Modeling materials using density functional theory

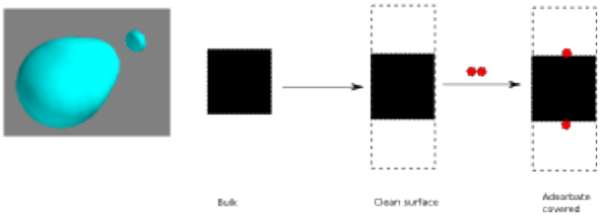
John Kitchin

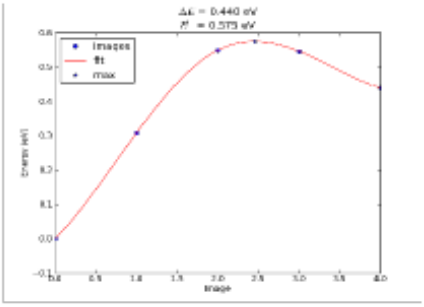
2012-07-11 Wed

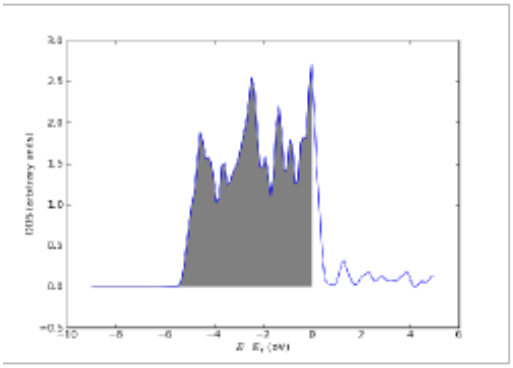
Contents

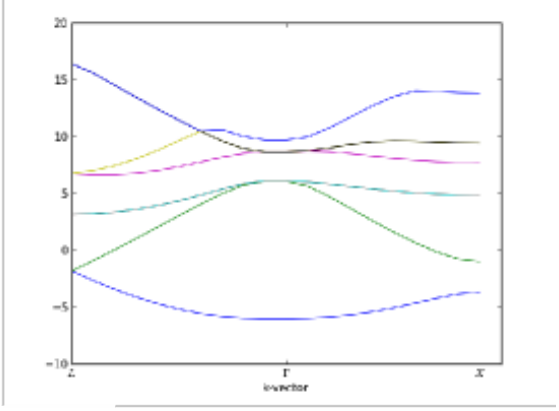
Modeling materials using density functional theory

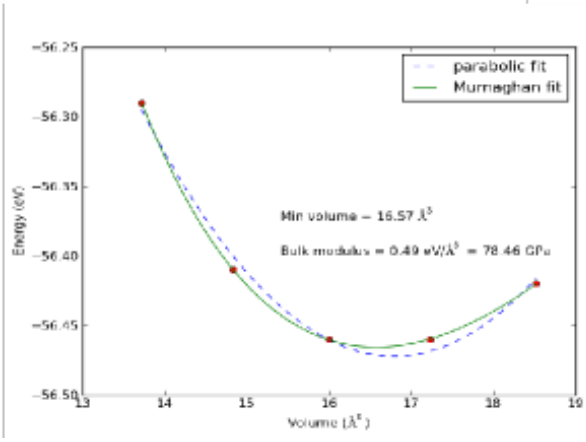
By John R. Kitchin

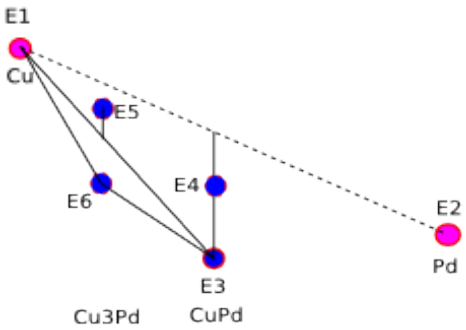












1 List of figures

List of Figures

2 List of tables

List of Tables

3 Introduction to this book

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

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

This book makes heavy use of many computational tools including:

- Python
  - Module index
- Atomic Simulation Environment (ase)
- numpy
- scipy
- matplotlib
- emacs
  - org-mode This book is written in org-mode, and is best read in emacs in org-mode. This format provides clickable links, easy navigation, syntax highlighting, as well as the ability to interact with the tables and code. The book is also available in pdf.
- git This book is available at https://github.com/jkitchin/dft-book

- jasp

jasp is a sophisticated wrapper to the `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 probably requires the latest svn version of ase to work since I have been adding new functionality to `ase.calculators.vasp` as `jasp` is developed.

The goal in writing `jasp` was to create a computational environment where you write one script to create, run, and analyze your calculations. The code is smart, and submits jobs to a queue system when required, does not submit jobs more than once, allows many jobs to run in parallel, and gives you answers when available. `jasp` takes care of changing into VASP calculation directories, and then changing back to where you started from. `jasp` has an interface to sqlite, and to version control (git) software. `jasp` has some features that `ase.calculators.vasp` does not have yet. They may get incorporated into ase one day.

`jaspsum` is a utility program provided in jasp which prints a pretty representation of the state of a calculation, plots the relaxation trajectory, or prints a code representation of a vasp directory.

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 `ase.calculators.vasp`, so you can replace this kind of code:

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

with this code:

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

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

The DFT code used primarily in this book is VASP.

- VASP wiki
- VASP Manual

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

Exercise 3.1
Review all the hyperlinks in this chapter.

4 Molecules

In this chapter we consider how to construct models of molecules, how to manipulate them, and how to calculate many properties of molecules. For a nice comparison of VASP and Gaussian see [30].

4.1 Defining and visualizing molecules

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

4.1.1 Predefined molecules

`ase` defines a number of molecular geometries in the `ase.data.molecules` database. For example, the databse includes the molecules in the G2/97 database [8]. This database contains a broad set of atoms and molecules for which good experimental data exists, making them useful for benchmarking studies. See this site for the original files.

The coordinates for the atoms in the database are MP2(full)/6-31G(d) optimized geometries. Here is a list of all the species available in `ase.data.g2`. You may be interested in reading about some of the other databases in `ase.data` too.

```
from ase.data import g2
keys = g2.data.keys()
# print in 3 columns
for i in range(len(keys)/3):
    print '{0:25s}{1:25s}{2:25s}'.format(*tuple(keys[i*3:i*3+3]))
```

```
isobutene                 CH3CH2OH                  CH3COOH
COF2                      CH3NO2                    CF3CN
CH3OH                     CCH                       CH3CH2NH2
PH3                       Si2H6                     O3
O2                        BCl3                      CH2_s1A1d
Be                        H2CCl2                    C3H9C
C3H9N                     CH3CH2OCH3                BF3
CH3                       CH4                       S2
C2H6CHOH                  SiH2_s1A1d                H3CNH2
CH3O                      H                         BeH
P                         C3H4_C3v                  C2F4
OH                        methylenecyclopropane     F2O
SiCl4                     HCF3                      HCCl3
C3H7                      CH3CH2O                   AlF3
CH2NHCH2                  SiH2_s3B1d                H2CF2
SiF4                      H2CCO                     PH2
OCS                       HF                        NO2
SH2                       C3H4_C2v                  H2O2
CH3CH2Cl                  isobutane                 CH3COF
HCOOH                     CH3ONO                    C5H8
2-butyne                  SH                        NF3
HOCl                      CS2                       P2
```

C	CH3S	O
C4H4S	S	C3H7Cl
H2CCHCl	C2H6	CH3CHO
C2H4	HCN	C2H2
C2Cl4	bicyclobutane	H2
C6H6	N2H4	C4H4NH
H2CCHCN	H2CCHF	cyclobutane
HCl	CH3OCH3	Li2
Na	CH3SiH3	NaCl
CH3CH2SH	OCHCHO	SiH4
C2H5	SiH3	NH
ClO	AlCl3	CCl4
NO	C2H3	ClF
HCO	CH3CONH2	CH2SCH2
CH3COCH3	C3H4_D2d	CH
CO	CN	F
CH3COCl	N	CH3Cl
Si	C3H8	CS
N2	Cl2	NCCN
F2	CO2	Cl
CH2OCH2	H2O	CH3CO
SO	HCOOCH3	butadiene
ClF3	Li	PF3
B	CH3SH	CF4
C3H6_Cs	C2H6NH	N2O
LiF	H2COH	cyclobutene
LiH	SiO	Si2
C2H6SO	C5H5N	trans-butane
Na2	C4H4O	SO2
NH3	NH2	CH2_s3B1d
ClNO	C3H6_D3h	Al
CH3SCH3	H2CO	CH3CN

Some other databases include the `ase.data.s22` for weakly interacting dimers and complexes, and `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 1Å × 1Å × 1Å 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 Å of vacuum on each side. In the write command we use the option `show_unit_cell` =2 to draw the unit cell boundaries. See Figure 1.

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

c60 = molecule('CH3CN')

c60.center(vacuum=6)
print 'unit cell'
```

```
print '---------'
print c60.get_cell()

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

```
unit cell
---------
[[ 13.775328   0.         0.      ]
 [  0.        13.537479   0.      ]
 [  0.         0.        15.014576]]
```

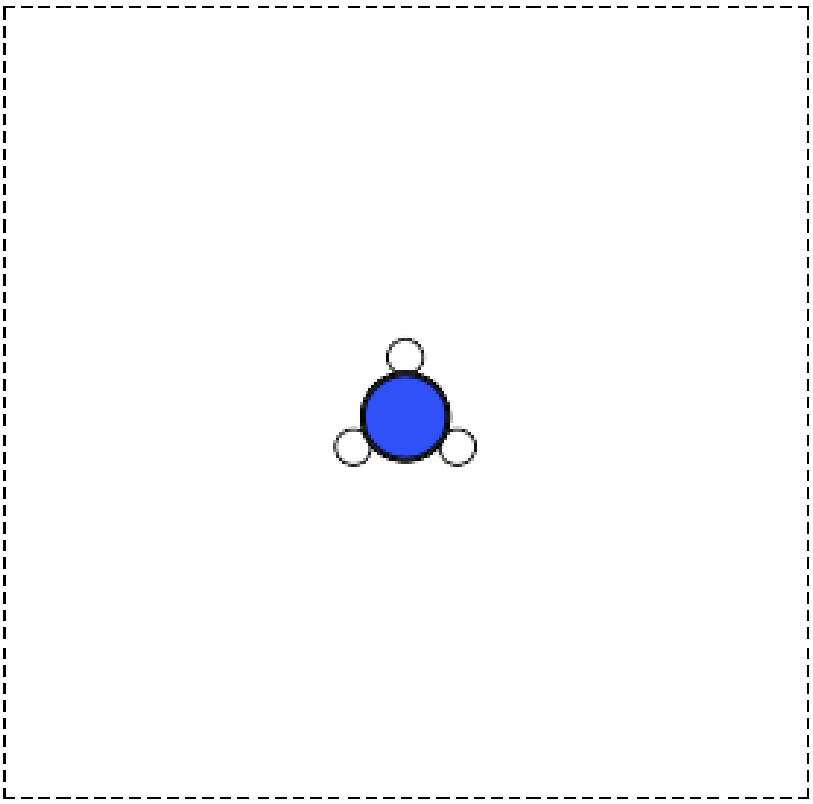


Figure 1: A CH_3CN molecule in a box.

It is possible to rotate the atoms with `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. See Figure 2

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

atoms = molecule('CH3CN')

```

```
atoms.center(vacuum=6)
print 'unit cell'
print '---------'
print atoms.get_cell()

write('images/ch3cn-rotated.png', atoms,
      show_unit_cell=2,rotation='45x,45y,0z')
```

```
unit cell
---------
[[ 13.775328   0.         0.      ]
 [  0.        13.537479   0.      ]
 [  0.         0.        15.014576]]
```

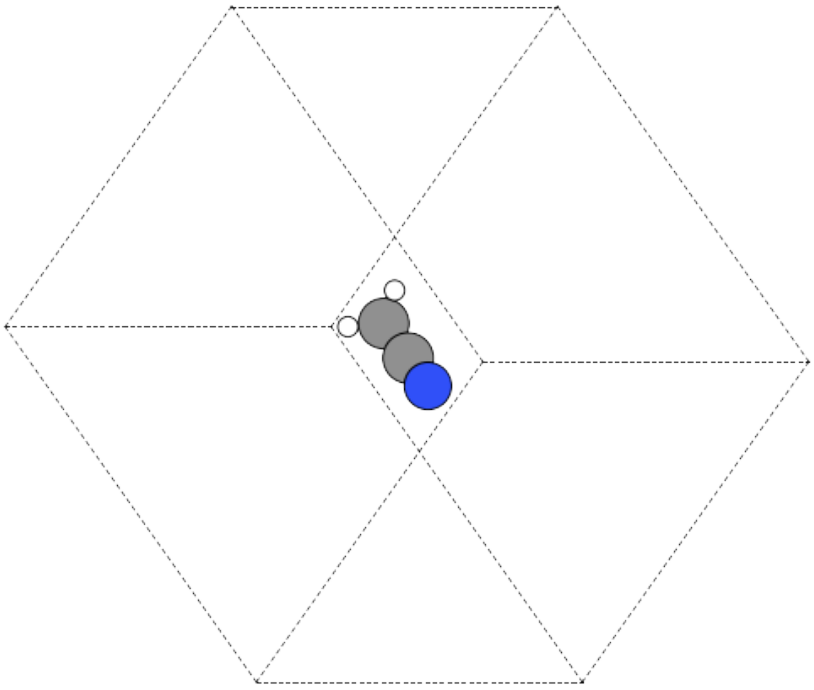


Figure 2: The rotated version of CH_3CN.

If you actually want to rotate the coordinates, there is a nice way to do that to, with the `ase.Atoms.rotate` method. Actually there are some subtleties 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. Note that the write function uses the rotation angle in degrees, while the rotate function uses radians.

```
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('images/ch3cn-rotated-2.png',atoms,show_unit_cell=2)
print 'difference in positions after rotating'
print 'atom    difference vector'
print '-------------------------------------'
p2 = atoms.get_positions()

diff = p2 - p1
for i,d in enumerate(diff):
    print '{0} {1}'.format(i, d)
```

```
difference in positions after rotating
atom    difference vector
-------------------------------------
0 [-0.65009456  0.91937255  0.65009456]
1 [ 0.08030744 -0.11357187 -0.08030744]
2 [ 0.66947344 -0.94677841 -0.66947344]
3 [-0.32532156  0.88463727  1.35030756]
4 [-1.35405183  1.33495444 -0.04610517]
5 [-0.8340703   1.33495444  1.2092413 ]
```

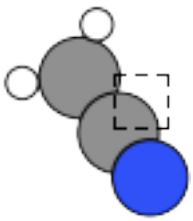


Figure 3: Rotated CH_3CN molecule

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

4.1.2 Reading other data formats into a calculation

`ase.io.read` supports reading in many different formats:

```
Known formats:

=========================  ===========
format                     short name
=========================  ===========
GPAW restart-file          gpw
Dacapo netCDF output file  dacapo
Old ASE netCDF trajectory  nc
```

```
Virtual Nano Lab file      vnl
ASE pickle trajectory      traj
ASE bundle trajectory      bundle
GPAW text output           gpaw-text
CUBE file                  cube
XCrySDen Structure File    xsf
Dacapo text output         dacapo-text
XYZ-file                   xyz
VASP POSCAR/CONTCAR file   vasp
VASP OUTCAR file           vasp_out
SIESTA STRUCT file         struct_out
ABINIT input file          abinit
V_Sim ascii file           v_sim
Protein Data Bank          pdb
CIF-file                   cif
FHI-aims geometry file     aims
FHI-aims output file       aims_out
VTK XML Image Data         vti
VTK XML Structured Grid    vts
VTK XML Unstructured Grid  vtu
TURBOMOLE coord file       tmol
TURBOMOLE gradient file    tmol-gradient
exciting input             exi
AtomEye configuration      cfg
WIEN2k structure file      struct
DftbPlus input file        dftb
CASTEP geom file           cell
CASTEP output file         castep
CASTEP trajectory file     geom
ETSF format                etsf.nc
DFTBPlus GEN format        gen
CMR db/cmr-file            db
CMR db/cmr-file            cmr
LAMMPS dump file           lammps
Gromacs coordinates        gro
=========================  ===========
```

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

```
14

C       0.000000000000000      0.000000000000000      0.376949000000000
H       0.000000000000000      0.000000000000000      1.475269000000000
C       0.000000000000000      1.450290000000000     -0.096234000000000
H       0.000000000000000      1.493997000000000     -1.190847000000000
H      -0.885482000000000      1.984695000000000      0.261297000000000
```

```
H       0.885482000000000      1.984695000000000      0.261297000000000
C       1.255988000000000     -0.725145000000000     -0.096234000000000
H       1.293839000000000     -0.746998000000000     -1.190847000000000
H       2.161537000000000     -0.225498000000000      0.261297000000000
H       1.276055000000000     -1.759198000000000      0.261297000000000
C      -1.255988000000000     -0.725145000000000     -0.096234000000000
H      -1.293839000000000     -0.746998000000000     -1.190847000000000
H      -1.276055000000000     -1.759198000000000      0.261297000000000
H      -2.161537000000000     -0.225498000000000      0.261297000000000
```

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 (Figure 4).

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

```
None
```

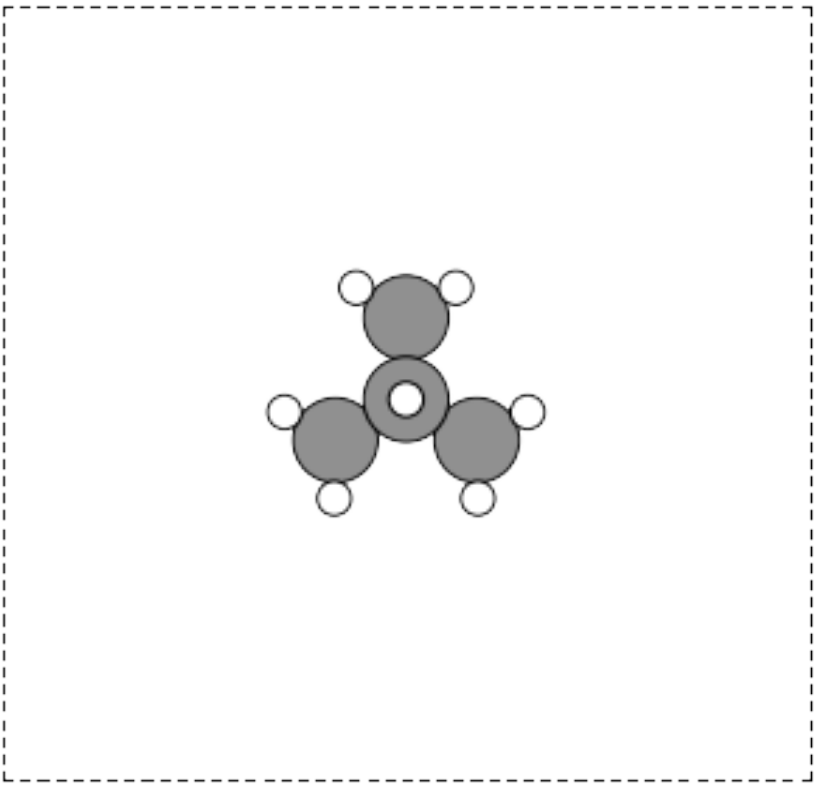


Figure 4: An isobutane molecule read in from an xyz data file.

4.1.3 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 for a CO molecule (Figure 5).

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

```
V = 1000 Ang^3
```

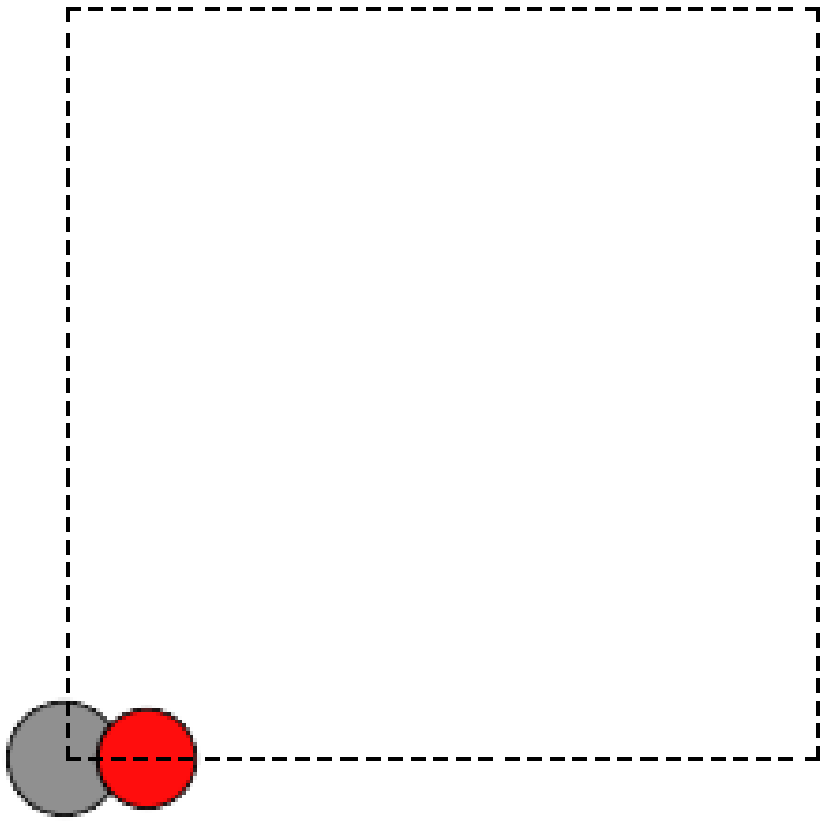


Figure 5: Image of a CO molecule with the C at the origin.

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 (see Visualizing electron density).

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 Å away. The volume of the unit cell is 1000 Å^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 in Figure 6, where we have guessed values for b until the CO molecules are on average 10 Å apart. Note the final volume is only about 715 Å^3, which is smaller than the cube. This will result in less computational time to compute properties.

```
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('images/fcc-cell.png',atoms2,show_unit_cell=2)
```

```
V = 715.8 Ang^3
```

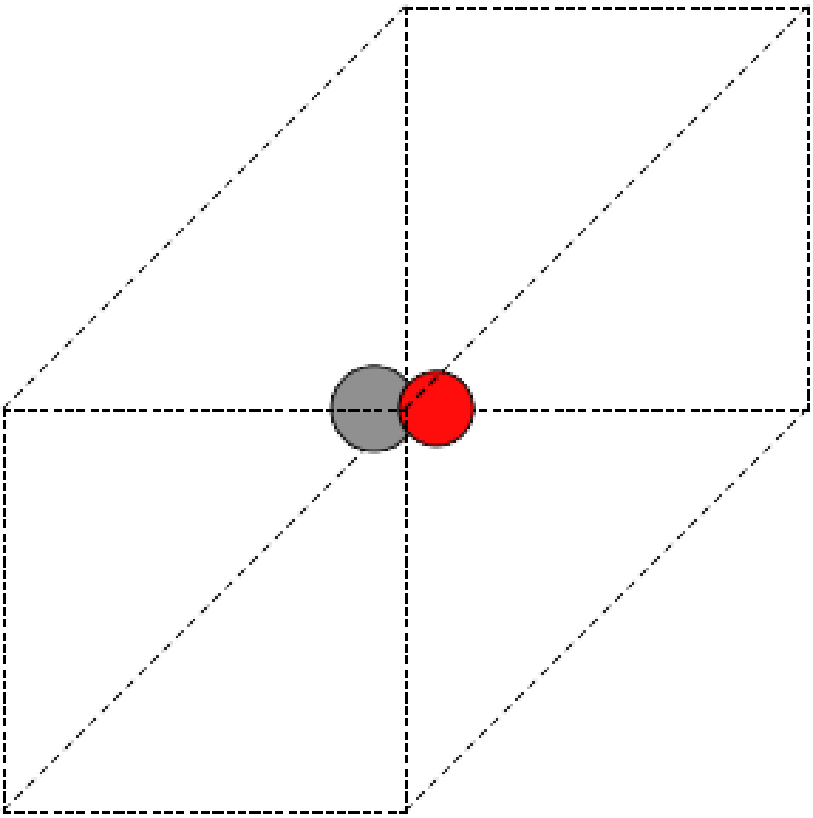


Figure 6: CO in an fcc cell.

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
```

```
|a1| = 10.04 Ang
|a2| = 10.04 Ang
|a3| = 10.04 Ang
```

4.1.4 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. See Figure 7.

```
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('images/bothatoms.png', bothatoms, show_unit_cell=2, rotation='90x')
```

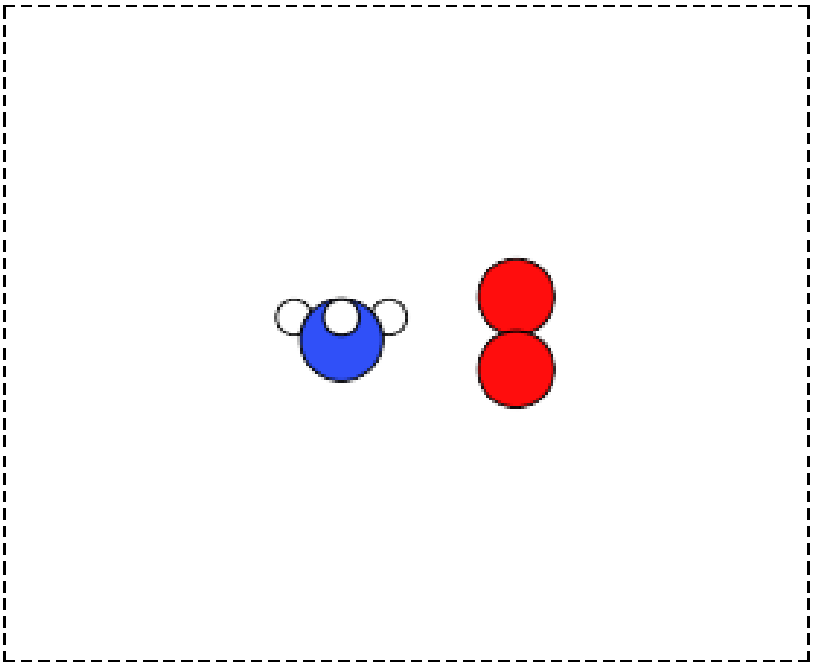


Figure 7: Image of the combined ammonia and oxygen molecule.

4.2 Simple properties

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

4.2.1 Getting cartesian positions

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

4.2.2 Molecular weight and molecular formula

We can quickly compute the molecular weight of a molecule with this recipe. We use the `ase.Atoms.get_masses`.

```
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
```

Note the argument `reduce=True` for `ase.Atoms.get_chemical_symbols`. This collects all the symbols and provides a molecular formula.

4.2.3 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 `ase.Atoms.get_center_of_mass`.

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

The center of mass is essentially the average position of the atoms, weighted by the mass of each atom.

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

# ammonia
atoms = molecule('NH3')

print 'COM1 = ',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 'COM2 = ', COM

# one-line linear algebra definition of COM
print 'COM3 = ',np.dot(masses, pos)/np.sum(masses)
```

```
COM1 =  [  0.00000000e+00   5.91843349e-08   4.75457009e-02]
COM2 =  [  0.00000000e+00   5.91843349e-08   4.75457009e-02]
COM3 =  [  0.00000000e+00   5.91843349e-08   4.75457009e-02]
```

You can see these are all the same.

4.2.4 Moments of inertia

The moment of inertia is a measure of resistance to changes in rotation. It is defined by $I = \sum_{i=1}^{N} m_i r_i^2$ where r_i is the distance to an axis of rotation. There are typically three moments of inertia, although some may be zero depending on symmetry, and some may be degenerate. There is a convenient function to get the moments of inertia: `ase.Atoms.get_moments_of_inertia`

Here are several examples.

```
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()
print

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
  C3H7Cl moments of inertia:  [  19.41351508  213.18961963  223.16255537]
```

4.3 Simple properties that require computations

In this section we cover properties that require simple calculations, but not yet DFT calculations.

4.3.1 Computing bond lengths and angles

A typical question we might ask is what is the structure of a molecule? By that we mean what are the bond lengths, angles between bonds, etc... The Atoms object contains a `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 that the indices start at 0.

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

# ammonia
atoms = molecule('NH3')

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

# N-H bond length
s = 'The N-H distance is {0:1.3f} angstroms'
print s.format(atoms.get_distance(0,1))
```

```
atom symbol
===========
 0   N
 1   H
 2   H
 3   H
The N-H distance is 1.017 angstroms
```

Bond angles are a little trickier. If we had vectors describing the directions between three atoms, we could use some simple trigonometry: $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$, so we can calculate the angle as $\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}\right)$, we just have to define our two vectors $\vec{a}$ and $\vec{b}$. 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-H_1 and N-H_2. In the next example, we utilize functions in `numpy` to perform the calculations, specifically the `numpy.arccos` function, the `numpy.dot` function, and `numpy.linalg.norm` functions.

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

# ammonia
atoms = molecule('NH3')

print 'atom symbol'
print '============'
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)
```

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

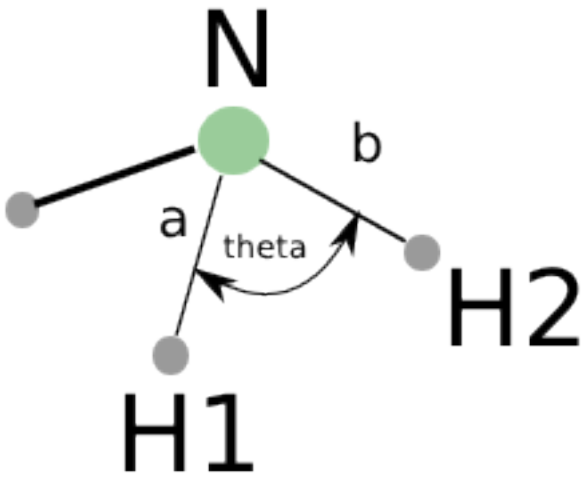


Figure 8: Schematic of the vectors defining the H-N-H angle.

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

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

print 'theta = {0} degrees'.format(atoms.get_angle([1,0,2])*180./pi)
```

```
theta = 106.334624232 degrees
```

Dihedral angles There is support in ase for computing dihedral angles. Let us 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°. See Figure 9.

```
# calculate an ethane dihedral angle
from ase import Atoms
from ase.structure import molecule
from ase.visualize import view
import numpy as np

atoms = molecule('C2H6')
view(atoms)

print 'atom symbol'
print '============'
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
```

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

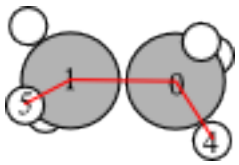


Figure 9: Schematic of the calculated ethane dihedral angle.

4.3.2 Energy and forces

Two of the most important quantities we are interested in is the total energy and the forces on the atoms. To get these, we have to define a calculator and attach it to an `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. We will see how to minimize this force in Manual determination and Automatic geometry optimization with VASP.

Note:
This is your first DFT calculation in the book! see ISMEAR, SIGMA, NBANDS, and ENCUT to learn more about these VASP keywords.

```
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',  # the exchange-correlation functional
          nbands=6,  # number of bands
          encut=350, # planewave cutoff
          ismear=1,  # Methfessel-Paxton smearing
          sigma=0.01,# very small smearing factor for a molecule
          atoms=co):

    print 'energy = {0} eV'.format(co.get_potential_energy())
    print co.get_forces()
```

```
energy = -14.687906 eV
[[ 5.095  0.     0.   ]
 [-5.095  0.     0.   ]]
```

4.3.3 The density of states

The density of states gives you the number of electronic states (i.e. orbitals) that have a particular energy. We can get this information from the last calculation we just ran without having to run another DFT calculation.

Warning:
One way that `jasp` is unintuitive is that whenever you are "inside" (indented) a `with` statement, the current working directory may be different than where your script started. In this example that can lead to your figure not being saved where you expect it!

```
from ase.dft.dos import *
from jasp import *
from pylab import *

with jasp('molecules/simple-co') as calc:
```

```
    dos = DOS(calc)
    plot(dos.get_energies(),dos.get_dos())
    xlabel('Energy - $E_f$ (eV)')
    ylabel('DOS')

# make sure you save the figure outside the with statement, or provide
# the correct relative or absolute path to where you want it.
savefig('images/co-dos.png')
```

```
None
```

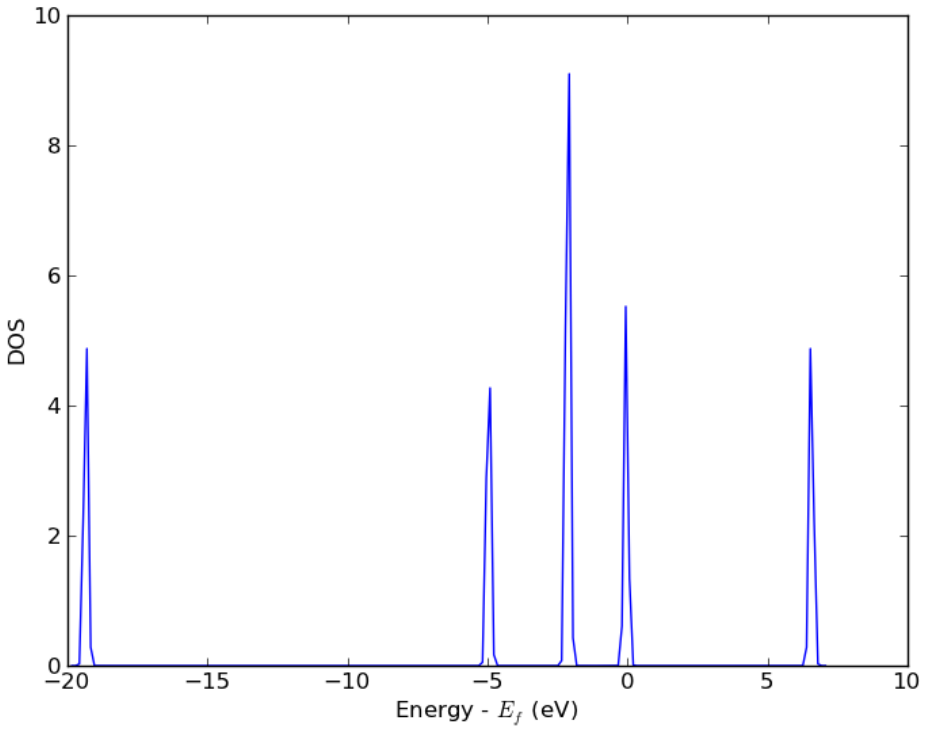


Figure 10: Density of states for a CO molecule.

4.3.4 Atom-projected density of states on molecules

Let us consider which states in the density of states belong to which atoms in a molecule. This can only be a qualitative consideration because the orbitals on the atoms often hybridize to form molecular orbitals, e.g. in methane the s and p orbitals can form what we call sp_3 orbitals. We can compute atom-projected density of states in VASP, which is done by projecting the wave function onto localized atomic orbitals. Here is an example. We will consider the CO molecule. To get atom-projected density of states, we must set RWIGS for each atom. This parameter defines the radius of the sphere around the atom which cuts off the projection.

Note that unlike the DOS, here we must run another calculation because we did not specify the atom-projected keywords above. Our strategy is to get the atoms from the previous calcu-

lation, and use them in a new calculation. You could redo the calculation in the same directory, but

Note:
The RWIGS is not uniquely determined for an element. There are various natural choices, e.g. the ionic radius of an atom, or a value that minimizes overlap of neighboring spheres, but these values can change slightly in different environments.

The total density of states and projected density of states information comes from the DOSCAR file.

Note:
You can get spin-polarized atom-projected density of states, and magnetization projected density of states. See http://cms.mpi.univie.ac.at/vasp/vasp/DOSCAR_file.html#doscar for more details.

```
from ase.dft.dos import *
from jasp import *
from pylab import *

# get the geometry from another calculation
with jasp('molecules/simple-co') as calc:
    atoms = calc.get_atoms()

with jasp('molecules/co-ados',
    encut=300,
    xc='PBE',
    rwigs=[1.0, 1.0],     # these are the cutoff radii for projected states
    atoms=atoms) as calc:
    calc.calculate()

    # now get results

    dos = DOS(calc)
    plot(dos.get_energies(), dos.get_dos() + 10)

    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy

    plot(energies, ados.dos + 8) # these are the total DOS

    c_s = ados.site_dos(0, 's')
    c_p = ados.site_dos(0, 'p')
    o_s = ados.site_dos(1, 's')
    o_p = ados.site_dos(1, 'p')

    c_d = ados.site_dos(0,'d')
    o_d = ados.site_dos(1,'d')

    plot(energies, c_s + 6, energies, o_s + 5)
    plot(energies, c_p + 4, energies, o_p + 3)
    plot(energies, c_d, energies, o_d + 2)
    xlabel('Energy - $E_f$ (eV)')
    ylabel('DOS')
    legend(['DOS',
            'C$_s$', 'O$_s$',
            'C$_p$', 'O$_p$',
            'C$_d$', 'O$_d$'],
           ncol=2,loc='best')
savefig('images/co-ados.png')
```

```
None
```

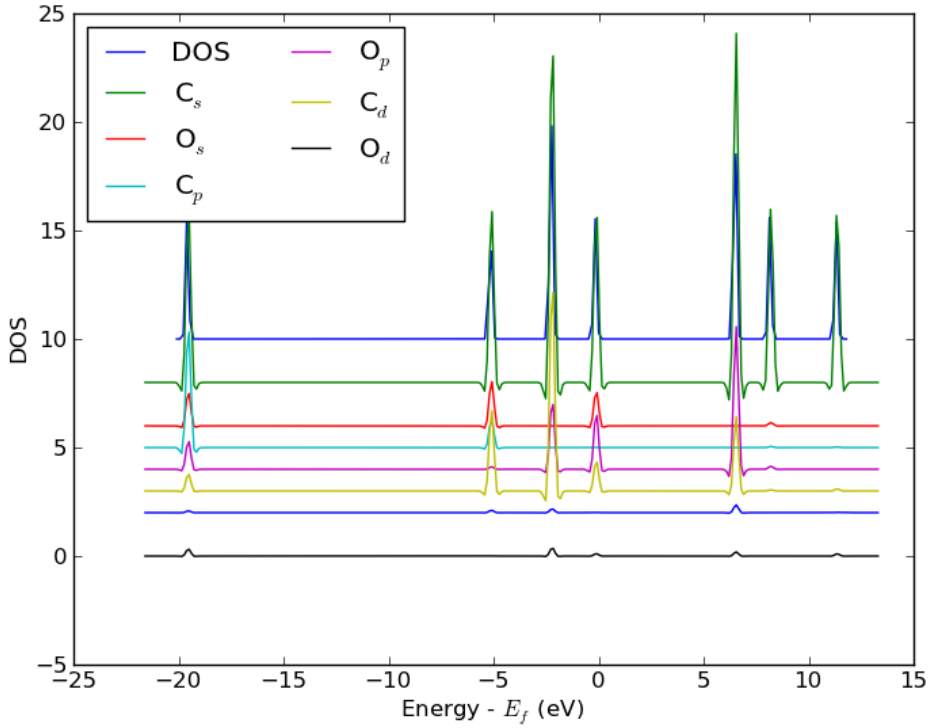


Figure 11: Atom-projected DOS for a CO molecule. The total density of states and the s, p and d states on the C and O are shown.

```
None
```

4.3.5 Visualizing electron density

The electron density is a $3d$ quantity: for every (x, y, z) point, there is a charge density. That means we need 4 numbers for each point: (x, y, z) and $\rho(x, y, z)$. Below we show an example (Figure 12) of plotting the charge density.

```
from enthought.mayavi import mlab
from jasp import *

with jasp('molecules/simple-co') as calc:
    x,y,z,cd = calc.get_charge_density()

mlab.contour3d(x,y,z,cd)
mlab.savefig('images/co-cd.png')
mlab.show()
```

```
None
```

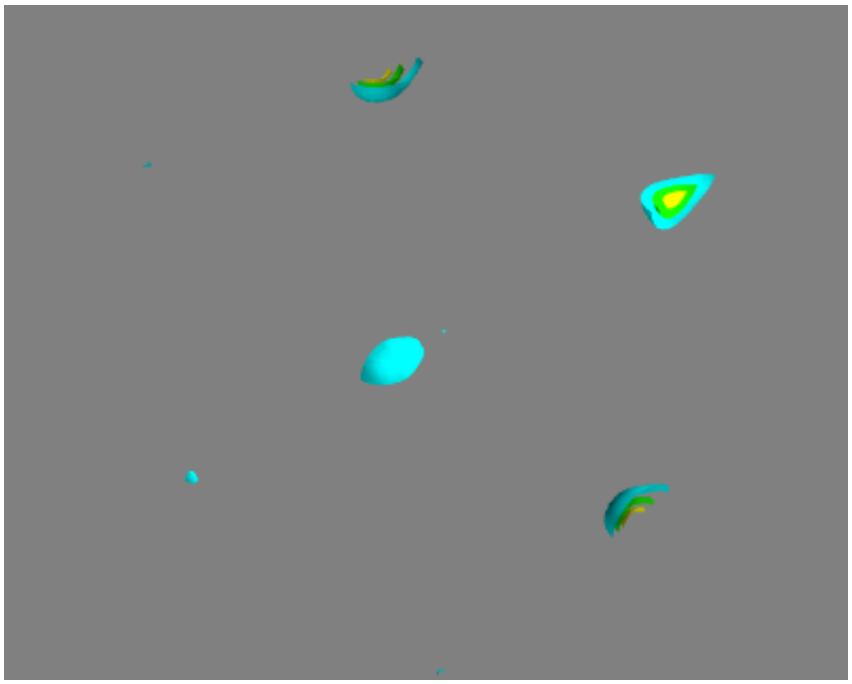


Figure 12: Charge density of a CO molecule.

```
from enthought.mayavi import mlab
from jasp import *

with jasp('molecules/co-centered') as calc:
    x,y,z,cd = calc.get_charge_density()

mlab.contour3d(x,y,z,cd)
mlab.savefig('images/co-centered-cd.png')
mlab.show()
```

```
None
```

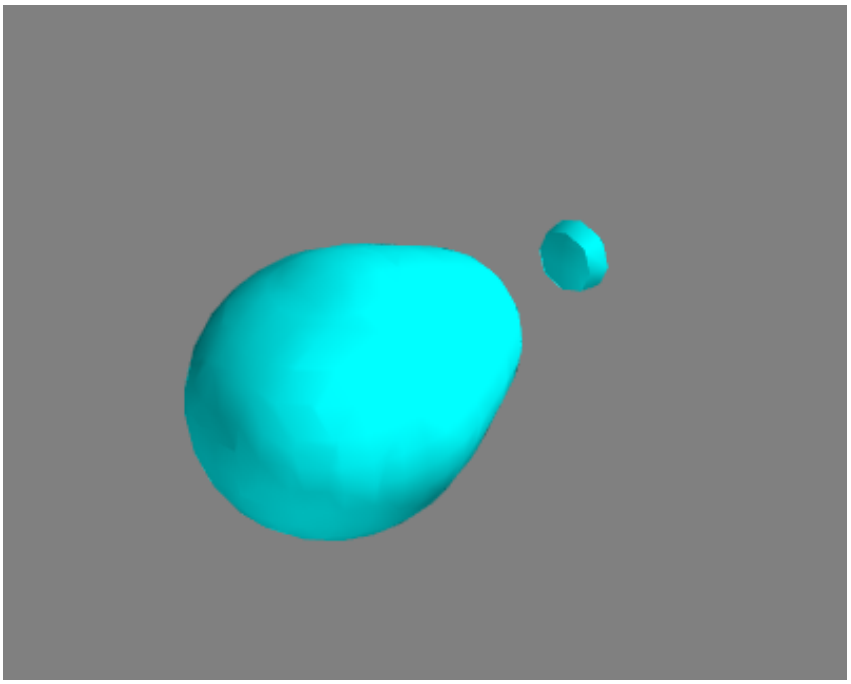


Figure 13: Charge density of a CO molecule centered in the unit cell.

