/H2O') as calc:
24      print '#+begin_example'
25      print calc
26      print '#+end_example'
```

3.8.3 Temperature dependent water gas shift equilibrium constant

To correct the reaction energy for temperature effects, we must compute the vibrational frequencies of each species, and estimate the temperature dependent contributions to vibrational energy and entropy. We will break these calculations into several pieces. First we do each vibrational calculation. After those are done, we can get the data and construct the thermochemistry objects we need to estimate the reaction energy as a function of temperature (at constant pressure).

CO vibrations

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '1:00:00'
3
4   # get relaxed geometry
5   with jasp('molecules/wgs/CO') as calc:
6       CO = calc.get_atoms()
7
8   # now do the vibrations
9   with jasp('molecules/wgs/CO-vib',
10          xc='PBE',
```

```
11              encut=350,
12              ismear=0,
13              ibrion=6,
14              nfree=2,
15              potim=0.02,
16              nsw=1,
17              atoms=CO) as calc:
18      calc.calculate()
19      vib_freq = calc.get_vibrational_frequencies()
20      print vib_freq
```

CO has only one vibrational mode.

CO2 vibrations

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '1:00:00'
3
4   # get relaxed geometry
5   with jasp('molecules/wgs/CO2') as calc:
6       CO2 = calc.get_atoms()
7
8   # now do the vibrations
9   with jasp('molecules/wgs/CO2-vib',
10              xc='PBE',
11              encut=350,
12              ismear=0,
13              ibrion=6,
14              nfree=2,
15              potim=0.02,
16              nsw=1,
17              atoms=CO2) as calc:
18      calc.calculate()
19      vib_freq = calc.get_vibrational_frequencies()
20      print vib_freq
```

CO2 is a linear molecule with 3N-5 = 4 vibrational modes. They are the first four frequencies.

H2 vibrations

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '1:00:00'
3
4   # get relaxed geometry
5   with jasp('molecules/wgs/H2') as calc:
6       H2 = calc.get_atoms()
7
8   # now do the vibrations
9   with jasp('molecules/wgs/H2-vib',
10              xc='PBE',
```

```
11              encut=350,
12              ismear=0,
13              ibrion=6,
14              nfree=2,
15              potim=0.02,
16              nsw=1,
17              atoms=H2) as calc:
18      calc.calculate()
19      vib_freq = calc.get_vibrational_frequencies()
20      print vib_freq
```

There is only one frequency of importance (the one at 4281 cm^{-1}) for the linear H2 molecule.

H2O vibrations

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '1:00:00'
3
4   # get relaxed geometry
5   with jasp('molecules/wgs/H2O') as calc:
6       H2O = calc.get_atoms()
7
8   # now do the vibrations
9   with jasp('molecules/wgs/H2O-vib',
10          xc='PBE',
11          encut=350,
12          ismear=0,
13          ibrion=6,
14          nfree=2,
15          potim=0.02,
16          nsw=1,
17          atoms=H2O) as calc:
18      calc.calculate()
19      vib_freq = calc.get_vibrational_frequencies()
20      print vib_freq
```

Water has 3N-6 = 3 vibrational modes.

TODO thermochemistry   Now we are ready. We have the electronic energies and vibrational frequencies of each species in the reaction.

```
1   from ase.thermochemistry import IdealGasThermo
2   from jasp import *
3   import numpy as np
4   import matplotlib.pyplot as plt
5
6   # first we get the electronic energies
7   with jasp('molecules/wgs/CO') as calc:
8       CO = calc.get_atoms()
```

```
9        E_CO = CO.get_potential_energy()

10
11   with jasp('molecules/wgs/CO2') as calc:
12        CO2 = calc.get_atoms()
13        E_CO2 = CO2.get_potential_energy()

14
15   with jasp('molecules/wgs/H2') as calc:
16        H2 = calc.get_atoms()
17        E_H2 = H2.get_potential_energy()

18
19   with jasp('molecules/wgs/H2O') as calc:
20        H2O = calc.get_atoms()
21        E_H2O = H2O.get_potential_energy()

22
23   # now we get the vibrational energies
24   h = 4.1356675e-15 # eV*s
25   c = 3.0e10 #cm/s

26
27   with jasp('molecules/wgs/CO-vib') as calc:
28        vib_freq = calc.get_vibrational_frequencies()
29        CO_vib_energies = [h*c*nu for nu in vib_freq]

30
31   with jasp('molecules/wgs/CO2-vib') as calc:
32        vib_freq = calc.get_vibrational_frequencies()
33        CO2_vib_energies = [h*c*nu for nu in vib_freq]

34
35   with jasp('molecules/wgs/H2-vib') as calc:
36        vib_freq = calc.get_vibrational_frequencies()
37        H2_vib_energies = [h*c*nu for nu in vib_freq]

38
39   with jasp('molecules/wgs/H2O-vib') as calc:
40        vib_freq = calc.get_vibrational_frequencies()
41        H2O_vib_energies = [h*c*nu for nu in vib_freq]

42
43   # now we make a thermo object for each molecule
44   CO_t = IdealGasThermo(vib_energies=CO_vib_energies[0:0],
45                         electronicenergy=E_CO, atoms=CO,
46                         geometry='linear', symmetrynumber=1,
47                         spin=0)

48
49   CO2_t = IdealGasThermo(vib_energies=CO2_vib_energies[0:4],
50                         electronicenergy=E_CO2, atoms=CO2,
51                         geometry='linear', symmetrynumber=2,
52                         spin=0)

53
54   H2_t = IdealGasThermo(vib_energies=H2_vib_energies[0:0],
55                         electronicenergy=E_H2, atoms=H2,
56                         geometry='linear', symmetrynumber=2,
57                         spin=0)

58
59   H2O_t = IdealGasThermo(vib_energies=H2O_vib_energies[0:3],
60                         electronicenergy=E_H2O, atoms=H2O,
61                         geometry='nonlinear', symmetrynumber=2,
62                         spin=0)

63
64   # now we can compute G_rxn for a range of temperatures from 200 to 1000 K
```

```
65  Trange = np.linspace(200,1000,20) #K
66  P = 101325. # Pa
67  Grxn = np.array([(CO2_t.get_free_energy(temperature=T, pressure=P)
68                    + H2_t.get_free_energy(temperature=T, pressure=P)
69                    - H2O_t.get_free_energy(temperature=T, pressure=P)
70                    - CO_t.get_free_energy(temperature=T, pressure=P))*96.485 for T in Trange])
71
72  Hrxn = np.array([(CO2_t.get_enthalpy(temperature=T)
73                    + H2_t.get_enthalpy(temperature=T)
74                    - H2O_t.get_enthalpy(temperature=T)
75                    - CO_t.get_enthalpy(temperature=T))*96.485 for T in Trange])
76
77  plt.plot(Trange, Grxn, 'bo-',label='$\Delta G_{rxn}$')
78  plt.plot(Trange, Hrxn, 'ro:',label='$\Delta H_{rxn}$')
79  plt.xlabel('Temperature (K)')
80  plt.ylabel('$\Delta G_{rxn}$ (kJ/mol)')
81  plt.legend(loc='best')
82  plt.savefig('images/wgs-dG-T.png')
83
84  plt.figure()
85  R = 8.314e-3 # gas constant in kJ/mol/K
86
87  Keq = np.exp(-Grxn/R/Trange)
88  plt.plot(Trange, Keq)
89  plt.ylim([0, 100])
90  plt.xlabel('Temperature (K)')
91  plt.ylabel('$K_{eq}$')
92  plt.savefig('images/wgs-Keq.png')
93  plt.show()
```
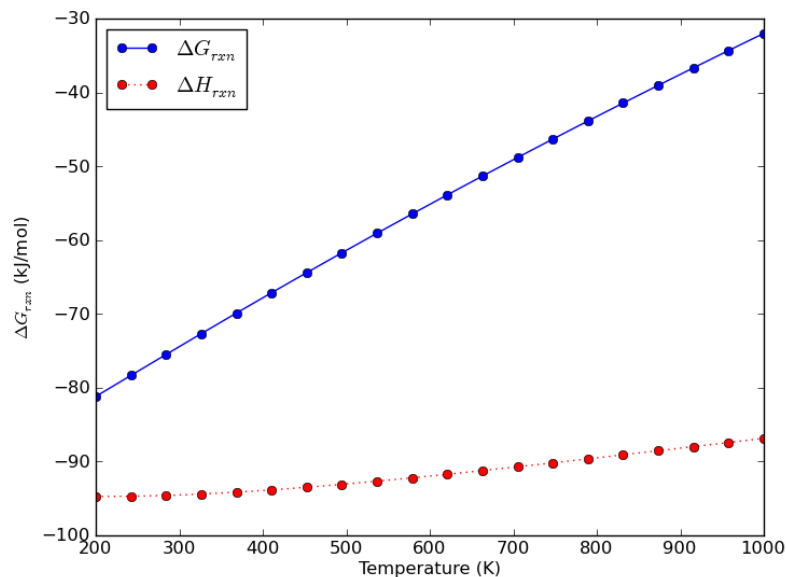
You can see a few things here. One is that at near 298K, the Gibbs free energy is about -40 kJ/mol. This is too negative for a standard state free energy, which we estimated to be about -29 kJ/mol from the Nist webbook. There could be several reasons for this disagreement, but the most likely one is errors in the exchange-correlation functional. Second, it appears the reaction enthalpy gets slightly more exothermic with temperature. This does not seem correct.

At 1000K we estimate the Gibbs free energy to be about -4 kJ/mol, compared to about -3 kJ/mol estimated from the Nist webbook.
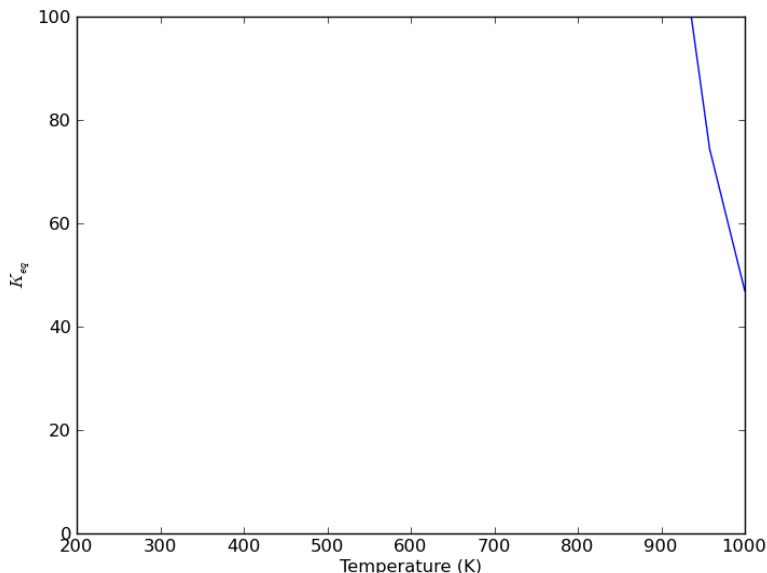
Despite the error in ΔG, the equilibrium constant at 1000K seems reasonable. If you zoom in to 1000K you find the K_{eq} is about 1.5, and it is known to be 1.44 (see this example).

TODO something is wrong with the thermochemistry I think.

3.9 TODO Molecular reaction barriers

3.10 Simulated infrared spectra

At http://homepage.univie.ac.at/david.karhanek/downloads.html#Entry02 there is a recipe for computing the Infrared vibrational spectroscopy intensities in Vasp. We are going to do that for water here. First, we will relax a water molecule.

```
1  from ase import Atoms, Atom
2  from jasp import *
3
4  atoms = Atoms([Atom('H',[0.5960812,  -0.7677068,   0.0000000]),
5                 Atom('O',[0.0000000,   0.0000000,   0.0000000]),
6                 Atom('H',[0.5960812,   0.7677068,   0.0000000])],
7                 cell=(8,8,8))
8
9  with jasp('molecules/h2o_relax',
10             xc='PBE',
```

```
11              encut=400,
12              ismear=0,# Gaussian smearing
13              ibrion=2,
14              ediff=1e-8,
15              nsw=10,
16              atoms=atoms) as calc:
17
18          print atoms.get_forces()
```

```
 [[ 0.00033  -0.000422  0.      ]
  [-0.00066   0.        0.      ]
  [ 0.00033   0.000422  0.      ]]
```

Next, we instruct Vasp to compute the vibrational modes using density functional perturbation theory with IBRION=7. Note, this is different than in Vibrational frequencies where finite differences were used.

```
1   from ase import Atoms, Atom
2   from jasp import *
3
4   #read in relaxed geometry
5   with jasp('molecules/h2o_relax') as calc:
6       atoms = calc.get_atoms()
7
8   # now define a new calculator
9   with jasp('molecules/h2o_vib_dfpt',
10             xc='PBE',
11             encut=400,
12             ismear=0,# Gaussian smearing
13             ibrion=7, # switches on the DFPT vibrational analysis (with no symmetry constraints)
14             nfree=2,
15             potim=0.015,
16             lepsilon=True, # enables to calculate and to print the BEC tensors
17             lreal=False,
18             nsw=1,
19             nwrite=3, # affects OUTCAR verbosity: explicitly forces SQRT(mass)-divided eigenvectors to be printed
20             atoms=atoms) as calc:
21      calc.calculate(atoms)
```

To analyze the results, this shell script was provided to extract the results.

```
1   #!/bin/bash
2   # A utility for calculating the vibrational intensities from VASP output (OUTCAR)
3   # (C) David Karhanek, 2011-03-25, ICIQ Tarragona, Spain (www.iciq.es)
4
5   # extract Born effective charges tensors
6   printf "..reading OUTCAR"
7   BORN_NROWS=`grep NIONS OUTCAR | awk '{print $12*4+1}'`
```

```bash
 8  if [ `grep 'BORN' OUTCAR | wc -l` = 0 ] ; then \
 9      printf " .. FAILED! Born effective charges missing! Bye! \n\n" ; exit 1 ; fi
10  grep "in e, cummulative" -A $BORN_NROWS OUTCAR > born.txt
11
12  # extract Eigenvectors and eigenvalues
13  if [ `grep 'SQRT(mass)' OUTCAR | wc -l` != 1 ] ; then \
14      printf " .. FAILED! Restart VASP with NWRITE=3! Bye! \n\n" ; exit 1 ; fi
15  EIG_NVIBS=`grep -A 2000 'SQRT(mass)' OUTCAR | grep 'cm-1' | wc -l`
16  EIG_NIONS=`grep NIONS OUTCAR | awk '{print $12}'`
17  EIG_NROWS=`echo "($EIG_NIONS+3)*$EIG_NVIBS+3" | bc`
18  grep -A $(($EIG_NROWS+2)) 'SQRT(mass)' OUTCAR | tail -n $(($EIG_NROWS+1)) | sed 's/f\/i/fi /g' > eigenvectors.txt
19  printf " ..done\n"
20
21  # set up a new directory, split files - prepare for parsing
22  printf "..splitting files"
23  mkdir intensities ; mv born.txt eigenvectors.txt intensities/
24  cd intensities/
25  let NBORN_NROWS=BORN_NROWS-1
26  let NEIG_NROWS=EIG_NROWS-3
27  let NBORN_STEP=4
28  let NEIG_STEP=EIG_NIONS+3
29  tail -n $NBORN_NROWS born.txt > temp.born.txt
30  tail -n $NEIG_NROWS eigenvectors.txt > temp.eige.txt
31  mkdir inputs ; mv born.txt eigenvectors.txt inputs/
32  split -a 3 -d -l $NEIG_STEP temp.eige.txt temp.ei.
33  split -a 3 -d -l $NBORN_STEP temp.born.txt temp.bo.
34  mkdir temps01 ; mv temp.born.txt temp.eige.txt temps01/
35  for nu in `seq 1 $EIG_NVIBS` ; do
36   let nud=nu-1 ; ei=`printf "%03u" $nu` ; eid=`printf "%03u" $nud` ; mv temp.ei.$eid eigens.vib.$ei
37  done
38  for s in `seq 1 $EIG_NIONS` ; do
39   let sd=s-1 ; bo=`printf "%03u" $s` ; bod=`printf "%03u" $sd` ; mv temp.bo.$bod borncs.$bo
40  done
41  printf " ..done\n"
42
43  # parse deviation vectors (eig)
44  printf "..parsing eigenvectors"
45  let sad=$EIG_NIONS+1
46  for nu in `seq 1 $EIG_NVIBS` ; do
47   nuu=`printf "%03u" $nu`
48   tail -n $sad eigens.vib.$nuu | head -n $EIG_NIONS | awk '{print $4,$5,$6}' > e.vib.$nuu.allions
49   split -a 3 -d -l 1 e.vib.$nuu.allions temp.e.vib.$nuu.ion.
50   for s in `seq 1 $EIG_NIONS` ; do
51    let sd=s-1; bo=`printf "%03u" $s`; bod=`printf "%03u" $sd`; mv temp.e.vib.$nuu.ion.$bod e.vib.$nuu.ion.$bo
52   done
53  done
54  printf " ..done\n"
55
56  # parse born effective charge matrices (born)
57  printf "..parsing eff.charges"
58  for s in `seq 1 $EIG_NIONS` ; do
59   ss=`printf "%03u" $s`
60   awk '{print $2,$3,$4}' borncs.$ss | tail -3 > bornch.$ss
61  done
62  mkdir temps02 ; mv eigens.* borncs.* temps02/
63  printf " ..done\n"
```

```bash
64
65   # parse matrices, multiply them and collect squares (giving intensities)
66   printf "..multiplying matrices, summing "
67   for nu in `seq 1 $EIG_NVIBS` ; do
68    nuu=`printf "%03u" $nu`
69    int=0.0
70    for alpha in 1 2 3 ;  do                # summing over alpha coordinates
71     sumpol=0.0
72     for s in `seq 1 $EIG_NIONS` ; do   # summing over atoms
73      ss=`printf "%03u" $s`
74      awk -v a="$alpha" '(NR==a){print}' bornch.$ss > z.ion.$ss.alpha.$alpha
75      # summing over beta coordinates and multiplying Z(s,alpha)*e(s) done by the following awk script
76      paste z.ion.$ss.alpha.$alpha  e.vib.$nuu.ion.$ss | \
77      awk '{pol=$1*$4+$2*$5+$3*$6; print $0," ",pol}' > matr-vib-${nuu}-alpha-${alpha}-ion-${ss}
78     done
79     sumpol=`cat matr-vib-${nuu}-alpha-${alpha}-ion-* | awk '{sum+=$7} END {print sum}'`
80     int=`echo "$int+($sumpol)^2" | sed 's/[eE]/*10^/g' |  bc -l`
81    done
82    freq=`awk '(NR==1){print $8}' temps02/eigens.vib.$nuu`
83    echo "$nuu $freq $int">> exact.res.txt
84    printf "."
85   done
86   printf " ..done\n"
87
88   # format results, normalize intensities
89   printf "..normalizing intensities"
90   max=`awk '(NR==1){max=$3} $3>=max {max=$3} END {print max}' exact.res.txt`
91   awk -v max="$max" '{printf "%03u %6.1f %5.3f\n",$1,$2,$3/max}' exact.res.txt > results.txt
92   printf " ..done\n"
93
94   # clean up, display results
95   printf "..finalizing:\n"
96   mkdir temps03; mv bornch.* e.vib.*.allions temps03/
97   mkdir temps04; mv z.ion* e.vib.*.ion.* temps04/
98   mkdir temps05; mv matr-* temps05/
99   mkdir results; mv *res*txt results/
100  let NMATRIX=$EIG_NVIBS**2
101  printf "%5u atoms found\n%5u vibrations found\n%5u matrices evaluated" \
102         $EIG_NIONS $EIG_NVIBS $NMATRIX > results/statistics.txt
103    # fast switch to clean up all temporary files
104    rm -r temps*
105  cat results/results.txt
```

```
..reading OUTCAR ..done
..splitting files ..done
..parsing eigenvectors ..done
..parsing eff.charges ..done
..multiplying matrices, summing ......... ..done
..normalizing intensities ..done
..finalizing:
001 3827.3 0.227
002 3713.0 0.006
```

```
003 1587.2 0.312
004  235.5 1.000
005   19.1 0.006
006    2.3 0.000
007   16.6 0.005
008   45.0 0.000
009  136.1 0.345
```

Note the results above include the rotational and vibrational modes (modes 4-9). The following shell script (also from http://homepage.univie.ac.at/david.karhanek/downlo removes those, and recalculates the intensities.

```bash
#!/bin/bash
# reformat intensities, just normal modes: 3N -> (3N-6)
printf "..reformatting and normalizing intensities"
cd intensities/results/
nlns=`wc -l exact.res.txt | awk '{print $1}' `; let bodylns=nlns-6
head -n $bodylns exact.res.txt > temp.reform.res.txt
max=`awk '(NR==1){max=$3} $3>=max {max=$3} END {print max}' temp.reform.res.txt`
awk -v max="$max" '{print $1,$2,$3/max}' temp.reform.res.txt > exact.reform.res.txt
awk -v max="$max" '{printf "%03u %6.1f %5.3f\n",$1,$2,$3/max}' temp.reform.res.txt > reform.res.txt
printf " ..done\n..normal modes:\n"
rm temp.reform.res.txt
cat reform.res.txt
cd ../..
```

```
..reformatting and normalizing intensities ..done
..normal modes:
001 3827.3 0.726
002 3713.0 0.019
003 1587.2 1.000
```

The interpretation of these results is that the mode at 3713 cm^{-1} would be nearly invisible in the IR spectrum. Earlier we interpreted that as the symmetric stretch. In this mode, there is only a small change in the molecule dipole moment, so there is a small IR intensity.

3.10.1 TODO Shell scripts are rough. One day I should rewrite this in python.

3.11 Bader analysis

[11]
   [22]

4 Bulk systems

See this paper (http://arxiv.org/pdf/1204.2733.pdf) for a very informative comparison of DFT codes for computing different bulk properties.

4.1 Defining and visualizing bulk systems

4.1.1 Built-in functions in ase

```python
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic


atoms = FaceCenteredCubic(directions=[[1,0,0],
                                      [0,1,0],
                                      [0,0,1]],
                                      size=(1,1,1),
                                      symbol='Ag',
                                      latticeconstant=4.0)

write('images/Ag-bulk.png',atoms,show_unit_cell=2)

# to make an alloy, we can replace one atom with another kind
atoms[0].symbol = 'Pd'
write('images/AgPd-bulk.png',atoms,show_unit_cell=2)
```
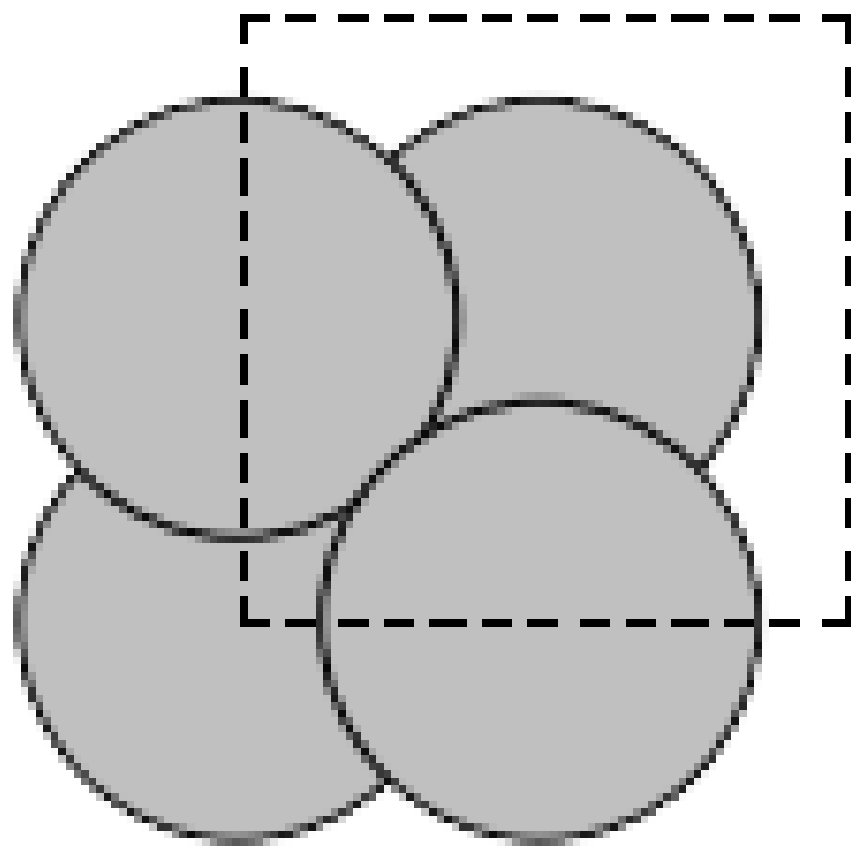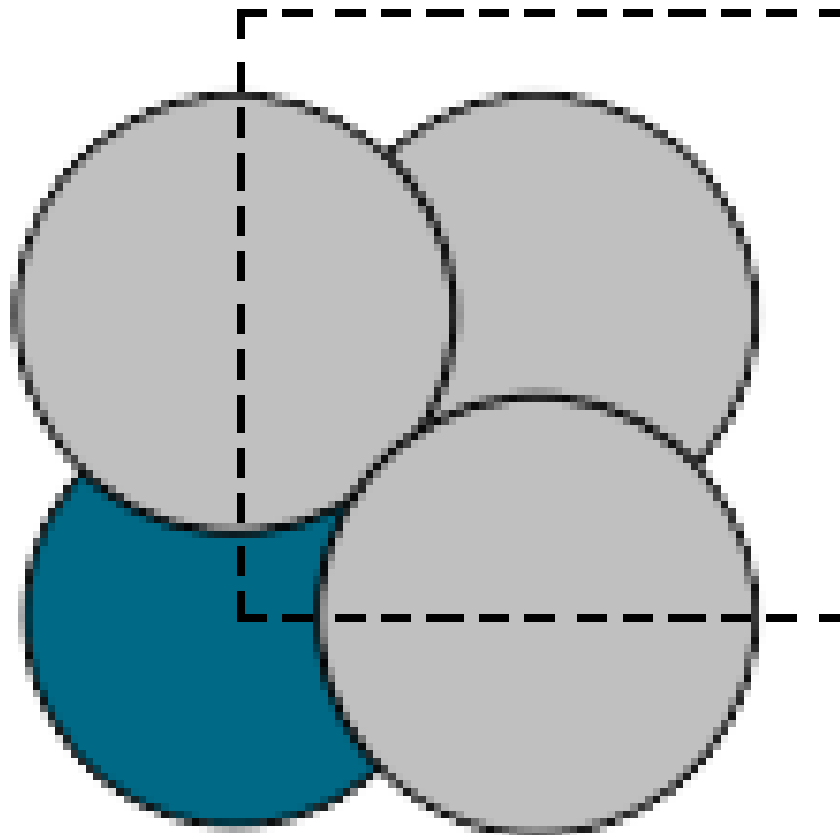
4.2 Determining bulk structures

4.2.1 fcc/bcc crystal structures

The fcc and bcc structures are simple. They only have one degree of freedom: the lattice constant. In this section we show how to calculate the equilibrium volume of each structure, and determine which one is more stable. We start with the fcc crystal structure of Cu. We will manually define the crystal structure based on the definitions in Kittel [14] (Chapter 1).

