

Modeling materials using density functional theory

John Kitchin

2012-07-11 Wed

Contents

1	List	of figu	ires	6
2	List	of tab	les	9
3	Intr	oducti	on to this book	9
4	Mo	lecules		11
	4.1	Definir	ng and visualizing molecules	11
		4.1.1	Predefined molecules	11
		4.1.2	Reading other data formats into a calculation	16
		4.1.3	From scratch	19
		4.1.4	Combining Atoms objects	23
	4.2	Simple	properties	23
		4.2.1	Getting cartesian positions	24
		4.2.2	Molecular weight and molecular formula	25
		4.2.3	Center of mass	26
		4.2.4	Moments of inertia	27
	4.3	Simple	properties that require computations	28
		4.3.1	Computing bond lengths and angles	28
			Dihedral angles	30
		4.3.2	Energy and forces	31
		4.3.3	The density of states	33
		4.3.4	Atom-projected density of states on molecules	34
		4.3.5	Visualizing electron density	36
		4.3.6	Bader analysis	37
		4.3.7	Dipole moments	41
	4.4	Geome	etry optimization	43
			Rond lengths	43

		Manual determination	43
		Automatic geometry optimization with vasp	46
		Relaxation of a water molecule	47
4.5	Vibra	tional frequencies	47
	4.5.1	Manual calculation of vibrational frequency	47
	4.5.2	Automated vibrational calculations	50
		Zero-point energy for multiple modes	51
4.6	Simul	ated infrared spectra	52
	4.6.1	TODO Shell scripts are rough. One day I should	
		rewrite this in python	56
4.7	Thern	nochemical properties of molecules	56
4.8	Molec	rular reaction energies	58
	4.8.1	O_2 dissociation	58
		Simple estimate of O_2 dissociation energy	58
		Estimating O_2 dissociation energy with spin polariza-	
		tion in triplet ground states	59
		Looking at the two spin densities	60
		Convergence study of the O_2 dissociation energy	61
		Illustration of the effect of sigma	64
		Estimating triplet oxygen dissociation energy with low	
		symmetry	65
		Estimating singlet oxygen dissociation energy	69
		Verifying the magnetic moments on each atom	71
		Using a different potential	72
	4.8.2	Water gas shift example	73
		calculation summaries	75
		CO	7 5
		CO_2	7 6
		H2	77
		H\$_2\$O	7 8
	4.8.3	Temperature dependent water gas shift equilibrium	
		constant	7 9
		CO vibrations	7 9
		CO_2 vibrations	80
		H_2 vibrations	80
		H\$_2\$O vibrations	81
		TODO thermochemistry	82
		TODO something is wrong with the thermo-	
		chemistry I think	85
4.9	Molec	rular reaction barriers	85

		4.9.1	Get initial and final states	85
		4.9.2	Run band calculation	86
		4.9.3	Make a movie of the animation	87
	4.10	Simula	ated infrared spectra	
			TODO Shell scripts are rough. One day I should	
			rewrite this in python	91
5	Bull	k syste	oms	91
0	5.1	v	ng and visualizing bulk systems	
	0.1	5.1.1	Built-in functions in ase	
		5.1.2	Using http://materialsproject.org	
	5.2		nining bulk structures	
	0.2	5.2.1	fcc/bcc crystal structures	
		5.2.1 $5.2.2$	hcp	
		5.2.3	Complex structures with internal degrees of freedom.	
		5.2.4	TODO the volume is very small (60) compared to	100
		0.2.4	what we expected (64) something may be wrong!	111
	5.3	Bulls o	calculation parameters	
	5.5	5.3.1	Effect of SIGMA	
	5.4	0.0.1	ive energy	
	$5.4 \\ 5.5$		of pressure on phase stability	
	5.6		eaction energies	
	5.0	5.6.1	Alloy formation energies	
		5.0.1	Basic alloy formation energy	
		5.6.2	Metal oxide oxidation energies	
		5.0.2	Cu\$_2\$O calculation	
			CuO calculation	
	5.7	Bulls d	Reaction energy calculation	
	5.1	5.7.1	TODO discuss why these should not be related to	199
		0.1.1	spectrosopy, but often are	122
	5.8	Atom	projected density of states	
	5.9		structures	
	5.10	_	Determinism if a magnetic galution is an appetically fa	140
		5.10.1	Determining if a magnetic solution is energetically fa-	1.40
		5 10 9	vorable	
		5.10.2	0 1	
	E 11		TODO NiO-FeO formation energies with magnetism	142
			O phonons	
	0.12	TODO	O solid state NEB	142

3	Sur	aces 143
	6.1	Surface structures
		6.1.1 Simple surfaces
		6.1.2 vicinal surfaces
	6.2	Surface relaxation
		6.2.1 TODO Surface reconstruction
	6.3	Work function
	6.4	Surface energy
		6.4.1 Advanced topics in surface energy 154
	6.5	Dipole correction
		6.5.1 TODO slab with no dipole correction 155
		6.5.2 slab with a dipole correction
		6.5.3 Comparing no dipole correction with a dipole correction 157
	6.6	Adsorption energies
		6.6.1 simple estimate
		Calculations
		clean slab calculation
		fcc site
		bridge site
		hcp site
		Analysis of adsorption energies
		Adsorption on bridge site with constraints 165
		6.6.2 Coverage dependence $\dots \dots 166$
		clean slab calculation
		fcc site at 1 ML coverage
		Adsorption energy at 1ML
		6.6.3 Effect of adsorption on the surface energy 169
	6.7	Adsorbate vibrations
		6.7.1 Vibrations of the bridge site
	6.8	Surface Diffusion barrier
		6.8.1 Standard nudged elastic band method 173
		6.8.2 Climbing image NEB
		6.8.3 TODO use vibrations to confirm transition state 176
	6.9	TODO Diffusion rates with transition state theory 176
7	Ato	mistic thermodynamics 176
	7.1	bulk phase stability of oxides
		7.1.1 TODO save graph
	7.2	Effect on adsorption
	7.3	Atomistic therodynamics and multiple reactions 183

8	Adv	anced	electronic structure methods			185
	8.1	DFT+	U			185
		8.1.1	Metal oxide oxidation energies with DFT+U $$			185
			Cu2O calculation with U=4.0			185
			CuO calculation with U=4.0			187
			Reaction energy calculation with DFT+U			188
			How much does U affect the reaction energy?			
	8.2	Hybrid	functionals			190
		8.2.1	FCC Ni DOS			190
	8.3	TODO) vdW			192
	8.4	TODO	D DFT+D			192
	8.5					
	8.6		O Charge partitioning schemes			
	8.7		Modeling Core level shifts			
0	A -1-		I			104
9	ACK	nowied	gments			194
10	\mathbf{App}	endice	s			194
	10.1	Recipe	s			194
		10.1.1	Modifying Atoms by deleting atoms			194
		10.1.2	Advanced tagging			195
		10.1.3	Using units in ase			198
		10.1.4	Extracting parts of an array			198
		10.1.5	Statistics			199
			Confidence intervals			199
		10.1.6	Curve fitting			200
			Linear fitting			200
		10.1.7	Nonlinear curve fitting			202
		10.1.8	Nonlinear curve fitting by direct least squares mi	m	-	
			imization			203
		10.1.9	Nonlinear curve fitting with confidence intervals			204
		10.1.10	Interpolation with splines			207
		10.1.11	Interpolation in 3D			207
		10.1.12	Reading and writing data			214
			Builtin io modules			
			From scratch			
		10.1.13	Integration			
			Numerical differentiation			
			Simple loops to define finite difference derivatives			
			FFT derivatives			220

	10.1.15	$5\mathrm{NetCDF}$ files $$. 222
	10.1.16	$6\mathrm{python}$ modules	. 223
	10.1.17	TODO reading and writing excel files	223
	10.1.18	STODO making movies	223
10.2	Compi	utational geometry	. 224
	10.2.1	Changing coordinate systems	. 224
	10.2.2	Simple distances, angles	. 227
	10.2.3	Unit cell properties	. 228
	10.2.4	d-spacing	. 228
	10.2.5	Center of mass	. 230
		Moments of inertia	
10.3	Equati	ions of State	231
	10.3.1	Birch-Murnaghan	232
	10.3.2	Murnaghan	232
		Birch	
		The Anton-Schmidt Equation of state [23]	
		Fitting data to these equations of state	
10.4		laneous jasp/Vasp tips	
		Using a special setup	
		Running jasp in parallel	
	10.4.3	Exporting data json, xml, python, sqlite	
		python	
		json	
		xml	238
11 Pyt	hon		240
		nstall as a user	
	-	r division math gotchas	
11.2	1111080	a division matti gottimo	10
12 Ref	erences	s	241
13 Ind	ex		246
1 L	ist of	figures	
		-	
List o	of Fig	gures	
-	A 00TT4	b obcort 1 1 · 1	
1	A CHS	\$_3\$CN molecule in a box	. 14

2	The rotated version of CH\$_3\$CN	15
3	Rotated CH\$_3\$CN molecule	17
4	An isobutane molecule read in from an xyz data file	20
5	Image of a CO molecule with the C at the origin	21
6	CO in an fcc cell	22
7	Image of the combined ammonia and oxygen molecule	24
8	Schematic of the vectors defining the H-N-H angle	30
9	Schematic of the calculated ethane dihedral angle	32
10	Charge density of a CO molecule	37
11	Charge density of a CO molecule centered in the unit cell	38
12	Energy vs CO bond length. The minimum appears to be near	
	1.15 Å by inspection.	45
13	Spin-polarized density of states for the O_2 molecule	61
14	Convergence study of the O_2 dissociation energy as a function	
	of ENCUT	63
15	Effect of SIGMA on the oxygen dissociation energy	65
16	Thermodynamic energies of the water gas shift reaction as a	
	function of temperature	84
17	NEB for ammonia flipping	87
18	A simple fcc Ag bulk structure	92
19	A simple Ag\$_3\$Pd bulk structure	93
20	An RuO_2 unit cell prepared from a cif file	96
21	Total energy vs. fcc lattice contant for Cu. It appears the	
	minimum is near 3.65 Å	99
22	Total energy vs. volume for fcc Cu with fitted cubic polyno-	
	mial equation of state	100
23	Comparison of energies between fcc and bcc Cu. The fcc	
	structure is lower in energy	
24	Total energy vs. c/a for different values of a	104
25	Contour plot of the total energy of hcp Ru for different values	
	of a and c/a	105
26	Total energy vs volume for TIO_2 with frozen atoms and frozen	
	shape	107
27	Total energy vs. volume of TiO ₂ where shape and atoms are	
		109
28		112
29	Equations of state $(E(V))$ for an atase and rutile TiO_2	119
30	Illustration of the common tangent that shows the pressure	
	where anatase and rutile coexist before anatase converts to	
	rutile	121

31	Conceptual picture of two alloys with exothermic formation
	energies
32	Illustration of phase separation
33	Illustration of phase separation
34	Total DOS for bulk Pd
35	Total DOS for Pd computed with ISMEAR=-5 135
36	Atom projected d-band for bulk Pd. The shaded area corre-
	sponds to the occupied states below the Fermi level 137
37	Calculated band-structure for Si
38	Total energy vs. total magnetic moment for bcc Fe 141
39	An Al(111) slab with three layers and 20 Å of vacuum 144
40	An Au(211) surface constructed with mod:ase 145
41	xy averaged local electrostatic potential of an Al(111) slab 151
42	Schematic figure illustrating the calculation of a surface energy. 152
43	Surface energy of an Al(111) slab as a function of thickness 154
44	comparison of the electrostatic potentials with a dipole cor-
	rection and without it
45	Final geometry of the fcc site
46	Final geometry of the hcp site
47	Final geometry of the bridge site. You can see that the oxygen
	atom ended up in the fcc site
48	Initial geometry of the bridge site. It is definitely on the bridge. 164
49	Final geometry of the bridge site. It has fallen into the fcc site. 164
50	Initial state of the O atom on the bridge site
51	Final state of the constrained O atom, still on the bridge site. 167
52	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
53	Energy pathway for O diffusion from an fcc to hcp site with
	a spline fit to determine the barrier
54	Climbing image NEB
55	Effect of temperature on the Gibbs free energy of an O_2
	molecule at 1 atm
56	Effects of pressure on the ideal gas Gibbs free energy of O_2 178
57	$\Delta\mu_{O_2}(T,p)$ at different pressures and temperatures 180
58	Temperature dependent decomposition pressure for Ag\$_2\$O. 182
59	Effect of oxygen chemical potential on the adsorption energy. 184
60	Comparison of DOS from GGA, and two hybrid GGAs (PBE0
	ad HSE06)
61	ELF for an isosurface of 0.3 for CF_4

62	ELF for an isosurface of 0.75 for CF_4
63	Example of slicing out part of an array. The solid line repre-
	sents the whole array, and the symbols are the array between
	$2 < x < 4. \dots $
64	Fitting a nonlinear function
65	Nonlinear fit to data
66	Illustration of a spline fit to data and finding the maximum
	point
67	Trilinear interpolation scheme
68	An example of interpolated charge density of A CO molecule
	along the axis of molecule
69	Comparison of different numerical derivatives
70	Comparison of different numerical derivatives
71	Comparison of 2 point and 4 point numerical derivatives 221
72	Comparison of FFT numerical derivatives

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 mod:ase.calculators.vasp python interface to the VASP calculator. It was written by me to facilitate writing this book, and to develop the best possible way to run DFT calculations. The best way to learn to use jasp is from this book. jasp is currently available at https://bitbucket.org/jkitchin/jasp. It probably requires the latest syn version of ase to work since I have been adding new functionality to mod .calculators.vasp as mod:jasp is developed.

The goal in writing mod: 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. mod:jasp takes care of changing into VASP calculation directories, and then changing back to where you started from. mod:jasp has an interface to sqlite, and to version control (git) software. mod:jasp has some features that mod: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 mod:jasp. You should be able to copy the actual VASP input files (which were generated by jasp) from a directory and simply run vasp. mod:jasp is mostly compatible with mod:ase.calculators.vasp, so you can replace this kind of code:

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

with this code:

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

try:
    #do stuff
finally:
```

The DFT code used primarily in this book is VASP.

• VASP wiki

os.chdir(CWD)

• 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 [29].

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

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

```
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
СНЗОН	CCH	CH3CH2NH2
РНЗ	Si2H6	03
02	BC13	CH2_s1A1d
Ве	H2CC12	СЗН9С
C3H9N	СНЗСН2ОСНЗ	BF3
СНЗ	CH4	S2
С2Н6СНОН	SiH2_s1A1d	H3CNH2
CH30	Н	ВеН
P	C3H4_C3v	C2F4
ОН	methylenecyclopropane	F20
SiCl4	HCF3	HCC13
C3H7	CH3CH2O	A1F3
CH2NHCH2	SiH2_s3B1d	H2CF2
SiF4	H2CCO	PH2
OCS	HF	NO2
SH2	C3H4_C2v	H202
CH3CH2C1	isobutane	CH3COF
HCOOH	CH3ONO	C5H8
2-butyne	SH	NF3
HOC1	CS2	P2
C	CH3S	0
C4H4S	S	C3H7Cl
H2CCHC1	C2H6	СНЗСНО
C2H4	HCN	C2H2
C2C14	bicyclobutane	H2
C6H6	N2H4	C4H4NH
H2CCHCN	H2CCHF	cyclobutane
HC1	СНЗОСНЗ	Li2
Na	CH3SiH3	NaCl

CH3CH2SH	ОСНСНО	SiH4
C2H5	SiH3	NH
C10	AlC13	CC14
NO	C2H3	ClF
HCO	CH3CONH2	CH2SCH2
CH3COCH3	C3H4_D2d	CH
CO	CN	F
CH3COC1	N	CH3Cl
Si	C3H8	CS
N2	C12	NCCN
F2	C02	Cl
CH20CH2	H20	CH3CO
SO	НСООСНЗ	butadiene
C1F3	Li	PF3
В	CH3SH	CF4
C3H6_Cs	C2H6NH	N20
LiF	H2COH	cyclobutene
LiH	SiO	Si2
C2H6SO	C5H5N	trans-butane
Na2	C4H4O	S02
NH3	NH2	CH2_s3B1d
ClNO	C3H6_D3h	Al
CH3SCH3	H2C0	CH3CN

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

Here is an example of getting the geometry of an acetonitrile molecule and writing an image to a file. Note that the default unit cell is a $1\text{\AA}\times1\text{Å}\times1\text{Å}$ 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'
```

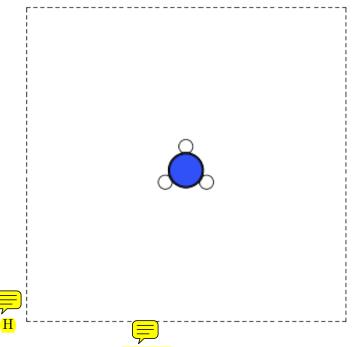


Figure 1: A CH\$_3\$CN molecule in a box.

```
9 print '----'
10 print c60.get_cell()
11
12 write('images/ch3cn.png', c60, show_unit_cell=2)
```

It is possible to rotate the atoms with functions.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
```

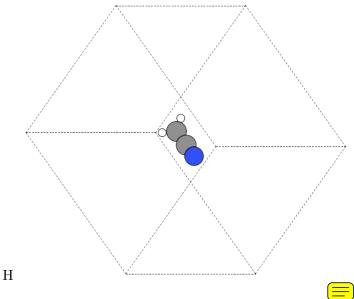


Figure 2: The rotated version of CH $3\$ CN.

```
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')
```

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

```
from ase import Atoms
2 from ase.data.molecules import molecule
3 from ase.io import write
   from numpy import pi
6 # ammonia
7 atoms = molecule('CH3CN')
8 p1 = atoms.get_positions()
10
   atoms.rotate('x', pi/4, center='COM', rotate_cell=False)
atoms.rotate('y', pi/4, center='COM', rotate_cell=False)
12
write('images/ch3cn-rotated-2.png',atoms,show_unit_cell=2)
14
   print 'difference in positions after rotating'
print 'atom difference vector'
16 print '-----,
17  p2 = atoms.get_positions()
18
19 diff = p2 - p1
20 for i,d in enumerate(diff):
       print '{0} {1}'.format(i, d)
21
```

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]

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

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

Known formats:

```
format short name short name GPAW restart-file gpw Dacapo netCDF output file dacapo
```

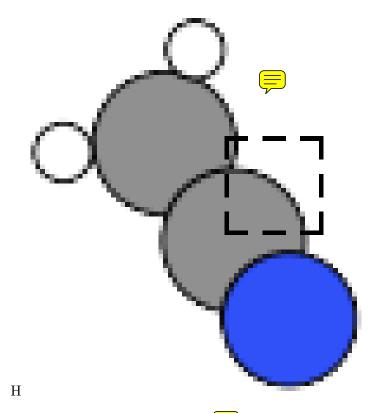


Figure 3: Rotated CH\$ N molecule

```
Old ASE netCDF trajectory
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 mod:ase.Atoms objects. Here is what an xyz file might look like:

14

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).