4.3.6 Dipole moments

The dipole moment is a vector describing the separation of nuclear (positive) and electrical (negative) charge. Whether a molecule has a dipole moment or not depends on its symmetry. Below we compute the dipole moment of CO. We must integrate the electron density to find the center of electrical charge, and sum over the nuclei to find the center of positive charge.

```
from jasp import *
from ase.calculators.vasp import *
from ase.units import Debye
from ase import Atom, Atoms

atoms = Atoms([Atom('C',[2.422,0,0]),
               Atom('O',[3.578,0,0])],
              cell=(10,10,10))

atoms.center()

with jasp('molecules/co-centered',
          encut=350,
          xc='PBE',
          atoms=atoms) as calc:
    atoms.get_potential_energy()

    vcd = VaspChargeDensity()

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

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

```

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

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

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

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

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


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

    electron_dipole_moment = -electron_density_center*total_electron_charge

    # now the ion charge center. We only need the Zval listed in the potcar
    from jasp.POTCAR import get_ZVAL

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

    zval = {}
    for sym, ppath, hash in LOP:
        fullpath = os.path.join(ppp, ppath)
        z = get_ZVAL(fullpath)
        zval[sym] = z
    ion_charge_center = np.array([0.0, 0.0, 0.0])
    total_ion_charge = 0.0
    for atom in atoms:
        Z = zval[atom.symbol]
        total_ion_charge += Z
        pos = atom.position
        ion_charge_center += Z*pos

    ion_charge_center /= total_ion_charge
    ion_dipole_moment = ion_charge_center*total_ion_charge

    dipole_vector = (ion_dipole_moment + electron_dipole_moment)

    dipole_moment = ((dipole_vector**2).sum())**0.5/Debye

    print 'The dipole moment is {0:1.2f} Debye'.format(dipole_moment)
```

```
The dipole moment is 0.10 Debye
```

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

```
from jasp import *
from ase.units import Debye
```

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

```
The dipole moment is 0.10 Debye
```

4.3.7 Bader analysis

Bader analysis is a charge partitioning scheme where charge is divided by surfaces of zero flux that define atomic basins of charge. The most modern way of calculating the Bader charges is using the `bader` program from Graeme Henkelmen's group [15, 23]. Let us consider a water molecule, centered in a box. The strategy is first to run the calculation, then run the bader program on the results.

```
1  from jasp import *
2  from ase.structure import molecule
3  atoms = molecule('H2O')
4  atoms.center(vacuum=6)
5
6  with jasp('molecules/h2o-bader',
7            xc='PBE',
8            encut=350,
9            atoms=atoms) as calc:
10     calc.calculate()
11
12     os.system('bader -p all_atom -p atom_index CHG')
```

```
 GRID BASED BADER ANALYSIS  (Version 0.27e 05/09/11)

 OPEN ... CHG
 VASP-STYLE INPUT FILE
VASP5 format
 FFT-grid:   112 x 128 x 120
 CLOSE ... CHG

 RUN TIME:    0.73 SECONDS

 CALCULATING BADER CHARGE DISTRIBUTION
               0  10  25  50  75  100
 PERCENT DONE:  **********************

 REFINING AUTOMATICALLY
 ITERATION: 1
 EDGE POINTS:       1400048
 REASSIGNED POINTS:  111249
 ITERATION: 2
 CHECKED POINTS:     913184
 REASSIGNED POINTS:    1424
```

```
ITERATION: 3
CHECKED POINTS:      15784
REASSIGNED POINTS:     362
ITERATION: 4
CHECKED POINTS:       3190
REASSIGNED POINTS:     176
ITERATION: 5
CHECKED POINTS:       1228
REASSIGNED POINTS:      99
ITERATION: 6
CHECKED POINTS:        688
REASSIGNED POINTS:      21
ITERATION: 7
CHECKED POINTS:        267
REASSIGNED POINTS:       1
ITERATION: 8
CHECKED POINTS:         26
REASSIGNED POINTS:       0

RUN TIME:      27.04 SECONDS

CALCULATING MINIMUM DISTANCES TO ATOMS
               0  10  25  50  75  100
PERCENT DONE:  **********************

RUN TIME:    1.30 SECONDS

WRITING BADER ATOMIC CHARGES TO ACF.dat
WRITING BADER VOLUME CHARGES TO BCF.dat

NUMBER OF BADER MAXIMA FOUND:           5407
    SIGNIFICANT MAXIMA FOUND:             10
         NUMBER OF ELECTRONS:        7.99999

WRITING ATOMIC VOLUMES
               0  10  25  50  75  100
PERCENT DONE:  **********************

RUN TIME:    4.06 SECONDS

WRITING BADER ATOMIC INDEX TO AtIndex.dat
```

```
1 cat ACF.dat
```

```
   #         X           Y           Z        CHARGE     MIN DIST    ATOMIC VOL
--------------------------------------------------------------------------------
```

```
    1       6.0000       7.5265       6.0000       0.0007       1.6422     759.8654
    2       6.0000       6.0000       6.0000       0.0003       1.6422     640.1559
    3       6.0000       6.7632       6.5963       7.9990       2.2193     644.5831
--------------------------------------------------------------------------------
    VACUUM CHARGE:                0.0000
    VACUUM VOLUME:                0.0000
    NUMBER OF ELECTRONS:          8.0000
```

This script should work, but only if displacement is set to None or we convert the positions to Bohr. Otherwise, the positions do not match because of a Bohr/Angstrom units problem in an assertions statement. Note the complex sorting that has to be done, because the order of atoms in the Atoms object is not the same as in the POSCAR file.

```
from ase.io.bader import attach_charges
from ase.units import Bohr
from jasp import *

with jasp('molecules/h2o-bader') as calc:
    atoms = calc.get_atoms()

    symbols = np.array(atoms.get_chemical_symbols())[calc.sort]
    pos = atoms.positions[calc.sort]*Bohr
    newatoms = Atoms(symbols, positions=pos, cell=atoms.get_cell())

    attach_charges(newatoms, 'ACF.dat')

    print '#+tblname: bader'
    print '#+caption: Bader charges for a water molecule'
    print '| atom | Bader charge|'
    print '|-'
    for atom in newatoms:
        print '|{0} | {1} |'.format(atom.symbol, atom.charge)
```

Table 1: Bader charges for a water molecule

atom	Bader charge
H	0.9993
H	0.9997
O	0.001

4.4 Geometry optimization

4.4.1 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 to point out here. We want to compute 5 bond lengths, and each calculation is independent of all the others. `jasp` is setup to automatically handle jobs for you by submitting them to the queue. It raises a variety of exceptions to let you know what

has happened, and you must handle these to control the workflow. We will illustrate this by examples.

```
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 = []

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):
            energies.append(None)
            pass

if not None in energies:
    import matplotlib.pyplot as plt
    plt.plot(bond_lengths, energies, 'bo-')
    plt.xlabel('Bond length ($\AA$)')
    plt.ylabel('Total energy (eV)')
    plt.savefig('images/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.   ]]
```

To find the minimum we could run more calculations, but a simpler and faster way is to fit a polynomial to the data, and find the analytical minimum. The results are shown in Figure 14.

```
from jasp import *
import numpy as np
import matplotlib.pyplot as plt
np.set_printoptions(precision=3,suppress=True)

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

for d in bond_lengths: #possible bond lengths

    with jasp('molecules/co-{0}'.format(d)) as calc:
        atoms = calc.get_atoms()
        energies.append(atoms.get_potential_energy())

# Now we fit an equation - cubic polynomial
pp = np.polyfit(bond_lengths, energies, 3)
dp = np.polyder(pp)  # first derivative - quadratic

# we expect two roots from the quadratic eqn. These are where the
# first derivative is equal to zero.
roots = np.roots(dp)

# The minimum is where the second derivative is positive.
dpp = np.polyder(dp) # second derivative - line
secd = np.polyval(dpp, roots)

minV = roots[secd > 0]
minE = np.polyval(pp, minV)

print 'The minimum energy is {0} eV at V = {1} Ang^3'.format(minE,minV)

# plot the fit
x = np.linspace(1.05, 1.25)
fit = np.polyval(pp, x)

plt.plot(bond_lengths, energies, 'bo ')
plt.plot(x,fit, 'r-')
plt.plot(minV, minE, 'm* ')
plt.legend(['DFT','fit','minimum'], numpoints=1)
```

```
plt.xlabel('Bond length ($\AA$)')
plt.ylabel('Total energy (eV)')
plt.savefig('images/co-bondlengths.png')
plt.show()
```

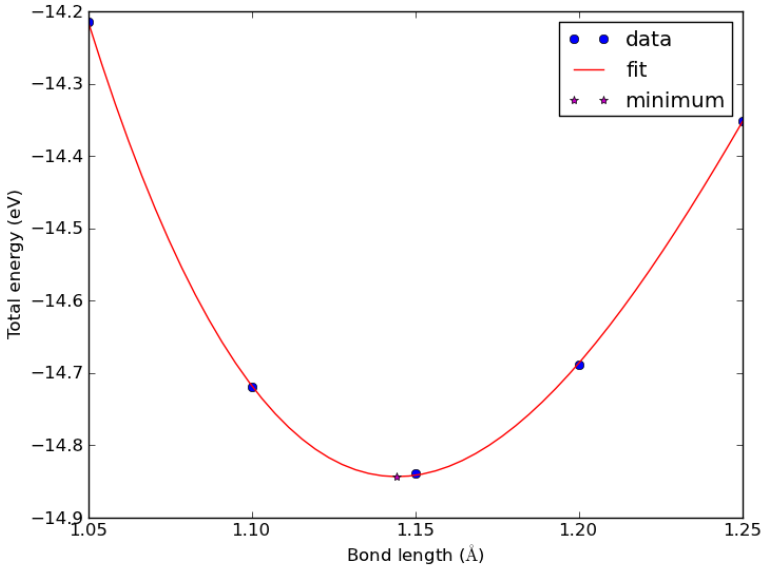


Figure 14: Energy vs CO bond length.

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.

```
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-cg',
          xc='PBE',
          nbands=6,
          encut=350,
          ismear=1,
          sigma=0.01, # this is small for a molecule
          ibrion=2,   # conjugate gradient optimizer
          nsw=5,      # do at least 5 steps to relax
          atoms=co) as calc:

    print 'Forces'
    print '======='
    print  co.get_forces()

    pos = co.get_positions()
```

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

```
Forces
=======
[[ 0.003  0.     0.   ]
 [-0.003  0.     0.   ]]
Bondlength = 1.14 angstroms
```

Relaxation of a water molecule It is not more complicated to relax more atoms, it just may take longer. Here we 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 "forces"
    print '======='
    print atoms.get_forces()
```

```
forces
=======
[[ 0.    -0.     0.   ]
 [-0.001  0.     0.   ]
 [ 0.     0.     0.   ]]
```

4.5 Vibrational frequencies

4.5.1 Manual calculation of vibrational frequency

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

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

$k = \frac{\partial^2 E}{\partial x^2}$ at the equilibrium bond length. We actually already have the data to do this from Manual determination. We only need to fit an equation to the energy vs. bond-length data, find the minimum energy bond-length, and then evaluate the second derivative of the fitted

function at the minimum. We will use a cubic polynomial for demonstration here. Polynomials are numerically convenient because they are easy to fit, and it is trivial to get the roots and derivatives of the polynomials, as well as to evaluate them at other points using `numpy.polyfit`, `numpy.polyder`, and `numpy.polyval`.

```
from jasp import *
from ase.units import *

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

for d in bond_lengths:
    with jasp('molecules/co-{0}'.format(d)) as calc:
        atoms = calc.get_atoms()
        energies.append(atoms.get_potential_energy())

# fit the data
pars = np.polyfit(bond_lengths, energies, 3)
xfit = np.linspace(1.05, 1.25)
efit = np.polyval(pars, xfit)

# first derivative
dpars = np.polyder(pars)
# find where the minimum is. chose the second one because it is the
# minimum we need.
print 'roots of first derivative are {0}'.format(np.roots(dpars))

d_min = 1.14425395 # we manually copy this from the output to here
# second derivative
ddpars = np.polyder(dpars)

#curvature at minimum = force constant
k = np.polyval(ddpars, d_min)/kg*s**2

# reduced mass
from ase.data import atomic_masses
C_mass = atomic_masses[6]/kg
O_mass = atomic_masses[8]/kg

mu = (C_mass*O_mass)/(C_mass + O_mass)

frequency = 1./(2.*np.pi)*np.sqrt(k/mu)
print 'The CO vibrational frequency is {0} cm^{{-1}}'.format(frequency/3e10)

import matplotlib.pyplot as plt
plt.plot(bond_lengths, energies, 'bo ')
plt.plot(xfit, efit, 'b-')
plt.xlabel('Bond length ($\AA$)')
plt.ylabel('Total energy (eV)')
plt.show()
```

```
roots of first derivative are [ 1.43384501  1.14425395]
The frequency is 2143.60766258 cm^{-1}
```

This result is in good agreement with experiment. The procedure used above is how many vibrational calculations are done. With more atoms, you have to determine a force constant matrix and diagonalize it. For more details, see [9]. In practice, we usually allow a packaged code to automate this, which we cover in Automated vibrational calculations

We now consider how much energy is in this vibration. This is commonly called zero-point energy (ZPE) and it is defined as $E_{ZPE} = \frac{1}{2}h\nu$ for a single mode, and h is Planck's constant

(4.135667516e-15 eV/s).

```
c = 3e10 # speed of light cm/s
h = 4.135667516e-15 # eV/s

nu = 2143.6076625*c # 1/s

E_zpe = 0.5*h*nu

print 'E_ZPE = {0:1.3f} eV'.format(E_zpe)
```

```
E_ZPE = 0.133 eV
```

This is a reasonable amount of energy! Zero-point energy increases with increasing vibrational frequency, and tends to be very important for small atoms.

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

4.5.2 Automated vibrational calculations

Vasp has built-in capability for performing vibrational calculations.

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

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 'Forces'
    print '======'
    print atoms.get_forces()

    # vibrational energies are in eV
    energies, modes = calc.get_vibrational_modes()
    print 'energies\n========'
    for i,e in enumerate(energies):
        print '{0:02d}: {1} eV'.format(i,e)
```

```
Forces
======
[[ 0.006 -0.022 -0.002]
```

```
 [-0.013  0.     0.003]
 [ 0.006  0.022 -0.002]]
energies
==========
00: 0.475622564 eV
01: 0.461551072 eV
02: 0.19626758 eV
03: 0.004419568 eV
04: 0.002650653 eV
05: (0.000307624+0j) eV
06: (0.011665989+0j) eV
07: (0.011977051+0j) eV
08: (0.015144825+0j) eV
```

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.

Zero-point energy for multiple modes For a molecule with lots of vibrational modes the zero-point energy is defined as the sum over all the vibrational modes:

$E_{ZPE} = \sum_i \frac{1}{2} h\nu_i$

Here is an example for water. Note we do not sum over the imaginary modes. We should also ignore the rotational and translational modes (some of those are imaginary, but some are just small).

```
from jasp import *
import numpy as np
c = 3e10 # speed of light cm/s
h = 4.135667516e-15 # eV/s

# first, get the frequencies.
with jasp('molecules/h2o_vib') as calc:
    freq = calc.get_vibrational_frequencies()

ZPE = 0.0
for f in freq:
    if not isinstance(f,float):
        continue #skip complex numbers
    nu = f*c # convert to frequency
    ZPE += 0.5*h*nu

```

```
print 'The ZPE of water is {0:1.3f} eV'.format(ZPE)

# one liner
ZPE = np.sum([0.5*h*f*c for f in freq if isinstance(f, float)])
print 'The ZPE of water is {0:1.3f} eV'.format(ZPE)
```

```
The ZPE of water is 0.571 eV
```

Note the zero-point energy of water is also fairly high (more than 0.5 eV). That is because of the high frequency O-H stretches.

4.6 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 'Forces'
    print '============================'
    print atoms.get_forces()
```

```
Forces
============================
[[ 0.     -0.      0.    ]
 [-0.001   0.      0.    ]
 [ 0.      0.      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 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.

See also [12]. For HREELS simulations see [18].

The shell script above has been translated to a convenient python function in `jasp`.

```
from jasp import *
with jasp('molecules/h2o_vib_dfpt') as calc:
    print 'mode  Relative intensity'
    for i, intensity in enumerate(calc.get_infrared_intensities()):
        print '{0:02d}     {1:1.3f}'.format(i, intensity)
```

```
mode  Relative intensity
00     0.227
```

```
01      0.006
02      0.312
03      1.000
04      0.006
05      0.000
06      0.005
07      0.000
08      0.345
```

4.7 Thermochemical properties of molecules

ase provides a thermochemistry module.

```
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 'vibrational energies\n====================='
    for i,e in enumerate(vib_energies):
        print '{0:02d}: {1} eV'.format(i,e)

# # 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.)
```

```
vibrational energies
```

```
====================
00: 0.29159234324 eV
01: 0.0169775241059 eV
02: 0.0169775241059 eV
03: 2.853610575e-09 eV
04: 8.68490175e-10 eV
05: 0.0 eV
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
=======================
```

4.8 Molecular reaction energies

4.8.1 O_2 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 the energy of an oxygen molecule and evaluate the

formula. Let us do that.

Simple estimate of O_2 dissociation energy

```
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)
```

```
O2 -> 2O  D = 8.521 eV
```

The answer we have obtained is way too high! Experimentally the dissociation energy is more like 5.2 eV (need reference).

We implicitly neglected spin-polarization in the example above. That could be a problem, since the O_2 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 us try again with spin polarization.

Estimating O_2 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)
```

```
Magnetic moment on O = 2.0000072 Bohr magnetons
Magnetic moment on O2 = 2.0000083 Bohr magnetons
O2 -> 2O  D = 6.668 eV
```

This is much closer to accepted literature values for the DFT-GGA O_2 dissociation energy. It is still more than 1 eV above an experimental value, but most of that error is due to the GGA exchange correlation functional. Some additional parameters that might need to be checked for convergence are the SIGMA value (it is probably too high for a molecule), as well as the cutoff energy. Oxygen is a "hard" atom that requires a high cutoff energy to achieve high levels of convergence.

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

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

with jasp('molecules/O2-sp-triplet') as calc:
    dos = DOS(calc, width=0.2)
    d_up = dos.get_dos(spin=0)
    d_down = dos.get_dos(spin=1)
    e = dos.get_energies()

ind = e <= 0.0
# integrate up to 0eV
print 'number of up states = {0}'.format(np.trapz(d_up[ind],e[ind]))
print 'number of down states = {0}'.format(np.trapz(d_down[ind],e[ind]))

import pylab as plt
plt.plot(e, d_up, e, -d_down)
plt.xlabel('energy [eV]')
plt.ylabel('DOS')
plt.legend(['up','down'])
plt.savefig('images/O2-sp-dos.png')
```

```
number of up states = 7.00000343617
number of down states = 5.00001715094
```

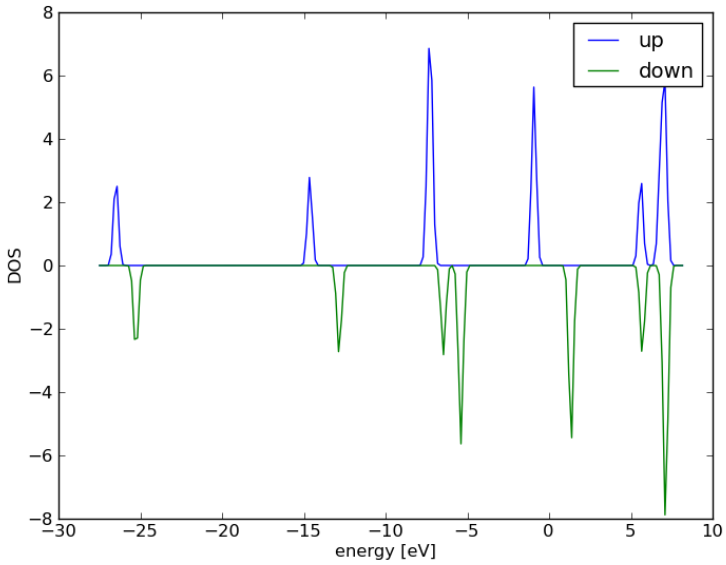


Figure 15: Spin-polarized density of states for the O_2 molecule.

You can see in Figure 15 that there are two different densities of states for the two spins. One has 7 electrons in it, and the other has 5 electrons in it. The difference of two electrons leads to the magnetic moment of 2 which we calculated earlier.

Convergence study of the O_2 dissociation energy

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

```
encuts = [250, 300, 350, 400, 450, 500, 550]

D = []
for encut in encuts:
    atoms = Atoms([Atom('O',[5,5,5],magmom=2)],
                  cell=(10,10,10))

    with jasp('molecules/O-sp-triplet-{0}'.format(encut),
              xc='PBE',
              encut=encut,
              ismear=0,
              ispin=2,
              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],magmom=1),
                   Atom('O',[6.22, 5,5],magmom=1)],
              cell=(10,10,10))

    with jasp('molecules/O2-sp-triplet-{0}'.format(encut),
              xc='PBE',
              encut=encut,
              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

    if None not in (E_O, E_O2):
        d = 2*E_O - E_O2
        D.append(d)
        print 'O2 -> 2O encut = {0}  D = {1:1.3f} eV'.format(encut, d)

import matplotlib.pyplot as plt
plt.plot(encuts, D)
plt.xlabel('ENCUT (eV)')
plt.ylabel('O$_2$ dissociation energy (eV)')
plt.savefig('images/O2-dissociation-convergence.png')
```

```
O2 -> 2O encut = 250  D = 6.696 eV
O2 -> 2O encut = 300  D = 6.727 eV
O2 -> 2O encut = 350  D = 6.708 eV
O2 -> 2O encut = 400  D = 6.668 eV
O2 -> 2O encut = 450  D = 6.650 eV
O2 -> 2O encut = 500  D = 6.648 eV
O2 -> 2O encut = 550  D = 6.650 eV
```

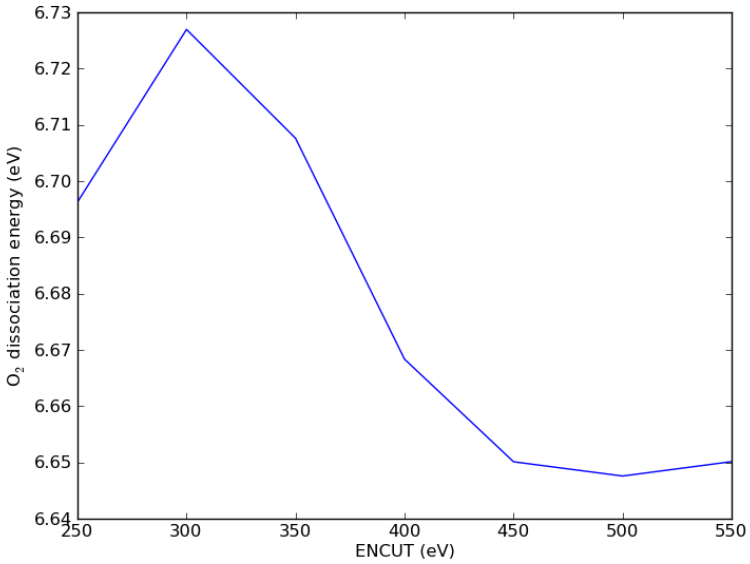


Figure 16: Convergence study of the O_2 dissociation energy as a function of ENCUT.

Based on these results (Figure 16), you could argue the dissociation energy is converged to about 2 meV at a planewave cutoff of 450 eV, and within 50 meV at 350 eV cutoff. You have to decide what an appropriate level of convergence is. Note that increasing the planewave cutoff significantly increases the computational time, so you are balancing level of convergence with computational speed. It would appear that planewave cutoff is not the cause for the discrepancy between our calculations and literature values.

```
encuts = [250, 300, 350, 400, 450, 500, 550]

for encut in encuts:
    OUTCAR = 'molecules/O2-sp-triplet-{0}/OUTCAR'.format(encut)
    f = open(OUTCAR, 'r')
    for line in f:
        if 'Total CPU time used (sec)' in line:
            print '{0}: {1}'.format(encut, line)
    f.close()
```

```
250:                  Total CPU time used (sec):       67.926

300:                  Total CPU time used (sec):      121.296

350:                  Total CPU time used (sec):      150.808

400:                  Total CPU time used (sec):      166.956

450:                  Total CPU time used (sec):      199.381

500:                  Total CPU time used (sec):      240.705
```

```
550:                    Total CPU time used (sec):      266.318
```

Illustration of the effect of sigma We were not careful in selecting a good value for SIGMA in the calculations above. The default value of SIGMA is 0.2, which may be fine for metals, but it is not correct for molecules. SIGMA is the broadening factor used to smear the electronic density of states at the Fermi level. For a metal with a continuous density of states, this appropriate, but for molecules with discrete energy states it does not make sense. We are somewhat forced to use the machinery designed for metals on molecules. The solution is to use a very small SIGMA. Ideally you would use SIGMA=0, but that is not practical for convergence reasons, so we try to find what is small enough.

Let us examine the effect of SIGMA on the dissociation energy here.

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

sigmas = [0.2, 0.1, 0.05, 0.02, 0.01, 0.001]

D = []
for sigma in sigmas:
    atoms = Atoms([Atom('O',[5, 5, 5], magmom=2)],
                  cell=(10,10,10))

    with jasp('molecules/O-sp-triplet-sigma-{0}'.format(sigma),
              xc='PBE',
              encut=400,
              ismear=0,
              sigma=sigma,
              ispin=2,
              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],magmom=1),
                   Atom('O',[6.22, 5, 5],magmom=1)],
              cell=(10,10,10))

    with jasp('molecules/O2-sp-triplet-sigma-{0}'.format(sigma),
              xc='PBE',
              encut=400,
              ismear=0,
              sigma=sigma,
              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

    if None not in (E_O, E_O2):
        d = 2*E_O - E_O2
        D.append(d)
        print 'O2 -> 2O sigma = {0}  D = {1:1.3f} eV'.format(sigma, d)

import matplotlib.pyplot as plt
```

```
plt.plot(sigmas, D, 'bo-')
plt.xlabel('SIGMA (eV)')
plt.ylabel('O$_2$ dissociation energy (eV)')
plt.savefig('images/O2-dissociation-sigma-convergence.png')
```

```
O2 -> 2O sigma = 0.2  D = 6.668 eV
O2 -> 2O sigma = 0.1  D = 6.746 eV
O2 -> 2O sigma = 0.05  D = 6.784 eV
O2 -> 2O sigma = 0.02  D = 6.807 eV
O2 -> 2O sigma = 0.01  D = 6.815 eV
O2 -> 2O sigma = 0.001  D = 6.822 eV
```

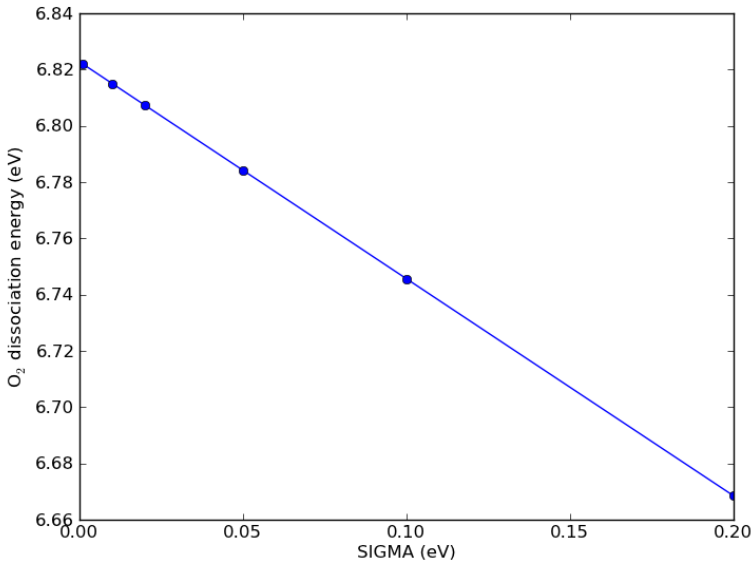


Figure 17: Effect of SIGMA on the oxygen dissociation energy.

Clearly SIGMA has an effect, but it does not move the dissociation energy closer to the literature values!

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. We will examine the total energy of an oxygen atom in a few geometries. First, let us consider variations of a square box.

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

# square box origin
atoms = Atoms([Atom('O',[0,0,0],magmom=2)],
              cell=(10,10,10))

with jasp('molecules/O-square-box-origin',
```

```
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Square box (origin): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass

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

with jasp('molecules/O-square-box-center',
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Square box (center): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass

# square box random
atoms = Atoms([Atom('O',[2.13, 7.32, 1.11],magmom=2)],
              cell=(10,10,10))

with jasp('molecules/O-square-box-random',
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Square box (random): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass
```

```
Square box (origin): E = -1.516623 eV
Square box (center): E = -1.516623 eV
Square box (random): E = -1.515359 eV
```

There is no difference of significance in these energies. The origin and center are identical (by symmetry too). The meV variation in the random position is probably only due to the atom being off the fft grid points a little bit.

Now, let us consider some non-square boxes.

```
# calculate O atom energy in orthorhombic boxes
from jasp import *
from ase import Atom, Atoms

# orthorhombic box origin
atoms = Atoms([Atom('O',[0,0,0],magmom=2)],
              cell=(8,9,10))

with jasp('molecules/O-orthorhombic-box-origin',
```

```
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Orthorhombic box (origin): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass

# orthrhombic box center
atoms = Atoms([Atom('O',[4, 4.5, 5],magmom=2)],
              cell=(8,9,10))

with jasp('molecules/O-orthorhombic-box-center',
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Orthorhombic box (center): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass

# orthorhombic box random
atoms = Atoms([Atom('O',[2.13, 7.32, 1.11],magmom=2)],
              cell=(8, 9, 10))

with jasp('molecules/O-orthorhombic-box-random',
            xc='PBE',
            encut=400,
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        print 'Orthorhombic box (random): E = {0} eV'.format(atoms.get_potential_energy())
    except (VaspSubmitted, VaspQueued):
        pass
```

```
Orthorhombic box (origin): E = -1.8941 eV
Orthorhombic box (center): E = -1.894124 eV
Orthorhombic box (random): E = -1.494285 eV
```

This is a surprisingly large difference in energy! Nearly 0.4 eV. This is precisely the amount of energy we were in disagreement with the literature values. Surprisingly, the "random" position is higher in energy, similar to the cubic boxes. Finally, we put this all together. We use a non-symmetric box for the O-atom

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

atoms = Atoms([Atom('O',[5.1, 4.2, 6.1],magmom=2)],
              cell=(8,9,10))

with jasp('molecules/O-sp-triplet-lowsym',
            xc='PBE',
            encut=400,
```

```
            ismear=0,
            sigma=0.01,
            ispin=2,
            atoms=atoms) as calc:
    try:
        E_O = atoms.get_potential_energy()
        print 'Magnetic moment on O = {0} Bohr  magnetons'.format(atoms.get_magnetic_moment())
    except (VaspSubmitted, VaspQueued):
        E_O = None

# 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,
            sigma=0.01,
            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()
        # verify magnetic moment
        print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())

    except (VaspSubmitted, VaspQueued):
        E_O2 = None

if None not in (E_O, E_O2):
    print 'E_O: ',E_O
    print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)
```

```
Magnetic moment on O = 2.0000016 Bohr  magnetons
Magnetic moment on O2 = 2.0000083 Bohr magnetons
E_O:  -1.893571
O2 -> 2O  D = 6.061 eV
```

This actually agrees within 30-50 meV of reported literature values. Note that with a different "random" position, we get the lower energy for the O atom. All the disagreement we had been seeing was apparently in the O atom energy. So, if you don't need D in your analysis, you will not see the error. Also note that this error is specific to there being a spherical atom in a symmetric cell. This is not a problem for most molecules, which are generally non-spherical.

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

```
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)
```

```
Magnetic moment on O = 1.9998232 Bohr magnetons
0.0
O2 -> 2O  D = 5.650 eV
```

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 'singlet: ',calc.get_atoms().get_potential_energy()

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

```
singlet:  -8.830101
triplet:  -9.848238
```

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

Verifying the magnetic moments on each atom It is one thing to see the total magnetic moment of a singlet state, and another to ask what are the magnetic moments on each atom. In VASP you must use LORBIT=11 to get the magnetic moments of the atoms written out.

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

with jasp('molecules/O2-sp-singlet') as calc:
    calc.clone('molecules/O2-sp-singlet-magmoms')

with jasp('molecules/O2-sp-singlet-magmoms') as calc:
    calc.set(lorbit=11)
    atoms = calc.get_atoms()
    magmoms = atoms.get_magnetic_moments()

    print 'singlet ground state'
    for i,atom in enumerate(atoms):
        print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
    print atoms.get_magnetic_moment()

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

with jasp('molecules/O2-sp-triplet-magmoms') as calc:
    calc.set(lorbit=11)
    atoms = calc.get_atoms()
    magmoms = atoms.get_magnetic_moments()
    print
    print 'triplet ground state'
    for i,atom in enumerate(atoms):
        print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
    print atoms.get_magnetic_moment()
```

```
singlet ground state
atom 0: magmom = 0.0
atom 1: magmom = 0.0
0.0

triplet ground state
atom 0: magmom = 0.815
atom 1: magmom = 0.815
2.0000084
```

Note the atomic magnetic moments do not add up to the total magnetic moment. The atomic magnetic moments are not really true observable properties. The moments are determined by a projection method that probably involves a spherical orbital, so the moments may be over or underestimated.

Using a different potential It is possible we need a higher quality potential to get the 6.02 eV value quoted by many. Here we try the O_sv potential, which treats the 1s electrons as valence electrons. Note however, the ENMIN in the POTCAR is very high!

```
grep ENMIN $VASP_PP_PATH/potpaw_PBE/O_sv/POTCAR
```

```
 ENMAX  = 1421.493; ENMIN  = 1066.119 eV
```

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

atoms = Atoms([Atom('O',[4, 4.5, 5],magmom=2)],
              cell=(8,9,10))

with jasp('molecules/O-sp-triplet-lowsym-sv',
          xc='PBE',
          ismear=0,
          ispin=2,
          sigma=0.01,
          setups={'O':'_sv'},
          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-sv',
          xc='PBE',
          ismear=0,
          sigma=0.01,
          ispin=2,  # turn spin-polarization on
          ibrion=2, # make sure we relax the geometry
          nsw=10,
          setups={'O':'_sv'},
          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)
```

4.8.2 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 *
```

```
3
4   # first we define our molecules. These will automatically be at the coordinates from the G2 database.
5
6   CO =  molecule('CO')
7   CO.set_cell([8,8,8], scale_atoms=False)
8
9   H2O = molecule('H2O')
10  H2O.set_cell([8,8,8], scale_atoms=False)
11
12  CO2 =  molecule('CO2')
13  CO2.set_cell([8,8,8], scale_atoms=False)
14
15  H2 = molecule('H2')
16  H2.set_cell([8,8,8], scale_atoms=False)
17
18  # now the calculators to get the energies
19  with jasp('molecules/wgs/CO',
20            xc='PBE',
21            encut=350,
22            ismear=0,
23            ibrion=2,
24            nsw=10,
25            atoms=CO) as calc:
26      try:
27          eCO = CO.get_potential_energy()
28      except (VaspSubmitted, VaspQueued):
29          eCO = None
30
31  with jasp('molecules/wgs/CO2',
32            xc='PBE',
33            encut=350,
34            ismear=0,
35            ibrion=2,
36            nsw=10,
37            atoms=CO2) as calc:
38      try:
39          eCO2 = CO2.get_potential_energy()
40      except (VaspSubmitted, VaspQueued):
41          eCO2 = None
42
43  with jasp('molecules/wgs/H2',
44            xc='PBE',
45            encut=350,
46            ismear=0,
47            ibrion=2,
48            nsw=10,
49            atoms=H2) as calc:
50      try:
51          eH2 = H2.get_potential_energy()
52      except (VaspSubmitted, VaspQueued):
53          eH2 = None
54
55  with jasp('molecules/wgs/H2O',
56            xc='PBE',
57            encut=350,
58            ismear=0,
59            ibrion=2,
60            nsw=10,
61            atoms=H2O) as calc:
62      try:
63          eH2O = H2O.get_potential_energy()
64      except (VaspSubmitted, VaspQueued):
65          eH2O = None
66
67  if None in (eCO2, eH2, eCO, eH2O):
68      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)
```

```
\Delta E = -0.720 eV
\Delta E = -16.614 kcal/mol
\Delta E = -69.514 kJ/mol
```

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.

```
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'
```

calculation summaries

CO

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/molecules/wgs/CO
  converged: True
  Energy = -14.789536 eV

  Unit cell vectors (angstroms)
        x       y       z      length
  a0 [ 8.000   0.000   0.000] 8.000
```

```
  a1 [ 0.000  8.000  0.000] 8.000
  a2 [ 0.000  0.000  8.000] 8.000
  a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0
  Unit cell volume = 512.000 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
             0.005  0.005  0.005 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
   0    O   [0.000      0.000      0.490]   0   0.01      T T T
   1    C   [0.000      0.000      7.346]   0   0.01      T T T
-----------------------------------------------

INCAR Parameters:
-----------------
        nbands: 9
        ismear: 0
           nsw: 10
        ibrion: 2
         encut: 350.0
          prec: Normal
          kpts: [1 1 1]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
C: potpaw_PBE/C/POTCAR (git-hash: 2272d6745da89a3d872983542cef1d18750fc952)
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
```

CO_2

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/molecules/wgs/CO2
  converged: True
  Energy = -22.959572 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 8.000  0.000  0.000] 8.000
  a1 [ 0.000  8.000  0.000] 8.000
  a2 [ 0.000  0.000  8.000] 8.000
  a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0
  Unit cell volume = 512.000 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
             0.009  0.009  0.008 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
```

```
   0    C   [0.000      0.000      0.000]   0   0.00      T T T
   1    O   [0.000      0.000      1.177]   0   0.01      T T T
   2    O   [0.000      0.000      6.823]   0   0.01      T T T
---------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 12
        ismear: 0
           nsw: 10
        ibrion: 2
         encut: 350.0
          prec: Normal
          kpts: [1 1 1]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
C: potpaw_PBE/C/POTCAR (git-hash: 2272d6745da89a3d872983542cef1d18750fc952)
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
```

$\mathbf{H_2}$

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/molecules/wgs/H2
  converged: True
  Energy = -6.744001 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 8.000  0.000  0.000] 8.000
  a1 [ 0.000  8.000  0.000] 8.000
  a2 [ 0.000  0.000  8.000] 8.000
  a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0
  Unit cell volume = 512.000 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
              0.000  0.000  0.000 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
   0    H   [0.000      0.000      0.376]   0   0.00      T T T
   1    H   [0.000      0.000      7.624]   0   0.00      T T T
---------------------------------------------------

INCAR Parameters:
-----------------
```

```
        nbands: 5
        ismear: 0
           nsw: 10
        ibrion: 2
         encut: 350.0
          prec: Normal
          kpts: [1 1 1]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
H: potpaw_PBE/H/POTCAR (git-hash: fbc0773b08b32f553234b0b50cc6ad6f5085c816)
```

H_2O

```
: ----------------------------
  VASP calculation from /home/jkitchin/dft-org/molecules/wgs/H2O
  converged: True
  Energy = -14.193569 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 8.000  0.000  0.000] 8.000
  a1 [ 0.000  8.000  0.000] 8.000
  a2 [ 0.000  0.000  8.000] 8.000
  a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0
  Unit cell volume = 512.000 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
             0.005  0.004  0.005 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
   0    O   [0.000      0.000      0.122]   0   0.06      T T T
   1    H   [0.000      0.766      7.522]   0   0.04      T T T
   2    H   [0.000      7.234      7.522]   0   0.04      T T T
-------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 8
        ismear: 0
           nsw: 10
        ibrion: 2
         encut: 350.0
          prec: Normal
          kpts: [1 1 1]
```

```
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
H: potpaw_PBE/H/POTCAR (git-hash: fbc0773b08b32f553234b0b50cc6ad6f5085c816)
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
```

4.8.3 Temperature dependent water gas shift equilibrium constant

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

CO vibrations

```
from jasp import *

# get relaxed geometry
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()
    for i,f in enumerate(vib_freq):
        print '{0:02d}: {1} cm^(-1)'.format(i,f)
```

```
00: 2115.528894 cm^(-1)
01: 60.594878 cm^(-1)
02: 60.594878 cm^(-1)
03: (0.987178+0j) cm^(-1)
04: (17.958586+0j) cm^(-1)
05: (17.958586+0j) cm^(-1)
```

CO has only one vibrational mode.

CO_2 vibrations

```
from jasp import *

# 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()
    for i,f in enumerate(vib_freq):
        print '{0:02d}: {1} cm^(-1)'.format(i,f)
```

```
00: 2352.901285 cm^(-1)
01: 1316.689504 cm^(-1)
02: 635.015913 cm^(-1)
03: 635.015913 cm^(-1)
04: (0.344306+0j) cm^(-1)
05: (1.763867+0j) cm^(-1)
06: (1.763867+0j) cm^(-1)
07: (62.700411+0j) cm^(-1)
08: (62.700411+0j) cm^(-1)
```

CO_2 is a linear molecule with 3N-5 = 4 vibrational modes. They are the first four frequencies in the output above.

H_2 vibrations

```
from jasp import *

# 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()
    for i,f in enumerate(vib_freq):
        print '{0:02d}: {1} cm^(-1)'.format(i,f)
```

```
00: 4281.917749 cm^(-1)
01: 129.146855 cm^(-1)
02: 129.146855 cm^(-1)
03: 0.0 cm^(-1)
04: 0.0 cm^(-1)
05: (1e-05+0j) cm^(-1)
```

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

H_2O vibrations

```
from jasp import *

# 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()
    for i,f in enumerate(vib_freq):
        print '{0:02d}: {1} cm^(-1)'.format(i,f)
```

```
00: 3782.062213 cm^(-1)
01: 3672.1246 cm^(-1)
02: 1586.23055 cm^(-1)
03: 135.82763 cm^(-1)
04: 16.280411 cm^(-1)
05: (0.208582+0j) cm^(-1)
06: (26.297061+0j) cm^(-1)
07: (106.869518+0j) cm^(-1)
08: (131.286732+0j) cm^(-1)
```

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 *
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)

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()
```

```
None
```

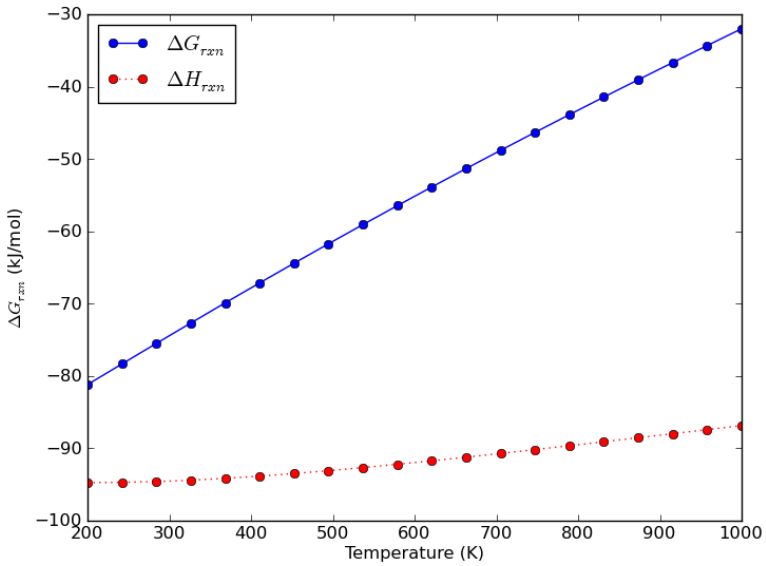
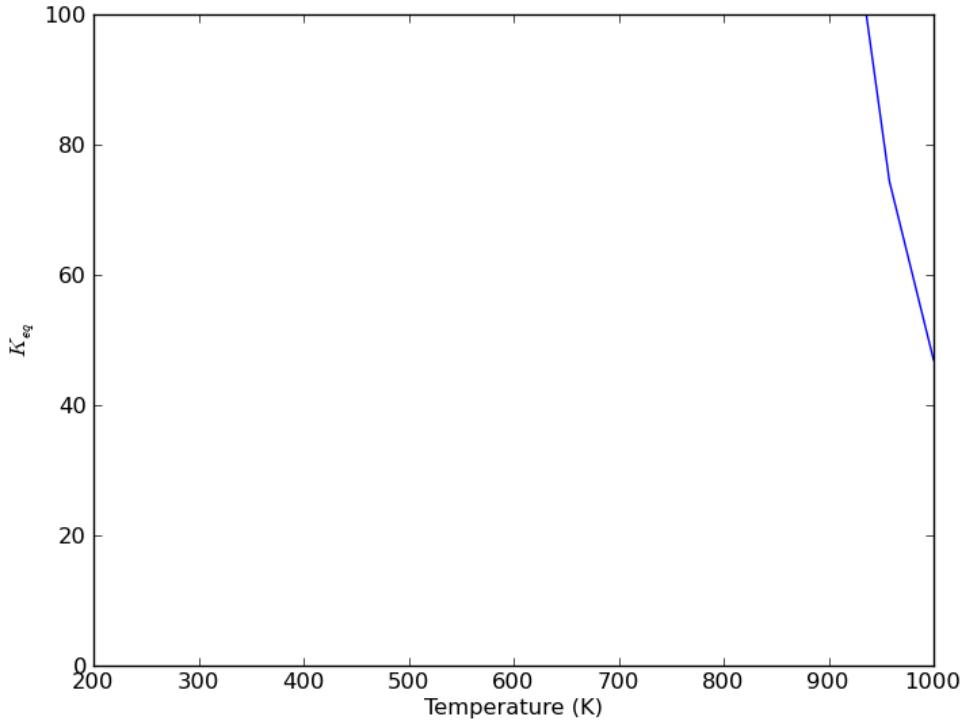


Figure 18: Thermodynamic energies of the water gas shift reaction as a function of temperature.

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.

4.9 Molecular reaction barriers

We will consider a simple example of the barrier for NH_3 inversion. We have to create an NH_3 molecule in the initial and inverted state (these have exactly the same energy), and then interpolate a band of images. Then, we use the NEB method to compute the barrier to inversion.

4.9.1 Get initial and final states

```
# compute initial and final states
from ase import Atoms
from ase.data.molecules import molecule
import numpy as np
from jasp import *
from ase.constraints import FixAtoms

atoms = molecule('NH3')
constraint = FixAtoms(mask=[atom.symbol == 'N' for atom in atoms])
atoms.set_constraint(constraint)

Npos = atoms.positions[0]

# move N to origin
atoms.translate(-Npos)
```

```
atoms.set_cell((10,10,10),scale_atoms=False)

atoms2 = atoms.copy()
pos2 = atoms2.positions

for i,atom in enumerate(atoms2):
    if atom.symbol == 'H':
        # reflect through z
        pos2[i] *= np.array([1,1,-1])
atoms2.positions = pos2

#now move N to center of box
atoms.translate([5,5,5])
atoms2.translate([5,5,5])

with jasp('molecules/nh3-initial',
          xc='PBE',
          encut=350,
          ibrion=1,
          nsw=10,
          atoms=atoms) as calc:
    try:
        calc.calculate()
    except (VaspSubmitted, VaspQueued):
        pass

with jasp('molecules/nh3-final',
          xc='PBE',
          encut=350,
          ibrion=1,
          nsw=10,
          atoms=atoms2) as calc:
    try:
        calc.calculate()
    except (VaspSubmitted, VaspQueued):
        pass
```

4.9.2 Run band calculation

Now we do the band calculation.

