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.

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:

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

with this code:

```python
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

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.

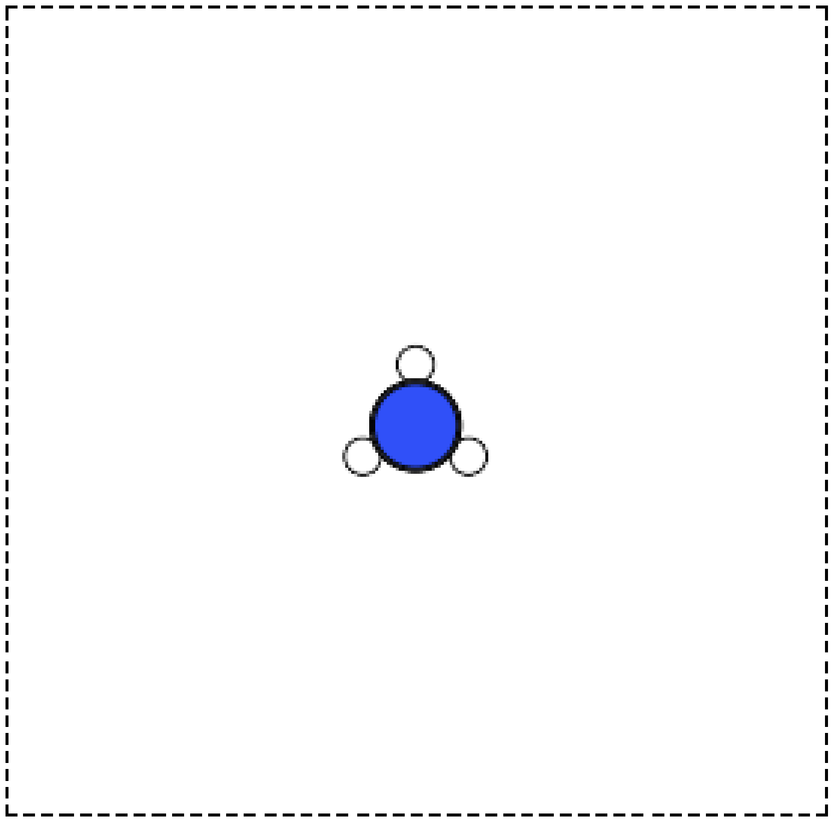
All numbers are experimental values, except for coordinates, which are MP2(full)/6-31G(d) optimized geometries. Here is a list of all the species available in `mod:ase.data.g2`

```python
from ase.data import g2
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.

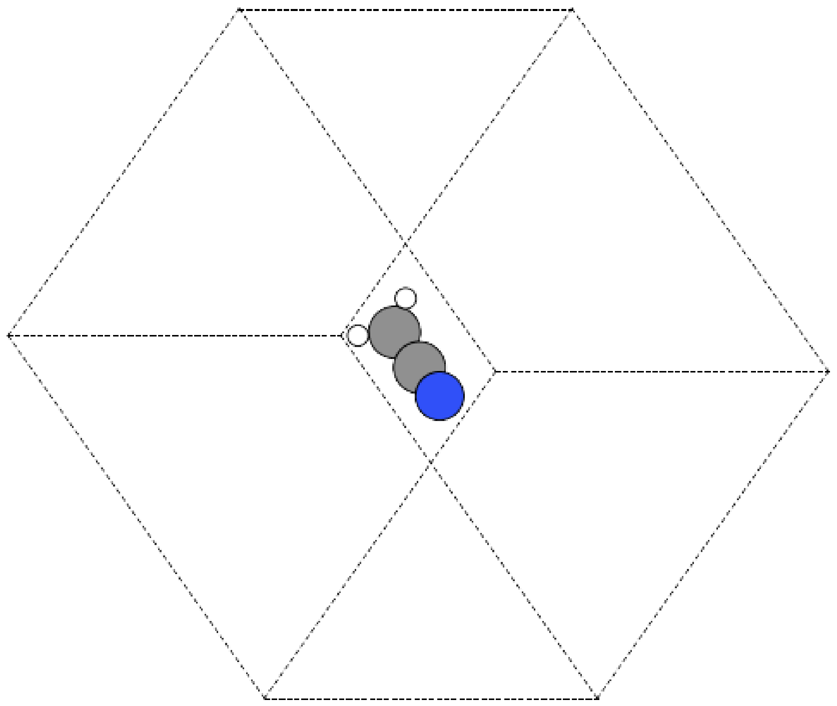
Here is an example of getting the geometry of an acetonitrile molecule and writing an image to a file. Note that the default unit cell is a $1\AA\times1\AA\times1\AA$ cubic cell. That is too small to use if your calculator uses periodic boundary conditions. We center the atoms in the unit cell and add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

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

c60 = molecule('CH3CN')

c60.center(vacuum=6)
print c60.get_cell()

write('molecules/ch3cn.png', c60, show_unit_cell=2)
```



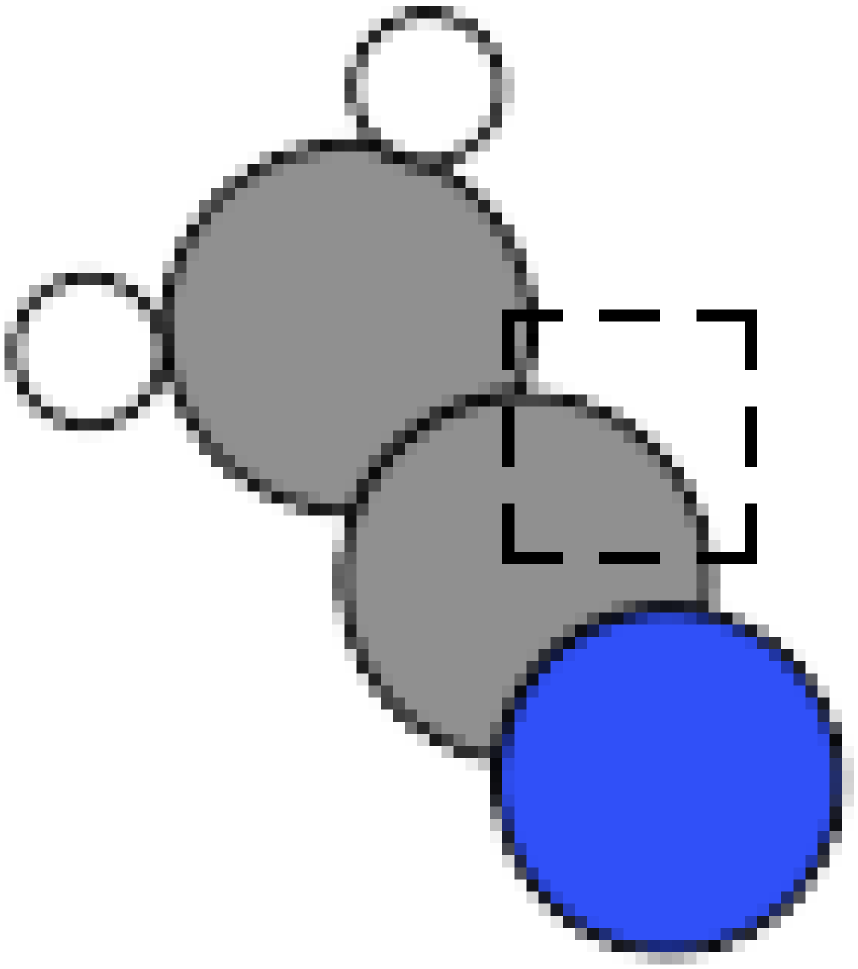
It is possible to rotate the atoms with `func:ase.io.write` if you wanted to see pictures from another angle. In the next example we rotate 45 degrees about the x-axis, then 45 degrees about the y-axis. Note that this only affects the image, not the actual coordinates

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

atoms = molecule('CH3CN')

atoms.center(vacuum=6)
print atoms.get_cell()

write('molecules/ch3cn-rotated.png', atoms,
      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.