```
1  from jasp import *
2  from ase import Atom, Atoms
3
4  # fcc
5  LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
6  fcc_energies = []
7  for a in LC:
```

```
8        atoms = Atoms([Atom('Cu',(0,0,0))],
9                  cell=0.5*a*np.array([[1.0, 1.0, 0.0],
10                                       [0.0, 1.0, 1.0],
11                                       [1.0, 0.0, 1.0]]))
12
13      with jasp('bulk/Cu-{0}'.format(a),
14                  xc='PBE',
15                  encut=350,
16                  kpts=(8,8,8),
17                  atoms=atoms) as calc:
18          e = atoms.get_potential_energy()
19          fcc_energies.append(e)
20
21  import matplotlib.pyplot as plt
22  plt.plot(LC, fcc_energies)
23  plt.xlabel('Lattice constant ($\AA$)')
24  plt.ylabel('Total energy (eV)')
25  plt.savefig('images/Cu-fcc.png')
```
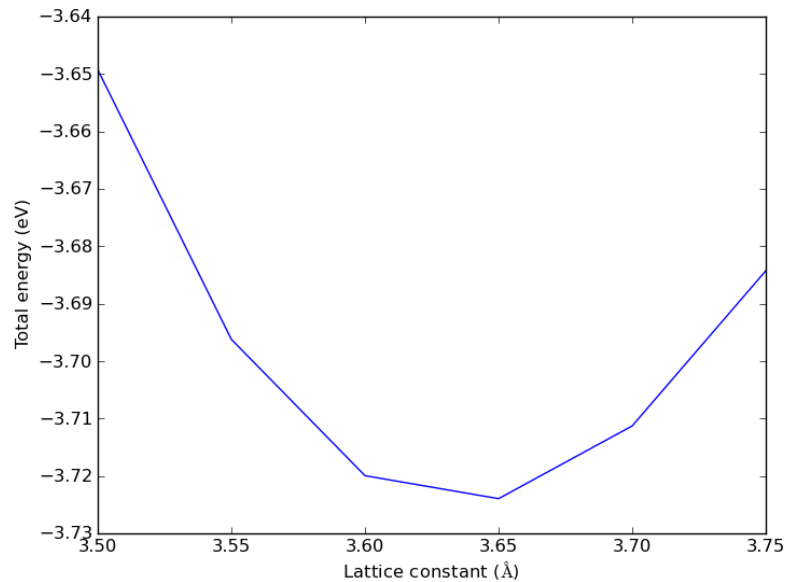

If you want to know the lattice constant that gives the lowest energy, you would fit an equation of state to the data. Here is an example:

```
1  from jasp import *
2  from ase.utils.eos import *
3  LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
4  energies = []
5  volumes = []
6  for a in LC:
```

```
 7      with jasp('bulk/Cu-{0}'.format(a)) as calc:
 8          atoms = calc.get_atoms()
 9          volumes.append(atoms.get_volume())
10          energies.append(atoms.get_potential_energy())
11
12  eos = EquationOfState(volumes,energies)
13  v0, e0, B = eos.fit()
14
15  print v0, e0, B
16
17  eos.plot('images/Cu-fcc-eos.png')
```


E: -3.725 eV, V: 12.017 Å^3, B: 137.627 GPa
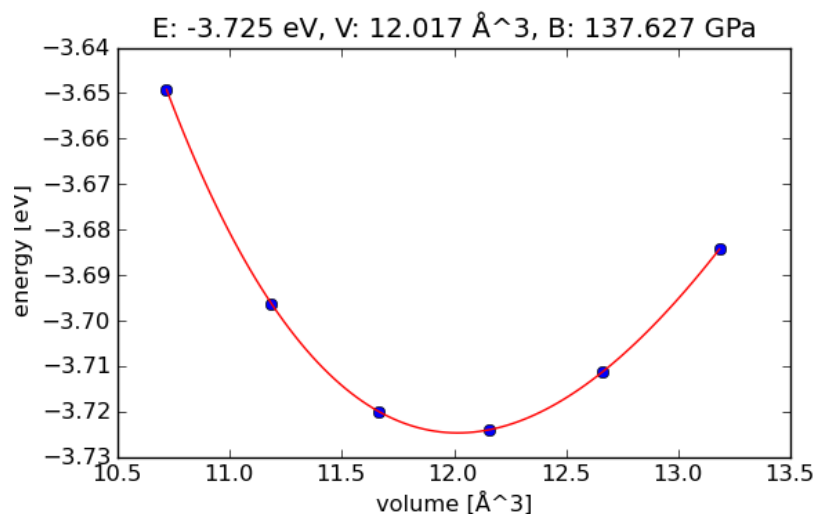
Before we jump into the bcc calculations, let's consider what range of lattice constants we should choose. The fcc lattice is close-packed, and the volume of the primitive cell is $V = 1/4a^3$ or about 11.8 $Å^3$/atom. The volume of the equilibrium bcc primitive cell will probably be similar to that. The question is: what bcc lattice constant gives that volume? The simplest way to answer this is to compute the answer using ase. We will make a bcc crystal at the fcc lattice constant, and then compute the scaling factor needed to make it the right volume.

```
1  from ase import Atom, Atoms
2
3  a = 3.61
4
5  atoms = Atoms([Atom('Cu', [0,0,0])],
6                cell=0.5*a*np.array([[1.0, 1.0, -1.0],
7                                     [-1.0, 1.0, 1.0],
8                                     [1.0, -1.0, 1.0]]))
```

58

```
 9
10    print a*(11.8/atoms.get_volume())**(1./3.)
```

Now we run the equation of state calculations.

```
 1    from jasp import *
 2    JASPRC['mode'] = 'run'
 3    from ase import Atom, Atoms
 4
 5    LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
 6
 7    for a in LC:
 8        atoms = Atoms([Atom('Cu', [0,0,0])],
 9                      cell=0.5*a*np.array([[1.0, 1.0, -1.0],
10                                           [-1.0, 1.0, 1.0],
11                                           [1.0, -1.0, 1.0]]))
12
13
14        with jasp('bulk/Cu-bcc-{0}'.format(a),
15                  xc='PBE',
16                  encut=350,
17                  kpts=(8,8,8),
18                  atoms=atoms) as calc:
19            e = atoms.get_potential_energy()
```

Finally, we will compare the two crystal structures.

```
 1    from jasp import *
 2
 3    bcc_LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
 4    bcc_volumes = []
 5    bcc_energies = []
 6    for a in bcc_LC:
 7        with jasp('bulk/Cu-bcc-{0}'.format(a)) as calc:
 8            atoms = calc.get_atoms()
 9            bcc_volumes.append(atoms.get_volume())
10            bcc_energies.append(atoms.get_potential_energy())
11
12    # fcc
13    fcc_LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
14    fcc_volumes = []
15    fcc_energies =[]
16    for a in fcc_LC:
17        with jasp('bulk/Cu-{0}'.format(a)) as calc:
18            atoms = calc.get_atoms()
19            fcc_volumes.append(atoms.get_volume())
20            fcc_energies.append(atoms.get_potential_energy())
21
22    import matplotlib.pyplot as plt
23    plt.plot(fcc_volumes, fcc_energies, label='fcc')
24    plt.plot(bcc_volumes, bcc_energies,label='bcc')
25
```

```
26   plt.xlabel('Atomic volume ($\AA^3$/atom)')
27   plt.ylabel('Total energy (eV)')
28   plt.legend()
29   plt.savefig('images/Cu-bcc-fcc.png')
30   plt.show()
```

Note we plot the energy vs. atomic volume. That is because the lattice constants of the two crystal structures are very different. It also shows that the atomic volumes in the two structures are similar.

What can we say here? The fcc structure has a lower energy than the bcc structure, so we can conclude the fcc structure is more favorable. In fact, the fcc structure is the experimentally found structure for Cu. Some caution is in order; if you run these calculations at a $4 \times 4 \times 4$ k-point grid, the bcc structure is more stable because the results are not converged!

What can we say about the relative stability of fcc to hcp? Nothing, until we calculate the hcp equation of state.

4.2.2   hcp

```
1    from ase.lattice.hexagonal import HexagonalClosedPacked
2    from jasp import *
3    import matplotlib.pyplot as plt
4
5    atoms = HexagonalClosedPacked(symbol='Ru',
6                                  latticeconstant={'a':2.7, 'c/a':1.584})
7
8    a_list = [2.5, 2.6, 2.7, 2.8, 2.9]
9    covera_list = [1.4, 1.5, 1.6, 1.7, 1.8]
10
11   for a in a_list:
12       energies = []
13       for covera in covera_list:
14
15           atoms = HexagonalClosedPacked(symbol='Ru',
16                               latticeconstant={'a':a, 'c/a':covera})
17
18           wd = 'bulk/Ru/{0:1.2f}-{1:1.2f}'.format(a,covera)
19
20           with jasp(wd,
21                     xc='PBE',
22                     kpts=(6,6,4),
23                     encut=350,
24                     atoms=atoms) as calc:
25               try:
26                   energies.append(atoms.get_potential_energy())
27               except (VaspSubmitted, VaspQueued):
28                   pass
29       plt.plot(covera_list, energies, label='a={0}'.format(a))
30
31   plt.xlabel('$c/a$ ($\AA$)')
```

```
32    plt.ylabel('Energy (eV)')
33    plt.legend()
34    plt.savefig('images/Ru-covera-scan.png')
```
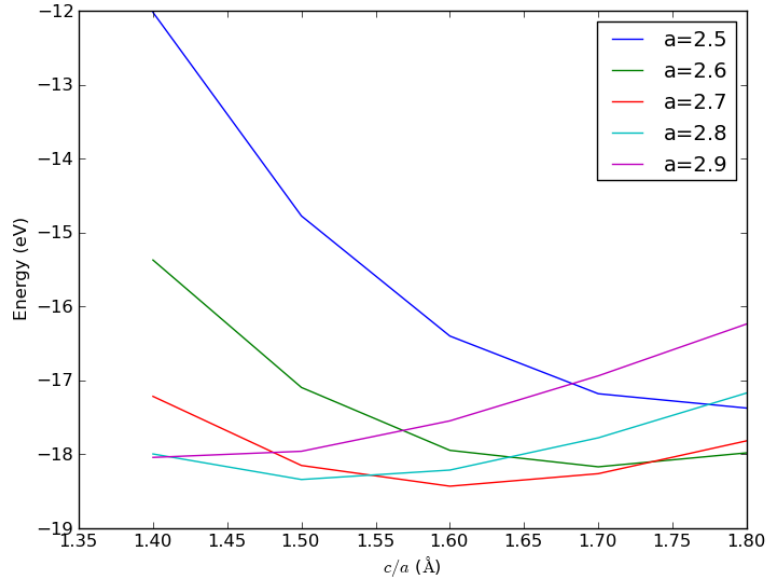

It looks like there is a minimum in the a=2.7 $\mathring{A}$ curve, at a c/a ratio of about 1.6. We can look at the same data in a contour plot which shows more clearly there is minimum in all directions near that point.

```
1    from jasp import *
2    import matplotlib.pyplot as plt
3    from pylab import *
4
5    x = [2.5, 2.6, 2.7, 2.8, 2.9]
6    y = [1.4, 1.5, 1.6, 1.7, 1.8]
7
8    X,Y = meshgrid(x,y)
9    Z = np.zeros(X.shape)
10
11   for i,a in enumerate(x):
12       for j,covera in enumerate(y):
13
14           wd = 'bulk/Ru/{0:1.2f}-{1:1.2f}'.format(a,covera)
15
16           with jasp(wd) as calc:
17               atoms = calc.get_atoms()
18               try:
19                   Z[i][j] = atoms.get_potential_energy()
20               except (VaspSubmitted, VaspQueued):
```

61

```
21                      pass
22
23   cf = plt.contourf(X,Y,Z,20,
24                     cmap=cm.jet)
25
26   cbar = colorbar(cf)
27   cbar.ax.set_ylabel('Energy (eV)')
28
29   plt.xlabel('$a$ ($\AA$)')
30   plt.ylabel('$c/a$')
31
32   plt.legend()
33   plt.savefig('images/ru-contourf.png')
34   plt.show()
```
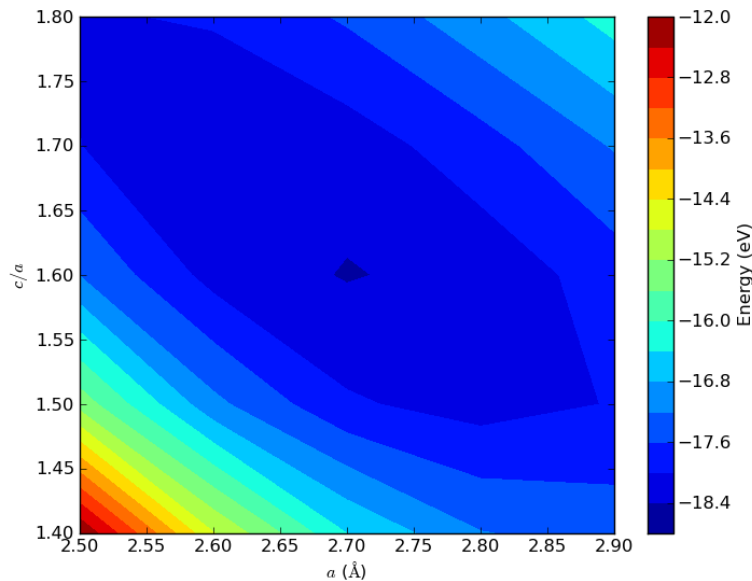

4.2.3   complex structures with internal degrees of freedom

A unit cell has six degrees of freedom: the lengths of each unit cell vector, and the angle between each vector. There may additionally be internal degrees of freedom for the atoms. It is impractical to try the approach used for Ru on anything complicated.

1. With the atoms frozen find the volume (at constant shape) that minimizes the total energy. The goal here is to just get an idea of where the right volume is.

2. Perform a volume equation of state calculation where each calculation is at constant volume, but the shape is allowed to change.

3. Finally, do a final calculation near the minimum energy allowing the volume to also change.

This multistep process is pretty reasonable to get a converged structure pretty quickly.

```python
from ase import *
import numpy as np
from jasp import *
import matplotlib.pyplot as plt

'''
create a TiO2 structure from the lattice vectors at
http://cst-www.nrl.navy.mil/lattice/struk/c4.html
'''
a = 4.59 # experimental degrees of freedom.
c = 2.96
u = 0.3 #internal degree of freedom!

#primitive vectors
a1 = a*np.array([1.0, 0.0, 0.0])
a2 = a*np.array([0.0, 1.0, 0.0])
a3 = c*np.array([0.0, 0.0, 1.0])

atoms = Atoms([Atom('Ti',[0., 0., 0.]),
               Atom('Ti',0.5*a1 + 0.5*a2 + 0.5*a3),
               Atom('O',u*a1+u*a2),
               Atom('O',-u*a1 -u*a2),
               Atom('O',(0.5+u)*a1 + (0.5-u)*a2 + 0.5*a3),
               Atom('O',(0.5-u)*a1 + (0.5+u)*a2 + 0.5*a3)],
              cell=[a1,a2,a3])

v0 = atoms.get_volume()
cell0 = atoms.get_cell()

factors = [0.9, 0.95, 1.0, 1.05, 1.1] #to change volume by

energies, volumes = [], []
for f in factors:
    v1 = f*v0
    cell_factor = (v1/v0)**(1./3.)

    atoms.set_cell(cell0*cell_factor, scale_atoms=True)

    with jasp('bulk/tio2/step1-{0:1.2f}'.format(f),
              encut=350,
              kpts=(4,4,4),
              xc='PBE',
              sigma=0.05,
              atoms=atoms) as calc:
```

```
45          try:
46              energies.append(atoms.get_potential_energy())
47              volumes.append(atoms.get_volume())
48          except (VaspSubmitted, VaspQueued):
49              pass
50
51  plt.plot(volumes, energies)
52  plt.xlabel('Vol. ($\AA^3)$')
53  plt.ylabel('Total energy (eV)')
54  plt.savefig('images/tio2-step1.png')
```
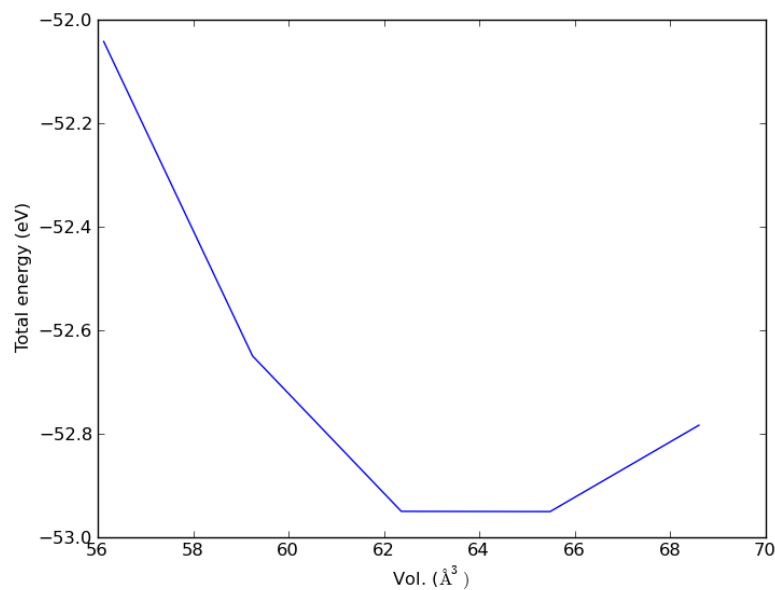

In step 2 we will actually do some relaxations. We will do a series of relaxations, from 62 to 66 $\AA^3$, and allow the ions and cell shape to relax (ISIF=4) at constant volume.

```
1   from jasp import *
2   import matplotlib.pyplot as plt
3
4   JASPRC['queue.walltime'] = '10:00'
5
6   with jasp('bulk/tio2/step1-1.00') as calc:
7       atoms = calc.get_atoms()
8       v0 = atoms.get_volume()
9       cell0 = atoms.get_cell()
10
11
12  volumes = [62., 63., 64., 65., 66.]
13
```

```
14    energies = []
15    for v in volumes:
16        atoms.set_cell((v/v0)**(1./3.)*cell0, scale_atoms=True)
17        with jasp('bulk/tio2/step2-{0}'.format(v),
18                  encut=350,
19                  kpts=(4,4,4),
20                  xc='PBE',
21                  sigma=0.05,
22                  isif=4,      # relax ions, cell shape. not volume
23                  ibrion=2,
24                  nsw=10,
25                  atoms=atoms) as calc:
26            try:
27                energies.append(atoms.get_potential_energy())
28            except (VaspSubmitted, VaspQueued):
29                pass
30
31    plt.plot(volumes, energies)
32    plt.xlabel('Vol. ($\AA^3)$')
33    plt.ylabel('Total energy (eV)')
34    plt.savefig('images/tio2-step2.png')
```
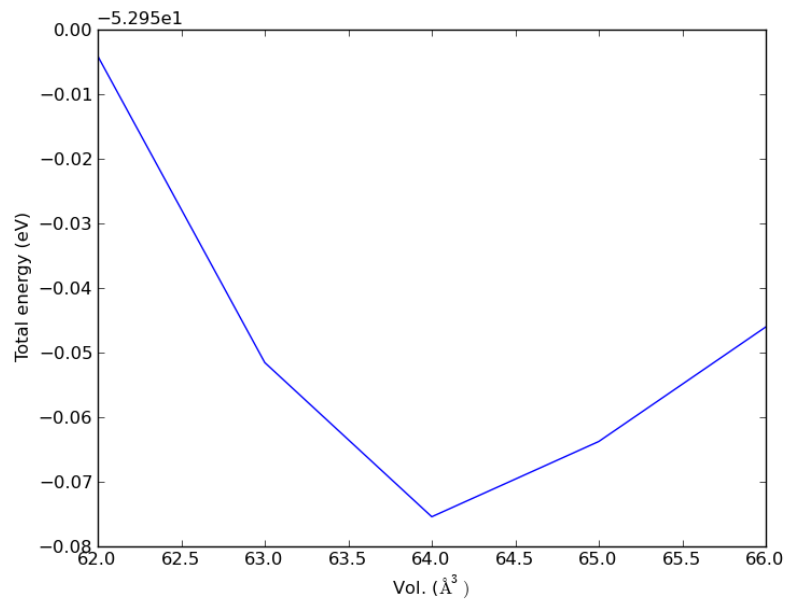

Now, we take the final step, with ISIF=3 using the best guess from step 2.

```
1    from jasp import *
2
3    with jasp('bulk/tio2/step2-64.0') as calc:
```

```
 4          atoms = calc.get_atoms()
 5
 6  with jasp('bulk/tio2/step3',
 7              encut=350,
 8              kpts=(4,4,4),
 9              xc='PBE',
10              sigma=0.05,
11              isif=3,      # relax ions, cell shape, and volume
12              ibrion=2,
13              nsw=10,
14              atoms=atoms) as calc:
15          try:
16              calc.calculate()
17              print calc
18          except (VaspSubmitted, VaspQueued):
19              pass
20
21  from pyspglib import spglib
22  print '\nThe spacegroup is {0}'.format(spglib.get_spacegroup(atoms))
```

The points to note here are the following:

1. The stress on the unit cell is very low.

2. The forces on the atoms are very small (less than 0.0 eV$/\mathring{A}$).

3. The agreement with experimental lattice parameters is pretty good.

4. The spacegroup is still correct.

4.3 Bulk calculation parameters

4.3.1 Effect of SIGMA

In the self-consistent cycle of a DFT calculation, the total energy is minimized with respect to occupation of the Kohn-Sham orbitals. At absolute zero, a band is either occupied or empty. This discrete occupation results in discontinuous changes in energy with changes in occupation, which makes it difficult to converge. One solution is to artificially broaden the band occupancies, as if they were occupied at a higher temperature where partial occupation is possible. This results in a continuous dependence of energy on the partial occupancy, and dramatically increases the rate of convergence. ISMEAR, SIGMA affect how the partial occupancies of the bands are determined.

Some rules to keep in mind:

1. The smearing methods were designed for metals. For molecules, semiconductors and insulators you should use a very small SIGMA (e.g. 0.01).

2. Standard values for metallic systems is SIGMA=0.1, but the best SIGMA may be material specific.