```
# Run NH3 NEB calculations
from jasp import *
from ase.neb import NEB

with jasp('molecules/nh3-initial') as calc:
    atoms = calc.get_atoms()

with jasp('molecules/nh3-final') as calc:
    atoms2 = calc.get_atoms()

# 5 images including endpoints
images = [atoms]
images += [atoms.copy() for i in range(3)]
images += [atoms2]

neb = NEB(images)
neb.interpolate()

with jasp('molecules/nh3-neb',
          xc='PBE',
          ibrion=1,
          nsw=90,
```

```
            spring=-5, debug=logging.DEBUG,
            atoms=images) as calc:
    images,energies = calc.get_neb()

    calc.plot_neb(show=False)
import matplotlib.pyplot as plt
plt.savefig('images/nh3-neb.png')
```

```
None
```

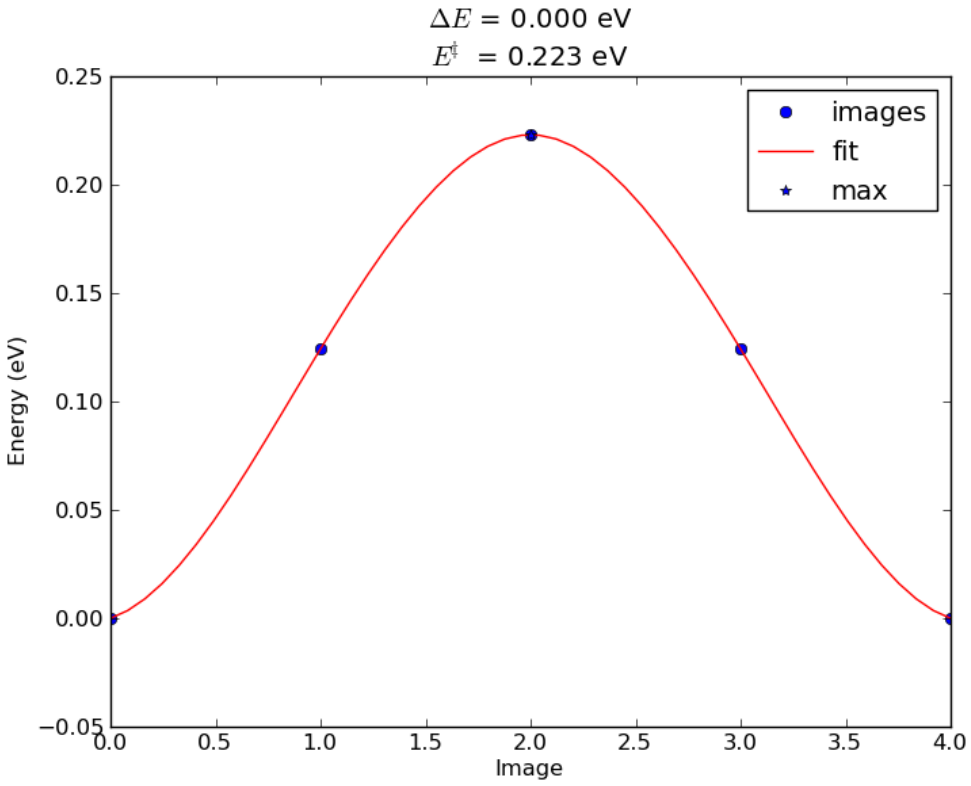


Figure 19: NEB for ammonia flipping.

4.9.3 Make a movie of the animation

```
# make neb movie
from ase.io import write
from ase.visualize import view
from jasp import *

with jasp('molecules/nh3-neb') as calc:
    images,energies = calc.get_neb()

# this rotates the atoms 90 degrees about the y-axis
[atoms.rotate('y', np.pi/2.) for atoms in images]

for i,atoms in enumerate(images):
    write('images/00{0}-nh3.png'.format(i), atoms, show_unit_cell=2)
```

```
14
15  # animated gif
16  os.system('convert -delay 50 -loop 0 images/00*-nh3.png nh3-neb.gif')
17
18  # Shockwave flash
19  os.system('png2swf -o nh3-neb.swf images/00*-nh3.png ')
```

5 Bulk systems

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

5.1 Defining and visualizing bulk systems

5.1.1 Built-in functions in ase

As with molecules, `ase` provides several helper functions to create bulk structures. We highlight a few of them here.

```
1   from ase.io import write
2   from ase.lattice.cubic import FaceCenteredCubic
3
4
5   atoms = 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  write('images/Ag-bulk.png',atoms,show_unit_cell=2)
13
14  # to make an alloy, we can replace one atom with another kind
15  atoms[0].symbol = 'Pd'
16  write('images/AgPd-bulk.png',atoms,show_unit_cell=2)
```

```
None
```

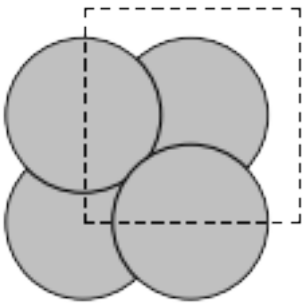


Figure 20: A simple fcc Ag bulk structure.

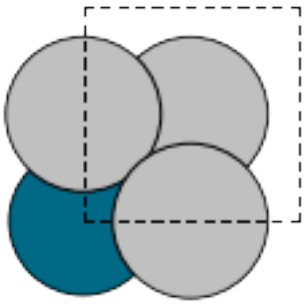


Figure 21: A simple Ag_3Pd bulk structure.

5.1.2 Using http://materialsproject.org

The Materials Project offers web access to a pretty large number of materials (over 21,000 at the time of this writing), including structure and other computed properties. You must sign up for an account at the website, and then you can access the information. You can search for materials with lots of different criteria including formula, unit cell formula, by elements, by structure, etc... The website allows you to download the VASP files used to create the calculations. They also develop the pymatgen project (which requires python 2.7+).

For example, I downloaded this cif file for a RuO_2 structure (Material ID 825).

```
#\#CIF1.1
##########################################################################
#               Crystallographic Information Format file
#               Produced by PyCifRW module
#
#  This is a CIF file.  CIF has been adopted by the International
#  Union of Crystallography as the standard for data archiving and
#  transmission.
#
#  For information on this file format, follow the CIF links at
#  http://www.iucr.org
##########################################################################

data_RuO2
_symmetry_space_group_name_H-M          'P 1'
_cell_length_a                          3.13970109
_cell_length_b                          4.5436378
_cell_length_c                          4.5436378
_cell_angle_alpha                       90.0
_cell_angle_beta                        90.0
_cell_angle_gamma                       90.0
_chemical_name_systematic               'Generated by pymatgen'
_symmetry_Int_Tables_number             1
_chemical_formula_structural            RuO2
_chemical_formula_sum                   'Ru2 O4'
_cell_volume                            64.8180127062
_cell_formula_units_Z                   2
loop_
  _symmetry_equiv_pos_site_id
  _symmetry_equiv_pos_as_xyz
   1  'x, y, z'

loop_
  _atom_site_type_symbol
  _atom_site_label
  _atom_site_symmetry_multiplicity
```

```
  _atom_site_fract_x
  _atom_site_fract_y
  _atom_site_fract_z
  _atom_site_attached_hydrogens
  _atom_site_B_iso_or_equiv
  _atom_site_occupancy
   O  O1  1  0.000000  0.694330  0.694330  0  .  1
   O  O2  1  0.500000  0.805670  0.194330  0  .  1
   O  O3  1  0.000000  0.305670  0.305670  0  .  1
   O  O4  1  0.500000  0.194330  0.805670  0  .  1
   Ru  Ru5  1  0.500000  0.500000  0.500000  0  .  1
   Ru  Ru6  1  0.000000  0.000000  0.000000  0  .  1
```

We can read this file in with `ase.io.read`

```
from ase.io import read, write

atoms = read('bulk/Ru2O4_1.cif')

write('images/Ru2O4.png', atoms, show_unit_cell=2)
```

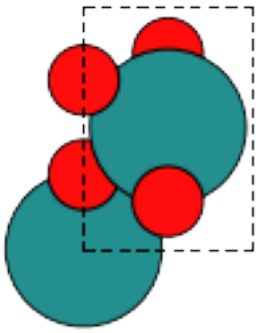


Figure 22: An RuO_2 unit cell prepared from a cif file.

You can also download the VASP files. I have copied these files (INCAR, POSCAR, KPOINTS) to a directory, and now we can run a calculation like this:

```
from jasp import *

with jasp('bulk/Ru2O4',
          xc='PBE',
          setups={'Ru':'_pv'}) as calc:
    atoms = calc.get_atoms()
    atoms.get_potential_energy()
    print calc
```

Some notes: we had to specify the PBE functional. That is good habit, since the default may be something else. We also had to specify the special setups used, since those were not defaults either.

We get exactly the same result as the website! Why is this helpful then? Well, we could use this as a starting point to look at an equation of state, or to plot some interesting electronic structure, etc... Or we can gain a little confidence that our version of VASP is working like the one they used.

5.2 Determining bulk structures

What we typically mean by determining bulk structures includes the following:

- What is the most stable crystal structure for a material?
- What is the lattice constant of fcc Cu?
- What are the lattice parameters and internal atom parameters for TiO_2?

All of these questions can often be addressed by finding the volume, shape and atomic positions that minimize the total energy of a bulk system. This is true at 0K. At higher temperatures, one must consider minimizing the free energy, rather than the internal energy.

5.2.1 fcc/bcc crystal structures

The fcc and bcc structures are simple. They only have one degree of freedom: the lattice constant. In this section we show how to calculate the equilibrium volume of each structure, and determine which one is more stable. We start with the fcc crystal structure of Cu. We will manually define the crystal structure based on the definitions in Kittel [21] (Chapter 1).

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

# fcc
LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
fcc_energies = []
for a in LC:
    atoms = Atoms([Atom('Cu',(0,0,0))],
                  cell=0.5*a*np.array([[1.0, 1.0, 0.0],
                                       [0.0, 1.0, 1.0],
                                       [1.0, 0.0, 1.0]]))

    with jasp('bulk/Cu-{0}'.format(a),
              xc='PBE',
              encut=350,
              kpts=(8,8,8),
              atoms=atoms) as calc:

        e = atoms.get_potential_energy()
        fcc_energies.append(e)

import matplotlib.pyplot as plt
plt.plot(LC, fcc_energies)
plt.xlabel('Lattice constant ($\AA$)')
plt.ylabel('Total energy (eV)')
plt.savefig('images/Cu-fcc.png')

print '#+tblname: cu-fcc-energies'
print '| lattice constant ($\AA$) | Total Energy (eV) |'
for lc, e in zip(LC,fcc_energies):
    print '| {0} | {1} |'.format(lc, e)
```

lattice constant ($\AA$)	Total Energy (eV)
3.5	-3.649238
3.55	-3.696204
3.6	-3.719946
3.65	-3.723951
3.7	-3.711284
3.75	-3.68426

Exercise 5.2
Use the data in the table above to plot the total energy as a function of the lattice constant. Fit a cubic polynomial to the data, and find the volume that minimizes the total energy.

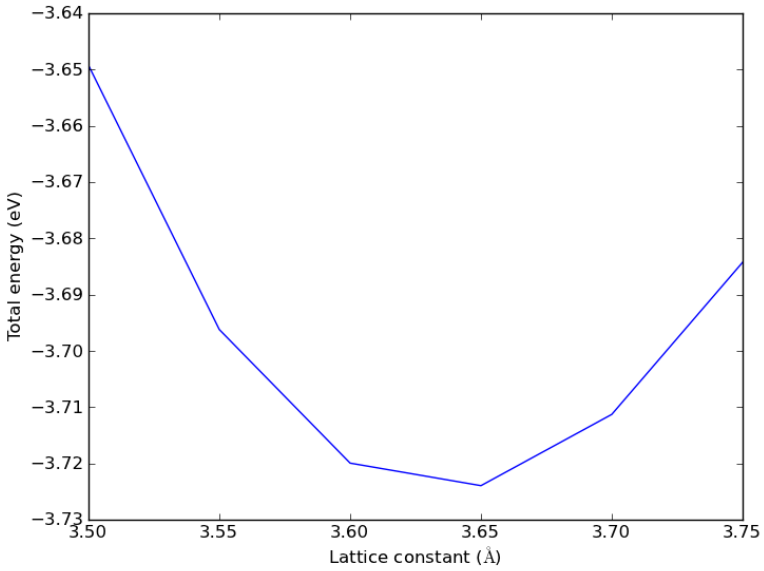


Figure 23: Total energy vs. fcc lattice contant for Cu. It appears the minimum is near 3.65 Å.

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

```
from jasp import *
from ase.utils.eos import *
LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
energies = []
volumes = []
for a in LC:
    with jasp('bulk/Cu-{0}'.format(a)) as calc:
        atoms = calc.get_atoms()
        volumes.append(atoms.get_volume())
        energies.append(atoms.get_potential_energy())

eos = EquationOfState(volumes,energies)
v0, e0, B = eos.fit()

```

```
print v0, e0, B

eos.plot('images/Cu-fcc-eos.png')
```

```
12.0167187475 -3.72468102446 0.858998381274
```

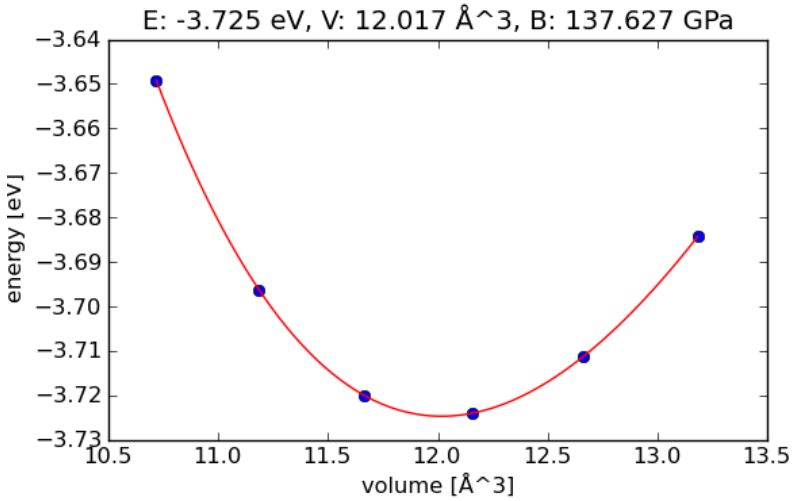


Figure 24: Total energy vs. volume for fcc Cu with fitted cubic polynomial equation of state.

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

```
from ase import Atom, Atoms
import numpy as np
a = 3.61 # lattice constant

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

print 'BCC lattice constant = {0} Ang'.format(a*(11.8/atoms.get_volume())**(1./3.))
```

```
BCC lattice constant = 2.86838428403 Ang
```

Now we run the equation of state calculations.

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

LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
```

```
5
6  for a in LC:
7      atoms = Atoms([Atom('Cu', [0,0,0])],
8                    cell=0.5*a*np.array([[1.0, 1.0, -1.0],
9                                         [-1.0, 1.0, 1.0],
10                                        [1.0, -1.0, 1.0]]))
11
12
13     with jasp('bulk/Cu-bcc-{0}'.format(a),
14               xc='PBE',
15               encut=350,
16               kpts=(8,8,8),
17               atoms=atoms) as calc:
18         calc.calculate()
```

Finally, we will compare the two crystal structures.

```
1  from jasp import *
2
3  bcc_LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
4  bcc_volumes = []
5  bcc_energies = []
6  for a in bcc_LC:
7      with jasp('bulk/Cu-bcc-{0}'.format(a)) as calc:
8          atoms = calc.get_atoms()
9          bcc_volumes.append(atoms.get_volume())
10         bcc_energies.append(atoms.get_potential_energy())
11
12 # fcc
13 fcc_LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
14 fcc_volumes = []
15 fcc_energies =[]
16 for a in fcc_LC:
17     with jasp('bulk/Cu-{0}'.format(a)) as calc:
18         atoms = calc.get_atoms()
19         fcc_volumes.append(atoms.get_volume())
20         fcc_energies.append(atoms.get_potential_energy())
21
22 import matplotlib.pyplot as plt
23 plt.plot(fcc_volumes, fcc_energies, label='fcc')
24 plt.plot(bcc_volumes, bcc_energies,label='bcc')
25
26 plt.xlabel('Atomic volume ($\AA^3$/atom)')
27 plt.ylabel('Total energy (eV)')
28 plt.legend()
29 plt.savefig('images/Cu-bcc-fcc.png')
30 plt.show()
31
32 # print table of data
33 print '#+tblname: bcc-data'
34 print '#+caption: Total energy vs. lattice constant for BCC Cu'
35 print '| Lattice constant (\AA$^3$) | Total energy (eV) |'
36 print '|-'
37 for lc,e in zip(bcc_LC, bcc_energies):
38     print '| {0} | {1} |'.format(lc, e)
```

Exercise 5.3
Use the data for FCC and BCC Cu to plot the total energy as a function of the lattice constant.

Table 2: Total energy vs. lattice constant for BCC Cu

Lattice constant (Å^3)	Total energy (eV)
2.75	-3.587523
2.8	-3.668112
2.85	-3.708699
2.9	-3.716496
2.95	-3.697781
3.0	-3.657913

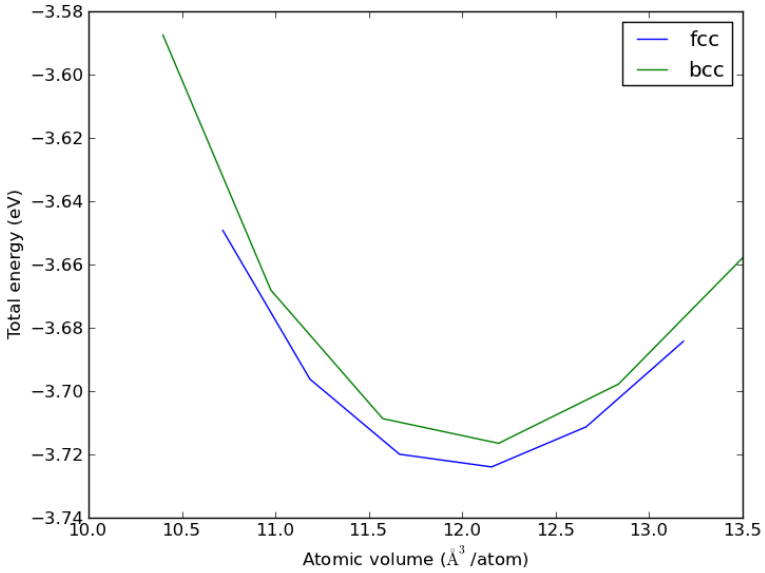


Figure 25: Comparison of energies between fcc and bcc Cu. The fcc structure is lower in energy.

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

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

Exercise 5.4
Compute the energy vs. volume for fcc and bcc Cu for different k-point grids. Determine when each result has converged, and which structure is more stable.

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

5.2.2 hcp

The hcp lattice is more complicated because there are two lattice parameters: a and c or equivalently: a and c/a. We will start by making a grid of values and find the set of parameters that minimizes the energy. See Figure 26.

```
from ase.lattice.hexagonal import HexagonalClosedPacked
from jasp import *
import matplotlib.pyplot as plt

atoms = HexagonalClosedPacked(symbol='Ru',
                              latticeconstant={'a':2.7, 'c/a':1.584})

a_list = [2.5, 2.6, 2.7, 2.8, 2.9]
covera_list = [1.4, 1.5, 1.6, 1.7, 1.8]

for a in a_list:
    energies = []
    for covera in covera_list:

        atoms = HexagonalClosedPacked(symbol='Ru',
                              latticeconstant={'a':a, 'c/a':covera})

        wd = 'bulk/Ru/{0:1.2f}-{1:1.2f}'.format(a,covera)

        with jasp(wd,
                  xc='PBE',
                  kpts=(6,6,4),
                  encut=350,
                  atoms=atoms) as calc:
            try:
                energies.append(atoms.get_potential_energy())
            except (VaspSubmitted, VaspQueued):
                pass
    plt.plot(covera_list, energies, label='a={0}'.format(a))

plt.xlabel('$c/a$ ($\AA$)')
plt.ylabel('Energy (eV)')
plt.legend()
plt.savefig('images/Ru-covera-scan.png')
```

It looks like there is a minimum in the a=2.7 Å curve, at a c/a ratio of about 1.6. We can look at the same data in a contour plot which shows more clearly there is minimum in all directions near that point (Figure 27).

```
from jasp import *
import matplotlib.pyplot as plt
from pylab import *

x = [2.5, 2.6, 2.7, 2.8, 2.9]
y = [1.4, 1.5, 1.6, 1.7, 1.8]

X,Y = meshgrid(x,y)
Z = np.zeros(X.shape)

for i,a in enumerate(x):
    for j,covera in enumerate(y):

        wd = 'bulk/Ru/{0:1.2f}-{1:1.2f}'.format(a,covera)

        with jasp(wd) as calc:
```

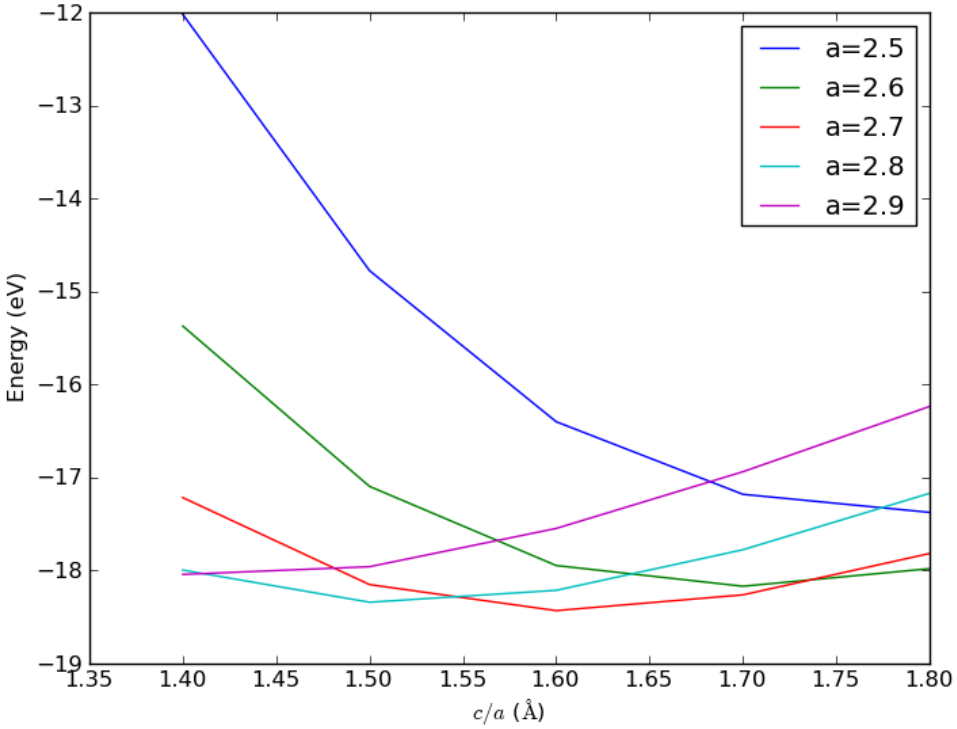


Figure 26: Total energy vs. c/a for different values of a.

```
            atoms = calc.get_atoms()
            try:
                Z[i][j] = atoms.get_potential_energy()
            except (VaspSubmitted, VaspQueued):
                pass

cf = plt.contourf(X,Y,Z,20,
                  cmap=cm.jet)

cbar = colorbar(cf)
cbar.ax.set_ylabel('Energy (eV)')

plt.xlabel('$a$ ($\AA$)')
plt.ylabel('$c/a$')

plt.legend()
plt.savefig('images/ru-contourf.png')
plt.show()
```

5.2.3 Complex structures with internal degrees of freedom

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

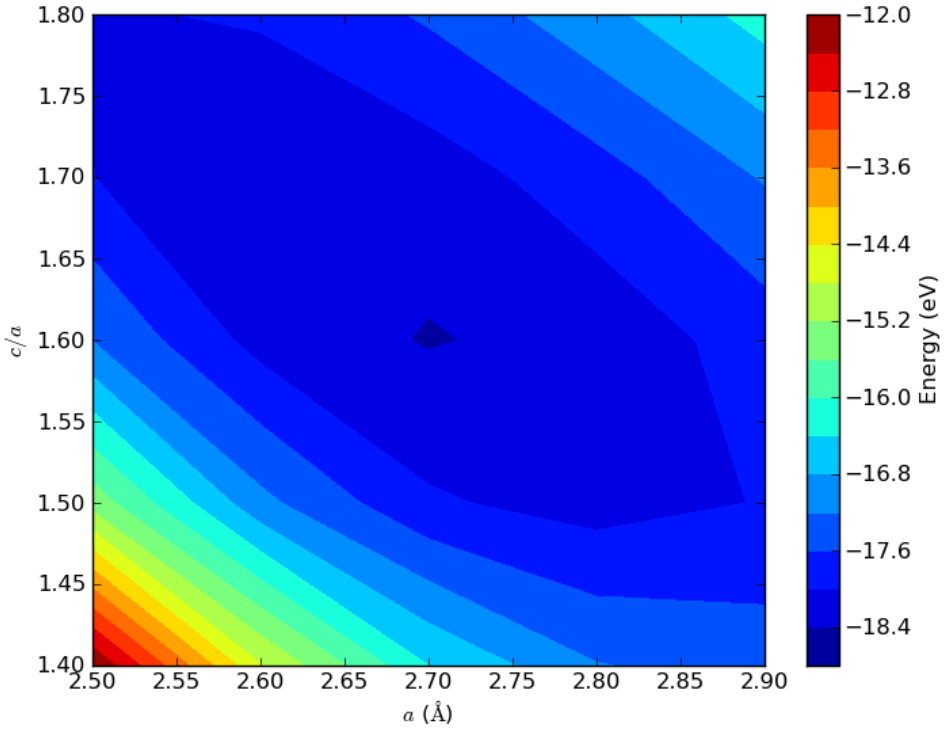


Figure 27: Contour plot of the total energy of hcp Ru for different values of a and c/a.

is usually not efficient to make a wild guess of the geometry and then turn VASP loose on to optimize it. Instead, the following algorithm works pretty well.

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

2. Perform a set of calculations at constant volume around the minimum from step 1, but the shape is allowed to change (ISIF=4)and the atoms allowed to relax.

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

This multistep process is pretty reasonable to get a converged structure pretty quickly. We will illustrate the process on TiO_2 which has two lattice parameters and one internal degree of freedom.

```
# step 1 frozen atoms and shape at different volumes
from ase import *
import numpy as np
from jasp import *
import matplotlib.pyplot as plt

'''
create a TiO2 structure from the lattice vectors at
```

```
http://cst-www.nrl.navy.mil/lattice/struk/c4.html
'''
a = 4.59 # experimental degrees of freedom.
c = 2.96
u = 0.3 #internal degree of freedom!

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

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

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

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

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

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

    with jasp('bulk/tio2/step1-{0:1.2f}'.format(f),
              encut=350,
              kpts=(4,4,4),
              xc='PBE',
              sigma=0.05,
              atoms=atoms) as calc:
        try:
            energies.append(atoms.get_potential_energy())
            volumes.append(atoms.get_volume())
        except (VaspSubmitted, VaspQueued):
            pass

plt.plot(volumes, energies)
plt.xlabel('Vol. ($\AA^3)$')
plt.ylabel('Total energy (eV)')
plt.savefig('images/tio2-step1.png')

print '#+tblname: tio2-vol-ene'
print '#+caption: Total energy of TiO_{2} vs. volume.'
print '| Volume ($\AA^3$) | Energy (eV) |'
print '|-'
for v,e in zip(volumes, energies):
    print '| {0} | {1} |'.format(v,e)
```

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

```
# step 2, constant volume, relax shape and atoms
from jasp import *
import matplotlib.pyplot as plt

with jasp('bulk/tio2/step1-1.00') as calc:
    atoms = calc.get_atoms()
```

Table 3: Total energy of TiO_2 vs. volume.

Volume ($\AA^3$)	Energy (eV)
56.1254184	-52.042193
59.2434972	-52.649555
62.361576	-52.949853
65.4796548	-52.950336
68.5977336	-52.783494

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

volumes = [62., 63., 64., 65., 66.]

energies = []
for v in volumes:
    atoms.set_cell((v/v0)**(1./3.)*cell0, scale_atoms=True)
    with jasp('bulk/tio2/step2-{0}'.format(v),
              encut=350,
              kpts=(4,4,4),
              xc='PBE',
              sigma=0.05,
              isif=4,     # relax ions, cell shape. not volume
              ibrion=2,
              nsw=10,
              atoms=atoms) as calc:
        try:
            energies.append(atoms.get_potential_energy())
        except (VaspSubmitted, VaspQueued):
            pass

plt.plot(volumes, energies)
plt.xlabel('Vol. ($\AA^3)$')
plt.ylabel('Total energy (eV)')
plt.savefig('images/tio2-step2.png')
```

```
None
```

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

```
# step 3 relax whole unit cell
from jasp import *

with jasp('bulk/tio2/step2-64.0') as calc:
    atoms = calc.get_atoms()

with jasp('bulk/tio2/step3',
          encut=350,
          kpts=(4,4,4),
          xc='PBE',
          sigma=0.05,
          isif=3,     # relax ions, cell shape, and volume
          ibrion=2,
          nsw=10,
          atoms=atoms) as calc:
        try:
            calc.calculate()
```

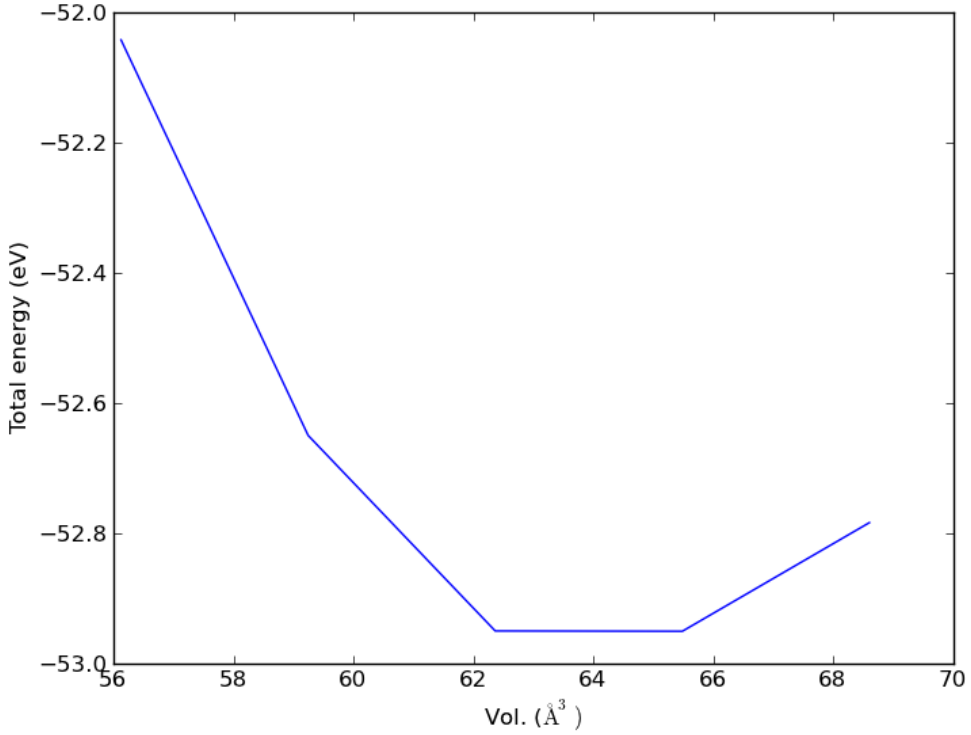


Figure 28: Total energy vs volume for TiO_2 with frozen atoms and frozen shape.

```
18            print calc
19        except (VaspSubmitted, VaspQueued):
20            pass
21
22 from pyspglib import spglib
23 print '\nThe spacegroup is {0}'.format(spglib.get_spacegroup(atoms))
```

```
: ----------------------------
  VASP calculation from /home/jkitchin/dft-org/bulk/tio2/step3
  converged: True
  Energy = -53.201939 eV

  Unit cell vectors (angstroms)
        x       y      z      length
  a0 [ 4.553  0.000  0.000] 4.553
  a1 [ 0.000  4.553  0.000] 4.553
  a2 [ 0.000  0.000  2.932] 2.932
  a,b,c,alpha,beta,gamma (deg): 4.553 4.553 2.932 90.0 90.0 90.0
  Unit cell volume = 60.759 Ang^3
  Stress (GPa):xx,    yy,    zz,    yz,    xz,    xy
            -0.001 -0.001  0.000 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce
```

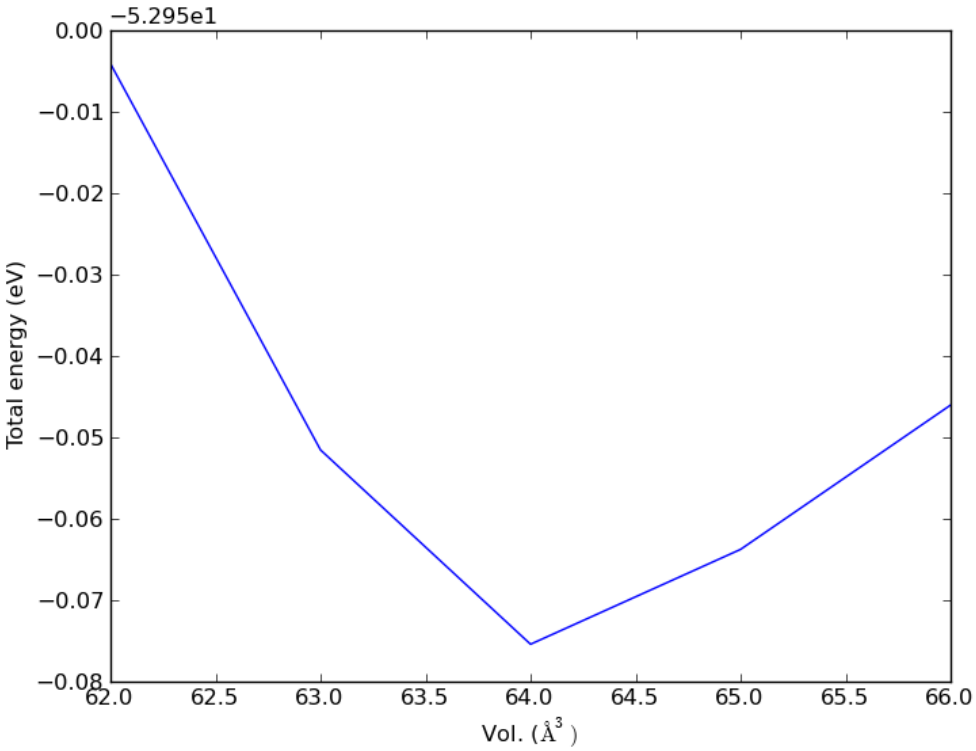


Figure 29: Total energy vs. volume of TiO_2 where shape and atoms are allowed to relax.

```
   0    Ti   [0.000      0.000      0.000]   0   0.00
   1    Ti   [2.276      2.276      1.466]   0   0.00
   2    O    [1.379      1.379      0.000]   0   0.00
   3    O    [3.174      3.174      0.000]   0   0.00
   4    O    [3.655      0.898      1.466]   0   0.00
   5    O    [0.898      3.655      1.466]   0   0.00
--------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 20
           nsw: 10
        ibrion: 2
          isif: 3
         encut: 350.0
         sigma: 0.05
          prec: Normal
          kpts: [4 4 4]
    reciprocal: False
            xc: PBE
```

```
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
Ti: potpaw_PBE/Ti/POTCAR (git-hash: a29a4f0f14083cf90544a3cf1ae7fc220b31829c)

The spacegroup is  (136)
```

The points to note here are the following:

1. The stress on the unit cell is very low.
2. The forces on the atoms are very small (less than 0.0 eV/Å).
3. The agreement with experimental lattice parameters is pretty good.
4. The spacegroup is still correct.

5.2.4 TODO the volume is very small (60) compared to what we expected (64) something may be wrong!

5.3 Bulk calculation parameters

5.3.1 Effect of SIGMA

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

Some rules to keep in mind:

1. The smearing methods were designed for metals. For molecules, semiconductors and insulators you should use a very small SIGMA (e.g. 0.01).
2. Standard values for metallic systems is SIGMA=0.1, but the best SIGMA may be material specific.

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

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

```

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

for sigma in SIGMA:

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

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

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

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

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

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

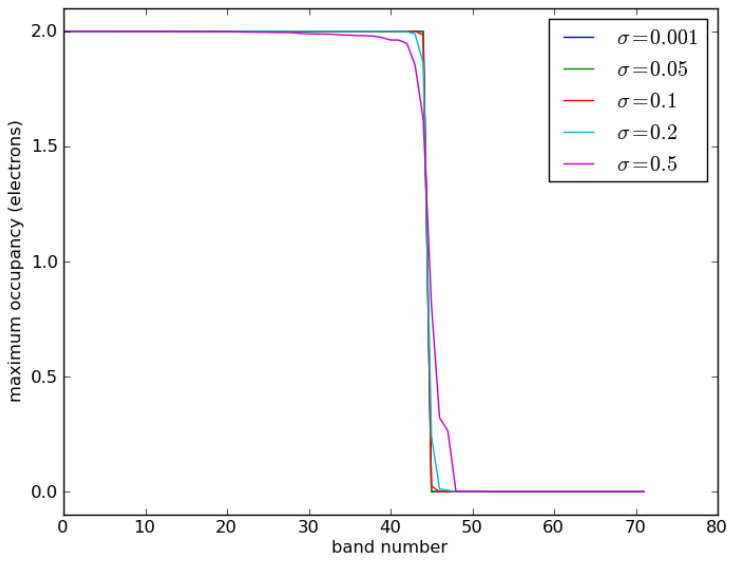


Figure 30: Effects of SIGMA on the occupancies of the Cu system.

5.4 Cohesive energy

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

```
from jasp import *
from ase.lattice.cubic import FaceCenteredCubic
from ase import Atoms, Atom
# bulk system
atoms = FaceCenteredCubic(directions=[[0,1,1],
                                      [1,0,1],
                                      [1,1,0]],
                                      size=(1,1,1),
                                      symbol='Rh')

with jasp('bulk/bulk-rh',
          xc='PBE',
          encut=350,
          kpts=(4,4,4),
          isif=3,
          ibrion=2,
          nsw=10,
          atoms=atoms) as calc:
    bulk_energy = atoms.get_potential_energy()

# atomic system
atoms = Atoms([Atom('Rh',[5,5,5])],
              cell=(7,8,9))

with jasp('bulk/atomic-rh',
          xc='PBE',
          encut=350,
          kpts=(1,1,1),
          atoms=atoms) as calc:
    atomic_energy = atoms.get_potential_energy()

cohesive_energy = atomic_energy - bulk_energy
print 'The cohesive energy is {0:1.3f} eV'.format(cohesive_energy)
```

```
The cohesive energy is 6.187 eV
```

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

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

```
from jasp import *

with jasp('bulk/bulk-rh') as calc:
    atoms = calc.get_atoms()


kpts = [3, 4, 6, 9, 12, 15, 18]

for k in kpts:
    with jasp('bulk/bulk-rh-kpts-{0}'.format(k),
          xc='PBE',
          encut=350,
```

```
            kpts=(k,k,k),
            atoms=atoms) as calc:
        e = atoms.get_potential_energy()

    print '({0:2d}, {0:2d}, {0:2d}): total energy = {1} eV'.format(k,e)
```

```
( 3,  3,  3): total energy = -5.891945 eV
( 4,  4,  4): total energy = -7.244046 eV
( 6,  6,  6): total energy = -7.265141 eV
( 9,  9,  9): total energy = -7.263049 eV
(12, 12, 12): total energy = -7.26984 eV
(15, 15, 15): total energy = -7.269792 eV
(18, 18, 18): total energy = -7.270505 eV
```

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

```
from jasp import *
from ase.lattice.cubic import FaceCenteredCubic
from ase import Atoms, Atom
# bulk system
atoms = FaceCenteredCubic(directions=[[0,1,1],
                                      [1,0,1],
                                      [1,1,0]],
                                      size=(1,1,1),
                                      symbol='Rh')

with jasp('bulk/bulk-rh',
          xc='PBE',
          encut=350,
          kpts=(4,4,4),
          isif=3,
          ibrion=2,
          nsw=10,
          atoms=atoms) as calc:
    bulk_energy = atoms.get_potential_energy()

# atomic system
atoms = Atoms([Atom('Rh',[5,5,5],magmom=1)],
              cell=(7,8,9))

with jasp('bulk/atomic-rh-sp',
          xc='PBE',
          encut=350,
          kpts=(1,1,1),
          ispin=2,
          atoms=atoms) as calc:
    atomic_energy = atoms.get_potential_energy()

cohesive_energy = atomic_energy - bulk_energy
print 'The cohesive energy is {0:1.3f} eV'.format(cohesive_energy)
```

```
The cohesive energy is 5.887 eV
```

Again, the value in Kittel [21] is 5.75 eV which is very close to this value. Finally, it is also possible there is a lower energy non-spherical atom energy; we did not check that at all (see Estimating triplet oxygen dissociation energy with low symmetry).

5.5 Effect of pressure on phase stability

So far we have only considered relative stability at a pressure of 0 Pa. We now consider the relative stability of two phases under pressure. We will consider TiO_2 in the rutile and anatase phases.

The pressure is defined by: $P = -\left(\frac{\partial E}{\partial V}\right)_T$. So if we have an equation of state $E(V)$ we can calculate the pressure at any volume, or alternatively, given a pressure, compute the volume. Pressure can affect the energy of two phases differently, so that one may become stable under pressure. The condition where a phase transition occurs is when the pressure in the two phases is the same, which occurs at a common tangent.

To show this, we need $E_{rutile}(V)$ and $E_{anatase}(V)$.