```
from ase import Atoms
from ase.data.molecules import molecule
from ase.io import write
from numpy import pi

# ammonia
atoms = molecule('CH3CN')
p1 = atoms.get_positions()

atoms.rotate('x', pi/4, center='COM', rotate_cell=False)
atoms.rotate('y', pi/4, center='COM', rotate_cell=False)

write('molecules/ch3cn-rotated-2.png',atoms,show_unit_cell=2)
p2 = atoms.get_positions()
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.

```
from ase.io.xyz import *
from ase.io import write
```

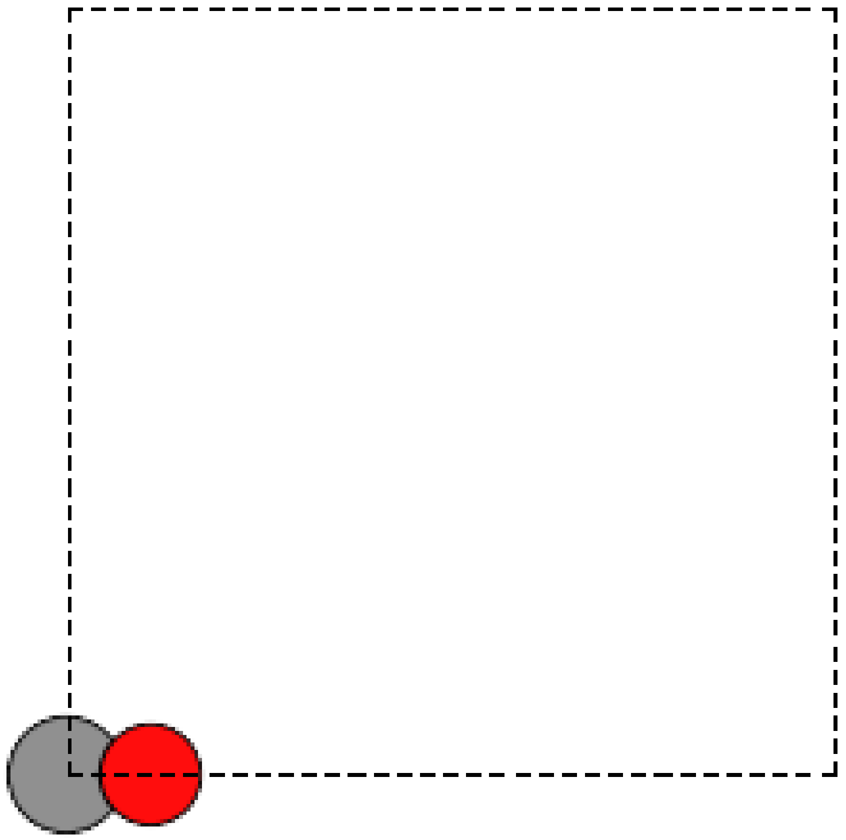
```python
import numpy as np

atoms = read_xyz('molecules/isobutane.xyz')
atoms.center(vacuum=5)
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.

```python
from ase import Atoms, Atom
from ase.io import write

#method 1 - simple cubic unit cell
atoms = Atoms([Atom('C',[0., 0.,0.]),
               Atom('O',[1.1,0.,0.])],
              cell=(10,10,10))

print 'V = %1.0f Ang^3' % atoms.get_volume()

write('molecules/simple-cubic-cell.png',atoms,show_unit_cell=2)
```



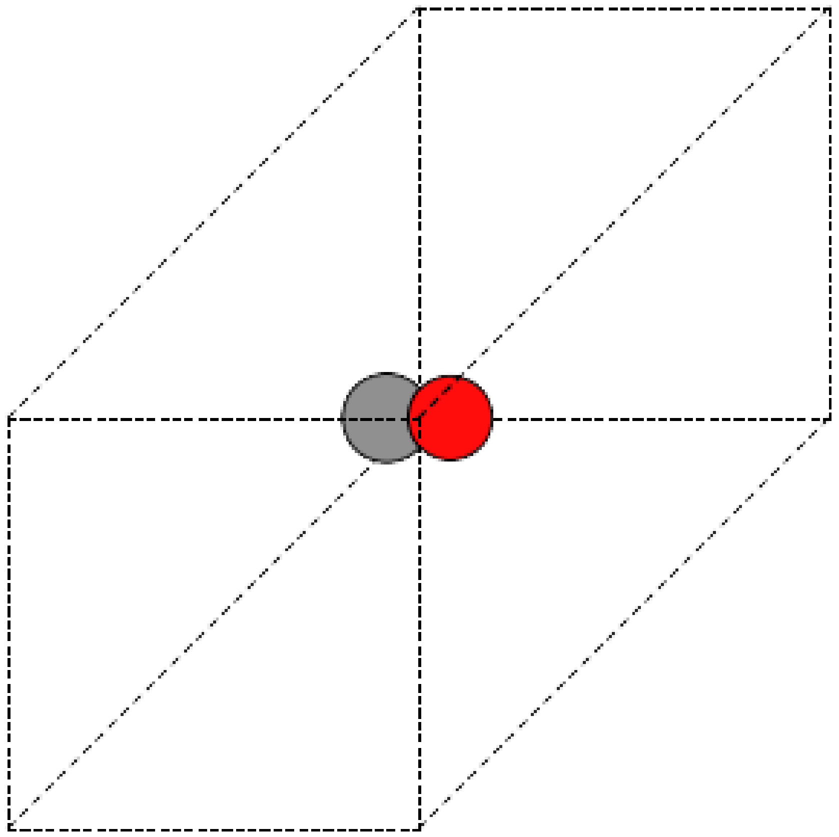
There are two inconvenient features of the simple cubic cell:

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 $\AA^3$.

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 $\AA^3$, which is smaller than the cube. This will result in less computational time to compute properties.

```python
from ase import Atoms, Atom
from ase.io import write

b = 7.1
atoms2 = Atoms([Atom('C',[0., 0.,0.]),
                Atom('O',[1.1,0.,0.])],
               cell=[[b, b, 0.],
                     [b, 0., b],
                     [0., b, b]])
print 'V = %1.1f Ang^3' % atoms2.get_volume()

atoms2.center() #translate atoms to center of unit cell
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.

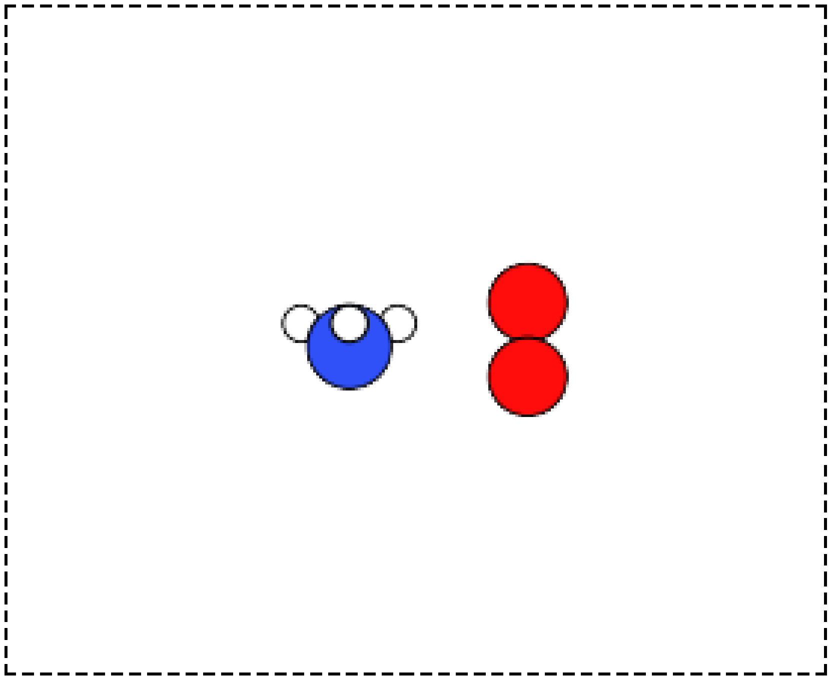
```
from ase import Atoms, Atom
import numpy as np

b = 7.1
atoms2 = Atoms([Atom('C',[0., 0.,0.]),
                Atom('O',[1.1,0.,0.])],
                cell=[[b, b, 0.],
                      [b, 0., b],
                      [0., b, b]])

# get unit cell vectors and their lengths
(a1, a2, a3) = atoms2.get_cell()
print '|a1| = %1.2f Ang' % np.sum(a1**2)**0.5
print '|a2| = %1.2f Ang' % np.sum(a2**2)**0.5
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.

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

atoms1 = molecule('NH3')

atoms2 = molecule('O2')
atoms2.translate([3,0,0])

bothatoms = atoms1 + atoms2
bothatoms.center(5)

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`.