```
from ase.io import read,write
import numpy as np

atoms = read('molecules/isobutane.xyz')
atoms.center(vacuum=5)
write('images/isobutane_xyz.png', atoms, show_unit_cell=2)
```

None

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).

 $V = 1000 Ang^3$

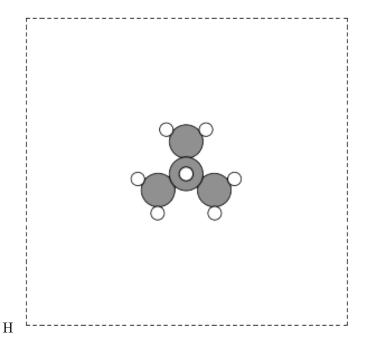


Figure 4: An isobutane molecule read in from an xyz data file.

There are two inconvenient features of the simple cubic cell:

- 1. Since the CO atom is at the corner, its electron density is spread over the 8 corners of the box, which is not convenient for visualization later.
- 2. Due to the geometry of the cube, you need fairly large cubes to decouple the molecule from its images. Here, the CO molecule has 6 images due to periodic boundary conditions that are 10 Å away. The volume of the unit cell is $1000~\text{Å}^3$.

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 Å³, 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
```

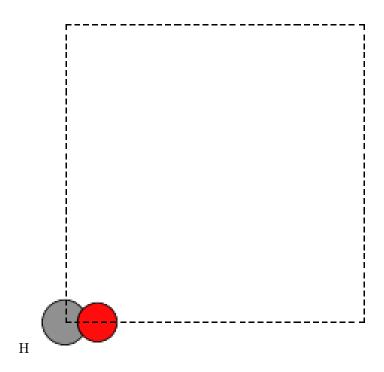


Figure 5: Image of a CO molecule with the C at the origin.

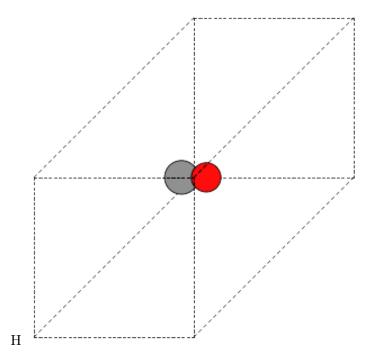


Figure 6: CO in an fcc cell.

$V = 715.8 \text{ Ang}^3$

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 mod: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
```

```
3
    b = 7.1
4
    atoms2 = Atoms([Atom('C', [0., 0.,0.]),
                     Atom('0',[1.1,0.,0.])],
6
                     cell=[[b, b, 0.],
                            [b, O., b],
8
9
                            [0., b, b]])
10
    # get unit cell vectors and their lengths
11
12
    (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
13
    print '|a3| = %1.2f Ang' % np.sum(a3**2)**0.5
       |a1| = 10.04 \text{ Ang}
       |a2| = 10.04 \text{ Ang}
       |a3| = 10.04 \text{ 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
3
    from ase.io import write
    atoms1 = molecule('NH3')
5
    atoms2 = molecule('02')
7
8
    atoms2.translate([3,0,0])
9
    bothatoms = atoms1 + atoms2
10
11
    bothatoms.center(5)
12
    write('images/bothatoms.png', bothatoms, show_unit_cell=2, rotation='90x')
```

4.2 Simple properties

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

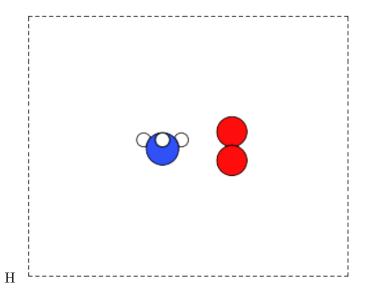


Figure 7: Image of the combined ammonia and oxygen molecule.

4.2.1 Getting cartesian positions

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

```
from ase import Atoms
   from ase.data.molecules import molecule
2
3
   from ase.io import write
   atoms = molecule('C6H6')
5
   # access properties on each atom
   8
   print '----
9
  for i,atom in enumerate(atoms):
10
11
     print '%3i%3s%8.2f%8.2f%8.2f' % (i,atom.symbol,atom.x,atom.y,atom.z)
12
13
   # get all properties in arrays
sym = atoms.get_chemical_symbols()
pos = atoms.get_positions()
  num = atoms.get_atomic_numbers()
17
   atom_indices = range(len(atoms))
18
19
20 print
21 print ' # sym at# p_x
                              р_у
22 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)
```

24

25

#	sym	p_x	р_у	p_z	
0	С	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	С	-1.21	-0.70	0.00	
5	С	-1.21	0.70	0.00	
6	H	0.00	2.48	0.00	
7	H	2.15	1.24	0.00	
8	Н	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	р_у	p_z
# 	sym 	at# 6		P_y 1.40	
			0.00		
0	C	6	0.00 1.21	1.40	0.00
0	C C	6 6	0.00 1.21	1.40 0.70	0.00
0 1 2	C C	6 6 6	0.00 1.21 1.21 0.00	1.40 0.70 -0.70	0.00 0.00 0.00 0.00
0 1 2 3	C C C	6 6 6 6	0.00 1.21 1.21 0.00	1.40 0.70 -0.70 -1.40	0.00 0.00 0.00 0.00
0 1 2 3 4	C C C C	6 6 6 6 6	0.00 1.21 1.21 0.00 -1.21 -1.21	1.40 0.70 -0.70 -1.40 -0.70	0.00 0.00 0.00 0.00 0.00 0.00
0 1 2 3 4 5	C C C C	6 6 6 6 6 6	0.00 1.21 1.21 0.00 -1.21 -1.21 0.00	1.40 0.70 -0.70 -1.40 -0.70 0.70	0.00 0.00 0.00 0.00 0.00 0.00
0 1 2 3 4 5	C C C C C C	6 6 6 6 6 6	0.00 1.21 1.21 0.00 -1.21 -1.21 0.00	1.40 0.70 -0.70 -1.40 -0.70 0.70 2.48 1.24	0.00 0.00 0.00 0.00 0.00 0.00
0 1 2 3 4 5 6 7	C C C C C C H H	6 6 6 6 6 6 1	0.00 1.21 1.21 0.00 -1.21 -1.21 0.00 2.15 2.15	1.40 0.70 -0.70 -1.40 -0.70 0.70 2.48 1.24	0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.0
0 1 2 3 4 5 6 7 8	C C C C C C H H	6 6 6 6 6 6 1 1	0.00 1.21 1.21 0.00 -1.21 -1.21 0.00 2.15 2.15 0.00	1.40 0.70 -0.70 -1.40 -0.70 0.70 2.48 1.24 -1.24 -2.48	0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.0

4.2.2 Molecular weight and molecular formula

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

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

```
atoms = molecule('C6H6')
4
5
      masses = atoms.get_masses()
6
      molecular_weight = sum(masses)
7
      print 'The molecular weight of %s is %f' % (atoms.get_chemical_symbols(reduce=True),
9
                                                   molecular_weight)
10
```

The molecular weight of C6H6 is 78.113640

Note the argument reduce=True for func: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... an example of getting the center of mass from an Atoms object using func:ase.Atoms.get_center_of_mass.

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

 $\begin{array}{l} \text{COM} = \frac{\sum m_i \cdot r_i}{\sum m_i} \\ \text{The center of mass is essentially the average position of the atoms,} \end{array}$ weighted by the mass of each atom.

```
from ase import Atoms
    from ase.structure import molecule
2
    from ase.io import write
    # ammonia
5
    atoms = molecule('NH3')
6
    print 'COM1 = ',atoms.get_center_of_mass() # cartesian coordinates
9
    # compute the center of mass by hand
10
11
    from ase.data import atomic_masses
12
    import numpy as np
13
    pos = atoms.positions
    masses = atoms.get_masses()
14
15
    COM = np.array([0., 0., 0.])
16
    for m,p in zip(masses, pos):
17
18
      COM += m*p
    COM /= sum(masses)
19
20
    print 'COM2 = ', COM
21
22
    # one-line linear algebra definition of COM
23
    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: func:ase.Atoms.get_moments_of_inertia

Here are several examples.

```
from ase import *
2 from ase.structure import molecule
    from ase.units import *
3
   print 'linear rotors: I = [Ia Ia 0]'
   atoms = molecule('CO2')
7
    print ' CO2 moments of inertia: ',atoms.get_moments_of_inertia()
8
    print
10 print 'symmetric rotors (IA = Ib) < Ic'
11 atoms = molecule('NH3')
    print ' NH3 moments of inertia: ',atoms.get_moments_of_inertia()
12
13
    print
14
    atoms = molecule('C6H6')
15
16
    print ' C6H6 moments of inertia: ',atoms.get_moments_of_inertia()
    print
17
18
    print 'symmetric rotors (IA = Ib) > Ic'
19
  atoms = molecule('CH3C1')
20
21 print 'CHCl3 moments of inertia: ',atoms.get_moments_of_inertia()
22
    print
23
    print 'spherical rotors Ia = Ib = Ic'
24
25 atoms = molecule('CH4')
26 print ' CH4 moments of inertia: ',atoms.get_moments_of_inertia()
27 print
28
    print 'unsymmetric rotors Ia != Ib != Ic'
29
30 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
CHC13 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
    C3H7C1 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 func:ase.Atoms.get_distance method to make this easy. You have to specify the indices of the two atoms you want the distance between. Don't forget 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))
```

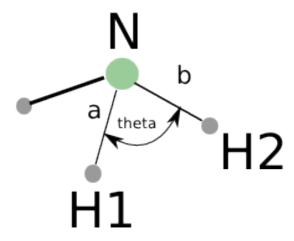
```
atom symbol
=======

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-H₁ and N-H₂. In the next example, we utilize functions in mod:numpy to perform the calculations, specifically the func:numpy.arccos function, the func:numpy.dot function, and func:numpy.linalg.norm functions.

```
from ase import Atoms
    from ase.structure import molecule
    from ase.io import write
    # ammonia
5
6
    atoms = molecule('NH3')
    print 'atom symbol'
8
    print '=====;
9
    for i, atom in enumerate(atoms):
10
      print '%2i %3s' % (i,atom.symbol)
11
12
    a = atoms.positions[0] - atoms.positions[1]
13
    b = atoms.positions[0] - atoms.positions[2]
14
15
16
    from numpy import arccos, dot, pi
    from numpy.linalg import norm
17
18
    theta_rad = arccos(dot(a,b)/(norm(a)*norm(b))) # in radians
19
20
^{21}
    print 'theta = %1.1f degrees' % (theta_rad*180./pi)
```

2



Η

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

```
3 H
theta = 106.3 degrees
```

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

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

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

theta = 106.334624232

Dihedral angles There is support in ase for computing dihedral angles. Lets illustrate that for ethane. We will compute the dihedral angle between atoms 5, 1, 0, and 4. That is a H-C-C-H dihedral angle, and one can visually

see (although not here) that these atoms have dihedral angle of 60° . See Figure 9.

```
# calculate an ethane dihedral angle
2 from ase import Atoms
3 from ase.structure import molecule
   from ase.visualize import view
4
5
    import numpy as np
    atoms = molecule('C2H6')
    view(atoms)
9
10
    print 'atom symbol'
    print '=====;
11
    for i, atom in enumerate(atoms):
12
13
      print '%2i %3s' % (i,atom.symbol)
14
15
    da = atoms.get_dihedral([5,1,0,4])*180./np.pi
    print 'dihedral angle = %1.2f degrees' % da
16
```

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

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 mod:ase.Atoms object so that it knows how to get the data, and then a DFT calculation must be run.

Here is an example of getting the energy and forces from a CO molecule. The forces in this case are very high, indicating that this geometry is not close to the ground state geometry. Note that the forces are only along the x-axis, which is along the molecular axis. We will see to minimize this force in sections Manual determination and Automatic geometry optimization with vasp.

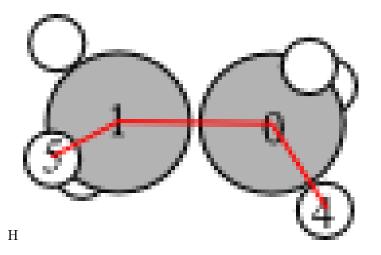


Figure 9: Schematic of the calculated ethane dihedral angle.

Note: This is your first DFT calculation in the book! see ISMEAR

```
from ase import Atoms, Atom
    from jasp import *
2
    import numpy as np
3
    np.set_printoptions(precision=3, suppress=True)
    6
7
8
9
10
    with jasp('molecules/simple-co', #output dir
              xc='PBE', # the exchange-correlation functional nbands=6, # number of bands
11
12
              encut=350, # planewave cutoff
13
14
              ismear=1, # Methfessel-Paxton smearing
              sigma=0.01,# very small smearing factor for a molecule
15
              atoms=co):
16
17
        print 'energy = {0} eV'.format(co.get_potential_energy())
18
        print co.get_forces()
19
```

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

4.3.3 The density of states

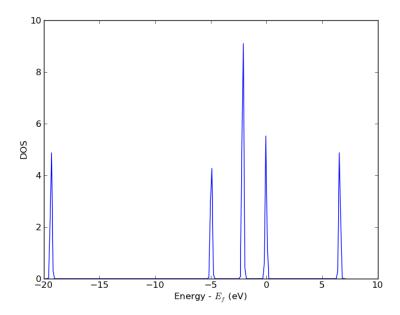
The density of states gives you the number of electronic states (i.e. orbitals) that have a particular energy. We can get this information from the last calculation we just ran without having to run another DFT calculation.

Warning:

Once y that jasp is unintuitive is that whenever you are "inside" (indented) a wirn 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 *
5
    with jasp('molecules/simple-co') as calc:
        dos = DOS(calc)
        plot(dos.get_energies(),dos.get_dos())
        xlabel('Energy - $E_f$ (eV)')
8
        ylabel('DOS')
9
10
    \# make sure you save the figure outside the with statement, or provide
11
12
    # the correct relative or absolute path to where you want it.
    savefig('images/co-dos.png')
```

None



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 calculation, 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

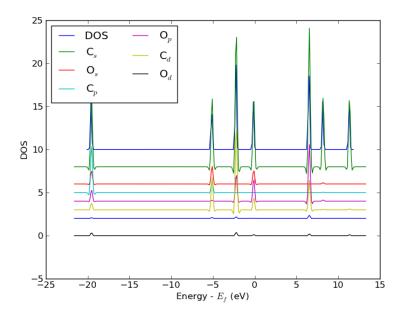


You can get spin-polarized atom-projected densi states, and magnetization projected density of states. http://cms.mpi.univie.ac.at/vasp/vasp/DOSCAR_file.html#doscar for details.

comes from the DOSCAR file.

```
from ase.dft.dos import *
1
    from jasp import *
2
    from pylab import *
3
    # get the geometry from another calculation
    with jasp('molecules/simple-co') as calc:
6
         atoms = calc.get_atoms()
7
8
9
    with jasp('molecules/co-ados',
10
         encut=300,
         xc='PBE',
11
                                # these are the cutoff radii for projected states
12
         rwigs=[1.0, 1.0],
         atoms=atoms) as calc:
13
         calc.calculate()
14
15
         # now get results
16
17
         dos = DOS(calc)
18
         plot(dos.get_energies(), dos.get_dos() + 10)
19
20
         ados = VaspDos(efermi=calc.get_fermi_level())
21
22
         energies = ados.energy
23
24
         plot(energies, ados.dos + 8) # these are the total DOS
25
         c_s = ados.site_dos(0, 's')
26
27
         c_p = ados.site_dos(0, 'p')
         o_s = ados.site_dos(1, 's')
28
29
         o_p = ados.site_dos(1, 'p')
30
         c_d = ados.site_dos(0,'d')
31
32
         o_d = ados.site_dos(1,'d')
33
34
         plot(energies, c_s + 6, energies, o_s + 5)
         plot(energies, c_p + 4, energies, o_p + 3)
35
36
         plot(energies, c_d, energies, o_d + 2)
         xlabel('Energy - $E_f$ (eV)')
37
         ylabel('DOS')
38
39
         legend(['DOS',
                 'C$_s$', 'O$_s$',
'C$_p$', 'O$_p$',
'C$_d$', 'O$_d$'],
40
41
42
                 ncol=2,loc='best')
43
    savefig('images/co-ados.png')
44
```

None



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 10) 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

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

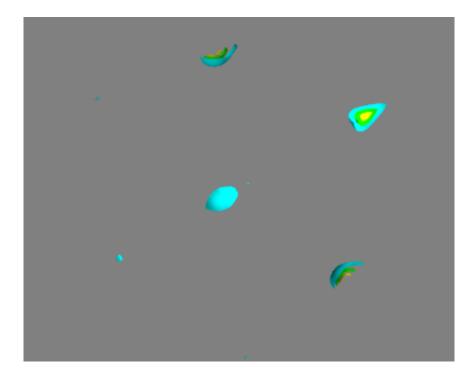


Figure 10: Charge density of a CO molecule.

None

4.3.6 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 [14, 22]. 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.