```
# run the rutile calculations
from jasp import *
from ase import Atom, Atoms

B='Ti'; X='O'; a=4.59; c=2.958; u=0.305;
'''
create a rutile structure from the lattice vectors at
http://cst-www.nrl.navy.mil/lattice/struk/c4.html

spacegroup: 136 P4_2/mnm
'''
a1 = a*np.array([1.0, 0.0, 0.0])
a2 = a*np.array([0.0, 1.0, 0.0])
a3 = c*np.array([0.0, 0.0, 1.0])

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

nTiO2 = len(atoms)/3.
v0 = atoms.get_volume()
cell0 = atoms.get_cell()

volumes = [28., 30., 32., 34., 36.]  #vol of one TiO2

for v in volumes:
    atoms.set_cell(cell0*((nTiO2*v/v0)**(1./3.)), scale_atoms=True)

    with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v),
              encut=350,
              kpts=(6,6,6),
              xc='PBE',
              ismear=0,
              sigma=0.001,
              isif=2,
              ibrion=2,
              nsw=20,
              atoms=atoms) as calc:
        try:
            calc.calculate()
        except (VaspSubmitted, VaspQueued):
            pass
```

```
# run the anatase calculations
from jasp import *
```

```
from ase import Atom, Atoms
# http://cst-www.nrl.navy.mil/lattice/struk/c5.html

B='Ti'; X='O'; a=3.7842; c=2*4.7573; z=0.0831;

a1 = a*np.array([1.0, 0.0, 0.0])
a2 = a*np.array([0.0, 1.0, 0.0])
a3 = np.array([0.5*a, 0.5*a, 0.5*c])

atoms = Atoms([Atom(B, -0.125*a1 + 0.625*a2 + 0.25*a3),
               Atom(B,  0.125*a1 + 0.375*a2 + 0.75*a3),
               Atom(X, -z*a1 + (0.25-z)*a2 + 2.*z*a3),
               Atom(X, -(0.25+z)*a1 + (0.5-z)*a2 + (0.5+2*z)*a3),
               Atom(X, z*a1 - (0.25 - z)*a2 + (1-2*z)*a3),
               Atom(X, (0.25 + z)*a1 + (0.5 + z)*a2 + (0.5-2*z)*a3)],
               cell=[a1,a2,a3])

nTiO2 = len(atoms)/3.
v0 = atoms.get_volume()
cell0 = atoms.get_cell()

volumes = [30., 33., 35., 37., 39.]  #vol of one TiO2

for v in volumes:
    atoms.set_cell(cell0*((nTiO2*v/v0)**(1./3.)), scale_atoms=True)

    with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v),
              encut=350,
              kpts=(6,6,6),
              xc='PBE',
              ismear=0,
              sigma=0.001,
              isif=2,
              ibrion=2,
              nsw=20,
              atoms=atoms) as calc:
        try:
            calc.calculate()
        except (VaspSubmitted, VaspQueued):
            pass
```

Now we will fit cubic polynomials to the data.

```
# fit cubic polynomials to E(V) for rutile and anatase
from jasp import *
import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(precision=2)

# anatase equation of stte
volumes = [30., 33., 35., 37., 39.]  #vol of one TiO2
a_volumes, a_energies = [], []
for v in volumes:
    with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v)) as calc:
        atoms = calc.get_atoms()
        nTiO2 = len(atoms)/3.0
        a_volumes.append(atoms.get_volume()/nTiO2)
        a_energies.append(atoms.get_potential_energy()/nTiO2)

# rutile equation of state
volumes = [28., 30., 32., 34., 36.]  #vol of one TiO2
r_volumes, r_energies = [], []
for v in volumes:
    with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v)) as calc:
```

```
        atoms = calc.get_atoms()
        nTiO2 = len(atoms)/3.0
        r_volumes.append(atoms.get_volume()/nTiO2)
        r_energies.append(atoms.get_potential_energy()/nTiO2)

# cubic polynomial fit to equation of state E(V) = pars*[V^3 V^2 V^1 V^0]
apars = np.polyfit(a_volumes, a_energies, 3)
rpars = np.polyfit(r_volumes, r_energies, 3)

print 'E_anatase(V) = {0:1.2f}*V^3 + {1:1.2f}*V^2 + {2:1.2f}*V + {3:1.2f}'.format(*apars)
print 'E_rutile(V) =  {0:1.2f}*V^3 + {1:1.2f}*V^2 + {2:1.2f}*V + {3:1.2f}'.format(*rpars)
print 'anatase epars: {0!r}'.format(apars)
print 'rutile epars: {0!r}'.format(rpars)
# get pressure parameters P(V) = -dE/dV
dapars = -np.polyder(apars)
drpars = -np.polyder(rpars)

print 'anatase ppars: {0!r}'.format(dapars)
print 'rutile ppars: {0!r}'.format(drpars)

print
print 'P_anatase(V) = {0:1.2f}*V^2 + {1:1.2f}*V + {2:1.2f}'.format(*dapars)
print 'P_rutile(V) =  {0:1.2f}*V^2 + {1:1.2f}*V + {2:1.2f}'.format(*drpars)

vfit = np.linspace(28,40)

# plot the equations of state
plt.plot(a_volumes, a_energies,'bo ', label='Anatase')
plt.plot(vfit, np.polyval(apars, vfit), 'b-')

plt.plot(r_volumes, r_energies,'gs ', label='Rutile')
plt.plot(vfit, np.polyval(rpars, vfit), 'g-')

plt.xlabel('Volume ($\AA^3$/f.u.)')
plt.ylabel('Total energy (eV/f.u.)')
plt.legend()
plt.xlim([25,40])
plt.ylim([-27, -26])
plt.savefig('images/rutile-anatase-eos.png')
```

```
E_anatase(V) = -0.00*V^3 + 0.13*V^2 + -5.23*V + 42.47
E_rutile(V) =  -0.00*V^3 + 0.14*V^2 + -5.33*V + 38.59
anatase epars: array([ -0.  ,   0.13,  -5.23,  42.47])
rutile epars: array([ -0.  ,   0.14,  -5.33,  38.59])
anatase ppars: array([ 0.  , -0.26,  5.23])
rutile ppars: array([ 0.  , -0.29,  5.33])

P_anatase(V) = 0.00*V^2 + -0.26*V + 5.23
P_rutile(V) =  0.00*V^2 + -0.29*V + 5.33
```

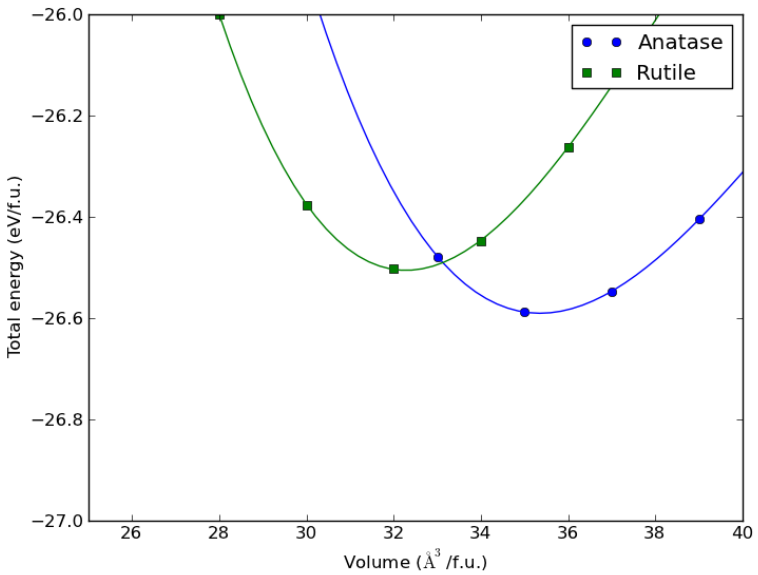


Figure 31: Equations of state (E(V)) for anatase and rutile TiO_2.

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

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

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

This is a nonlinear algebra problem. We use the `scipy.optimize.fsolve` to solve this problem.

```
from ase.units import GPa
from numpy import array, linspace, polyval

anatase_epars = array([ -1.06049246e-03,   1.30279404e-01,  -5.23520055e+00,
         4.25202869e+01])
rutile_epars = array([ -1.24680208e-03,   1.42966536e-01,  -5.33239733e+00,
         3.85903670e+01])

anatase_ppars = array([  3.18147737e-03,  -2.60558808e-01,   5.23520055e+00])
rutile_ppars = array([  3.74040625e-03,  -2.85933071e-01,   5.33239733e+00])

def func(V):
    V1 = V[0] # rutile volume
    V2 = V[1] # anatase volume

    E_rutile = polyval(rutile_epars,V1)
    E_anatase = polyval(anatase_epars,V2)

    P_rutile =  polyval(rutile_ppars,V1)
    P_anatase = polyval(anatase_ppars,V2)

    return [(E_anatase - E_rutile)/(V1-V2) - P_anatase,
            (E_anatase - E_rutile)/(V1-V2) - P_rutile]

from scipy.optimize import fsolve
x0 = fsolve(func,[28,34])
print 'The solutions are at V = {0}'.format(x0)
```

```
print 'Anatase pressure: {0} GPa'.format(polyval(anatase_ppars,x0[1])/GPa)
print 'Rutile  pressure: {0} GPa'.format(polyval(rutile_ppars,x0[0])/GPa)

# illustrate the common tangent
import matplotlib.pyplot as plt

vfit = linspace(28,40)
plt.plot(vfit, polyval(anatase_epars,vfit),label='anatase')
plt.plot(vfit, polyval(rutile_epars,vfit),label='rutile')
plt.plot(x0, [polyval(rutile_epars,x0[0]),
              polyval(anatase_epars,x0[1])], 'ko-', label='common tangent')
plt.legend()
plt.xlabel('Volume ($\AA^3$/f.u.)')
plt.ylabel('Total energy (eV/f.u.)')
plt.savefig('images/eos-common-tangent.png')
```

```
The solutions are at V = [ 31.67490656  34.60893508]
Anatase pressure: 4.52495142656 GPa
Rutile  pressure: 4.5249514267 GPa
```

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

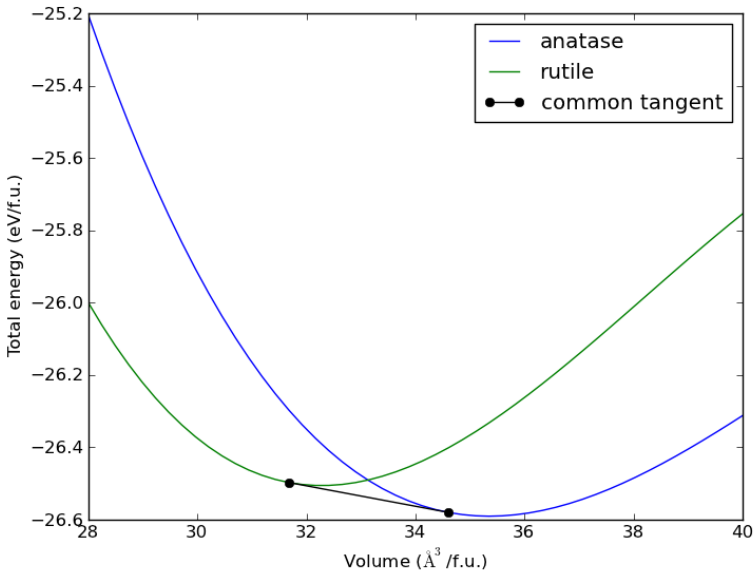


Figure 32: Illustration of the common tangent that shows the pressure where anatase and rutile coexist before anatase converts to rutile.

5.6 Bulk reaction energies

5.6.1 Alloy formation energies

In this section we will consider how to calculate the formation energy of an fcc Cu-Pd alloy and how to use that information to discuss relative stabilities. First, we compute the bulk energies of fcc Cu and fcc Pd.

Basic alloy formation energy

```
# get bulk Cu and Pd energies.
from jasp import *

from ase import Atom, Atoms

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

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

atoms = Atoms([Atom('Pd',  [0.000,      0.000,      0.000])],
              cell=[[ 1.978,  0.000,  1.978],
                    [ 1.978,  1.978,  0.000],
                    [0.000,  1.978,  1.978]])

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

print 'Cu energy = {0} eV'.format(cu)
print 'Pd energy = {0} eV'.format(pd)
```

```
Cu energy = -3.723305 eV
Pd energy = -5.183823 eV
```

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

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

atoms = Atoms([Atom('Cu',  [0.000,      0.000,      0.000]),
               Atom('Pd',  [-1.652,     0.000,      2.039])],
              cell=  [[ 0.000, -2.039,  2.039],
                      [ 0.000,  2.039,  2.039],
                      [ -3.303,  0.000,  0.000]])

with jasp('bulk/alloy/cupd-1',
          xc='PBE',
          encut=350,
```

```
            kpts=(12,12,8),
            nbands=17,
            ibrion=2,
            isif=4,
            nsw=10,
            atoms=atoms) as calc:
    cupd1 = atoms.get_potential_energy()


atoms = Atoms([Atom('Cu',  [-0.049,     0.049,      0.049]),
               Atom('Cu',  [-11.170,   11.170,     11.170]),
               Atom('Pd',  [-7.415,     7.415,      7.415]),
               Atom('Pd',  [-3.804 ,    3.804,      3.804])],
              cell=[[-5.629,  3.701,  5.629 ],
                    [-3.701,  5.629,  5.629 ],
                    [-5.629,  5.629,  3.701 ]])

with jasp('bulk/alloy/cupd-2',
            xc='PBE',
            encut=350,
            kpts=(8,8,8),
            nbands=34,
            ibrion=2,
            isif=4,
            nsw=10,
            atoms=atoms) as calc:
    cupd2 = atoms.get_potential_energy()

print 'cupd-1 = {0} eV'.format(cupd1)
print 'cupd-2 = {0} eV'.format(cupd2)
```

```
cupd-1 = -9.147752 eV
cupd-2 = -17.97924 eV
```

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

```
from jasp import *

with jasp('bulk/alloy/cupd-1') as calc:
    atoms = calc.get_atoms()
    e1 = atoms.get_potential_energy()/len(atoms)

with jasp('bulk/alloy/cupd-2') as calc:
    atoms = calc.get_atoms()
    e2 = atoms.get_potential_energy()/len(atoms)

print 'cupd-1: {0} eV/atom'.format(e1)
print 'cupd-2: {0} eV/atom'.format(e2)
```

```
cupd-1: -4.573876 eV/atom
cupd-2: -4.49481 eV/atom
```

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

$2Cu + 2Pd \rightarrow Cu_2Pd_2$ for the two configurations we examined? Below, we show some pretty general code that computes these formation energies, and normalizes them by the number of atoms in the unit cell.

```
from jasp import *

with jasp('bulk/alloy/cu') as calc:
    atoms = calc.get_atoms()
    cu = atoms.get_potential_energy()/len(atoms)

with jasp('bulk/alloy/pd') as calc:
    atoms = calc.get_atoms()
    pd = atoms.get_potential_energy()/len(atoms)

with jasp('bulk/alloy/cupd-1') as calc:
    atoms = calc.get_atoms()
    e1 = atoms.get_potential_energy()
    for atom in atoms:
        if atom.symbol == 'Cu':
            e1 -= cu
        else:
            e1 -= pd
    e1 /= len(atoms)

with jasp('bulk/alloy/cupd-2') as calc:
    atoms = calc.get_atoms()
    e2 = atoms.get_potential_energy()
    for atom in atoms:
        if atom.symbol == 'Cu':
            e2 -= cu
        else:
            e2 -= pd
    e2 /= len(atoms)

print 'Delta Hf cupd-1 = {0:1.2f} eV/atom'.format(e1)
print 'Delta Hf cupd-2 = {0:1.2f} eV/atom'.format(e2)
```

```
Delta Hf cupd-1 = -0.12 eV/atom
Delta Hf cupd-2 = -0.04 eV/atom
```

The answer is yes. Both structures are energetically more favorable than an equal composition mixture of the parent metals. The heat of formation for both structures is exothermic, but the cupd-1 structure is more stable than the cupd-2 structure. This is shown conceptually in Figure 33.

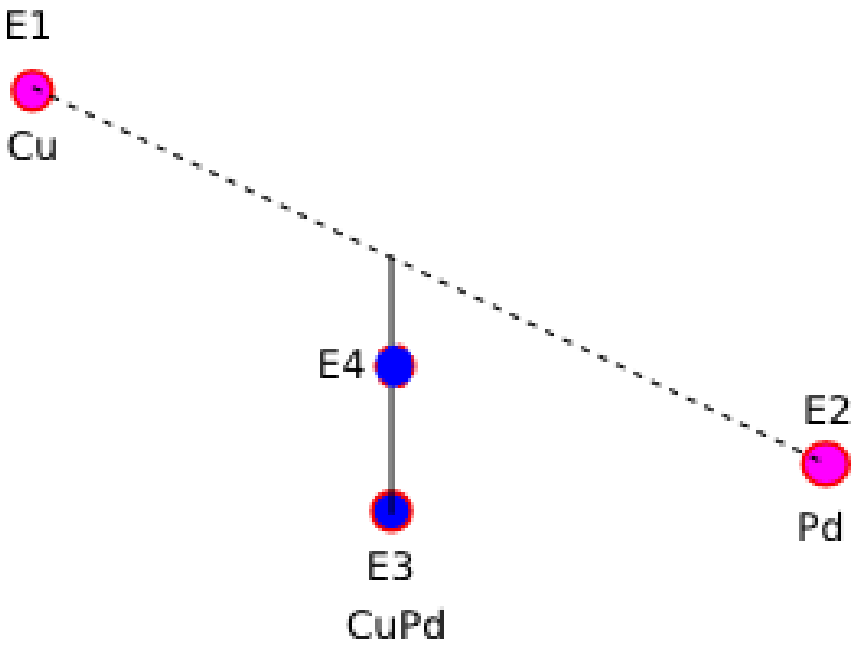


Figure 33: Conceptual picture of two alloys with exothermic formation energies.

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

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

# parent metals
with jasp('bulk/alloy/cu') as calc:
    atoms = calc.get_atoms()
    cu = atoms.get_potential_energy()/len(atoms)

with jasp('bulk/alloy/pd') as calc:
    atoms = calc.get_atoms()
    pd = atoms.get_potential_energy()/len(atoms)

atoms = Atoms([Atom('Cu',  [-3.672,     3.672,     3.672]),
               Atom('Cu',  [0.000,     0.000,     0.000]),
               Atom('Cu',  [-10.821,   10.821,    10.821]),
               Atom('Pd',  [-7.246,     7.246,     7.246])],
               cell=[[-5.464,  3.565,  5.464],
                     [-3.565,  5.464,  5.464],
                     [-5.464,  5.464,  3.565]])

with jasp('bulk/alloy/cu3pd-1',
          xc='PBE',
          encut=350,
          kpts=(8,8,8),
          nbands=34,
          ibrion=2,
          isif=4,
          nsw=10,
          atoms=atoms) as calc:
    e3 = atoms.get_potential_energy()
    for atom in atoms:
        if atom.symbol == 'Cu':
            e3 -= cu
        else:
```

```
            e3 -= pd
    e3 /= len(atoms)
print 'Delta Hf cu3pd-1 = {0:1.2f} eV/atom'.format(e3)
```

```
Delta Hf cu3pd-1 = -0.02 eV/atom
```

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

$H_{f,Cu}$ = 0.0 eV/atom, $x_0 = 0$, $H_{f,cupd-1}$ = -0.12 eV/atom, $x_3 = 0.5$.

The composition weighted average at $x_{Pd} = 0.25$ is:

$H_f = H_{f,Cu} + \frac{x0-x}{x0-x3}(H_{f,cupd-1} - H_{f,Cu})$

```
x0 = 0.0; x3 = 0.5; x = 0.25;
Hf1 = 0.0; Hf3 = -0.12;

print 'Composition weighted average  = {0} eV'.format(Hf1 + (x0-x)/(x0-x3)*(Hf3 - Hf1))
```

```
Composition weighted average  = -0.06 eV
```

We find the weighted composition formation energy of pure Cu and cupd-1 is more favorable than the formation energy of cu3pd-1. Therefore, we could expect that structure to phase separate into a mixture of pure Cu and cupd-1. Schematically what we are seeing is shown in Figure 34.

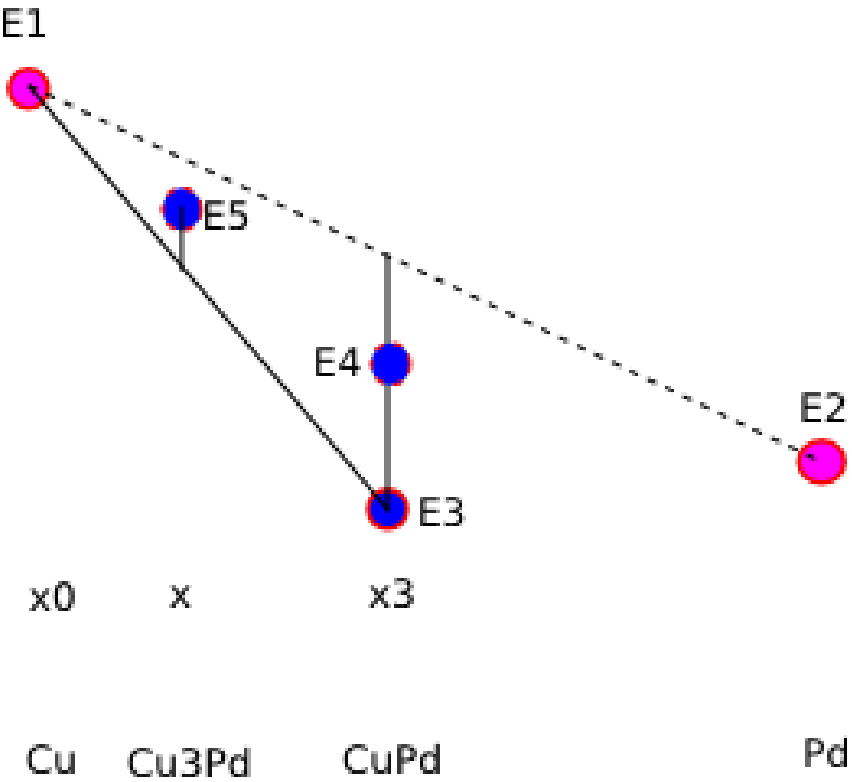


Figure 34: Illustration of phase separation.

Finally, let us consider one more structure with the Cu_3Pd stoichiometry.

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

# parent metals
with jasp('bulk/alloy/cu') as calc:
    atoms = calc.get_atoms()
    cu = atoms.get_potential_energy()/len(atoms)

with jasp('bulk/alloy/pd') as calc:
    atoms = calc.get_atoms()
    pd = atoms.get_potential_energy()/len(atoms)

atoms = Atoms([Atom('Cu',  [-1.867,     1.867,      0.000]),
               Atom('Cu',  [0.000,      0.000,      0.000]),
               Atom('Cu',  [0.000,      1.867,      1.867]),
               Atom('Pd',  [-1.867,     0.000,      1.86])],
               cell=[[-3.735,  0.000,  0.000],
                     [0.000,  0.000,  3.735],
                     [0.000,  3.735,  0.000]])

with jasp('bulk/alloy/cu3pd-2',
          xc='PBE',
          encut=350,
          kpts=(8,8,8),
          nbands=34,
          ibrion=2,
          isif=4,
          nsw=10,
          atoms=atoms) as calc:
    e4 = atoms.get_potential_energy()
    for atom in atoms:
        if atom.symbol == 'Cu':
            e4 -= cu
        else:
            e4 -= pd
    e4 /= len(atoms)
print 'Delta Hf cu3pd-2 = {0:1.2f} eV/atom'.format(e4)
```

```
Delta Hf cu3pd-2 = -0.10 eV/atom
```

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

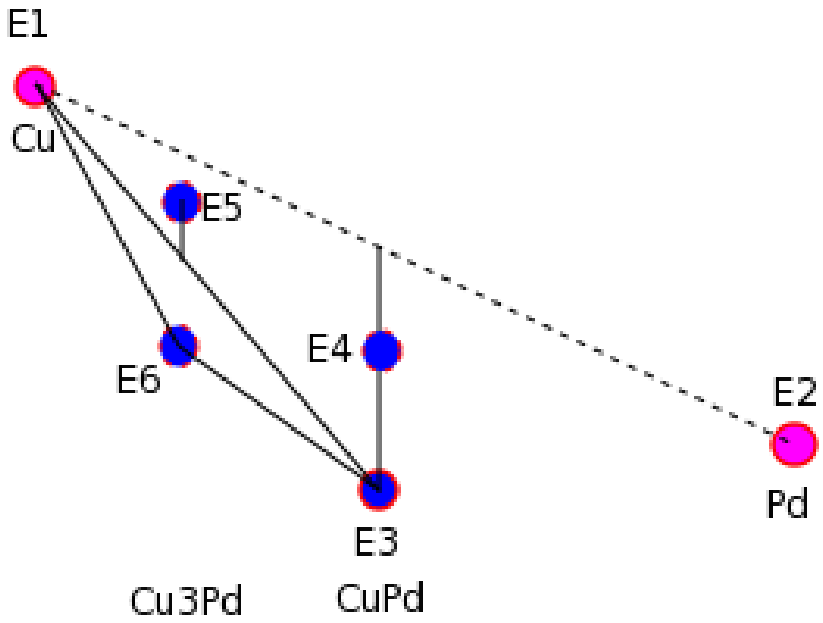


Figure 35: Illustration of phase separation.

The construction of alloy phase diagrams is difficult. You are always faced with the possibility that there is a phase that you have not calculated that is more stable than the ones you did calculate. One approach is to use a tool that automates the discovery of relevant structures such as the Alloy Theoretic Automated Toolkit (ATAT) [34, 35] which uses a cluster expansion methodology.

5.6.2 Metal oxide oxidation energies

We will consider here the reaction $2Cu_2O + O_2 \rightleftharpoons 4CuO$. The reaction energy is:
$\Delta E = 4E_{CuO} - 2E_{Cu_2O} - E_{O_2}$. We need to compute the energy of each species.

Cu_2O calculation

```
# run Cu2O calculation
from jasp import *
from ase.visualize import *
from ase import Atom, Atoms

#http://phycomp.technion.ac.il/~ira/types.html#Cu2O
a = 4.27

atoms = Atoms([Atom('Cu',[0,0,0]),
               Atom('Cu',[0.5, 0.5, 0.0]),
               Atom('Cu',[0.5, 0.0, 0.5]),
```

```
12                Atom('Cu',[0.0, 0.5, 0.5]),
13                Atom('O',[0.25, 0.25, 0.25]),
14                Atom('O',[0.75, 0.75, 0.75])])
15
16  atoms.set_cell((a,a,a), scale_atoms=True)
17
18  with jasp('bulk/Cu2O',
19            encut=400,
20            kpts=(8,8,8),
21            ibrion=2,
22            isif=3,
23            nsw=30,
24            xc='PBE',
25            atoms=atoms) as calc:
26      calc.set_nbands()
27      calc.calculate()
28      print calc
```

```
: ----------------------------
  VASP calculation from /home/jkitchin/dft-org/bulk/Cu2O
  converged: True
  Energy = -27.204590 eV

  Unit cell vectors (angstroms)
        x       y      z      length
  a0 [ 4.270  0.000  0.000] 4.270
  a1 [ 0.000  4.270  0.000] 4.270
  a2 [ 0.000  0.000  4.270] 4.270
  a,b,c,alpha,beta,gamma (deg): 4.270 4.270 4.270 90.0 90.0 90.0
  Unit cell volume = 77.854 Ang^3
  Stress (GPa):xx,    yy,    zz,    yz,    xz,    xy
             0.000  0.000  0.000 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]          tag  rmsForce constraints
   0    Cu  [0.000      0.000      0.000]    0   0.00      T T T
   1    Cu  [2.135      2.135      0.000]    0   0.00      T T T
   2    Cu  [2.135      0.000      2.135]    0   0.00      T T T
   3    Cu  [0.000      2.135      2.135]    0   0.00      T T T
   4    O   [1.067      1.067      1.067]    0   0.00      T T T
   5    O   [3.202      3.202      3.202]    0   0.00      T T T
--------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 37
           nsw: 30
        ibrion: 2
          isif: 3
         encut: 400.0
        magmom: None
          prec: Normal
          kpts: (8, 8, 8)
```

```
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
Cu: potpaw_PBE/Cu/POTCAR (git-hash: a44c591415026f53deb16a99ca3f06b1e69be10b)
```

CuO calculation

```
# run CuO calculation
from jasp import *
from ase.visualize import *
from ase import Atom, Atoms

# CuO
# http://cst-www.nrl.navy.mil/lattice/struk/b26.html
# http://www.springermaterials.com/docs/info/10681727_51.html
a = 4.6837
b = 3.4226
c = 5.1288
beta = 99.54/180*np.pi
y = 0.5819

a1 = np.array([0.5*a, -0.5*b, 0.0])
a2 = np.array([0.5*a, 0.5*b, 0.0])
a3 = np.array([c*np.cos(beta), 0.0, c*np.sin(beta)])

atoms = Atoms([Atom('Cu', 0.5*a2),
               Atom('Cu', 0.5*a1 + 0.5*a3),
               Atom('O', -y*a1 + y*a2 + 0.25*a3),
               Atom('O',  y*a1 - y*a2 - 0.25*a3)],
               cell=(a1, a2, a3))

with jasp('bulk/CuO',
          encut=400,
          kpts=(8,8,8),
          ibrion=2,
          isif=3,
          nsw=30,
          xc='PBE',
          atoms=atoms) as calc:
    calc.set_nbands()
    calc.calculate()
    print calc
```

```
: -----------------------------
  VASP calculation from /home/jkitchin/dft-org/bulk/CuO
  converged: True
  Energy = -19.509498 eV

  Unit cell vectors (angstroms)
        x       y      z      length
  a0 [ 2.302 -1.776  0.046] 2.908
  a1 [ 2.302  1.776  0.046] 2.908
```

```
  a2 [-0.762  0.000  5.087] 5.144
  a,b,c,alpha,beta,gamma (deg): 2.908 2.908 5.144 95.8 95.8 95.8
  Unit cell volume = 41.730 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
             0.008  0.013  0.009 -0.000  0.000 -0.000
 Atom#  sym       position [x,y,z]          tag  rmsForce constraints
   0    Cu   [1.151      0.888      0.023]   0   0.00      T T T
   1    Cu   [0.770     -0.888      2.566]   0   0.00      T T T
   2    O    [2.111     -0.168      1.318]   0   0.03      T T T
   3    O    [1.730      0.168      3.861]   0   0.03      T T T
--------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 23
           nsw: 30
        ibrion: 2
          isif: 3
         encut: 400.0
        magmom: None
          prec: Normal
          kpts: (8, 8, 8)
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
Cu: potpaw_PBE/Cu/POTCAR (git-hash: a44c591415026f53deb16a99ca3f06b1e69be10b)
```

Reaction energy calculation

```
from jasp import *

# don't forget to normalize your total energy to a formula unit. Cu2O
# has 3 atoms, so the number of formula units in an atoms is
# len(atoms)/3.
with jasp('bulk/Cu2O') as calc:
    atoms = calc.get_atoms()
    cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)

with jasp('bulk/CuO') as calc:
    atoms = calc.get_atoms()
    cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)

# make sure to use the same cutoff energy for the O2 molecule!
with jasp('molecules/O2-sp-triplet-400') as calc:
    atoms = calc.get_atoms()
    o2_energy = atoms.get_potential_energy()

```

```
rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
print 'Reaction energy = {0} eV'.format(rxn_energy)
```

```
Reaction energy = -1.966168 eV
```

There are a few reasons why our number does not agree with the experimental reaction energy. One reason is related to errors in the O_2 dissociation energy, and another reason is related to localization of electrons in the Cu $3d$ orbitals [37]. The first error of incorrect O_2 dissociation error is a systematic error that can be corrected empirically [37]. Fixing the second error requires the application of DFT+U (see DFT+U)

5.7 Bulk density of states

The density of states refers to the number of electronic states in a particular energy range.

5.7.1 TODO discuss why these should not be related to spectrosopy, but often are.

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

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

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

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

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

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

```
None
```

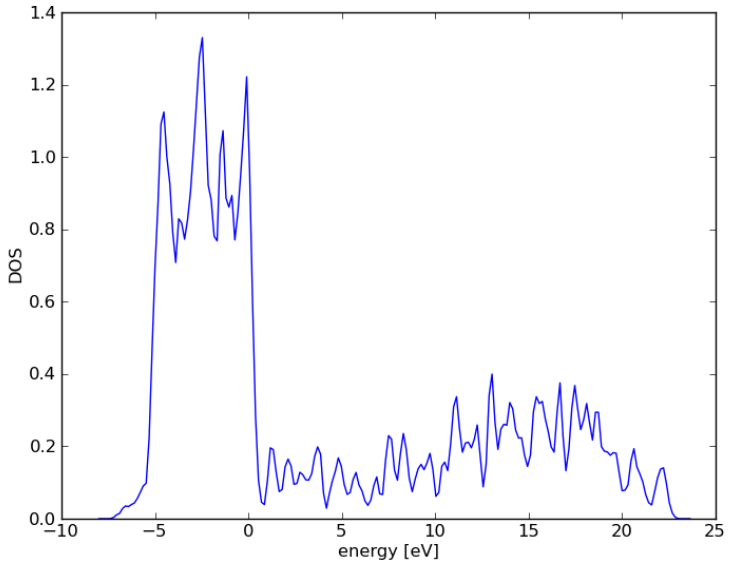


Figure 36: Total DOS for bulk Pd.

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

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

```
from jasp import *
from ase.dft import DOS
with jasp('bulk/pd-dos') as calc:
    calc.clone('bulk/pd-dos-ismear-5')

with jasp('bulk/pd-dos-ismear-5') as calc:
    bulk = calc.get_atoms()

    calc.set(ismear=-5)

    bulk.get_potential_energy()
    dos = DOS(calc, width=0.2)
    d = dos.get_dos()
    e = dos.get_energies()

import pylab as plt
plt.plot(e, d)
plt.xlabel('energy [eV]')
plt.ylabel('DOS')
plt.savefig('images/pd-dos-ismear-5.png')
```

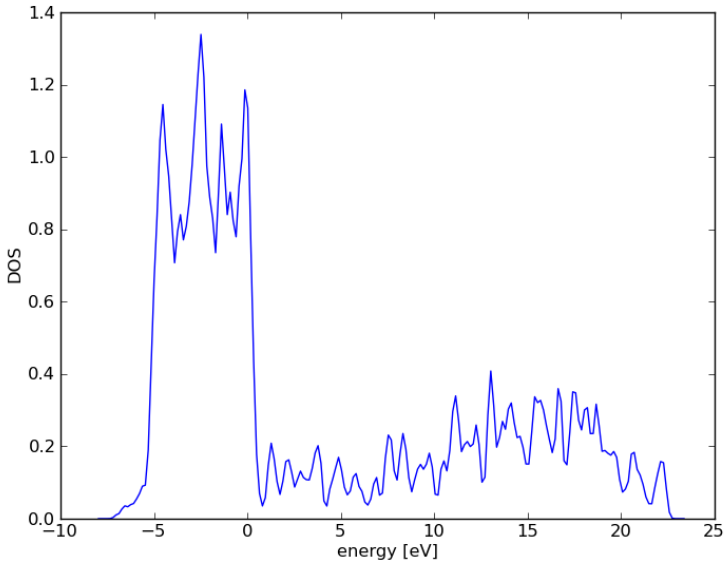


Figure 37: Total DOS for Pd computed with ISMEAR=-5

This is not that different from the regular ISMEAR, so I am not sure this is correct. Maybe there is another way to compute DOS when the tetrahedral method is used? This looks like the eigenvalue spectrum.

5.8 Atom projected density of states

In this example, we consider how to get the atom-projected density of states (ADOS). We are interested in properties of the *d*-band on Pd, such as the *d*-band center and *d*-band width. You must set the RWIGS tag to get ADOS, and these are the Wigner-Seitz radii for each atom.

It is important to note that these projected density of states are not physical observables. They are the wavefunctions projected onto atomic orbitals. For some situations this makes sense, e.g. the *d* orbitals are fairly localized and reasonably approximated by atomic orbitals. The *s* valence orbitals in a metal, in contrast, are almost totally delocalized. Depending on the cutoff radius (RWIGS) you choose, you can see very different ADOS.

```
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')

    #we will select energies in the range of -10, 5
    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 'number of 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('images/pd-ados.png')
```

```
Total # states = 9.29
number of occupied states = 8.18
d-band center = -2.00 eV
d-band width  = 2.72 eV
```

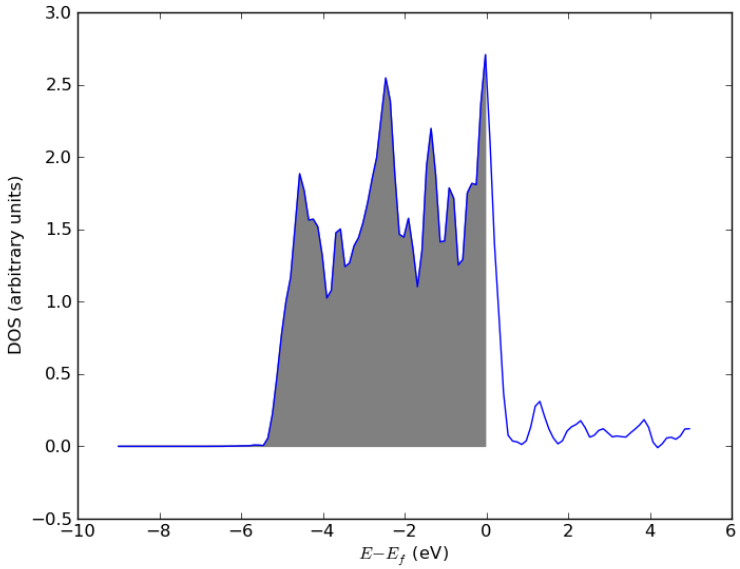


Figure 38: Atom projected d-band for bulk Pd. The shaded area corresponds to the occupied states below the Fermi level.

5.9 Band structures

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

```
from jasp import *
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:
    calc.calculate()
```

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 *

wd = 'bulk/Si-bandstructure'

with jasp('bulk/Si-selfconsistent') as calc:
    calc.clone(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,
          kpts=kpts,
          reciprocal=True,
          kpts_nintersections=10,
          icharg=11) as calc:
    calc.calculate()
```

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

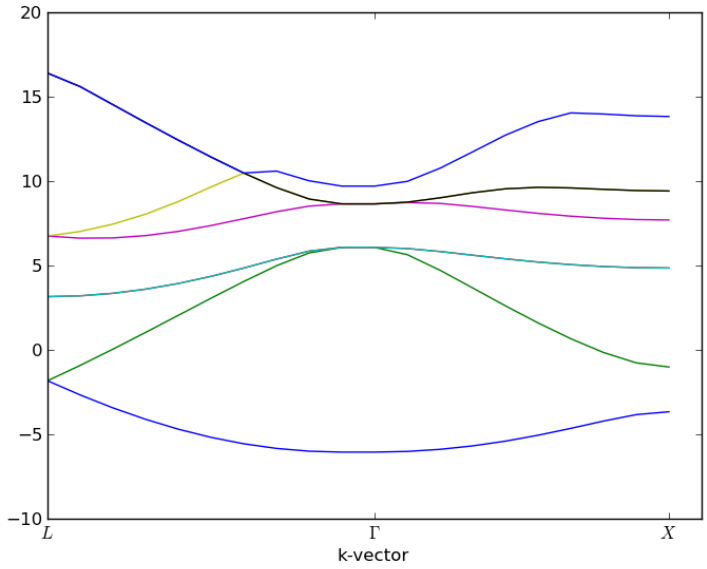


Figure 39: Calculated band-structure for Si.

5.10 Magnetism

5.10.1 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 *
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')
```

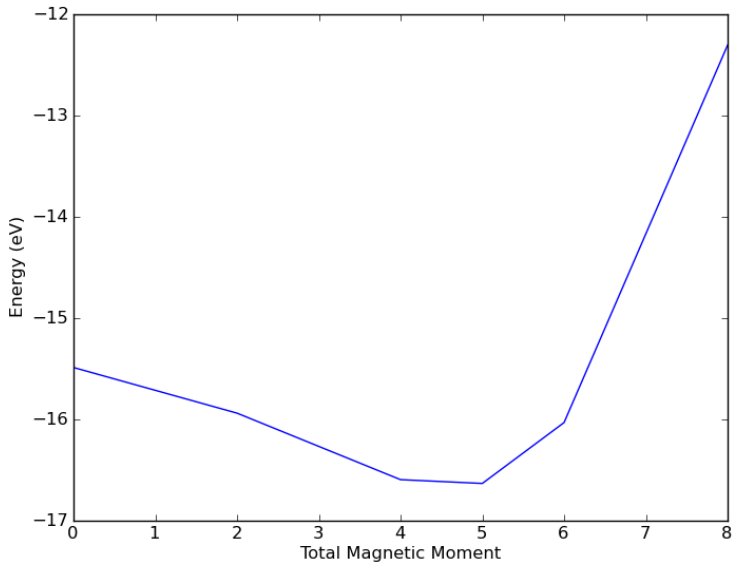


Figure 40: Total energy vs. total magnetic moment for bcc Fe.

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 *
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
```

5.10.2 Antiferromagnetic spin states

In an antiferromagnetic material, there are equal numbers of spin up and down electrons that align in a regular pattern, but pointing in opposite directions so that there is no net magnetism. It is possible to model this by setting the magnetic moments on each `ase.Atom` object.

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

atoms = Atoms([Atom('Fe',[0.00,  0.00,  0.00], magmom=5),
               Atom('Fe',[4.3,   4.3,   4.3],  magmom=-5),
               Atom('O', [2.15,  2.15,  2.15], magmom=0),
               Atom('O', [6.45,  6.45,  6.45], magmom=0)],
               cell=[[4.3,    2.15,    2.15],
                     [2.15,    4.3,     2.15],
                     [2.15,    2.15,    4.3]])

with jasp('bulk/afm-feo',
          encut=350,
          prec='Normal',
          ispin=2,
          nupdown=0, # this forces a non-magnetic solution
          lorbit=11,
          lreal=False,
          atoms=atoms) as calc:
    print 'Magnetic moments = ',atoms.get_magnetic_moments()
    print 'Total magnetic moment = ',atoms.get_magnetic_moment()
```

```
[ 2.652 -2.652  0.     0.   ]
0.0
```

You can see that even though the total magnetic moment is 0, there is a spin on both Fe atoms, and they are pointing in opposite directions.

5.10.3 TODO NiO-FeO formation energies with magnetism

5.11 TODO phonons

[2]

5.12 TODO solid state NEB

[6] Carter paper [31] recent Henkelman paper

6 Surfaces

6.1 Surface structures

As with molecules and bulk systems `ase` provides several convenience functions for making surfaces.

6.1.1 Simple surfaces

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

```
from ase.lattice.surface import *
from ase.io import write

slab = fcc111('Al', size=(2,2,3), vacuum=10.0)

write('images/Al-slab.png', slab, rotation='90x',show_unit_cell=2)
```

```
None
```

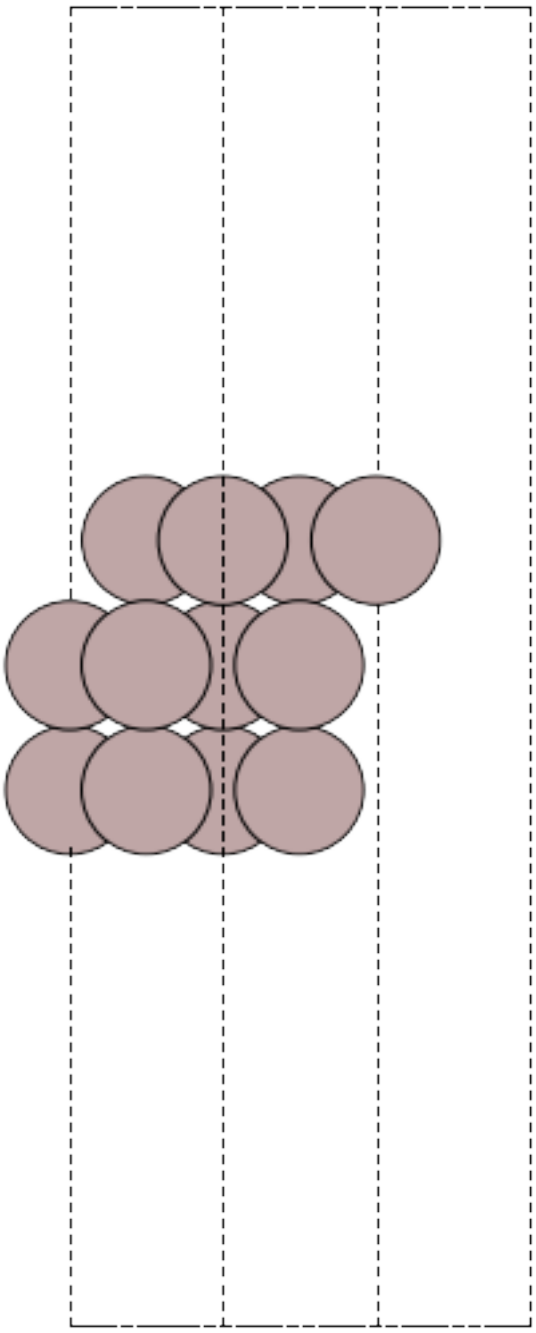


Figure 41: An Al(111) slab with three layers and 20 Å of vacuum.