```
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()
```

```
pos = atoms.get_positions()
num = atoms.get_atomic_numbers()

atom_indices = range(len(atoms))

print
print '  # sym   at#     p_x     p_y     p_z'
print '-------------------------------------'
for i,s,n,p in zip(atom_indices,sym,num,pos):
  px,py,pz = p
  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.

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

atoms = molecule('C6H6')
masses = atoms.get_masses()

molecular_weight = sum(masses)

print 'The molecular weight of %s is %f' % (atoms.get_chemical_symbols(reduce=True),
                                            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`.

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

```python
from ase import Atoms
from ase.structure import molecule
from ase.io import write

# ammonia
atoms = molecule('NH3')

print atoms.get_center_of_mass()  # cartesian coordinates

# compute the center of mass by hand
from ase.data import atomic_masses
import numpy as np
pos = atoms.positions
masses = atoms.get_masses()

COM = np.array([0., 0., 0.])
for m,p in zip(masses, pos):
    COM += m*p
COM /= sum(masses)

print COM

# one-line linear algebra definition of COM
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

```
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]
```

```
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

3.3.5 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.

```python
from ase import Atoms
from ase.structure import molecule
from ase.io import write

# ammonia
atoms = molecule('NH3')

for i, atom in enumerate(atoms):
    print '%2i %3s' % (i,atom.symbol)

# N-H bond length
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.

```python
from ase import Atoms
from ase.structure import molecule
from ase.io import write

# ammonia
atoms = molecule('NH3')

for i, atom in enumerate(atoms):
    print '%2i %3s' % (i,atom.symbol)

a = atoms.positions[0] - atoms.positions[1]
b = atoms.positions[0] - atoms.positions[2]

from numpy import arccos, dot, pi
from numpy.linalg import norm

theta_rad = arccos(dot(a,b)/(norm(a)*norm(b))) # in radians

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.

```python
from ase import Atoms
from ase.data.molecules import molecule
from numpy import pi
# ammonia
atoms = molecule('NH3')

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.

```python
from ase import Atoms
from ase.structure import molecule
from ase.visualize import view
import numpy as np

atoms = molecule('C2H6')

for i, atom in enumerate(atoms):
    print '%2i %3s' % (i,atom.symbol)

da = atoms.get_dihedral([5,1,0,4])*180./np.pi
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
```

3.3.6 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.

```
#!/usr/bin/env python
import torque
from ase import Atoms, Atom
from ase.calculators.jacapo import Jacapo
import numpy as np

# this makes array printing a little more readable
np.set_printoptions(precision=3,suppress=True)

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

calc = Jacapo('molecules/co.nc',   #output filename
              nbands=6,
              pw=350,
              ft=0.01,
              atoms=co)

print 'Total energy = ',co.get_potential_energy()
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).

```
#!/usr/bin/env python
from ase import Atoms,Atom
from jasp import *
import numpy as np
np.set_printoptions(precision=3,suppress=True)

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

with jasp('molecules/simple-co', #output dir
          xc='PBE',
          nbands=6,
          encut=350,
          ismear=1,
          sigma=0.01,
          atoms=co):

    print co.get_potential_energy()
    print co.get_forces()
```

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

3.3.7 TODO The density of states

3.3.8 TODO Visualizing electron density

3.3.9 TODO Visualizing Kohn-Sham orbitals

3.3.10 TODO Bader analysis

3.3.11 TODO Dipole moments

3.3.12 TODO Atom-projected density of states

3.3.13 TODO Multicenter density of states

3.4 Geometry optimization

3.4.1 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.

```python
#!/usr/bin/env python
from ase import *
from jasp import *
import numpy as np
np.set_printoptions(precision=3,suppress=True)

bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
energies = []

ready = True # flag for when all calculations are done.
for d in bond_lengths: #possible bond lengths

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

    with jasp('molecules/co-{0}'.format(d), #output dir
              xc='PBE',
              nbands=6,
              encut=350,
              ismear=1,
              sigma=0.01,
              atoms=co):
        try:
            e = co.get_potential_energy()
            energies.append(e)
            print 'd = %1.2f ang' % d
            print 'energy = %f eV' % e
            print 'forces = (eV/ang)\n', co.get_forces()
            print '' #blank line
        except (VaspSubmitted, VaspQueued):
            ready = False
            pass

if ready:
    import matplotlib.pyplot as plt
    plt.plot(bond_lengths, energies, 'bo-')
    plt.xlabel('Bond length ($\AA$)')
    plt.ylabel('Total energy (eV)')
    plt.savefig('molecules/co-bondlengths.png')
    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
```

3.5 Vibrational frequencies

Vasp has built-in capability for performing vibrational calculations.

```python
#adapted from http://cms.mpi.univie.ac.at/wiki/index.php/H2O_vibration
from ase import Atoms, Atom
from jasp import *

atoms = Atoms([Atom('H',[0.5960812,  -0.7677068,   0.0000000]),
               Atom('O',[0.0000000,   0.0000000,   0.0000000]),
               Atom('H',[0.5960812,   0.7677068,   0.0000000])],
               cell=(8,8,8))

with jasp('molecules/h2o_vib',
          xc='PBE',
          encut=400,
          ismear=0,# Gaussian smearing
          ibrion=6,# finite differences with symmetry
          nfree=2, # central differences (default)
          potim=0.015,# default as well
          ediff=1e-8,
          nsw=1,
          atoms=atoms) as calc:

    print atoms.get_forces()
    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$_{\text{vibrationalfrequencies}}$() is only available in jasp (7/12/2012).

3.6 Thermochemical properties of molecules

ase provides a thermochemistry module.

```python
from ase.structure import molecule
from ase.thermochemistry import IdealGasThermo
from jasp import *

atoms = molecule('N2')
atoms.set_cell((10,10,10), scale_atoms=False)

# first we relax a molecule
with jasp('molecules/n2-relax',
          xc='PBE',
          encut=300,
          ibrion=2,
          nsw=5,
          atoms=atoms) as calc:

    electronicenergy = atoms.get_potential_energy()

# next, we get vibrational modes
with jasp('molecules/n2-vib',
          xc='PBE',
          encut=300,
          ibrion=6,
          nfree=2,
          potim=0.15,
          nsw=1,
          atoms=atoms) as calc:
    calc.calculate()
    vib_freq = calc.get_vibrational_frequencies() # in cm^1

    #convert wavenumbers to energy
    h = 4.1356675e-15 # eV*s
    c = 3.0e10 #cm/s
    vib_energies = [h*c*nu for nu in vib_freq]
    print vib_energies

# # now we can get some properties. Note we only need one vibrational
# energy since there is only one mode. This example does not work if
# you give all the energies because one energy is zero.
thermo = IdealGasThermo(vib_energies=vib_energies[0:0],
                        electronicenergy=electronicenergy, atoms=atoms,
                        geometry='linear', symmetrynumber=2, spin=0)