```python
from jasp import *
from ase import Atom, Atoms
import matplotlib.pyplot as plt

a = 3.61
atoms = Atoms([Atom('Cu',(0,0,0))],
              cell=0.5*a*np.array([[1.0, 1.0, 0.0],
                                   [0.0, 1.0, 1.0],
                                   [1.0, 0.0, 1.0]])).repeat((2,2,2))

SIGMA = [0.001, 0.05, 0.1, 0.2, 0.5]

for sigma in SIGMA:

    with jasp('bulk/Cu-sigma-{0}'.format(sigma),
              xc='PBE',
              encut=350,
              kpts=(4,4,4),
              ismear=-1,
              sigma=sigma,
              nbands=9*8,
              atoms=atoms) as calc:
        e = atoms.get_potential_energy()

        nbands = calc.nbands
        nkpts = len(calc.get_ibz_k_points())

        occ = np.zeros((nkpts, nbands))
        for i in range(nkpts):
            occ[i,:] = calc.get_occupation_numbers(kpt=i)

        max_occ = np.max(occ,axis=0)  #axis 0 is columns

        plt.plot(range(nbands), max_occ, label='$\sigma = {0}$'.format(sigma))

plt.xlabel('band number')
plt.ylabel('maximum occupancy (electrons)')
plt.ylim([-0.1, 2.1])
plt.legend(loc='best')
plt.savefig('images/occ-sigma.png')
plt.show()
```

4.4 Cohesive energy

The cohesive energy is defined as the energy to separate neutral atoms in their ground electronic state from the solid at 0K at 1 atm. We will compute this rhodium. Rh is normally an fcc metal, so we will use that and let Vasp find the equilibrium volume for us.

```
1    from jasp import *
2    from ase.lattice.cubic import FaceCenteredCubic
3    from ase import Atoms, Atom
4    # bulk system
5    atoms = FaceCenteredCubic(directions=[[0,1,1],
6                                          [1,0,1],
7                                          [1,1,0]],
8                                          size=(1,1,1),
9                                          symbol='Rh')
10
11   with jasp('bulk/bulk-rh',
12             xc='PBE',
13             encut=350,
14             kpts=(4,4,4),
15             isif=3,
16             ibrion=2,
17             nsw=10,
18             atoms=atoms) as calc:
19      bulk_energy = atoms.get_potential_energy()
20
21   # atomic system
22   atoms = Atoms([Atom('Rh',[5,5,5])],
23                 cell=(7,8,9))
24
25   with jasp('bulk/atomic-rh',
26             xc='PBE',
27             encut=350,
28             kpts=(1,1,1),
29             atoms=atoms) as calc:
30      atomic_energy = atoms.get_potential_energy()
31
32   cohesive_energy = atomic_energy - bulk_energy
33   print 'The cohesive energy is {0:1.3f} eV'.format(cohesive_energy)
```

According to Kittel, the cohesive energy of Rh is 5.75 eV. There are a few reasons we may have discrepancy here:

1. The k-point grid used in the bulk state is not very dense. However, you can see below that the total energy is pretty converged by a $6 \times 6 \times 6$ k-point grid.

```
1    from jasp import *
2
3    with jasp('bulk/bulk-rh') as calc:
4        atoms = calc.get_atoms()
5
6
7    kpts = [3, 4, 6, 9, 12, 15, 18]
8
9    for k in kpts:
10       with jasp('bulk/bulk-rh-kpts-{0}'.format(k),
11             xc='PBE',
```

```
12          encut=350,
13          kpts=(k,k,k),
14          atoms=atoms) as calc:
15        e = atoms.get_potential_energy()
16
17    print '({0:2d}, {0:2d}, {0:2d}): total energy = {1} eV'.format(k,e)
```

1. We neglected spin on the atomic state. Rh in the atomic state has this electronic structure: [Kr] 4d8 5s1 and is a doublet. We will examine this next.

```
1    from jasp import *
2    JASPRC['mode'] = 'run'
3    from ase.lattice.cubic import FaceCenteredCubic
4    from ase import Atoms, Atom
5    # bulk system
6    atoms = FaceCenteredCubic(directions=[[0,1,1],
7                                           [1,0,1],
8                                           [1,1,0]],
9                                           size=(1,1,1),
10                                          symbol='Rh')
11
12   with jasp('bulk/bulk-rh',
13           xc='PBE',
14           encut=350,
15           kpts=(4,4,4),
16           isif=3,
17           ibrion=2,
18           nsw=10,
19           atoms=atoms) as calc:
20       bulk_energy = atoms.get_potential_energy()
21
22   # atomic system
23   atoms = Atoms([Atom('Rh',[5,5,5],magmom=1)],
24                 cell=(7,8,9))
25
26   with jasp('bulk/atomic-rh-sp',
27           xc='PBE',
28           encut=350,
29           kpts=(1,1,1),
30           ispin=2,
31           atoms=atoms) as calc:
32       atomic_energy = atoms.get_potential_energy()
33
34   cohesive_energy = atomic_energy - bulk_energy
35   print 'The cohesive energy is {0:1.3f} eV'.format(cohesive_energy)
```

Again, the value in Kittel [14] is 5.75 eV which is very close to this value.

4.5 Effect of pressure on phase stability

So far we have only considered relative stability at a pressure of 0 Pa. We now consider the relative stability of two phases under pressure. We will consider TiO2 in the rutile and anatase phases.

The pressure is defined by: $P = -\frac{\partial E}{\partial V}T$

so if we have an equation of state $E(V)$ we can calculate the pressure at any volume, or alternatively, given a pressure, compute the volume. Pressure can affect the energy of two phases differently, so that one may become stable under pressure. The condition where a phase transition occurs is when the pressure in the two phases is the same, which occurs at a common tangent.

Here is an example of that.

http://prb.aps.org/abstract/PRB/v47/i11/p6299_1

ZnS http://cst-www.nrl.navy.mil/lattice/struk/b3.html http://cst-www.nrl.navy.mil/lattice/struk/

http://en.wikipedia.org/wiki/Birch%E2%80%93Murnaghan_equation_of_state

```
1   from jasp import *
2   from ase import Atom, Atoms
3
4   B='Ti'; X='O'; a=4.59; c=2.958; u=0.305;
5   '''
6   create a rutile structure from the lattice vectors at
7   http://cst-www.nrl.navy.mil/lattice/struk/c4.html
8
9   spacegroup: 136 P4_2/mmm
10  '''
11  a1 = a*np.array([1.0, 0.0, 0.0])
12  a2 = a*np.array([0.0, 1.0, 0.0])
13  a3 = c*np.array([0.0, 0.0, 1.0])
14
15  atoms = Atoms([Atom(B, [0., 0., 0.]),
16                Atom(B, 0.5*a1 + 0.5*a2 + 0.5*a3),
17                Atom(X,  u*a1 + u*a2),
18                Atom(X, -u*a1 - u*a2),
19                Atom(X, (0.5+u)*a1 + (0.5-u)*a2 + 0.5*a3),
20                Atom(X, (0.5-u)*a1 + (0.5+u)*a2 + 0.5*a3)],
21                cell=[a1, a2, a3])
22
23  nTiO2 = len(atoms)/3.
24  v0 = atoms.get_volume()
25  cell0 = atoms.get_cell()
26
27  volumes = [28., 30., 32., 34., 36.]  #vol of one TiO2
28
29  for v in volumes:
30      atoms.set_cell(cell0*((nTiO2*v/v0)**(1./3.)), scale_atoms=True)
31
32      with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v),
33                encut=350,
34                kpts=(6,6,6),
```

```
35                    xc='PBE',
36                    ismear=0,
37                    sigma=0.001,
38                    isif=2,
39                    ibrion=2,
40                    nsw=20,
41                    atoms=atoms) as calc:
42            try:
43                print atoms.get_potential_energy()
44            except (VaspSubmitted, VaspQueued):
45                pass
```

```
1    from jasp import *
2    from ase import Atom, Atoms
3    # http://cst-www.nrl.navy.mil/lattice/struk/c5.html
4
5    B='Ti'; X='O'; a=3.7842; c=2*4.7573; z=0.0831;
6
7    a1 = a*np.array([1.0, 0.0, 0.0])
8    a2 = a*np.array([0.0, 1.0, 0.0])
9    a3 = np.array([0.5*a, 0.5*a, 0.5*c])
10
11   atoms = Atoms([Atom(B, -0.125*a1 + 0.625*a2 + 0.25*a3),
12                 Atom(B,  0.125*a1 + 0.375*a2 + 0.75*a3),
13                 Atom(X, -z*a1 + (0.25-z)*a2 + 2.*z*a3),
14                 Atom(X, -(0.25+z)*a1 + (0.5-z)*a2 + (0.5+2*z)*a3),
15                 Atom(X, z*a1 - (0.25 - z)*a2 + (1-2*z)*a3),
16                 Atom(X, (0.25 + z)*a1 + (0.5 + z)*a2 + (0.5-2*z)*a3)],
17                 cell=[a1,a2,a3])
18
19   nTiO2 = len(atoms)/3.
20   v0 = atoms.get_volume()
21   cell0 = atoms.get_cell()
22
23   volumes = [30., 33., 35., 37., 39.]   #vol of one TiO2
24
25   for v in volumes:
26       atoms.set_cell(cell0*((nTiO2*v/v0)**(1./3.)), scale_atoms=True)
27
28       with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v),
29                 encut=350,
30                 kpts=(6,6,6),
31                 xc='PBE',
32                 ismear=0,
33                 sigma=0.001,
34                 isif=2,
35                 ibrion=2,
36                 nsw=20,
37                 atoms=atoms) as calc:
38           try:
39               print atoms.get_potential_energy()
40           except (VaspSubmitted, VaspQueued):
41               pass
```

```
1    from jasp import *
2    import matplotlib.pyplot as plt
3
4    # anatase equation of stte
5    volumes = [30., 33., 35., 37., 39.]  #vol of one TiO2
6    a_volumes, a_energies = [], []
7    for v in volumes:
8        with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v)) as calc:
9            atoms = calc.get_atoms()
10           nTiO2 = len(atoms)/3.0
11           a_volumes.append(atoms.get_volume()/nTiO2)
12           a_energies.append(atoms.get_potential_energy()/nTiO2)
13
14
15
16   # rutile equation of state
17   volumes = [28., 30., 32., 34., 36.]  #vol of one TiO2
18   r_volumes, r_energies = [], []
19   for v in volumes:
20       with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v)) as calc:
21           atoms = calc.get_atoms()
22           nTiO2 = len(atoms)/3.0
23           r_volumes.append(atoms.get_volume()/nTiO2)
24           r_energies.append(atoms.get_potential_energy()/nTiO2)
25
26
27   # cubic polynomial fit to equation of state E(V) = pars*[V^3 V^2 V^1 V^0]
28   apars = np.polyfit(a_volumes, a_energies, 3)
29   rpars = np.polyfit(r_volumes, r_energies, 3)
30
31   print 'E_anatase(V) = {0}*V^3 + {1}*V^2 + {2}*V + {3}'.format(*apars)
32   print 'E_rutile(V) = {0}*V^3 + {1}*V^2 + {2}*V + {3}'.format(*rpars)
33   print 'anatase epars: {0!r}'.format(apars)
34   print 'rutile epars: {0!r}'.format(rpars)
35   # get pressure parameters P(V) = -dE/dV
36   dapars = -np.polyder(apars)
37   drpars = -np.polyder(rpars)
38
39   print 'anatase ppars: {0!r}'.format(dapars)
40   print 'rutile ppars: {0!r}'.format(drpars)
41
42   print
43   print 'P_anatase(V) = {0}*V^2 + {0}*V + {2}'.format(*dapars)
44   print 'P_rutile(V) =  {0}*V^2 + {0}*V + {2}'.format(*drpars)
45
46   vfit = np.linspace(28,40)
47
48   # plot the equations of state
49   plt.plot(a_volumes, a_energies,'bo ', label='Anatase')
50   plt.plot(vfit, np.polyval(apars, vfit), 'b-')
51
52   plt.plot(r_volumes, r_energies,'gs ', label='Rutile')
53   plt.plot(vfit, np.polyval(rpars, vfit), 'g-')
54
55   plt.xlabel('Volume ($\AA^3$/f.u.)')
56   plt.ylabel('Total energy (eV/f.u.)')
```
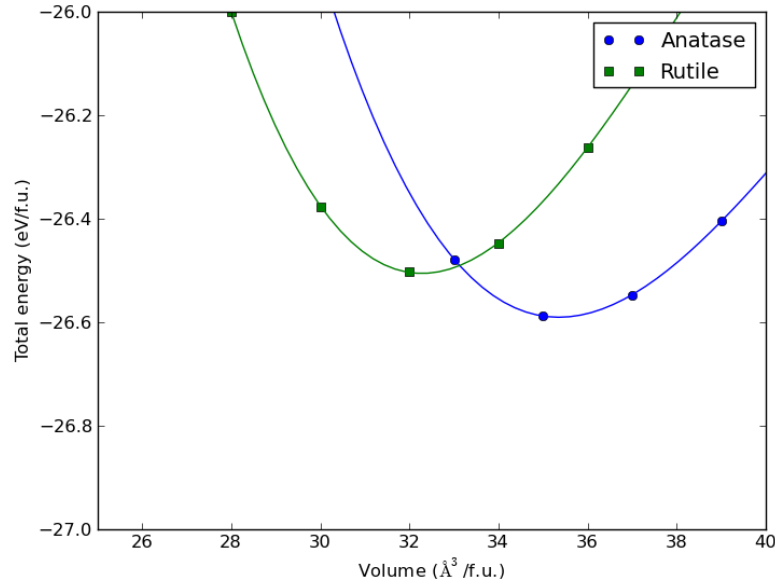
```
57    plt.legend()
58    plt.xlim([25,40])
59    plt.ylim([-27, -26])
60    plt.savefig('images/rutile-anatase-eos.png')
```


To find the conditions where a phase transition occurs, we have to find the common tangent line between the rutile and anatase phases. In other words we have to solve these two equations:

$(E_{anatase}(V1) - E_{rutile}(V2))/(V1-V2) = P_{anatase}(V1)$ $(E_{anatase}(V1) - E_{rutile}(V2))/(V1-V2) = P_{rutile}(V2)$

This is a nonlinear algebra problem.

```
1    from ase.units import GPa
2    from numpy import array, linspace, polyval
3
4    anatase_epars = array([ -1.06049246e-03,    1.30279404e-01,   -5.23520055e+00,
5             4.25202869e+01])
6    rutile_epars = array([ -1.24680208e-03,    1.42966536e-01,   -5.33239733e+00,
7             3.85903670e+01])
8
9    anatase_ppars = array([  3.18147737e-03,   -2.60558808e-01,    5.23520055e+00])
10   rutile_ppars = array([  3.74040625e-03,   -2.85933071e-01,    5.33239733e+00])
11
12   def func(V):
13       V1 = V[0] # rutile volume
14       V2 = V[1] # anatase volume
15
```

```
16        E_rutile = polyval(rutile_epars,V1)
17        E_anatase = polyval(anatase_epars,V2)
18
19        P_rutile =  polyval(rutile_ppars,V1)
20        P_anatase = polyval(anatase_ppars,V2)
21
22        return [(E_anatase - E_rutile)/(V1-V2) - P_anatase,
23                (E_anatase - E_rutile)/(V1-V2) - P_rutile]
24
25  from scipy.optimize import fsolve
26  x0 = fsolve(func,[28,34])
27  print x0
28  print polyval(anatase_ppars,x0[1])/GPa
29  print polyval(rutile_ppars,x0[0])/GPa
30
31  # illustrate the common tangent
32  import matplotlib.pyplot as plt
33
34  vfit = linspace(28,40)
35  plt.plot(vfit, polyval(anatase_epars,vfit),label='anatase')
36  plt.plot(vfit, polyval(rutile_epars,vfit),label='rutile')
37  plt.plot(x0, [polyval(rutile_epars,x0[0]),
38               polyval(anatase_epars,x0[1])], 'ko-', label='common tangent')
39  plt.legend()
40  plt.xlabel('Volume ($\AA^3$/f.u.)')
41  plt.ylabel('Total energy (eV/f.u.)')
42  plt.savefig('images/eos-common-tangent.png')
```

At a pressure of 4.5 GPa, we expect that anatase will start converting into rutile. Along this common tangent, a mixture of the two phases will be more stable than either pure phase.
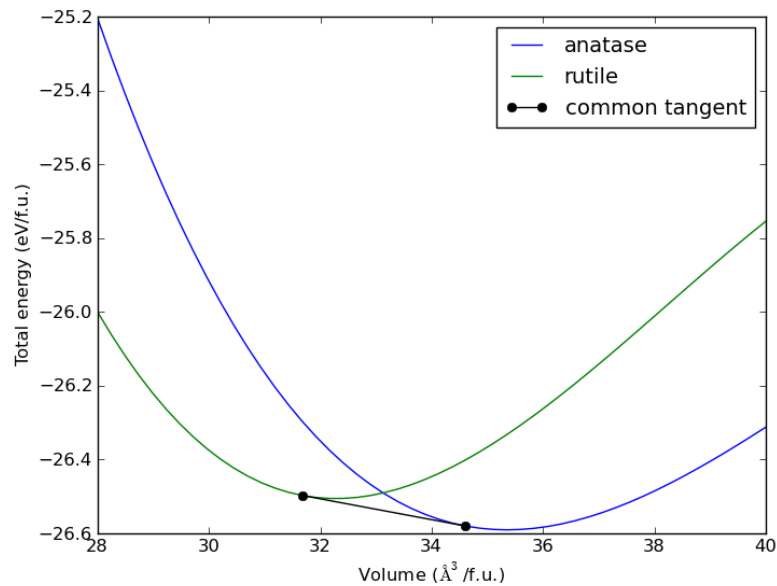
4.6 Bulk reaction energies

4.6.1 Alloy formation energies

In this section we will consider how to calculate the formation energy of an fcc Cu-Pdalloy and how to use that information to discuss relative stabilities.

Basic alloy formation energy

```python
from jasp import *
JASPRC['mode']='run'
from ase import Atom, Atoms

atoms = Atoms([Atom('Cu', [0.000,     0.000,     0.000])],
              cell= [[ 1.818,  0.000,  1.818],
                     [ 1.818,  1.818,  0.000],
                     [ 0.000,  1.818,  1.818]])

with jasp('bulk/alloy/cu',
          xc='PBE',
          encut=350,
          kpts=(13,13,13),
          nbands=9,
          ibrion=2,
          isif=4,
          nsw=10,
          atoms=atoms) as calc:
```

```
19      cu = atoms.get_potential_energy()
20
21
22  atoms = Atoms([Atom('Pd', [0.000,      0.000,      0.000])],
23              cell=[[ 1.978,  0.000,  1.978],
24                    [ 1.978,  1.978,  0.000],
25                    [0.000,  1.978,  1.978]])
26
27
28  with jasp('bulk/alloy/pd',
29          xc='PBE',
30          encut=350,
31          kpts=(13,13,13),
32          nbands=9,
33          ibrion=2,
34          isif=4,
35          nsw=10,
36          atoms=atoms) as calc:
37      pd = atoms.get_potential_energy()
38
39  print cu, pd
```

Note that the Pd energy is more negative than the Cu energy. This does not mean anything significant. We cannot say Pd is more stable than Cu; it is not like Cu could transmutate into Pd!

```
1   from jasp import *
2   JASPRC['mode']='run'
3   from ase import Atom, Atoms
4
5   atoms = Atoms([Atom('Cu', [0.000,      0.000,      0.000]),
6                 Atom('Pd', [-1.652,      0.000,      2.039])],
7              cell= [[ 0.000, -2.039,  2.039],
8                     [ 0.000,  2.039,  2.039],
9                     [ -3.303,  0.000,  0.000]])
10
11  with jasp('bulk/alloy/cupd-1',
12          xc='PBE',
13          encut=350,
14          kpts=(12,12,8),
15          nbands=17,
16          ibrion=2,
17          isif=4,
18          nsw=10,
19          atoms=atoms) as calc:
20      cupd1 = atoms.get_potential_energy()
21
22
23  atoms = Atoms([Atom('Cu', [-0.049,      0.049,      0.049]),
24                 Atom('Cu', [-11.170,     11.170,     11.170]),
25                 Atom('Pd', [-7.415,      7.415,      7.415]),
26                 Atom('Pd', [-3.804 ,     3.804,      3.804])],
27              cell=[[-5.629,  3.701,  5.629 ],
28                    [-3.701,  5.629,  5.629 ],
```

```
29                        [-5.629,  5.629,  3.701 ]])
30
31  with jasp('bulk/alloy/cupd-2',
32           xc='PBE',
33           encut=350,
34           kpts=(8,8,8),
35           nbands=34,
36           ibrion=2,
37           isif=4,
38           nsw=10,
39           atoms=atoms) as calc:
40      cupd2 = atoms.get_potential_energy()
41
42  print cupd1, cupd2
```

Looking at these energies, you could be tempted to say cupd-2 is more stable than cupd-1 because its energy is much lower. This is wrong, however, because cupd-2 has twice as many atoms as cupd-1. We should compare the normalized total energies.

```
1   from jasp import *
2
3   with jasp('bulk/alloy/cupd-1') as calc:
4       atoms = calc.get_atoms()
5       e1 = atoms.get_potential_energy()/len(atoms)
6
7   with jasp('bulk/alloy/cupd-2') as calc:
8       atoms = calc.get_atoms()
9       e2 = atoms.get_potential_energy()/len(atoms)
10
11  print 'cupd-1: {0} eV/atom'.format(e1)
12  print 'cupd-2: {0} eV/atom'.format(e2)
```

After normalizing by number of atoms, we can see that cupd-1 is a more stable structure. However, we are looking at total energies, and we might ask: is cupd-1 more stable than a mixture of the parent compounds, fcc Cu and Pd?

```
1   from jasp import *
2
3   with jasp('bulk/alloy/cu') as calc:
4       atoms = calc.get_atoms()
5       cu = atoms.get_potential_energy()/len(atoms)
6
7   with jasp('bulk/alloy/pd') as calc:
8       atoms = calc.get_atoms()
9       pd = atoms.get_potential_energy()/len(atoms)
10
11
12  with jasp('bulk/alloy/cupd-1') as calc:
```

```
13      atoms = calc.get_atoms()
14      e1 = atoms.get_potential_energy()
15      for atom in atoms:
16          if atom.symbol == 'Cu':
17              e1 -= cu
18          else:
19              e1 -= pd
20      e1 /= len(atoms)
21
22  with jasp('bulk/alloy/cupd-2') as calc:
23      atoms = calc.get_atoms()
24      e2 = atoms.get_potential_energy()
25      for atom in atoms:
26          if atom.symbol == 'Cu':
27              e2 -= cu
28          else:
29              e2 -= pd
30      e2 /= len(atoms)
31
32  print 'Delta Hf cupd-1 = {0:1.2f} eV/atom'.format(e1)
33  print 'Delta Hf cupd-2 = {0:1.2f} eV/atom'.format(e2)
```

The answer is yes. Both structures are energetically more favorable than an equal composition mixture of the parent metals. The heat of formation for both structures is exothermic, but the cupd-1 structure is more stable than the cupd-2 structure. This is shown conceptually in Figure **??**.

./images/alloy-stability.svg

We will now examine another structure at another composition and its stability.