6.1.2 vicinal surfaces

Many stepped surfaces are not that difficult to make now. The main idea in generating them is described here. `ase` provides a general function for making vicinal surfaces. Here is an example of a (211) surface.

```
from ase.lattice.surface import surface
from ase.io import write

# Au(211) with 9 layers
s1 = surface('Au', (2, 1, 1), 9)
s1.center(vacuum=10, axis=2)

write('images/Au-211.png',
      s1.repeat((3,3,1)),
      rotation='-30z,90x', # change the orientation for viewing
      show_unit_cell=2)
```

```
None
```

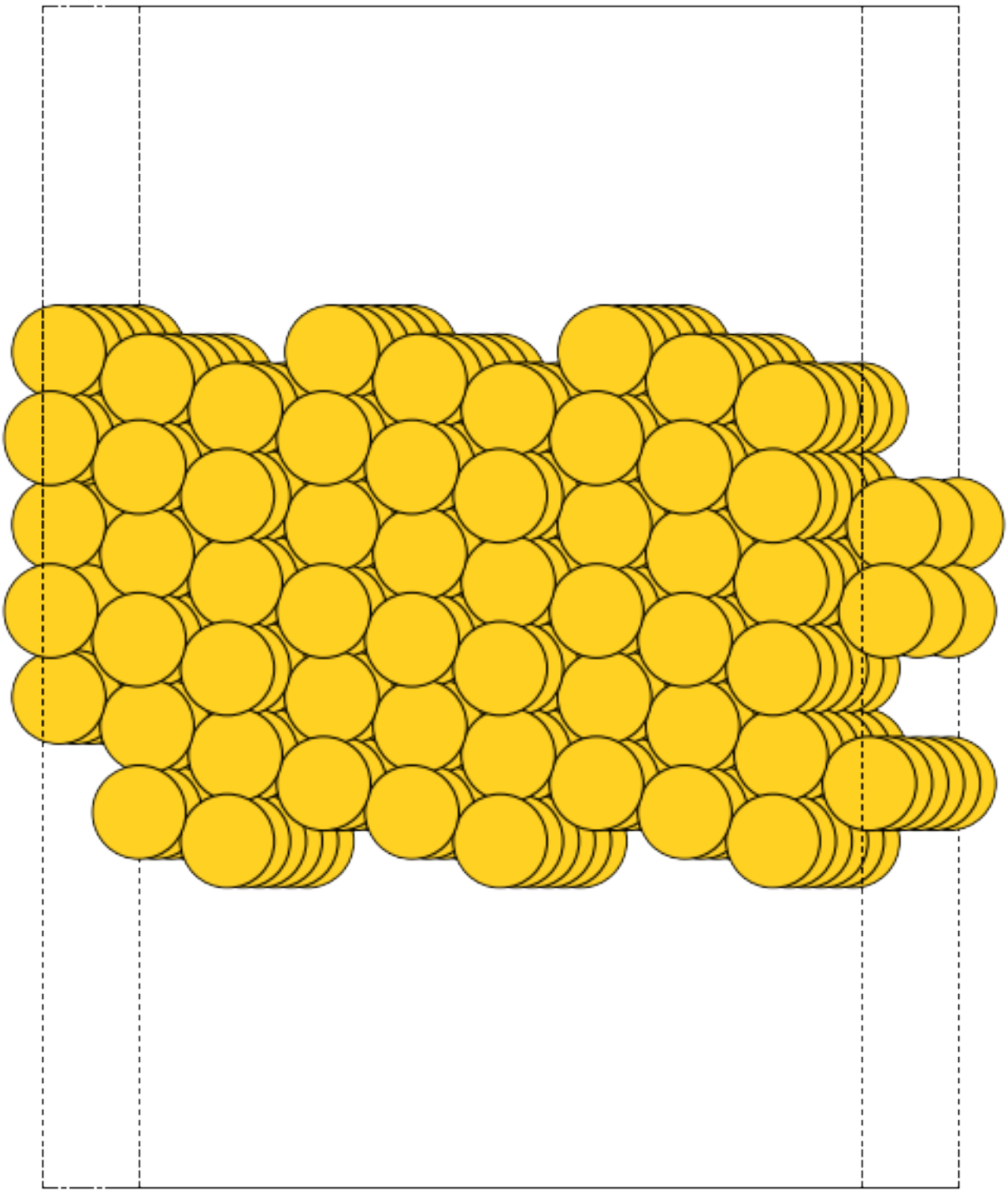


Figure 42: An Au(211) surface constructed with `ase`.

6.2 Surface relaxation

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

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

atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)

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

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/surfaces/Al-slab-unrelaxed
  converged: True
  Energy = -14.177069 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 2.864  0.000  0.000] 2.864
  a1 [ 1.432  2.480  0.000] 2.864
  a2 [ 0.000  0.000  27.015] 27.015
  a,b,c,alpha,beta,gamma (deg): 2.864 2.864 27.015 90.0 90.0 90.0
  Unit cell volume = 191.872 Ang^3
  Stress (GPa):xx,    yy,    zz,    yz,    xz,    xy
              0.007  0.007  0.002 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]          tag  rmsForce
   0    Al   [0.000      0.000     10.000]   4   0.01
   1    Al   [1.432      0.827     12.338]   3   0.18
   2    Al   [2.864      1.653     14.677]   2   0.18
   3    Al   [0.000      0.000     17.015]   1   0.01
-----------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 10
         encut: 350.0
          prec: Normal
          kpts: [6 6 1]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False
```

```
Pseudopotentials used:
----------------------
Al: potpaw_PBE/Al/POTCAR (git-hash: c8d9ecb0b6ebec0256c5f5072cee4de6a046dac2)
```

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

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

atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)

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

    for i in range(1,len(atoms)):
        print '{0}  deltaz = {1} angstroms'.format(i,atoms[i].z - atoms[i-1].z)
```

```
1  deltaz = 2.33826859022 angstroms
2  deltaz = 2.33826859022 angstroms
3  deltaz = 2.33826859022 angstroms
```

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

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

```
from jasp import *
from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)

with jasp('surfaces/Al-slab-relaxed',
          xc='PBE',
          kpts=(6,6,1),
          encut=350,
          ibrion=2,
          isif=2,
          nsw=10,
          atoms=atoms) as calc:

    constraint = FixAtoms(mask=[atom.tag >= 3 for atom in atoms])
    atoms.set_constraint(constraint)

    atoms.get_forces()
    print calc
```

```
: ----------------------------
  VASP calculation from /home/jkitchin/dft-org/surfaces/Al-slab-relaxed
  converged: True
  Energy = -14.181417 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 2.864  0.000  0.000] 2.864
  a1 [ 1.432  2.480  0.000] 2.864
  a2 [ 0.000  0.000  27.015] 27.015
  a,b,c,alpha,beta,gamma (deg): 2.864 2.864 27.015 90.0 90.0 90.0
  Unit cell volume = 191.872 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
              0.006  0.006  0.001 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]          tag  rmsForce
   0    Al  [0.000      0.000     10.000]   4   0.00
   1    Al  [1.432      0.827     12.338]   3   0.00
   2    Al  [2.864      1.653     14.637]   2   0.05
   3    Al  [0.000      0.000     16.982]   1   0.02
-------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 10
           nsw: 10
        ibrion: 2
          isif: 2
         encut: 350.0
          prec: Normal
          kpts: [6 6 1]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
Al: potpaw_PBE/Al/POTCAR (git-hash: c8d9ecb0b6ebec0256c5f5072cee4de6a046dac2)
```

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

```
1 from jasp import *
2 from ase.lattice.surface import *
```

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

```
d_(1,0) = 2.33826859022 angstroms
d_(2,1) = 2.29891157834 angstroms
d_(3,2) = 2.34530960922 angstroms
```

6.2.1 TODO Surface reconstruction

maybe a 110 missing row reconstruction? [7]

6.3 Work function

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

```
1  from jasp import *
2  from ase.lattice.surface import *
3  from ase.constraints import FixAtoms
4  from pylab import *
5  from jasp.volumetric_data import *
6
7  with jasp('surfaces/Al-slab-relaxed') as calc:
8      atoms = calc.get_atoms()
9
10 with jasp('surfaces/Al-slab-locpot',
11           xc='PBE',
12           kpts=(6,6,1),
13           encut=350,
14           lvtot=True, # write out local potential
15           lvhar=True, # write out only electrostatic potential, not xc pot
16           atoms=atoms) as calc:
17
18     calc.calculate()
19     ef = calc.get_fermi_level()
20
21     atoms = calc.get_atoms()
22
23     x,y,z,lp = calc.get_local_potential()
24
25 nx, ny, nz = lp.shape
26
27 axy = np.array([np.average(lp[:,:,z]) for z in range(nz)])
28 # setup the x-axis in realspace
29 uc = atoms.get_cell()
30 xaxis = np.linspace(0,uc[2][2],nz)
31
32 plot(xaxis, axy)
33 plot([min(xaxis), max(xaxis)], [ef, ef],'k:')
34 savefig('images/Al-wf.png')
35
36 ind = (xaxis > 0) & (xaxis < 5)
37 wf = np.average(axy[ind]) - ef
38 print ' The workfunction is {0:1.2f} eV'.format(wf)
```

```
The workfunction is 4.04 eV
```

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

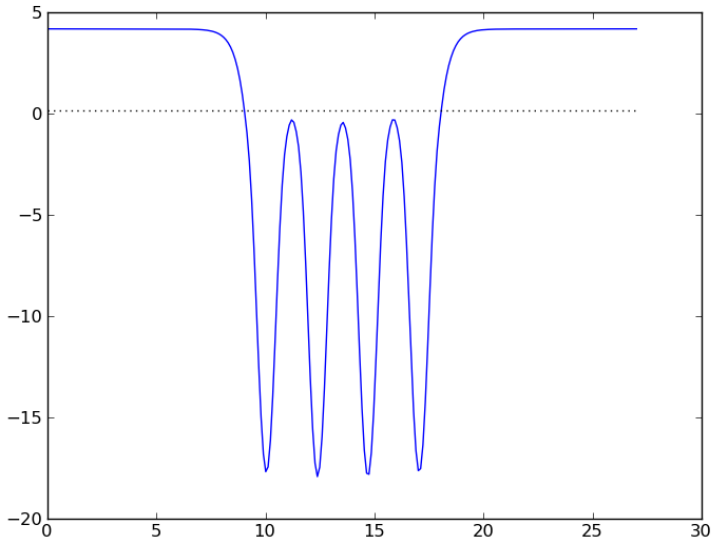


Figure 43: xy averaged local electrostatic potential of an Al(111) slab.

6.4 Surface energy

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

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

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

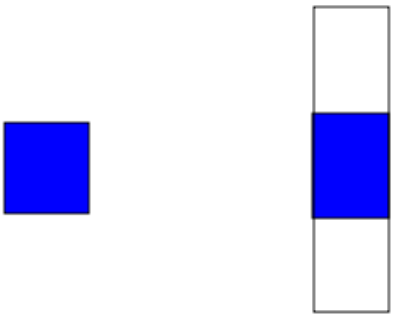


Figure 44: Schematic figure illustrating the calculation of a surface energy.

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

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

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

An alternative method for calculating surface energies that does not involve an explicit bulk calculation follows Ref. [5]. The method follows from equation (ref{eq:se}) where for a N-atom slab, in the limit of N $\rightarrow \infty$,

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

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

$\sigma = \lim_{N\rightarrow\infty} \frac{1}{2}(E^N_{slab} - N\Delta E_N)$

where $\Delta E_N = E^N_{slab} - E^{N-1}_{slab}$.

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

```
from jasp import *
from ase.units import *
from ase.lattice.surface import *
import matplotlib.pyplot as plt

Nlayers = [3, 4, 5, 6, 7, 8, 9, 10, 11]
energies = []
sigmas = []

for n in Nlayers:

    slab = fcc111('Cu', size=(1,1,n), vacuum=10.0)
    slab.center()

    with jasp('bulk/Cu-layers/{0}'.format(n),
              xc='PBE',
              encut=350,
              kpts=(8,8,1),
              atoms=slab) as calc:
        calc.set_nbands(f=2) # the default nbands in Vasp is too low for Al
        try:
            energies.append(slab.get_potential_energy())
        except (VaspSubmitted, VaspQueued):
            pass

for i in range(len(Nlayers)-1):
    N = Nlayers[i]
    DeltaE_N = energies[i+1] - energies[i]
    sigma = 0.5*(-N*energies[i+1] + (N+1)*energies[i])
    sigmas.append(sigma)
    print 'nlayers = {1:2d} sigma = {0:1.3f} eV/atom'.format(sigma, N)

plt.plot(Nlayers[0:-1], sigmas,'bo-')
plt.xlabel('Number of layers')
plt.ylabel('Surface energy (eV/atom)')
plt.savefig('images/Cu-unrelaxed-surface-energy.png')
```

```
nlayers =  3 sigma = 0.552 eV/atom
nlayers =  4 sigma = 0.398 eV/atom
nlayers =  5 sigma = 0.590 eV/atom
nlayers =  6 sigma = 0.318 eV/atom
nlayers =  7 sigma = 0.580 eV/atom
nlayers =  8 sigma = 0.321 eV/atom
nlayers =  9 sigma = 0.593 eV/atom
nlayers = 10 sigma = 0.402 eV/atom
```

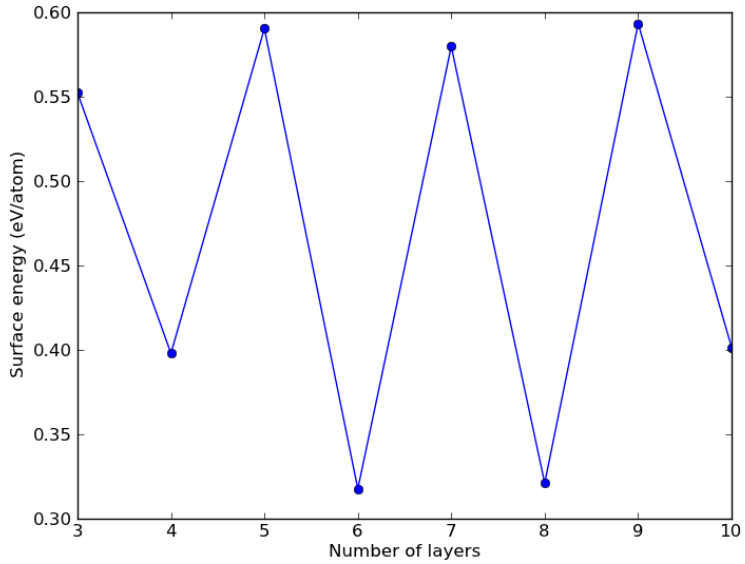


Figure 45: Surface energy of an Al(111) slab as a function of thickness.

One reason for the oscillations may be quantum size effects [10]. In [33] the surface energy of Cu(111) is reported as 0.48 eV/atom, or 1.36 J/m^2. Here is an example showing a conversion between these two units. We use ase to compute the area of the unit cell from the norm of the cross-product of the vectors defining the surface unit cell.

```
from ase.lattice.surface import *
from ase.units import *
import numpy as np

slab = fcc111('Cu', size=(1,1,3), vacuum=10.0)
cell = slab.get_cell()

area = np.linalg.norm(np.cross(cell[0], cell[1]))

sigma = 0.48 #eV/atom

print 'sigma = {0} J/m^2'.format(sigma/area/(J/m**2))
```

```
sigma = 1.3628146074 J/m^2
```

6.4.1 Advanced topics in surface energy

The surface energies can be used to estimate the shapes of nanoparticles using a Wulff construction. See [13] for an example of computing Mo_2C surface energies and particle shapes, and [38] for an example of the influence of adsorbates on surface energies and particle shapes of Cu.

For a classic paper on trends in surface energies see [36] .

6.5 Dipole correction

Slabs that are not symmetric (e.g. they have adsorbates on only one side) can develop dipole moments. Periodic dipoles can be problematic, as they create an artificial field that may affect the surface properties such as the work function and the convergence of the calculations. The solution to this problem is to introduce a dipole field in the cell to exactly cancel the dipole moment [28] (see [3] for a small correction) to minimize the effect.

Here we will just illustrate the effect.

6.5.1 slab with no dipole correction

```
# compute local potential of slab with no dipole
from ase.lattice.surface import *
from jasp import *
from jasp.volumetric_data import *
import matplotlib.pyplot as plt
slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
add_adsorbate(slab,'Na',height=1.2,position='fcc')

slab.center()

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

```
None
```

6.5.2 slab with a dipole correction

Note this takes a considerably longer time to run than without a dipole correction! In VASP there are several levels of dipole correction to apply. You can use the IDIPOL tag to turn it on, and specify which direction to apply it in (1=x, 2=y, 3=z, 4=(x, y, z)). This simply corrects the total energy and forces. It does not change the contents of LOCPOT. For that, you have to also set the LDIPOL and DIPOL tags. It is not efficient to set all three at the same time for some reason. The VASP manual recommends you first set IDIPOL to get a converged electronic structure, and then set LDIPOL to True, and set the center of electron density in DIPOL. That makes these calculations a multistep process, because we must run a calculation, analyze the charge density to get the center of charge, and then run a second calculation.

```
# compute local potential with dipole calculation on
from ase.lattice.surface import *
from jasp import *

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

slab.center()

with jasp('surfaces/Al-Na-dip',
          xc='PBE',
```

```
            encut=340,
            kpts=(2,2,1),
            idipol=3,   # only along z-axis
            lvtot=True, # write out local potential
            lvhar=True, # write out only electrostatic potential, not xc pot
            atoms=slab) as calc:

    calc.calculate()

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

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

    print 'electron-density center = {0}'.format(electron_density_center)
    uc = slab.get_cell()

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

    calc.clone('surfaces/Al-Na-dip-step2')

with jasp('surfaces/Al-Na-dip-step2',
          ldipol=True, dipol=sedc) as calc:
    calc.calculate()
```

6.5.3 Comparing no dipole correction with a dipole correction

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

```
from jasp import *
from pylab import *

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

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

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

    e1 = atoms.get_potential_energy()

with jasp('surfaces/Al-Na-dip-step2') as calc:
    atoms = calc.get_atoms()

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

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

```
    xaxis_2 = np.linspace(0,uc[2][2],nz)

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

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

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

```
The difference in energy is 0.006391 eV.
```

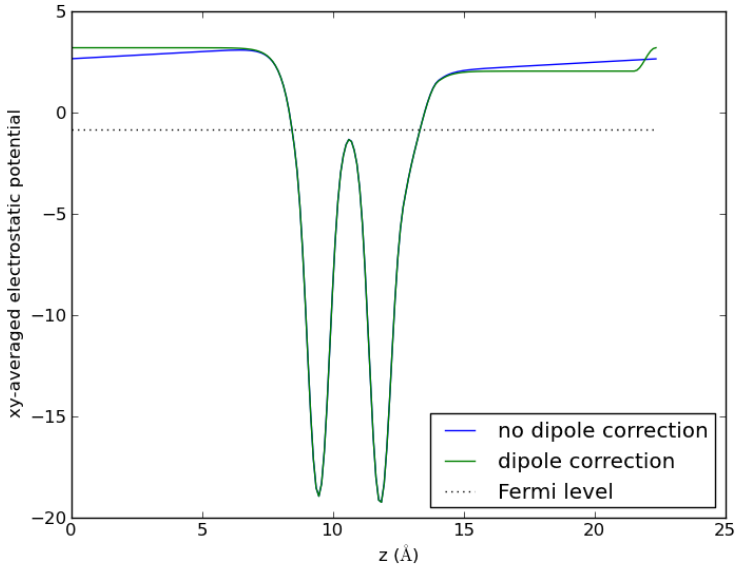


Figure 46: comparison of the electrostatic potentials with a dipole correction and without it.

The key points to notice in this figure are:

1. The two deep dips are where the atoms are.
2. Without a dipole correction, the electrostatic potential never flattens out. there is near constant slope in the vacuum region, which means there is an electric field there.
3. With a dipole moment the potential is flat in the vacuum region, except for the step jump near 23 Å.
4. The difference between the Fermi level and the flat vacuum potential is the work function.
5. The difference in energy with and without the dipole correction here is small.

6.6 Adsorption energies

6.6.1 simple estimate

Calculating an adsorption energy amounts to computing the energy of the following kind of reaction:

slab + gas-phase molecule → slab_adsorbate + products

There are many variations of this idea. The slab may already have some adsorbates on it, the slab may reconstruct on adsorption, the gas-phase molecule may or may not dissociate, and the products may or may not stick to the surface. We have to decide where to put the adsorbates, i.e. what site to put them on, and some sites will be more stable than others. We will consider the dissociative adsorption of O_2 on three sites of a Pt(111) slab. We will assume the oxygen molecule has split in half, and that the atoms have moved far apart. We will model the oxygen coverage at 0.25 ML, which means we need to use a 2×2 surface unit cell. For computational speed, we will freeze the slab, but allow the adsorbate to relax.

$\Delta H_{ads}(eV/O) = E_{slab+O} - E_{slab} - 0.5 * E_{O_2}$

Calculations

clean slab calculation

```
from jasp import *
from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
constraint = FixAtoms(mask=[True for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab',
          xc='PBE',
          kpts=(4,4,1),
          encut=350,
          atoms=atoms) as calc:
    slab_e = atoms.get_potential_energy()
```

fcc site

```
from jasp import *

from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)

# note this function only works when atoms are created by the surface module.
add_adsorbate(atoms, 'O', height=1.2, position='fcc')

constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab-O-fcc',
          xc='PBE',
          kpts=[4,4,1],
          encut=350,
          ibrion=2,
```

```
          nsw=25,
          atoms=atoms) as calc:

    calc.calculate()
```

bridge site

```
from jasp import *
from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)

# note this function only works when atoms are created by the surface module.
add_adsorbate(atoms, 'O', height=1.2, position='bridge')

constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab-O-bridge',
          xc='PBE',
          kpts=(4,4,1),
          encut=350,
          ibrion=2,
          nsw=25,
          atoms=atoms) as calc:

    calc.calculate()
```

hcp site

```
from jasp import *
from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)

# note this function only works when atoms are created by the surface module.
add_adsorbate(atoms, 'O', height=1.2, position='hcp')

constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab-O-hcp',
          xc='PBE',
          kpts=(4,4,1),
          encut=350,
          ibrion=2,
          nsw=25,
          atoms=atoms) as calc:
    calc.calculate()
```

Analysis of adsorption energies

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

with jasp('surfaces/Pt-slab') as calc:
    atoms = calc.get_atoms()
```

```
    e_slab = atoms.get_potential_energy()
write('images/pt-slab.png',atoms,show_unit_cell=2)

with jasp('surfaces/Pt-slab-O-fcc') as calc:
    atoms = calc.get_atoms()
    e_slab_o_fcc = atoms.get_potential_energy()
write('images/pt-slab-fcc-o.png',atoms,show_unit_cell=2)

with jasp('surfaces/Pt-slab-O-hcp') as calc:
    atoms = calc.get_atoms()
    e_slab_o_hcp = atoms.get_potential_energy()
write('images/pt-slab-hcp-o.png',atoms,show_unit_cell=2)

with jasp('surfaces/Pt-slab-O-bridge') as calc:
    atoms = calc.get_atoms()
    e_slab_o_bridge = atoms.get_potential_energy()
write('images/pt-slab-bridge-o.png',atoms,show_unit_cell=2)

with jasp('molecules/O2-sp-triplet-350') as calc:
    atoms = calc.get_atoms()
    e_O2 = atoms.get_potential_energy()

Hads_fcc = e_slab_o_fcc - e_slab - 0.5*e_O2
Hads_hcp = e_slab_o_hcp - e_slab - 0.5*e_O2
Hads_bridge = e_slab_o_bridge - e_slab - 0.5*e_O2

print 'Hads (fcc)    = {0} eV/O'.format(Hads_fcc)
print 'Hads (hcp)    = {0} eV/O'.format(Hads_hcp)
print 'Hads (bridge) = {0} eV/O'.format(Hads_bridge)
```

```
Hads (fcc)    = -1.0384925 eV/O
Hads (hcp)    = -0.5986145 eV/O
Hads (bridge) = -1.0384575 eV/O
```

You can see the hcp site is not as energetically favorable as the fcc site. Interestingly, the bridge site seems to be as favorable as the fcc site. This is not correct, and to see why, we have to look at the final geometries of each calculation. First the fcc (Figure 47 and hcp (Figure 48 sites, which look like we expect.

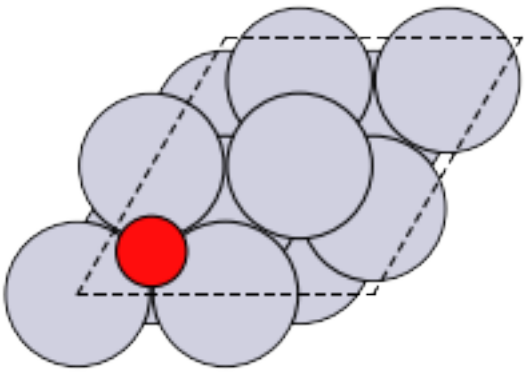


Figure 47: Final geometry of the fcc site.

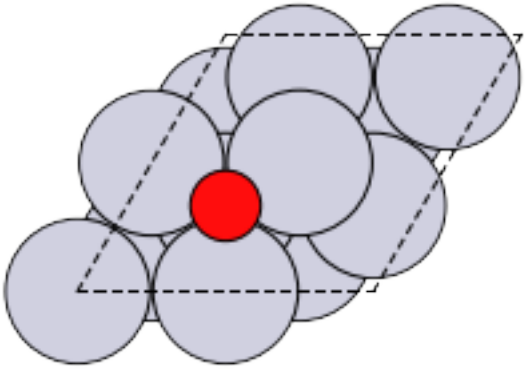


Figure 48: Final geometry of the hcp site.

The bridge site (Figure 49, however, is clearly not at a bridge site!

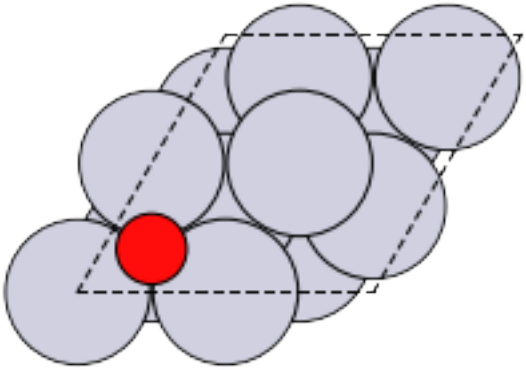


Figure 49: Final geometry of the bridge site. You can see that the oxygen atom ended up in the fcc site.

Let us see what the original geometry and final geometry for the bridge site were. The POSCAR contains the initial geometry (as long as you haven't copied CONTCAR to POSCAR), and the CONTCAR contains the final geometry.

```
from ase.io import read, write

atoms = read('surfaces/Pt-slab-O-bridge/POSCAR')
write('images/Pt-o-brige-ori.png', atoms, show_unit_cell=2)

atoms = read('surfaces/Pt-slab-O-bridge/CONTCAR')
write('images/Pt-o-brige-final.png', atoms, show_unit_cell=2)
```

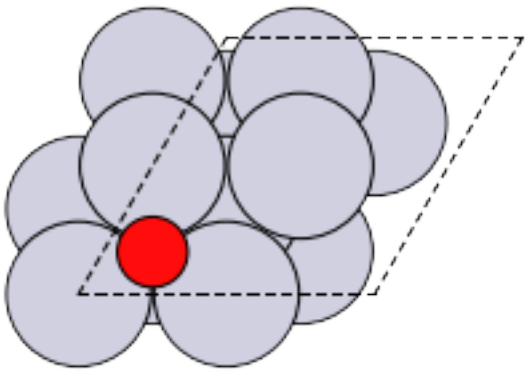


Figure 50: Initial geometry of the bridge site. It is definitely on the bridge.

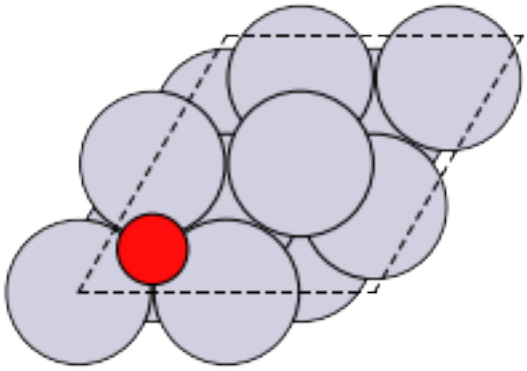


Figure 51: Final geometry of the bridge site. It has fallen into the fcc site.

You can see the problem. We should not call the adsorption energy from this calculation a bridge site adsorption energy because the O atom is actually in an fcc site! This kind of result can happen with relaxation, and you should always check that the result you get makes sense. Next, we consider how to get a bridge site adsorption energy by using constraints.

Some final notes:

1. We did not let the slabs relax in these examples, and allowing them to relax is likely to have a big effect on the adsorption energies. You have to decide how many layers to relax, and check for convergence with respect to the number of layers.

2. The slabs were pretty thin. It is typical these days to see slabs that are 4-5 or more layers thick.

3. We did not consider how well converged the calculations were with respect to k-points or ENCUT.

4. We did not consider the effect of the error in O_2 dissociation energy on the adsorption energies.

5. We did not consider coverage effects (see Coverage dependence).

Adsorption on bridge site with constraints To prevent the oxygen atom from sliding down into the fcc site, we have to constrain it so that it only moves in the z-direction. This is an artificial constraint; the bridge site is only metastable. But there are lots of reasons you might want to do this anyway. One is the bridge site is a transition state for diffusion between the fcc and hcp sites. Another is to understand the role of coordination in the adsorption energies. We use a `ase.constraints.FixScaled` constraint in ase to constrain the O atom so it can only move in the z-direction (actually so it can only move in the direction of the third unit cell vector, which only has a z-component).

```
from jasp import *
JASPRC['mode'] = None
from ase.lattice.surface import *
from ase.constraints import FixAtoms, FixScaled
from ase.io import write

atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)

# note this function only works when atoms are created by the surface module.
add_adsorbate(atoms, 'O', height=1.2, position='bridge')
constraint1 = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
# fix in xy-direction, free in z. actually, freeze movement in surface
# unit cell, and free along 3rd lattice vector
constraint2 = FixScaled(atoms.get_cell(), 12, [True, True, False])

atoms.set_constraint([constraint1, constraint2])
write('images/Pt-O-bridge-constrained-initial.png', atoms, show_unit_cell=2)
print 'Initial O position: {0}'.format(atoms.positions[-1])

with jasp('surfaces/Pt-slab-O-bridge-xy-constrained',
          xc='PBE',
          kpts=(4,4,1),
          encut=350,
          ibrion=2,
          nsw=25,
          atoms=atoms) as calc:
    e_bridge = atoms.get_potential_energy()

write('images/Pt-O-bridge-constrained-final.png', atoms, show_unit_cell=2)
print 'Final O position  : {0}'.format(atoms.positions[-1])

# now compute Hads
with jasp('surfaces/Pt-slab') as calc:
    atoms = calc.get_atoms()
    e_slab = atoms.get_potential_energy()


with jasp('molecules/O2-sp-triplet-350') as calc:
    atoms = calc.get_atoms()
    e_O2 = atoms.get_potential_energy()

Hads_bridge = e_bridge - e_slab - 0.5*e_O2

print 'Hads (bridge) = {0:1.3f} eV/O'.format(Hads_bridge)
```

You can see that only the z-position of the O atom changed. Also, the adsorption energy of O on the bridge site is **much** less favorable than on the fcc or hcp sites.

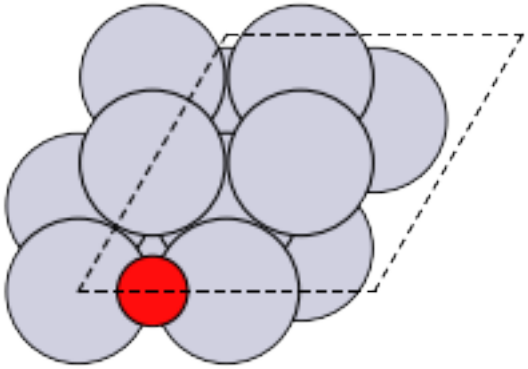


Figure 52: Initial state of the O atom on the bridge site.

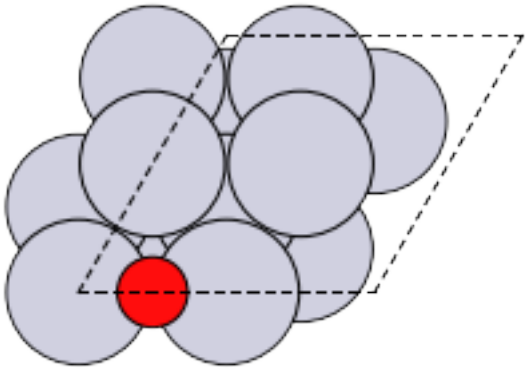


Figure 53: Final state of the constrained O atom, still on the bridge site.

6.6.2 Coverage dependence

The adsorbates on the surface can interact with each other which results in coverage dependent adsorption energies [16]. Coverage dependence is not difficult to model; we simply compute adsorption energies in different size unit cells, and/or with different adsorbate configurations. Here we consider dissociative oxygen adsorption at 1ML on Pt(111) in an fcc site, which is one oxygen atom in a 1×1 unit cell.

For additional reading, see these references from our work:

- Correlations of coverage dependence of oxygen adsorption on different metals [25, 26]
- Coverage effects of atomic adsorbates on Pd(111) [19]
- Simple model for estimating coverage dependence [16]
- Coverage effects on alloys [20]

clean slab calculation

```
1 from jasp import *
2 from ase.lattice.surface import *
```

```
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(1,1,3), vacuum=10.0)
constraint = FixAtoms(mask=[True for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab-1x1',
          xc='PBE',
          kpts=(8,8,1),
          encut=350,
          atoms=atoms) as calc:
    slab_e = atoms.get_potential_energy()
```

fcc site at 1 ML coverage

```
from jasp import *

from ase.lattice.surface import *
from ase.constraints import FixAtoms

atoms = fcc111('Pt', size=(1,1,3), vacuum=10.0)

# note this function only works when atoms are created by the surface module.
add_adsorbate(atoms, 'O', height=1.2, position='fcc')

constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
atoms.set_constraint(constraint)

with jasp('surfaces/Pt-slab-1x1-O-fcc',
          xc='PBE',
          kpts=(8,8,1),
          encut=350,
          ibrion=2,
          nsw=25,
          atoms=atoms) as calc:
    calc.calculate()
```

Adsorption energy at 1ML

```
from jasp import *

with jasp('surfaces/Pt-slab-1x1-O-fcc') as calc:
    atoms = calc.get_atoms()
    e_slab_o = atoms.get_potential_energy()

# clean slab
with jasp('surfaces/Pt-slab-1x1') as calc:
    atoms = calc.get_atoms()
    e_slab = atoms.get_potential_energy()

with jasp('molecules/O2-sp-triplet-350') as calc:
    atoms = calc.get_atoms()
    e_O2 = atoms.get_potential_energy()

hads = e_slab_o - e_slab - 0.5*e_O2
print 'Hads (1ML) = {0:1.3f} eV'.format(hads)
```

```
Hads (1ML) = -0.099 eV
```

The adsorption energy is **much** less favorable at 1ML coverage than at 0.25 ML coverage! We will return what this means in Atomistic thermodynamics effect on adsorption.

6.6.3 Effect of adsorption on the surface energy

There is a small point to make here about what adsorption does to surface energies. Let us define a general surface formation energy scheme like this:

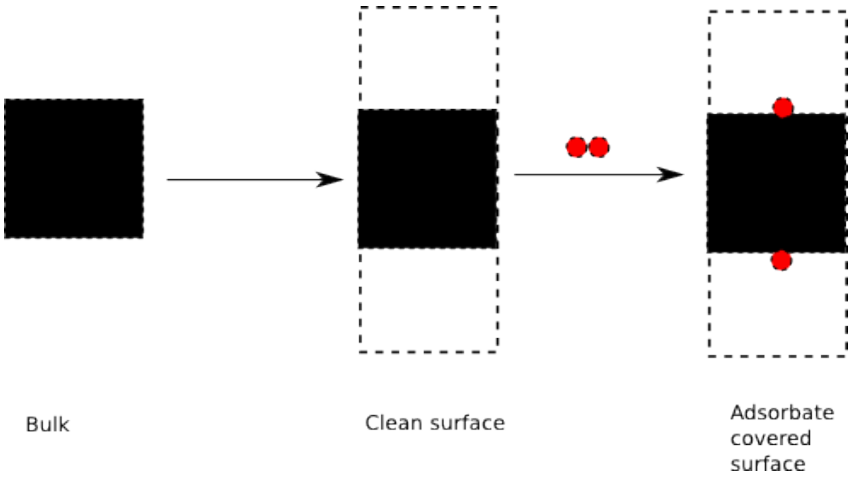


Figure 54: Schematic of forming a surface with adsorbates. First we form two clean surfaces by cleaving the bulk, then allow adsorption to occur on the surfaces.

Let us presume the surfaces are symmetric, and that each surface contributes half of the energy change. The overall change in energy:

$\Delta E = E_{slab,ads} - E_{ads} - E_{bulk}$

where the the energies are appropriately normalized for the stoichiometry. Let us rearrange the terms, and add and subtract a constant term E_{slab}.

$\Delta E = E_{slab,ads} - E_{slab} - E_{ads} - E_{bulk} + E_{slab}$

We defined $\gamma_{clean} = \frac{1}{2A}(E_{slab} - E_{bulk})$, and we defined $H_{ads} = E_{slab,ads} - E_{slab} - E_{ads}$ for adsorption on a single side of a slab. In this case, there are adsorbates on both sides of the slab, so $E_{slab,ads} - E_{slab} - E_{ads} = 2\Delta H_{ads}$. If we normalize by $2A$, the area for both sides of the slab, we get

$\frac{\Delta E}{2A} = \gamma = \gamma_{clean} + \frac{H_{ads}}{A}$

You can see here that the adsorption energy serves to stabilize, or reduce the surface energy, provided that the adsorption energy is negative.

Some final notes about the equations above:

- We were not careful about stoichiometry. As written, it is assumed there are the same number of atoms (not including the adsorbates) in the slabs and bulk, and the same number of adsorbate atoms in the slab and E_{ads}. Appropriate normalization factors must be included if that is not true.
- It is not necessary to perform a symmetric slab calculation to determine the effect of adsorption on the surface energy! You can examine $\gamma - \gamma_{clean}$ with knowledge of only the adsorption energies!

6.7 Adsorbate vibrations

Note: you can limit the number of modes calculated with constraints (`FixAtoms` or `FixScaled`) and you use IBRION=5. The other settings (6, 7, 8) do not respect the selective dynamics constraints.

```
from jasp import *

with jasp('surfaces/Pt-slab-O-fcc') as calc:
    calc.clone('surfaces/Pt-slab-O-fcc-vib')

with jasp('surfaces/Pt-slab-O-fcc-vib') as calc:
    calc.set(ibrion=5,# finite differences with selective dynamics
             nfree=2, # central differences (default)
             potim=0.015,# default as well
             ediff=1e-8,
             nsw=1)
    atoms = calc.get_atoms()
    f,v = calc.get_vibrational_modes(0)

from ase.units import *
c = 3e10 # cm/s
h = 4.135667516e-15 # eV*s

print 'vibrational energy = {0} eV'.format(f)
print 'vibrational energy = {0} meV'.format(f/meV)
print 'vibrational freq   = {0} 1/s'.format(f/h)
print 'vibrational freq   = {0} cm^(-1)'.format(f/(h*c))
```

```
vibrational energy = 0.061606647 eV
vibrational energy = 61.606647 meV
vibrational freq   = 1.48964216204e+13 1/s
vibrational freq   = 496.547387346 cm^(-1)
```

There are three modes for the free oxygen atom. One of them is a mode normal to the surface (the one with highest frequency. The other two are called frustrated translations. Note that we did not include the surface Pt atoms in the calculation, and this will have an effect on the result because the O atom could be coupled to the surface modes. It is typical to neglect this coupling because of the large difference in mass between O and Pt. Next we look at the difference in results when we calculate all the modes. Let us compare to IBRION=6.

```
from jasp import *

with jasp('surfaces/Pt-slab-O-fcc') as calc:
    calc.clone('Pt-slab-O-fcc-vib-ibrion=6')

with jasp('surfaces/Pt-slab-O-fcc-vib-ibrion=6') as calc:
    calc.set(ibrion=6,# finite differences with symmetry
             nfree=2, # central differences (default)
             potim=0.015,# default as well
             ediff=1e-8,
             nsw=1)
    atoms = calc.get_atoms()

    f,m = calc.get_vibrational_modes(0)
    allfreq = calc.get_vibrational_modes()[0]

from ase.units import *
```

```
c = 3e10 # cm/s
h = 4.135667516e-15 # eV*s

print 'For mode 0:'
print 'vibrational energy = {0} eV'.format(f)
print 'vibrational energy = {0} meV'.format(f/meV)
print 'vibrational freq   = {0} 1/s'.format(f/h)
print 'vibrational freq   = {0} cm^(-1)'.format(f/(h*c))
```

```
For mode 0:
vibrational energy = 0.063537929 eV
vibrational energy = 63.537929 meV
vibrational freq   = 1.53634035507e+13 1/s
vibrational freq   = 512.113451691 cm^(-1)
```

Note that now there are 39 modes, which is 3*N where N=13 atoms in the unit cell. Many of the modes are low in frequency, which correspond to slab modes that are essentially phonons. The O frequencies are not that different from the previous calculation (497 vs 512 cm^{-1}. This is why it is common to keep the slab atoms frozen.

Calculating these results took 39*2 finite differences. It took about a day to get these results on a single CPU. It pays to use constraints to minimize the number of these calculations.

6.7.1 Vibrations of the bridge site

```
from jasp import *
from ase.constraints import FixAtoms

with jasp('surfaces/Pt-slab-O-bridge-xy-constrained') as calc:
    calc.clone('surfaces/Pt-slab-O-bridge-vib')
    atoms = calc.get_atoms()

del atoms.constraints
constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
atoms.set_constraint([constraint])

with jasp('surfaces/Pt-slab-O-bridge-vib') as calc:
    calc.set(ibrion=5,# finite differences with selective dynamics
             nfree=2, # central differences (default)
             potim=0.015,# default as well
             ediff=1e-8,
             nsw=1)
    atoms.set_calculator(calc) # reset atoms
    atoms.get_potential_energy()
    #print calc.get_vibrational_frequencies()
    f,v = calc.get_vibrational_modes(2)
    print calc.get_vibrational_modes()[0]

from ase.units import *
c = 3e10 # cm/s
h = 4.135667516e-15 # eV*s

print 'vibrational energy = {0} eV'.format(f)
print 'vibrational energy = {0} meV'.format(f/meV)
print 'vibrational freq   = {0} 1/s'.format(f/h)
print 'vibrational freq   = {0} cm^(-1)'.format(f/(h*c))
```

```
[0.065486064, 0.045079206000000004, (0.019955987+0j)]
```

```
vibrational energy = (0.019955987+0j) eV
vibrational energy = (19.955987+0j) meV
vibrational freq   = (4.82533639921e+12+0j) 1/s
vibrational freq   = (160.84454664+0j) cm^(-1)
```

Note that we have one imaginary mode. This corresponds to the motion of the O atom falling into one of the neighboring 3-fold sites. It also indicates this position is not a stable minimum, but rather a saddle point. This position is a transition state for hopping between the fcc and hcp sites.

6.8 Surface Diffusion barrier

See this review [29] of diffusion on transition metal surfaces.

