Modeling materials using density functional theory

John Kitchin

2012-07-11 Wed

Contents

https://wiki.fysik.dtu.dk/ase/

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 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.
- 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:

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

with this code:

```
1   CWD = os.getcwd()
2   os.chdir('directory')
3   calc=Vasp(lotsofkeywords)
4   atoms.set_calculator(calc)
5
6   try:
7       #do stuff
8   finally:
9       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`

```
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 $1\mathring{A} \times 1\mathring{A} \times 1\mathring{A}$ 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)
```
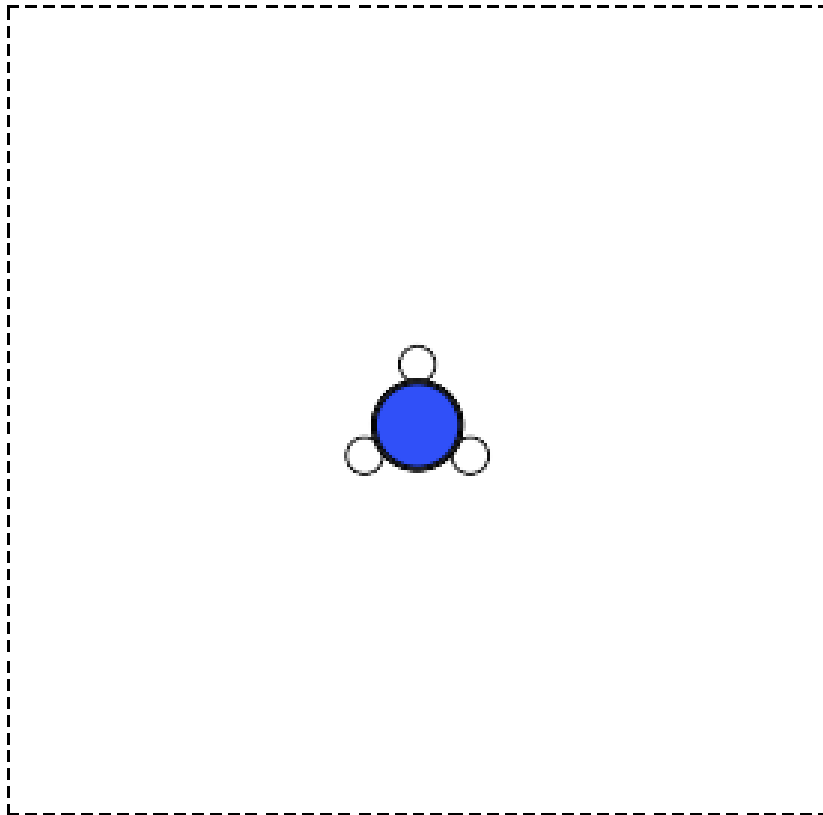
It is possible to rotate the atoms with `func:ase.io.write` if you wanted to see pictures from another angle. In the next example we rotate 45 degrees about the x-axis, then 45 degrees about the y-axis. Note that this only affects the image, not the actual coordinates