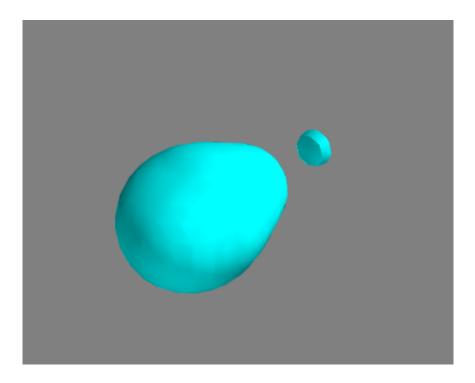


Figure 11: Charge density of a CO molecule centered in the unit cell.

```
from jasp import *
1
    from ase.structure import molecule
   atoms = molecule('H2O')
   atoms.center(vacuum=6)
   with jasp('molecules/h2o-bader',
6
              xc='PBE',
7
              encut=350,
8
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

cat ACF.dat

#	Х	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')

for atom in newatoms:
    print 'Atom', atom.symbol, 'Bader charge', atom.charge
```

```
Atom H Bader charge 0.9993
Atom H Bader charge 0.9997
Atom O Bader charge 0.001
```

4.3.7 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 *
1
2
    from ase.calculators.vasp import *
    from ase.units import Debye
3
    from ase import Atom, Atoms
5
6
    atoms = Atoms([Atom(^{\circ}C^{\circ},[2.422,0,0]),
                    Atom('0',[3.578,0,0])],
                 cell=(10,10,10))
8
    atoms.center()
10
11
    with jasp('molecules/co-centered',
12
               encut=350,
13
               xc='PBE',
               atoms=atoms) as calc:
15
         atoms.get_potential_energy()
16
17
        vcd = VaspChargeDensity()
18
19
         cd = np.array(vcd.chg[0])
20
21
        n0, n1, n2 = cd.shape
22
        s0 = 1.0/n0
23
24
        s1 = 1.0/n1
        s2 = 1.0/n2
25
26
        X, Y, Z = np.mgrid[0.0:1.0:s0,
27
                             0.0:1.0:s1,
28
                             0.0:1.0:s2]
29
```

```
30
        C = np.column_stack([X.ravel(),
31
32
                              Y.ravel(),
                              Z.ravel()])
33
34
         atoms = calc.get_atoms()
35
         uc = atoms.get_cell()
36
        real = np.dot(C, uc)
37
38
39
         #now convert arrays back to unitcell shape
         x = np.reshape(real[:, 0], (n0, n1, n2))
40
41
        y = np.reshape(real[:, 1], (n0, n1, n2))
        z = np.reshape(real[:, 2], (n0, n1, n2))
42
43
44
         nelements = n0*n1*n2
         voxel_volume = atoms.get_volume()/nelements
45
46
         total_electron_charge = -cd.sum()*voxel_volume
47
48
         electron_density_center = np.array([(cd*x).sum(),
49
50
                                              (cd*y).sum(),
51
                                              (cd*z).sum()])
         electron_density_center *= voxel_volume
52
         electron_density_center /= total_electron_charge
53
54
         electron_dipole_moment = -electron_density_center*total_electron_charge
55
56
         # now the ion charge center. We only need the Zval listed in the potcar
57
58
         from jasp.POTCAR import get_ZVAL
59
         LOP = calc.get_pseudopotentials()
60
        ppp = os.environ['VASP_PP_PATH']
61
62
        zval = {}
63
64
        for sym, ppath, hash in LOP:
             fullpath = os.path.join(ppp, ppath)
65
            z = get_ZVAL(fullpath)
66
67
            zval[sym] = z
         ion_charge_center = np.array([0.0, 0.0, 0.0])
68
         total_ion_charge = 0.0
69
70
         for atom in atoms:
            Z = zval[atom.symbol]
71
            total_ion_charge += Z
72
73
            pos = atom.position
             ion_charge_center += Z*pos
74
75
         ion_charge_center /= total_ion_charge
76
         ion_dipole_moment = ion_charge_center*total_ion_charge
77
78
         dipole_vector = (ion_dipole_moment + electron_dipole_moment)
79
80
         dipole_moment = ((dipole_vector**2).sum())**0.5/Debye
81
82
        print 'The dipole moment is {0:1.2f} Debye'.format(dipole_moment)
83
```

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

with jasp('molecules/co-centered') as calc:
dipole_vector = calc.get_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

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. mod: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.

```
1
    from ase import *
    from jasp import *
2
    import numpy as np
    np.set_printoptions(precision=3,suppress=True)
4
    bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
6
    energies = []
7
    for d in bond_lengths: #possible bond lengths
9
10
         co = Atoms([Atom('C',[0,0,0]),
11
                     Atom('0',[d,0,0])],
12
                    cell=(6,6,6))
13
14
        with jasp('molecules/co-{0}'.format(d), #output dir
15
                   xc='PBE',
16
                   nbands=6,
17
18
                   encut=350,
                   ismear=1,
19
20
                   sigma=0.01,
                   atoms=co):
21
             try:
```

```
e = co.get_potential_energy()
23
               energies.append(e)
24
              print 'd = %1.2f ang' % d
25
              print 'energy = %f eV' % e
26
              print 'forces = (eV/ang)\n', co.get_forces()
27
              print '' #blank line
28
29
           except (VaspSubmitted, VaspQueued):
30
               energies.append(None)
31
              pass
32
   if not None in energies:
33
34
       import matplotlib.pyplot as plt
       plt.plot(bond_lengths, energies, 'bo-')
35
36
       plt.xlabel('Bond length ($\AA$)')
       plt.ylabel('Total energy (eV)')
37
       plt.savefig('images/co-bondlengths.png')
38
    d = 1.05 ang
    energy = -14.215189 \text{ eV}
    forces = (eV/ang)
    [[-14.903
                  0.
                            0.
                  0.
     [ 14.903
                                  ]]
                            0.
    d = 1.10 \text{ ang}
    energy = -14.719882 eV
    forces = (eV/ang)
    [[-5.8 0. 0.]
     [5.8 0.
                   0.]]
    d = 1.15 \text{ 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)
```

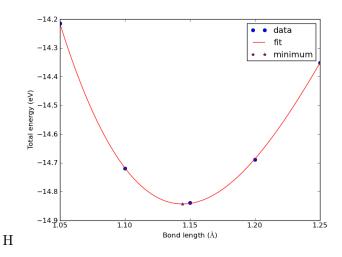


Figure 12: Energy vs CO bond length. The minimum appears to be near 1.15 Å by inspection.

```
[[ 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.

```
from jasp import *
2
    import numpy as np
    import matplotlib.pyplot as plt
3
    \verb"np.set_printoptions" (\texttt{precision=3,suppress=True})
5
    bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
    energies = []
7
8
    for d in bond_lengths: #possible bond lengths
9
10
         with jasp('molecules/co-{0}'.format(d)) as calc:
11
12
             atoms = calc.get_atoms()
13
             energies.append(atoms.get_potential_energy())
14
    # Now we fit an equation - cubic polynomial
15
    pp = np.polyfit(bond_lengths, energies, 3)
16
    dp = np.polyder(pp) # first derivative - quadratic
17
18
19
    # we expect two roots from the quadratic eqn. These are where the
    # first derivative is equal to zero.
20
    roots = np.roots(dp)
```

```
22
    # The minimum is where the second derivative is positive.
23
    dpp = np.polyder(dp) # second derivative - line
24
    secd = np.polyval(dpp, roots)
25
26
    minV = roots[secd > 0]
27
    minE = np.polyval(pp, minV)
28
29
    print 'The minimum energy is {0} eV at V = {1} Ang^3'.format(minE,minV)
30
31
   # plot the fit
32
33
    x = np.linspace(1.05, 1.25)
   fit = np.polyval(pp, x)
34
35
36
   plt.plot(bond_lengths, energies, 'bo ')
37
    plt.plot(x,fit, 'r-')
    plt.plot(minV, minE, 'm* ')
38
39 plt.legend(['DFT','fit','minimum'])
40 plt.xlabel('Bond length ($\AA$)')
41 plt.ylabel('Total energy (eV)')
42
    plt.savefig('images/co-bondlengths.png')
43
    plt.show()
```

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 *
1
    from jasp import *
3
    import numpy as np
    co = Atoms([Atom('C', [0,0,0]),
                Atom('0',[1.2,0,0])],
6
                cell=(6,6,6))
8
9
    with jasp('molecules/co-cg',
10
              xc='PBE',
              nbands=6,
11
              encut=350,
13
              ismear=1.
              sigma=0.01, # this is small for a molecule
14
15
              ibrion=2, # conjugate gradient optimizer
              nsw=5.
                          # do at least 5 steps to relax
16
17
              atoms=co) as calc:
18
        print 'Forces'
19
        print '=====;
20
21
        print co.get_forces()
22
        pos = co.get_positions()
23
        d = ((pos[0] - pos[1])**2).sum()**0.5
        print 'Bondlength = %1.2f angstroms' % d
25
```

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
2
    from jasp import *
    atoms = Atoms([Atom('H', [0.5960812, -0.7677068,
                                                             0.0000000]),
4
                     Atom('0',[0.0000000, 0.0000000, Atom('H',[0.5960812, 0.7677068,
                                                             0.0000000]),
5
                                                             0.0000000])],
6
                     cell=(8,8,8))
7
    with jasp('molecules/h2o_relax',
9
                xc='PBE',
10
                encut=400,
11
                ismear=0,# Gaussian smearing
12
                ibrion=2,
13
                ediff=1e-8,
14
                nsw=10,
15
                atoms=atoms) as calc:
16
17
         print "forces"
18
         print '=====;
19
20
         print atoms.get_forces()
```

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 section Manual determination. e 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 func:numpy.polyfit, func:numpy.polyder, and func:numpy.polyval.

```
from jasp import *
1
    from ase.units import *
2
    bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
    energies = []
6
7
    for d in bond_lengths:
        with jasp('molecules/co-{0}'.format(d)) as calc:
8
             atoms = calc.get_atoms()
             energies.append(atoms.get_potential_energy())
10
11
12
    # fit the data
    pars = np.polyfit(bond_lengths, energies, 3)
13
    xfit = np.linspace(1.05, 1.25)
14
15
    efit = np.polyval(pars, xfit)
16
17
    # first derivative
    dpars = np.polyder(pars)
18
    # find where the minimum is. chose the second one because it is the
19
    # minimum we need.
20
    print 'roots of first derivative are {0}'.format(np.roots(dpars))
21
22
    d_{min} = 1.14425395
23
24
    # second derivative
    ddpars = np.polyder(dpars)
25
26
    #curvature at minimum = force constant
27
28
    k = np.polyval(ddpars, d_min)/kg*s**2
29
30
    # reduced mass
    from ase.data import atomic_masses
31
    C_mass = atomic_masses[6]/kg
32
    0_mass = atomic_masses[8]/kg
```

```
mu = (C_mass*0_mass)/(C_mass + 0_mass)
35
36
37
    frequency = 1./(2.*np.pi)*np.sqrt(k/mu)
    print 'The CO vibrational frequency is {0} cm^{{-1}}'.format(frequency/3e10)
38
39
40
    import matplotlib.pyplot as plt
    plt.plot(bond_lengths, energies, 'bo ')
41
    plt.plot(xfit, efit, 'b-')
42
   plt.xlabel('Bond length ($\AA$)')
43
44
    plt.ylabel('Total energy (eV)')
    plt.show()
45
```

```
roots of first derivative are [ 1.43384501 1.14425395] The frequency is 2143.60766258 cm<sup>-{-1}</sup>
```

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 [8]. In practice, we usuall wow a packaged code to automate this, which we cover in the next section.

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).

 $E_ZPE = 0.133 \text{ eV}$

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
2 from ase import Atoms, Atom
3
   from jasp import *
    import ase.units
6
    atoms = Atoms([Atom('H', [0.5960812, -0.7677068,
                                                        0.0000000]),
7
                   Atom('0', [0.0000000, 0.0000000, 0.0000000]),
                   Atom('H', [0.5960812, 0.7677068, 0.0000000])],
8
9
                   cell=(8,8,8))
10
    with jasp('molecules/h2o_vib',
11
12
              xc='PBE',
              encut=400,
13
14
              ismear=0,# Gaussian smearing
              ibrion=6,# finite differences with symmetry
15
              nfree=2, # central differences (default)
16
              potim=0.015,# default as well
17
              ediff=1e-8,
18
19
              nsw=1,
              atoms=atoms) as calc:
20
21
        print 'Forces'
22
23
        print '====='
24
        print atoms.get_forces()
25
        # vibrational energies are in eV
26
        energies, modes = calc.get_vibrational_modes()
27
28
29
        print energies
```

Forces

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

- 1. a symmetric stretch at 3657 cm^{-1} (3723)
- 2. an asymmetric stretch at 3756 cm^{-1} (3836)

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

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

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

Note: the calc.get_vibrational_frequencies() is only available in jasp (7/12/2012).

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 *
1
    import numpy as np
    c = 3e10 # speed of light cm/s
    h = 4.135667516e-15 \# eV/s
6
    # first, get the frequencies.
    with jasp('molecules/h2o_vib') as calc:
7
8
        freq = calc.get_vibrational_frequencies()
    ZPE = 0.0
10
11
    for f in freq:
12
        if not isinstance(f,float):
            continue #skip complex numbers
13
        nu = f*c # convert to frequency
14
15
        ZPE += 0.5*h*nu
16
17
    print 'The ZPE of water is {0:1.3f} eV'.format(ZPE)
18
19
    ZPE = np.sum([0.5*h*f*c for f in freq if isinstance(f, float)])
20
    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.

```
1
    from ase import Atoms, Atom
2
    from jasp import *
    atoms = Atoms([Atom('H', [0.5960812, -0.7677068,
                                                         0.0000000]),
                    Atom('0', [0.0000000, 0.0000000,
                                                         0.0000000]),
5
                    Atom('H', [0.5960812, 0.7677068,
6
7
                    cell=(8,8,8))
8
    with jasp('molecules/h2o_relax',
              xc='PBE',
10
               encut=400,
11
              ismear=0,# Gaussian smearing
12
               ibrion=2,
13
               ediff=1e-8,
              nsw=10,
15
               atoms=atoms) as calc:
16
17
18
        print atoms.get_forces()
```

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

Next, we instruct 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 *
2
3
4
    #read in relaxed geometry
    with jasp('molecules/h2o_relax') as calc:
5
        atoms = calc.get_atoms()
8
    # now define a new calculator
    with jasp('molecules/h2o_vib_dfpt',
9
              xc='PBE',
10
11
               encut=400,
12
               ismear=0,# Gaussian smearing
               ibrion=7, # switches on the DFPT vibrational analysis (with no symmetry constraints)
13
              nfree=2.
14
              potim=0.015,
15
               lepsilon=True, # enables to calculate and to print the BEC tensors
16
```

```
17 lreal=False,
18 nsw=1,
19 nwrite=3, # affects OUTCAR verbosity: explicitly forces SQRT(mass)-divided eigenvectors to be printed
20 atoms=atoms) as calc:
21 calc.calculate(atoms)
```

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

```
1
    #!/bin/bash
    # A utility for calculating the vibrational intensities from VASP output (OUTCAR)
    # (C) David Karhanek, 2011-03-25, ICIQ Tarragona, Spain (www.iciq.es)
3
    # extract Born effective charges tensors
    printf "..reading OUTCAR"
6
    BORN_NROWS='grep NIONS OUTCAR | awk '{print $12*4+1}'
    if [ 'grep 'BORN' OUTCAR | wc -1' = 0 ]; then \
8
       printf " .. FAILED! Born effective charges missing! Bye! \n\n" ; exit 1 ; fi
9
    grep "in e, cummulative" -A $BORN_NROWS OUTCAR > born.txt
10
11
    # extract Eigenvectors and eigenvalues
12
    if [ 'grep 'SQRT(mass)' OUTCAR | wc -1' != 1 ]; then \
13
       printf " .. FAILED! Restart VASP with NWRITE=3! Bye! \n\n"; exit 1; fi
14
    EIG_NVIBS='grep -A 2000 'SQRT(mass)' OUTCAR | grep 'cm-1' | wc -1'
15
    EIG_NIONS='grep NIONS OUTCAR | awk '{print $12}'
16
    EIG_NROWS='echo "($EIG_NIONS+3)*$EIG_NVIBS+3" | bc'
17
    grep -A $(($EIG_NROWS+2)) 'SQRT(mass)' OUTCAR | tail -n $(($EIG_NROWS+1)) | sed 's/f\/i/fi /g' > eigenvectors.txt
18
    printf " ..done\n"
19
20
21
    # set up a new directory, split files - prepare for parsing
22
    printf "..splitting files"
    mkdir intensities; mv born.txt eigenvectors.txt intensities/
23
    cd intensities/
   let NBORN_NROWS=BORN_NROWS-1
25
    let NEIG_NROWS=EIG_NROWS-3
26
27
    let NBORN_STEP=4
    let NEIG_STEP=EIG_NIONS+3
28
    tail -n $NBORN_NROWS born.txt > temp.born.txt
   tail -n $NEIG_NROWS eigenvectors.txt > temp.eige.txt
30
    mkdir inputs; mv born.txt eigenvectors.txt inputs/
31
32
    split -a 3 -d -l $NEIG_STEP temp.eige.txt temp.ei.
    split -a 3 -d -1 $NBORN_STEP temp.born.txt temp.bo.
33
    mkdir temps01; mv temp.born.txt temp.eige.txt temps01/
34
    for nu in 'seq 1 $EIG_NVIBS'; do
35
     let nud=nu-1; ei='printf "%03u" $nu'; eid='printf "%03u" $nud'; mv temp.ei.$eid eigens.vib.$ei
36
37
    done
    for s in 'seq 1 $EIG_NIONS'; do
38
    let sd=s-1; bo='printf "%03u" $s'; bod='printf "%03u" $sd'; mv temp.bo.$bod borncs.$bo
39
40
    done
    printf " ..done\n"
41
42
43 # parse deviation vectors (eig)
44 printf "..parsing eigenvectors"
   let sad=$EIG_NIONS+1
```

```
for nu in 'seq 1 $EIG_NVIBS'; do
46
     nuu='printf "%03u" $nu'
47
     tail -n $sad eigens.vib.$nuu | head -n $EIG_NIONS | awk '{print $4,$5,$6}' > e.vib.$nuu.allions
48
     split -a 3 -d -l 1 e.vib.$nuu.allions temp.e.vib.$nuu.ion.
49
50
     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
51
52
     done
53
    done
   printf " ..done\n"
54
55
    # parse born effective charge matrices (born)
56
57
    printf "..parsing eff.charges"
    for s in 'seq 1 $EIG_NIONS'; do
58
    ss='printf "%03u" $s'
     awk '{print $2,$3,$4}' borncs.$ss | tail -3 > bornch.$ss
60
61
    done
62
    mkdir temps02; mv eigens.* borncs.* temps02/
    printf " ..done\n"
63
    # parse matrices, multiply them and collect squares (giving intensities)
65
    printf "..multiplying matrices, summing "
66
    for nu in 'seq 1 $EIG_NVIBS'; do
67
    nuu='printf "%03u" $nu'
68
     int=0.0
69
70
     for alpha in 1 2 3; do
                                          # summing over alpha coordinates
71
      sumpol=0.0
      for s in 'seq 1 $EIG_NIONS'; do # summing over atoms
72
       ss='printf "%03u" $s'
73
       awk -v a="$alpha" '(NR==a){print}' bornch.$ss > z.ion.$ss.alpha.$alpha
74
        \textit{\# summing over beta coordinates and multiplying } \textit{Z(s,alpha)*e(s) done by the following awk script } \\
75
       paste z.ion.$ss.alpha.$alpha e.vib.$nuu.ion.$ss | \
76
       awk '{pol=$1*$4+$2*$5+$3*$6; print $0," ",pol}' > matr-vib-${nuu}-alpha-${alpha}-ion-${ss}
77
78
      sumpol='cat matr-vib-${nuu}-alpha-${alpha}-ion-* | awk '{sum+=$7} END {print sum}''
79
80
      int='echo "$int+($sumpol)^2" | sed 's/[eE]/*10^/g' | bc -1'
81
     freq='awk '(NR==1){print $8}' temps02/eigens.vib.$nuu'
82
     echo "$nuu $freq $int">> exact.res.txt
83
     printf "."
84
    done
85
    printf " ..done\n"
86
87
    # format results, normalize intensities
88
89
    printf "..normalizing intensities"
    max='awk '(NR==1){max=$3} $3>=max {max=$3} END {print max}' exact.res.txt'
90
91
    awk -v max="$max" '{printf "%03u %6.1f %5.3f\n",$1,$2,$3/max}' exact.res.txt > results.txt
    printf " ..done\n"
92
    # clean up, display results
94
95
    printf "..finalizing:\n"
96
    mkdir temps03; mv bornch.* e.vib.*.allions temps03/
    mkdir temps04; mv z.ion* e.vib.*.ion.* temps04/
97
98 mkdir temps05; mv matr-* temps05/
99
    mkdir results; mv *res*txt results/
    let NMATRIX=$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
800
      45.0 0.000
009 136.1 0.345
```

 $102 \\ 103$

104

105

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

```
#!/bin/bash
printf "..reformatting and normalizing intensities"
   cd intensities/results/
5 nlns='wc -l exact.res.txt | awk '{print $1}' '; let bodylns=nlns-6
6 head -n $bodylns exact.res.txt > temp.reform.res.txt
   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
10 printf " ..done\n..normal modes:\n"
11    rm temp.reform.res.txt
12 cat reform.res.txt
13
   cd ../..
     ..reformatting and normalizing intensities ..done
     ..normal modes:
     001 3827.3 0.726
     002 3713.0 0.019
     003 1587.2 1.000
```



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

See also [11]. For HREELS simulations see [17].