# temperature in K, pressure in Pa, G in eV
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
=======================
```

3.7 Molecular reaction energies

3.7.1 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

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

atoms = Atoms([Atom('O',[5,5,5])],
              cell=(10,10,10))
```

```

with jasp('molecules/O',
          xc='PBE',
          encut=400,
          ismear=0,
          atoms=atoms) as calc:
    try:
        E_O = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O = None

# now relaxed O2 dimer

atoms = Atoms([Atom('O',[5,5,5]),
               Atom('O',[6.22, 5,5])],
              cell=(10,10,10))

with jasp('molecules/O2',
          xc='PBE',
          encut=400,
          ismear=0,
          ibrion=2, # make sure we relax the geometry
          nsw=10,
          atoms=atoms) as calc:
    try:
        E_O2 = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O2 = None

if None not in (E_O, E_O2):
    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.

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

```

atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
              cell=(10,10,10))

with jasp('molecules/O-sp-triplet',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,
          atoms=atoms) as calc:
    try:
        E_O = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O = None

print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

# now relaxed O2 dimer
atoms = Atoms([Atom('O',[5,5,5],magmom=1),
               Atom('O',[6.22, 5,5],magmom=1)],
              cell=(10,10,10))

with jasp('molecules/O2-sp-triplet',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,  # turn spin-polarization on
          ibrion=2, # make sure we relax the geometry
          nsw=10,
          atoms=atoms) as calc:
    try:
        E_O2 = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O2 = None

# verify magnetic moment
print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

if None not in (E_O, E_O2):
    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.

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.

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

atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
              cell=(8,9,10))

with jasp('molecules/O-sp-triplet-lowsym',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,
          atoms=atoms) as calc:
    try:
        E_O = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O = None

print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

# now relaxed O2 dimer
atoms = Atoms([Atom('O',[5,5,5],magmom=1),
               Atom('O',[6.22, 5,5],magmom=1)],
              cell=(10,10,10))

with jasp('molecules/O2-sp-triplet',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,  # turn spin-polarization on
          ibrion=2, # make sure we relax the geometry
          nsw=10,
          atoms=atoms) as calc:
    try:
        E_O2 = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O2 = None

# verify magnetic moment
print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

if None not in (E_O, E_O2):
    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.

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

atoms = Atoms([Atom('O',[5,5,5],magmom=0)],
              cell=(10,10,10))

with jasp('molecules/O-sp-singlet',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,
          atoms=atoms) as calc:
    try:
        E_O = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O = None

print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

# now relaxed O2 dimer
atoms = Atoms([Atom('O',[5,5,5],magmom=1),
               Atom('O',[6.22, 5,5],magmom=-1)],
              cell=(10,10,10))

with jasp('molecules/O2-sp-singlet',
          xc='PBE',
          encut=400,
          ismear=0,
          ispin=2,  # turn spin-polarization on
          ibrion=2, # make sure we relax the geometry
          nsw=10,
          atoms=atoms) as calc:
    try:
        E_O2 = atoms.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        E_O2 = None

# verify magnetic moment
print atoms.get_magnetic_moment()

if None not in (E_O, E_O2):
    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:

```
from jasp import *

with jasp('molecules/O2-sp-singlet') as calc:
    print calc.get_atoms().get_potential_energy()

with jasp('molecules/O2-sp-triplet') as calc:
    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.

3.7.2 Water gas shift example

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

$CO + H_2O \leftrightharpoons CO_2 + H_2$

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.

```
from ase.data.molecules import molecule
from jasp import *
JASPRC['queue.walltime'] = '1:00:00'

# first we define our molecules. These will automatically be at the coordinates from the G2 database.

CO =  molecule('CO')
CO.set_cell([8,8,8], scale_atoms=False)

H2O = molecule('H2O')
H2O.set_cell([8,8,8], scale_atoms=False)

CO2 =  molecule('CO2')
CO2.set_cell([8,8,8], scale_atoms=False)

H2 = molecule('H2')
H2.set_cell([8,8,8], scale_atoms=False)

# now the calculators to get the energies
with jasp('molecules/wgs/CO',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=2,
          nsw=10,
          atoms=CO) as calc:
    try:
        eCO = CO.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        eCO = None

with jasp('molecules/wgs/CO2',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=2,
          nsw=10,
          atoms=CO2) as calc:
    try:
        eCO2 = CO2.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        eCO2 = None

with jasp('molecules/wgs/H2',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=2,
          nsw=10,
          atoms=H2) as calc:
    try:
        eH2 = H2.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        eH2 = None

with jasp('molecules/wgs/H2O',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=2,
          nsw=10,
          atoms=H2O) as calc:
    try:
        eH2O = H2O.get_potential_energy()
    except (VaspSubmitted, VaspQueued):
        eH2O = None

if None in (eCO2, eH2, eCO, eH2O):
    pass
else:
    dE = eCO2 + eH2 - eCO - eH2O
    print '\Delta E = {0:1.3f} eV'.format(dE)
    print '\Delta E = {0:1.3f} kcal/mol'.format(dE*23.06035)
    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.

```python
from jasp import *

print '**** calculation summaries'
print '***** CO'
with jasp('molecules/wgs/CO') as calc:
    print '#+begin_example'
    print calc
    print '#+end_example'

print '***** CO2'
with jasp('molecules/wgs/CO2') as calc:
    print '#+begin_example'
    print calc
    print '#+end_example'

print '***** H2'
with jasp('molecules/wgs/H2') as calc:
    print '#+begin_example'
    print calc
    print '#+end_example'

print '***** H2O'
with jasp('molecules/wgs/H2O') as calc:
    print '#+begin_example'
    print calc
    print '#+end_example'
```

3.7.3 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

```python
from jasp import *
JASPRC['queue.walltime'] = '1:00:00'

# get relaxed geometry
```

```python
with jasp('molecules/wgs/CO') as calc:
    CO = calc.get_atoms()

# now do the vibrations
with jasp('molecules/wgs/CO-vib',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=6,
          nfree=2,
          potim=0.02,
          nsw=1,
          atoms=CO) as calc:
    calc.calculate()
    vib_freq = calc.get_vibrational_frequencies()
    print vib_freq
```

CO has only one vibrational mode.

CO2 vibrations

```python
from jasp import *
JASPRC['queue.walltime'] = '1:00:00'

# get relaxed geometry
with jasp('molecules/wgs/CO2') as calc:
    CO2 = calc.get_atoms()

# now do the vibrations
with jasp('molecules/wgs/CO2-vib',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=6,
          nfree=2,
          potim=0.02,
          nsw=1,
          atoms=CO2) as calc:
    calc.calculate()
    vib_freq = calc.get_vibrational_frequencies()
    print vib_freq
```

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

H2 vibrations

```python
from jasp import *
JASPRC['queue.walltime'] = '1:00:00'

# get relaxed geometry
```

```
with jasp('molecules/wgs/H2') as calc:
    H2 = calc.get_atoms()

# now do the vibrations
with jasp('molecules/wgs/H2-vib',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=6,
          nfree=2,
          potim=0.02,
          nsw=1,
          atoms=H2) as calc:
    calc.calculate()
    vib_freq = calc.get_vibrational_frequencies()
    print vib_freq
```

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

H2O vibrations

```
from jasp import *
JASPRC['queue.walltime'] = '1:00:00'

# get relaxed geometry
with jasp('molecules/wgs/H2O') as calc:
    H2O = calc.get_atoms()

# now do the vibrations
with jasp('molecules/wgs/H2O-vib',
          xc='PBE',
          encut=350,
          ismear=0,
          ibrion=6,
          nfree=2,
          potim=0.02,
          nsw=1,
          atoms=H2O) as calc:
    calc.calculate()
    vib_freq = calc.get_vibrational_frequencies()
    print vib_freq
```

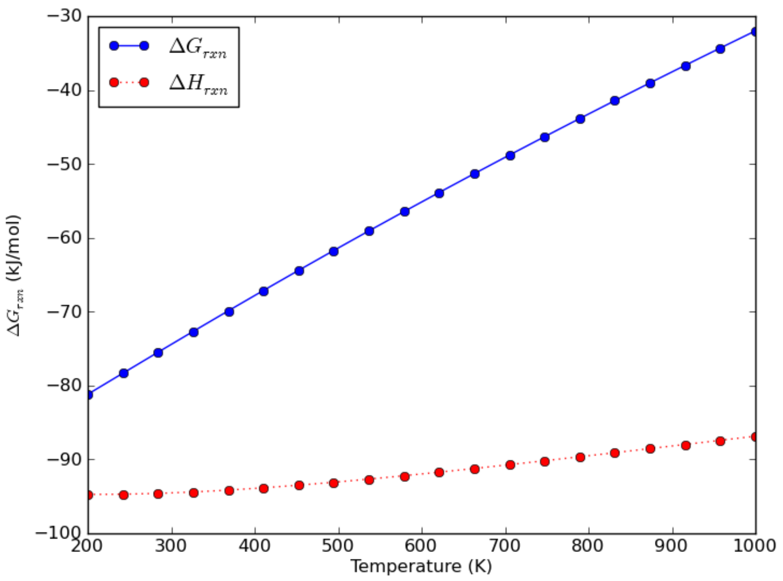
Water has 3N-6 = 3 vibrational modes.