6.8.1 Standard nudged elastic band method

Here we illustrate a standard NEB method. You need an initial and final state to start with.

```
from jasp import *
from ase.neb import NEB
import matplotlib.pyplot as plt
from scipy import interpolate
import numpy as np

with jasp('surfaces/Pt-slab-O-fcc') as calc:
    initial_atoms = calc.get_atoms()

with jasp('surfaces/Pt-slab-O-hcp') as calc:
    final_atoms = calc.get_atoms()

# here is our estimated transition state. we use vector geometry to
# define the bridge position, and add 1.451 Ang to z based on our
# previous bridge calculation.
ts = initial_atoms.copy()
ts.positions[-1] = 0.5*(ts.positions[9] + ts.positions[10]) + [0,0,1.451]

# construct the band
images = [initial_atoms]
images += [initial_atoms.copy()]
images += [ts.copy()]  # this is the TS

neb = NEB(images)
# Interpolate linearly the positions of these images:
neb.interpolate()

# now add the second half
images2 = [ts.copy()]
images2 += [ts.copy()]
images2 += [final_atoms]

neb2 = NEB(images2)
neb2.interpolate()

# collect final band. Note we do not repeat the TS in the second half
final_images = images + images2[1:]

with jasp('surfaces/Pt-O-fcc-hcp-neb',
          ibrion=1,
          nsw=90,
          spring=-5,
```

```
43          atoms=final_images) as calc:
44
45      try:
46          images, energies = calc.get_neb()
47          p = calc.plot_neb(show=False)
48
49          # remember you are in surfaces/Pt-O-fcc-hcp-neb, so to save in
50          # the images directory you need ../../ in the path to get you
51          # back up.
52          plt.savefig('../../images/pt-o-fcc-hcp-neb.png')
53      except (VaspSubmitted, VaspQueued):
54          pass
```

```
Optimization terminated successfully.
         Current function value: -0.575389
         Iterations: 12
         Function evaluations: 24
```

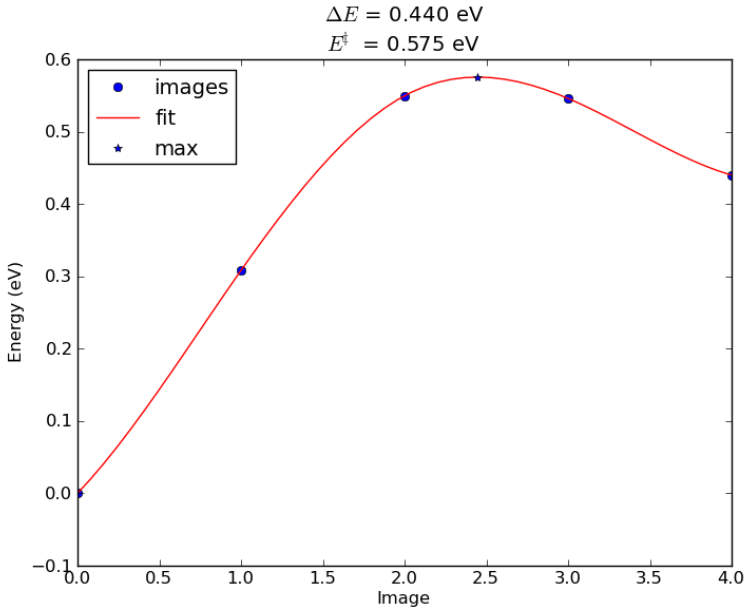


Figure 55: Energy pathway for O diffusion from an fcc to hcp site with a spline fit to determine the barrier

6.8.2 Climbing image NEB

One issue with the standard NEB method is there is no image that is exactly at the transition state. That means there is some uncertainty of the true energy of the transition state, and there is no way to verify the transition state by vibrational analysis. The climbing image NEB method solves that problem by making one image climb to the top. You set LCLIMB=True in `jasp` to turn on the climbing image method. Here we use the previous calculation as a starting point and turn on the climbing image method.

```
# perform a climbing image NEB calculation
from jasp import *
with jasp('surfaces/Pt-O-fcc-hcp-neb') as calc:
    calc.clone('surfaces/Pt-O-fcc-hcp-cineb')

with jasp('surfaces/Pt-O-fcc-hcp-cineb',debug=logging.DEBUG) as calc:
    calc.set(ichain=0,lclimb=True)

    images, energies = calc.get_neb(npi=4)
    calc.plot_neb(show=False)
import matplotlib.pyplot as plt
plt.savefig('images/pt-o-cineb.png')
```

```
Optimization terminated successfully.
         Current function value: -0.575288
         Iterations: 12
         Function evaluations: 24
```

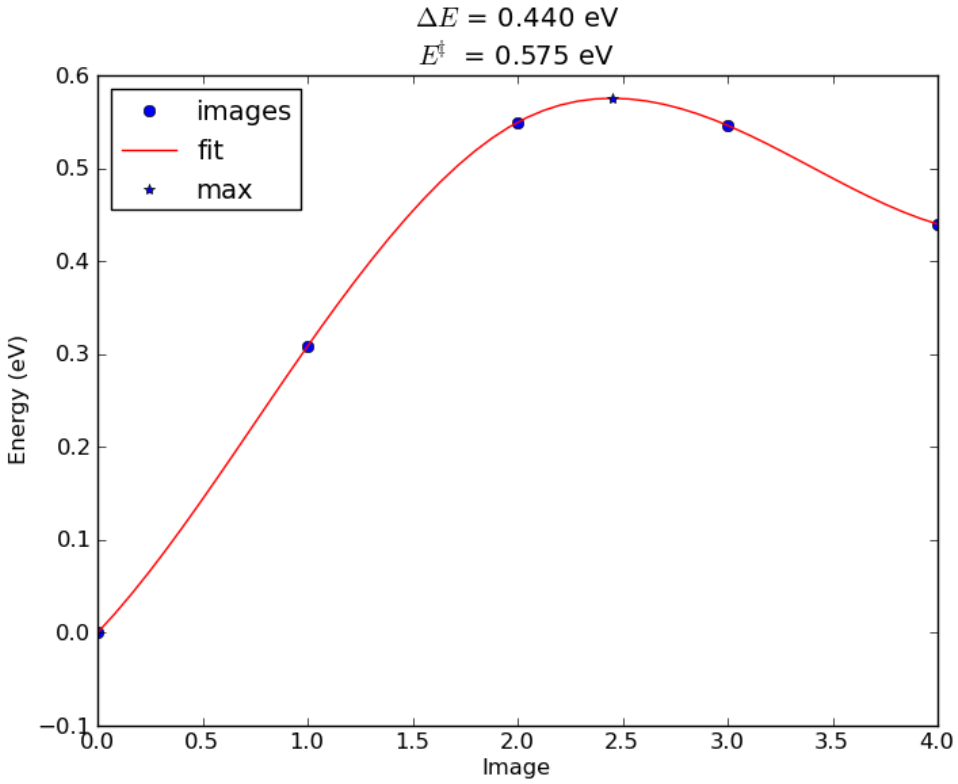


Figure 56: Climbing image NEB

This did not do exactly what I expected. I thought there would be an image that had climbed to the top. Maybe this is an artifact of the spline fitting. More band points would probably clarify that.

6.8.3 TODO use vibrations to confirm transition state

6.9 TODO Diffusion rates with transition state theory

7 Atomistic thermodynamics

Let us consider how much the Gibbs free energy of an O_2 molecule changes as a function of temperature, at 1 atm. We use the Shomate polynomials to approximate the temperature dependent entropy and enthalpy, and use the parameters from the NIST Webbook for O_2.

```
import numpy as np
import matplotlib.pyplot as plt
from ase.units import *
K = 1. #not defined in ase.units!

# Shomate parameters
A = 31.32234; B = -20.23531; C = 57.86644
D = -36.50624; E = -0.007374; F = -8.903471
G = 246.7945; H = 0.0

def entropy(T):
    '''entropy returned as eV/K
    T in K
    '''
    t = T/1000.
    s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
    return s*J/mol/K

def enthalpy(T):
    ''' H - H(298.15) returned as eV/molecule'''
    t = T/1000.
    h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
    return h*kJ/mol

T = np.linspace(100,700)

G = enthalpy(T) - T*entropy(T)

plt.plot(T,G)
plt.xlabel('Temperature (K)')
plt.ylabel('$\Delta G^\circ$ (eV)')
plt.savefig('images/O2-mu.png')
```

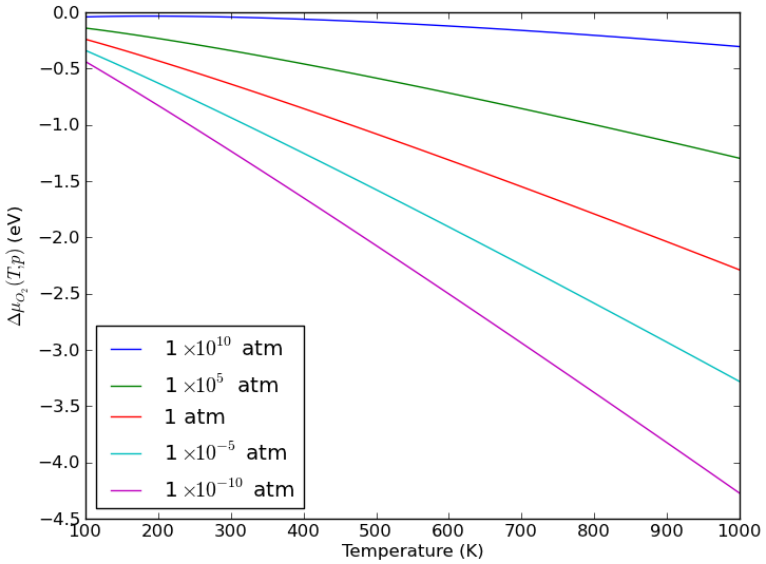


Figure 57: Effect of temperature on the Gibbs free energy of an O_2 molecule at 1 atm.

This is clearly a big effect! Between 500-600K, the energy has dropped by nearly 1 eV.

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

atm = 101325*Pascal #atm is not defined in units
K = 1

# examine range over 10^-10 to 10^10 atm
P = np.logspace(-10,10)*atm

plt.semilogx(P/atm, kB*(300*K)*np.log(P/(1*atm)),label='300K')
plt.semilogx(P/atm, kB*(600*K)*np.log(P/(1*atm)),label='600K')
plt.xlabel('Pressure (atm)')
plt.ylabel('$\Delta G$ (eV)')
plt.legend(loc='best')
plt.savefig('images/O2-g-p.png')
```

```
None
```

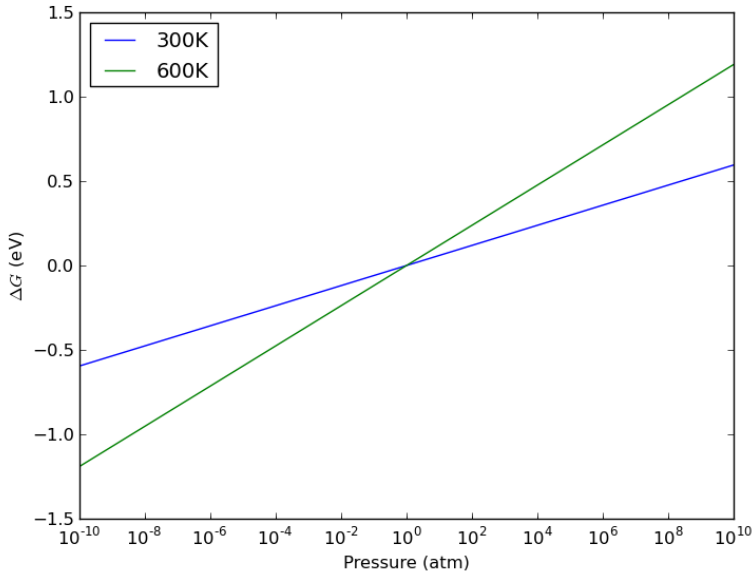


Figure 58: Effects of pressure on the ideal gas Gibbs free energy of O_2.

Similarly, you can see that simply changing the pressure has a large effect on the Gibbs free energy of an ideal gas through the term: $kT\ln(P/P_0)$, and that this effect is also temperature dependent. This leads us to the final formula we will use for the chemical potential of oxgyen:

$\mu_{O_2} = E_{O_2}^{DFT} + E_{O_2}^{ZPE} + \Delta\mu(T) + kT\ln(P/P_0)$

7.1 Bulk phase stability of oxides

We will consider the effects of oxygen pressure and temperature on the formation energy of Ag_2O and Cu_2O.

$2Cu + 1/2O_2 \rightarrow Cu_2O$

In atomistic thermodynamics, we define the free energy of formation as:

$G_f = G_{Cu_2O} - 2G_{Cu} - 0.5G_{O_2}$

We will at this point assume that the solids are incompressible so that $p\Delta V \approx 0$, and that $S_{Cu_2O} - 2S_{Cu} \approx 0$, which leads to $G_{Cu_2O} - 2G_{Cu} \approx E_{Cu_2O} - 2E_{Cu}$, which we directly compute from DFT. We express $G_{O_2} = \mu_{O_2} = E_{O_2}^{DFT} + E_{O_2}^{ZPE} + \Delta\mu(T) + kT\ln(P/P_0)$. In this example we neglect the zero-point energy of the oxygen molecule, and finally arrive at:

$G_f \approx E_{Cu_2O} - 2E_{Cu} - 0.5(E_{O_2}^{DFT} + \delta\mu(T) + kT\ln(P/P_0))$

Which, after grouping terms is:

$G_f \approx E_{Cu_2O} - 2E_{Cu} - 0.5(E_{O_2}^{DFT}) - 0.5 * \Delta\mu_{O_2}(P,T)$

with $\Delta\mu_{O_2}(P,T) = \delta\mu(T) + kT\ln(P/P_0)$. We get $\delta\mu(T)$ from the Janaf Tables, or the NIST Webbook.

You will recognize in this equation the standard formation energy we calculated in Metal oxide oxidation energies plus a correction for the non standard state pressure and temperature ($\Delta\mu_{O_2}(P,T) = 0$ at standard state).

$G_f \approx H_f - 0.5 * \Delta\mu_{O_2}(P,T)$

The formation energy of Cu_2O is -1.9521 eV/formula unit. The formation energy for Ag_2O is -0.99 eV/formula unit. Let us consider what temperature the oxides decompose at a fixed oxygen pressure of 1×10^{-10} atm. We need to find the temperature where:

$H_f = 0.5 * \Delta\mu_{O_2}(P, T)$

which will make the formation energy be 0.

```
import numpy as np
import matplotlib.pyplot as plt
from ase.units import *
from scipy.optimize import fsolve

K = 1. #not defined in ase.units!
atm = 101325*Pascal

# Shomate parameters valid from 100-700K
A = 31.32234; B = -20.23531; C = 57.86644
D = -36.50624; E = -0.007374; F = -8.903471
G = 246.7945; H = 0.0

def entropy(T):
    '''entropy returned as eV/K
    T in K
    '''
    t = T/1000.
    s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
    return s*J/mol/K

def enthalpy(T):
    ''' H - H(298.15) returned as eV/molecule'''
    t = T/1000.
    h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
    return h*kJ/mol

def DeltaMu(T,P):
    '''
    T in K
    P in atm
    '''
    return enthalpy(T) - T*entropy(T) + kB*T*np.log(P/atm)

T = np.linspace(100,1000)
P = 1e-10*atm

def func(T):
    'Cu2O'
    return -1.95 - 0.5*DeltaMu(T,P)

print 'Cu2O decomposition temperature is {0:1.0f} K'.format(fsolve(func, 900)[0])

def func(T):
    'Ag2O'
    return -0.99 - 0.5*DeltaMu(T,P)

print 'Ag2O decomposition temperature is {0:1.0f} K'.format(fsolve(func, 470)[0])

# you have use \\times to escape the first \ in pyplot
plt.plot(T, DeltaMu(T,1e10*atm),label='1$\\times 10^{10}$ atm')
plt.plot(T, DeltaMu(T,1e5*atm),label='1$\\times 10^5$ atm')
plt.plot(T, DeltaMu(T,1*atm),label='1 atm')
plt.plot(T, DeltaMu(T,1e-5*atm),label='1$\\times 10^{-5}$ atm')
plt.plot(T, DeltaMu(T,1e-10*atm),label='1$\\times 10^{-10}$ atm')

plt.xlabel('Temperature (K)')
plt.ylabel('$\Delta \mu_{O_2}(T,p)$ (eV)')
```

```
plt.legend(loc='best')
plt.savefig('images/O2-mu.png')
```

```
Cu2O decomposition temperature is 917 K
Ag2O decomposition temperature is 478 K
```

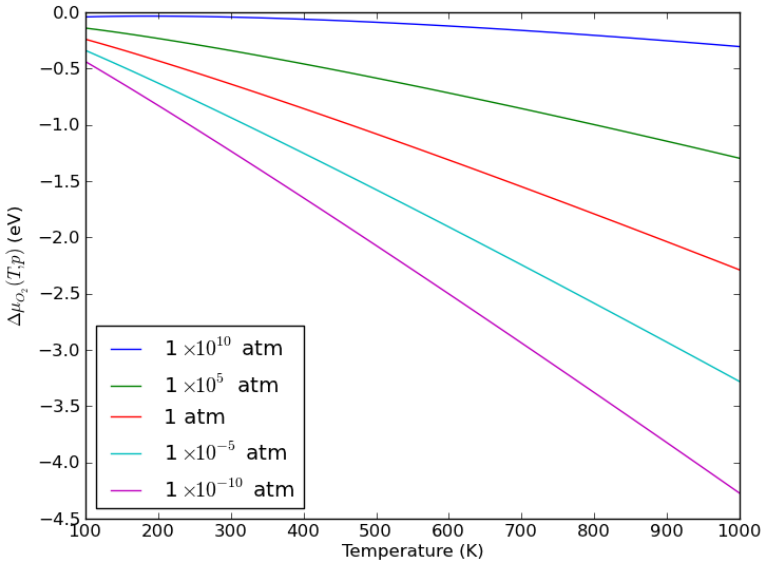


Figure 59: $\Delta\mu_{O_2}(T,p)$ at different pressures and temperatures.

```
import numpy as np
import matplotlib.pyplot as plt
from ase.units import *
from scipy.optimize import fsolve

K = 1. #not defined in ase.units!
atm = 101325*Pascal

# Shomate parameters valid from 100-700K
A = 31.32234; B = -20.23531; C = 57.86644
D = -36.50624; E = -0.007374; F = -8.903471
G = 246.7945; H = 0.0

def entropy(T):
    '''entropy returned as eV/K
    T in K
    '''
    t = T/1000.
    s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
    return s*J/mol/K

def enthalpy(T):
    ''' H - H(298.15) returned as eV/molecule'''
    t = T/1000.
    h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
    return h*kJ/mol

```

```
def DeltaMu(T,P):
    '''
    T in K
    P in atm
    '''
    return enthalpy(T) - T*entropy(T) + kB*T*np.log(P/atm)

P = np.logspace(-11,1,10)*atm
T = []
for p in P:

    def func(T):
        return -0.99 - 0.5*DeltaMu(T,p)
    T.append(fsolve(func, 450)[0])

plt.semilogy(T,P/atm)
plt.xlabel('Temperature (K)')
plt.ylabel('Pressure (atm)')
plt.text(800,1e-7,'Ag')
plt.text(600,1e-3,'Ag$_2$O')
plt.savefig('images/Ag2O-decomposition.png')
```

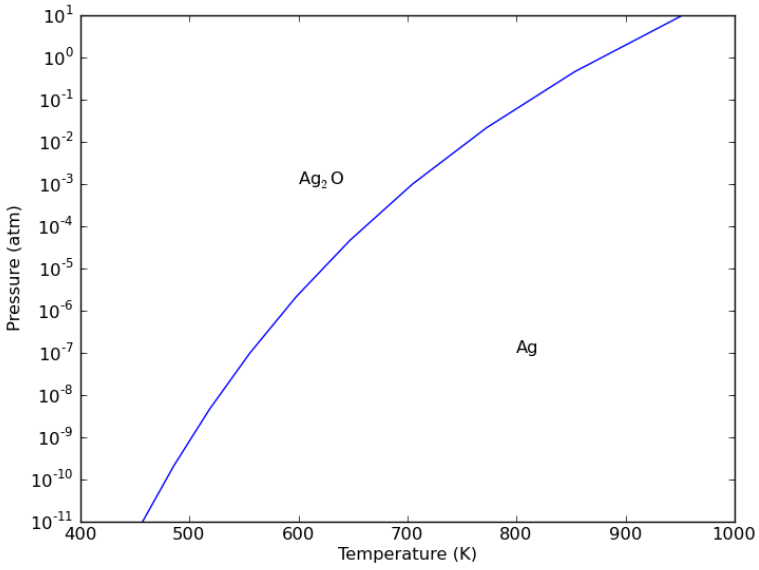


Figure 60: Temperature dependent decomposition pressure for Ag_2O.

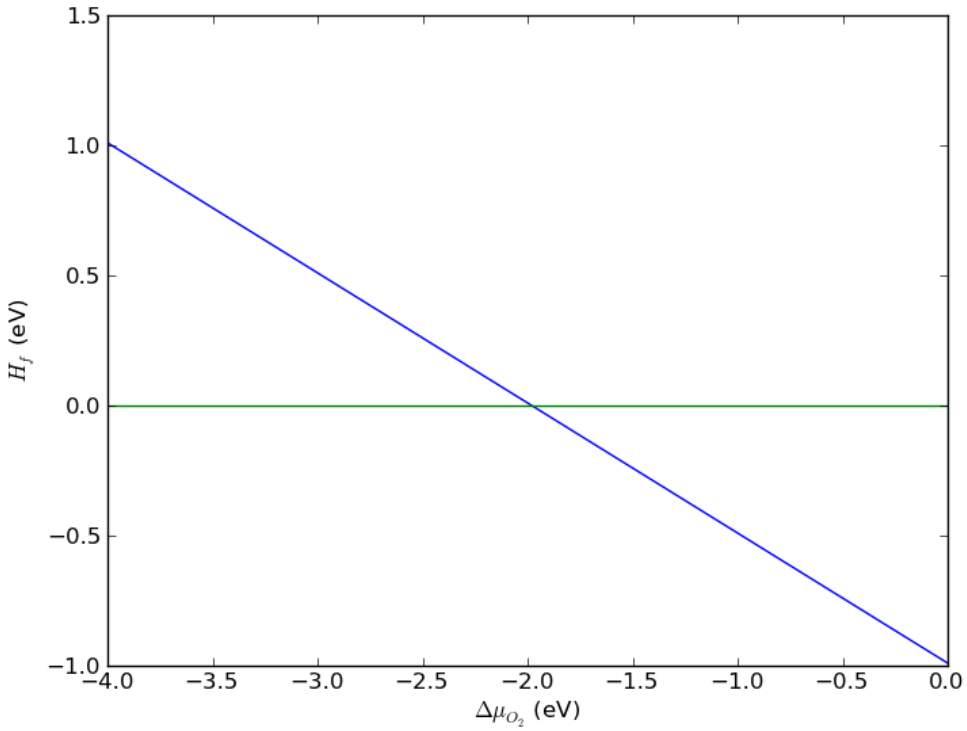
This shows that at high temperature and low p_{O_2} metallic silver is stable, but if the p_{O_2} gets high enough, the oxide becomes thermodynamically favorable. Here is another way to look at it.

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

K = 1. #not defined in ase.units!
atm = 101325*Pascal
Hf = -0.99
```

```
8
9  P = 1*atm
10
11 Dmu = np.linspace(-4,0)
12
13 Hf = -0.99 - 0.5*Dmu
14
15 plt.plot(Dmu, Hf, label='Ag$_2$O')
16 plt.plot(Dmu, np.zeros(Hf.shape), label='Ag')
17 plt.xlabel('$\Delta \mu_{O_2}$ (eV)')
18 plt.ylabel('$H_f$ (eV)')
19 plt.savefig('images/atomistic-thermo-hf-mu.png')
```

```
None
```



This graph shows graphically the $\Delta\mu_{O_2}$ required to make the metal more stable than the oxide. Anything less than about -2 eV will have the metal more stable. That can be achieved by any one of the following combinations (graphically estimated from Figure 59): About 500K at 1×10^{-10} atm, 600K at 1×10^{-5} atm, 900K at 1atm, etc...

7.2 Effect on adsorption

We now consider the question: Given a pressure and temperature, what coverage would you expect on a surface? We saw earlier that adsorption energies depend on the site and coverage. We lso know the coverage depends on the pressure and temperature. Above some temperature, desorption occurs, and below some pressure adsorption will not be favorable. We seek to develop a quantitative method to determine those conditions.

We redefine the adsorption energy as:

$\Delta G_{ads} \approx E_{slab,ads} - E_{slab} - \mu_{ads}$

where again we neglect all contributions to the free energy of the slabs from vibrational energy and entropy, as well as configurational entropy if that is relevant. That leaves only the pressure and temperature dependence of the adsorbate, which we treat in the ideal gas limit.

We expand μ_{ads} as $E_{ads} + \Delta\mu(T, p)$, and thus:

$\Delta G_{ads} \approx E_{slab,ads} - E_{slab} - E_{ads} - \Delta\mu(T, p)$

or

$\Delta G_{ads} \approx \Delta H_{ads} - \Delta\mu(T, p)$

where ΔH_{ads} is the adsorption energy we defined earlier. Now we can examine the effect of $\Delta\mu(T, p)$ on the adsorption energies. We will use the adsorption energies for the oxygen on Pt(111) system we computed earlier:

Table 4: Adsorption site dependence of adsorption energies of oxygen on Pt(111).

system	$\Delta H(eV/O)$
fcc (0.25 ML)	-1.04
hcp (0.25 ML)	-0.60
bridge (0.25 ML)	-0.49
fcc(1ML)	-0.10

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

fcc25 = -1.04
hcp25 = -0.60
bridge25 = -0.49
fcc1 = -0.10

Dmu = np.linspace(-4,0)

plt.plot(Dmu, np.zeros(Dmu.shape), label='Pt(111)')
plt.plot(Dmu, fcc25 - 0.5*Dmu, label='fcc - 0.25 ML')
plt.plot(Dmu, hcp25 - 0.5*Dmu, label='hcp - 0.25 ML')
plt.plot(Dmu, bridge25 - 0.5*Dmu, label='bridge - 0.25 ML')
plt.plot(Dmu, fcc1 - 0.5*Dmu, label='fcc - 1.0 ML')

plt.xlabel('$\Delta \mu O_2$ (eV)')
plt.ylabel('$\Delta G_{ads}$ (eV/O)')
plt.legend(loc='best')
plt.savefig('images/atomistic-thermo-adsorption.png')
```

```
None
```

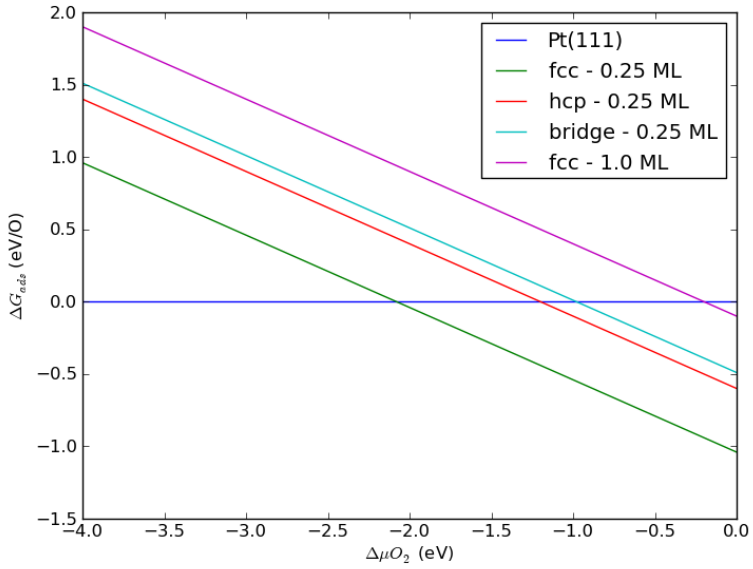


Figure 61: Effect of oxygen chemical potential on the adsorption energy.

7.3 Atomistic therodynamics and multiple reactions

In [38] we considered multiple reactions in an atomistic thermodynamic framework. Let us consider these three reactions of dissociative adsorption of hydrogen and hydrogen disulfide, and consider how to compute the reaction energy for the third reaction.

1. $H_2 + 2* \leftrightharpoons 2H*$
2. $H_2S + 2* \leftrightharpoons H* + SH*$
3. $SH* + * \leftrightharpoons S* + H*$

The reaction energy of interest is $E_{rxn} = \mu_{S*} + \mu H* - \mu SH*$ The question is, what are these chemical potentials? We would like them in terms of pressures and temperature, preferrably of molecules that can be approximated as ideal gases. By equilibrium arguments we can say that $\mu_{H*} = \frac{1}{2}\mu_{H_2}$. It follows that at equilibrium:

$\mu_{H*} + \mu_{SH*} = \mu_{H_2S}$ and $\mu_{S*} + \mu_{S*} = \mu_{SH*}$.

From the first equation we have:

$\mu_{SH*} = \mu_{H_2S} - \frac{1}{2}\mu_{H_2}$

and from the second equation we have:

$\mu_{S*} = \mu_{SH*} - \mu_{H*} = \mu_{H_2S} - \mu_{H_2}$.

Thus, the chemical potentials of all these three adsorbed species depend on the chemical potentials of two gas-phase species. The chemical potentials of each of these gases can be defined as:

$\mu_{gas}(T, p) = E_{gas}(0K) + \delta\mu + kT \ln\left(p/p^0\right)$, as we have defined before, so that only simple DFT calculations are needed to estimate them.

8 Advanced electronic structure methods

8.1 DFT+U

VASP manual on DFT+U

8.1.1 Metal oxide oxidation energies with DFT+U

We will reconsider here the reaction (see Metal oxide oxidation energies) $2Cu_2O + O_2 \rightleftharpoons 4CuO$. We need to compute the energy of each species, now with DFT+U. In [37] they use a U parameter of 4 eV for Cu which gave the best agreement with the experimental value. We will also try that.

Cu_2O calculation with U=4.0

```
from jasp import *

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

with jasp('bulk/Cu2O') as calc:
    calc.clone('bulk/Cu2O-U=4.0')

with jasp('bulk/Cu2O-U=4.0') as calc:
    calc.set(ldau=True,   # turn DFT+U on
             ldautype=2,  # select simplified rotationally invariant option
             ldau_luj={'Cu':{'L':2,  'U':4.0, 'J':0.0},
                       'O':{'L':-1, 'U':0.0, 'J':0.0}},
             ldauprint=1,
             ibrion=-1,  #do not rerelax
             nsw=0)
    calc.calculate()
    print calc
```

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/bulk/Cu2O-U=4.0
  converged: True
  Energy = -22.228203 eV

  Unit cell vectors (angstroms)
        x       y       z      length
  a0 [ 4.270  0.000  0.000] 4.270
  a1 [ 0.000  4.270  0.000] 4.270
  a2 [ 0.000  0.000  4.270] 4.270
  a,b,c,alpha,beta,gamma (deg): 4.270 4.270 4.270 90.0 90.0 90.0
  Unit cell volume = 77.854 Ang^3
  Stress (GPa):xx,    yy,    zz,    yz,    xz,    xy
             0.018  0.018  0.018 -0.000 -0.000 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
   0    Cu  [0.000      0.000      0.000]   0   0.00      T T T
   1    Cu  [2.135      2.135      0.000]   0   0.00      T T T
   2    Cu  [2.135      0.000      2.135]   0   0.00      T T T
```

```
   3    Cu  [0.000      2.135      2.135]   0   0.00      T T T
   4    O   [1.067      1.067      1.067]   0   0.00      T T T
   5    O   [3.202      3.202      3.202]   0   0.00      T T T
-----------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 37
           nsw: 0
        ibrion: -1
      ldautype: 2
          isif: 3
     ldauprint: 1
         encut: 400.0
          ldau: True
        magmom: None
         ldaul: [-1.0, 2.0]
         ldauj: [0.0, 0.0]
         ldauu: [0.0, 4.0]
      ldau_luj: {'O': {'J': 0.0, 'U': 0.0, 'L': -1}, 'Cu': {'J': 0.0, 'U': 4.0, 'L': 2}}
          prec: Normal
          kpts: [8, 8, 8]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
Cu: potpaw_PBE/Cu/POTCAR (git-hash: a44c591415026f53deb16a99ca3f06b1e69be10b)
```

```
1 grep -A 3 "LDA+U is selected, type is set to LDAUTYPE" bulk/Cu2O-U=4.0/OUTCAR
```

```
  LDA+U is selected, type is set to LDAUTYPE =  2
    angular momentum for each species LDAUL =    -1    2
    U (eV)           for each species LDAUU =   0.0  4.0
    J (eV)           for each species LDAUJ =   0.0  0.0
```

CuO calculation with U=4.0

```
1 from jasp import *
2
3 from ase.visualize import *
4 from ase import Atom, Atoms
5
6 with jasp('bulk/CuO') as calc:
7     calc.clone('bulk/CuO-U=4.0')
```

```
8
9  with jasp('bulk/CuO-U=4.0') as calc:
10     calc.set(ldau=True,   # turn DFT+U on
11              ldautype=2,  # select simplified rotationally invariant option
12              ldau_luj={'Cu':{'L':2,  'U':4.0, 'J':0.0},
13                        'O':{'L':-1, 'U':0.0, 'J':0.0}},
14              ldauprint=1,
15              ibrion=-1,  #do not rerelax
16              nsw=0)
17     calc.calculate()
18     print calc
```

```
: ------------------------------
  VASP calculation from /home/jkitchin/dft-org/bulk/CuO-U=4.0
  converged: True
  Energy = -16.870130 eV

  Unit cell vectors (angstroms)
        x       y     z      length
  a0 [ 2.302 -1.776  0.046] 2.908
  a1 [ 2.302  1.776  0.046] 2.908
  a2 [-0.762  0.000  5.087] 5.144
  a,b,c,alpha,beta,gamma (deg): 2.908 2.908 5.144 95.8 95.8 95.8
  Unit cell volume = 41.730 Ang^3
  Stress (GPa):xx,   yy,    zz,    yz,    xz,    xy
               0.030  0.018  0.027 -0.000 -0.007 -0.000
 Atom#  sym       position [x,y,z]         tag  rmsForce constraints
   0    Cu  [1.151      0.888      0.023]   0   0.00      T T T
   1    Cu  [0.770     -0.888      2.566]   0   0.00      T T T
   2    O   [2.111     -0.168      1.318]   0   0.04      T T T
   3    O   [1.730      0.168      3.861]   0   0.04      T T T
-------------------------------------------------

INCAR Parameters:
-----------------
        nbands: 23
           nsw: 0
        ibrion: -1
      ldautype: 2
          isif: 3
     ldauprint: 1
          encut: 400.0
          ldau: True
        magmom: None
          ldaul: [-1.0, 2.0]
          ldauj: [0.0, 0.0]
          ldauu: [0.0, 4.0]
      ldau_luj: {'O': {'J': 0.0, 'U': 0.0, 'L': -1}, 'Cu': {'J': 0.0, 'U': 4.0, 'L': 2}}
          prec: Normal
```

```
          kpts: [8, 8, 8]
    reciprocal: False
            xc: PBE
           txt: -
         gamma: False

Pseudopotentials used:
----------------------
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)
Cu: potpaw_PBE/Cu/POTCAR (git-hash: a44c591415026f53deb16a99ca3f06b1e69be10b)
```

Reaction energy calculation with DFT+U

```
from jasp import *

# don't forget to normalize your total energy to a formula unit. Cu2O
# has 3 atoms, so the number of formula units in an atoms is
# len(atoms)/3.
with jasp('bulk/Cu2O-U=4.0') as calc:
    atoms = calc.get_atoms()
    cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)

with jasp('bulk/CuO-U=4.0') as calc:
    atoms = calc.get_atoms()
    cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)

# make sure to use the same cutoff energy for the O2 molecule!
with jasp('molecules/O2-sp-triplet-400') as calc:
    atoms = calc.get_atoms()
    o2_energy = atoms.get_potential_energy()

rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
print 'Reaction energy  = {0} eV'.format(rxn_energy)
print 'Corrected energy = {0} eV'.format(rxn_energy - 1.36)
```

```
Reaction energy  = -1.663819 eV
Corrected energy = -3.023819 eV
```

This is still not in quantitative agreement with the result in [37], which at U=4 eV is about -3.14 eV (estimated from a graph). We have not applied the O_2 correction here yet. In that paper, they apply a constant shift of -1.36 eV per O_2. After we apply that correction, we agree within 0.12 eV, which is pretty good considering we have not checked for convergence.

How much does U affect the reaction energy? It is reasonable to consider how sensitive our results are to the U parameter. We do that here.

```
from jasp import *
for U in [2.0, 4.0, 6.0]:
    ## Cu2O ########################################
    with jasp('bulk/Cu2O') as calc:
        calc.clone('bulk/Cu2O-U={0}'.format(U))

    with jasp('bulk/Cu2O-U={0}'.format(U)) as calc:
        calc.set(ldau=True,   # turn DFT+U on
```

```
                ldautype=2,  # select simplified rotationally invariant option
                ldau_luj={'Cu':{'L':2,  'U':U, 'J':0.0},
                          'O':{'L':-1, 'U':0.0, 'J':0.0}},
                ldauprint=1,
                ibrion=-1,  #do not rerelax
                nsw=0)
        atoms = calc.get_atoms()
        cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)

    ## CuO ########################################
    with jasp('bulk/CuO') as calc:
        calc.clone('bulk/CuO-U={0}'.format(U))

    with jasp('bulk/CuO-U={0}'.format(U)) as calc:
        calc.set(ldau=True,   # turn DFT+U on
                 ldautype=2,  # select simplified rotationally invariant option
                 ldau_luj={'Cu':{'L':2,  'U':U, 'J':0.0},
                           'O':{'L':-1, 'U':0.0, 'J':0.0}},
                ldauprint=1,
                ibrion=-1,  #do not rerelax
                nsw=0)
        atoms = calc.get_atoms()
        cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)

    ## O2 #########################################
    # make sure to use the same cutoff energy for the O2 molecule!
    with jasp('molecules/O2-sp-triplet-400') as calc:
        atoms = calc.get_atoms()
        o2_energy = atoms.get_potential_energy()

    rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
    print 'U = {0}  reaction energy = {1}'.format(U,rxn_energy - 1.99)
```

```
U = 2.0  reaction energy = -3.876906
U = 4.0  reaction energy = -3.653819
U = 6.0  reaction energy = -3.397605
```

In [37], the difference in reaction energy from U=2 eV to U=4 eV was about 0.5 eV (estimated from graph). Here we see a range of 0.48 eV from U=2 eV to U=4 eV. Note that for U=0 eV, we had a (corrected reaction energy of -3.96 eV). Overall, the effect of adding U decreases this reaction energy.

This example highlights the challenge of using an approach like DFT+U. On one hand, U has a clear effect of changing the reaction energy. On the other hand, so does the correction factor for the O_2 binding energy. In [37] the authors tried to get the O_2 binding energy correction from oxide calculations where U is not important, so that it is decoupled from the non-cancelling errors that U fixes. See [17] for additional discussion of how to mix GGA and GGA+U results.

In any case, you should be careful to use well converged results to avoid compensating for convergence errors with U.

8.2 Hybrid functionals

8.2.1 FCC Ni DOS

This example is adapted from http://cms.mpi.univie.ac.at/wiki/index.php/FccNi_DOS

```
from jasp import *
from ase.lattice.cubic import FaceCenteredCubic
```

```
from ase.dft import DOS

atoms = FaceCenteredCubic(directions=[[0,1,1],
                                      [1,0,1],
                                      [1,1,0]],
                                      size=(1,1,1),
                                      symbol='Ni')
atoms[0].magmom = 1

with jasp('bulk/Ni-PBE',
          ismear=-5,
          kpts=(5,5,5),
          xc='PBE',
          ispin=2,lorbit=11,
          atoms=atoms) as calc:
    print 'PBE energy:   ',atoms.get_potential_energy()
    dos = DOS(calc,width=0.2)
    e_pbe = dos.get_energies()
    d_pbe = dos.get_dos()

    calc.clone('bulk/Ni-PBE0')
    calc.clone('bulk/Ni-HSE06')

with jasp('bulk/Ni-PBE0') as calc:
     calc.set(lhfcalc=True,
              algo='D',
              time=0.4)
     atoms = calc.get_atoms()
     print 'PBE0 energy:  ',atoms.get_potential_energy()
     dos = DOS(calc,width=0.2)
     e_pbe0 = dos.get_energies()
     d_pbe0 = dos.get_dos()

with jasp('bulk/Ni-HSE06') as calc:
     calc.set(lhfcalc=True,
              hfscreen=0.2,
              algo='D', time=0.4)
     atoms = calc.get_atoms()
     print 'HSE06 energy: ', atoms.get_potential_energy()
     dos = DOS(calc,width=0.2)
     e_hse06 = dos.get_energies()
     d_hse06 = dos.get_dos()

import pylab as plt
plt.plot(e_pbe, d_pbe, label='PBE')
plt.plot(e_pbe0, d_pbe0, label='PBE0')
plt.plot(e_hse06, d_hse06, label='HSE06')
plt.xlabel('energy [eV]')
plt.ylabel('DOS')
plt.legend()
plt.savefig('images/ni-dos-pbe-pbe0-hse06.png')
```

```
PBE energy:    -5.530247
PBE0 energy:   -6.848931
HSE06 energy:  -6.293369
```

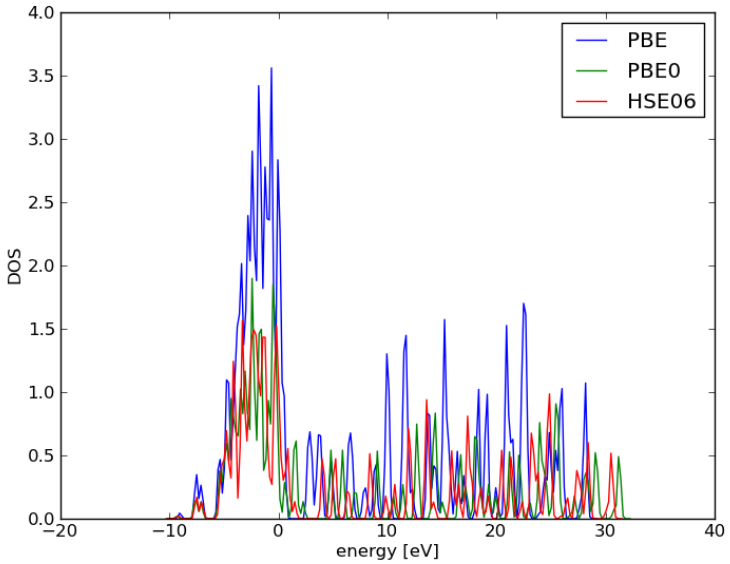


Figure 62: Comparison of DOS from GGA, and two hybrid GGAs (PBE0 ad HSE06).