4.6.1 TODO ell scripts are rough. One day I should rewrite this in python.

4.7 Thermochemical properties of molecules

ase provides a thermochemistry module.

```
1
    from ase.structure import molecule
    from ase.thermochemistry import IdealGasThermo
    from jasp import *
    atoms = molecule('N2')
5
6
    atoms.set_cell((10,10,10), scale_atoms=False)
    # first we relax a molecule
9
    with jasp('molecules/n2-relax',
              xc='PBE',
10
11
               encut=300.
              ibrion=2,
12
13
              nsw=5,
              atoms=atoms) as calc:
14
15
        electronicenergy = atoms.get_potential_energy()
16
17
    # next, we get vibrational modes
18
    with jasp('molecules/n2-vib',
19
              xc='PBE',
20
              encut=300.
21
22
              ibrion=6,
23
              nfree=2,
              potim=0.15,
24
25
              nsw=1,
26
              atoms=atoms) as calc:
27
        calc.calculate()
28
        vib_freq = calc.get_vibrational_frequencies() # in cm^1
29
30
         #convert wavenumbers to energy
        h = 4.1356675e-15 \# eV*s
31
        c = 3.0e10 \# cm/s
        vib_energies = [h*c*nu for nu in vib_freq]
33
34
        print vib_energies
35
   # # now we can get some properties. Note we only need one vibrational
36
    # energy since there is only one mode. This example does not work if
    # you give all the energies because one energy is zero.
38
    thermo = IdealGasThermo(vib_energies=vib_energies[0:0],
```

```
40 electronicenergy=electronicenergy, atoms=atoms,
41 geometry='linear', symmetrynumber=2, spin=0)
42
43 # temperature in K, pressure in Pa, G in eV
44 G = thermo.get_free_energy(temperature=298.15, pressure=101325.)
```

[0.291] 34323953859, 0.016977524105896053, 0.016977524105896053, 2.8536105750000002e Enthalpy components at T = 298.15 K:

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:

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: $O_2 \to 2O$. 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's do that.

Simple estimate of O₂ dissociation energy

```
from jasp import *
1
2
    from ase import Atom, Atoms
    atoms = Atoms([Atom('0',[5,5,5])],
                   cell=(10,10,10))
6
7
    with jasp('molecules/0',
              xc='PBE',
8
9
               encut=400,
10
               ismear=0,
               atoms=atoms) as calc:
11
12
        try:
            E_O = atoms.get_potential_energy()
13
        except (VaspSubmitted, VaspQueued):
14
            E_0 = None
15
16
    # now relaxed 02 dimer
17
18
    atoms = Atoms([Atom('0',[5,5,5]),
19
20
                    Atom('0',[6.22, 5,5])],
                   cell=(10,10,10)
21
22
    with jasp('molecules/02',
23
              xc='PBE',
24
               encut=400.
25
26
               ismear=0,
27
               ibrion=2, # make sure we relax the geometry
              nsw=10.
28
               atoms=atoms) as calc:
30
        try:
            E_02 = atoms.get_potential_energy()
31
32
         except (VaspSubmitted, VaspQueued):
            E_02 = None
33
34
    if None not in (E_0, E_02):
35
        print '02 -> 20 D = \{0:1.3f\} eV'.format(2*E_0 - E_02)
36
```

 $02 \rightarrow 20 D = 8.521 eV$

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 Vapo use spin-polarization, and give initial guesses for the magnetic moments of the atoms. Let's 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 INCAR=2, and we set initial guesses for magnetic moments on the atoms with the magmom keyword. In a triplet state there are two electrons with spins of the same sign.

```
1
    from jasp import *
2
    from ase import Atom, Atoms
3
    atoms = Atoms([Atom('0', [5,5,5], magmom=2)],
                   cell=(10,10,10)
5
6
    with jasp('molecules/0-sp-triplet',
7
              xc='PBE',
8
9
               encut=400,
               ismear=0,
10
               ispin=2,
11
12
               atoms=atoms) as calc:
13
             E_0 = atoms.get_potential_energy()
14
        except (VaspSubmitted, VaspQueued):
15
16
             E_0 = None
17
18
    print 'Magnetic moment on 0 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
    # now relaxed 02 dimer
20
    atoms = Atoms([Atom('0',[5,5,5],magmom=1),
21
                   Atom('0', [6.22, 5,5], magmom=1)],
22
                   cell=(10,10,10))
23
24
    with jasp('molecules/02-sp-triplet',
25
               xc='PBE'.
26
               encut=400,
27
               ismear=0,
28
               ispin=2, # turn spin-polarization on
29
               ibrion=2, # make sure we relax the geometry
30
31
               nsw=10.
               atoms=atoms) as calc:
32
33
        try:
```

```
E_02 = atoms.get_potential_energy()
34
35
         except (VaspSubmitted, VaspQueued):
            E_02 = None
36
37
38
    # verify magnetic moment
    print 'Magnetic moment on 02 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
39
40
41
    if None not in (E_0, E_02):
        print '02 -> 20 D = \{0:1.3f\} eV'.format(2*E_0 - E_02)
42
```

```
Magnetic moment on 0 = 2.0000072 Bohr magnetons Magnetic moment on 02 = 2.0000083 Bohr magnetons 02 \rightarrow 20 D = 6.668 eV
```

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

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

```
from jasp import *
1
2
    from ase.dft.dos import *
3
    with jasp('molecules/02-sp-triplet') as calc:
5
        dos = DOS(calc, width=0.2)
6
        d_up = dos.get_dos(spin=0)
7
        d_down = dos.get_dos(spin=1)
        e = dos.get_energies()
8
    ind = e <= 0.0
10
    # integrate up to OeV
11
    print 'number of up states = {0}'.format(np.trapz(d_up[ind],e[ind]))
12
    print 'number of down states = {0}'.format(np.trapz(d_down[ind],e[ind]))
13
14
    import pylab as plt
15
   plt.plot(e, d_up, e, -d_down)
17 plt.xlabel('energy [eV]')
    plt.ylabel('DOS')
18
19
    plt.legend(['up','down'])
    plt.savefig('images/02-sp-dos.png')
```

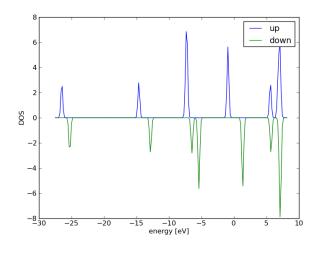


Figure 13: Spin-polarized density of states for the O₂ molecule.

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

Η

You can see in Figure 13 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 O₂ dissociation energy

```
from jasp import *
2
    from ase import Atom, Atoms
    encuts = [250, 300, 350, 400, 450, 500, 550]
4
5
6
    for encut in encuts:
        atoms = Atoms([Atom('0', [5,5,5], magmom=2)],
7
                        cell=(10,10,10))
8
9
10
         with jasp('molecules/0-sp-triplet-{0}'.format(encut),
                   xc='PBE',
11
                   encut=encut,
12
13
                   ismear=0,
14
                   ispin=2,
15
                   atoms=atoms) as calc:
16
                 E_0 = atoms.get_potential_energy()
17
             except (VaspSubmitted, VaspQueued):
18
```

```
E_0 = None
19
20
21
         # now relaxed 02 dimer
         atoms = Atoms([Atom('0',[5,5,5],magmom=1),
22
23
                         Atom(^{0}, [6.22, 5,5], magmom=1)],
                   cell=(10,10,10)
24
25
         with jasp('molecules/02-sp-triplet-{0}'.format(encut),
26
                   xc='PBE',
27
28
                   encut=encut
                   ismear=0,
29
30
                   ispin=2, # turn spin-polarization on
                   ibrion=2, # make sure we relax the geometry
31
32
                   atoms=atoms) as calc:
33
34
             try:
35
                 E_02 = atoms.get_potential_energy()
             except (VaspSubmitted, VaspQueued):
36
                 E_02 = None
37
38
         if None not in (E_0, E_02):
39
             d = 2*E_0 - E_02
40
             D.append(d)
41
             print '02 \rightarrow 20 encut = {0} D = {1:1.3f} eV'.format(encut, d)
42
43
    import matplotlib.pyplot as plt
44
45
    plt.plot(encuts, D)
    plt.xlabel('ENCUT (eV)')
46
47
    plt.ylabel('0$_2$ dissociation energy (eV)')
    plt.savefig('images/02-dissociation-convergence.png')
48
```

```
02 -> 20 encut = 250 D = 6.696 eV

02 -> 20 encut = 300 D = 6.727 eV

02 -> 20 encut = 350 D = 6.708 eV

02 -> 20 encut = 400 D = 6.668 eV

02 -> 20 encut = 450 D = 6.650 eV

02 -> 20 encut = 500 D = 6.648 eV

02 -> 20 encut = 550 D = 6.650 eV
```

Based on these results (Figure 14), 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.

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

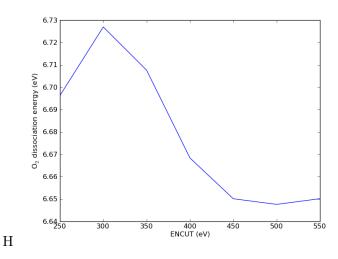


Figure 14: Convergence study of the \mathcal{O}_2 dissociation energy as a function of ENCUT.

```
for encut in encuts:
      OUTCAR = 'molecules/02-sp-triplet-{0}/OUTCAR'.format(encut)
      f = open(OUTCAR, 'r')
6
      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
                              Total CPU time used (sec):
   550:
                                                                   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.

```
1
    from jasp import *
    from ase import Atom, Atoms
2
3
    sigmas = [0.2, 0.1, 0.05, 0.02, 0.01, 0.001]
4
5
6
7
    for sigma in sigmas:
         atoms = Atoms([Atom('0',[5, 5, 5], magmom=2)],
8
                        cell=(10,10,10))
9
10
        with jasp('molecules/0-sp-triplet-sigma-{0}'.format(sigma),
11
                   xc='PBE',
12
13
                   encut=400,
14
                   ismear=0.
                   sigma=sigma,
15
16
                   ispin=2,
17
                   atoms=atoms) as calc:
18
                 E_0 = atoms.get_potential_energy()
19
20
             except (VaspSubmitted, VaspQueued):
21
                 E_0 = None
22
23
         # now relaxed 02 dimer
         atoms = Atoms([Atom('0',[5,
                                         5, 5],magmom=1),
24
                        Atom('0',[6.22, 5, 5],magmom=1)],
25
                   cell=(10,10,10)
26
27
28
         with jasp('molecules/02-sp-triplet-sigma-{0}'.format(sigma),
                   xc='PBE',
29
30
                   encut=400,
                   ismear=0.
31
32
                   sigma=sigma,
33
                   ispin=2, # turn spin-polarization on
                   ibrion=2, # make sure we relax the geometry
34
35
                   nsw=10.
                   atoms=atoms) as calc:
36
37
                 E_02 = atoms.get_potential_energy()
38
             except (VaspSubmitted, VaspQueued):
```

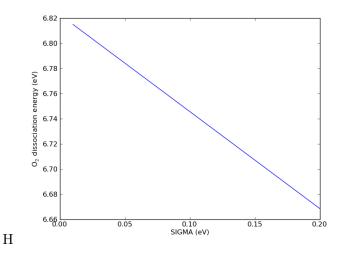


Figure 15: Effect of SIGMA on the oxygen dissociation energy.

```
40
                 E_02 = None
41
        if None not in (E_0, E_02):
42
43
            d = 2*E_0 - E_02
            D.append(d)
44
            print '02 -> 20 sigma = {0} D = {1:1.3f} eV'.format(sigma, d)
45
46
47
    import matplotlib.pyplot as plt
48
    plt.plot(sigmas, D, 'bo-')
    plt.xlabel('SIGMA (eV)')
49
    plt.ylabel('0$_2$ dissociation energy (eV)')
    plt.savefig('images/02-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 = 784 eV O2 -; 2O sigma = 0.02 D = 6.807 eV O2 -; 2O sigma = 0.01 D = 6.815 eV =

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 *
1
    from ase import Atom, Atoms
2
3
    # square box origin
    atoms = Atoms([Atom('0',[0,0,0],magmom=2)],
                  cell=(10,10,10))
    with jasp('molecules/0-square-box-origin',
8
              xc='PBE',
10
              encut=400,
              ismear=0,
11
12
              sigma=0.01,
              ispin=2,
13
14
              atoms=atoms) as calc:
15
            print 'Square box (origin): E = {0} eV'.format(atoms.get_potential_energy())
16
17
        except (VaspSubmitted, VaspQueued):
            pass
18
19
    # square box center
20
    atoms = Atoms([Atom('0',[5,5,5],magmom=2)],
21
                  cell=(10,10,10))
22
23
    with jasp('molecules/0-square-box-center',
25
              xc='PBE',
              encut=400,
26
27
              ismear=0,
              sigma=0.01,
28
29
              ispin=2,
              atoms=atoms) as calc:
30
31
            print 'Square box (center): E = {0} eV'.format(atoms.get_potential_energy())
32
33
        except (VaspSubmitted, VaspQueued):
34
            pass
35
    # square box random
36
    atoms = Atoms([Atom('0',[2.13, 7.32, 1.11],magmom=2)],
37
                  cell=(10,10,10))
38
39
    with jasp('molecules/0-square-box-random',
40
41
              xc='PBE',
              encut=400.
42
              ismear=0,
43
              sigma=0.01,
44
              ispin=2,
45
46
              atoms=atoms) as calc:
47
            print 'Square box (random): E = {0} eV'.format(atoms.get_potential_energy())
48
        except (VaspSubmitted, VaspQueued):
49
50
     Square box (origin): E = -1.516623 \text{ eV}
     Square box (center): E = -1.516623 eV
     Square box (random): E = -1.515359 \text{ 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's consider some non-square boxes.

```
# calculate O atom energy in orthorhombic boxes
1
2
    from jasp import *
    from ase import Atom, Atoms
3
    # orthorhombic box origin
5
    atoms = Atoms([Atom('0',[0,0,0],magmom=2)],
6
                   cell=(8,9,10))
8
    with jasp('molecules/O-orthorhombic-box-origin',
9
10
               xc='PBE',
11
               encut=400,
               ismear=0.
12
               sigma=0.01,
13
               ispin=2,
14
               atoms=atoms) as calc:
15
16
             print 'Orthorhombic box (origin): E = {0} eV'.format(atoms.get_potential_energy())
17
         except (VaspSubmitted, VaspQueued):
18
19
             pass
20
    # orthrhombic box center
^{21}
    atoms = Atoms([Atom(^{\circ}0^{\circ},[4, 4.5, 5],magmom=2)],
22
^{23}
                   cell=(8,9,10))
24
25
    with jasp('molecules/O-orthorhombic-box-center',
               xc='PBE',
26
               encut=400,
27
28
               ismear=0,
               sigma=0.01,
29
30
               ispin=2,
31
               atoms=atoms) as calc:
32
             print 'Orthorhombic box (center): E = {0} eV'.format(atoms.get_potential_energy())
33
         except (VaspSubmitted, VaspQueued):
34
35
36
    # orthorhombic box random
37
    atoms = Atoms([Atom('0',[2.13, 7.32, 1.11],magmom=2)],
38
                   cell=(8, 9, 10))
39
40
    with jasp('molecules/O-orthorhombic-box-random',
41
               xc='PBE',
42
43
               encut=400,
44
               ismear=0,
45
               sigma=0.01,
               ispin=2,
46
               atoms=atoms) as calc:
47
48
        try:
```

```
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

```
1
    from jasp import *
2
    from ase import Atom, Atoms
3
    atoms = Atoms([Atom('0',[5.1, 4.2, 6.1],magmom=2)],
4
                   cell=(8,9,10))
5
6
    with jasp('molecules/0-sp-triplet-lowsym',
7
               xc='PBE',
8
9
               encut=400,
               ismear=0.
10
               sigma=0.01,
11
               ispin=2,
12
13
               atoms=atoms) as calc:
14
            E_0 = atoms.get_potential_energy()
15
            print 'Magnetic moment on 0 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
16
        except (VaspSubmitted, VaspQueued):
17
            E_0 = None
18
19
    # now relaxed 02 dimer
20
^{21}
    atoms = Atoms([Atom('0', [5,5,5], magmom=1),
                    Atom('0',[6.22, 5,5],magmom=1)],
22
23
                   cell=(10,10,10))
24
    with jasp('molecules/02-sp-triplet',
25
               xc='PBE',
26
               encut=400,
27
28
               ismear=0,
               sigma=0.01,
29
               ispin=2, # turn spin-polarization on
30
31
               ibrion=2, # make sure we relax the geometry
               nsw=10,
32
33
               atoms=atoms) as calc:
34
        try:
            E_02 = atoms.get_potential_energy()
35
36
             # verify magnetic moment
            print 'Magnetic moment on 02 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
```