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

```
from ase.thermochemistry import IdealGasThermo
from jasp import *
```

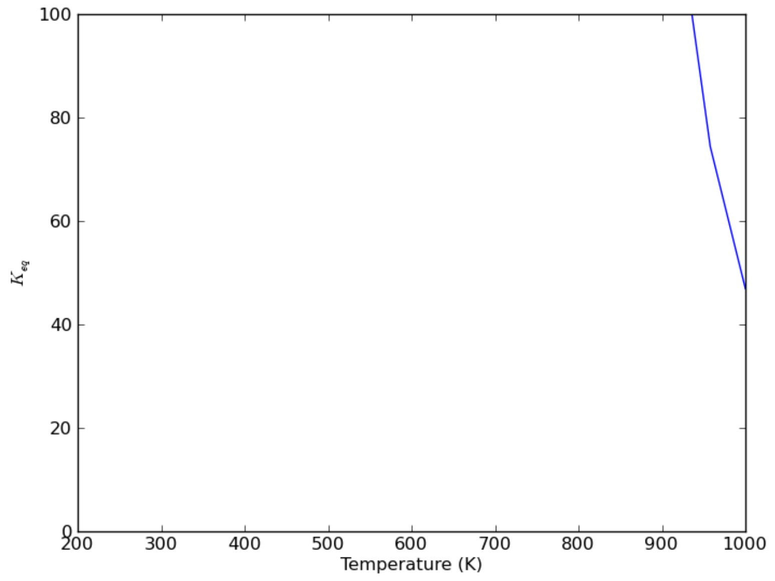
```python
import numpy as np
import matplotlib.pyplot as plt

# first we get the electronic energies
with jasp('molecules/wgs/CO') as calc:
    CO = calc.get_atoms()
    E_CO = CO.get_potential_energy()

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

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

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

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

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

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

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

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

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

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

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

```

```python
H2O_t = IdealGasThermo(vib_energies=H2O_vib_energies[0:3],
                      electronicenergy=E_H2O, atoms=H2O,
                      geometry='nonlinear', symmetrynumber=2,
                      spin=0)

# now we can compute G_rxn for a range of temperatures from 200 to 1000 K
Trange = np.linspace(200,1000,20) #K
P = 101325. # Pa
Grxn = np.array([(CO2_t.get_free_energy(temperature=T, pressure=P)
                  + H2_t.get_free_energy(temperature=T, pressure=P)
                  - H2O_t.get_free_energy(temperature=T, pressure=P)
                  - CO_t.get_free_energy(temperature=T, pressure=P))*96.485 for T in Trange])

Hrxn = np.array([(CO2_t.get_enthalpy(temperature=T)
                  + H2_t.get_enthalpy(temperature=T)
                  - H2O_t.get_enthalpy(temperature=T)
                  - CO_t.get_enthalpy(temperature=T))*96.485 for T in Trange])

plt.plot(Trange, Grxn, 'bo-',label='$\Delta G_{rxn}$')
plt.plot(Trange, Hrxn, 'ro:',label='$\Delta H_{rxn}$')
plt.xlabel('Temperature (K)')
plt.ylabel('$\Delta G_{rxn}$ (kJ/mol)')
plt.legend(loc='best')
plt.savefig('images/wgs-dG-T.png')

plt.figure()
R = 8.314e-3 # gas constant in kJ/mol/K

Keq = np.exp(-Grxn/R/Trange)
plt.plot(Trange, Keq)
plt.ylim([0, 100])
plt.xlabel('Temperature (K)')
plt.ylabel('$K_{eq}$')
plt.savefig('images/wgs-Keq.png')
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.

3.8 TODO Molecular reaction barriers

3.9 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.

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

atoms = Atoms([Atom('H',[0.5960812,  -0.7677068,   0.0000000]),
               Atom('O',[0.0000000,   0.0000000,   0.0000000]),
               Atom('H',[0.5960812,   0.7677068,   0.0000000])],
               cell=(8,8,8))

with jasp('molecules/h2o_relax',
          xc='PBE',
          encut=400,
          ismear=0,# Gaussian smearing
          ibrion=2,
          ediff=1e-8,
          nsw=10,
          atoms=atoms) as calc:

    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.

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

#read in relaxed geometry
with jasp('molecules/h2o_relax') as calc:
    atoms = calc.get_atoms()

# now define a new calculator
with jasp('molecules/h2o_vib_dfpt',
          xc='PBE',
          encut=400,
          ismear=0,# Gaussian smearing
          ibrion=7, # switches on the DFPT vibrational analysis (with no symmetry constraints)
          nfree=2,
          potim=0.015,
          lepsilon=True, # enables to calculate and to print the BEC tensors
          lreal=False,
          nsw=1,
          nwrite=3, # affects OUTCAR verbosity: explicitly forces SQRT(mass)-divided eigenvectors to be printed
          atoms=atoms) as calc:
    calc.calculate(atoms)
```