```python
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')
```
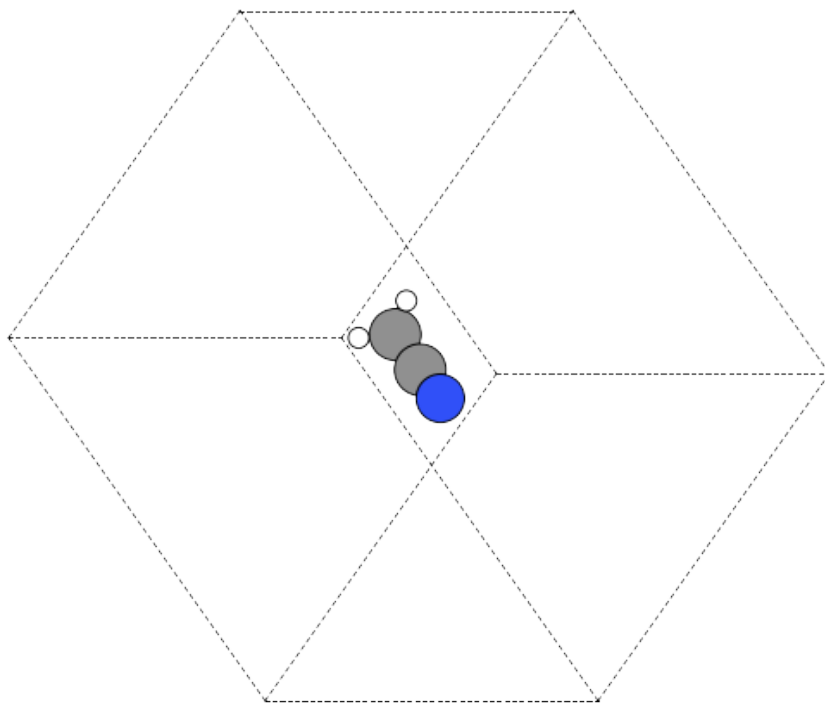
If you actually want to rotate the coordinates, there is a nice way to do that to, with the `func:ase.Atoms.rotate` method. Actually there are some subtelties in rotation. One rotates the molecule an angle (in radians) around a vector, but you have to choose whether the center of mass should be fixed 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.

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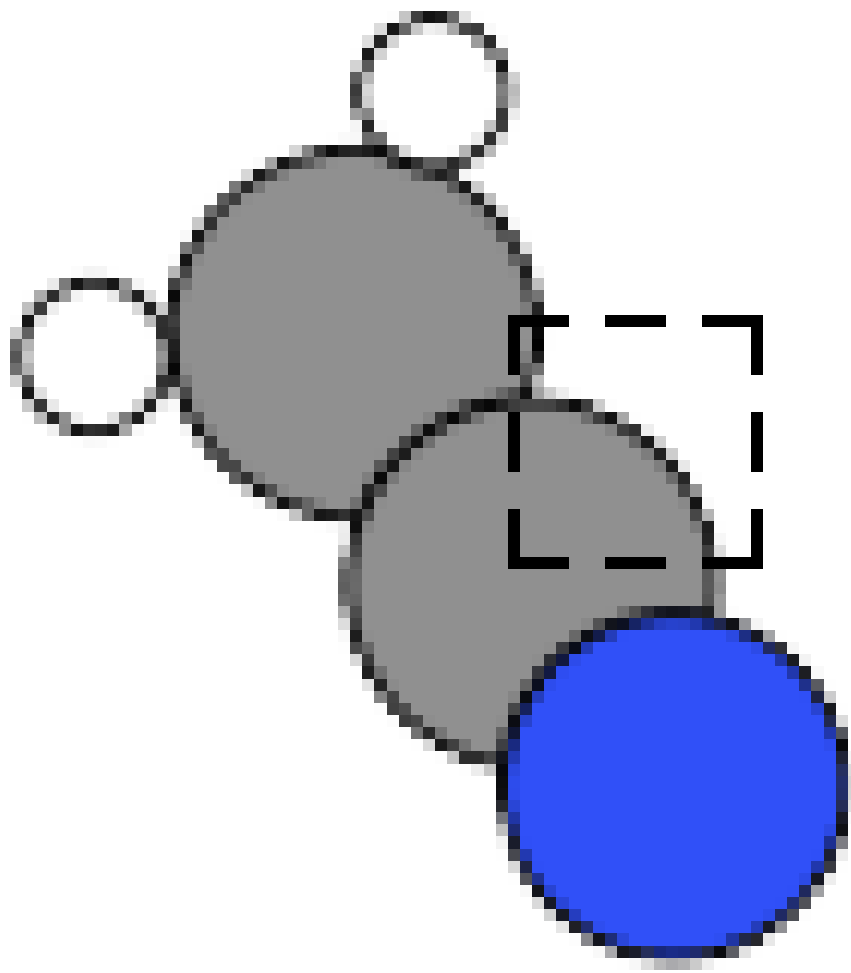
Note in this last case the unit cell is oriented differently than the previous example, since we chose not to rotate the unit cell.

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.