```
Magnetic moment on 0 = 2.0000016 Bohr magnetons Magnetic moment on 02 = 2.0000083 Bohr magnetons E_0: -1.893571 02 \rightarrow 20 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
2
3
    atoms = Atoms([Atom('0', [5,5,5], magmom=0)],
4
                   cell=(10,10,10))
5
    with jasp('molecules/0-sp-singlet',
7
               xc='PBE',
8
9
               encut=400,
10
               ismear=0,
11
               ispin=2,
               atoms=atoms) as calc:
12
13
         try:
             E_O = atoms.get_potential_energy()
14
         except (VaspSubmitted, VaspQueued):
15
16
             E_0 = None
17
18
    print 'Magnetic moment on 0 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20
    # now relaxed 02 dimer
    atoms = Atoms([Atom('0',[5,5,5],magmom=1),
21
                   Atom(^{0}, [6.22, 5,5], magmom=-1)],
22
23
                   cell=(10,10,10))
24
```

```
with jasp('molecules/02-sp-singlet',
25
              xc='PBE',
26
27
               encut=400.
              ismear=0,
28
29
               ispin=2, # turn spin-polarization on
               ibrion=2, # make sure we relax the geometry
30
               nsw=10,
31
32
               atoms=atoms) as calc:
33
34
            E_02 = atoms.get_potential_energy()
         except (VaspSubmitted, VaspQueued):
35
36
            E_02 = None
37
38
    # verify magnetic moment
39
    print atoms.get_magnetic_moment()
40
41
    if None not in (E_0, E_02):
        print '02 -> 20 D = \{0:1.3f\} eV'.format(2*E_0 - E_02)
42
```

```
Magnetic moment on 0 = 1.9998232 Bohr magnetons 0.0 02 \rightarrow 20 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/02-sp-singlet') as calc:
print 'singlet: ',calc.get_atoms().get_potential_energy()

with jasp('molecules/02-sp-triplet') as calc:
print 'triplet: ',calc.get_atoms().get_potential_energy()

-8.830101
```

-9.848287

You can see here the triplet state has an energy that is $1~{\rm eV}$ 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 *
1
2
    from ase import Atom, Atoms
    with jasp('molecules/02-sp-singlet') as calc:
        calc.clone('molecules/02-sp-singlet-magmoms')
6
    with jasp('molecules/02-sp-singlet-magmoms') as calc:
7
8
        calc.set(lorbit=11)
        atoms = calc.get_atoms()
9
10
        magmoms = atoms.get_magnetic_moments()
11
12
        print 'singlet ground state'
13
        for i,atom in enumerate(atoms):
            print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
14
15
        print atoms.get_magnetic_moment()
16
17
    with jasp('molecules/02-sp-triplet') as calc:
        calc.clone('molecules/02-sp-triplet-magmoms')
18
19
    with jasp('molecules/02-sp-triplet-magmoms') as calc:
20
21
        calc.set(lorbit=11)
22
        atoms = calc.get_atoms()
        magmoms = atoms.get_magnetic_moments()
23
        print
        print 'triplet ground state'
25
26
        for i,atom in enumerate(atoms):
            print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
27
        print atoms.get_magnetic_moment()
28
```

```
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 *
1
2
    from ase import Atom, Atoms
3
    atoms = Atoms([Atom('0', [4, 4.5, 5], magmom=2)],
                  cell=(8,9,10))
5
6
    with jasp('molecules/0-sp-triplet-lowsym-sv',
7
              xc='PBE',
8
              ismear=0,
              ispin=2,
10
              sigma=0.01,
11
              setups={'0':'_sv'},
12
              atoms=atoms) as calc:
13
14
            E_0 = atoms.get_potential_energy()
15
16
        except (VaspSubmitted, VaspQueued):
            E_0 = None
17
18
    print 'Magnetic moment on 0 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20
21
    # now relaxed 02 dimer
    atoms = Atoms([Atom('0',[5,5,5],magmom=1),
22
                   Atom('0',[6.22, 5,5],magmom=1)],
24
                  cell=(10,10,10))
25
    with jasp('molecules/02-sp-triplet-sv',
26
              xc='PBE',
27
28
              ismear=0,
              sigma=0.01,
29
30
              ispin=2, # turn spin-polarization on
31
              ibrion=2, # make sure we relax the geometry
              nsw=10,
32
              setups={'0':'_sv'},
33
              atoms=atoms) as calc:
34
35
            E_02 = atoms.get_potential_energy()
36
        except (VaspSubmitted, VaspQueued):
37
38
            E_02 = None
39
    # verify magnetic moment
40
    print 'Magnetic moment on 02 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
41
42
43
    if None not in (E_O, E_O2):
        print '02 -> 20 D = \{0:1.3f\} eV'.format(2*E_0 - E_02)
44
```



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.

```
1
    from ase.data.molecules import molecule
    from jasp import *
    # first we define our molecules. These will automatically be at the coordinates from the G2 database.
    CO = molecule('CO')
6
7
    CO.set_cell([8,8,8], scale_atoms=False)
    H20 = molecule('H20')
9
10
    H20.set_cell([8,8,8], scale_atoms=False)
11
12
    CO2 = molecule('CO2')
    CO2.set_cell([8,8,8], scale_atoms=False)
13
14
    H2 = molecule('H2')
15
16
    H2.set_cell([8,8,8], scale_atoms=False)
17
    # now the calculators to get the energies
18
    with jasp('molecules/wgs/CO',
19
              xc='PBE'.
20
               encut=350,
^{21}
22
               ismear=0,
               ibrion=2,
23
24
              nsw=10,
               atoms=CO) as calc:
25
26
             eC0 = C0.get_potential_energy()
27
        except (VaspSubmitted, VaspQueued):
28
29
             eCO = None
30
31
    with jasp('molecules/wgs/CO2',
               xc='PBE'.
32
               encut=350,
33
34
               ismear=0,
35
               ibrion=2,
36
               nsw=10,
               atoms=CO2) as calc:
37
38
             eCO2 = CO2.get_potential_energy()
39
```

```
except (VaspSubmitted, VaspQueued):
40
             eCO2 = None
41
42
    with jasp('molecules/wgs/H2',
43
44
               xc='PBE',
               encut=350,
45
               ismear=0,
46
47
               ibrion=2.
               nsw=10,
48
49
               atoms=H2) as calc:
50
51
             eH2 = H2.get_potential_energy()
         except (VaspSubmitted, VaspQueued):
52
53
             eH2 = None
54
    with jasp('molecules/wgs/H20',
55
56
               xc='PBE',
               encut=350,
57
               ismear=0,
58
               ibrion=2,
59
               nsw=10,
60
61
               atoms=H2O) as calc:
62
             eH20 = H20.get_potential_energy()
63
         except (VaspSubmitted, VaspQueued):
64
             eH20 = None
65
66
    if None in (eCO2, eH2, eCO, eH2O):
67
68
        pass
    else:
69
70
        dE = eCO2 + eH2 - eCO - eH2O
        print '\Delta E = {0:1.3f} eV'.format(dE)
71
        print '\Delta E = {0:1.3f} kcal/mol'.format(dE*23.06035)
72
        print '\Delta E = {0:1.3f} kJ/mol'.format(dE*96.485)
73
```

```
\text{Delta E = -0.720 eV} \text{Delta E = -16.614 kcal/mol} \text{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'
```

```
4 print '***** CO'
    with jasp('molecules/wgs/CO') as calc:
5
       print '#+begin_example'
        print calc
7
        print '#+end_example'
9
    print '***** CO2'
10
    with jasp('molecules/wgs/CO2') as calc:
11
       print '#+begin_example'
12
13
        print calc
        print '#+end_example'
14
15
    print '***** H2'
16
17
   with jasp('molecules/wgs/H2') as calc:
18
        print '#+begin_example'
19
        print calc
20
        print '#+end_example'
21
22 print '**** H20'
with jasp('molecules/wgs/H2O') as calc:
24
        print '#+begin_example'
25
        print calc
        print '#+end_example'
26
```

calculation summaries

\mathbf{CO}

```
· -----
 VASP calculation from /home/jkitchin/dft-org/molecules/wgs/CO
 converged: True
 Energy = -14.789536 eV
 Unit cell vectors (angstroms)
       Х
                    z
                          length
              У
 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,
                         zz,
                   уу,
                                yz,
                                     ΧZ,
                                             ху
           0.005 0.005 0.005 -0.000 -0.000 -0.000
Atom# sym
                position [x,y,z]
                                      tag rmsForce constraints
          [0.000
  0
                      0.000
                                0.490]
                                            0.01
                                                     TTT
       0
                                        0
  1
          [0.000
                      0.000
                                7.346] 0
                                            0.01
                                                     TTT
```

```
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)
                     Z
                            length
               У
  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,
                           zz,
                    уу,
                                  yz,
                                        ΧZ,
                                               хy
            0.009 0.009 0.008 -0.000 -0.000 -0.000
 Atom# sym
                 position [x,y,z]
                                         tag rmsForce constraints
                                  0.000]
   0
           [0.000
                       0.000
                                              0.00
       С
                                           0
                                                        TTT
   1
            [0.000
                       0.000
                                  1.177]
                                           0
                                              0.01
                                                        TTT
   2
            [0.000
                       0.000
                                  6.823]
                                           0
                                              0.01
                                                        TTT
```

INCAR Parameters:

```
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)
   H2
  VASP calculation from /home/jkitchin/dft-org/molecules/wgs/H2
  converged: True
  Energy = -6.744001 \text{ eV}
  Unit cell vectors (angstroms)
                У
                             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,
                     уу,
                            zz,
                                   yz,
                                          ΧZ,
             0.000 0.000 0.000 -0.000 -0.000 -0.000
 Atom#
       \operatorname{\mathtt{sym}}
                  position [x,y,z]
                                           tag rmsForce constraints
   0
        Η
            [0.000
                        0.000
                                   0.376]
                                            0
                                                0.00
                                                          TTT
   1
       Η
            [0.000
                        0.000
                                   7.624]
                                            0
                                                0.00
                                                          TTT
```

```
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$_2$O
: ------
 VASP calculation from /home/jkitchin/dft-org/molecules/wgs/H20
 converged: True
 Energy = -14.193569 eV
 Unit cell vectors (angstroms)
                    Z
                           length
              У
 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,
                          zz,
                   уу,
                                yz,
                                      ΧZ,
            0.005 0.004 0.005 -0.000 -0.000 -0.000
Atom# sym
                position [x,y,z]
                                       tag rmsForce constraints
  0
           [0.000
                      0.000
                                           0.06
                                                      TTT
       0
                                0.122]
```

7.522]

7.522]

0 0.04

0 0.04

TTT

TTT

0.766

7.234

1

2

Η

H

[0.000

[0.000

```
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',
9
               encut=350,
10
11
               ismear=0,
               ibrion=6,
12
13
               nfree=2,
               potim=0.02,
14
               nsw=1,
15
               atoms=CO) as calc:
16
         calc.calculate()
17
18
        vib_freq = calc.get_vibrational_frequencies()
        print vib_freq
19
```

[2115.528894, 60.59487800000001, \$\square\$ 94878000000001, (0.987178+0j), (17.958586+0j),

CO has only one vibrational mode.

CO_2 vibrations

```
from jasp import *
1
    # get relaxed geometry
    with jasp('molecules/wgs/CO2') as calc:
        CO2 = calc.get_atoms()
6
    # now do the vibrations
    with jasp('molecules/wgs/CO2-vib',
8
               xc='PBE',
               encut=350,
10
11
               ismear=0,
12
               ibrion=6,
              nfree=2,
13
               potim=0.02,
               nsw=1,
15
               atoms=CO2) as calc:
16
^{17}
         calc.calculate()
        vib_freq = calc.get_vibrational_frequencies()
18
19
        print vib_freq
```

[2352.9012849999999, 1316.6895039999999, 635.01591299999995, 635.01591299999995, (0.

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

H₂ vibrations

```
from jasp import *

# get relaxed geometry
with jasp('molecules/wgs/H2') as calc:
```

```
H2 = calc.get_atoms()
5
6
    # now do the vibrations
    with jasp('molecules/wgs/H2-vib',
8
9
               xc='PBE',
               encut=350,
10
               ismear=0,
11
12
               ibrion=6,
               nfree=2,
13
14
               potim=0.02,
               nsw=1,
15
16
               atoms=H2) as calc:
        calc.calculate()
17
18
        vib_freq = calc.get_vibrational_frequencies()
19
        print vib_freq
```

[4281.9177490000002, 129.14685499999999, 129.14685499999999, 0.0, 0.0, (1.0000000000

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

H\$_2\$O vibrations

```
from jasp import *
1
    # get relaxed geometry
3
    with jasp('molecules/wgs/H2O') as calc:
        H20 = calc.get_atoms()
6
    # now do the vibrations
    with jasp('molecules/wgs/H20-vib',
8
               xc='PBE',
               encut=350,
10
               ismear=0,
11
12
               ibrion=6,
               nfree=2,
13
               potim=0.02,
               nsw=1,
15
               atoms=H2O) as calc:
16
17
         calc.calculate()
18
        vib_freq = calc.get_vibrational_frequencies()
19
        print vib_freq
```

[3782.0622130000002, 3672.1246000000001, 1586.23055, 135.82763, 16.280411000000001,

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
1
    from jasp import *
3
    import numpy as np
    import matplotlib.pyplot as plt
6
    # first we get the electronic energies
    with jasp('molecules/wgs/CO') as calc:
7
        CO = calc.get_atoms()
8
9
        E_CO = CO.get_potential_energy()
10
    with jasp('molecules/wgs/CO2') as calc:
11
         CO2 = calc.get_atoms()
12
        E_CO2 = CO2.get_potential_energy()
13
14
    with jasp('molecules/wgs/H2') as calc:
15
        H2 = calc.get_atoms()
16
17
        E_H2 = H2.get_potential_energy()
18
    with jasp('molecules/wgs/H20') as calc:
19
20
        H20 = calc.get_atoms()
        E_H20 = H20.get_potential_energy()
21
22
    # now we get the vibrational energies
23
    h = 4.1356675e-15 \# eV*s
24
    c = 3.0e10 \# cm/s
25
26
27
    with jasp('molecules/wgs/CO-vib') as calc:
         vib_freq = calc.get_vibrational_frequencies()
28
        CO_vib_energies = [h*c*nu for nu in vib_freq]
29
30
    with jasp('molecules/wgs/CO2-vib') as calc:
31
32
         vib_freq = calc.get_vibrational_frequencies()
        CO2_vib_energies = [h*c*nu for nu in vib_freq]
33
    with jasp('molecules/wgs/H2-vib') as calc:
35
36
         vib_freq = calc.get_vibrational_frequencies()
        H2_vib_energies = [h*c*nu for nu in vib_freq]
37
38
    with jasp('molecules/wgs/H2O-vib') as calc:
39
         vib_freq = calc.get_vibrational_frequencies()
40
        H2O_vib_energies = [h*c*nu for nu in vib_freq]
41
42
    # now we make a thermo object for each molecule
43
44
    {\tt CO\_t = IdealGasThermo(vib\_energies=CO\_vib\_energies[0:0],}
                           electronicenergy=E_CO, atoms=CO,
45
                           geometry='linear', symmetrynumber=1,
46
                           spin=0)
47
48
    CO2_t = IdealGasThermo(vib_energies=CO2_vib_energies[0:4],
49
                           electronicenergy=E_CO2, atoms=CO2,
50
                           geometry='linear', symmetrynumber=2,
51
                           spin=0)
52
```

```
53
54
    H2_t = IdealGasThermo(vib_energies=H2_vib_energies[0:0],
55
                           electronicenergy=E_H2, atoms=H2,
                           geometry='linear', symmetrynumber=2,
56
57
                           spin=0)
58
    H2O_t = IdealGasThermo(vib_energies=H2O_vib_energies[0:3],
59
60
                           electronicenergy=E_H2O, atoms=H2O,
                           geometry='nonlinear', symmetrynumber=2,
61
62
                           spin=0)
63
64
    # now we can compute G_rxn for a range of temperatures from 200 to 1000 K
    Trange = np.linspace(200,1000,20) #K
65
66
    P = 101325. \# Pa
67
    Grxn = np.array([(CO2_t.get_free_energy(temperature=T, pressure=P)
                       + H2_t.get_free_energy(temperature=T, pressure=P)
68
69
                       - H2O_t.get_free_energy(temperature=T, pressure=P)
                       - CO_t.get_free_energy(temperature=T, pressure=P))*96.485 for T in Trange])
70
71
    Hrxn = np.array([(CO2_t.get_enthalpy(temperature=T)
72
73
                       + H2_t.get_enthalpy(temperature=T)
                       - H2O_t.get_enthalpy(temperature=T)
74
                       - CO_t.get_enthalpy(temperature=T))*96.485 for T in Trange])
75
76
    plt.plot(Trange, Grxn, 'bo-',label='$\Delta G_{rxn}$')
77
    plt.plot(Trange, Hrxn, 'ro:',label='$\Delta H_{rxn}$')
78
    plt.xlabel('Temperature (K)')
79
    plt.ylabel('$\Delta G_{rxn}$ (kJ/mol)')
80
    plt.legend(loc='best')
    plt.savefig('images/wgs-dG-T.png')
82
83
84
    plt.figure()
85
    R = 8.314e-3 \# gas constant in kJ/mol/K
86
87
    Keq = np.exp(-Grxn/R/Trange)
    plt.plot(Trange, Keq)
88
    plt.ylim([0, 100])
89
    plt.xlabel('Temperature (K)')
90
    plt.ylabel('$K_{eq}$')
91
    plt.savefig('images/wgs-Keq.png')
92
    plt.show()
```

None

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.

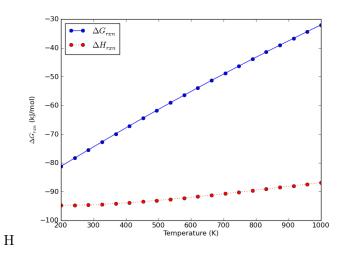
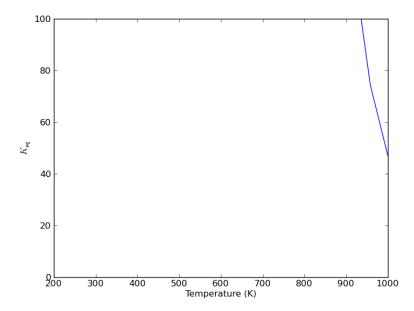


Figure 16: Thermodynamic energies of the water gas shift reaction as a function of temperature.

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 NH₃ inversion. We have to create an NH₃ 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
1
2 from ase import Atoms
3 from ase.data.molecules import molecule
   import numpy as np
    from jasp import *
5
    from ase.constraints import FixAtoms
    atoms = molecule('NH3')
8
    constraint = FixAtoms(mask=[atom.symbol == 'N' for atom in atoms])
9
10
    atoms.set_constraint(constraint)
11
    Npos = atoms.positions[0]
12
13
    # move N to origin
14
    atoms.translate(-Npos)
15
16
    atoms.set_cell((10,10,10),scale_atoms=False)
17
    atoms2 = atoms.copy()
18
19
    pos2 = atoms2.positions
20
^{21}
    for i,atom in enumerate(atoms2):
        if atom.symbol == 'H':
22
23
            # reflect through z
            pos2[i] *= np.array([1,1,-1])
24
25
    atoms2.positions = pos2
26
27
    #now move N to center of box
28
    atoms.translate([5,5,5])
    atoms2.translate([5,5,5])
29
30
    with jasp('molecules/nh3-initial',
31
32
             xc='PBE',
33
              encut=350,
              ibrion=1,
34
              nsw=10,
```

```
atoms=atoms) as calc:
36
37
         try:
38
             calc.calculate()
         except (VaspSubmitted, VaspQueued):
39
40
             pass
41
42
    with jasp('molecules/nh3-final',
               xc='PBE',
43
               encut=350,
44
               ibrion=1,
45
               nsw=10,
46
47
               atoms=atoms2) as calc:
         try:
48
49
             calc.calculate()
         except (VaspSubmitted, VaspQueued):
50
51
```