```
1   from jasp import *
2   JASPRC['mode']='queue'
3   from ase import Atom, Atoms
4
5   # parent metals
6   with jasp('bulk/alloy/cu') as calc:
7       atoms = calc.get_atoms()
8       cu = atoms.get_potential_energy()/len(atoms)
9
10  with jasp('bulk/alloy/pd') as calc:
11      atoms = calc.get_atoms()
12      pd = atoms.get_potential_energy()/len(atoms)
13
14  atoms = Atoms([Atom('Cu', [-3.672,     3.672,     3.672]),
15                 Atom('Cu', [0.000,      0.000,     0.000]),
16                 Atom('Cu', [-10.821,   10.821,    10.821]),
17                 Atom('Pd', [-7.246,     7.246,     7.246])],
18                 cell=[[-5.464,  3.565,  5.464],
19                       [-3.565,  5.464,  5.464],
20                       [-5.464,  5.464,  3.565]])
21
22  with jasp('bulk/alloy/cu3pd-1',
```

```
23                xc='PBE',
24                encut=350,
25                kpts=(8,8,8),
26                nbands=34,
27                ibrion=2,
28                isif=4,
29                nsw=10,
30                atoms=atoms) as calc:
31      e3 = atoms.get_potential_energy()
32      for atom in atoms:
33          if atom.symbol == 'Cu':
34              e3 -= cu
35          else:
36              e3 -= pd
37      e3 /= len(atoms)
38  print 'Delta Hf cu3pd-1 = {0:1.2f} eV/atom'.format(e3)
```

The formation energy is slightly exothermic, which means the structure is more stable than a mixture of the parent metals. However, let us consider whether the structure is stable with respect to phase separation into pure Cu and the cupd-1 structure. We define the following quantities:

$H_{f,Cu} = 0.0$ eV/atom, $x_0 = 0$ $H_{f,cupd-1} = $ -0.12 eV/atom, $x_3 = 0.5$
The composition weighted average at $x_Pd = 0.25$ is:
$H_f = H_{f,Cu} + \frac{x0-x}{x0-x3}(H_{f,cupd-1} - H_{f,Cu}$

```
1  x0 = 0.0; x3 = 0.5; x = 0.25;
2  Hf1 = 0.0; Hf3 = -0.12;
3
4  print Hf1 + (x0-x)/(x0-x3)*(Hf3 - Hf1)
```

We find the weighted composition formation energy of pure Cu and cupd-1 is more favorable than the formation energy of cu3pd-1. Therefore, we could expect that structure to phase separate into a mixture of pure Cu and cupd-1. Schematically what we are seeing is shown in Figure **??**.

```
1  inkscape images/alloy-phase-stability.svg
```

./images/alloy-phase-stability.svg
Finally, let us consider one more structure with the Cu3Pd stoichiometry.

```
1  from jasp import *
2  JASPRC['mode']='queue'
3  from ase import Atom, Atoms
4
5  # parent metals
```

```
6   with jasp('bulk/alloy/cu') as calc:
7       atoms = calc.get_atoms()
8       cu = atoms.get_potential_energy()/len(atoms)
9
10  with jasp('bulk/alloy/pd') as calc:
11      atoms = calc.get_atoms()
12      pd = atoms.get_potential_energy()/len(atoms)
13
14  atoms = Atoms([Atom('Cu',  [-1.867,     1.867,      0.000]),
15                 Atom('Cu',  [0.000,      0.000,      0.000]),
16                 Atom('Cu',  [0.000,      1.867,      1.867]),
17                 Atom('Pd',  [-1.867,     0.000,      1.86])],
18                 cell=[[-3.735,  0.000,  0.000],
19                       [0.000,   0.000,  3.735],
20                       [0.000,   3.735,  0.000]])
21
22  with jasp('bulk/alloy/cu3pd-2',
23            xc='PBE',
24            encut=350,
25            kpts=(8,8,8),
26            nbands=34,
27            ibrion=2,
28            isif=4,
29            nsw=10,
30            atoms=atoms) as calc:
31      e4 = atoms.get_potential_energy()
32      for atom in atoms:
33          if atom.symbol == 'Cu':
34              e4 -= cu
35          else:
36              e4 -= pd
37      e4 /= len(atoms)
38  print 'Delta Hf cu3pd-2 = {0:1.2f} eV/atom'.format(e4)
```

This looks promising: the formation energy is much more favorable than cu3pd-1, and it is below the composition weighted formation energy of -0.06 eV/atom. Consequently, we conclude that this structure will not phase separate into a mixture of Cu and CuPd. We cannot say, however, if there is a more stable phase, or if it might phase separate into two other phases. We also note here that we have ignored a few other contributions to alloy stability. We have only considered the electronic energy contributions to the formation energy. At temperatures above absolute zero there are additional contributions including configurational and vibrational entropy, which may stabilize some structures more than others. Finally, our analysis is limited to comparisons of the structures computed on the fcc lattice. In fact, it is known that the CuPd alloy forms a bcc structure. We did not calculate that structure, so we can not say if it is more or less stable than the obvious fcc structure we found.

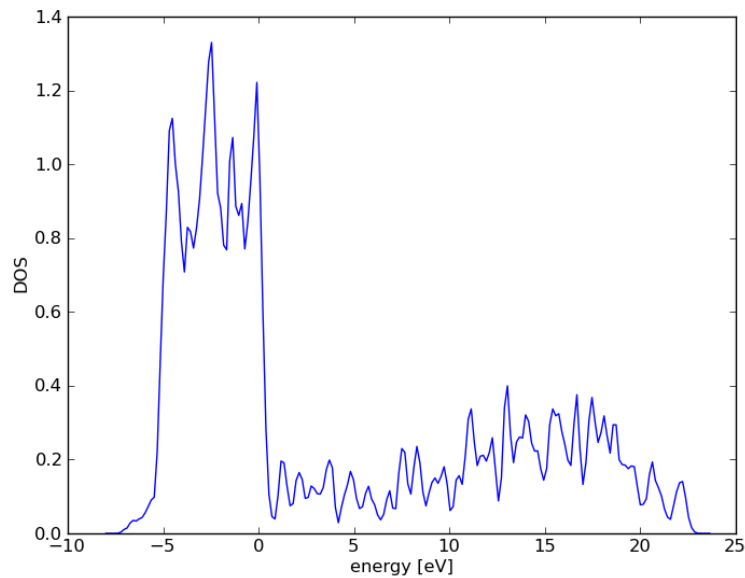./images/alloy-phase-stability-2.svg

80

The construction of alloy phase diagrams is difficult. You are always faced with the possibility that there is a phase that you have not calculated that is more stable than the ones you did calculate. One approach is to use a tool that automates the discovery of relevant structures such as the Alloy Theoretic Automated Toolkit (ATAT) [31, 32] which uses a cluster expansion methodology.

4.7 Bulk density of states

```python
from ase import Atoms, Atom
from jasp import *
import sys

from pylab import *
import numpy as np
from ase.dft import DOS

a = 3.9  # approximate lattice constant
b = a / 2.
bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
             cell=[(0, b, b),
                   (b, 0, b),
                   (b, b, 0)])

with jasp('bulk/pd-dos',
          encut=300,
          xc='PBE',
          lreal=False,
          kpts=(8, 8, 8),  # this is too low for high quality DOS
          atoms=bulk) as calc:

    # this runs the calculation
    bulk.get_potential_energy()
    dos = DOS(calc, width=0.2)
    d = dos.get_dos()
    e = dos.get_energies()

import pylab as plt
plt.plot(e,d)
plt.xlabel('energy [eV]')
plt.ylabel('DOS')
plt.savefig('images/pd-dos.png')
```

This DOS looks roughly like you would expect. The peak between -5 to 0 eV is the Pd d-band.

The vasp manual recommends a final run be made with ISMEAR=-5, which uses the tetrahedron method with Bloöhl corrections.

```
1  from jasp import *
2  from ase.dft import DOS
3  with jasp('bulk/pd-dos') as calc:
4      calc.clone('bulk/pd-dos-ismear-5')
5
6  with jasp('bulk/pd-dos-ismear-5') as calc:
7      bulk = calc.get_atoms()
8
9      calc.set(ismear=-5)
10
11     bulk.get_potential_energy()
12     dos = DOS(calc, width=0.2)
13     d = dos.get_dos()
14     e = dos.get_energies()
15
16  import pylab as plt
17  plt.plot(e, d)
18  plt.xlabel('energy [eV]')
19  plt.ylabel('DOS')
20  plt.savefig('images/pd-dos-ismear-5.png')
```
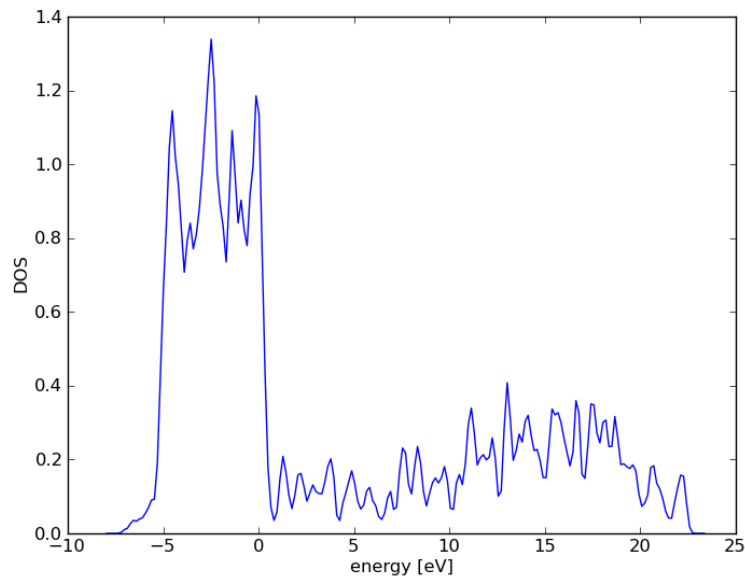
This is not that different from the regular ISMEAR, so I am not sure this is correct. Maybe there is another way to compute DOS when the tetrahedral method is used? This looks like the eigenvalue spectrum.

4.8   Atom projected density of states

In this example, we consider how to get the atom-projected density of states. We are interested in properties of the d-band on Pd, such as the d-band center and d-band width.

```python
from ase import Atoms, Atom
from jasp import *
from ase.calculators.vasp import VaspDos
import sys

from pylab import *
import numpy as np

a = 3.9  # approximate lattice constant
b = a / 2.
bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
             cell=[(0, b, b),
                   (b, 0, b),
                   (b, b, 0)])

with jasp('bulk/pd-ados',
          encut=300,
```

```python
18                  xc='PBE',
19                  lreal=False,
20                  rwigs=[1.5],   # wigner-seitz radii for ados
21                  kpts=(8, 8, 8),
22                  atoms=bulk) as calc:
23
24          # this runs the calculation
25          bulk.get_potential_energy()
26
27          # now get results
28          ados = VaspDos(efermi=calc.get_fermi_level())
29
30          energies = ados.energy
31          dos = ados.site_dos(0, 'd')
32
33          #we will select energies in the range of -10, 5
34          ind = (energies < 5) & (energies > -10)
35
36          energies = energies[ind]
37          dos = dos[ind]
38
39          Nstates = np.trapz(dos, energies)
40          occupied = energies <= 0.0
41          N_occupied_states = np.trapz(dos[occupied], energies[occupied])
42          ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
43          wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)
44
45          print 'Total # states = %1.2f' % Nstates
46          print 'number of occupied states = %1.2f' % N_occupied_states
47          print 'd-band center = %1.2f eV' % ed
48          print 'd-band width  = %1.2f eV' % np.sqrt(wd2)
49
50          # plot the d-band
51          plot(energies, dos, label='$d$-orbitals')
52
53          # plot the occupied states in shaded gray
54          fill_between(x=energies[occupied],
55                       y1=dos[occupied],
56                       y2=zeros(dos[occupied].shape),
57                       color='gray')
58
59          xlabel('$E - E_f$ (eV)')
60          ylabel('DOS (arbitrary units)')
61
62  savefig('images/pd-ados.png')
```
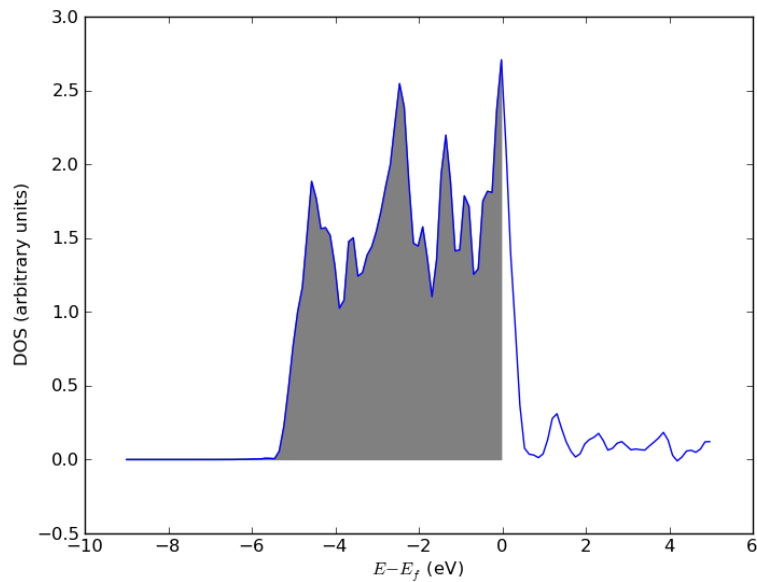
4.9 Band structures

To compute a band structure we do two things. First, we compute the self-consistent band structure. Then we compute the band structure at the desired k-points. We will use Si as an example (adapted from http://bbs.sciencenet.cn/bbs/upload/20083418325986.pdf

```
1  from jasp import *
2  JASPRC['queue.walltime'] = '10:00:00'
3
4  from ase import Atom, Atoms
5  from ase.visualize import view
6
7  a = 5.38936
8  atoms = Atoms([Atom('Si',[0,0,0]),
9                 Atom('Si',[0.25, 0.25, 0.25])])
10
11 atoms.set_cell([[a/2., a/2., 0.0],
12                 [0.0,  a/2., a/2.],
13                 [a/2., 0.0, a/2.]],scale_atoms=True)
14
15 with jasp('bulk/Si-selfconsistent',
16           xc='PBE',
17           prec='Medium',
18           istart=0,
19           icharg=2,
20           ediff=0.1e-03,
```

```
21            kpts=(4,4,4),
22            atoms=atoms) as calc:
23        print atoms.get_potential_energy()
```

Now, we run a new calculation along the k-point path desired. The standard VASP way of doing this is to modify the INCAR and KPOINTS file and rerun VASP. We will not do that. Doing that results in some lost information if you overwrite the old files. We will copy the old directory to a new directory, using code to ensure this only happens one time.

```
1   from jasp import *
2   JASPRC['queue.walltime'] = '10:00:00'
3
4   import os, shutil
5
6   wd = 'bulk/Si-bandstructure'
7
8   if not os.path.isdir(wd):
9       shutil.copytree('bulk/Si-selfconsistent', wd)
10
11  kpts = [[0.5,0.5,0.0],    # L
12         [0,0,0],          # Gamma
13         [0,0,0],
14         [0.5, 0.5, 0.5]]  # X
15
16  with jasp(wd) as calc:
17      calc.set(icharg=11) # update incar
18      calc.write_kpoints(mode='line',
19                         kpts=kpts,
20                         kpt_format='rec',
21                         intersections=10) # make new kpoint file
22      print calc.calculate()
23
24  # note you cannot run this script more than once because vasp.py does not know how to read in our new kpoint file ye
```

:func:ase.calculators.vasp.Vasp.read_kpoints does not support reading in this KPOINTS file yet. jasp does not have a fancy way of extracting band structures yet, so we will learn how to manually parse the EIGENVAL file here to generate the band structure.

```
1   f = open('bulk/Si-bandstructure/EIGENVAL', 'r')
2
3   line1 = f.readline()
4   line2 = f.readline()
5   line3 = f.readline()
6   line4 = f.readline()
7   comment = f.readline()
8   unknown, npoints, nbands = [int(x) for x in f.readline().split()]
9
```

```
10    blankline = f.readline()
11
12    band_energies = [[] for i in range(nbands)]
13
14    for i in range(npoints):
15        x,y,z, weight = [float(x) for x in f.readline().split()]
16
17        for j in range(nbands):
18            fields = f.readline().split()
19            id, energy = int(fields[0]), float(fields[1])
20            band_energies[id-1].append(energy)
21        blankline = f.readline()
22    f.close()
23
24    import matplotlib.pyplot as plt
25
26    for i in range(nbands):
27        plt.plot(range(npoints), band_energies[i])
28
29    ax = plt.gca()
30    ax.set_xticks([]) # no tick marks
31    plt.xlabel('k-vector')
32    ax.set_xticks([0,10,19])
33    ax.set_xticklabels(['$L$', '$\Gamma$', '$X$'])
34    plt.savefig('bulk/Si-bandstructure/bandstructure.png')
```
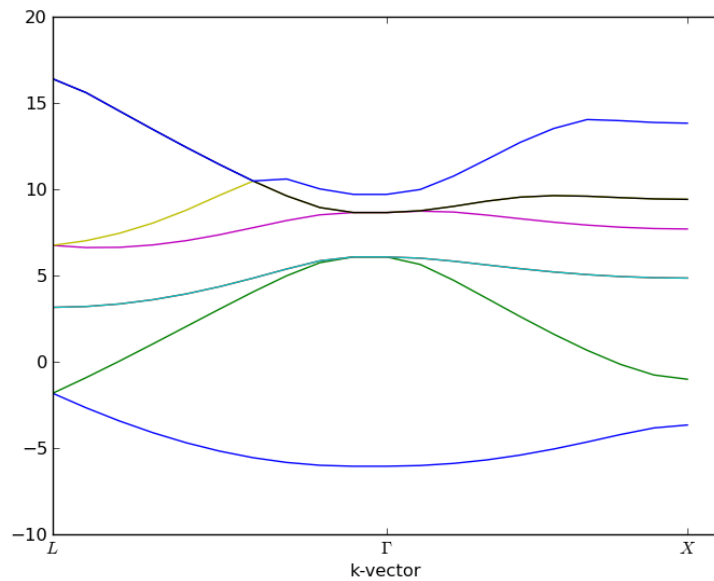
4.10 Simulated XRD spectra

You do not really need a DFT calculation for this, except to get the optimized structure. A few notes on using this module:

1. You need a large repeat unit cell.

2. It is is not fast. There is a double loop over the atoms, meaning N^2 operations.

3. This example is primarily a proof of concept. I have not thought about how good the results are.

```python
from ase import Atom,Atoms
from ase.xrdebye import *
import numpy as np

wavelength = 1.051967 # in angstroms

XRD = XrDebye(wavelength=wavelength,alpha=1.01,damping=0.0)

a = 3.92  # approximate lattice constant
b = a / 2.
atoms = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
              cell=[(0, b, b),
                    (b, 0, b),
                    (b, b, 0)]).repeat((5,5,5))

thetas = np.linspace(0.2, np.pi/4.)

I = [XRD.get(atoms,s=s) for s in 2*np.sin(thetas)/wavelength]

import matplotlib.pyplot as plt
plt.plot((2*thetas)*180./np.pi,I)
plt.xlabel('$2\Theta$')
plt.ylabel('Intensity (arb. units)')
plt.savefig('images/pd-xrd.png')
```
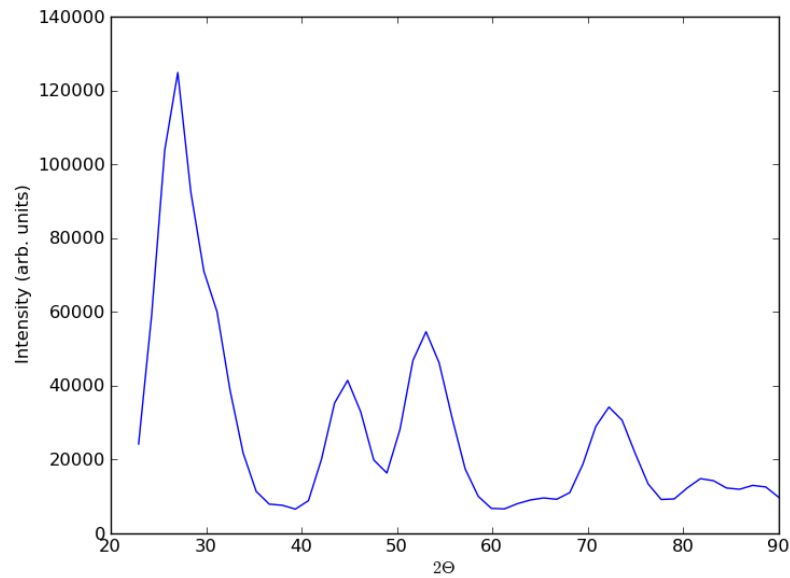
4.11 Magnetism

4.11.1 Determining if a magnetic solution is energetically favorable

We can force a total magnetic moment onto a unit cell and compute the total energy as function of the total magnetic moment. If there is a minimum in the energy, then we know there is a lower energy magnetic solution than a non-magnetic solution.
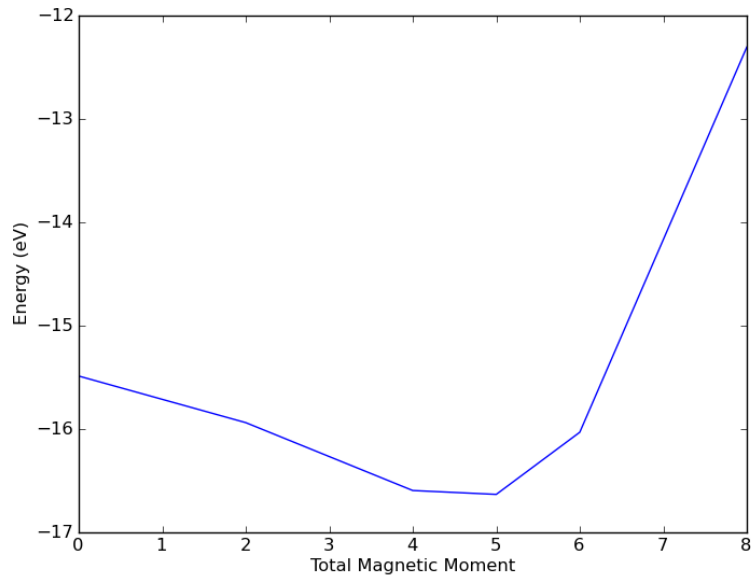
```
1   from jasp import *
2   JASPRC['queue.walltime'] = '1:00:00'
3   from ase.lattice.cubic import BodyCenteredCubic
4
5   atoms = BodyCenteredCubic(directions=[[1,0,0],
6                                         [0,1,0],
7                                         [0,0,1]],
8                                         size=(1,1,1),
9                                         symbol='Fe')
10
11  NUPDOWNS = [0.0, 2.0, 4.0, 5.0, 6.0, 8.0]
12  energies = []
13  for B in NUPDOWNS:
14      with jasp('bulk/Fe-bcc-fixedmagmom-{0:1.2f}'.format(B),
15          xc='PBE',
16          encut=300,
```

```
17              kpts=(4,4,4),
18              ispin=2,
19              nupdown=B,
20              atoms=atoms) as calc:
21          try:
22              e = atoms.get_potential_energy()
23              energies.append(e)
24          except (VaspSubmitted, VaspQueued):
25              pass
26
27  import matplotlib.pyplot as plt
28  plt.plot(NUPDOWNS, energies)
29  plt.xlabel('Total Magnetic Moment')
30  plt.ylabel('Energy (eV)')
31  plt.savefig('images/Fe-fixedmagmom.png')
```


You can see here there is a minimum in energy at a total magnetic moment somewhere between 4 and 5. There are two Fe atoms in the unit cell, which means the magnetic moment on each atom must be about 2.5 Bohr-Magnetons. This is a good guess for a real calculation. Note that Vasp recommends you overestimate the magnetic moment guesses if you are looking for ferromagnetic solutions.