```
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)
```
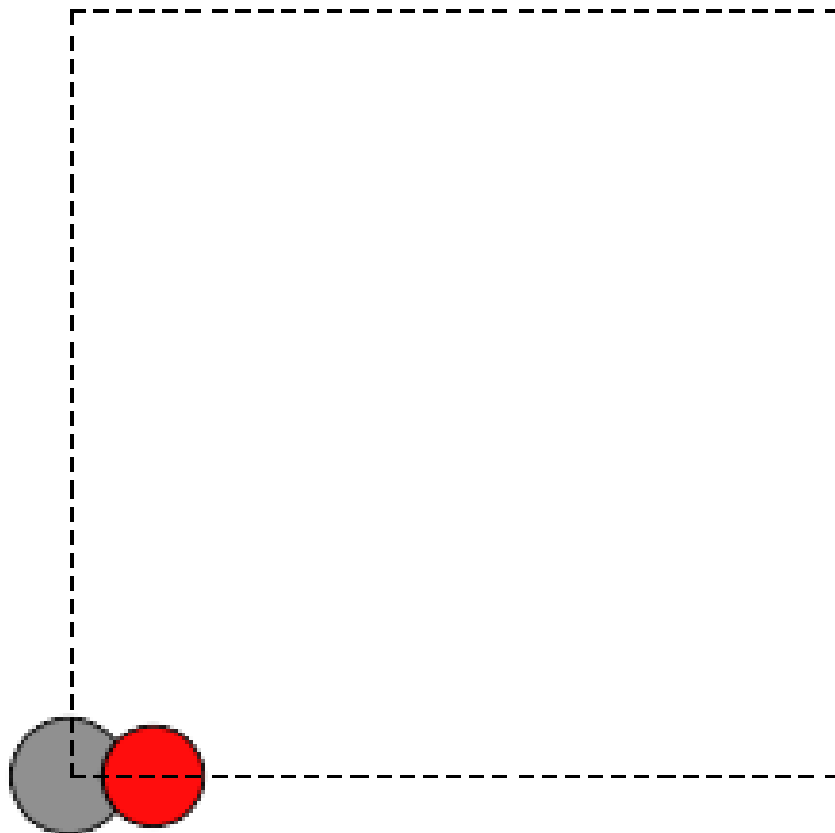
There are two inconvenient features of the simple cubic cell:

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)
```
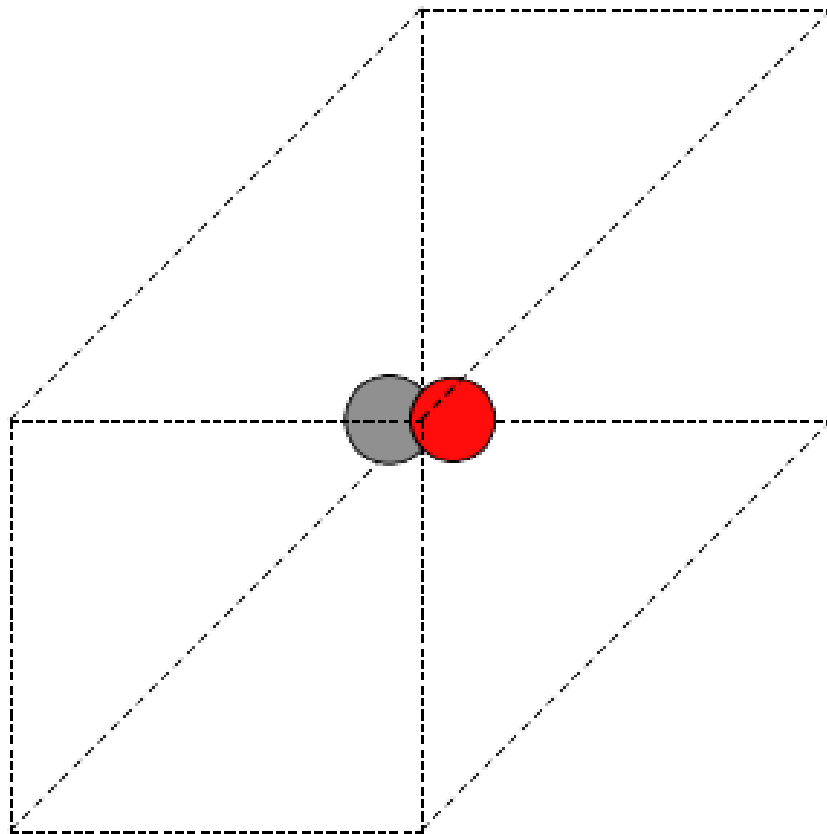

At this point you might ask "How do you know the distance to the neighboring image?" The `ag` viewer lets you compute this graphically, but we can use code to determine this too. All we have to do is figure out the length of each lattice vector. We use the numpy module to compute the distance of a vector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