None



4.9.2 Run band calculation

Now we do the band calculation.

```
# Run NH3 NEB calculations
1
2
    from jasp import *
    from ase.neb import NEB
3
    with jasp('molecules/nh3-initial') as calc:
5
6
         atoms = calc.get_atoms()
7
8
    with jasp('molecules/nh3-final') as calc:
        atoms2 = calc.get_atoms()
9
10
    # 5 images including endpoints
11
    images = [atoms]
12
13
    images += [atoms.copy() for i in range(3)]
    images += [atoms2]
14
15
    neb = NEB(images)
16
    neb.interpolate()
17
18
    with jasp('molecules/nh3-neb',
19
20
               xc='PBE',
^{21}
               ibrion=1,
               nsw=90,
22
23
               spring=-5, debug=logging.DEBUG,
               atoms=images) as calc:
24
25
         images,energies = calc.get_neb()
26
27
         calc.plot_neb(show=False)
28
    import matplotlib.pyplot as plt
    plt.savefig('images/nh3-neb.png')
29
```

None

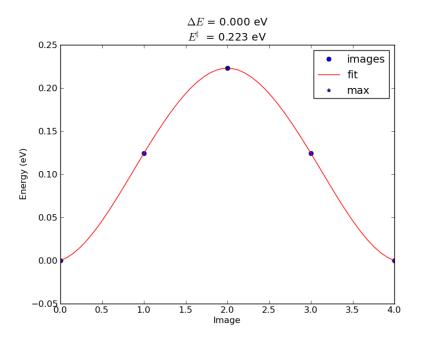


Figure 17: NEB for ammonia flipping.

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 *
4
5
    with jasp('molecules/nh3-neb') as calc:
6
        images,energies = calc.get_neb()
9
    # this rotates the atoms 90 degrees about the y-axis
10
    [atoms.rotate('y', np.pi/2.) for atoms in images]
11
    for i,atoms in enumerate(images):
12
        write('images/00{0}-nh3.png'.format(i), atoms, show_unit_cell=2)
13
14
15
    os.system('convert -delay 50 -loop 0 images/00*-nh3.png nh3-neb.gif')
16
17
    # Shockwave flash
18
    os.system('png2swf -o nh3-neb.swf images/00*-nh3.png ')
19
```

4.10 Simulated infrared spectra

At http://homepage.univie.ac.at/david.karhanek/downloads.html#Entry02 there is a recipe for computing the Infrared vibrational spectroscopy is sities in VASP. We are going to do that for water here. We instruct Vary 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 with IBRION=5 or 6. Remember we already relaxed the water molecule in a previous section (Relaxation of a water molecule). We will reuse that calculation here.



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

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

```
1
    # 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
5
6
    printf "..reading OUTCAR"
    BORN_NROWS='grep NIONS OUTCAR | awk '{print $12*4+1}'
    if [ 'grep 'BORN' OUTCAR | wc -1' = 0 ]; then \
8
       printf " .. FAILED! Born effective charges missing! Bye! \n\n" ; exit 1 ; fi
9
10
    grep "in e, cummulative" -A $BORN_NROWS OUTCAR > born.txt
11
    # extract Eigenvectors and eigenvalues
12
    if [ 'grep 'SQRT(mass)' OUTCAR | wc -1' != 1 ]; then \
13
       printf " .. FAILED! Restart VASP with NWRITE=3! Bye! \n\n"; exit 1; fi
```

```
15 EIG_NVIBS='grep -A 2000 'SQRT(mass)' OUTCAR | grep 'cm-1' | wc -1'
    EIG_NIONS='grep NIONS OUTCAR | awk '{print $12}'
16
17
    EIG_NROWS='echo "($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
18
19
    printf " ..done\n"
20
    # set up a new directory, split files - prepare for parsing
21
    printf "..splitting files"
22
   mkdir intensities; mv born.txt eigenvectors.txt intensities/
23
24
   cd intensities/
    let NBORN_NROWS=BORN_NROWS-1
25
26
    let NEIG_NROWS=EIG_NROWS-3
    let NBORN_STEP=4
27
   let NEIG_STEP=EIG_NIONS+3
29
   tail -n $NBORN_NROWS born.txt > temp.born.txt
30
    tail -n $NEIG_NROWS eigenvectors.txt > temp.eige.txt
31
    mkdir inputs; mv born.txt eigenvectors.txt inputs/
    split -a 3 -d -l $NEIG_STEP temp.eige.txt temp.ei.
32
33 split -a 3 -d -l $NBORN_STEP temp.born.txt temp.bo.
34
    mkdir temps01; mv temp.born.txt temp.eige.txt temps01/
35
    for nu in 'seq 1 $EIG_NVIBS'; do
    let nud=nu-1; ei='printf "%03u" $nu'; eid='printf "%03u" $nud'; mv temp.ei.$eid eigens.vib.$ei
36
    done
37
   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.$bo
39
40
    printf " ..done\n"
41
42
43
   # parse deviation vectors (eig)
    printf "..parsing eigenvectors"
44
    let sad=$EIG_NIONS+1
45
    for nu in 'seq 1 $EIG_NVIBS'; do
46
47
     nuu='printf "%03u" $nu'
     tail -n $sad eigens.vib.$nuu | head -n $EIG_NIONS | awk '{print $4,$5,$6}' > e.vib.$nuu.allions
48
49
     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
51
52
53
    done
    printf " ..done\n"
54
55
   # parse born effective charge matrices (born)
56
    printf "..parsing eff.charges"
57
58
   for s in 'seq 1 $EIG_NIONS'; do
    ss='printf "%03u" $s'
59
     awk '{print $2,$3,$4}' borncs.$ss | tail -3 > bornch.$ss
60
61
    mkdir temps02; mv eigens.* borncs.* temps02/
63
    printf " ..done\n"
64
    # parse matrices, multiply them and collect squares (giving intensities)
65
    printf "..multiplying matrices, summing "
66
67
   for nu in 'seq 1 $EIG_NVIBS'; do
    nuu='printf "%03u" $nu'
68
     int=0.0
70
     for alpha in 1 2 3; do
                                         # summing over alpha coordinates
```

```
sumpol=0.0
71
       for s in 'seq 1 $EIG_NIONS'; do # summing over atoms
72
       ss='printf "%03u" $s'
73
       awk -v a="$alpha" '(NR==a){print}' bornch.$ss > z.ion.$ss.alpha.$alpha
74
75
         \textit{\# summing over beta coordinates and multiplying } \textit{Z(s,alpha)**e(s)} \textit{ done by the following awk script } 
       paste z.ion.$ss.alpha.$alpha e.vib.$nuu.ion.$ss | \
76
       awk '{pol=$1*$4+$2*$5+$3*$6; print $0," ",pol}' > matr-vib-${nuu}-alpha-${alpha}-ion-${ss}
77
78
       sumpol='cat matr-vib-${nuu}-alpha-${alpha}-ion-* | awk '{sum+=$7} END {print sum}''
79
80
      int='echo "$int+($sumpol)^2" | sed 's/[eE]/*10^/g' | bc -1'
81
82
      freq='awk '(NR==1){print $8}' temps02/eigens.vib.$nuu'
     echo "$nuu $freq $int">> exact.res.txt
83
84
     printf "."
85
    done
    printf " ..done\n"
86
   # format results, normalize intensities
88
89 printf "..normalizing intensities"
90
    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,$3/max}' exact.res.txt > results.txt
91
    printf " ..done\n"
92
93
    # clean up, display results
95
    printf "..finalizing:\n"
    mkdir temps03; mv bornch.* e.vib.*.allions temps03/
96
97
    mkdir temps04; mv z.ion* e.vib.*.ion.* temps04/
98 mkdir temps05; mv matr-* temps05/
99 mkdir results; mv *res*txt results/
    let NMATRIX=$EIG_NVIBS**2
100
    printf "%5u atoms found\n%5u vibrations found\n%5u matrices evaluated" \
           $EIG_NIONS $EIG_NVIBS $NMATRIX > results/statistics.txt
102
103
       # fast switch to clean up all temporary files
104
      rm -r temps*
105
    cat results/results.txt
     ..reading OUTCAR ..done
     ..splitting files ..done
     ..parsing eigenvectors ..done
     ..parsing eff.charges ..done
     ..multiplying matrices, summing ........done
     ..normalizing intensities ..done
     ..finalizing:
     001 3827.3 0.227
     002 3713.0 0.006
     003 1587.2 0.312
     004 235.5 1.000
     005
            19.1 0.006
     006
             2.3 0.000
     007
            16.6 0.005
```

```
008 45.0 0.000
009 136.1 0.345
```

Note the results above include the rotational and vibrational modes (modes 4-9). The following shell script (also from http://homepage.univie.ac.at/david.karhanek/downloremoves 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
    \label{local_max} \verb|max='awk''(NR==1){max=$3} $3>=max {max=$3} END {print max}' temp.reform.res.txt'| \\
    awk -v max="$max" '{print $1,$2,$3/max}' temp.reform.res.txt > exact.reform.res.txt
8
    awk -v max="$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"
10
    rm temp.reform.res.txt
11
12
    cat reform.res.txt
      ..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.

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

5 Bulk systems

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

5.1 Defining and visualizing bulk systems

5.1.1 Built-in functions in ase

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



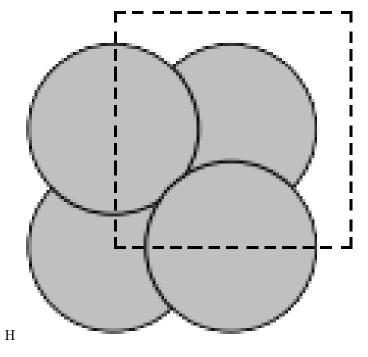


Figure 18: A simple fcc Ag bulk structure.

```
from ase.io import write
    from ase.lattice.cubic import FaceCenteredCubic
2
    atoms = FaceCenteredCubic(directions=[[1,0,0],
                                              [0,1,0],
                                              [0,0,1]],
                                              size=(1,1,1),
                                              symbol='Ag',
9
                                              latticeconstant=4.0)
10
11
12
    write('images/Ag-bulk.png',atoms,show_unit_cell=2)
13
    # to make an alloy, we can replace one atom with another kind atoms[0].symbol = 'Pd' \,
14
15
    write('images/AgPd-bulk.png',atoms,show_unit_cell=2)
16
```

 ${\tt None}$

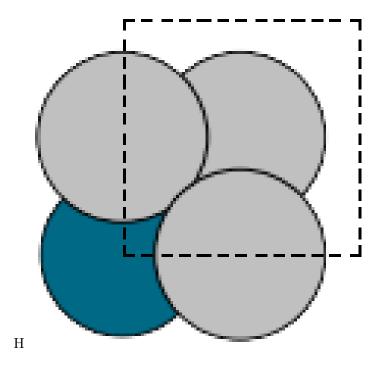


Figure 19: A simple Ag\$_3\$Pd bulk structure.

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
1
   2
                  Crystallographic Information Format file
3
                  Produced by PyCifRW module
4
5
   #
   #
      This is a CIF file. CIF has been adopted by the International
6
   #
      Union of Crystallography as the standard for data archiving and
   #
8
      transmission.
10
   # For information on this file format, follow the CIF links at
   # http://www.iucr.org
11
   12
13
14
   data_Ru02
                                       'P 1'
15
   _symmetry_space_group_name_H-M
16
    _cell_length_a
                                       3.13970109
17
    _cell_length_b
                                       4.5436378
   _cell_length_c
                                       4.5436378
18
   _cell_angle_alpha
                                       90.0
20
   _cell_angle_beta
                                       90.0
21
    _cell_angle_gamma
                                       90.0
22
    _chemical_name_systematic
                                       'Generated by pymatgen'
   _symmetry_Int_Tables_number
23
   _chemical_formula_structural
                                       Ru02
                                       'Ru2 04'
   _chemical_formula_sum
25
    _cell_volume
                                       64.8180127062
26
27
    _cell_formula_units_Z
28
29
     _symmetry_equiv_pos_site_id
     _symmetry_equiv_pos_as_xyz
30
31
      1 'x, y, z'
32
33
34
     _atom_site_type_symbol
     _atom_site_label
35
      _atom_site_symmetry_multiplicity
36
      _atom_site_fract_x
37
38
     _atom_site_fract_y
39
      _atom_site_fract_z
```

```
40
     _atom_site_attached_hydrogens
41
      _atom_site_B_iso_or_equiv
42
      _atom_site_occupancy
      0 01 1 0.000000 0.694330 0.694330 0 . 1
43
44
      0 02 1 0.500000 0.805670 0.194330 0 . 1
      0 03 1 0.000000 0.305670 0.305670 0 . 1
45
      0 04 1 0.500000 0.194330 0.805670 0
46
      Ru Ru5 1 0.500000 0.500000 0.500000 0 . 1
47
      Ru Ru6 1 0.000000 0.000000 0.000000 0
48
```

We can read this file in with func:ase.io.read

```
from ase.io import read, write

atoms = read('bulk/Ru204_1.cif')

write('images/Ru204.png', atoms, show_unit_cell=2)
```

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 *
   JASPRC['mode'] = 'run'
2
  with jasp('bulk/Ru204',
           xc='PBE',
           setups={'Ru':'_pv'}) as calc:
6
      atoms = calc.get_atoms()
7
8
      atoms.get_potential_energy()
      print calc
     VASP calculation from /home/jkitchin/dft-org/bulk/Ru204
     converged: True
     Energy = -44.302895 eV
     Unit cell vectors (angstroms)
                                   length
                           Z
                    У
     a0 [ 3.137 0.000 0.000] 3.137
     a1 [ 0.000 4.541 0.000] 4.541
     a2 [ 0.000 0.000 4.541] 4.541
     a,b,c,alpha,beta,gamma (deg): 3.137 4.541 4.541 90.0 90.0 90.0
     Unit cell volume = 64.694 Ang^3
     Stress (GPa):xx,
                          уу,
                                  zz,
                                         yz,
                                                 ΧZ,
                                                         хy
```

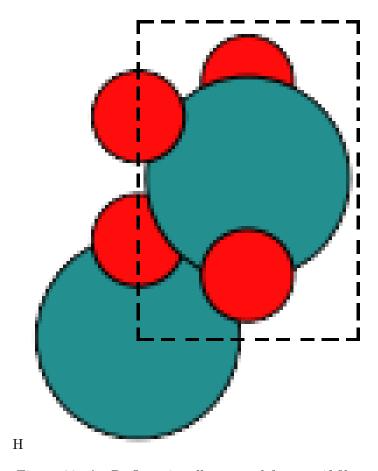


Figure 20: An RuO_2 unit cell prepared from a cif file.

```
-0.000 -0.001 -0.001 -0.000 -0.000 -0.000
                                        tag rmsForce
                position [x,y,z]
Atom#
      sym
                      2.270
                                             0.00
 0
      Ru [1.569
                                2.270]
                                         0
  1
      Ru [0.000
                      0.000
                                0.000]
                                         0.00
  2
         [0.000
                                3.153]
                                         0.00
      0
                      3.153
  3
                                         0.00
      0
          [1.569
                      3.659
                                0.882]
  4
          [0.000
                      1.388
                                1.388]
                                             0.00
  5
          [1.569
                      0.882
                                3.659]
                                             0.00
```

INCAR Parameters:

nbands: 31
ismear: 1
 nelm: 100
nelmin: 3
icharg: 1
 nsw: 99
ibrion: 2
 npar: 1
 isif: 3
 encut: 520.0
 sigma: 0.2
 ediff: 0.0003
lwave: True

magmom: [0.59999999999999, 0.599999999999, 0.5999999999999, 0.59999999999, 0.599

prec: Accurate
algo: Fast
lreal: Auto
kpts: [8 6 6]
reciprocal: False
xc: PBE
txt: gamma: False

Pseudopotentials used:

Ru: potpaw_PBE/Ru/POTCAR (git-hash: dee616f2a1e7a5430bb588f1710bfea3001d54ea)
0: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)

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 TiO₂?

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 [20] (Chapter 1).