```
1  from jasp import *
2  JASPRC['queue.walltime'] = '1:00:00'
3  from ase.lattice.cubic import BodyCenteredCubic
4
```

```
5
6   atoms = BodyCenteredCubic(directions=[[1,0,0],
7                                          [0,1,0],
8                                          [0,0,1]],
9                                          size=(1,1,1),
10                                         symbol='Fe')
11
12  for atom in atoms:
13      atom.magmom = 2.5
14
15  with jasp('bulk/Fe-bcc-sp-1',
16            xc='PBE',
17            encut=300,
18            kpts=(4,4,4),
19            ispin=2,
20            atoms=atoms) as calc:
21          try:
22              e = atoms.get_potential_energy()
23              print atoms.get_magnetic_moment()
24              print atoms.get_magnetic_moments()
25          except (VaspSubmitted, VaspQueued):
26              pass
```

4.11.2   TODO find out how to get atoms magnetic moments.

4.12   TODO phonons
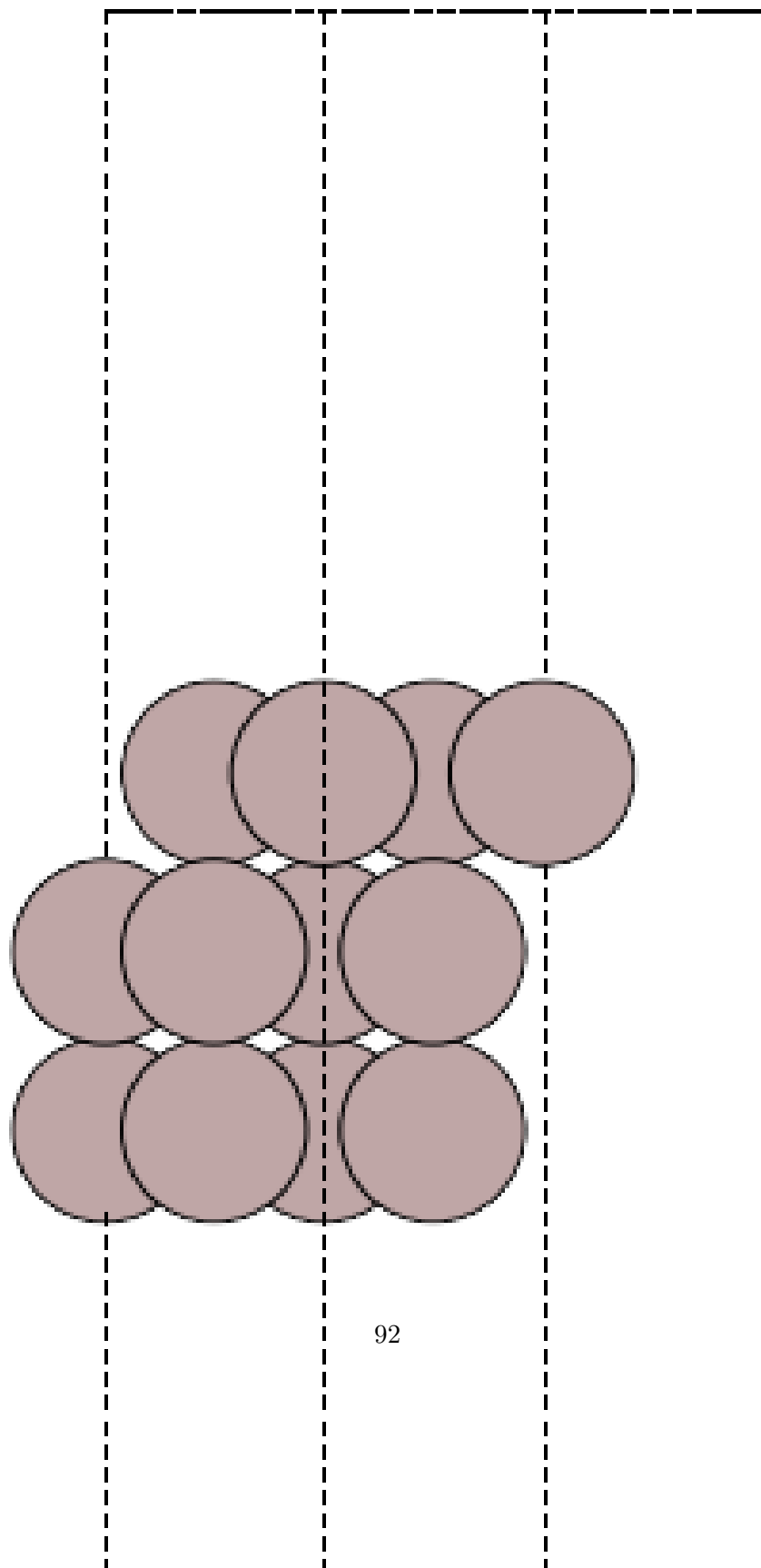
5   Surfaces

5.1   Surface structures

5.1.1   Simple surfaces

ase provides many utility functions to setup surfaces. Here is a simple example of an fcc111 Al surface. There are built in functions for fcc111, bcc110, bcc111, hcp001 and diamond111.

```
1   from ase.lattice.surface import *
2   from ase.io import write
3
4   slab = fcc111('Al', size=(2,2,3), vacuum=10.0)
5
6   write('images/Al-slab.png', slab, rotation='90x',show_unit_cell=2)
```

92

5.1.2 vicinal surfaces

Many stepped surfaces are not that difficult to make now. The main idea in generating them is described here. ase provides a general function for making vicinal surfaces. Here is an example.

```
1   from ase.lattice.surface import surface
2   from ase.io import write
3
4   # Au(211) with 9 layers
5   s1 = surface('Au', (2, 1, 1), 9)
6   s1.center(vacuum=10, axis=2)
7
8   write('images/Au-211.png',
9         s1.repeat((3,3,1)),
10        rotation='-30z,90x', # change the orientation for viewing
11        show_unit_cell=2)
```
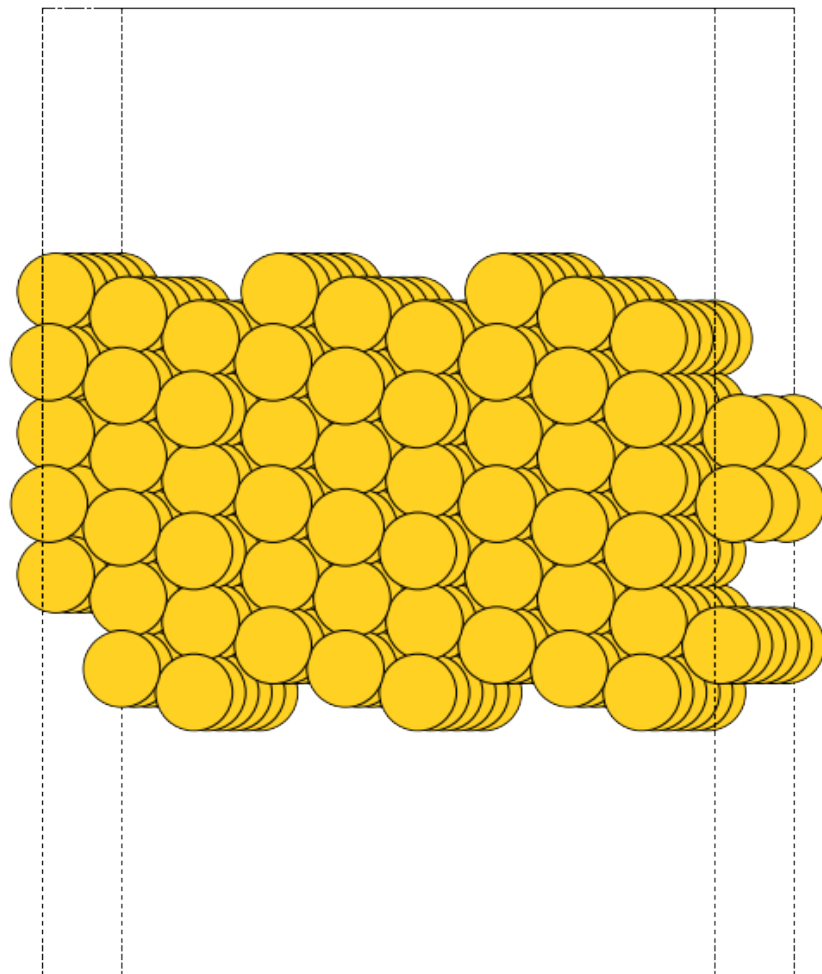
5.2 Surface relaxation

When a surface is created, the bulk symmetry is broken and consenquently there will be forces on the surface atoms. We will examine some consequences of this with a simple Al slab. First, we show there are forces on the slab atoms.

```
1  from jasp import *
2  from ase.lattice.surface import *
3
4  atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
5
```

```
6   with jasp('surfaces/Al-slab-unrelaxed',
7           xc='PBE',
8           kpts=(6,6,1),
9           encut=350,
10          atoms=atoms) as calc:
11      atoms.get_forces()
12      print calc
```

Some points to note. The forces on the atoms have symmetry to them.

```
1   from jasp import *
2   from ase.lattice.surface import *
3
4   atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
5
6   with jasp('surfaces/Al-slab-unrelaxed') as calc:
7       atoms = calc.get_atoms()
8
9       for i in range(1,len(atoms)):
10          print atoms[i].z - atoms[i-1].z
```

We have to make some decisions about how to relax the slab. One choice would be to relax all the atoms in the slab. If we do that, then there will be no atoms with bulk like spacing unless we increase the slab thickness pretty dramatically. It is pretty common to freeze some atoms at the bulk coordinates, and let the others relax. We will freeze the bottom two layers (defined by tags 3 and 4) and let the first two layers relax. To do that we add constraints to the slab.

Note: the ase constraints are only partially used by jasp/vasp. The FixAtoms constraint gets written to the POSCAR file, and is then used internally in Vasp. The only other constraint that Vasp can use internally is FixScaled. The other constraints are not written to the POSCAR and are not used by Vasp.

TODO: show how to use other ase.constraints with jasp.

```
1   from jasp import *
2   from ase.lattice.surface import *
3   from ase.constraints import FixAtoms
4
5   atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
6
7   with jasp('surfaces/Al-slab-relaxed',
8           xc='PBE',
9           kpts=(6,6,1),
10          encut=350,
11          ibrion=2,
12          isif=2,
```

```
13              nsw=10,
14              atoms=atoms) as calc:
15
16       constraint = FixAtoms(mask=[atom.tag >= 3 for atom in atoms])
17       atoms.set_constraint(constraint)
18
19       atoms.get_forces()
20       print calc
```

You can see that atoms 2 and 3 (the ones we relaxed) now have very
low forces on them and it appears that atoms 0 and 1 have no forces on
them. That is because the FixAtoms constraint works by setting the forces
on those atoms to zero. We can see in the next example that the z-positions
of the relaxed atoms have indeed relaxed and changed, while the position of
the frozen atoms did not change.

```
1    from jasp import *
2    from ase.lattice.surface import *
3
4    atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
5
6    with jasp('surfaces/Al-slab-relaxed') as calc:
7        atoms = calc.get_atoms()
8
9        for i in range(1,len(atoms)):
10           print 'd_({0},{1}) = {2}'.format(i,i-1,
11                                       atoms[i].z - atoms[i-1].z)
```

5.3   Work function

To get the work function, we need to have the local potential. this is not
written by default in vasp, and we have to tell it to do that with the LVTOT
keyword.

```
1    from jasp import *
2    from ase.lattice.surface import *
3    from ase.constraints import FixAtoms
4    from pylab import *
5    from jasp.volumetric_data import *
6
7    with jasp('surfaces/Al-slab-relaxed') as calc:
8        atoms = calc.get_atoms()
9
10   with jasp('surfaces/Al-slab-locpot',
11           xc='PBE',
12           kpts=(6,6,1),
13           encut=350,
14           lvtot=True, # write out local potential
15           lvhar=True, # write out only electrostatic potential, not xc pot
```

```
16              atoms=atoms) as calc:

17
18      calc.calculate()
19      ef = calc.get_fermi_level()

20
21      atoms = calc.get_atoms()

22
23      x,y,z,lp = calc.get_local_potential()

24
25  nx, ny, nz = lp.shape

26
27  axy = np.array([np.average(lp[:,:,z]) for z in range(nz)])
28  # setup the x-axis in realspace
29  uc = atoms.get_cell()
30  xaxis = np.linspace(0,uc[2][2],nz)

31
32  plot(xaxis, axy)
33  plot([min(xaxis), max(xaxis)], [ef, ef],'k:')
34  savefig('images/Al-wf.png')

35
36  ind = (xaxis > 0) & (xaxis < 5)
37  wf = np.average(axy[ind]) - ef
38  print ' The workfunction is {0:1.2f} eV'.format(wf)
```

The workfunction of Al is listed as 4.08 at http://hyperphysics.phy-astr.gsu.edu/hbase/tables/photoelec.html.
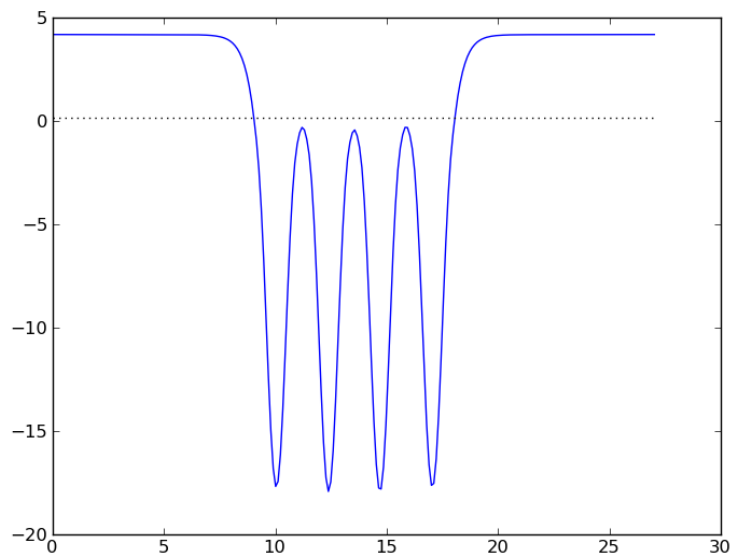
5.4 Surface energy

The easiest way to calculate surface energies is from this equation:

$$\sigma = \frac{1}{2}(E_{slab} - \frac{N_{slab}}{N_{bulk}} E_{bulk})$$

where E_{slab} is the total energy of a symmetric slab (i.e. one with inversion symmetry, and where both sides of the slab have been relaxed), E_{bulk} is the total energy of a bulk unit cell, N_{slab} is the number of atoms in the slab, and N_{bulk} is the number of atoms in the bulk unit cell. One should be sure that the bulk energy is fully converged with respect to k-points, and that the slab energy is also converged with respect to k-points. The energies should be compared at the same cutoff energies. The idea is then to increase the thickness of the slab until the surface energy σ converges.

---

1   inkscape images/surface-construction.svg

./images/surface-construction.svg

Unfortunately, this approach does not always work. The bulk system is treated subtly different than the slab system, particularly in the z-direction where the vacuum is (where typically only one k-point is used in slabs). Consequently, the k-point sampling is not equivalent in the two systems, and one can in general expect some errors due to this, with the best case being cancellation of the errors due to total k-point convergence. In the worst case, one can get a linear divergence in the surface energy with slab thickness [4].

A variation of this method that usually results in better k-point error cancellation is to calculate the bulk unit cell energy using the slab unit cell with no vacuum space, with the same k-point mesh in the x and y directions, but with increased k-points in the z-direction. Thus, the bulk system and slab system have the same Brillouin zone in at least two dimensions. This maximizes the cancellation of k-point errors, but still does not guarantee convergence of the surface energy, as discussed in [4, 5].

For quick estimates of the surface energy, one of the methods described above is likely sufficient. The advantage of these methods is the small number of calculations required to obtain the estimate, one needs only a bulk calculation (which must be done anyhow to get the bulk lattice constant to create the slab), and a slab calculation that is sufficiently thick to get the estimate. Additional calculations are only required to test the convergence of the surface energy.

An alternative method for calculating surface energies that does not involve an explicit bulk calculation follows Ref. [5]. The method follows

from equation (ref{eq:se}) where for a N-atom slab, in the limit of $N \to \infty$,

$E_{slab} \approx 2\sigma + \frac{N_{slab}}{N_{bulk}} E_{bulk}$

Then, we can estimate E_{bulk} by plotting the total energy of the slab as a function of the slab thickness.

$\sigma = \lim N \to \infty \frac{1}{2}(E_{slab}^N - N\Delta E_N)$

where $\Delta E_N = E_{slab}^N - E_{slab}^{N-1}$.

We will examine this approach here. We will use unrelaxed slabs for computational efficiency.

```
1   from jasp import *
2   from ase.units import *
3   from ase.lattice.surface import *
4   import matplotlib.pyplot as plt
5
6   Nlayers = [3, 4, 5, 6, 7, 8, 9, 10, 11]
7   energies = []
8   sigmas = []
9
10  for n in Nlayers:
11
12      slab = fcc111('Cu', size=(1,1,n), vacuum=10.0)
13      slab.center()
14
15      with jasp('bulk/Cu-layers/{0}'.format(n),
16                xc='PBE',
17                encut=350,
18                kpts=(8,8,1),
19                atoms=slab) as calc:
20          calc.set_nbands(f=2) # the default nbands in Vasp is too low for Al
21          try:
22              energies.append(slab.get_potential_energy())
23          except (VaspSubmitted, VaspQueued):
24              pass
25
26  for i in range(len(Nlayers)-1):
27      N = Nlayers[i]
28      DeltaE_N = energies[i+1] - energies[i]
29      sigma = 0.5*(-N*energies[i+1] + (N+1)*energies[i])
30      sigmas.append(sigma)
31      print 'sigma = {0} eV/atom'.format(sigma)
32
33  plt.plot(Nlayers[0:-1], sigmas,'bo-')
34  plt.xlabel('Number of layers')
35  plt.ylabel('Surface energy (eV/atom)')
36  plt.savefig('images/Cu-unrelaxed-surface-energy.png')
```
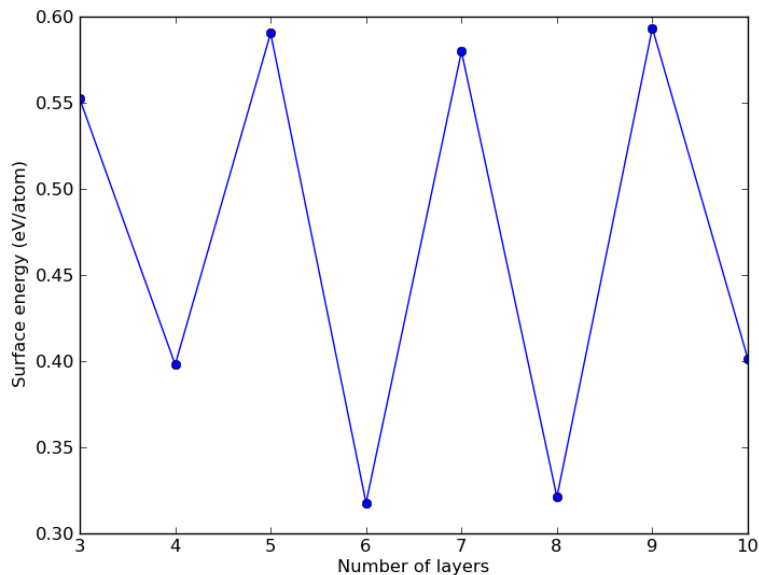
One
reason for the oscillations may be quantum size effects [7]. In [30] the surface energy of Cu(111) is reported as 0.48 eV/atom, or 1.36 J/m^2. Here is an example showing a conversion between these two units. We use ase to compute the area of the unit cell from the norm of the cross-product of the vectors defining the surface unit cell.

```
1  from ase.lattice.surface import *
2  from ase.units import *
3  import numpy as np
4
5  slab = fcc111('Cu', size=(1,1,3), vacuum=10.0)
6  cell = slab.get_cell()
7
8  area = np.linalg.norm(np.cross(cell[0], cell[1]))
9
10 sigma = 0.48 #eV/atom
11
12 print 'sigma = {0} J/m^2'.format(sigma/area/(J/m**2))
```

5.4.1 Advanced topics in surface energy

The surface energies can be used to estimate the shapes of nanoparticles using a Wulff construction. See [9] for an example of computing Mo2C surface energies and particle shapes, and [33] for an example of the influence of adsorbates on surface energies and particle shapes of Cu.

100

5.5 Dipole correction

Slabs that are not symmetric (e.g. they have adsorbates on only one side) can develop dipole moments. Periodic dipoles can be problematic, as they create an artificial field that may affect the surface properties such as the work function and the convergence of the calculations. The solution to this problem is to introduce a dipole field in the cell to exactly cancel the dipole moment [27] (see [2] for a small correction) to minimize the effect.

Here we will just illustrate the effect.

5.5.1 slab with no dipole correction

```python
#!/usr/bin/env python
from ase.lattice.surface import *
from jasp import *
from jasp.volumetric_data import *

slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
add_adsorbate(slab,'Na',height=1.2,position='fcc')

slab.center()

with jasp('surfaces/Al-Na-nodip',
          xc='PBE',
          encut=340,
          kpts=(2,2,1),
          lvtot=True, # write out local potential
          lvhar=True, # write out only electrostatic potential, not xc pot
          atoms=slab) as calc:
    calc.calculate()

    x,y,z,lp = calc.get_local_potential()
    nx, ny, nz = lp.shape

    axy = [np.average(lp[:,:,z]) for z in range(nz)]
    # setup the x-axis in realspace
    uc = atoms.get_cell()
    xaxis = np.linspace(0,uc[2][2],nz)

    plot(xaxis, axy)
    show()
```

5.5.2 slab with a dipole correction

Note this takes a considerably longer time to run than without a dipole correction! In Vasp there are several levels of dipole correction to apply. You can use the IDIPOL tag to turn it on, and specify which direction to apply it in (1=x, 2=y, 3=z, 4=all directions). This simply corrects the total energy and forces. It does not change the contents of LOCPOT. For that,

you ahve to also set the LDIPOL and DIPOL tags. It is not efficient to set all three at the same time for some reason. The Vasp manual recommends you first set IDIPOL to get a converged electronic structure, and then set LDIPOL to True, and set the center of electron density in DIPOL. That makes these calculations a multistep process, because we must run a calculation, analyze the charge density to get the center of charge, and then run a second calculation.

```python
from ase.lattice.surface import *
from jasp import *
JASPRC['mode'] = 'run'
slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
add_adsorbate(slab,'Na',height=1.2,position='fcc')

slab.center()

with jasp('surfaces/Al-Na-dip',
          xc='PBE',
          encut=340,
          kpts=(2,2,1),
          idipol=3,   # only along z-axis
          lvtot=True, # write out local potential
          lvhar=True, # write out only electrostatic potential, not xc pot
          atoms=slab) as calc:
    # pre-converge
    calc.calculate()

    x,y,z,cd = calc.get_charge_density()
    n0, n1, n2 = cd.shape
    nelements = n0*n1*n2
    voxel_volume = slab.get_volume()/nelements
    total_electron_charge = cd.sum()*voxel_volume

    electron_density_center = np.array([(cd*x).sum(),
                                        (cd*y).sum(),
                                        (cd*z).sum()])
    electron_density_center *= voxel_volume
    electron_density_center /= total_electron_charge

    print electron_density_center
    uc = slab.get_cell()

    # get scaled electron charge density center
    sedc = np.dot(np.linalg.inv(uc.T),electron_density_center.T).T

    ## # now turn on potential calculation: violates jasp workflow!
    calc.set(ldipol=True, dipol=sedc)

    calc.calculate()
```

### 5.5.3 Comparing no dipole correction with a dipole correction

To see the difference in what the dipole correction does, we now plot the potentials from each calculation.