11

```
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')
```
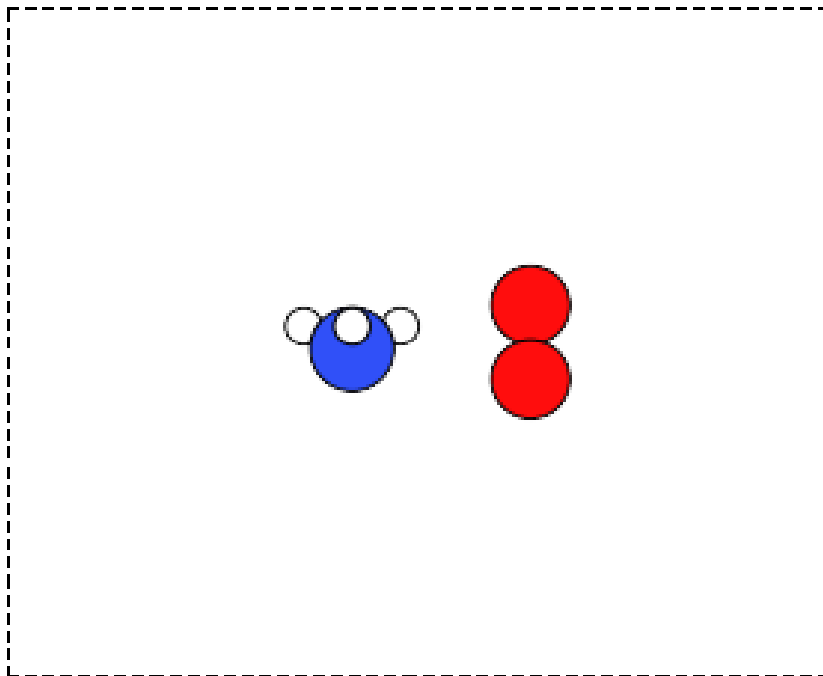
3.3 Simple properties

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)
```

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)
```

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.

$$\text{COM} = \sum m_i \cdot r_i \overline{\sum m_i}$$

```
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)
```

3.3.4   Moments of inertia

func:ase.Atoms.get_moments_of_inertia

```
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     atoms = molecule('CO2')
7     print '  CO2 moments of inertia: ',atoms.get_moments_of_inertia()
8     print

9

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

13

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

17

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

22

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

27

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

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)
```

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 $\theta = \frac{a \cdot b}{|a||b|}$, 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)
```

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
```

- 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
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
```

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.

It is straightforward to create a calculator that uses Dacapo to calculate everything. You just create an instance of `mod:ase.calculators.jacapo.Jacapo` as in this example. Jacapo is a python interface to Dacapo that also interacts with ase3. 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.

```
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.    ]]
```

3.3.7 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
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
```

3.5   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, 21.378951000000001, (2.4811570000000001+0j), (94.092513999999994+0j), (96.601394999999997+0j), (122.151211+0j)]

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).

3.6 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   TODO Molecular reaction energies

3.7.1   Water gas shift example

3.7.2   Temperature dependent water gas shift equilibrium constant

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 inten-

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

5 TODO Bulk systems

5.1 Defining and visualizing bulk systems

5.2 TODO Determining bulk structures

5.2.1 fcc/bcc

5.2.2 hcp

5.2.3 complex structures with internal degrees of freedom

5.3 TODO Bulk reaction energies

5.4 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()
```

```
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      ind = (energies < 5) & (energies > -10)
34
35      energies = energies[ind]
36      dos = dos[ind]
37
38      Nstates = np.trapz(dos, energies)
39      occupied = energies <= 0.0
40      N_occupied_states = np.trapz(dos[occupied], energies[occupied])
41      ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
42      wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)
43
44      print 'Total # states = %1.2f' % Nstates
45      print '# occupied states = %1.2f' % N_occupied_states
46      print 'd-band center = %1.2f eV' % ed
47      print 'd-band width  = %1.2f eV' % np.sqrt(wd2)
48
49      # plot the d-band
50      plot(energies, dos, label='$d$-orbitals')
51
52      # plot the occupied states in shaded gray
53      fill_between(x=energies[occupied],
54                   y1=dos[occupied],
55                   y2=zeros(dos[occupied].shape),
56                   color='gray')
57
58      xlabel('$E - E_f$ (eV)')
59      ylabel('DOS (arbitrary units)')
60      savefig('pd-ados.png')
61      print '[[./{0}/pd-ados.png]]'.format(calc.vaspdir)
62      print os.getcwd()
```

5.5   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