```
from jasp import *
1
2
    from ase import Atom, Atoms
3
4
    LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
5
6
    fcc_energies = []
7
    for a in LC:
        atoms = Atoms([Atom('Cu',(0,0,0))],
8
                   cell=0.5*a*np.array([[1.0, 1.0, 0.0],
9
10
                                         [0.0, 1.0, 1.0],
                                         [1.0, 0.0, 1.0]]))
11
12
```

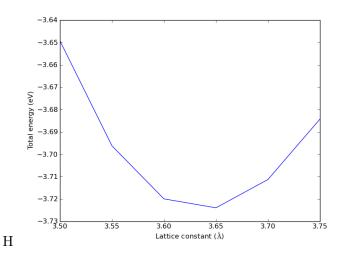


Figure 21: Total energy vs. fcc lattice contant for Cu. It appears the minimum is near 3.65 Å.

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

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 *
1
    from ase.utils.eos import *
2
    LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
    energies = []
    volumes = []
6
    for a in LC:
        with jasp('bulk/Cu-{0}'.format(a)) as calc:
7
            atoms = calc.get_atoms()
8
            volumes.append(atoms.get_volume())
9
10
            energies.append(atoms.get_potential_energy())
11
```

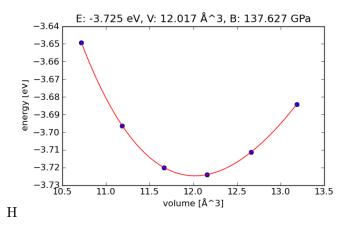


Figure 22: Total energy vs. volume for fcc Cu with fitted cubic polynomial equation of state.

```
12  eos = EquationOfState(volumes,energies)
13  v0, e0, B = eos.fit()
14
15  print v0, e0, B
16
17  eos.plot('images/Cu-fcc-eos.png')
```

12.0167187475 -3.72468102446 0.858998381274

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

```
1
    from ase import Atom, Atoms
2
    a = 3.61 # lattice constant
3
    atoms = Atoms([Atom('Cu', [0,0,0])],
5
6
                   cell=0.5*a*np.array([[1.0, 1.0, -1.0],
                                         [-1.0, 1.0, 1.0],
7
8
                                         [1.0, -1.0, 1.0]))
9
    print a*(11.8/atoms.get_volume())**(1./3.)
10
```

2.86838428403

Now we run the equation of state calculations.

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

Finally, we will compare the two crystal structures.

```
from jasp import *
1
2
    bcc_LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
    bcc_volumes = []
4
5
    bcc_energies = []
    for a in bcc_LC:
6
        with jasp('bulk/Cu-bcc-{0}'.format(a)) as calc:
            atoms = calc.get_atoms()
            bcc_volumes.append(atoms.get_volume())
9
            bcc_energies.append(atoms.get_potential_energy())
10
11
    # fcc
12
    fcc_LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
    fcc_volumes = []
14
    fcc_energies =[]
15
    for a in fcc_LC:
16
        with jasp('bulk/Cu-{0}'.format(a)) as calc:
17
18
            atoms = calc.get_atoms()
            fcc_volumes.append(atoms.get_volume())
19
20
            fcc_energies.append(atoms.get_potential_energy())
21
    import matplotlib.pyplot as plt
22
23
    plt.plot(fcc_volumes, fcc_energies, label='fcc')
    plt.plot(bcc_volumes, bcc_energies,label='bcc')
24
26
    plt.xlabel('Atomic volume ($\AA^3$/atom)')
   plt.ylabel('Total energy (eV)')
27
    plt.legend()
```

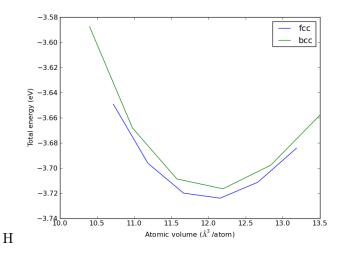


Figure 23: Comparison of energies between fcc and bcc Cu. The fcc structure is lower in energy.

```
29 plt.savefig('images/Cu-bcc-fcc.png')
30 plt.show()
```

None

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

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

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

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 24.

```
{\tt from \ ase.lattice.hexagonal \ import \ Hexagonal Closed Packed}
1
2
    from jasp import *
3
    import matplotlib.pyplot as plt
    atoms = HexagonalClosedPacked(symbol='Ru',
                                    latticeconstant={'a':2.7, 'c/a':1.584})
6
    a_list = [2.5, 2.6, 2.7, 2.8, 2.9]
8
    covera_list = [1.4, 1.5, 1.6, 1.7, 1.8]
9
10
    for a in a_list:
11
12
        energies = []
        for covera in covera_list:
13
14
             atoms = HexagonalClosedPacked(symbol='Ru',
15
                                    latticeconstant={'a':a, 'c/a':covera})
16
17
             wd = 'bulk/Ru/{0:1.2f}-{1:1.2f}'.format(a,covera)
18
19
             with jasp(wd,
20
                       xc='PBE',
21
22
                       kpts=(6,6,4),
                       encut=350,
23
                       atoms=atoms) as calc:
25
                     energies.append(atoms.get_potential_energy())
26
                 except (VaspSubmitted, VaspQueued):
27
                     pass
28
        plt.plot(covera_list, energies, label='a={0}'.format(a))
29
30
31
    plt.xlabel('$c/a$ ($\AA$)')
    plt.ylabel('Energy (eV)')
32
    plt.legend()
33
    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 25).

```
from jasp import *
1
    import matplotlib.pyplot as plt
    from pylab import *
3
    x = [2.5, 2.6, 2.7, 2.8, 2.9]
    y = [1.4, 1.5, 1.6, 1.7, 1.8]
6
    X,Y = meshgrid(x,y)
8
    Z = np.zeros(X.shape)
9
10
11
    for i,a in enumerate(x):
12
         for j,covera in enumerate(y):
13
             wd = \frac{bulk}{Ru}/{0:1.2f}-{1:1.2f}, format(a, covera)
14
15
```

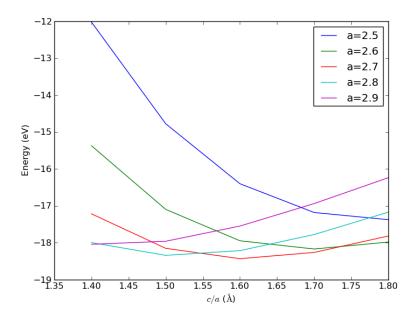


Figure 24: Total energy vs. c/a for different values of a.

```
with jasp(wd) as calc:
16
                 atoms = calc.get_atoms()
17
18
                     Z[i][j] = atoms.get_potential_energy()
19
20
                 except (VaspSubmitted, VaspQueued):
^{21}
                     pass
22
    cf = plt.contourf(X,Y,Z,20,
23
                       cmap=cm.jet)
24
25
    cbar = colorbar(cf)
26
27
    cbar.ax.set_ylabel('Energy (eV)')
28
    plt.xlabel('$a$ ($\AA$)')
29
30
    plt.ylabel('$c/a$')
31
32
    plt.legend()
    plt.savefig('images/ru-contourf.png')
33
    plt.show()
```

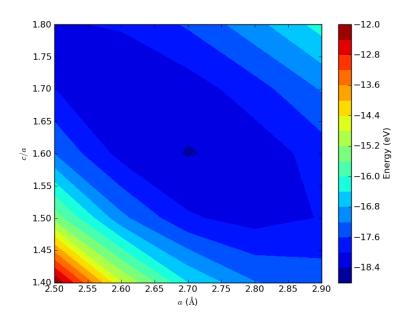


Figure 25: Contour plot of the total energy of hcp Ru for different values of a and c/a.

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 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 ${\rm TiO_2}$ which has two lattice parameters and one internal degree of freedom.

```
# step 1 frozen atoms and shape at different volumes
1
   from ase import *
   import numpy as np
   from jasp import *
4
5
    import matplotlib.pyplot as plt
6
8
    create a TiO2 structure from the lattice vectors at
9
    http://cst-www.nrl.navy.mil/lattice/struk/c4.html
10
    a = 4.59 # experimental degrees of freedom.
11
    c = 2.96
12
   u = 0.3 #internal degree of freedom!
13
14
    #primitive vectors
15
    a1 = a*np.array([1.0, 0.0, 0.0])
16
    a2 = a*np.array([0.0, 1.0, 0.0])
17
18
    a3 = c*np.array([0.0, 0.0, 1.0])
19
    atoms = Atoms([Atom('Ti',[0., 0., 0.]),
20
                    Atom('Ti', 0.5*a1 + 0.5*a2 + 0.5*a3),
21
                    Atom('0',u*a1+u*a2),
                    Atom('0',-u*a1 -u*a2),
23
24
                    Atom(^{,0}, (0.5+u)*a1 + (0.5-u)*a2 + 0.5*a3),
                    Atom('0',(0.5-u)*a1 + (0.5+u)*a2 + 0.5*a3)],
25
                   cell=[a1,a2,a3])
26
27
    v0 = atoms.get_volume()
28
    cell0 = atoms.get_cell()
29
30
    factors = [0.9, 0.95, 1.0, 1.05, 1.1] #to change volume by
31
32
    energies, volumes = [], []
33
34
    for f in factors:
        v1 = f*v0
35
        cell_factor = (v1/v0)**(1./3.)
36
37
        atoms.set_cell(cell0*cell_factor, scale_atoms=True)
38
39
        with jasp('bulk/tio2/step1-{0:1.2f}'.format(f),
40
                   encut=350,
41
42
                   kpts=(4,4,4),
43
                   xc='PBE',
                   sigma=0.05,
                   atoms=atoms) as calc:
45
46
                 energies.append(atoms.get_potential_energy())
47
```

```
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')
```

[-52.04219299999997, -52.649554999999999, -52.94985299999997, -52.950336, -52.7834

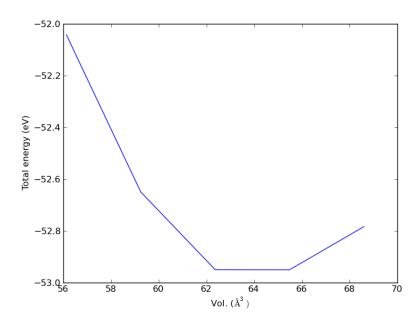


Figure 26: Total energy vs volume for TIO_2 with frozen atoms and frozen shape.

In step 2 we will actually do some relaxations. We will do a series of relaxations, from 62 to 66 ${\rm \AA}^3$, 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()
```

```
7
         v0 = atoms.get_volume()
         cell0 = atoms.get_cell()
8
9
    volumes = [62., 63., 64., 65., 66.]
10
11
    energies = []
12
13
    for v in volumes:
         atoms.set_cell((v/v0)**(1./3.)*cell0, scale_atoms=True)
14
         with jasp('bulk/tio2/step2-{0}'.format(v),
15
16
                    encut=350,
                    kpts=(4,4,4),
17
                    xc='PBE',
18
                    sigma=0.05,
19
20
                    isif=4,
                                 # relax ions, cell shape. not volume
^{21}
                    ibrion=2,
                    nsw=10,
22
23
                    atoms=atoms) as calc:
24
             try:
                 energies.append(atoms.get_potential_energy())
25
             {\tt except} \ ({\tt VaspSubmitted}, \ {\tt VaspQueued}):
26
27
                 pass
28
    plt.plot(volumes, energies)
29
   plt.xlabel('Vol. ($\AA^3)$')
    plt.ylabel('Total energy (eV)')
31
    plt.savefig('images/tio2-step2.png')
```

None

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

```
1
    # step 3 relax whole unit cell
    from jasp import *
2
3
    with jasp('bulk/tio2/step2-64.0') as calc:
4
        atoms = calc.get_atoms()
5
6
    with jasp('bulk/tio2/step3',
7
               encut=350,
8
               kpts=(4,4,4),
9
               xc='PBE',
10
11
               sigma=0.05,
               isif=3,
                           # relax ions, cell shape, and volume
12
13
               ibrion=2,
               nsw=10,
14
               atoms=atoms) as calc:
15
16
                calc.calculate()
17
                 print calc
             except (VaspSubmitted, VaspQueued):
19
20
                 pass
21
```

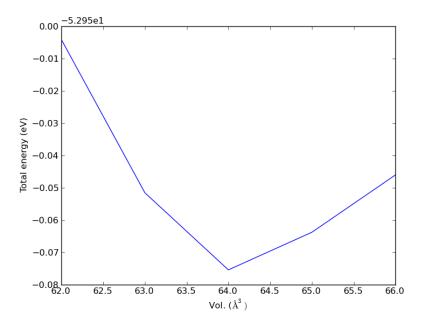


Figure 27: Total energy vs. volume of ${\rm TiO_2}$ where shape and atoms are allowed to relax.

```
from pyspglib import spglib
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)
                              length
        х
                 У
                       z
  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,
                             zz,
                      уу,
                                     yz,
                                            ΧZ,
                                                   хy
            -0.001 -0.001 0.000 -0.000 -0.000 -0.000
                   position [x,y,z]
                                             tag rmsForce
 Atom# sym
```

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

0: 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 $0.0~{\rm eV/\AA}$).
- 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, semi-conductors 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 *
1
    from ase import Atom. Atoms
2
3
    import matplotlib.pyplot as plt
4
5
    a = 3.61
    atoms = Atoms([Atom('Cu',(0,0,0))],
6
                   cell=0.5*a*np.array([[1.0, 1.0, 0.0],
                                         [0.0, 1.0, 1.0],
8
                                         [1.0, 0.0, 1.0]])).repeat((2,2,2))
9
10
11
    SIGMA = [0.001, 0.05, 0.1, 0.2, 0.5]
12
    for sigma in SIGMA:
13
14
         with jasp('bulk/Cu-sigma-{0}'.format(sigma),
15
16
                   xc='PBE',
                   encut=350,
17
18
                   kpts=(4,4,4),
                   ismear=-1,
19
20
                   sigma=sigma,
21
                   nbands=9*8.
                   atoms=atoms) as calc:
22
```

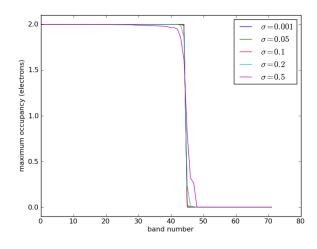


Figure 28: Effects of SIGMA on the occupancies of the Cu system.

```
23
             e = atoms.get_potential_energy()
24
            nbands = calc.nbands
25
26
             nkpts = len(calc.get_ibz_k_points())
27
28
             occ = np.zeros((nkpts, nbands))
             for i in range(nkpts):
29
30
                 occ[i,:] = calc.get_occupation_numbers(kpt=i)
31
            max_occ = np.max(occ,axis=0) #axis 0 is columns
32
33
            plt.plot(range(nbands), max_occ, label='$\sigma = {0}$'.format(sigma))
34
35
    plt.xlabel('band number')
36
    plt.ylabel('maximum occupancy (electrons)')
37
    plt.ylim([-0.1, 2.1])
38
    plt.legend(loc='best')
39
40
    plt.savefig('images/occ-sigma.png')
41
    plt.show()
```

5.4 Cohesive energy

Η

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

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

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 $6\times6\times6$ 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),
10
            xc='PBE',
11
12
            encut=350,
            kpts=(k,k,k),
13
14
            atoms=atoms) as calc:
          e = atoms.get_potential_energy()
15
16
       print '({0:2d}, {0:2d}, {0:2d}): total energy = {1} eV'.format(k,e)
17
           3, 3): total energy = -5.891945 eV
     (4, 4, 4): total energy = -7.244046 eV
           6, 6): total energy = -7.265141 \text{ eV}
           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
4
    atoms = FaceCenteredCubic(directions=[[0,1,1],
5
                                            [1,0,1],
                                            [1,1,0]],
                                            size=(1,1,1),
8
9
                                            symbol='Rh')
10
    with jasp('bulk/bulk-rh',
11
12
              xc='PBE',
              encut=350,
13
14
              kpts=(4,4,4),
              isif=3,
15
16
              ibrion=2,
^{17}
              nsw=10,
              atoms=atoms) as calc:
18
19
        bulk_energy = atoms.get_potential_energy()
20
    atoms = Atoms([Atom('Rh',[5,5,5],magmom=1)],
22
                   cell=(7,8,9))
23
24
   with jasp('bulk/atomic-rh-sp',
25
26
              xc='PBE',
              encut=350,
27
              kpts=(1,1,1),
28
               ispin=2,
29
```

```
30     atoms=atoms) as calc:
31     atomic_energy = atoms.get_potential_energy()
32
33     cohesive_energy = atomic_energy - bulk_energy
34     print 'The cohesive energy is {0:1.3f} eV'.format(cohesive_energy)
```

The cohesive energy is 5.887 eV

Again, the value in Kittel [20] 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 $E_{anatase}(V)$.

```
1
    # run the rutile calculations
    from jasp import *
    from ase import Atom, Atoms
3
    B='Ti'; X='0'; a=4.59; c=2.958; u=0.305;
5
    create a rutile structure from the lattice vectors at
7
    http://cst-www.nrl.navy.mil/lattice/struk/c4.html
8
9
    spacegroup: 136 P4_2/mnm
10
11
    a1 = a*np.array([1.0, 0.0, 0.0])
12
    a2 = a*np.array([0.0, 1.0, 0.0])
13
    a3 = c*np.array([0.0, 0.0, 1.0])
14
15
    atoms = Atoms([Atom(B, [0., 0., 0.]),
16
                    Atom(B, 0.5*a1 + 0.5*a2 + 0.5*a3),
17
18
                    Atom(X, u*a1 + u*a2),
                    Atom(X, -u*a1 - u*a2),
19
                    Atom(X, (0.5+u)*a1 + (0.5-u)*a2 + 0.5*a3),
20
                    Atom(X, (0.5-u)*a1 + (0.5+u)*a2 + 0.5*a3)],
21
```

```
cell=[a1, a2, a3])
22
23
    nTiO2 = len(atoms)/3.
24
    v0 = atoms.get_volume()
25
26
    cell0 = atoms.get_cell()
27
28
    volumes = [28., 30., 32., 34., 36.] #vol of one TiO2
29
    for v in volumes:
30
        atoms.set_cell(cell0*((nTiO2*v/vO)**(1./3.)), scale_atoms=True)
31
32
        with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v),
33
                  encut=350,
34
35
                  kpts=(6,6,6),
                  xc='PBE',
36
37
                  ismear=0,
                  sigma=0.001,
38
                  isif=2,
39
                  ibrion=2,
40
                  nsw=20,
41
                  atoms=atoms) as calc:
42
43
            try:
                print atoms.get_potential_energy()
44
45
            except (VaspSubmitted, VaspQueued):
                pass
46
     -52.001084
     -52.752668
     -53.004621
     -52.894179
      -52.523686
   # run the anatase calculations
2 from jasp import *
    from ase import Atom, Atoms
3
    # http://cst-www.nrl.navy.mil/lattice/struk/c5.html
5
    B='Ti'; X='0'; a=3.7842; c=2*4.7573; z=0.0831;
6
8
    a1 = a*np.array([1.0, 0.0, 0.0])
    a2 = a*np.array([0.0, 1.0, 0.0])
9
    a3 = np.array([0.5*a, 0.5*a, 0.5*c])
10
11
    atoms = Atoms([Atom(B, -0.125*a1 + 0.625*a2 + 0.25*a3),
12
13
                   Atom(B, 0.125*a1 + 0.375*a2 + 0.75*a3),
                   Atom(X, -z*a1 + (0.25-z)*a2 + 2.*z*a3),
14
                   Atom(X, -(0.25+z)*a1 + (0.5-z)*a2 + (0.5+2*z)*a3),
15
                   Atom(X, z*a1 - (0.25 - z)*a2 + (1-2*z)*a3),
16
                   Atom(X, (0.25 + z)*a1 + (0.5 + z)*a2 + (0.5-2*z)*a3)],
17
18
                   cell=[a1,a2,a3])
19
nTiO2 = len(atoms)/3.
v0 = atoms.get_volume()
```

```
cell0 = atoms.get_cell()
22
23
    volumes = [30., 33., 35., 37., 39.] #vol of one TiO2
24
25
26
    for v in volumes:
        atoms.set_cell(cell0*((nTi02*v/v0)**(1./3.)), scale_atoms=True)
27
28
         with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v),
29
                   encut=350,
30
31
                   kpts=(6,6,6),
                   xc='PBE',
32
33
                   ismear=0,
                   sigma=0.001,
34
35
                   isif=2,
36
                   ibrion=2,
37
                   nsw=20,
38
                   atoms=atoms) as calc:
39
             trv:
                 print atoms.get_potential_energy()
40
             except (VaspSubmitted, VaspQueued):
41
42
```