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

```
#!/bin/bash
# A utility for calculating the vibrational intensities from VASP output (OUTCAR)
# (C) David Karhanek, 2011-03-25, ICIQ Tarragona, Spain (www.iciq.es)

# extract Born effective charges tensors
printf "..reading OUTCAR"
BORN_NROWS=`grep NIONS OUTCAR | awk '{print $12*4+1}'`
```

```
if [ ‘grep ’BORN’ OUTCAR | wc -l‘ = 0 ] ; then \
   printf " .. FAILED! Born effective charges missing! Bye! \n\n" ; exit 1 ; fi
grep "in e, cummulative" -A $BORN_NROWS OUTCAR > born.txt

# extract Eigenvectors and eigenvalues
if [ ‘grep ’SQRT(mass)’ OUTCAR | wc -l‘ != 1 ] ; then \
   printf " .. FAILED! Restart VASP with NWRITE=3! Bye! \n\n" ; exit 1 ; fi
EIG_NVIBS=‘grep -A 2000 ’SQRT(mass)’ OUTCAR | grep ’cm-1’ | wc -l‘
EIG_NIONS=‘grep NIONS OUTCAR | awk ’{print $12}’‘
EIG_NROWS=‘echo "($EIG_NIONS+3)*$EIG_NVIBS+3" | bc‘
grep -A $(($EIG_NROWS+2)) ’SQRT(mass)’ OUTCAR | tail -n $(($EIG_NROWS+1)) | sed ’s/f\/i/fi /g’ > eigenvectors.txt
printf " ..done\n"

# set up a new directory, split files - prepare for parsing
printf "..splitting files"
mkdir intensities ; mv born.txt eigenvectors.txt intensities/
cd intensities/
let NBORN_NROWS=BORN_NROWS-1
let NEIG_NROWS=EIG_NROWS-3
let NBORN_STEP=4
let NEIG_STEP=EIG_NIONS+3
tail -n $NBORN_NROWS born.txt > temp.born.txt
tail -n $NEIG_NROWS eigenvectors.txt > temp.eige.txt
mkdir inputs ; mv born.txt eigenvectors.txt inputs/
split -a 3 -d -l $NEIG_STEP temp.eige.txt temp.ei.
split -a 3 -d -l $NBORN_STEP temp.born.txt temp.bo.
mkdir temps01 ; mv temp.born.txt temp.eige.txt temps01/
for nu in ‘seq 1 $EIG_NVIBS‘ ; do
 let nud=nu-1 ; ei=‘printf "%03u" $nu‘ ; eid=‘printf "%03u" $nud‘ ; mv temp.ei.$eid eigens.vib.$ei
done
for s in ‘seq 1 $EIG_NIONS‘ ; do
 let sd=s-1 ; bo=‘printf "%03u" $s‘ ; bod=‘printf "%03u" $sd‘ ; mv temp.bo.$bod borncs.$bo
done
printf " ..done\n"

# parse deviation vectors (eig)
printf "..parsing eigenvectors"
let sad=$EIG_NIONS+1
for nu in ‘seq 1 $EIG_NVIBS‘ ; do
 nuu=‘printf "%03u" $nu‘
 tail -n $sad eigens.vib.$nuu | head -n $EIG_NIONS | awk ’{print $4,$5,$6}’ > e.vib.$nuu.allions
 split -a 3 -d -l 1 e.vib.$nuu.allions temp.e.vib.$nuu.ion.
 for s in ‘seq 1 $EIG_NIONS‘ ; do
  let sd=s-1; bo=‘printf "%03u" $s‘; bod=‘printf "%03u" $sd‘; mv temp.e.vib.$nuu.ion.$bod e.vib.$nuu.ion.$bo
 done
done
printf " ..done\n"

# parse born effective charge matrices (born)
printf "..parsing eff.charges"
for s in ‘seq 1 $EIG_NIONS‘ ; do
 ss=‘printf "%03u" $s‘
 awk ’{print $2,$3,$4}’ borncs.$ss | tail -3 > bornch.$ss
done
mkdir temps02 ; mv eigens.* borncs.* temps02/
printf " ..done\n"
```

```

# parse matrices, multiply them and collect squares (giving intensities)
printf "..multiplying matrices, summing "
for nu in ‘seq 1 $EIG_NVIBS‘ ; do
 nuu=‘printf "%03u" $nu‘
 int=0.0
 for alpha in 1 2 3 ;  do             # summing over alpha coordinates
  sumpol=0.0
  for s in ‘seq 1 $EIG_NIONS‘ ; do    # summing over atoms
   ss=‘printf "%03u" $s‘
   awk -v a="$alpha" ’(NR==a){print}’ bornch.$ss > z.ion.$ss.alpha.$alpha
   # summing over beta coordinates and multiplying Z(s,alpha)*e(s) done by the following awk script
   paste z.ion.$ss.alpha.$alpha  e.vib.$nuu.ion.$ss | \
   awk ’{pol=$1*$4+$2*$5+$3*$6; print $0,"  ",pol}’ > matr-vib-${nuu}-alpha-${alpha}-ion-${ss}
  done
  sumpol=‘cat matr-vib-${nuu}-alpha-${alpha}-ion-* | awk ’{sum+=$7} END {print sum}’‘
  int=‘echo "$int+($sumpol)^2" | sed ’s/[eE]/*10^/g’ |  bc -l‘
 done
 freq=‘awk ’(NR==1){print $8}’ temps02/eigens.vib.$nuu‘
 echo "$nuu $freq $int">> exact.res.txt
 printf "."
done
printf " ..done\n"

# format results, normalize intensities
printf "..normalizing intensities"
max=‘awk ’(NR==1){max=$3} $3>=max {max=$3} END {print max}’ exact.res.txt‘
awk -v max="$max" ’{printf "%03u %6.1f %5.3f\n",$1,$2,$3/max}’ exact.res.txt > results.txt
printf " ..done\n"

# clean up, display results
printf "..finalizing:\n"
mkdir temps03; mv bornch.* e.vib.*.allions temps03/
mkdir temps04; mv z.ion* e.vib.*.ion.* temps04/
mkdir temps05; mv matr-* temps05/
mkdir results; mv *res*txt results/
let NMATRIX=$EIG_NVIBS**2
printf "%5u atoms found\n%5u vibrations found\n%5u matrices evaluated" \
       $EIG_NIONS $EIG_NVIBS $NMATRIX > results/statistics.txt
  # fast switch to clean up all temporary files
  rm -r temps*
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.

```
#!/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.

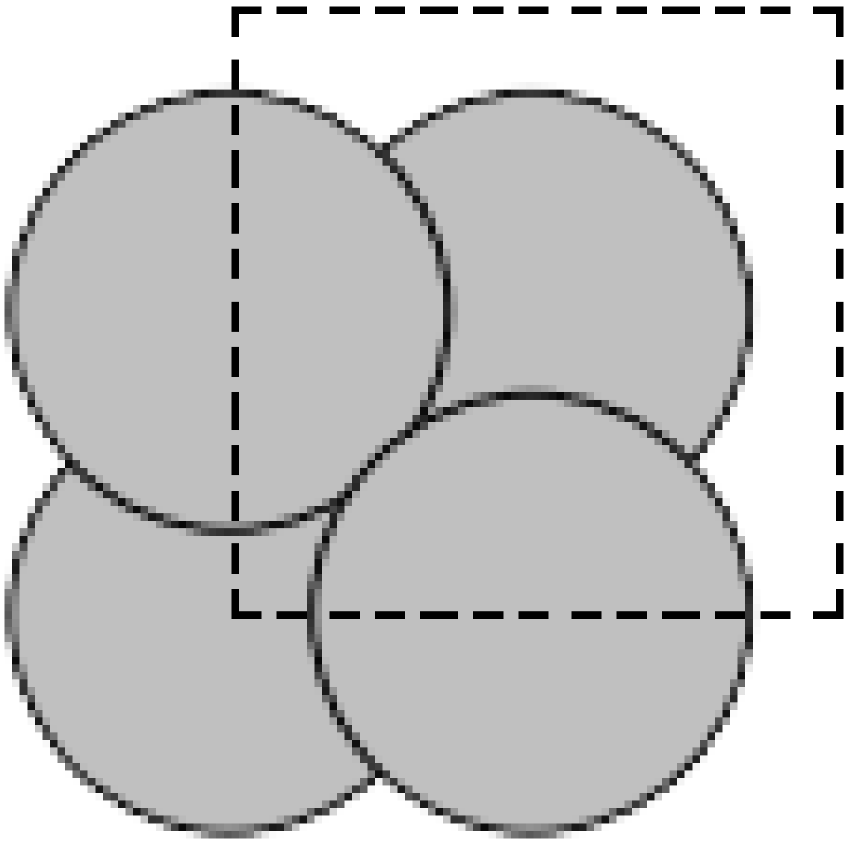
3.9.1 TODO Shell scripts are rough. One day I should rewrite this in python.