```python
from jasp import *
from pylab import *

with jasp('surfaces/Al-Na-nodip') as calc:
    atoms = calc.get_atoms()

    x,y,z,lp = calc.get_local_potential()
    nx, ny, nz = lp.shape

    axy_1 = [np.average(lp[:,:,z]) for z in range(nz)]
    # setup the x-axis in realspace
    uc = atoms.get_cell()
    xaxis_1 = np.linspace(0,uc[2][2],nz)

    e1 = atoms.get_potential_energy()

with jasp('surfaces/Al-Na-dip') as calc:
    atoms = calc.get_atoms()

    x,y,z,lp = calc.get_local_potential()
    nx, ny, nz = lp.shape

    axy_2 = [np.average(lp[:,:,z]) for z in range(nz)]
    # setup the x-axis in realspace
    uc = atoms.get_cell()
    xaxis_2 = np.linspace(0,uc[2][2],nz)

    ef2 = calc.get_fermi_level()
    e2 = atoms.get_potential_energy()

print 'The difference in energy is {0} eV.'.format(e2-e1)

plot(xaxis_1, axy_1, label='no dipole correction')
plot(xaxis_2, axy_2, label='dipole correction')
plot([min(xaxis_2), max(xaxis_2)],[ef2,ef2], 'k:', label='Fermi level')
xlabel('z ($\AA$)')
ylabel('xy-averaged electrostatic potential')
legend(loc='best')
savefig('images/dip-vs-nodip-esp.png')
```

The key points to notice in this figure are:

1. The two deep dips are where the atoms are.

2. Without a dipole correction, the electrostatic potential never flattens out. there is near constant slope in the vacuum region, which means there is an electric field there.

3. With a dipole moment the potential is flat in the vacuum region, except for the step jump near 23 $\AA$.

4. The difference between the Fermi level and the flat vacuum potential is the work function.

5. The difference in energy with and without the dipole correction here is small.
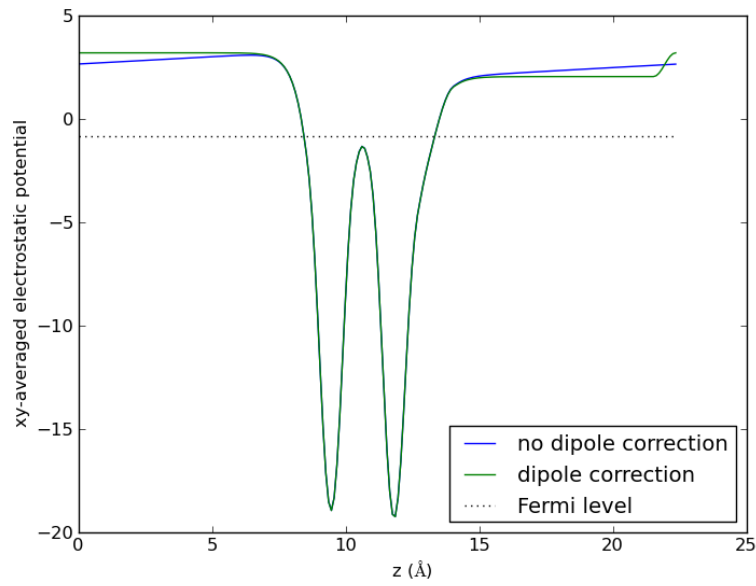
5.6 Adsorption energies

6 TODO Atomistic thermodynamics

7 Advanced electronic structure methods

7.1 TODO DFT+U

7.2 TODO Hybrid functionals

8 Appendices

8.1 Recipes

8.1.1 Modifying Atoms by deleting atoms

Sometimes it is convenient to create an Atoms object by deleting atoms from an existing object. Here is a recipe to delete all the hydrogen atoms in a molecule. The idea is to make a list of indices of which atoms to delete using list comprehension, then use list deletion to delete those indices.

```python
from ase.data.molecules import molecule
atoms = molecule('CH3CH2OH')
print atoms

ind2del = [atom.index for atom in atoms if atom.symbol=='H']
print ind2del

del atoms[ind2del]
print atoms
```

8.1.2 Using units in ase

```python
#!/usr/bin/env python
from ase.units import *

d = 1*Angstrom
print d/nm

e = 1*eV
print '1 eV = %f Hartrees' % (e/Hartree)
print '1 eV = %f Rydbergs' % (e/Rydberg)
print '1 eV = %f kJ/mol' % (e/(kJ/mol))
print '1 eV = %f kcal/mol' % (e/(kcal/mol))

print '1 Hartree = %f kcal/mol' % (1*Hartree/(kcal/mol))
print '1 Rydberg = %f eV' % (1*Rydberg/eV)
```

8.1.3 numpy arrays

See http://www.scipy.org/Cookbook/BuildingArrays for examples of making numpy arrays.

4.1.1. Extracting parts of an array

When analyzing numerical data you may often want to analyze only a part of the data. For example, suppose you have x and y data, (x=time, y=signal) and you want to integrate the date between a particular time interval. You can slice a numpy array to extract parts of it. See http://www.scipy.org/Cookbook/Indexing for several examples of this.

In this example we show how to extract the data in an interval. We have x data in the range of 0 to 6, and y data that is the cos(x). We want to extract the x and y data for 2 ¡ x ¡ 4, and the corresponding y-data. To do this, we utilize the numpy capability of slicing with a boolean array. We also show some customization of matplotlib.

```python
#!/usr/bin/env python

import numpy as np
import matplotlib as mpl
#http://matplotlib.sourceforge.net/users/customizing.html
mpl.rcParams['legend.numpoints'] = 1 #default is 2
from pylab import *

x = np.linspace(0,6,100)
y = np.cos(x)

plot(x,y,label='full')

ind = (x>2) & (x<4)

subx = x[ind]
suby = y[ind]

plot(subx,suby,'bo',label='sliced')
xlabel('x')
ylabel('cos(x)')
legend(loc='lower right')
savefig('images/np-array-slice.png')
```
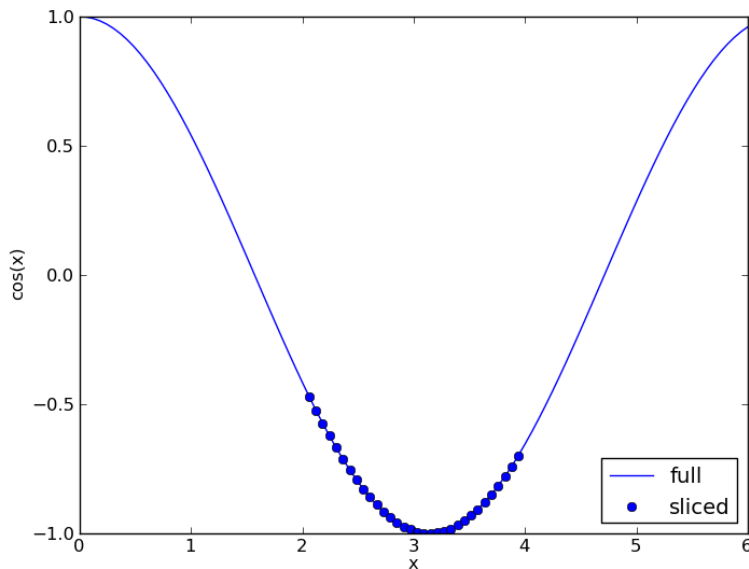
The
expression x¿2 returns an array of booleans (True where the element of x is
greater than 2, and False where it is not) equal in size to x. Similarly x¡4
returns a boolean array where x is less than 4. We take the logical and & of
these two boolean arrays to get another boolean array where both conditions
are True (i.e. x¡2 and x¿4). This final boolean array is True for the part of
the arrays we are interested in, and we can use it to extract the subarrays
we want.

8.1.4   Statistics

Confidence intervals

```python
#!/usr/bin/env python
from numpy import *
from scipy.stats.distributions import  t


n = 10 #number of measurements
dof = n - 1 #degrees of freedom
avg_x = 16.1 #average measurement
std_x = 0.01 #standard deviation of measurements

#Find 95% prediction interval for next measurement

alpha = 1.0 - 0.95

```

```
15    pred_interval = t.ppf(1-alpha/2.,dof)*std_x*sqrt(1.+1./n)
16
17    print 'We are 95%% confident the next measurement will be between %1.3f and %1.3f' % (avg_x - pred_interval, avg_x
```
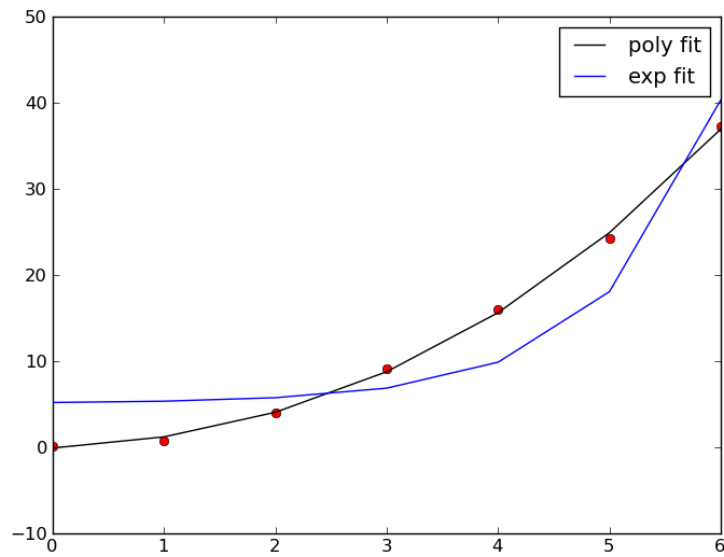
8.1.5 Curve fitting

Linear fitting

```
1     #examples of linear curve fitting using least squares
2     import numpy as np
3
4     xdata = np.array([0.,1.,2.,3.,4.,5.,6.])
5     ydata = np.array([0.1, 0.81, 4.03, 9.1, 15.99, 24.2, 37.2])
6
7     #fit a third order polynomial
8     from pylab import polyfit, plot, show, legend, savefig
9     pars = polyfit(xdata,ydata,3)
10    print pars
11
12    ## numpy method returns more data
13    A = np.column_stack([xdata**3,xdata**2,xdata,np.ones(len(xdata),np.float)])
14    pars_np,resids,rank,s = np.linalg.lstsq(A,ydata)
15    print pars_np
16
17    '''
18    we are trying to solve Ax = b for x in the least squares sense. There
19    are more rows in A than elements in x so, we can left multiply each
20    side by A^T, and then solve for x with an inverse.
21
22    A^TAx = A^Tb
23    x = (A^TA)^-1 A^T b
24    '''
25    # not as pretty but equivalent!
26    pars_man= np.dot(np.linalg.inv(np.dot(A.T,A)), np.dot(A.T,ydata))
27    print pars_man
28
29    #but, it is easy to fit an exponential function to it!
30    # y = a*exp(x)+b
31    Aexp = np.column_stack([np.exp(xdata), np.ones(len(xdata),np.float)])
32    pars_exp=np.dot(np.linalg.inv(np.dot(Aexp.T,Aexp)), np.dot(Aexp.T,ydata))
33
34    plot(xdata,ydata,'ro')
35    fity = np.dot(A,pars)
36    plot(xdata,fity,'k-',label='poly fit')
37    plot(xdata,np.dot(Aexp,pars_exp),'b-',label='exp fit')
38    legend()
39    savefig('images/curve-fit-1.png')
```

8.1.6 Nonlinear curve fitting

```python
from scipy.optimize import leastsq
import numpy as np

vols = np.array([13.71, 14.82, 16.0, 17.23, 18.52])

energies = np.array([-56.29, -56.41, -56.46, -56.463,-56.41])

def Murnaghan(parameters,vol):
    'From PRB 28,5480 (1983'
    E0 = parameters[0]
    B0 = parameters[1]
    BP = parameters[2]
    V0 = parameters[3]

    E = E0 + B0*vol/BP*(((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)

    return E

def objective(pars,y,x):
    #we will minimize this function
    err =  y - Murnaghan(pars,x)
    return err

x0 = [ -56., 0.54, 2., 16.5] #initial guess of parameters

plsq = leastsq(objective, x0, args=(energies,vols))
```

```
27
28    print plsq
29
30    from pylab import *
31    plot(vols,energies,'ro')
32
33    #plot the fitted curve on top
34    x = np.linspace(min(vols),max(vols),50)
35    y = Murnaghan(plsq[0],x)
36    plot(x,y,'k-')
37    xlabel('Volume')
38    ylabel('energy')
39    savefig('images/nonlinear-curve-fitting.png')
```


See additional examples at http://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html

8.1.7   Nonlinear curve fitting by direct least squares mimimization
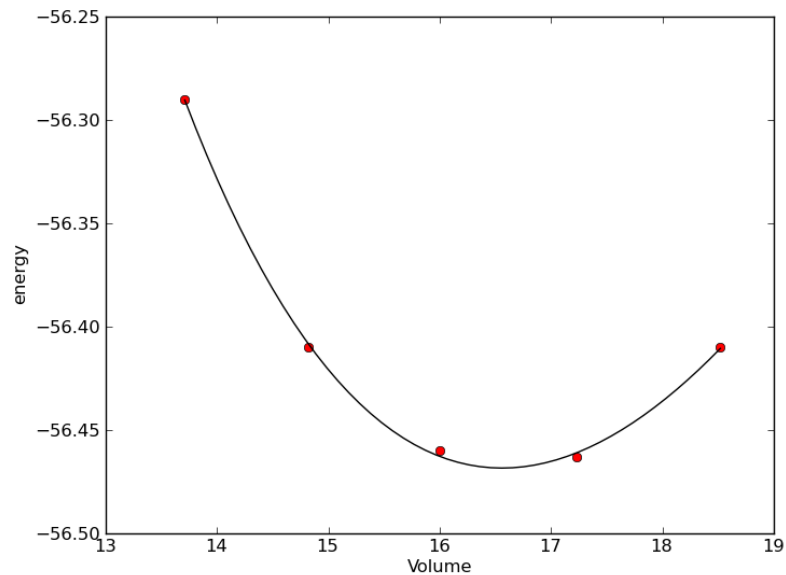
```
1    from scipy.optimize import fmin
2    import numpy as np
3
4    volumes = np.array([13.71, 14.82, 16.0, 17.23, 18.52])
5
6    energies = np.array([-56.29, -56.41, -56.46, -56.463,-56.41])
7
8    def Murnaghan(parameters,vol):
9        'From PRB 28,5480 (1983'
```

```
10      E0 = parameters[0]
11      B0 = parameters[1]
12      BP = parameters[2]
13      V0 = parameters[3]
14
15      E = E0 + B0*vol/BP*(((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
16
17      return E
18
19  def objective(pars,vol):
20      #we will minimize this function
21      err =  energies - Murnaghan(pars,vol)
22      return np.sum(err**2) #we return the summed squared error directly
23
24  x0 = [ -56., 0.54, 2., 16.5] #initial guess of parameters
25
26  plsq = fmin(objective,x0,args=(volumes,)) #note args is a tuple
27
28  print plsq
29
30  from pylab import *
31  plot(volumes,energies,'ro')
32
33  #plot the fitted curve on top
34  x = np.linspace(min(volumes),max(volumes),50)
35  y = Murnaghan(plsq,x)
36  plot(x,y,'k-')
37  savefig('images/nonlinear-fitting-lsq.png')
```
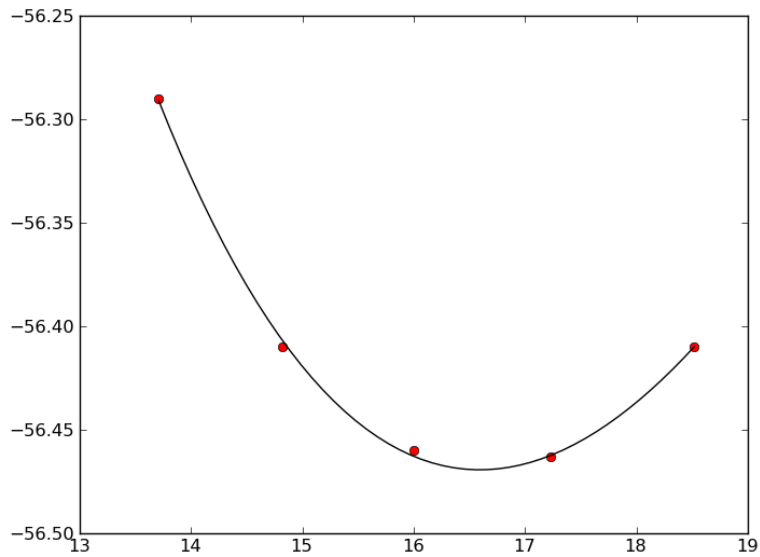
8.1.8   TODO Interpolation in 3D

You might ask, why would I need to interpolate in 3D? Suppose you want to plot the charge density along a line through a unit cell that does not correspond to grid points? What are you to do? Interpolate. In contrast to an abundance of methods for 1D and 2D interpolation, I could not find any standard library methods for 3D interpolation. The Scipy cookbook on interpolation suggests that it should be doable with ndimage, but I could not figure out how to make those examples work.

The principle we will use to develop an interpolation function in 3D is called trilinear interpolation, where we use multiple linear 1D interpolations to compute the value of a point inside a cube. As developed here, this solution only applies to rectangular grids. Later we will generalize the approach. We state the problem as follows:

we know a scalar field inside a unit cell on a regularly spaced grid. In Dacapo these fields may be the charge density or electrostatic potential for example, and they are known on the fft grids. we want to estimate the value of the scalar field at a point not on the grid, say P=(a,b,c).

Solution: Find the cube that contains the point, and is defined by points P1-P8 as shown in Figure Trilinear interpolation.

```
1   inkscape images/trilinear-interpolation.svg
```

```
1   convert images/trilinear-interpolation.svg images/trilinear-interpolation.png
```
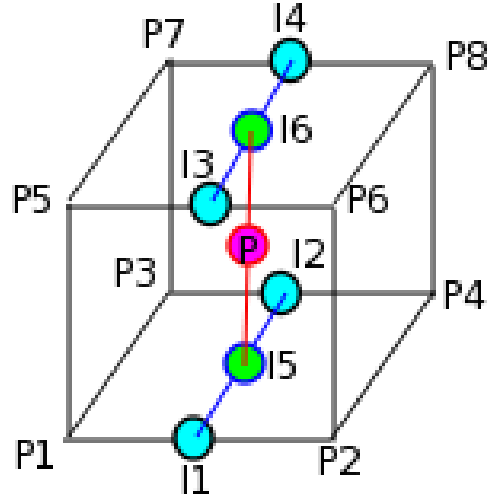
In a trilinear interpolation scheme in a cell we perform several interpolations. The red lines are 1D interpolations in x along constant y, and z. The light blue lines are 1D interpolations in y along constant x=a, z, and the vertical dark line through P is the final interpolation at constant x=a, y=b in z.

Now, we use 1D interpolation formulas to compute the value of the scalar field at points I1 by interpolating between P1 and P2, and the value of the scalar field at I2 by interpolating between P3 and P4. In these points the only variable changing is x, so it is a simple 1D interpolation. We can then compute the value of the scalar field at I5 by interpolating between I1 and I2. We repeat the process on the top of the cube, to obtain points I3, I4 and I5. Finally, we compute the value of the scalar field at point P by interpolating between points I5 and I6. Note that the point I5 has coordinates (a,b,z1) and (a,b,z2), so the final interpolation is again a 1D interpolation along z evaluated at z=c to get the final value of the scalar field at P=(a,b,c).

```python
from jasp import *
from ase.calculators.vasp import *

with jasp('molecules/co-centered') as calc:
    atoms = calc.get_atoms()

    vcd = VaspChargeDensity()

    cd = np.array(vcd.chg[0])
    n0, n1, n2 = cd.shape

    s0 = 1.0/n0
    s1 = 1.0/n1
    s2 = 1.0/n2

    X, Y, Z = np.mgrid[0.0:1.0:s0,
                       0.0:1.0:s1,
                       0.0:1.0:s2]

    C = np.column_stack([X.ravel(),
                         Y.ravel(),
                         Z.ravel()])

    atoms = calc.get_atoms()
    uc = atoms.get_cell()
    real = np.dot(C, uc)

    #now convert arrays back to unitcell shape
    x = np.reshape(real[:, 0], (n0, n1, n2))
    y = np.reshape(real[:, 1], (n0, n1, n2))
    z = np.reshape(real[:, 2], (n0, n1, n2))

def interp3d(x,y,z,cd,xi,yi,zi):
    '''
    interpolate a cubic 3D grid defined by x,y,z,cd at the point
    (xi,yi,zi)
    '''

    def get_index(value,vector):
        '''
        assumes vector ordered decreasing to increasing. A bisection
        search would be faster.
        '''
        for i,val in enumerate(vector):
            if val > value:
                return i-1
        return None

    xv = x[:,0,0]
    yv = y[0,:,0]
    zv = z[0,0,:]

    a,b,c = xi, yi, zi

    i = get_index(a,xv)
    j = get_index(b,yv)
```

```python
57          k = get_index(c,zv)
58
59          x1 = x[i,j,k]
60          x2 = x[i+1,j,k]
61          y1 = y[i,j,k]
62          y2 = y[i,j+1,k]
63          z1 = z[i,j,k]
64          z2 = z[i,j,k+1]
65
66          u1 = cd[i, j, k]
67          u2 = cd[i+1, j, k]
68          u3 = cd[i, j+1, k]
69          u4 = cd[i+1, j+1, k]
70          u5 = cd[i, j, k+1]
71          u6 = cd[i+1, j, k+1]
72          u7 = cd[i, j+1, k+1]
73          u8 = cd[i+1, j+1, k+1]
74
75          w1 = u2 + (u2-u1)/(x2-x1)*(a-x2)
76          w2 = u4 + (u4-u3)/(x2-x1)*(a-x2)
77          w3 = w2 + (w2-w1)/(y2-y1)*(b-y2)
78          w4 = u5 + (u6-u5)/(x2-x1)*(a-x1)
79          w5 = u7 + (u8-u7)/(x2-x1)*(a-x1)
80          w6 = w4 + (w5-w4)/(y2-y1)*(b-y1)
81          w7 = w3 + (w6-w3)/(z2-z1)*(c-z1)
82          u = w7
83
84          return u
85
86  pos = atoms.get_positions()
87
88  P1 = np.array([0.0, 5.0, 5.0])
89  P2 = np.array([9.0, 5.0, 5.0])
90
91  npoints = 60
92
93  points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]
94
95  R = [np.linalg.norm(p-P1) for p in points]
96
97  # interpolated line
98  icd = [interp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]
99
100 from pylab import *
101
102 plot(R,icd)
103 cR = np.linalg.norm(pos[0]-P1)
104 oR = np.linalg.norm(pos[1]-P1)
105 plot([cR,cR],[0,2],'r-') #markers for where the nuclei are
106 plot([oR,oR],[0,8],'r-')
107 xlabel('|R| ($\AA$)')
108 ylabel('Charge density (e/$\AA^3$)')
109 savefig('images/CO-charge-density.png')
110 show()
```
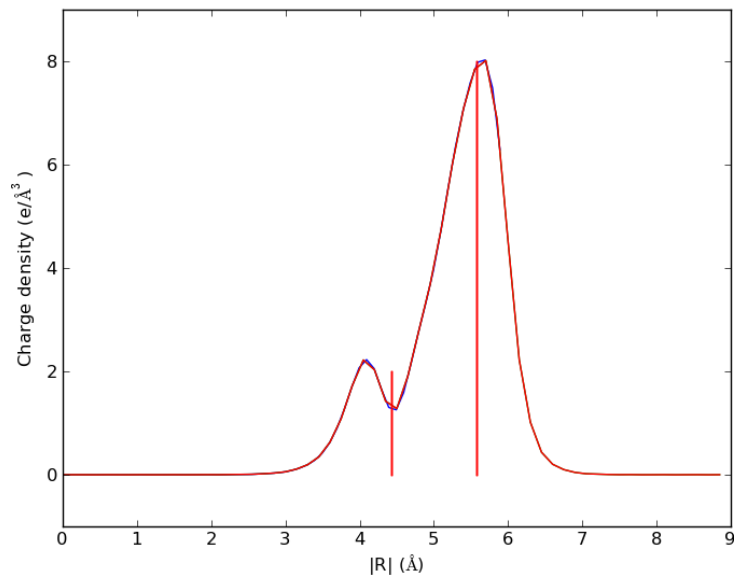
To generalize this to non-cubic cells, we need to do interpolation along arbitrary vectors. The overall strategy is the same:

Find the cell that contains the point (a,b,c). compute the scaled coordinates (sa,sb,sc) of the point inside the cell. Do the interpolations along the basis vectors. Given u1 at P1(x1,y1,z1) and u2 at P2(x2,y2,z2) where (P2-P1) is a cell basis vector a, u = u1 + sa*(u2-u1). There are still 7 interpolations to do.

Below is an example of this code, using a the python library bisect to find the cell.

```python
'''
3D vector interpolation in non-cubic unit cells with vector
interpolation.