```
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.

```python
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()
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')
```
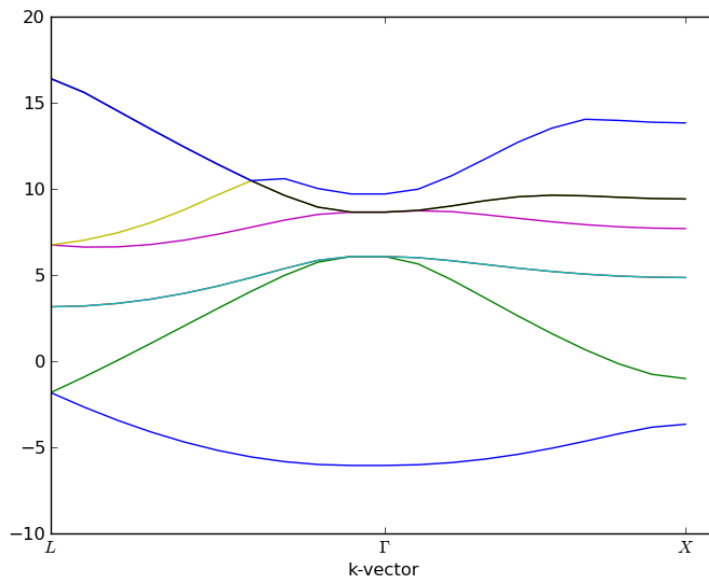
5.6 Simulated XRD spectra

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.

```
1  from ase import Atom,Atoms
2  from ase.xrdebye import *
3  import numpy as np
4
5  wavelength = 1.051967 # in angstroms
6
7  XRD = XrDebye(wavelength=wavelength,alpha=1.01,damping=0.0)
8
9  a = 3.92  # approximate lattice constant
10 b = a / 2.
11 atoms = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
12               cell=[(0, b, b),
```

33

```
13                       (b, 0, b),
14                       (b, b, 0)]).repeat((5,5,5))
15
16   thetas = np.linspace(0.2, np.pi/4.)
17
18   I = [XRD.get(atoms,s=s) for s in 2*np.sin(thetas)/wavelength]
19
20   import matplotlib.pyplot as plt
21   plt.plot((2*thetas)*180./np.pi,I)
22   plt.xlabel('$2\Theta$')
23   plt.ylabel('Intensity (arb. units)')
24   plt.savefig('images/pd-xrd.png')
```
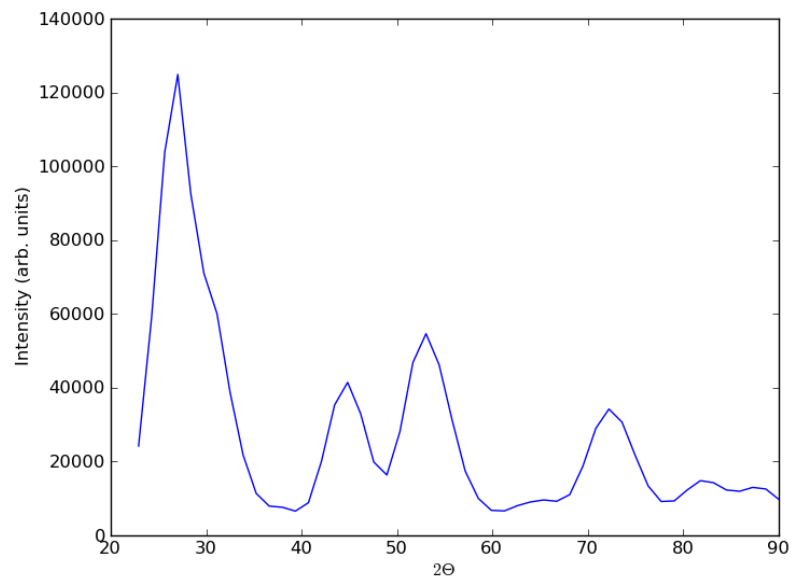