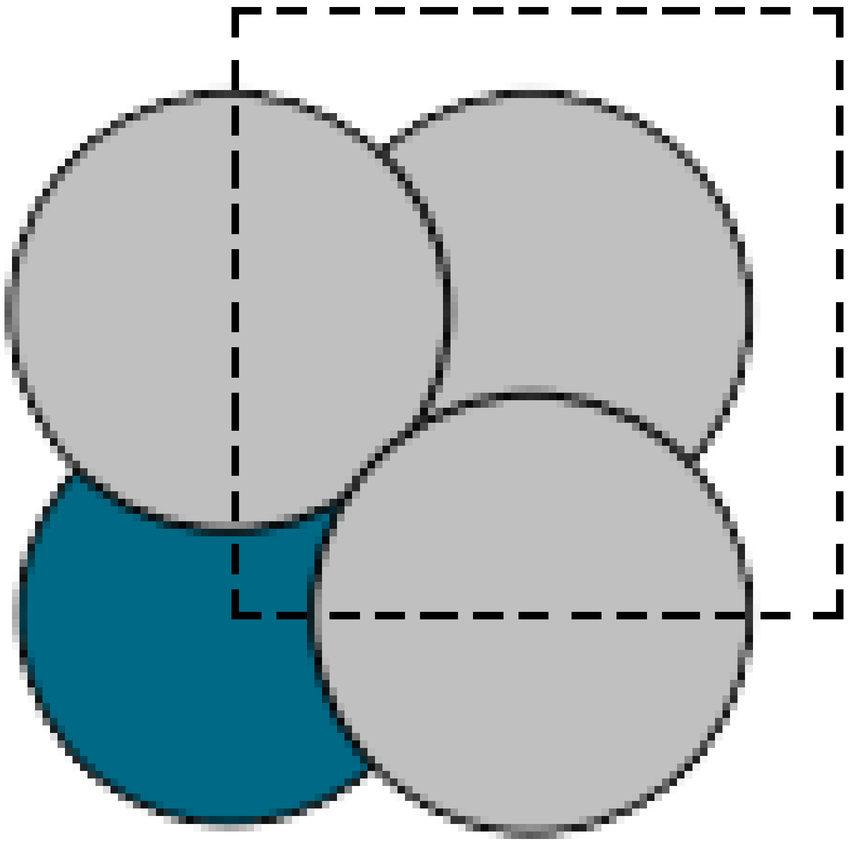
4 TODO Bulk systems

4.1 Defining and visualizing bulk systems

4.1.1 Built-in functions in ase

```
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 TODO Determining bulk structures

4.2.1 fcc/bcc

4.2.2 hcp

4.2.3 complex structures with internal degrees of freedom

4.3 TODO Bulk reaction energies

4.4 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.

```
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,
          xc='PBE',
          lreal=False,
          rwigs=[1.5],  # wigner-seitz radii for ados
          kpts=(8, 8, 8),
          atoms=bulk) as calc:

    # this runs the calculation
    bulk.get_potential_energy()

    # now get results
    ados = VaspDos(efermi=calc.get_fermi_level())

    energies = ados.energy
    dos = ados.site_dos(0, 'd')

    ind = (energies < 5) & (energies > -10)

    energies = energies[ind]
    dos = dos[ind]

    Nstates = np.trapz(dos, energies)
    occupied = energies <= 0.0
    N_occupied_states = np.trapz(dos[occupied], energies[occupied])
    ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
    wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)

    print 'Total # states = %1.2f' % Nstates
    print '# occupied states = %1.2f' % N_occupied_states
    print 'd-band center = %1.2f eV' % ed
    print 'd-band width  = %1.2f eV' % np.sqrt(wd2)

    # plot the d-band
    plot(energies, dos, label='$d$-orbitals')

    # plot the occupied states in shaded gray
    fill_between(x=energies[occupied],
                 y1=dos[occupied],
                 y2=zeros(dos[occupied].shape),
                 color='gray')

    xlabel('$E - E_f$ (eV)')
    ylabel('DOS (arbitrary units)')
    savefig('pd-ados.png')
    print '[[./{0}/pd-ados.png]]'.format(calc.vaspdir)
    print os.getcwd()
```

4.5 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

```
from jasp import *
JASPRC['queue.walltime'] = '10:00:00'

from ase import Atom, Atoms
from ase.visualize import view

a = 5.38936
atoms = Atoms([Atom('Si',[0,0,0]),
               Atom('Si',[0.25, 0.25, 0.25])])

atoms.set_cell([[a/2., a/2., 0.0],
                [0.0,  a/2., a/2.],
                [a/2., 0.0, a/2.]],scale_atoms=True)

with jasp('bulk/Si-selfconsistent',
          xc='PBE',
          prec='Medium',
          istart=0,
          icharg=2,
          ediff=0.1e-03,
          kpts=(4,4,4),
          atoms=atoms) as calc:
    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.

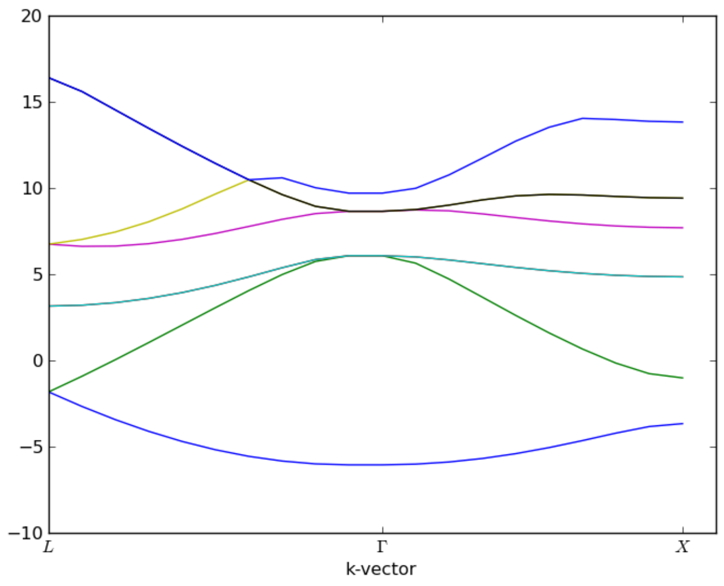
```
from jasp import *
JASPRC['queue.walltime'] = '10:00:00'

import os, shutil

wd = 'bulk/Si-bandstructure'
```

```

if not os.path.isdir(wd):
    shutil.copytree('bulk/Si-selfconsistent', wd)

kpts = [[0.5,0.5,0.0],   # L
        [0,0,0],         # Gamma
        [0,0,0],
        [0.5, 0.5, 0.5]] # X

with jasp(wd) as calc:
    calc.set(icharg=11) # update incar
    calc.write_kpoints(mode='line',
                       kpts=kpts,
                       kpt_format='rec',
                       intersections=10) # make new kpoint file
    print calc.calculate()

# 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.

```
f = open('bulk/Si-bandstructure/EIGENVAL', 'r')

line1 = f.readline()
line2 = f.readline()
line3 = f.readline()
line4 = f.readline()
comment = f.readline()
unknown, npoints, nbands = [int(x) for x in f.readline().split()]

blankline = f.readline()

band_energies = [[] for i in range(nbands)]

for i in range(npoints):
    x,y,z, weight = [float(x) for x in f.readline().split()]

    for j in range(nbands):
        fields = f.readline().split()
        id, energy = int(fields[0]), float(fields[1])
        band_energies[id-1].append(energy)
    blankline = f.readline()
f.close()

import matplotlib.pyplot as plt

for i in range(nbands):
    plt.plot(range(npoints), band_energies[i])

ax = plt.gca()
```

```python
ax.set_xticks([]) # no tick marks
plt.xlabel('k-vector')
ax.set_xticks([0,10,19])
ax.set_xticklabels(['$L$', '$\Gamma$', '$X$'])
plt.savefig('bulk/Si-bandstructure/bandstructure.png')
```



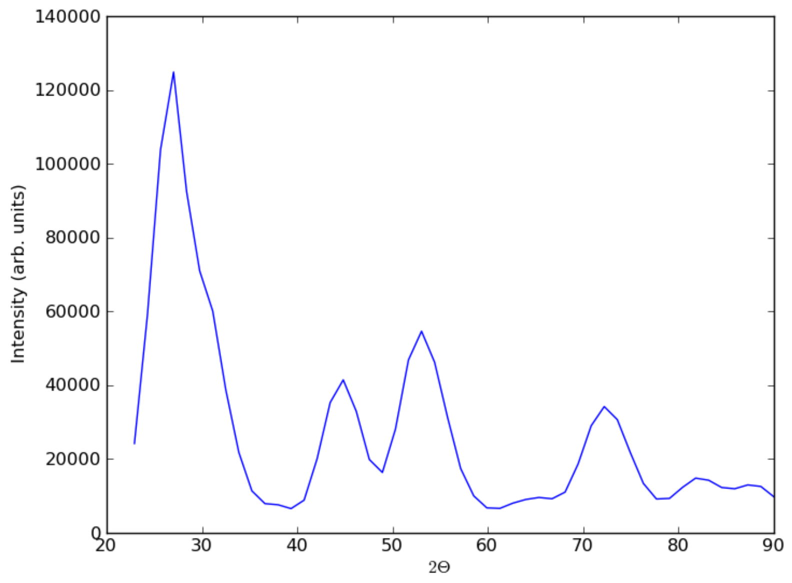
4.6 Simulated XRD spectra

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')
```