This function should work for any shape unit cell.
'''
from ase.calculators.jacapo import Jacapo
import bisect
import numpy as np

atoms = Jacapo.read_atoms('/home/jkitchin/www/dft/source/jacapo/1-molecules/1.2-calculating-simple-properties/1.2.6
calc = atoms.get_calculator()
x,y,z,cd = calc.get_charge_density()

def vinterp3d(x,y,z,u,xi,yi,zi):

```

```
17        p = np.array([xi,yi,zi])
18
19        #1D arrays of cooridinates
20        xv = x[:,0,0]
21        yv = y[0,:,0]
22        zv = z[0,0,:]
23
24        # we subtract 1 because bisect tells us where to insert the
25        # element to maintain an ordered list, so we want the index to the
26        # left of that point
27        i = bisect.bisect_right(xv,xi) - 1
28        j = bisect.bisect_right(yv,yi) - 1
29        k = bisect.bisect_right(zv,zi) - 1
30
31        #points at edge of cell. We only need P1, P2, P3, and P5
32        P1 = np.array([x[i,j,k],y[i,j,k],z[i,j,k]])
33        P2 = np.array([x[i+1,j,k],y[i+1,j,k],z[i+1,j,k]])
34        P3 = np.array([x[i,j+1,k],y[i,j+1,k],z[i,j+1,k]])
35        P5 = np.array([x[i,j,k+1],y[i,j,k+1],z[i,j,k+1]])
36
37        #values of u at edge of cell
38        u1 = u[i,j,k]
39        u2 = u[i+1,j,k]
40        u3 = u[i,j+1,k]
41        u4 = u[i+1,j+1,k]
42        u5 = u[i,j,k+1]
43        u6 = u[i+1,j,k+1]
44        u7 = u[i,j+1,k+1]
45        u8 = u[i+1,j+1,k+1]
46
47        #cell basis vectors, not the unit cell, but the voxel cell containing the point
48        cbasis = np.array([P2-P1,
49                           P3-P1,
50                           P5-P1])
51
52        #now get interpolated point in terms of the cell basis
53        s = np.dot(np.linalg.inv(cbasis.T),np.array([xi,yi,zi])-P1)
54
55        #now s = (sa, sb, sc) which are fractional coordinates in the vector space
56        #next we do the interpolations
57        ui1 = u1 + s[0]*(u2-u1)
58        ui2 = u3 + s[0]*(u4-u3)
59
60        ui3 = u5 + s[0]*(u6-u5)
61        ui4 = u7 + s[0]*(u8-u7)
62
63        ui5 = ui1 + s[1]*(ui2-ui1)
64        ui6 = ui3 + s[1]*(ui4-ui3)
65
66        ui7 = ui5 + s[2]*(ui6-ui5)
67
68        return ui7
69
70
71    # compute a line with 60 points in it through these two points
72    P1 = np.array([0.0, 3.0, 3.0])
```

```
73    P2 = np.array([5.0, 3.0, 3.0])

74

75    npoints = 60

76

77    points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]

78

79    # compute the distance along the line
80    R = [np.linalg.norm(p-P1) for p in points]

81

82    icd = [vinterp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]

83

84    from pylab import plot, xlabel, ylabel, show

85

86

87    plot(R,icd)
88    pos = atoms.get_positions()
89    cR = np.linalg.norm(pos[0]-P1)
90    oR = np.linalg.norm(pos[1]-P1)
91    plot([cR,cR],[0,2],'r-') #markers for where the nuclei are
92    plot([oR,oR],[0,8],'r-')
93    xlabel('|R| ($\AA$)')
94    ylabel('Charge density (e/$\AA^3$)')
95    show()
```

8.1.9   Reading and writing data

Builtin io modules   "pylab" has two convenient and powerful functions for saving and reading data, :func:'`pylab.save`' and :func:'`pylab.load`'.

```
1    pylab.save('pdat.dat',(x,y))
```

and later you can read these arrays back in with:

```
1    x,y = pylab.load('pdat.dat')
```

see also :func:'`pylab.csv2rec`' and :func:'`pylab.loadtxt`' and :func:'`pylab.savetxt`'.
See http://www.scipy.org/Cookbook/InputOutput for examples of numpy
io.

From scratch   You can save data in many ways from scratch. Basically, just open a file and write data to it. Likewise, any datafile that has some structure to it can probably be read by python.

Reading a datafile with ::

#header #ignore these lines john, 4 robert, 5 terry, 5

A standard approach would be to read in all the lines, skip the first two lines, split each line (remember each line is a string) at the ',', and append

the first field to one variable, and append the second field to another variable as an integer. For example::

```
1  v1 = []
2  v2 = []
3  lines = open('somefile','r').readlines()
4
5  for line in lines[2:]: #skip the first two lines
6      fields = line.split(',')
7      v1.append(fields[0]) #names
8      v2.append(int(fields[1])) #number
```

Writing datafiles is easy too.

```
1  v1 = ['john','robert','terry']
2  v2 = [4,5,6]
3  f = open('somefile', 'w') #note 'w' = write mode
4  f.write('#header\n')
5  f.write('#ignore these lines\n')
6  for a,b in zip(v1,v2):
7      f.write('%s, %i\n' % (a,b))
8  f.close()
```

Some notes:

1. opening a file in 'w' mode clobbers any existing file, so do that

with care!

1. when writing to a file you have to add
   n to each line. this is the

carriage return.

1. Manually writing and reading files is pretty tedious. Whenever

possible you should use the builtin methods of numpy or pylab.

8.1.10 Integration

Numerical integrations is easy with the numpy.trapz() method. Use it like this: numpy.trapz(y,x). Note that y comes first. y and x must be the same length.

Integration can be used to calculate average properties of continuous distributions. Suppose for example, we have a density of states, ρ as a

119

function of energy E. We can integrate the density of states to find the total number of states:

$$N_{states} = \int \rho dE$$

or, in python:

¿¿¿ Nstates = np.trapz(rho,E)

where rho is a vector that contains the density of states at each energy in the vector E (vector here means a list of numbers).

The average energy of distribution is:

$$E_{avg} = \frac{\int \rho E dE}{\int \rho dE}$$

or, in python:

¿¿¿ e$_{avg}$ = np.trapz(rho*E,E)/np.trapz(rho,E)

These last two examples are the zeroth and first moments of the density of states. The second moment is related to the width squared of the distribution, and the third and fourth moements are related to skewness and kurtosis of the distribution.

The nth moment is defined by:

$$m_n = \frac{\int \rho * E^n dE}{\int \rho dE}$$

To get the second moment of the density of states in python, we use::

¿¿¿ n = 2 ¿¿¿ mom$_2$ = np.trapz(rho*E**n,E)/np.trapz(rho,E)

8.1.11 Numerical differentiation

numpy has a function called numpy.diff() that is similar to the one found in matlab. It calculates the differences between the elements in your list, and returns a list that is one element shorter, which makes it unsuitable for plotting the derivative of a function. 4.3.1. Simple loops to define finite difference derivatives

Loops in python are pretty slow (relatively speaking) but they are usually trivial to understand. In this script we show some simple ways to construct derivative vectors using loops. It is implied in these formulas that the data points are equally spaced.

```python
1   import numpy as np
2   from pylab import *
3   import time
4
5   '''
6   These are the brainless way to calculate numerical derivatives. They
7   work well for very smooth data. they are surprisingly fast even up to
8   10000 points in the vector.
9   '''
10
```

```
11   x = np.linspace(0.78,0.79,100)
12   y = np.sin(x)
13   dy_analytical = np.cos(x)
14   '''
15   lets use a forward difference method:
16   that works up until the last point, where there is not
17   a forward difference to use. there, we use a backward difference.
18   '''
19
20   tf1 = time.time()
21   dyf = [0.0]*len(x)
22   for i in range(len(y)-1):
23       dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
24   #set last element by backwards difference
25   dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
26
27   print ' Forward difference took %1.1f seconds' % (time.time() - tf1)
28
29   '''and now a backwards difference'''
30   tb1 = time.time()
31   dyb = [0.0]*len(x)
32   #set first element by forward difference
33   dyb[0] = (y[0] - y[1])/(x[0] - x[1])
34   for i in range(1,len(y)):
35       dyb[i] = (y[i] - y[i-1])/(x[i]-x[i-1])
36
37   print ' Backward difference took %1.1f seconds' % (time.time() - tb1)
38
39   '''and now, a centered formula'''
40   tc1 = time.time()
41   dyc = [0.0]*len(x)
42   dyc[0] = (y[0] - y[1])/(x[0] - x[1])
43   for i in range(1,len(y)-1):
44       dyc[i] = (y[i+1] - y[i-1])/(x[i+1]-x[i-1])
45   dyc[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
46
47   print ' Centered difference took %1.1f seconds' % (time.time() - tc1)
48
49   '''
50   the centered formula is the most accurate formula here
51   '''
52
53   plot(x,y)
54   plot(x,dy_analytical,label='analytical derivative')
55   plot(x,dyf,'--',label='forward')
56   plot(x,dyb,'--',label='backward')
57   plot(x,dyc,'--',label='centered')
58
59   legend(loc='lower left')
60   savefig('images/simple-diffs.png')
61   show()
```
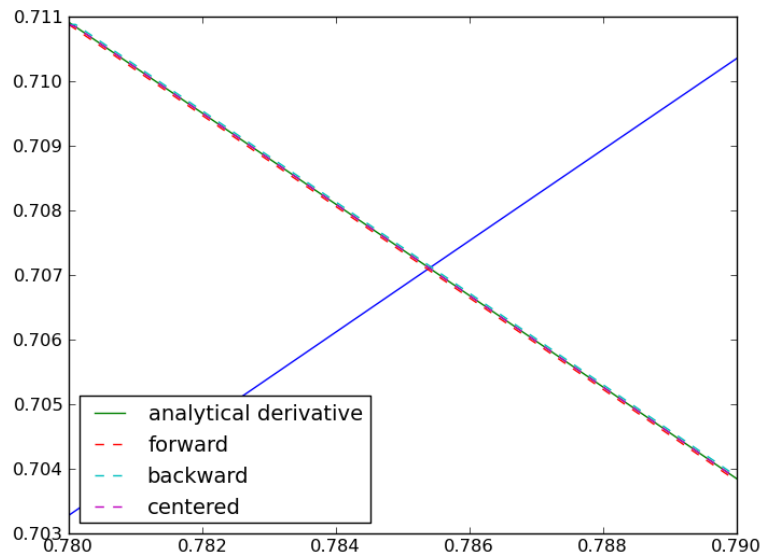
Loops are usually not great for performance. Numpy offers some vectorized methods that allow us to compute derivatives without loops, although this comes at the mental cost of harder to understand syntax:

```python
import numpy as np
from pylab import *

x = np.linspace(0,2*np.pi,100)
y = np.sin(x)
dy_analytical = np.cos(x)


# we need to specify the size of dy ahead because diff returns
#an array of n-1 elements
dy = np.zeros(y.shape,np.float) #we know it will be this size
dy[0:-1] = np.diff(y)/np.diff(x)
dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])


'''
calculate dy by center differencing using array slices
'''

dy2 = np.zeros(y.shape,np.float) #we know it will be this size
dy2[1:-1] = (y[2:] - y[0:-2])/(x[2:] - x[0:-2])
dy2[0] = (y[1]-y[0])/(x[1]-x[0])
dy2[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

plot(x,y)
```

```
26    plot(x,dy_analytical,label='analytical derivative')
27    plot(x,dy,label='forward diff')
28    plot(x,dy2,'k--',lw=2,label='centered diff')
29    legend(loc='lower left')
30    savefig('images/vectorized-diffs.png')
31    show()
```
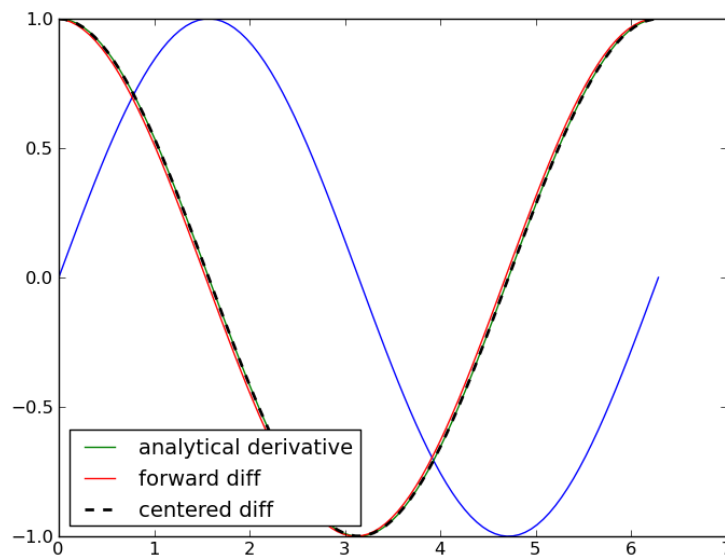

If your data is very noisy, you will have a hard time getting good derivatives; derivatives tend to magnify noise. In these cases, you have to employ smoothing techniques, either implicitly by using a multipoint derivative formula, or explicitly by smoothing the data yourself, or taking the derivative of a function that has been fit to the data in the neighborhood you are interested in.

Here is an example of a 4-point centered difference of some noisy data:
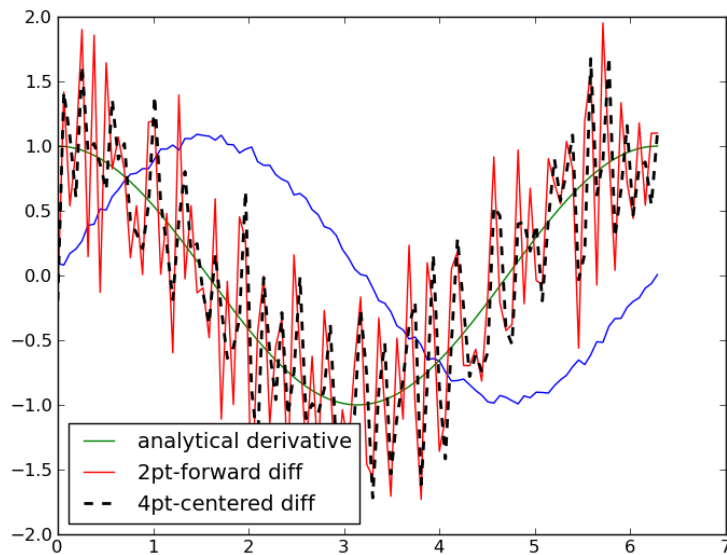
```
1     import numpy as np
2     from pylab import *
3
4     x = np.linspace(0,2*np.pi,100)
5     y = np.sin(x) + 0.1*np.random.random(size=x.shape)
6     dy_analytical = np.cos(x)
7
8     #2-point formula
9     dyf = [0.0]*len(x)
10    for i in range(len(y)-1):
11        dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
```

```
12    #set last element by backwards difference
13    dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
14
15    '''
16    calculate dy by 4-point center differencing using array slices
17
18    \frac{y[i-2] - 8y[i-1] + 8[i+1] - y[i+2]}{12h}
19
20    y[0] and y[1] must be defined by lower order methods
21    and y[-1] and y[-2] must be defined by lower order methods
22    '''
23
24    dy = np.zeros(y.shape,np.float) #we know it will be this size
25    h = x[1]-x[0] #this assumes the points are evenely spaced!
26    dy[2:-2] = (y[0:-4] - 8*y[1:-3] + 8*y[3:-1] - y[4:])/(12.*h)
27
28    dy[0] = (y[1]-y[0])/(x[1]-x[0])
29    dy[1] = (y[2]-y[1])/(x[2]-x[1])
30    dy[-2] = (y[-2] - y[-3])/(x[-2] - x[-3])
31    dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
32
33
34    plot(x,y)
35    plot(x,dy_analytical,label='analytical derivative')
36    plot(x,dyf,'r-',label='2pt-forward diff')
37    plot(x,dy,'k--',lw=2,label='4pt-centered diff')
38    legend(loc='lower left')
39    savefig('images/multipt-diff.png')
40    show()
```

The derivative is still noisy, but the four-point derivative is a little better than the two-pt formula.

FFT derivatives   It is possible to perform derivatives using FFT:

```python
import numpy as np
from pylab import *

N = 101 #number of points
L = 2*np.pi #interval of data

x = np.arange(0.0,L,L/float(N)) #this does not include the endpoint

#add some random noise
y = np.sin(x) + 0.05*np.random.random(size=x.shape)
dy_analytical = np.cos(x)

'''
http://sci.tech-archive.net/Archive/sci.math/2008-05/msg00401.html

you can use fft to calculate derivatives!
'''

if N % 2 == 0:
    k = np.asarray(range(0,N/2)+[0] + range(-N/2+1,0))
else:
    k = np.asarray(range(0,(N-1)/2) +[0] + range(-(N-1)/2,0))

k *= 2*np.pi/L

fd = np.fft.ifft(1.j*k * np.fft.fft(y))

plot(x,y)
plot(x,dy_analytical,label='analytical der')
plot(x,fd,label='fft der')
legend(loc='lower left')

savefig('images/fft-der.png')
show()
```
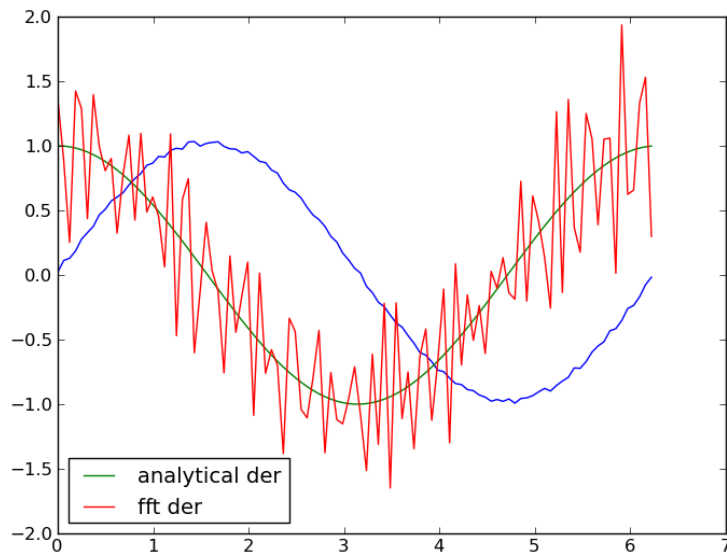
This example does not show any major advantage in the quality of the derivative, and it is almost certain I would never remember how to do this off the top of my head.

8.1.12 NetCDF files

NetCDF is a binary, but cross-platform structured data format. The input file and output file for Dacapo is the NetCDF format. On creating a NetCDF file you must define the dimensions and variables before you can store data in them. You can create and read NetCDF files in python using one of the following modules:

(http://dirac.cnrs-orleans.fr/plone/software/scientificpython/)

:mod:'netCDF3' (http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF3-module.html)

:mod:'pycdf' (http://pysclint.sourceforge.net/pycdf/) this is a very low level module modelled after the C-api. I am not sure it is completely bug-free (I have problems with character variables)

I suggest you read through the jacapo.py code for many examples of reading and writing NetCDF files.

8.1.13   python modules

the comma separated values (:mod:`csv`) module in python allows you to easily create datafiles:

"csv" writing::

import numpy as np

x = np.linspace(0.0,6.0,100) y = np.cos(x)

import csv writer = csv.writer(open("some.csv", "w")) writer.writerows(zip(x,y))

It is not so easy to read the data back in though because the module only returns strings, so you must turn the strings back into floats (or whatever other format they should be).

"csv" reading::

import csv reader = csv.reader(open("some.csv",'r'),delimiter=',')

x,y = [],[] for row in reader: #csv returns strings that must be cast as floats a,b = [float(z) for z in row] x.append(a) y.append(b)

This is almost as much work as manually reading the data though. The module is more powerful than I have shown here, so one day checkout "pydoc csv"

The :mod:`pickle` and :mod:`shelve` modules of python also offer some data storage functionality. Check them out some day too.

8.1.14   reading and writing excel files

8.2   Computational geometry

8.2.1   Changing coordinate systems

Let A, B, C be the unit cell vectors

1. A = A1 x + A2 y + A3 z

2. B = B1 x + B2 y + B3 z

3. C = C1 x + C2 y + C3 z

and we want to find the vector [s1 s2 s3] so that

1. P = s1 A + s2 B + s3 C

where

1. P = [p1x + p2y + p3z]

if we expand 4), we get: s1 A1 x + s1 A2 y + s1 A3 z

127

- s2 B1 x + s2 B2 y + s2 B3 z

- s3 C1 x + s3 C2 y + s3 C3 z = p1 x + p2 y + p3 z

If we now match coefficients on x, y, and z, we can write a set of linear equations as:

$$
\begin{matrix}
A1 & B1 & C1 \\
A2 & B2 & C2 \\
A3 & B3 & C3
\end{matrix}
\quad
\begin{matrix}
s1 \\
s2 \\
s3
\end{matrix}
\quad = \quad
\begin{matrix}
p1 \\
p2 \\
p3
\end{matrix}
$$

or, in standard form:

transpose(A) s = p

and we need to solve for s as:

$s = (A^T)^{-1} \cdot p$

p must be a column vector, so we will have to transpose the positions provided by the atoms class, and then transpose the final result to get the positions back into row-vector form:

$s = ((A^T)^{-1} p^T)^T$

Here we implement that in code:

```python
from ase import *
from ase.lattice.surface import *

np.set_printoptions(precision=3,suppress=True)

slab = fcc111('Pd',
              a=3.92,        # Pd lattice constant
              size=(2,2,3), #3-layer slab in 1x1 configuration
              vacuum=10.0)

pos = slab.get_positions() #these positions use x,y,z vectors as a basis

# we want to see the atoms in terms of the unitcell vectors
newbasis = slab.get_cell()

s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
print 'Coordinates in new bases are: \n',s

# what we just did is equivalent to the following atoms method
print 'Scaled coordinates from ase are: \n',slab.get_scaled_positions()
```

The method shown above is general to all basis set transformations. We examine another case next. Sometimes it is nice if all the coordinates are integers. For this example, we will use the bcc primitive lattice vectors and express the positions of each atom in terms of them. By definition each

atomic position should be an integer combination of the primitive lattice vectors (before relaxation, and assuming one atom is at the origin, and the unit cell is aligned with the primitive basis!)

```python
from ase.lattice.cubic import BodyCenteredCubic
from ase.calculators.jacapo import *

bulk = BodyCenteredCubic(directions=[[1,0,0],
                                     [0,1,0],
                                     [0,0,1]],
                         size=(2,2,2),
                         latticeconstant=2.87,
                         symbol='Fe')


newbasis = 2.87*np.array([[-0.5, 0.5, 0.5],
                          [0.5, -0.5, 0.5],
                          [0.5, 0.5, -0.5]])

pos = bulk.get_positions()

s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
print 'atom positions in primitive basis'
print s

#let's see the unit cell in terms of the primitive basis too
print 'unit cell in terms of the primitive basis'
print np.dot(np.linalg.inv(newbasis.T),bulk.get_cell().T).T
```

8.2.2 Simple distances, angles

Scientific.Geometry contains several useful functions for performing vector algebra including computing lengths and angles.

```python
import numpy as np
from Scientific.Geometry import *

A = Vector([1,1,1])    #Scientfic
a = np.array([1,1,1]) #numpy

B = Vector([0.0,1.0,0.0])

print '|A| = ',A.length()          #Scientific Python way
print '|a| = ',np.sum(a**2)**0.5 #numpy way
print '|a| = ',np.linalg.norm(a) #numpy way 2

print 'ScientificPython angle = ',A.angle(B) #in radians
print 'numpy angle =                ',np.arccos(np.dot(a/np.linalg.norm(a),B/np.linalg.norm(B)))

#cross products
print 'Scientific A .cross. B = ',A.cross(B)
print 'numpy A .cross. B      = ',np.cross(A,B) #you can use Vectors in numpy
```

8.2.3 Unit cell properties

The volume of a unit cell can be calculated from $V = (a_1 \times a_2) \cdot a_3$ where a_1, a_2 and a_3 are the unit cell vectors. It is more convenient, however, to simply evaluate that equation as the determinant of the matrix describing the unit cell, where each row of the matrix is a unit cell vector.

$V = |\det(ucell)|$

Why do we need to take the absolute value? The sign of the determinant depends on the handedness of the order of the unit cell vectors. If they are right-handed the determinant will be positive, and if they are left-handed the determinant will be negative. Switching any two rows will change the sign of the determinant and the handedness. :mod:'ase' implements a convenient function to get the volume of an :mod:'Atoms' object: :func:'ase.atoms.get_volume'.

Here are three equivalent ways to compute the unit cell volume.