8.3 TODO vdW

http://cms.mpi.univie.ac.at/vasp/vasp/vdW_DF_functional_Langreth_Lundqvist_et_al.html

8.4 TODO DFT+D

http://cms.mpi.univie.ac.at/vasp/vasp/DFT_D2_method_Grimme.html [22]

8.5 ELF

Need better intro here.

```
# compute ELF for CF4
from jasp import *
from ase.data.molecules import molecule
from enthought.mayavi import mlab

atoms = molecule('CF4')
atoms.center(vacuum=5)

with jasp('molecules/cf4-elf',
          encut=350,
          prec='high',
          ismear=0,
          sigma=0.01,
          xc='PBE',
          lelf=True,
          atoms=atoms) as calc:
    calc.calculate()

    x,y,z,elf = calc.get_elf()
    mlab.contour3d(x,y,z,elf,contours=[0.3])
    mlab.savefig('../../images/cf4-elf-3.png')
```

```
22
23      mlab.figure()
24      mlab.contour3d(x,y,z,elf,contours=[0.75])
25      mlab.savefig('../../images/cf4-elf-75.png')
```

```
None
```

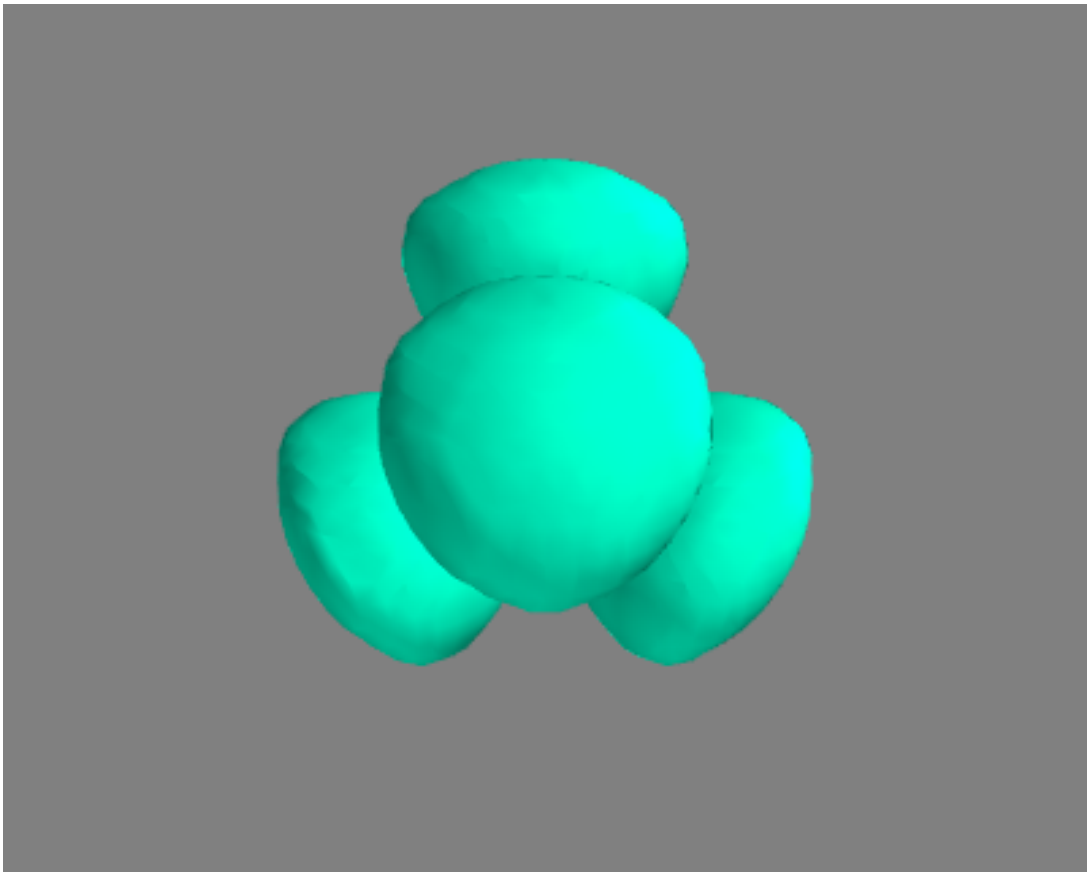


Figure 63: ELF for an isosurface of 0.3 for CF_4.

These images (Figure 63 and 64) are basically consistent with those in Reference [32].

8.6 TODO Charge partitioning schemes

8.7 TODO Modeling Core level shifts

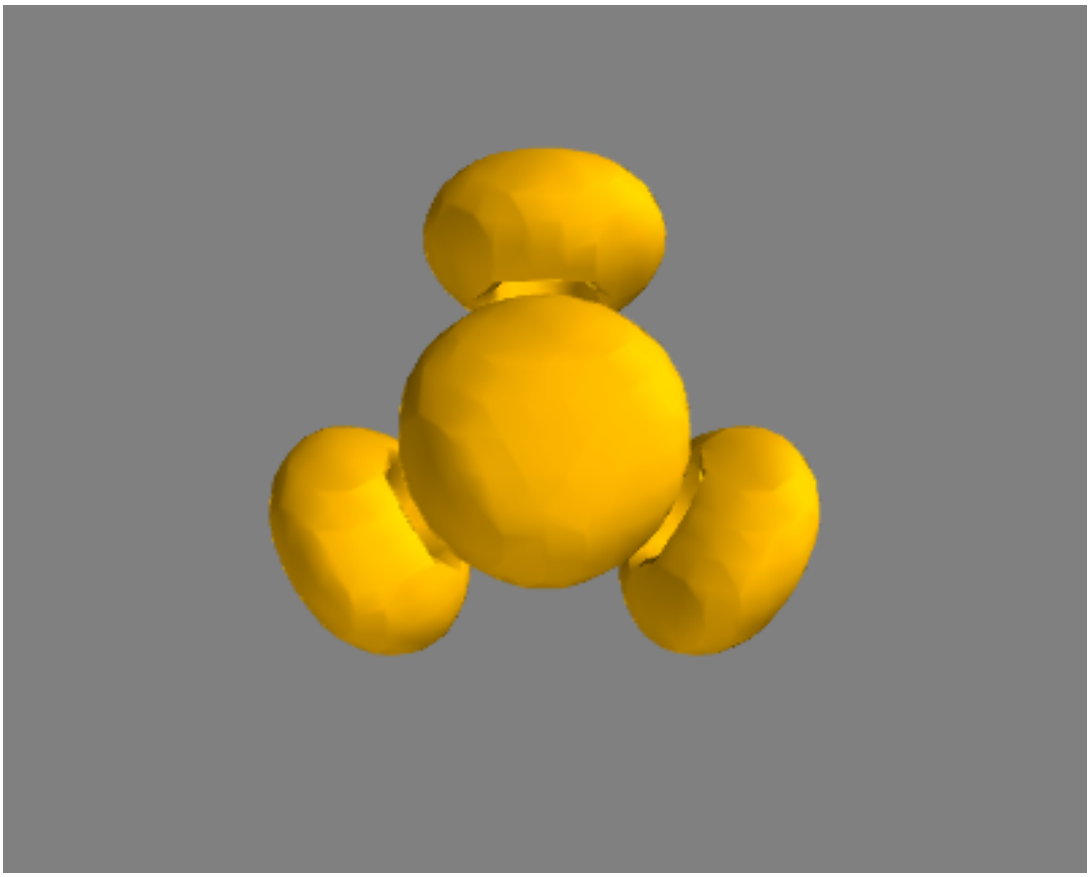


Figure 64: ELF for an isosurface of 0.75 for CF_4.

9 Acknowledgments

I would like to thank Zhongnan Xu for sending me some examples on magnetism. Alan McGaughey and Lars Grabow for sending me some NEB examples. Matt Curnan for examples of phonons.

Many thanks to students in my class who have pointed out typos, places of confusion, etc... These include:

- Bruno Alfa

10 Appendices

10.1 Recipes

10.1.1 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.

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

ind2del = [atom.index for atom in atoms if atom.symbol=='H']
print 'Indices to delete: ',ind2del

del atoms[ind2del]
print atoms
```

```
Atoms(symbols='C2OH6', positions=..., cell=[1.0, 1.0, 1.0], pbc=[False, False, False])
Indices to delete:  [3, 4, 5, 6, 7, 8]
Atoms(symbols='C2O', positions=..., cell=[1.0, 1.0, 1.0], pbc=[False, False, False])
```

10.1.2 Advanced tagging

We can label atoms with integer tags to help identify them later, e.g. which atoms are adsorbates, or surface atoms, or near an adsorbate, etc... We might want to refer to those atoms later for electronic structure, geometry analysis, etc...

The method uses integer tags that are powers of two, and then uses binary operators to check for matches. & is a bitwise AND. The key to understanding this is to look at the tags in binary form. The tags [1 2 4 8] can be represented by a binary string:

```
1 = [1 0 0 0]
2 = [0 1 0 0]
4 = [0 0 1 0]
8 = [0 0 0 1]
```

So, an atom tagged with 1 and 2 would have a tag of [1 1 0 0] or a tag of 3.

```
'''
adapted from https://listserv.fysik.dtu.dk/pipermail/campos/2004-September/001155.html
'''

from ase import *
from ase.io import write
from ase.lattice.surface import *
from ase.constraints import FixAtoms

# the bcc111 function automatically tags atoms
slab = bcc111('W',
              a=3.92,       # W lattice constant
              size=(2,2,6), #6-layer slab in 2x2 configuration
              vacuum=10.0)

#reset tags to be powers of two
slab.set_tags([2**a.get_tag() for a in slab])

# we had 6 layers, so we create new tags starting at 7
# Note you must use powers of two for all the tags!
LAYER1 = 2
ADSORBATE = 2**7
FREE = 2**8
NEARADSORBATE = 2**9

# let us tag LAYER1 atoms to be FREE too. we can address it by LAYER1 or FREE
```

```
tags = slab.get_tags()
for i,tag in enumerate(tags):
    if tag == LAYER1:
        tags[i] += FREE
slab.set_tags(tags)

#create a CO molecule
co= Atoms([Atom('C',[0., 0., 0. ], tag=ADSORBATE),
           Atom('O',[0., 0., 1.1], tag=ADSORBATE+FREE)]) #we will relax only O

add_adsorbate(slab,co,height=1.2,position='hollow')

#the adsorbate is centered between atoms 20, 21 and 22 (use
#view(slab)) and over atom12 let us label those atoms, so it is easy to
#do electronic structure analysis on them later.
tags = slab.get_tags() # len(tags) changed, so we reget them.
tags[12]+=NEARADSORBATE
tags[20]+=NEARADSORBATE
tags[21]+=NEARADSORBATE
tags[22]+=NEARADSORBATE
slab.set_tags(tags)
#update the tags
slab.set_tags(tags)

#extract pieces of the slab based on tags
#atoms in the adsorbate
ads = slab[(slab.get_tags() & ADSORBATE) == ADSORBATE]

#atoms in LAYER1
layer1 = slab[(slab.get_tags() & LAYER1) == LAYER1]

#atoms defined as near the adsorbate
nearads = slab[(slab.get_tags() & NEARADSORBATE) == NEARADSORBATE]

#atoms that are free
free = slab[(slab.get_tags() & FREE) == FREE]

#atoms that are FREE and part of the ADSORBATE
freeads = slab[(slab.get_tags() & FREE+ADSORBATE) == FREE+ADSORBATE]

#atoms that are NOT FREE
notfree = slab[(slab.get_tags() & FREE) != FREE]

constraint = FixAtoms(mask = (slab.get_tags() & FREE) != FREE)
slab.set_constraint(constraint)
write('images/tagged-bcc111.png',slab,rotation='-90x',show_unit_cell=2)
```

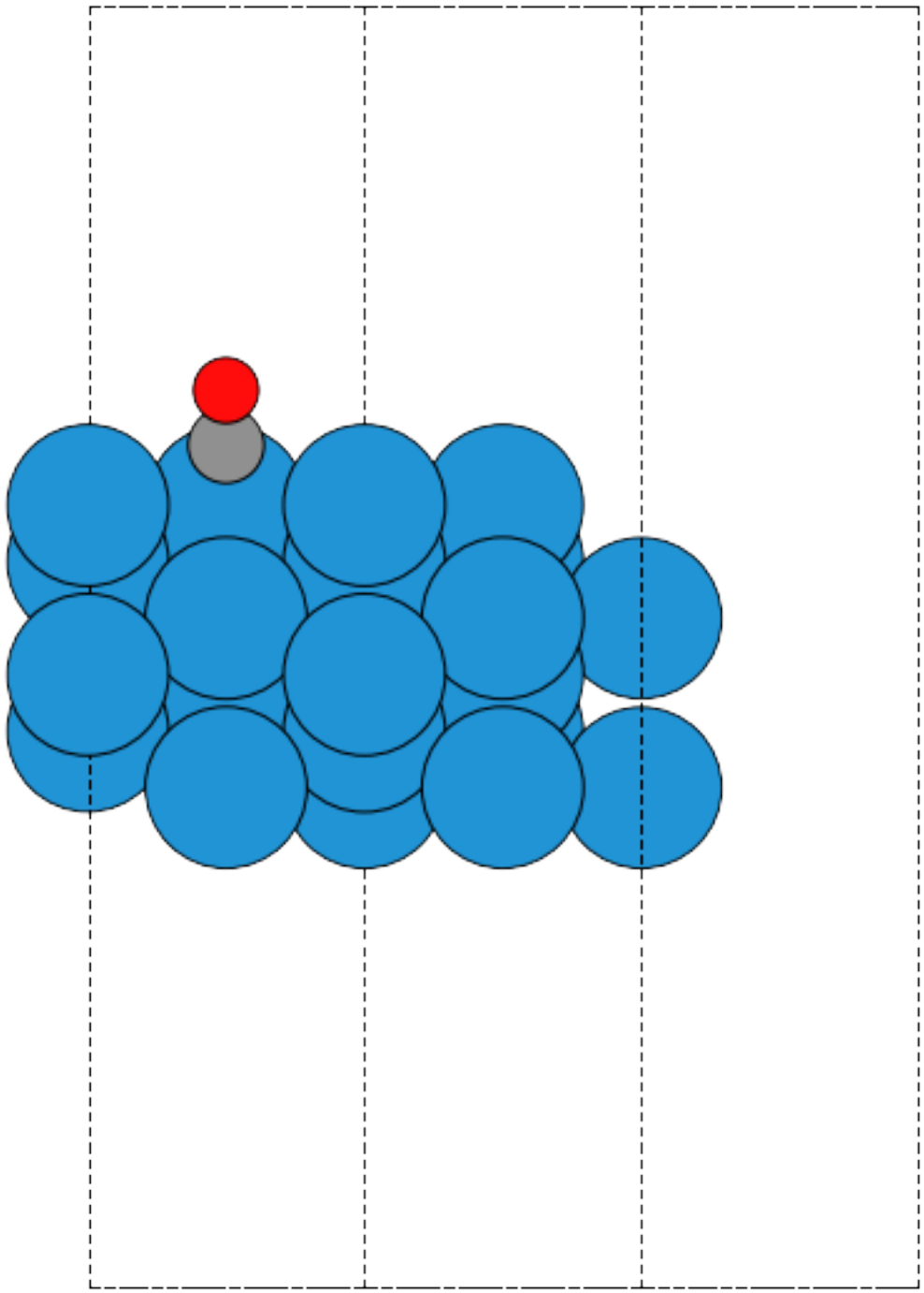


Figure 65: the tagged bcc(111) structure created above.

10.1.3 Using units in ase

`ase` uses a base set of atomic units. Other units are defined in terms of those units, and you can easily convert to alternative units by dividing your quantity in atomic units by the units

you want.

Not too many units are defined: ['A', 'AUT', 'Ang', 'Angstrom', 'Bohr', 'C', 'Debye', 'GPa', 'Ha', 'Hartree', 'J', 'Pascal', 'Ry', 'Rydberg', 'alpha', 'cm', 'eV', 'erg', 'fs', 'kB', 'kJ', 'kcal', 'kg', 'm', 'meV', 'mol', 'nm', 's', 'second']

It is not that hard to define your own derived units though. Note these are only conversion factors. No unit algebra is enforced!

```
from ase.units import *

d = 1*Angstrom
print ' d = {0} nm'.format(d/nm)

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

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

# derived units
minute = 60*s
hour = 60*minute

#convert 10 hours to minutes
print '10 hours = {0} minutes'.format(10*hour/minute)
```

```
 d = 0.1 nm
1 eV = 0.036749 Hartrees
1 eV = 0.073499 Rydbergs
1 eV = 96.485309 kJ/mol
1 eV = 23.060542 kcal/mol
1 Hartree = 627.509541 kcal/mol
1 Rydberg = 13.605698 eV
10 hours = 600.0 minutes
```

10.1.4 Extracting parts of an array

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

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

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

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

```
6
7  x = np.linspace(0,6,100)
8  y = np.cos(x)
9
10 plot(x,y,label='full')
11
12 ind = (x>2) & (x<4)
13
14 subx = x[ind]
15 suby = y[ind]
16
17 plot(subx,suby,'bo',label='sliced')
18 xlabel('x')
19 ylabel('cos(x)')
20 legend(loc='lower right')
21 savefig('images/np-array-slice.png')
```

```
None
```

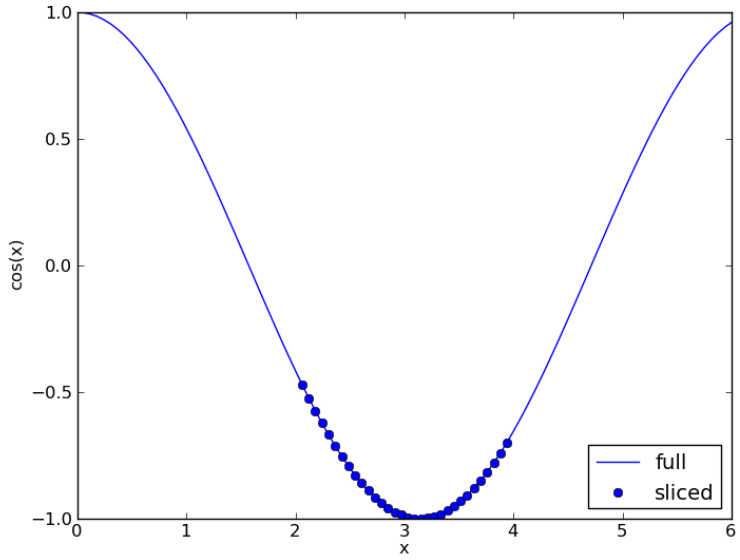


Figure 66: Example of slicing out part of an array. The solid line represents the whole array, and the symbols are the array between $2 < x < 4$.

The expression x> 2 returns an array of booleans (True where the element of x is greater than 2, and False where it is not) equal in size to x. Similarly $x < 4$ returns a boolean array where x is less than 4. We take the logical `and` of these two boolean arrays to get another boolean array where both conditions are True (i.e. $x < 2$ and $x > 4$). This final boolean array is `True` for the part of the arrays we are interested in, and we can use it to extract the subarrays we want.

10.1.5 Statistics

Confidence intervals

```
from numpy import *
from scipy.stats.distributions import  t

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

#Find 95% prediction interval for next measurement

alpha = 1.0 - 0.95

pred_interval = t.ppf(1-alpha/2.,dof)*std_x*sqrt(1.+1./n)

s = ['We are 95%% confident the next measurement',
       ' will be between %1.3f and %1.3f']
print ''.join(s) % (avg_x - pred_interval, avg_x + pred_interval)
```

```
We are 95% confident the next measurement will be between 16.076 and 16.124
```

10.1.6 Curve fitting

Linear fitting

```
#examples of linear curve fitting using least squares
import numpy as np

xdata = np.array([0.,1.,2.,3.,4.,5.,6.])
ydata = np.array([0.1, 0.81, 4.03, 9.1, 15.99, 24.2, 37.2])

#fit a third order polynomial
from pylab import polyfit, plot, show, legend, savefig
pars = polyfit(xdata,ydata,3)
print 'pars from polyfit: {0}'.format(pars)

## numpy method returns more data
A = np.column_stack([xdata**3,xdata**2,xdata,np.ones(len(xdata),np.float)])
pars_np,resids,rank,s = np.linalg.lstsq(A,ydata)
print 'pars from np.linalg.lstsq: {0}'.format(pars_np)

'''
we are trying to solve Ax = b for x in the least squares sense. There
are more rows in A than elements in x so, we can left multiply each
side by A^T, and then solve for x with an inverse.

A^TAx = A^Tb
x = (A^TA)^-1 A^T b
'''
# not as pretty but equivalent!
pars_man= np.dot(np.linalg.inv(np.dot(A.T,A)), np.dot(A.T,ydata))
print 'pars from linear algebra: {0}'.format(pars_man)

#but, it is easy to fit an exponential function to it!
# y = a*exp(x)+b
Aexp = np.column_stack([np.exp(xdata), np.ones(len(xdata),np.float)])
pars_exp=np.dot(np.linalg.inv(np.dot(Aexp.T,Aexp)), np.dot(Aexp.T,ydata))

plot(xdata,ydata,'ro')
fity = np.dot(A,pars)
plot(xdata,fity,'k-',label='poly fit')
plot(xdata,np.dot(Aexp,pars_exp),'b-',label='exp fit')
legend()
savefig('images/curve-fit-1.png')
```

```
pars from polyfit: [ 0.04861111  0.63440476  0.61365079 -0.08928571]
pars from np.linalg.lstsq: [ 0.04861111  0.63440476  0.61365079 -0.08928571]
pars from linear algebra: [ 0.04861111  0.63440476  0.61365079 -0.08928571]
```

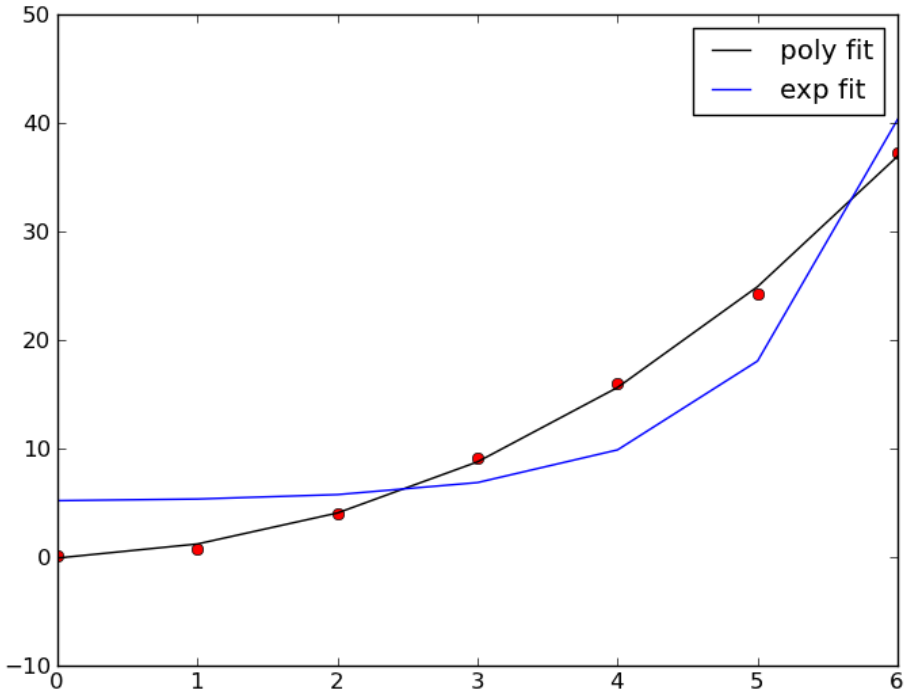


Figure 67: Example of linear least-squares curve fitting.

10.1.7 Nonlinear curve fitting

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

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

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

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

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

    return E
```

```
18
19 def objective(pars,y,x):
20     #we will minimize this function
21     err =  y - Murnaghan(pars,x)
22     return err
23
24 x0 = [ -56., 0.54, 2., 16.5] #initial guess of parameters
25
26 plsq = leastsq(objective, x0, args=(energies,vols))
27
28 print plsq
29
30 from pylab import *
31 plot(vols,energies,'ro')
32
33 #plot the fitted curve on top
34 x = np.linspace(min(vols),max(vols),50)
35 y = Murnaghan(plsq[0],x)
36 plot(x,y,'k-')
37 xlabel('Volume')
38 ylabel('energy')
39 savefig('images/nonlinear-curve-fitting.png')
```

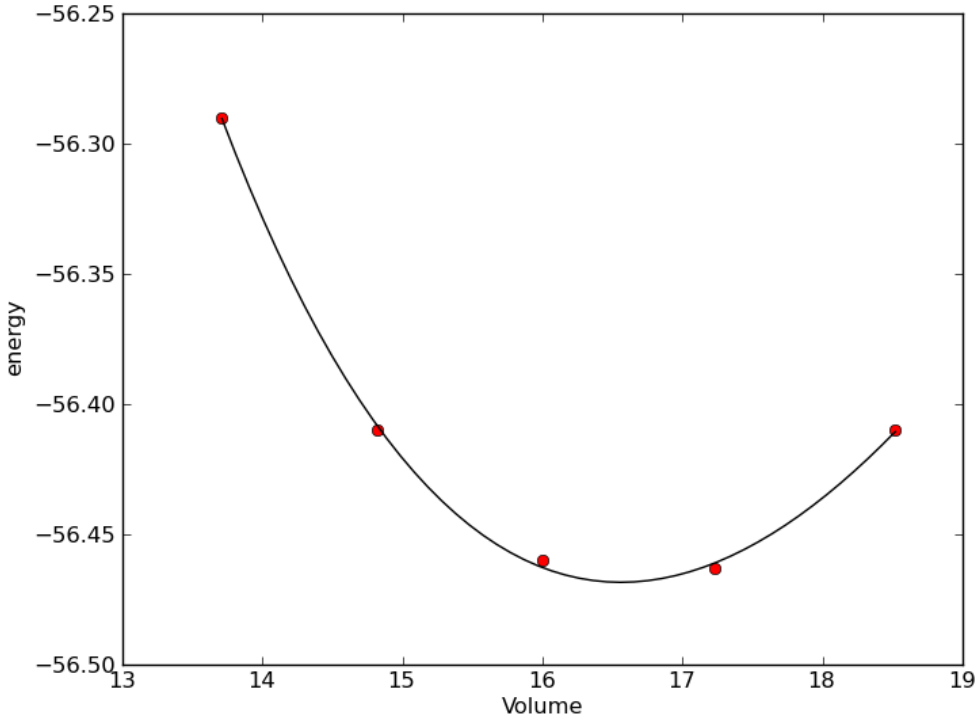


Figure 68: Example of least-squares non-linear curve fitting.

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

10.1.8 Nonlinear curve fitting by direct least squares mimimization

```
from scipy.optimize import fmin
import numpy as np

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

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

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

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

    return E

def objective(pars,vol):
    #we will minimize this function
    err =  energies - Murnaghan(pars,vol)
    return np.sum(err**2) #we return the summed squared error directly

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

plsq = fmin(objective,x0,args=(volumes,)) #note args is a tuple

print 'parameters = {0}'.format(plsq)

from pylab import *
plot(volumes,energies,'ro')

#plot the fitted curve on top
x = np.linspace(min(volumes),max(volumes),50)
y = Murnaghan(plsq,x)
plot(x,y,'k-')
savefig('images/nonlinear-fitting-lsq.png')
```

```
Optimization terminated successfully.
         Current function value: 0.000020
         Iterations: 137
         Function evaluations: 240
parameters = [-56.46932645   0.59141447   1.9044796   16.59341303]
```

10.1.9 Nonlinear curve fitting with confidence intervals

```
# Nonlinear curve fit with confidence interval
from numpy import *
from scipy.optimize import curve_fit
from scipy.stats.distributions import  t

'''
fit this equation to data
y = c1 exp(-x) + c2*x

this is actually a linear regression problem, but it is convenient to
use the nonlinear fitting routine because it makes it easy to get
confidence intervals. The downside is you need an initial gues.

from Matlab
b =
```

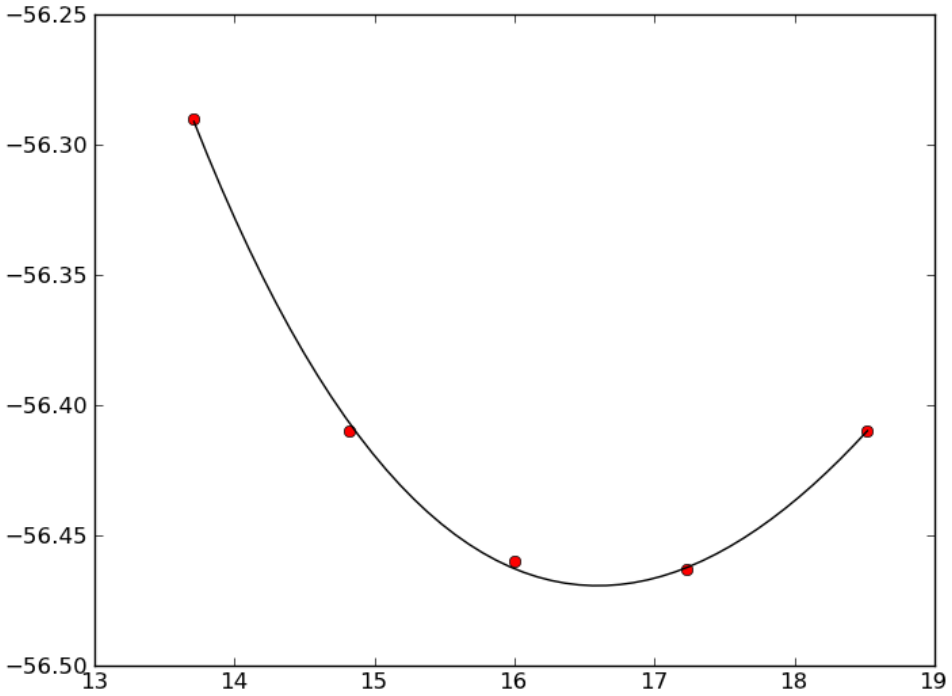


Figure 69: Fitting a nonlinear function.

```
16
17      4.9671
18      2.1100
19
20
21  bint =
22
23      4.6267    5.3075
24      1.7671    2.4528
25  '''
26
27  x = array([ 0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
28  y = array([ 4.70192769,  4.46826356,  4.57021389,  4.29240134,  3.88155125,
29              3.78382253,  3.65454727,  3.86379487,  4.16428541,  4.06079909])
30
31  # this is the function we want to fit to our data
32  def func(x,c0, c1):
33      return c0*exp(-x) + c1*x
34
35  pars, pcov = curve_fit(func, x, y, p0=[4.96, 2.11])
36
37  alpha = 0.05 # 95% confidence interval
38
39  n = len(y)    # number of data points
40  p = len(pars) # number of parameters
41
42  dof = max(0, n-p) # number of degrees of freedom
43
44  tval = t.ppf(1.0-alpha/2., dof) # student-t value for the dof and confidence level
```

```
45
46 for i, p,var in zip(range(n), pars, diag(pcov)):
47     sigma = var**0.5
48     print 'c{0}: {1} [{2}  {3}]'.format(i, p,
49                                        p - sigma*tval,
50                                        p + sigma*tval)
51
52 import matplotlib.pyplot as plt
53 plt.plot(x,y,'bo ')
54 xfit = linspace(0,1)
55 yfit = func(xfit, pars[0], pars[1])
56 plt.plot(xfit,yfit,'b-')
57 plt.legend(['data','fit'],loc='best')
58 plt.savefig('images/nonlin-fit-ci.png')
```

```
c0: 4.96713966556 [4.62674477087  5.30753456025]
c1: 2.10995112466 [1.76711622383  2.45278602549]
```

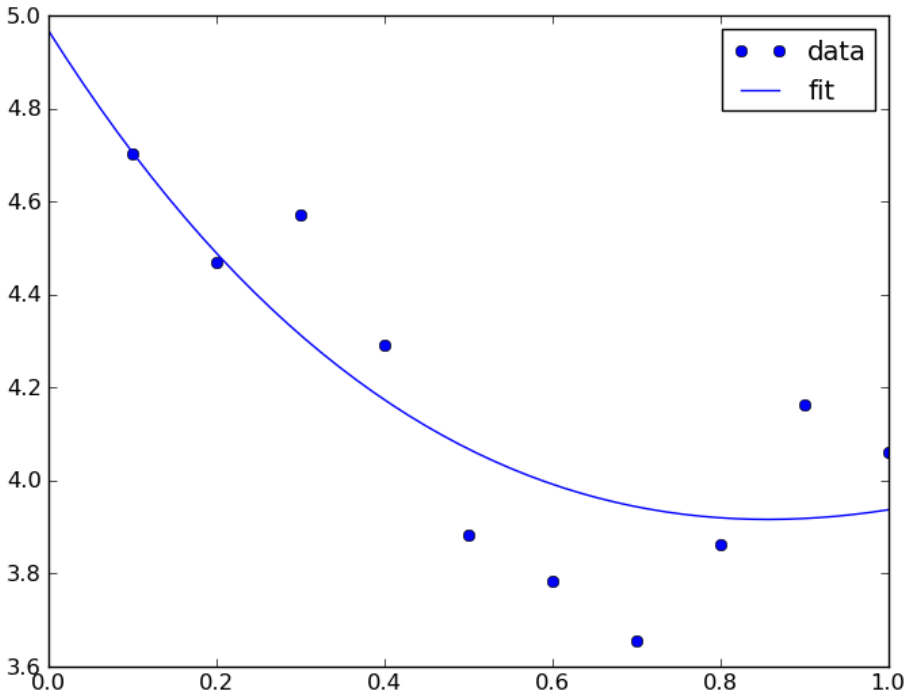


Figure 70: Nonlinear fit to data.

10.1.10 Interpolation with splines

When you do not know the functional form of data to fit an equation, you can still fit/interpolate with splines.

```
# use splines to fit and interpolate data
from scipy.interpolate import interp1d
from scipy.optimize import fmin
import numpy as np
import matplotlib.pyplot as plt


x = np.array([ 0,      1,      2,      3,      4    ])
y = np.array([ 0.,     0.308,  0.55,   0.546,  0.44 ])

# create the interpolating function
f = interp1d(x, y, kind='cubic', bounds_error=False)

# to find the maximum, we minimize the negative of the function. We cannot just multiply f by -1, so we create a new function
f2 = interp1d(x, -y, kind='cubic')
xmax = fmin(f2, 2.5)

xfit = np.linspace(0,4)

plt.plot(x,y,'bo')
plt.plot(xfit, f(xfit),'r-')
plt.plot(xmax, f(xmax),'g*')
plt.legend(['data','fit','max'], loc='best', numpoints=1)
plt.xlabel('x data')
plt.ylabel('y data')
plt.title('Max point = ({0:1.2f}, {1:1.2f})'.format(float(xmax),
                                                    float(f(xmax))))
plt.savefig('images/splinefit.png')
```

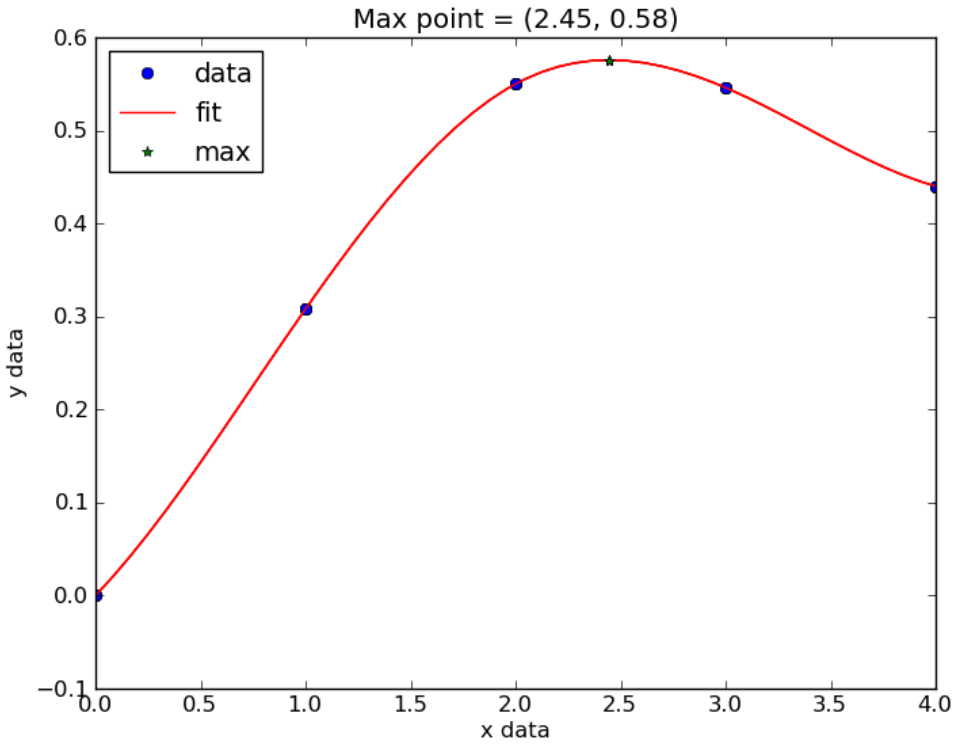


Figure 71: Illustration of a spline fit to data and finding the maximum point.

There are other good examples at http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html

10.1.11 Interpolation in 3D

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

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

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

Solution: Find the cube that contains the point, and is defined by points P1-P8 as shown in Figure 72.

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

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

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

    vcd = VaspChargeDensity()

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

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

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

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

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

```

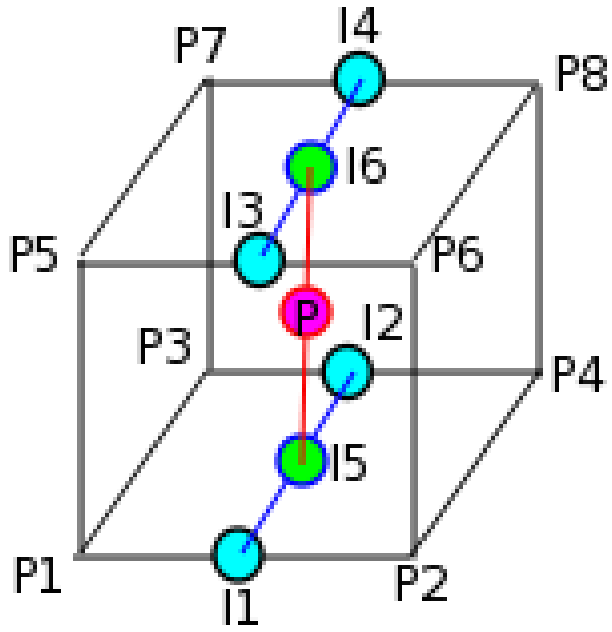


Figure 72: Trilinear interpolation scheme.

```
28      #now convert arrays back to unitcell shape
29      x = np.reshape(real[:, 0], (n0, n1, n2))
30      y = np.reshape(real[:, 1], (n0, n1, n2))
31      z = np.reshape(real[:, 2], (n0, n1, n2))
32
33  def interp3d(x,y,z,cd,xi,yi,zi):
34      '''
35      interpolate a cubic 3D grid defined by x,y,z,cd at the point
36      (xi,yi,zi)
37      '''
38
39      def get_index(value,vector):
40          '''
41          assumes vector ordered decreasing to increasing. A bisection
42          search would be faster.
43          '''
44          for i,val in enumerate(vector):
45              if val > value:
```

```
                return i-1
        return None

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

    a,b,c = xi, yi, zi

    i = get_index(a,xv)
    j = get_index(b,yv)
    k = get_index(c,zv)

    x1 = x[i,j,k]
    x2 = x[i+1,j,k]
    y1 = y[i,j,k]
    y2 = y[i,j+1,k]
    z1 = z[i,j,k]
    z2 = z[i,j,k+1]

    u1 = cd[i, j, k]
    u2 = cd[i+1, j, k]
    u3 = cd[i, j+1, k]
    u4 = cd[i+1, j+1, k]
    u5 = cd[i, j, k+1]
    u6 = cd[i+1, j, k+1]
    u7 = cd[i, j+1, k+1]
    u8 = cd[i+1, j+1, k+1]

    w1 = u2 + (u2-u1)/(x2-x1)*(a-x2)
    w2 = u4 + (u4-u3)/(x2-x1)*(a-x2)
    w3 = w2 + (w2-w1)/(y2-y1)*(b-y2)
    w4 = u5 + (u6-u5)/(x2-x1)*(a-x1)
    w5 = u7 + (u8-u7)/(x2-x1)*(a-x1)
    w6 = w4 + (w5-w4)/(y2-y1)*(b-y1)
    w7 = w3 + (w6-w3)/(z2-z1)*(c-z1)
    u = w7

    return u

pos = atoms.get_positions()

P1 = np.array([0.0, 5.0, 5.0])
P2 = np.array([9.0, 5.0, 5.0])

npoints = 60

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

R = [np.linalg.norm(p-P1) for p in points]

# interpolated line
icd = [interp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]

from pylab import *

plot(R,icd)
cR = np.linalg.norm(pos[0]-P1)
oR = np.linalg.norm(pos[1]-P1)
plot([cR,cR],[0,2],'r-') #markers for where the nuclei are
plot([oR,oR],[0,8],'r-')
xlabel('|R| ($\AA$)')
ylabel('Charge density (e/$\AA^3$)')
savefig('images/CO-charge-density.png')
show()
```

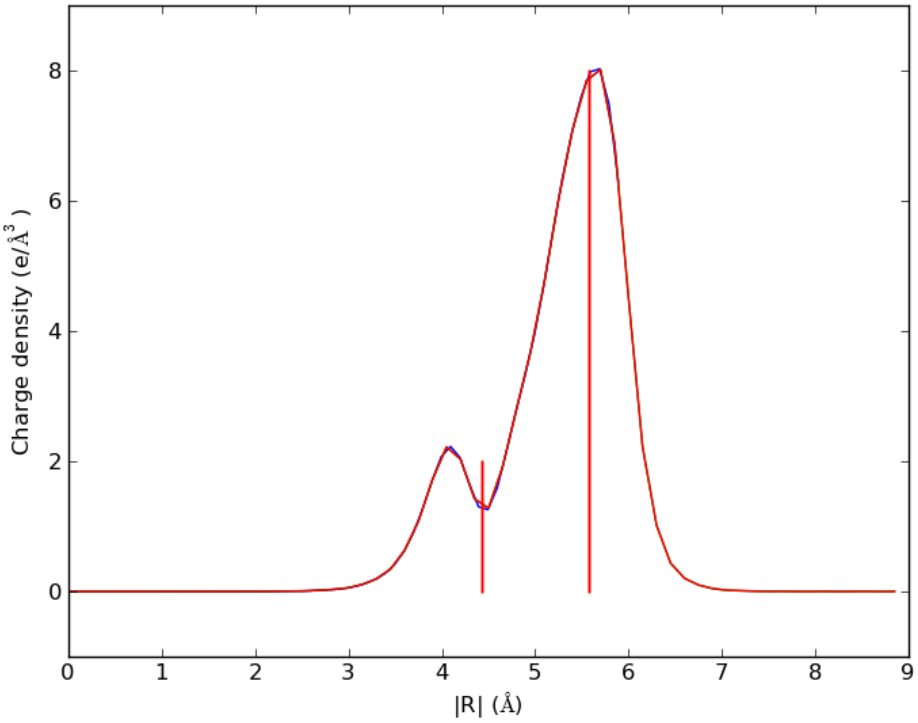


Figure 73: An example of interpolated charge density of A CO molecule along the axis of molecule.

To generalize this to non-cubic cells, we need to do interpolation along arbitrary vectors. The overall strategy is the same:

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

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

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

This function should work for any shape unit cell.
'''
from jasp import *
import bisect
import numpy as np
from pylab import plot, xlabel, ylabel, savefig, show

with jasp('molecules/co-centered') as calc:
    atoms = calc.get_atoms()
    x,y,z,cd = calc.get_charge_density()

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

```

```
    p = np.array([xi,yi,zi])

    #1D arrays of cooridinates
    xv = x[:,0,0]
    yv = y[0,:,0]
    zv = z[0,0,:]

    # we subtract 1 because bisect tells us where to insert the
    # element to maintain an ordered list, so we want the index to the
    # left of that point
    i = bisect.bisect_right(xv,xi) - 1
    j = bisect.bisect_right(yv,yi) - 1
    k = bisect.bisect_right(zv,zi) - 1

    #points at edge of cell. We only need P1, P2, P3, and P5
    P1 = np.array([x[i,j,k],y[i,j,k],z[i,j,k]])
    P2 = np.array([x[i+1,j,k],y[i+1,j,k],z[i+1,j,k]])
    P3 = np.array([x[i,j+1,k],y[i,j+1,k],z[i,j+1,k]])
    P5 = np.array([x[i,j,k+1],y[i,j,k+1],z[i,j,k+1]])

    #values of u at edge of cell
    u1 = u[i,j,k]
    u2 = u[i+1,j,k]
    u3 = u[i,j+1,k]
    u4 = u[i+1,j+1,k]
    u5 = u[i,j,k+1]
    u6 = u[i+1,j,k+1]
    u7 = u[i,j+1,k+1]
    u8 = u[i+1,j+1,k+1]

    #cell basis vectors, not the unit cell, but the voxel cell containing the point
    cbasis = np.array([P2-P1,
                       P3-P1,
                       P5-P1])

    #now get interpolated point in terms of the cell basis
    s = np.dot(np.linalg.inv(cbasis.T),np.array([xi,yi,zi])-P1)

    #now s = (sa, sb, sc) which are fractional coordinates in the vector space
    #next we do the interpolations
    ui1 = u1 + s[0]*(u2-u1)
    ui2 = u3 + s[0]*(u4-u3)

    ui3 = u5 + s[0]*(u6-u5)
    ui4 = u7 + s[0]*(u8-u7)

    ui5 = ui1 + s[1]*(ui2-ui1)
    ui6 = ui3 + s[1]*(ui4-ui3)

    ui7 = ui5 + s[2]*(ui6-ui5)

    return ui7

# compute a line with 60 points in it through these two points
P1 = np.array([0.0, 5.0, 5.0])
P2 = np.array([10.0, 5.0, 5.0])

npoints = 60

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

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

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

```

```
84
85
86  plot(R,icd)
87  pos = atoms.get_positions()
88  cR = np.linalg.norm(pos[0]-P1)
89  oR = np.linalg.norm(pos[1]-P1)
90  plot([cR,cR],[0,2],'r-') #markers for where the nuclei are
91  plot([oR,oR],[0,8],'r-')
92  xlabel('|R| ($\AA$)')
93  ylabel('Charge density (e/$\AA^3$)')
94  savefig('images/interpolated-charge-density.png')
95  show()
```

```
None
```

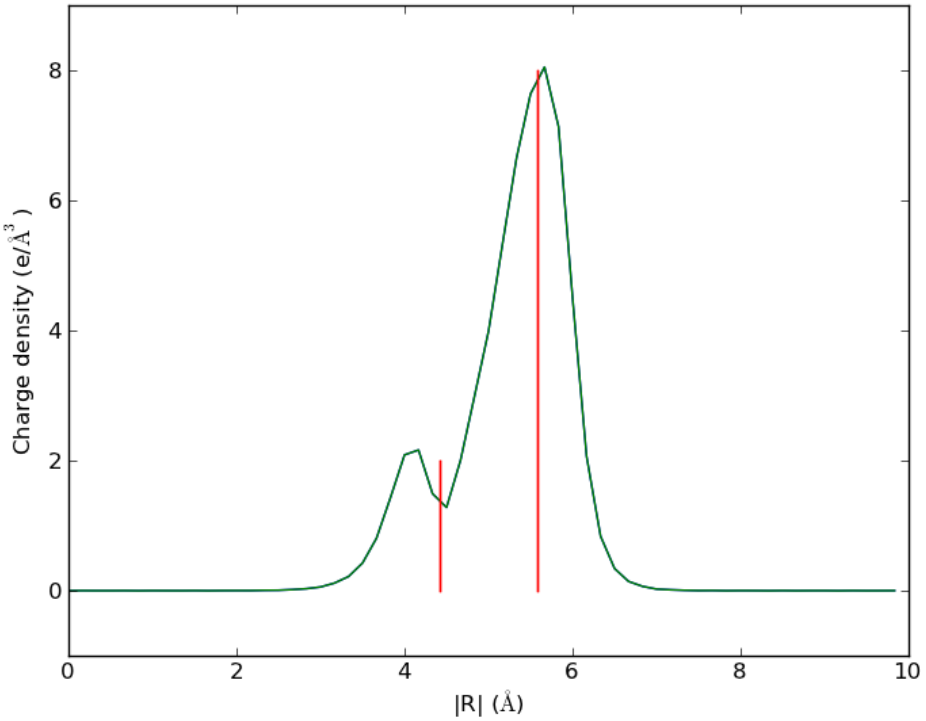


Figure 74: Interpolated charge density for a CO molecule.