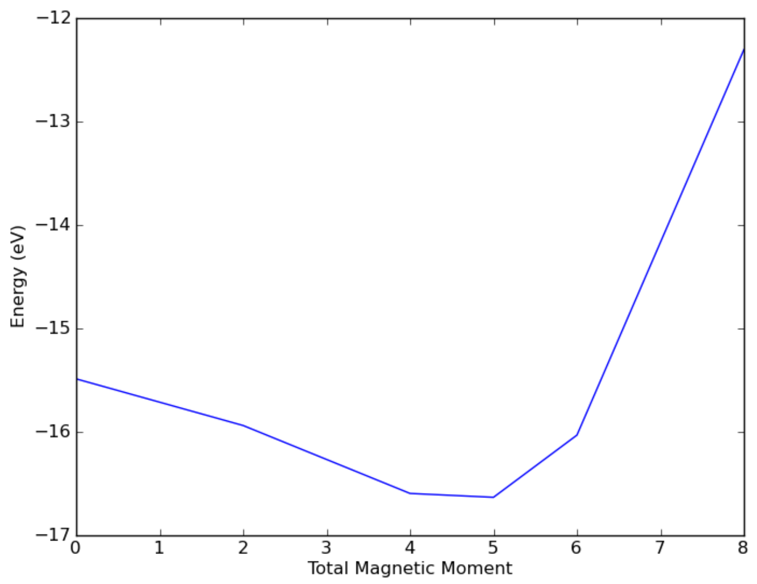


4.7 TODO Magnetism

4.7.1 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.

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

atoms = BodyCenteredCubic(directions=[[1,0,0],
                                      [0,1,0],
                                      [0,0,1]],
                                      size=(1,1,1),
                                      symbol='Fe')

NUPDOWNS = [0.0, 2.0, 4.0, 5.0, 6.0, 8.0]
energies = []
for B in NUPDOWNS:
    with jasp('bulk/Fe-bcc-fixedmagmom-{0:1.2f}'.format(B),
          xc='PBE',
          encut=300,
          kpts=(4,4,4),
          ispin=2,
          nupdown=B,
          atoms=atoms) as calc:
        try:
            e = atoms.get_potential_energy()
            energies.append(e)
        except (VaspSubmitted, VaspQueued):
            pass

import matplotlib.pyplot as plt
plt.plot(NUPDOWNS, energies)
plt.xlabel('Total Magnetic Moment')
plt.ylabel('Energy (eV)')
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.

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


atoms = BodyCenteredCubic(directions=[[1,0,0],
                                      [0,1,0],
                                      [0,0,1]],
                                      size=(1,1,1),
                                      symbol='Fe')

for atom in atoms:
    atom.magmom = 2.5

with jasp('bulk/Fe-bcc-sp-1',
          xc='PBE',
          encut=300,
          kpts=(4,4,4),
          ispin=2,
          atoms=atoms) as calc:
        try:
            e = atoms.get_potential_energy()
            print atoms.get_magnetic_moment()
            print atoms.get_magnetic_moments()
        except (VaspSubmitted, VaspQueued):
            pass
```

4.7.2 TODO find out how to get atoms magnetic moments.

5 TODO Surfaces

5.1 Surface structures

5.2 Surface relaxation

5.3 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.

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 [**?**]

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 [1, 2].

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.

```
inkscape images/surface-construction.svg
```

```
convert images/surface-construction.svg images/surface-construction.png
```

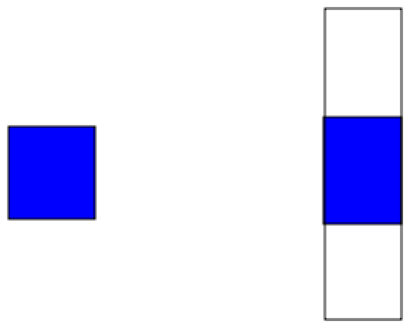


Figure 1: Schematic figure illustrating the calculation of a surface energy.

An alternative method for calculating surface energies that does not involve an explicit bulk calculation follows Ref. [2]. The method follows from equation (ref{eq:se}) where for a N-atom slab, in the limit of N $\rightarrow \infty$,

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

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

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

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

5.4 TODO Dipole correction

6 TODO Atomistic thermodynamics

7 TODO Advanced electronic structure methods

7.1 DFT+U

7.2 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
```

8.2 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

- 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:

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

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!)

```
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.

```
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),
                       symbol='Ag',
                       latticeconstant=4.0)

# these are the reciprocal lattice vectors
b1,b2,b3 = np.linalg.inv(ag.get_cell())

'''
g(111) = 1*b1 + 1*b2 + 1*b3

and |g(111)| = 1/d_111
'''
h,k,l = (1,1,1)
d = 1./np.linalg.norm(h*b1 + k*b2 + l*b3)

print 'd_111 spacing = %1.3f Angstroms' % d

#method #2
hkl = np.array([h,k,l])
G = np.array([b1,b2,b3]) #reciprocal unit cell

'''
Gstar is usually defined as this matrix of dot products:

Gstar = np.array([[dot(b1,b1), dot(b1,b2), dot(b1,b3)],
                  [dot(b1,b2), dot(b2,b2), dot(b2,b3)],
                  [dot(b1,b3), dot(b2,b3), dot(b3,b3)]])

but I prefer the notationally more compact:
Gstar = G .dot. transpose(G)

then, 1/d_hkl^2 = hkl .dot. Gstar .dot. hkl
'''

Gstar = np.dot(G,G.T)

id2 = np.dot(hkl,np.dot(Gstar,hkl))

print np.sqrt(1/id2)

# http://books.google.com/books?id=nJHSqEseuIUC&lpg=PA118&ots=YA9TBldoVH&dq=reciprocal%20metric%20tensor&pg=PA119#v=


'''Finally, many text books on crystallography use long algebraic
formulas for computing the d-spacing with sin and cos, vector lengths,
and angles. Below we compute these and use them in the general
triclinic structure formula which applies to all the structures.
'''
from Scientific.Geometry import Vector
```

```
import math

unitcell = ag.get_cell()
A = Vector(unitcell[0])
B = Vector(unitcell[1])
C = Vector(unitcell[2])

# lengths of the vectors
a = A.length()#*angstroms2bohr
b = B.length()#*angstroms2bohr
c = C.length()#*angstroms2bohr

# angles between the vectors in radians
alpha = B.angle(C)
beta = A.angle(C)
gamma = A.angle(B)

print '%1.3f %1.3f %1.3f %1.3f %1.3f %1.3f\n' % (a,b,c,
                                                  alpha,beta,gamma)

h,k,l = (1,1,1)

from math import sin, cos

id2 = ((h**2/a**2*sin(alpha)**2
       + k**2/b**2*sin(beta)**2
       + l**2/c**2*sin(gamma)**2
       +2*k*l/b/c*(cos(beta)*cos(gamma)-cos(alpha))
       +2*h*l/a/c*(cos(alpha)*cos(gamma)-cos(beta))
       +2*h*k/a/b*(cos(alpha)*cos(beta)-cos(gamma)))
       /(1-cos(alpha)**2-cos(beta)**2 - cos(gamma)**2
         +2*cos(alpha)*cos(beta)*cos(gamma)))

d = 1/math.sqrt(id2)

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.

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

print 'linear rotors: I = [Ia Ia 0]'
mlc = 'CO2'
atoms = molecule(mlc)

print '  %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
print

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

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

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

9 References

References

[1] J. C. Boettger. Nonconvergence of surface energies obtained from thin-film calculations. *Phys. Rev. B*, 49:16798–16800, Jun 1994.

[2] 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.