```
import numpy as np

a1 = [2, 0, 0]
a2 = [1, 1, 0]
a3 = [0, 0, 10]

uc = np.array([a1, a2, a3])

print np.dot(np.cross(a1,a2),a3)
print np.linalg.det(uc)

from ase import *

atoms = Atoms([],cell=uc) #empty list of atoms
print atoms.get_volume()
```

8.2.4 d-spacing

If you like to set up the vacuum in your slab calculations in terms of equivalent layers of atoms, you need to calculate the d-spacing for the hkl plane you are using. The script below shows several ways to accomplish that.

```
from ase import *
import numpy as np
from ase.lattice.cubic import FaceCenteredCubic

ag = FaceCenteredCubic(directions=[[1,0,0],
                                   [0,1,0],
                                   [0,0,1]],
                       size=(1,1,1),
```

```python
 9                               symbol='Ag',
10                               latticeconstant=4.0)
11
12    # these are the reciprocal lattice vectors
13    b1,b2,b3 = np.linalg.inv(ag.get_cell())
14
15    '''
16    g(111) = 1*b1 + 1*b2 + 1*b3
17
18    and |g(111)| = 1/d_111
19    '''
20    h,k,l = (1,1,1)
21    d = 1./np.linalg.norm(h*b1 + k*b2 + l*b3)
22
23    print 'd_111 spacing = %1.3f Angstroms' % d
24
25    #method #2
26    hkl = np.array([h,k,l])
27    G = np.array([b1,b2,b3]) #reciprocal unit cell
28
29    '''
30    Gstar is usually defined as this matrix of dot products:
31
32    Gstar = np.array([[dot(b1,b1), dot(b1,b2), dot(b1,b3)],
33                      [dot(b1,b2), dot(b2,b2), dot(b2,b3)],
34                      [dot(b1,b3), dot(b2,b3), dot(b3,b3)]])
35
36    but I prefer the notationally more compact:
37    Gstar = G .dot. transpose(G)
38
39    then, 1/d_hkl^2 = hkl .dot. Gstar .dot. hkl
40    '''
41
42    Gstar = np.dot(G,G.T)
43
44    id2 = np.dot(hkl,np.dot(Gstar,hkl))
45
46    print np.sqrt(1/id2)
47
48    # http://books.google.com/books?id=nJHSqEseuIUC&lpg=PA118&ots=YA9TBldoVH&dq=reciprocal%20metric%20tensor&pg=PA119#v=
49
50
51    '''Finally, many text books on crystallography use long algebraic
52    formulas for computing the d-spacing with sin and cos, vector lengths,
53    and angles. Below we compute these and use them in the general
54    triclinic structure formula which applies to all the structures.
55    '''
56    from Scientific.Geometry import Vector
57    import math
58
59    unitcell = ag.get_cell()
60    A = Vector(unitcell[0])
61    B = Vector(unitcell[1])
62    C = Vector(unitcell[2])
63
64    # lengths of the vectors
```

```
65    a = A.length()#*angstroms2bohr
66    b = B.length()#*angstroms2bohr
67    c = C.length()#*angstroms2bohr
68
69    # angles between the vectors in radians
70    alpha = B.angle(C)
71    beta = A.angle(C)
72    gamma = A.angle(B)
73
74    print '%1.3f %1.3f %1.3f %1.3f %1.3f %1.3f\n' % (a,b,c,
75                                                     alpha,beta,gamma)
76
77    h,k,l = (1,1,1)
78
79    from math import sin, cos
80
81    id2 = ((h**2/a**2*sin(alpha)**2
82           + k**2/b**2*sin(beta)**2
83           + l**2/c**2*sin(gamma)**2
84           +2*k*l/b/c*(cos(beta)*cos(gamma)-cos(alpha))
85           +2*h*l/a/c*(cos(alpha)*cos(gamma)-cos(beta))
86           +2*h*k/a/b*(cos(alpha)*cos(beta)-cos(gamma)))
87           /(1-cos(alpha)**2-cos(beta)**2 - cos(gamma)**2
88             +2*cos(alpha)*cos(beta)*cos(gamma)))
89
90    d = 1/math.sqrt(id2)
91
92    print d
```

```
2.30940107676
```

8.2.5   Center of mass

The center of mass is a convenient origin for many rotations, and moment of inertia calculations. The center of mass is computed from:

$$COM = \frac{\sum m_i \cdot r_i}{\sum m_i}$$

There is a convenience function for this: `:func:'ase.atoms.get_center_of_mass'`

8.2.6   Moments of inertia

Moments of inertia are used in the rotational partition function in statistical mechanics. There is a convenience function for this: `:func:'ase.atoms.get_moments_of_inertia'`. Here are examples of using the function to calculate the moments of inertia for several types of molecules.

```
1    from ase import *
2    from ase.structure import molecule
3    from ase.units import *
```

```
 4
 5    print 'linear rotors: I = [Ia Ia 0]'
 6    mlc = 'CO2'
 7    atoms = molecule(mlc)
 8
 9    print '  %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
10    print
11
12    print 'symmetric rotors (IA = Ib) < Ic'
13    mlc = 'NH3'
14    atoms = molecule(mlc)
15    print '  %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
16
17    mlc = 'C6H6'
18    atoms = molecule(mlc)
19    print '  %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
20    print
21
22    print 'spherical rotors Ia = Ib = Ic'
23    mlc = 'CH4'
24    atoms = molecule(mlc)
25    print '  %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
26    print
```

8.3 Equations of State

The module :mod:'ase.util.eos' uses a simple polynomial equation of
state to find bulk unit cell equilibrium volumes and bulk modulus. There are
several other choices you could use that are more standard in the literature.
Here we summarize them and provide references to the relevant literature.

8.3.1 Birch-Murnaghan

This is probably the most common equation of state used most often, and
is a modification of the original Murnaghan EOS described below. A cur-
rent description of the equation is in reference [10]. You can also find the
equations for the Vinet and Pourier-Tarantola equations of state in that
reference.

Birch-Murnaghan EOS:

$E(\eta) = E_0 + \frac{9B_0V_0}{16}(\eta^2 - 1)^2(6 + B_0'(\eta^2 - 1) - 4\eta^2)$

where $\eta = (V/V_0)^{1/3}$, B_0 and B_0' are the bulk modulus and its pressure
derivative at the equilibrium volume V_0. You may find other derivations of
this equation in the literature too.

Two other equations of state in that reference are the Vinet EOS:

$E(\eta) = E_0 + \frac{2B_0V_0}{(B_0'-1)^2}(2 - (5 + 3B_0'(\eta - 1)e^{-3(B_0'-1)(\eta-1)/2})$

and the Poirier-Tarantola EOS:

$$E(\varrho) = E_0 + \frac{B_0 V_0 \varrho^2}{6}(3 + \varrho(B_0' - 2))$$
with $\varrho = -3\ln(\eta)$.

8.3.2 Murnaghan

The equation most often used in the Murnaghan [26] equation of state is described in [8].

$$E = E_T + \frac{B_0 V}{B_0'}\left[\frac{(V_0/V)^{B_0'}}{B_0'-1} + 1\right] - \frac{V_0 B_0}{B_0'-1}$$

where V is the volume, B_0 and B_0' are the bulk modulus and its pressure derivative at the equilibrium volume V_0. All of these are parameters that are fitted to energy vs. unit cell volume (V) data. When fitting data to this equation a guess of 2-4 for B_0' is usually a good start.

8.3.3 Birch

The original Birch equation [1] is:

$$E = E_0 + \frac{9}{8}B_0 V_0\left(\left(\frac{V_0}{V}\right)^{\frac{2}{3}} - 1\right)^2 + \frac{9}{16}B_0 V_0(B_0' - 4)\left(\left(\frac{V}{V_0}\right)^{2/3} - 1\right)^3$$

8.3.4 The Anton-Schmidt Equation of state [23]

$$E(V) = E_\infty + \frac{BV_0}{n+1}\left(\frac{V}{V_0}\right)^{n+1}\left(\ln\frac{V}{V_0} - \frac{1}{n+1}\right)$$

where :math:E_∞ corresponds to the energy at infinite separation, although the model they use to derive this equation breaks down at large separations so this is usually not a good estimate of the cohesive energy. n is in the range of -2.

8.3.5 Fitting data to these equations of state

To use these equations of state to find the equilibrium cell volume and bulk modulus we need a set of calculations that give us the energy of the unit cell as a function of the cell volume. We then fit that data to one of the above equations to extract the parameters we want. All of these equations of state are non-linear in the cell volume, which means you have to provide some initial guesses for the parameters.

Here we describe a strategy for getting some estimates of the parameters using a linear least squares fitting of a parabola to the data to estimate E_0, V_0, B and B_0' which are used as initial guess for a non-linear least squares fit of the equation of state to the data.

The following example illustrates one approach to this problem for the Murnaghan equation of state:

```python
#!/usr/bin/env python

'''Example of fitting the Birch-Murnaghan EOS to data'''

from pylab import * #this includes numpy as np!
from scipy.optimize import leastsq

# raw data from 2.2.3-al-analyze-eos.py
v = np.array([13.72, 14.83, 16.0, 17.23, 18.52])
e = np.array([-56.29, -56.41, -56.46, -56.46, -56.42])

#make a vector to evaluate fits on with a lot of points so it looks smooth
vfit = np.linspace(min(v),max(v),100)

### fit a parabola to the data
# y = ax^2 + bx + c
a,b,c = polyfit(v,e,2) #this is from pylab

'''
the parabola does not fit the data very well, but we can use it to get
some analytical guesses for other parameters.

V0 = minimum energy volume, or where dE/dV=0
E = aV^2 + bV + c
dE/dV = 2aV + b = 0
V0 = -b/2a

E0 is the minimum energy, which is:
E0 = aV0^2 + bV0 + c

B is equal to V0*d^2E/dV^2, which is just 2a*V0

and from experience we know Bprime_0 is usually a small number like 4
'''

#now here are our initial guesses.
v0 = -b/(2*a)
e0 = a*v0**2 + b*v0 + c
b0 = 2*a*v0
bP = 4

#now we have to create the equation of state function
def Murnaghan(parameters,vol):
    '''
    given a vector of parameters and volumes, return a vector of energies.
    equation From PRB 28,5480 (1983)
    '''
    E0 = parameters[0]
    B0 = parameters[1]
    BP = parameters[2]
    V0 = parameters[3]

```

```
53        E = E0 + B0*vol/BP*(((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
54
55        return E
56
57    # and we define an objective function that will be minimized
58    def objective(pars,y,x):
59        #we will minimize this function
60        err =  y - Murnaghan(pars,x)
61        return err
62
63    x0 = [e0, b0, bP, v0] #initial guesses in the same order used in the Murnaghan function
64
65    murnpars, ier = leastsq(objective, x0, args=(e,v)) #this is from scipy
66
67    #now we make a figure summarizing the results
68    plot(v,e,'ro')
69    plot(vfit, a*vfit**2 + b*vfit + c,'--',label='parabolic fit')
70    plot(vfit, Murnaghan(murnpars,vfit), label='Murnaghan fit')
71    xlabel('Volume ($\AA^3$)')
72    ylabel('Energy (eV)')
73    legend(loc='best')
74
75    #add some text to the figure in figure coordinates
76    ax = gca()
77    text(0.4,0.5,'Min volume = %1.2f $\AA^3$' % murnpars[3],
78         transform = ax.transAxes)
79    text(0.4,0.4,'Bulk modulus = %1.2f eV/$\AA^3$ = %1.2f GPa' % (murnpars[1],
80                                                            murnpars[1]*160.21773)
81         , transform = ax.transAxes)
82    savefig('images/a-eos.png')
83    show()
84
85    print 'initial guesses  : ',x0
86    print 'fitted parameters: ', murnpars
```
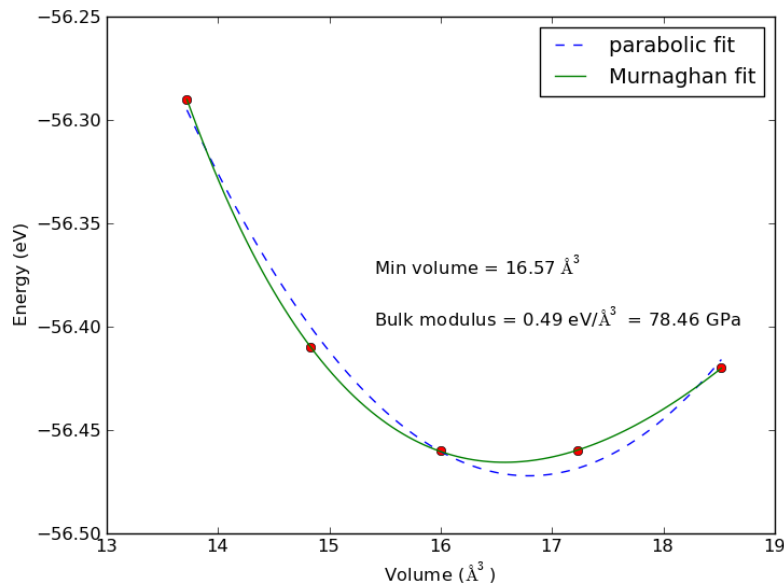
You can see the Murnaghan equation of state fits the data better than the parabola.

Here is a comparison of the initial guesses and final parameters. You can see our guesses from the parabola were actually pretty good, and are the main reason we converged to a solution. If you try other guesses you will probably find the 'scipy.optimize.leastsq' function does not converge.

8.4 Miscellaneous jasp/Vasp tips

8.4.1 Using a special setup

Vasp provides special setups for some elements. The following guidelines tell you what is in a potential:

No extension means the standard potential $_h$ means the potential is harder than the standard (i.e. needs a higher cutoff energy) $_s$ means the potential is softer than the standard (i.e. needs a lower cutoff energy)

$_{sv}$ s and p semi-core states are treated as valence states $_{pv}$ p semi-core states are treated as valence states $_d$ d semi-core states are treated as valence states

1st row elements Alkali and alkali-earth metals d-elements p-elements f-elements

Here we show how to select the O_{sv} potential in a calculation.

137

```python
1   from ase import Atoms, Atom
2   from jasp import *
3
4   JASPRC['mode']='run'
5
6   atoms = Atoms([Atom('O',[5,5,5],magmom=1)],
7                 cell=(6,6,6))
8
9   with jasp('molecules/O_sv',
10            encut=300,
11            xc='PBE',
12            ispin=2,
13            ismear=0,
14            sigma=0.001,
15            txt='O_test.txt',
16            setups={'O':'_sv'}, # specifies O_sv potential
17            atoms=atoms) as calc:
18
19      print atoms.get_potential_energy()
```

How do you know you got the right one? We can look at the first line of the POTCAR file in the calculation directory to see.

```
1   head -n 1 molecules/O_sv/POTCAR
```

9   Python

9.1   TODO easy$_{install}$ as a user

This usually works if that directory is on your PYTHONPATH
    easy$_{install}$ -d ~/lib/python2.6/site-packages/ pymatgen

10   References

References

[1] *Intermetallic compounds: Principles and Practice*, volume Vol I: Principles, pages 195–210.

[2] Lennart Bengtsson. Dipole correction for surface supercell calculations. *Phys. Rev. B*, 59:12301–12304, May 1999. URL: http://link.aps.org/doi/10.1103/PhysRevB.59.12301, doi:10.1103/PhysRevB.59.12301.

[3] P. E. Blöchl. Projector augmented-wave method. *Phys. Rev. B*, 50:17953–17979, Dec 1994. URL: http://link.aps.org/doi/10.1103/PhysRevB.50.17953, doi:10.1103/PhysRevB.50.17953.

[4] J. C. Boettger. Nonconvergence of surface energies obtained from thin-film calculations. *Phys. Rev. B*, 49:16798–16800, Jun 1994. URL: http://link.aps.org/doi/10.1103/PhysRevB.49.16798, doi:10.1103/PhysRevB.49.16798.

[5] J C Boettger, John R Smith, Uwe Birkenheuer, Notker Rösch, S B Trickey, John R Sabin, and S Peter Apell. Extracting convergent surface formation energies from slab calculations. *Journal of Physics: Condensed Matter*, 10(4):893, 1998. URL: http://stacks.iop.org/0953-8984/10/i=4/a=017.

[6] Jr. E. Bright Wilson, J.C. Decius, and Paul C. Cross. *Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra*. Dover Publications, 1955.

[7] Carlos Fiolhais, L.M. Almeida, and C. Henriques. Extraction of aluminium surface energies from slab calculations: perturbative and non-perturbative approaches. *Progress in Surface Science*, 74(1–8):209 – 217, 2003. ¡ce:title¿Maria Steslicka Memorial Volume¡/ce:title¿. URL: http://www.sciencedirect.com/science/article/pii/S0079681603000777, doi:10.1016/j.progsurf.2003.08.017.

[8] C. L. Fu and K. M. Ho. First-principles calculation of the equilibrium ground-state properties of transition metals: Applications to nb and mo. *Phys. Rev. B*, 28:5480–5486, Nov 1983. URL: http://link.aps.org/doi/10.1103/PhysRevB.28.5480, doi:10.1103/PhysRevB.28.5480.

[9] Jeong Woo Han, Liwei Li, and David S. Sholl. Density functional theory study of h and co adsorption on alkali-promoted mo2c surfaces. *The Journal of Physical Chemistry C*, 115(14):6870–6876, 2011. URL: http://pubs.acs.org/doi/abs/10.1021/jp200950a, arXiv:http://pubs.acs.org/doi/pdf/10.1021/jp200950a, doi:10.1021/jp200950a.

[10] M. Hebbache and M. Zemzemi. *Ab initio* study of high-pressure behavior of a low compressibility metal and a hard material: osmium and diamond. *Phys. Rev. B*, 70:224107, Dec 2004. URL:

http://link.aps.org/doi/10.1103/PhysRevB.70.224107, doi:10.1103/PhysRevB.70.224107.

[11] Graeme Henkelman, Andri Arnaldsson, and Hannes Jónsson. A fast and robust algorithm for bader decomposition of charge density. *Computational Materials Science*, 36(3):354 – 360, 2006. URL: http://www.sciencedirect.com/science/article/pii/S0927025605001849, doi:10.1016/j.commatsci.2005.04.010.

[12] Roald Hoffmann. A chemical and theoretical way to look at bonding on surfaces. *Rev. Mod. Phys.*, 60:601–628, Jul 1988. URL: http://link.aps.org/doi/10.1103/RevModPhys.60.601, doi:10.1103/RevModPhys.60.601.

[13] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964. URL: http://link.aps.org/doi/10.1103/PhysRev.136.B864, doi:10.1103/PhysRev.136.B864.

[14] Charles Kittel. *Introduction to Solid State Physics*. Wiley, 8th edition, 2005.

[15] W. Kohn. Nobel lecture: Electronic structure of matter¯wave functions and density functionals. *Rev. Mod. Phys.*, 71:1253–1266, Oct 1999. URL: http://link.aps.org/doi/10.1103/RevModPhys.71.1253, doi:10.1103/RevModPhys.71.1253.

[16] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965. URL: http://link.aps.org/doi/10.1103/PhysRev.140.A1133, doi:10.1103/PhysRev.140.A1133.

[17] G. Kresse and J. Furthmuller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Computational Materials Science*, 6(1):15 – 50, 1996. URL: http://www.sciencedirect.com/science/article/pii/0927025696000080, doi:10.1016/0927-0256(96)00008-0.

[18] G. Kresse and J. Furthmuller. Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996. URL: http://link.aps.org/doi/10.1103/PhysRevB.54.11169, doi:10.1103/PhysRevB.54.11169.

[19] G. Kresse and J. Hafner. *Ab initio* molecular dynamics for liquid metals. *Phys. Rev. B*, 47:558–561, Jan 1993. URL: http://link.aps.org/doi/10.1103/PhysRevB.47.558, doi:10.1103/PhysRevB.47.558.

[20] G. Kresse and J. Hafner. *Ab initio* molecular-dynamics simulation of the liquid-metal˘amorphous-semiconductor transition in germanium. *Phys. Rev. B*, 49:14251–14269, May 1994. URL: http://link.aps.org/doi/10.1103/PhysRevB.49.14251, doi:10.1103/PhysRevB.49.14251.

[21] G. Kresse and D. Joubert. From ultrasoft pseudopotentials to the projector augmented-wave method. *Phys. Rev. B*, 59:1758–1775, Jan 1999. URL: http://link.aps.org/doi/10.1103/PhysRevB.59.1758, doi:10.1103/PhysRevB.59.1758.

[22] Thomas A. Manz and David S. Sholl. Chemically meaningful atomic charges that reproduce the electrostatic potential in periodic and nonperiodic materials. *Journal of Chemical Theory and Computation*, 6(8):2455–2468, 2010. URL: http://pubs.acs.org/doi/abs/10.1021/ct100125x, arXiv:http://pubs.acs.org/doi/pdf/10.1021/ct100125x, doi:10.1021/ct100125x.

[23] B. Mayer, H. Anton, E. Bott, M. Methfessel, J. Sticht, J. Harris, and P.C. Schmidt. Ab-initio calculation of the elastic constants and thermal expansion coefficients of laves phases. *Intermetallics*, 11(1):23 – 32, 2003. URL: http://www.sciencedirect.com/science/article/pii/S0966979502001279, doi:10.1016/S0966-9795(02)00127-9.

[24] Hendrik J. Monkhorst and James D. Pack. Special points for brillouin-zone integrations. *Phys. Rev. B*, 13:5188–5192, Jun 1976. URL: http://link.aps.org/doi/10.1103/PhysRevB.13.5188, doi:10.1103/PhysRevB.13.5188.

[25] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71:035109, Jan 2005. URL: http://link.aps.org/doi/10.1103/PhysRevB.71.035109, doi:10.1103/PhysRevB.71.035109.

[26] F. D. Murnaghan. The compressibility of media under extreme pressures. *Proceedings of the National Academy of Sciences of the United States of America*, 30(9):pp. 244–247, 1944. URL: http://www.jstor.org/stable/87468.

[27] Jörg Neugebauer and Matthias Scheffler. Adsorbate-substrate and adsorbate-adsorbate interactions of na and k adlayers on al(111). *Phys. Rev. B*, 46:16067–16080, Dec 1992. URL: http://link.aps.org/doi/10.1103/PhysRevB.46.16067, doi:10.1103/PhysRevB.46.16067.

[28] John P. Perdew, Kieron Burke, and Matthias Ernzerhof. Generalized gradient approximation made simple. *Phys. Rev. Lett.*, 77:3865–3868, Oct 1996. URL: http://link.aps.org/doi/10.1103/PhysRevLett.77.3865, doi:10.1103/PhysRevLett.77.3865.

[29] John A. Pople. Nobel lecture: Quantum chemical models. *Rev. Mod. Phys.*, 71:1267–1274, Oct 1999. URL: http://link.aps.org/doi/10.1103/RevModPhys.71.1267, doi:10.1103/RevModPhys.71.1267.

[30] Fabien Tran, Robert Laskowski, Peter Blaha, and Karlheinz Schwarz. Performance on molecules, surfaces, and solids of the wu-cohen gga exchange-correlation energy functional. *Phys. Rev. B*, 75:115131, Mar 2007. URL: http://link.aps.org/doi/10.1103/PhysRevB.75.115131, doi:10.1103/PhysRevB.75.115131.

[31] A. van de Walle, M. Asta, and G. Ceder. The alloy theoretic automated toolkit: A user guide. *Calphad*, 26(4):539 – 553, 2002. URL: http://www.sciencedirect.com/science/article/pii/S0364591602800062, doi:10.1016/S0364-5916(02)80006-2.

[32] Axel van de Walle. Multicomponent multisublattice alloys, non-configurational entropy and other additions to the alloy theoretic automated toolkit. *Calphad*, 33(2):266 – 278, 2009. ¡ce:title¿Tools for Computational Thermodynamics¡/ce:title¿. URL: http://www.sciencedirect.com/science/article/pii/S0364591608001314, doi:10.1016/j.calphad.2008.12.005.

[33] Nilay İnoğlu and John R. Kitchin. Atomistic thermodynamics study of the adsorption and the effects of water–gas shift reactants on cu catalysts under reaction conditions. *Journal of Catalysis*, 261(2):188 – 194, 2009. URL: http://www.sciencedirect.com/science/article/pii/S0021951708004314, doi:10.1016/j.jcat.2008.11.020.

11 Index

Index