10.1.12 Reading and writing data

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

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

and later you can read these arrays back in with:

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

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

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

Reading a datafile with

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

A standard approach would be to read in all the lines, skip the first two lines, split each line (remember each line is a string) at the ‘,’, and append the first field to one variable, and append the second field to another variable as an integer. For example:

```
v1 = []
v2 = []
lines = open('somefile','r').readlines()

for line in lines[2:]: #skip the first two lines
    fields = line.split(',')
    v1.append(fields[0]) #names
    v2.append(int(fields[1])) #number
```

Writing datafiles is easy too.

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

Some notes:

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

with care!

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

carriage return.

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

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

10.1.13 Integration

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

Integration can be used to calculate average properties of continuous distributions. Suppose for example, we have a density of states, ρ as a function of energy E. We can integrate the density of states to find the total number of states:

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

or, in python:

```
Nstates = np.trapz(rho,E)
```

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

The average energy of distribution is:

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

or, in python:

```
e_avg = np.trapz(rho*E,E)/np.trapz(rho,E)
```

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

The nth moment is defined by:

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

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

```
n = 2
mom_2 = np.trapz(rho*E**n,E)/np.trapz(rho,E)
```

10.1.14 Numerical differentiation

numpy has a function called `numpy.diff` that is similar to the one found in Matlab. It calculates the differences between the elements in your list, and returns a list that is one element shorter, which makes it unsuitable for plotting the derivative of a function.

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

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

'''
These are the brainless way to calculate numerical derivatives. They
```

```
work well for very smooth data. they are surprisingly fast even up to
10000 points in the vector.
'''

x = np.linspace(0.78,0.79,100)
y = np.sin(x)
dy_analytical = np.cos(x)
'''
let us use a forward difference method:
that works up until the last point, where there is not
a forward difference to use. there, we use a backward difference.
'''

tf1 = time.time()
dyf = [0.0]*len(x)
for i in range(len(y)-1):
    dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
#set last element by backwards difference
dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

print ' Forward difference took %1.1f seconds' % (time.time() - tf1)

'''and now a backwards difference'''
tb1 = time.time()
dyb = [0.0]*len(x)
#set first element by forward difference
dyb[0] = (y[0] - y[1])/(x[0] - x[1])
for i in range(1,len(y)):
    dyb[i] = (y[i] - y[i-1])/(x[i]-x[i-1])

print ' Backward difference took %1.1f seconds' % (time.time() - tb1)

'''and now, a centered formula'''
tc1 = time.time()
dyc = [0.0]*len(x)
dyc[0] = (y[0] - y[1])/(x[0] - x[1])
for i in range(1,len(y)-1):
    dyc[i] = (y[i+1] - y[i-1])/(x[i+1]-x[i-1])
dyc[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

print ' Centered difference took %1.1f seconds' % (time.time() - tc1)

'''
the centered formula is the most accurate formula here
'''

plot(x,y)
plot(x,dy_analytical,label='analytical derivative')
plot(x,dyf,'--',label='forward')
plot(x,dyb,'--',label='backward')
plot(x,dyc,'--',label='centered')

legend(loc='lower left')
savefig('images/simple-diffs.png')
show()
```

```
Forward difference took 0.0 seconds
Backward difference took 0.0 seconds
Centered difference took 0.0 seconds
```

Loops are usually not great for performance. Numpy offers some vectorized methods that allow us to compute derivatives without loops, although this comes at the mental cost of harder to understand syntax:

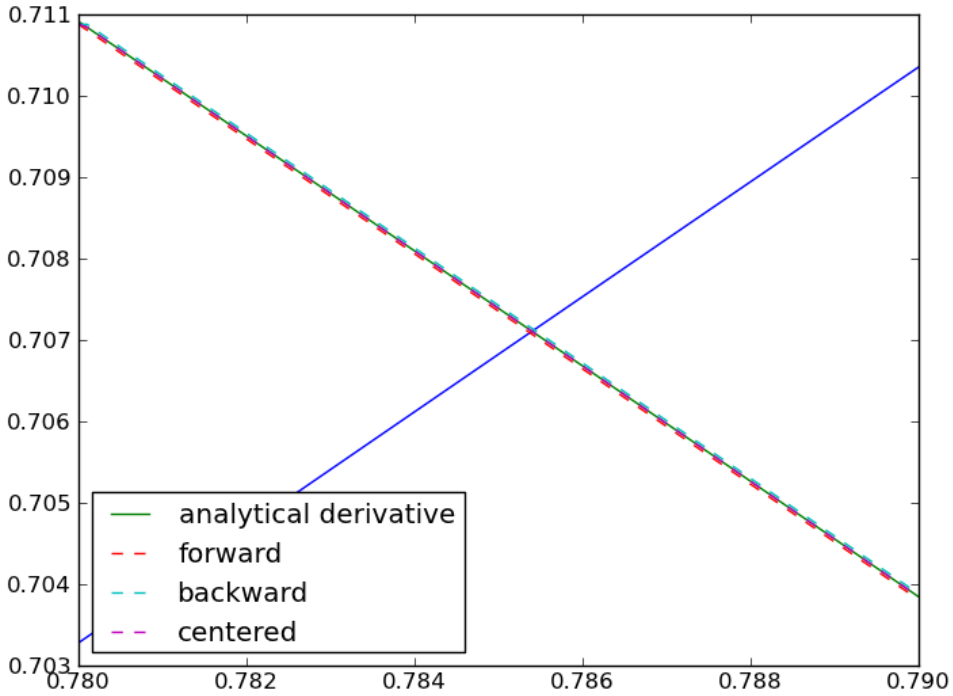


Figure 75: Comparison of different numerical derivatives.

```
import numpy as np
from pylab import *

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

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


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

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

plot(x,y)
plot(x,dy_analytical,label='analytical derivative')
plot(x,dy,label='forward diff')
plot(x,dy2,'k--',lw=2,label='centered diff')
legend(loc='lower left')
```

```
savefig('images/vectorized-diffs.png')
show()
```

```
None
```

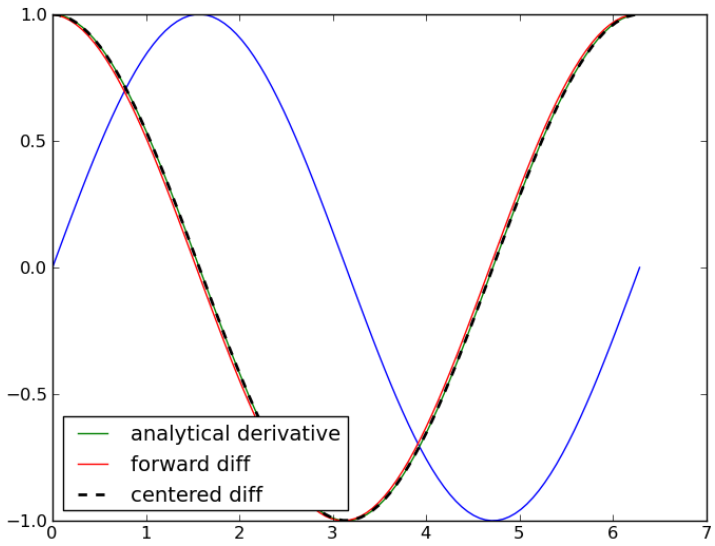


Figure 76: Comparison of different numerical derivatives.

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

Here is an example of a 4-point centered difference of some noisy data:

```
import numpy as np
from pylab import *

x = np.linspace(0,2*np.pi,100)
y = np.sin(x) + 0.1*np.random.random(size=x.shape)
dy_analytical = np.cos(x)

#2-point formula
dyf = [0.0]*len(x)
for i in range(len(y)-1):
    dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
#set last element by backwards difference
dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

'''
calculate dy by 4-point center differencing using array slices

\frac{y[i-2] - 8y[i-1] + 8[i+1] - y[i+2]}{12h}

```

```
    y[0] and y[1] must be defined by lower order methods
    and y[-1] and y[-2] must be defined by lower order methods
    '''

    dy = np.zeros(y.shape,np.float) #we know it will be this size
    h = x[1]-x[0] #this assumes the points are evenely spaced!
    dy[2:-2] = (y[0:-4] - 8*y[1:-3] + 8*y[3:-1] - y[4:])/(12.*h)

    dy[0] = (y[1]-y[0])/(x[1]-x[0])
    dy[1] = (y[2]-y[1])/(x[2]-x[1])
    dy[-2] = (y[-2] - y[-3])/(x[-2] - x[-3])
    dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

    plot(x,y)
    plot(x,dy_analytical,label='analytical derivative')
    plot(x,dyf,'r-',label='2pt-forward diff')
    plot(x,dy,'k--',lw=2,label='4pt-centered diff')
    legend(loc='lower left')
    savefig('images/multipt-diff.png')
    show()
```

```
None
```

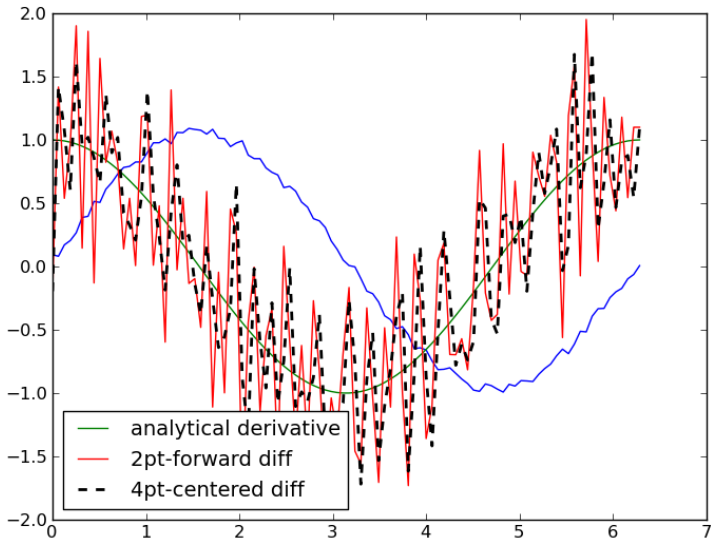


Figure 77: Comparison of 2 point and 4 point numerical derivatives.

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

FFT derivatives It is possible to perform derivatives using FFT:

```
    import numpy as np
    from pylab import *

    N = 101 #number of points
```

```
L = 2*np.pi #interval of data

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

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

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

you can use fft to calculate derivatives!
'''

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

k *= 2*np.pi/L

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

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

savefig('images/fft-der.png')
show()
```

```
None
```

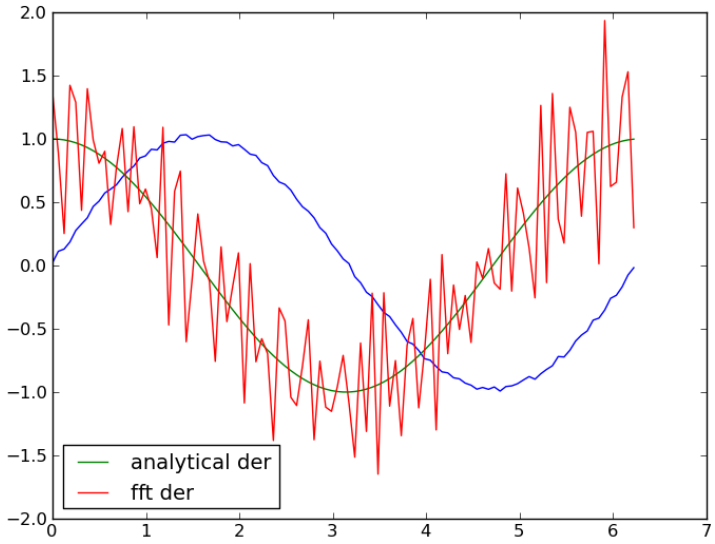


Figure 78: Comparison of FFT numerical derivatives.

This example does not show any major advantage in the quality of the derivative, and it is almost certain I would never remember how to do this off the top of my head.

10.1.15 NetCDF files

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

`Scientific.IO.NetCDF` (http://dirac.cnrs-orleans.fr/plone/software/scientificpython/)

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

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

10.1.16 python modules

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

csv writing:

```
import numpy as np

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

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

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

csv reading:

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

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

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

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

10.1.17 Writing and reading Excel files

Writing Excel files It is not too uncommon that it is convenient to do some analysis in Excel. We can create Excel files in python with `xlwt`. Google this module if you need to do this alot.

```
import numpy as np
import xlwt
```

```
3
4  wbk = xlwt.Workbook()
5  sheet = wbk.add_sheet('sheet 1')
6
7  volumes = np.array([13.72, 14.83, 16.0, 17.23, 18.52])
8  energies = np.array([-56.29, -56.41, -56.46, -56.46, -56.42])
9
10 for i, pair in enumerate(zip(volumes, energies)):
11     vol = pair[0]
12     energy = pair[1]
13     sheet.write(i,0,vol)
14     sheet.write(i,1,energy)
15 wbk.save('images/test-write.xls')
```

Reading Excel files We can also read Excel files (even on Linux!) with `xlrd`. Let us read in the data we just wrote. We wrote 5 volumes to column 0, and 5 energies to column 1.

```
1  import xlrd
2  wbk = xlrd.open_workbook('images/test-write.xls')
3  sheet1 = wbk.sheet_by_name('sheet 1')
4  print sheet1.col_values(0)
5  print sheet1.col_values(1)
```

```
[13.72, 14.83, 16.0, 17.23, 18.52]
[-56.29, -56.41, -56.46, -56.46, -56.42]
```

10.1.18 TODO making movies

1. using animate
2. using swftools (png2swf, pdf2swf)

10.2 Computational geometry

10.2.1 Changing coordinate systems

Let A, B, C be the unit cell vectors

$$A = A1x + A2y + A3z \tag{1}$$
$$B = B1x + B2y + B3z \tag{2}$$
$$C = C1x + C2y + C3z \tag{3}$$

and we want to find the vector $[s1s2s3]$ so that $P = s1A + s2B + s3C$
if we expand this, we get:

$$\begin{aligned} &s1A1x + s1A2y + s1A3z \\ &+ s2B1x + s2B2y + s2B3z \\ &+ s3C1x + s3C2y + s3C3z = p1x + p2y + p3z \end{aligned}$$

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

$$\begin{bmatrix} A1 & B1 & C1 \\ A2 & B2 & C2 \\ A3 & B3 & C3 \end{bmatrix} \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} = \begin{bmatrix} p1 \\ p2 \\ p3 \end{bmatrix} \tag{4}$$

or, in standard form:

$A^T s = p$

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:

```
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 basis 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()
```

```
Coordinates in new basis are:
[[ 0.167  0.167  0.408]
 [ 0.667  0.167  0.408]
 [ 0.167  0.667  0.408]
 [ 0.667  0.667  0.408]
 [-0.167  0.333  0.5  ]
 [ 0.333  0.333  0.5  ]
 [-0.167  0.833  0.5  ]
 [ 0.333  0.833  0.5  ]
 [ 0.     0.     0.592]
 [ 0.5    0.     0.592]
 [ 0.     0.5    0.592]
 [ 0.5    0.5    0.592]]
Scaled coordinates from ase are:
[[ 0.167  0.167  0.408]
 [ 0.667  0.167  0.408]
 [ 0.167  0.667  0.408]
 [ 0.667  0.667  0.408]
```

```
 [ 0.833  0.333  0.5  ]
 [ 0.333  0.333  0.5  ]
 [ 0.833  0.833  0.5  ]
 [ 0.333  0.833  0.5  ]
 [ 0.     0.     0.592]
 [ 0.5    0.     0.592]
 [ 0.     0.5    0.592]
 [ 0.5    0.5    0.592]]
```

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
import numpy as np
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 us 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
```

```
atom positions in primitive basis
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 0.  1.  1.]
 [ 1.  2.  2.]
 [ 1.  0.  1.]
 [ 2.  1.  2.]
 [ 1.  1.  2.]
 [ 2.  2.  3.]
 [ 1.  1.  0.]
 [ 2.  2.  1.]
 [ 1.  2.  1.]
 [ 2.  3.  2.]
```

```
 [ 2.  1.  1.]
 [ 3.  2.  2.]
 [ 2.  2.  2.]
 [ 3.  3.  3.]]
unit cell in terms of the primitive basis
[[ 0.  2.  2.]
 [ 2.  0.  2.]
 [ 2.  2.  0.]]
```

10.2.2 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
```

```
|A| =  1.73205080757
|a| =  1.73205080757
|a| =  1.73205080757
ScientificPython angle =  0.955316618125
numpy angle =             0.955316618125
Scientific A .cross. B =  [-1.0, 0.0, 1.0]
numpy A .cross. B      =  [-1.  0.  1.]
```

10.2.3 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. `ase` implements a convenient function to get the volume of an `Atoms` object: `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 'V = {0} ang^3 from dot/cross'.format(np.dot(np.cross(a1,a2),a3))
print 'V = {0} ang^3 from det'.format(np.linalg.det(uc))

from ase import *

atoms = Atoms([],cell=uc) #empty list of atoms
print 'V = {0} ang^3 from get_volume'.format(atoms.get_volume())
```

```
V = 20 ang^3 from dot/cross
V = 20.0 ang^3 from det
V = 20.0 ang^3 from get_volume
```

10.2.4 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 (method 1) = %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 'd_111 spacing (method 2) =',np.sqrt(1/id2)

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

'''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
print 'a   b   c   alpha beta gamma'
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_111 spacing (method 3) =',d
```

```
d_111 spacing (method 1) = 2.309 Angstroms
d_111 spacing (method 2) = 2.30940107676

a   b   c   alpha beta gamma
4.000 4.000 4.000 1.571 1.571 1.571

d_111 spacing (method 3) = 2.30940107676
```

10.3 Equations of State

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

10.3.1 Birch-Murnaghan

This is probably the most common equation of state used most often, and is a modification of the original Murnaghan EOS described below. A current description of the equation is in reference [14]. You can also find the equations for the Vinet and Pourier-Tarantola equations of state in that reference.

Birch-Murnaghan EOS:

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

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

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

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

and the Poirier-Tarantola EOS:

$E(\varrho) = E_0 + \frac{B_0V_0\varrho^2}{6}(3 + \varrho(B_0' - 2))$

with $\varrho = -3\ln(\eta)$.

10.3.2 Murnaghan

The equation most often used in the Murnaghan [27] equation of state is described in [11].

$$E = E_T + \frac{B_0V}{B_0'}\left[\frac{(V_0/V)^{B_0'}}{B_0'-1} + 1\right] - \frac{V_0B_0}{B_0'-1}$$

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

10.3.3 Birch

The original Birch equation [1] is:

$$E = E_0 + \frac{9}{8}B_0V_0\left(\left(\frac{V_0}{V}\right)^{\frac{2}{3}} - 1\right)^2 + \frac{9}{16}B_0V_0(B_0' - 4)\left(\left(\frac{V}{V_0}\right)^{2/3} - 1\right)^3$$

10.3.4 The Anton-Schmidt Equation of state [24]

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

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

10.3.5 Fitting data to these equations of state

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
39
40  #now we have to create the equation of state function
41  def Murnaghan(parameters,vol):
42      '''
43      given a vector of parameters and volumes, return a vector of energies.
44      equation From PRB 28,5480 (1983)
45      '''
46      E0 = parameters[0]
47      B0 = parameters[1]
48      BP = parameters[2]
49      V0 = parameters[3]
50
51      E = E0 + B0*vol/BP*(((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
52
53      return E
54
55  # and we define an objective function that will be minimized
56  def objective(pars,y,x):
57      #we will minimize this function
58      err =  y - Murnaghan(pars,x)
59      return err
60
61  x0 = [e0, b0, bP, v0] #initial guesses in the same order used in the Murnaghan function
62
63  murnpars, ier = leastsq(objective, x0, args=(e,v)) #this is from scipy
64
65  #now we make a figure summarizing the results
66  plot(v,e,'ro')
67  plot(vfit, a*vfit**2 + b*vfit + c,'--',label='parabolic fit')
68  plot(vfit, Murnaghan(murnpars,vfit), label='Murnaghan fit')
69  xlabel('Volume ($\AA^3$)')
70  ylabel('Energy (eV)')
71  legend(loc='best')
72
73  #add some text to the figure in figure coordinates
74  ax = gca()
75  text(0.4,0.5,'Min volume = %1.2f $\AA^3$' % murnpars[3],
76       transform = ax.transAxes)
77  text(0.4,0.4,'Bulk modulus = %1.2f eV/$\AA^3$ = %1.2f GPa' % (murnpars[1],
78                                                                murnpars[1]*160.21773)
79       , transform = ax.transAxes)
80  savefig('images/a-eos.png')
81  show()
82
83  np.set_printoptions(precision=3)
84  print 'initial guesses  : ', np.array(x0) #array for easy printing
85  print 'fitted parameters: ', murnpars
```

```
initial guesses  :  [-56.472   0.631   4.     16.79 ]
fitted parameters:  [-56.466   0.49    4.753  16.573]
```

You can see the Murnaghan equation of state fits the data better than the parabola.

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

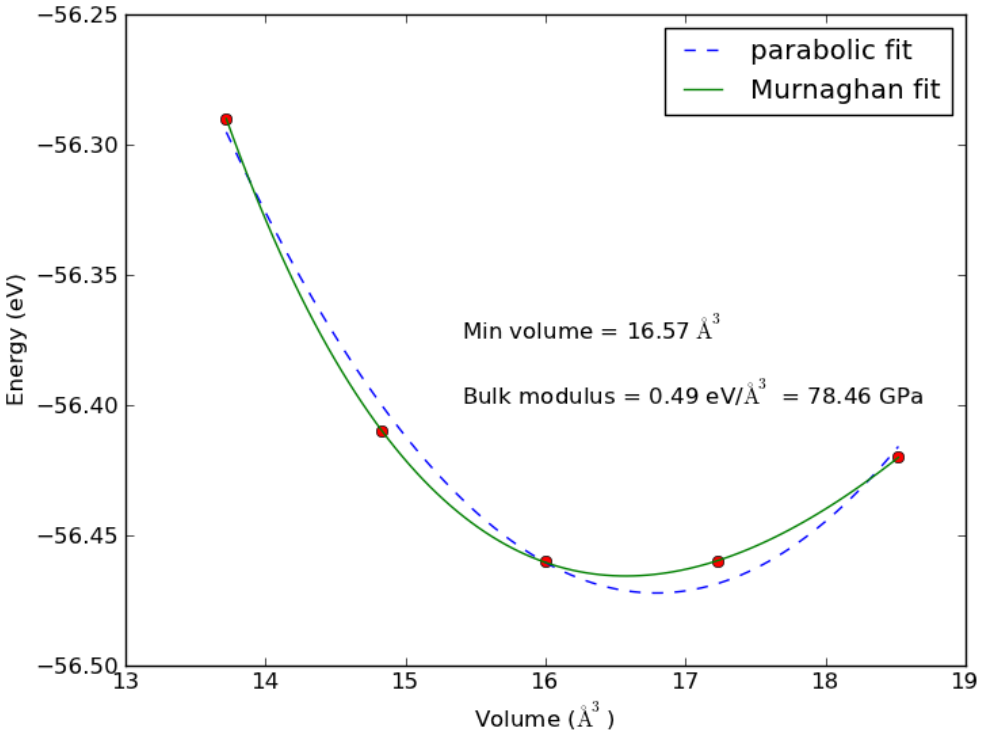


Figure 79: Fitted equation of state for bulk data. The initial fitted parabola is shown to illustrate how it is useful for making initial guesses of the minimum and bulk modulus.

10.4 Miscellaneous =jasp=/VASP tips

10.4.1 Using a special setup

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

No extension means the standard potential. The following extension mean:

Here are some links to information in the VASP manual for the setups.

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

Here we show how to select the O_sv potential in a calculation.

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

atoms = Atoms([Atom('O',[5,5,5],magmom=1)],
              cell=(6,6,6))

with jasp('molecules/O_sv',
          encut=300,
          xc='PBE',
          ispin=2,
          ismear=0,
          sigma=0.001,
          setups={'O':'_sv'}, # specifies O_sv potential
```

Table 5: Meaning of extensions on POTCAR files for special setups.

extension	¡40¿
_h	means the potential is harder than the standard (i.e. needs a higher cutoff energy)
_s	means the potential is softer than the standard (i.e. needs a lower cutoff energy)
_sv	s and p semi-core states are treated as valence states
_pv	p semi-core states are treated as valence states
_d	d semi-core states are treated as valence states

```
14          atoms=atoms) as calc:
15
16      print 'Total energy = {0} eV'.format(atoms.get_potential_energy())
```

```
Total energy = -1578.61345 eV
```

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

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

```
PAW_PBE O_sv 05Jul2007
```

10.4.2 Running jasp in parallel

jasp is smart. If you ask for more than one node, it will automatically try to run in parallel.

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

```
None
```

How do you know it ran on four nodes?

```
1  head molecules/O_sv-4nodes/OUTCAR
```

```
 vasp.5.2.12 11Nov11 complex

 executed on             LinuxIFC date 2012.08.31  09:10:10
 running on    4 nodes
 distr:  one band on    1 nodes,    4 groups


--------------------------------------------------------------------------------
```

10.4.3 Exporting data json, xml, python, sqlite

jasp has some capability for representing a calculation result in an archival format. The formats currently under development are json, xml, python and sqlite. The main point of these methods is to make it easy to create archive files that are machine readable for supplementary information in publications. These are under development.

python This is code that should reconstruct the python code needed to run a particular calculation. There are some limitations, e.g. it does not currently get magnetic moments on the atoms.

```
1 from jasp import *
2
3 with jasp('bulk/alloy/cu') as calc:
4     print calc.python
```

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

atoms = Atoms([Atom('Cu',[0.0, 0.0, 0.0]),
               cell = [[1.818, 0.0, 1.818],
                       [1.818, 1.818, 0.0],
                       [0.0, 1.818, 1.818]])

with jasp('bulk/alloy/cu',
          nbands = 9,
          nsw = 10,
          ibrion = 2,
          isif = 4,
          encut = 350.0,
          prec = 'Normal',
          kpts = array([13, 13, 13]),
          reciprocal = False,
          xc = 'PBE',
          txt = '-',
          gamma = False,
          atoms=atoms) as calc:
    # your code here
```

json

```
from jasp import *

with jasp('bulk/alloy/cu') as calc:
    print calc.pretty_json
```


```
{
    "INCAR": {
        "addgrid": null,
        "aexx": null,
        "aggac": null,
        "aggax": null,
        "aldac": null,
        "algo": null,
        "amin": null,
        "amix": null,
        "amix_mag": null,
        "bmix": null,
        "bmix_mag": null,
        "ddr": null,
        "deper": null,
        "dfnmax": null,
        "dfnmin": null,
        "dipol": null,
        "drotmax": null,
        "ebreak": null,
        "ediff": null,
        "ediffg": null,
        "eint": null,
        "emax": null,
        "emin": null,
        "enaug": null,
        "encut": 350.0,
        "encutfock": null,
        "encutgw": null,
        "falpha": null,
        "falphadec": null,
        "fdstep": null,
        "ferdo": null,
        "ferwe": null,
        "fnmin": null,
        "ftimedec": null,
        "ftimeinc": null,
        "ftimemax": null,
        "gga": null,
        "hfscreen": null,
```

```
"ialgo": null,
"iband": null,
"ibrion": 2,
"ichain": null,
"icharg": null,
"idipol": null,
"images": null,
"iniwav": null,
"invcurve": null,
"iopt": null,
"isif": 4,
"ismear": null,
"ispin": null,
"istart": null,
"isym": null,
"iwavpr": null,
"jacobian": null,
"kgamma": null,
"kpuse": null,
"kspacing": null,
"laechg": null,
"lasph": null,
"lasync": null,
"lbfgsmem": null,
"lcharg": null,
"lclimb": null,
"lcorr": null,
"ldau": null,
"ldau_luj": null,
"ldauj": null,
"ldaul": null,
"ldauprint": null,
"ldautype": null,
"ldauu": null,
"ldiag": null,
"ldipol": null,
"ldneb": null,
"lelf": null,
"lepsilon": null,
"lglobal": null,
"lhfcalc": null,
"llineopt": null,
"lmaxmix": null,
"lnebcell": null,
"loptics": null,
"lorbit": null,
```

```
"lpard": null,
"lplane": null,
"lscalapack": null,
"lscalu": null,
"lsepb": null,
"lsepk": null,
"ltangentold": null,
"lthomas": null,
"luse_vdw": null,
"lvdw": null,
"lvhar": null,
"lvtot": null,
"lwave": null,
"magmom": null,
"maxmix": null,
"maxmove": null,
"nbands": 9,
"nblk": null,
"nbmod": null,
"nelect": null,
"nelm": null,
"nelmdl": null,
"nelmin": null,
"nfree": null,
"ngx": null,
"ngxf": null,
"ngy": null,
"ngyf": null,
"ngz": null,
"ngzf": null,
"nkred": null,
"nkredx": null,
"nkredy": null,
"nkredz": null,
"nomega": null,
"nomegar": null,
"npar": null,
"nsim": null,
"nsw": 10,
"nupdown": null,
"nwrite": null,
"param1": null,
"param2": null,
"pomass": null,
"potim": null,
"prec": "Normal",
```

```
    "precfock": null,
    "ropt": null,
    "rwigs": null,
    "sdalpha": null,
    "sdr": null,
    "sigma": null,
    "smass": null,
    "snl": null,
    "spring": null,
    "stol": null,
    "symprec": null,
    "system": null,
    "tebeg": null,
    "teend": null,
    "time": null,
    "timestep": null,
    "vdwgr": null,
    "vdwrn": null,
    "voskown": null,
    "weimin": null,
    "zab_vdw": null,
    "zval": null
},
"atoms": {
    "cell": [
        [
            1.818,
            0.0,
            1.818
        ],
        [
            1.818,
            1.818,
            0.0
        ],
        [
            0.0,
            1.818,
            1.818
        ]
    ],
    "pbc": [
        true,
        true,
        true
    ],
```

```
        "positions": [
            [
                0.0,
                0.0,
                0.0
            ]
        ],
        "symbols": [
            "Cu"
        ],
        "tags": [
            0
        ]
    },
    "input": {
        "gamma": false,
        "kpts": [
            13,
            13,
            13
        ],
        "kpts_nintersections": null,
        "reciprocal": false,
        "setups": null,
        "txt": "-",
        "xc": "PBE"
    }
}
```

xml This relies on the pyxser module.

```
from jasp import *

with jasp('bulk/alloy/cu') as calc:
    print calc.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<pyxs:obj xmlns:pyxs="http://projects.coder.cl/pyxser/model/" version="1.0" type="vasp" modu
  <pyxs:col type="dict" name="d">
    <pyxs:col type="dict" name="INCAR">
      <pyxs:prop type="str" name="prec">Normal</pyxs:prop>
      <pyxs:prop type="float" name="encut">350.0</pyxs:prop>
      <pyxs:prop type="int" name="nbands">9</pyxs:prop>
      <pyxs:prop type="int" name="isif">4</pyxs:prop>
      <pyxs:prop type="int" name="nsw">10</pyxs:prop>
      <pyxs:prop type="int" name="ibrion">2</pyxs:prop>
    </pyxs:col>
```

```
  <pyxs:col type="dict" name="input">
    <pyxs:col type="list" name="kpts">
      <pyxs:prop type="int" name="kpts">13</pyxs:prop>
      <pyxs:prop type="int" name="kpts">13</pyxs:prop>
      <pyxs:prop type="int" name="kpts">13</pyxs:prop>
    </pyxs:col>
    <pyxs:prop type="bool" name="reciprocal">False</pyxs:prop>
    <pyxs:prop type="str" name="xc">PBE</pyxs:prop>
    <pyxs:prop type="str" name="txt">-</pyxs:prop>
    <pyxs:prop type="bool" name="gamma">False</pyxs:prop>
  </pyxs:col>
  <pyxs:col type="dict" name="atoms">
    <pyxs:col type="list" name="cell">
      <pyxs:col type="list" name="cell">
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
        <pyxs:prop type="float" name="cell">0.0</pyxs:prop>
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
      </pyxs:col>
      <pyxs:col type="list" name="cell">
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
        <pyxs:prop type="float" name="cell">0.0</pyxs:prop>
      </pyxs:col>
      <pyxs:col type="list" name="cell">
        <pyxs:prop type="float" name="cell">0.0</pyxs:prop>
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
        <pyxs:prop type="float" name="cell">1.818</pyxs:prop>
      </pyxs:col>
    </pyxs:col>
    <pyxs:col type="list" name="symbols">
      <pyxs:prop type="str" name="symbols">Cu</pyxs:prop>
    </pyxs:col>
    <pyxs:col type="list" name="pbc">
      <pyxs:prop type="bool" name="pbc">True</pyxs:prop>
      <pyxs:prop type="bool" name="pbc">True</pyxs:prop>
      <pyxs:prop type="bool" name="pbc">True</pyxs:prop>
    </pyxs:col>
    <pyxs:col type="list" name="positions">
      <pyxs:col type="list" name="positions">
        <pyxs:prop type="float" name="positions">0.0</pyxs:prop>
        <pyxs:prop type="float" name="positions">0.0</pyxs:prop>
        <pyxs:prop type="float" name="positions">0.0</pyxs:prop>
      </pyxs:col>
    </pyxs:col>
  </pyxs:col>
</pyxs:col>
```

</pyxs:obj>

11 Python

11.1 easy_install as a user

This usually works if the directory is on your PYTHONPATH
easy_install -d ~/lib/python2.6/site-packages/ pymatgen

11.2 Integer division math gotchas

It pays to be careful when dividing by integers because you can get unexpected results if you do not know the integer division rules. In python 2.6, if you divide two integers, you get an integer! This is usually not a problem if there is no remainder in the division, e.g. 6/3=2. But, if there is a remainder, and that remainder is important, you will lose it. Here is an example of calculating the mole fraction of a species from integer numbers of atoms in the unit cell. If you are not careful, you get the wrong answer! You can convert (also called casting) a number to a float using the float command.

```
nPd = 4
nCu = 5
x_Cu = nCu/(nPd + nCu)
print 'x_cu = {0} (integer division)'.format(x_Cu)

# now cast as floats
x_Cu = float(nCu)/float(nPd + nCu)
print 'x_cu = {0} (float division)'.format(x_Cu)
```

```
x_cu = 0 (integer division)
x_cu = 0.555555555556 (float division)
```

Note that if one of the numbers is a float, python will automatically cast the integer as a float, and return a float.

```
nPd = 4
nCu = 5

# now cast as floats
x_Cu = float(nCu)/(nPd + nCu)
print 'x_cu = {0}'.format(x_Cu)
```

```
x_cu = 0.555555555556
```

Finally, you can tell python a number is a float by adding a decimal to it. You do not need to put a 0 after the decimal, but you can.

```
nPd = 4.  # this is a float
nCu = 5

x_Cu = nCu/(nPd + nCu)
print 'x_cu = {0}'.format(x_Cu)
```

```
x_cu = 0.555555555556
```

12 References

References

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

[2] Stefano Baroni, Stefano de Gironcoli, Andrea Dal Corso, and Paolo Giannozzi. Phonons and related crystal properties from density-functional perturbation theory. *Rev. Mod. Phys.*, 73:515–562, Jul 2001. URL: http://link.aps.org/doi/10.1103/RevModPhys.73.515, doi:10.1103/RevModPhys.73.515.

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

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

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

[6] Kyle J. Caspersen and Emily A. Carter. Finding transition states for crystalline solidsolid phase transformations. *Proceedings of the National Academy of Sciences of the United States of America*, 102(19):6738–6743, 2005. URL: http://www.pnas.org/content/102/19/6738.abstract, arXiv:http://www.pnas.org/content/102/19/6738.full.pdf+html, doi:10.1073/pnas.0408127102.

[7] Wei Chen, David Schmidt, William F. Schneider, and C. Wolverton. First-principles cluster expansion study of missing-row reconstructions of fcc (110) surfaces. *Phys. Rev. B*, 83:075415, Feb 2011. URL: http://link.aps.org/doi/10.1103/PhysRevB.83.075415, doi:10.1103/PhysRevB.83.075415.

[8] Larry A. Curtiss, Krishnan Raghavachari, Paul C. Redfern, and John A. Pople. Assessment of gaussian-2 and density functional theories for the computation of enthalpies of formation. *The Journal of Chemical Physics*, 106(3):1063–1079, 1997. URL: http://link.aip.org/link/?JCP/106/1063/1, doi:10.1063/1.473182.

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

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

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

[12] Paolo Giannozzi and Stefano Baroni. Vibrational and dielectric properties of c_60 from density-functional perturbation theory. *The Journal of Chemical Physics*, 100(11):8537–8539, 1994. URL: http://link.aip.org/link/?JCP/100/8537/1, doi:10.1063/1.466753.

[13] Jeong Woo Han, Liwei Li, and David S. Sholl. Density functional theory study of H and CO adsorption on alkali-promoted Mo_2C surfaces. *The Journal of Physical Chemistry C*, 115(14):6870–6876, 2011. URL: http://pubs.acs.org/doi/abs/10.1021/jp200950a, arXiv:http://pubs.acs.org/doi/pdf/10.1021/jp200950a, doi:10.1021/jp200950a.

[14] M. Hebbache and M. Zemzemi. *Ab initio* study of high-pressure behavior of a low compressibility metal and a hard material: osmium and diamond. *Phys. Rev. B*, 70:224107, Dec 2004. URL: http://link.aps.org/doi/10.1103/PhysRevB.70.224107, doi:10.1103/PhysRevB.70.224107.

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

[16] Nilay İnoğlu and John R. Kitchin. Simple model explaining and predicting coverage-dependent atomic adsorption energies on transition metal surfaces. *Phys. Rev. B*, 82:045414, Jul 2010. URL: http://link.aps.org/doi/10.1103/PhysRevB.82.045414, doi:10.1103/PhysRevB.82.045414.

[17] Anubhav Jain, Geoffroy Hautier, Shyue Ping Ong, Charles J. Moore, Christopher C. Fischer, Kristin A. Persson, and Gerbrand Ceder. Formation enthalpies by mixing GGA and GGA + U calculations. *Phys. Rev. B*, 84:045115, Jul 2011. URL: http://link.aps.org/doi/10.1103/PhysRevB.84.045115, doi:10.1103/PhysRevB.84.045115.

[18] David Karhánek, Tomáš Bučko, and Jürgen Hafner. A density-functional study of the adsorption of methane-thiol on the (111) surfaces of the Ni-group metals: II. vibrational spectroscopy. *Journal of Physics: Condensed Matter*, 22(26):265006, 2010. URL: http://stacks.iop.org/0953-8984/22/i=26/a=265006.

[19] John R. Kitchin. Correlations in coverage-dependent atomic adsorption energies on pd(111). *Phys. Rev. B*, 79:205412, May 2009. URL: http://link.aps.org/doi/10.1103/PhysRevB.79.205412, doi:10.1103/PhysRevB.79.205412.

[20] John R. Kitchin, Karsten Reuter, and Matthias Scheffler. Alloy surface segregation in reactive environments: First-principles atomistic thermodynamics study of $ag_3pd(111)$ in oxygen atmospheres. *Phys. Rev. B*, 77:075437, Feb 2008. URL: http://link.aps.org/doi/10.1103/PhysRevB.77.075437, doi:10.1103/PhysRevB.77.075437.

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

[22] Kyuho Lee, Éamonn D. Murray, Lingzhu Kong, Bengt I. Lundqvist, and David C. Langreth. Higher-accuracy van der waals density functional. *Phys. Rev. B*, 82:081101, Aug 2010. URL: http://link.aps.org/doi/10.1103/PhysRevB.82.081101, doi:10.1103/PhysRevB.82.081101.

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

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

[25] Spencer D. Miller, Nilay Inoglu, and John R. Kitchin. Configurational correlations in the coverage dependent adsorption energies of oxygen atoms on late transition metal fcc(111) surfaces. *The Journal of Chemical Physics*, 134(10):104709, 2011. URL: http://link.aip.org/link/?JCP/134/104709/1, doi:10.1063/1.3561287.

[26] Spencer D. Miller and John R. Kitchin. Relating the coverage dependence of oxygen adsorption on Au and Pt fcc(111) surfaces through adsorbate-induced surface electronic structure effects. *Surface Science*, 603(5):794 – 801, 2009. URL: http://www.sciencedirect.com/science/article/pii/S0039602809001186, doi:10.1016/j.susc.2009.01.021.

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

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

[29] Anand Udaykumar Nilekar, Jeff Greeley, and Manos Mavrikakis. A simple rule of thumb for diffusion on transition-metal surfaces. *Angewandte Chemie International Edition*, 45(42):7046–7049, 2006. URL: http://dx.doi.org/10.1002/anie.200602223, doi:10.1002/anie.200602223.

[30] Joachim Paier, Robin Hirschl, Martijn Marsman, and Georg Kresse. The Perdew–Burke–Ernzerhof exchange-correlation functional applied to the G2-1 test set using a plane-wave basis set. *The Journal of Chemical Physics*, 122(23):234102, 2005. URL: http://link.aip.org/link/?JCP/122/234102/1, doi:10.1063/1.1926272.

[31] Daniel Sheppard, Penghao Xiao, William Chemelewski, Duane D. Johnson, and Graeme Henkelman. A generalized solid-state nudged elastic band method. *The Journal of Chemical Physics*, 136(7):074103, 2012. URL: http://link.aip.org/link/?JCP/136/074103/1, doi:10.1063/1.3684549.

[32] B. Silvi and A Savin. Classification of chemical bonds based on topological analysis of electron localization functions. *Nature*, 371:683–686, 1994. URL: http://dx.doi.org/10.1038/371683a0.

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

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

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

[36] L. Vitos, A.V. Ruban, H.L. Skriver, and J. Kollár. The surface energy of metals. *Surface Science*, 411(12):186 – 202, 1998. URL: http://www.sciencedirect.com/science/article/pii/S003960289800363X, doi:10.1016/S0039-6028(98)00363-X.

[37] Lei Wang, Thomas Maxisch, and Gerbrand Ceder. Oxidation energies of transition metal oxides within the GGA + U framework. *Phys. Rev. B*, 73:195107, May 2006. URL: http://link.aps.org/doi/10.1103/PhysRevB.73.195107, doi:10.1103/PhysRevB.73.195107.

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

13 Index

Index