6   TODO Surfaces

6.1   Surface structures

6.2   Surface relaxation

6.3   Surface energy

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

```
1   convert images/surface-construction.svg images/surface-construction.png
```
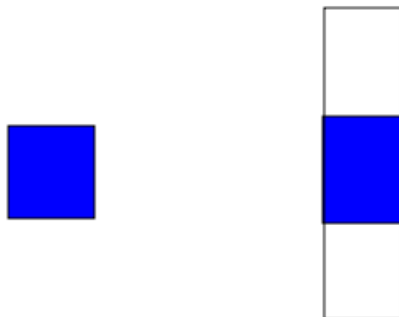
Figure 1: Schematic figure illustrating the calculation of a surface energy.

quantum size effects? A. Kiejna, J. Peisert and P. Scharoch, Surf. Sci. 432 (1999) 54

7   Atomistic thermodynamics

8   TODO Recipes

8.1   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
```

9 Computational geometry

9.1 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:

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

or, in standard form:

transpose(A) s = p

and we need to solve for s as:

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

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:

```
1  from ase import *
2  from ase.lattice.surface import *
3
4  np.set_printoptions(precision=3,suppress=True)
5
6  slab = fcc111('Pd',
7                a=3.92,        # Pd lattice constant
8                size=(2,2,3), #3-layer slab in 1x1 configuration
9                vacuum=10.0)
10
11 pos = slab.get_positions() #these positions use x,y,z vectors as a basis
12
13 # we want to see the atoms in terms of the unitcell vectors
14 newbasis = slab.get_cell()
15
16 s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
17 print 'Coordinates in new bases are: \n',s
18
19 # what we just did is equivalent to the following atoms method
20 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!)

```
1  from ase.lattice.cubic import BodyCenteredCubic
2  from ase.calculators.jacapo import *
3
4  bulk = BodyCenteredCubic(directions=[[1,0,0],
5                                        [0,1,0],
6                                        [0,0,1]],
7                           size=(2,2,2),
8                           latticeconstant=2.87,
9                           symbol='Fe')
10
11
12 newbasis = 2.87*np.array([[-0.5, 0.5, 0.5],
13                            [0.5, -0.5, 0.5],
14                            [0.5, 0.5, -0.5]])
15
16 pos = bulk.get_positions()
17
18 s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
19 print 'atom positions in primitive basis'
20 print s
21
22 #let's see the unit cell in terms of the primitive basis too
```

```
23    print 'unit cell in terms of the primitive basis'
24    print np.dot(np.linalg.inv(newbasis.T),bulk.get_cell().T).T
```

9.2   Simple distances, angles

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

```
1    import numpy as np
2    from Scientific.Geometry import *
3
4    A = Vector([1,1,1])    #Scientfic
5    a = np.array([1,1,1]) #numpy
6
7    B = Vector([0.0,1.0,0.0])
8
9    print '|A| = ',A.length()          #Scientific Python way
10   print '|a| = ',np.sum(a**2)**0.5 #numpy way
11   print '|a| = ',np.linalg.norm(a) #numpy way 2
12
13   print 'ScientificPython angle = ',A.angle(B) #in radians
14   print 'numpy angle =                ',np.arccos(np.dot(a/np.linalg.norm(a),B/np.linalg.norm(B)))
15
16   #cross products
17   print 'Scientific A .cross. B = ',A.cross(B)
18   print 'numpy A .cross. B      = ',np.cross(A,B) #you can use Vectors in numpy
```

9.3   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.

```
1    import numpy as np
2
```

```
3    a1 = [2, 0, 0]
4    a2 = [1, 1, 0]
5    a3 = [0, 0, 10]
6
7    uc = np.array([a1, a2, a3])
8
9    print np.dot(np.cross(a1,a2),a3)
10   print np.linalg.det(uc)
11
12   from ase import *
13
14   atoms = Atoms([],cell=uc) #empty list of atoms
15   print atoms.get_volume()
```

9.4   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.

```
1    from ase import *
2    import numpy as np
3    from ase.lattice.cubic import FaceCenteredCubic
4
5    ag = FaceCenteredCubic(directions=[[1,0,0],
6                                       [0,1,0],
7                                       [0,0,1]],
8                           size=(1,1,1),
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

9.5 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`

9.6 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
```

10   References