-51.835014 -52.956683 -53.174844 -53.094443

-52.809584

Now we will fit cubic polynomials to the data.

```
# fit cubic polynomials to E(V) for rutile and anatase
1
    from jasp import *
    import matplotlib.pyplot as plt
3
    # anatase equation of stte
5
    volumes = [30., 33., 35., 37., 39.] #vol of one TiO2
6
    a_volumes, a_energies = [], []
    for v in volumes:
8
9
         with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v)) as calc:
10
            atoms = calc.get_atoms()
            nTiO2 = len(atoms)/3.0
11
            a_volumes.append(atoms.get_volume()/nTiO2)
12
            {\tt a\_energies.append(atoms.get\_potential\_energy()/nTiO2)}
13
14
    # rutile equation of state
15
    volumes = [28., 30., 32., 34., 36.] #vol of one TiO2
16
17
    r_volumes, r_energies = [], []
    for v in volumes:
18
19
         with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v)) as calc:
            atoms = calc.get_atoms()
20
            nTiO2 = len(atoms)/3.0
^{21}
            r_volumes.append(atoms.get_volume()/nTiO2)
22
```

```
r_energies.append(atoms.get_potential_energy()/nTiO2)
23
24
   # cubic polynomial fit to equation of state E(V) = pars*[V^3 V^2 V^1 V^0]
25
   apars = np.polyfit(a_volumes, a_energies, 3)
26
   rpars = np.polyfit(r_volumes, r_energies, 3)
27
28
   print 'E_{anatase}(V) = \{0\}*V^3 + \{1\}*V^2 + \{2\}*V + \{3\}'.format(*apars)
29
   print 'E_rutile(V) = {0}*V^3 + {1}*V^2 + {2}*V + {3}'.format(*rpars)
30
  print 'anatase epars: {0!r}'.format(apars)
31
32 print 'rutile epars: {0!r}'.format(rpars)
   # get pressure parameters P(V) = -dE/dV
33
34
   dapars = -np.polyder(apars)
   drpars = -np.polyder(rpars)
35
36
37
  print 'anatase ppars: {0!r}'.format(dapars)
   print 'rutile ppars: {0!r}'.format(drpars)
38
39
40 print
  print 'P_anatase(V) = \{0\}*V^2 + \{0\}*V + \{2\}'.format(*dapars)
41
   print 'P_rutile(V) = {0}*V^2 + {0}*V + {2}'.format(*drpars)
42
43
   vfit = np.linspace(28,40)
44
45
   # plot the equations of state
   plt.plot(a_volumes, a_energies,'bo ', label='Anatase')
47
   plt.plot(vfit, np.polyval(apars, vfit), 'b-')
48
49
50 plt.plot(r_volumes, r_energies,'gs', label='Rutile')
51 plt.plot(vfit, np.polyval(rpars, vfit), 'g-')
52
   plt.xlabel('Volume ($\AA^3$/f.u.)')
53
54 plt.ylabel('Total energy (eV/f.u.)')
55 plt.legend()
56 plt.xlim([25,40])
57 plt.ylim([-27, -26])
   plt.savefig('images/rutile-anatase-eos.png')
   E_{anatase}(V) = -0.00106049245571*V^3 + 0.130279403916*V^2 + -5.23520055177*V + 42.520055177*V
   4.25202869e+01])
   rutile epars: array([ -1.24680208e-03,
                                                  1.42966536e-01, -5.33239733e+00,
              3.85903670e+011)
   anatase ppars: array([ 3.18147737e-03, -2.60558808e-01, 5.23520055e+00])
   rutile ppars: array([ 3.74040625e-03, -2.85933071e-01, 5.33239733e+00])
   P_{anatase}(V) = 0.00318147736712*V^2 + 0.00318147736712*V + 5.23520055177
   P_{\text{rutile}}(V) = 0.00374040625*V^2 + 0.00374040625*V + 5.33239732738
```

To find the conditions where a phase transition occurs, we have to find

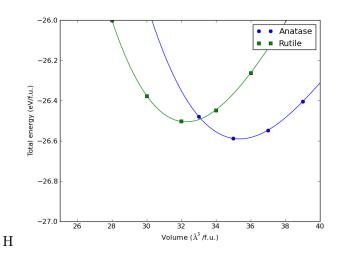


Figure 29: Equations of state (E(V)) for anatase and rutile TiO_2 .

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 func:scipy.optimize.fsolve to solve this problem.

```
from ase.units import GPa
    from numpy import array, linspace, polyval
3
    anatase_epars = array([ -1.06049246e-03,
                                           1.30279404e-01, -5.23520055e+00,
4
            4.25202869e+01])
5
    6
            3.85903670e+01])
8
9
    anatase_ppars = array([ 3.18147737e-03, -2.60558808e-01, 5.23520055e+00])
    rutile_ppars = array([ 3.74040625e-03, -2.85933071e-01, 5.33239733e+00])
10
11
    def func(V):
12
       V1 = V[0] # rutile volume
13
       V2 = V[1] # anatase volume
14
15
       E_rutile = polyval(rutile_epars,V1)
16
       E_anatase = polyval(anatase_epars, V2)
17
18
19
       P_rutile = polyval(rutile_ppars,V1)
       P_anatase = polyval(anatase_ppars, V2)
20
^{21}
       return [(E_anatase - E_rutile)/(V1-V2) - P_anatase,
22
```

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

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

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

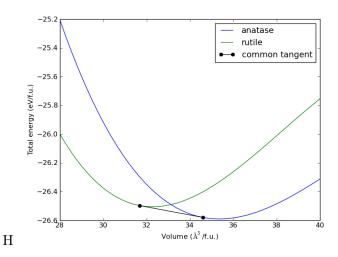


Figure 30: Illustration of the common tangent that shows the pressure where anatase and rutile coexist before anatase converts to rutile.

```
with jasp('bulk/alloy/cu',
11
               xc='PBE',
12
13
               encut=350,
               kpts=(13,13,13),
14
               nbands=9,
15
               ibrion=2,
16
17
               isif=4,
               nsw=10,
18
               atoms=atoms) as calc:
19
20
         cu = atoms.get_potential_energy()
21
    atoms = Atoms([Atom('Pd', [0.000,
                                              0.000,
                                                           0.000])],
22
                   cell=[[ 1.978, 0.000, 1.978],
23
                         [ 1.978, 1.978, 0.000],
24
                         [0.000, 1.978, 1.978]])
25
26
27
    with jasp('bulk/alloy/pd',
               xc='PBE',
28
               encut=350,
29
30
               kpts=(13,13,13),
               nbands=9,
31
32
               ibrion=2,
               isif=4,
33
               nsw=10,
34
               atoms=atoms) as calc:
35
        pd = atoms.get_potential_energy()
36
37
    print cu, pd
38
```

-3.723306 -5.183823

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 *
1
2
    from ase import Atom, Atoms
3
    atoms = Atoms([Atom('Cu', [0.000,
                                                0.000,
                                                             0.000]),
4
                    Atom('Pd', [-1.652,
                                                0.000,
                                                             2.039])],
5
                    cell= [[ 0.000, -2.039, 2.039],
6
7
                            [0.000, 2.039, 2.039],
                            [ -3.303, 0.000, 0.000]])
8
9
    with jasp('bulk/alloy/cupd-1',
10
               xc='PBE',
11
               encut=350,
12
13
               kpts=(12,12,8),
14
               nbands=17,
               ibrion=2,
15
               isif=4,
16
               nsw=10,
17
               atoms=atoms) as calc:
18
19
         cupd1 = atoms.get_potential_energy()
20
21
    atoms = Atoms([Atom('Cu', [-0.049,
                                                             0.049]),
22
                                               0.049.
                    Atom('Cu', [-11.170,
Atom('Pd', [-7.415,
                                              11.170,
                                                            11.170]),
23
24
                                               7.415,
                                                             7.415]).
                    Atom('Pd', [-3.804,
                                                3.804,
                                                             3.804])],
25
26
                    cell=[[-5.629, 3.701, 5.629],
                          [-3.701, 5.629, 5.629],
[-5.629, 5.629, 3.701]])
27
28
29
    with jasp('bulk/alloy/cupd-2',
30
31
               xc='PBE',
               encut=350.
32
33
               kpts=(8,8,8),
               nbands=34,
34
35
               ibrion=2,
36
               isif=4,
               nsw=10,
37
38
               atoms=atoms) as calc:
         cupd2 = atoms.get_potential_energy()
39
40
    print cupd1, cupd2
41
```

-9.147752 -17.97924

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

```
1
    from jasp import *
2
3
    with jasp('bulk/alloy/cu') as calc:
4
        atoms = calc.get_atoms()
        cu = atoms.get_potential_energy()/len(atoms)
5
    with jasp('bulk/alloy/pd') as calc:
7
        atoms = calc.get_atoms()
8
        pd = atoms.get_potential_energy()/len(atoms)
9
10
11
    with jasp('bulk/alloy/cupd-1') as calc:
        atoms = calc.get_atoms()
12
         e1 = atoms.get_potential_energy()
13
14
        for atom in atoms:
            if atom.symbol == 'Cu':
15
16
                e1 -= cu
            else:
17
18
                e1 -= pd
         e1 /= len(atoms)
19
20
    with jasp('bulk/alloy/cupd-2') as calc:
21
22
        atoms = calc.get_atoms()
23
         e2 = atoms.get_potential_energy()
        for atom in atoms:
24
            if atom.symbol == 'Cu':
25
                e2 -= cu
26
```

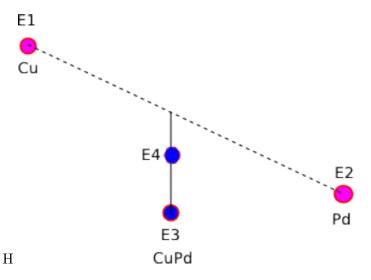


Figure 31: Conceptual picture of two alloys with exothermic formation energies.

```
27 else:
28 e2 -= pd
29 e2 /= len(atoms)
30
31 print 'Delta Hf cupd-1 = {0:1.2f} eV/atom'.format(e1)
32 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 31.

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:
9
10
         atoms = calc.get_atoms()
11
         pd = atoms.get_potential_energy()/len(atoms)
12
    atoms = Atoms([Atom('Cu', [-3.672,
                                                            3.672]),
13
                                               3.672.
                                                           0.000]),
                    Atom('Cu', [0.000,
                                              0.000.
14
                    Atom('Cu', [-10.821, Atom('Pd', [-7.246,
                                              10.821,
                                                           10.821])
15
                                                            7.246])],
16
                                               7.246.
                    cell=[[-5.464, 3.565, 5.464],
17
18
                           [-3.565, 5.464, 5.464],
                           [-5.464, 5.464, 3.565]])
19
20
    with jasp('bulk/alloy/cu3pd-1',
21
22
               xc='PBE',
23
               encut=350,
24
               kpts=(8,8,8),
25
               nbands=34,
               ibrion=2,
26
               isif=4,
27
               nsw=10,
28
29
               atoms=atoms) as calc:
30
         e3 = atoms.get_potential_energy()
         for atom in atoms:
31
             if atom.symbol == 'Cu':
33
                 e3 -= cu
34
             else:
                 e3 -= pd
35
         e3 /= len(atoms)
36
    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 \text{ eV/atom}, x_0 = 0, H_{f,cupd-1} = -0.12 \text{ eV/atom}, x_3 = 0.5. The composition weighted average at x_{Pd} = 0.25 is: H_f = H_{f,Cu} + \frac{x_0 - x}{x_0 - x_3} (H_{f,cupd-1} - H_{f,Cu})
```

```
1 x0 = 0.0; x3 = 0.5; x = 0.25;

2 Hf1 = 0.0; Hf3 = -0.12;

3 print Hf1 + (x0-x)/(x0-x3)*(Hf3 - Hf1)
```

-0.06

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

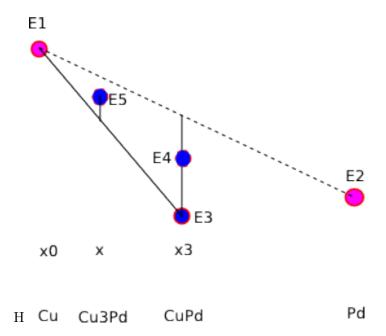


Figure 32: Illustration of phase separation.

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 32.

Finally, let us consider one more structure with the Cu3Pd stoichiometry.

```
from jasp import *
1
    from ase import Atom, Atoms
3
    # parent metals
4
    with jasp('bulk/alloy/cu') as calc:
5
        atoms = calc.get_atoms()
6
        cu = atoms.get_potential_energy()/len(atoms)
8
9
    with jasp('bulk/alloy/pd') as calc:
10
        atoms = calc.get_atoms()
        pd = atoms.get_potential_energy()/len(atoms)
11
12
                                                         0.000]),
    atoms = Atoms([Atom('Cu', [-1.867,
                                            1.867,
13
                   Atom('Cu', [0.000, Atom('Cu', [0.000,
14
                                            0.000,
                                                         0.000]),
                                                        1.867]),
                                            1.867,
15
                   Atom('Pd', [-1.867,
                                            0.000,
                                                        1.86])],
16
                   17
18
19
20
21
    with jasp('bulk/alloy/cu3pd-2',
              xc='PBE',
22
```

```
encut=350,
23
               kpts=(8,8,8),
24
25
               nbands=34,
               ibrion=2,
26
               isif=4,
27
               nsw=10.
28
29
               atoms=atoms) as calc:
30
         e4 = atoms.get_potential_energy()
         for atom in atoms:
31
             if atom.symbol == 'Cu':
32
                 e4 -= cu
33
34
             else:
                 e4 -= pd
35
         e4 /= len(atoms)
36
    print 'Delta Hf cu3pd-2 = {0:1.2f} eV/atom'.format(e4)
37
```

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.

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) [33, 34] 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.

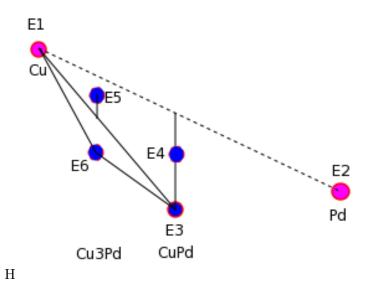


Figure 33: Illustration of phase separation.

Cu\$_2\$O calculation

```
# run Cu20 calculation
1
   from jasp import *
2
   from ase.visualize import *
   from ase import Atom, Atoms
   #http://phycomp.technion.ac.il/~ira/types.html#Cu20
6
   a = 4.27
7
8
   atoms = Atoms([Atom('Cu',[0,0,0]),
9
                 Atom('Cu',[0.5, 0.5, 0.0]),
10
11
                 Atom('Cu',[0.5, 0.0, 0.5]),
                 Atom('Cu',[0.0, 0.5, 0.5]),
12
                 Atom(^{,0}, [0.25, 0.25, 0.25]),
13
                 Atom('0',[0.75, 0.75, 0.75])])
14
15
16
   atoms.set_cell((a,a,a), scale_atoms=True)
17
   with jasp('bulk/Cu20',
18
19
            encut=400,
            kpts=(8,8,8),
20
^{21}
            ibrion=2,
            isif=3,
22
23
            nsw=30,
            xc='PBE',
24
25
            atoms=atoms) as calc:
26
       calc.set_nbands()
27
       print atoms.get_potential_energy()
       print calc
    -27.20459
      VASP calculation from /home/jkitchin/dft-org/bulk/Cu20
      converged: True
      Energy = -27.204590 \text{ eV}
      Unit cell vectors (angstroms)
                                       length
                              Z
                       У
      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,
                             уу,
                                     zz,
                                              yz,
                                                      ΧZ,
                                                               хy
                   0.000 0.000 0.000 -0.000 -0.000 -0.000
     Atom#
                         position [x,y,z]
                                                        tag rmsForce
             sym
             Cu [0.000
                                 0.000
                                              0.000]
                                                              0.00
                                                         0
```

```
Cu [2.135
                              0.000]
                                       0.00
                    2.135
2
    Cu [2.135
                   0.000
                              2.135]
                                       0
                                          0.00
    Cu [0.000
                                          0.00
3
                   2.135
                              2.135]
                                       0
4
        [1.067
                   1.067
                              1.067]
                                          0.00
    0
                                       0
5
        [3.202
                    3.202
                              3.202]
                                          0.00
```

INCAR Parameters:

nbands: 37
nsw: 30
ibrion: 2
isif: 3
encut: 400.0
prec: Normal
kpts: [8 8 8]
reciprocal: False
xc: PBE
txt: gamma: False

Pseudopotentials used:

0: 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
c = 5.1288
beta = 99.54/180*np.pi
y = 0.5819

4
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)])
```

```
18
   atoms = Atoms([Atom('Cu', 0.5*a2),
19
               Atom('Cu', 0.5*a1 + 0.5*a3),
20
               Atom('0', -y*a1 + y*a2 + 0.25*a3),
21
               Atom('0', y*a1 - y*a2 - 0.25*a3)], cell=(a1, a2, a3))
22
23
^{24}
25 with jasp('bulk/Cu0',
           encut=400,
26
27
           kpts=(8,8,8),
           ibrion=2,
28
29
           isif=3,
           nsw=30.
30
31
           xc='PBE',
           atoms=atoms) as calc:
32
33
      calc.set_nbands()
34
      print atoms.get_potential_energy()
      print calc
35
   -19.509498
    : ------
      VASP calculation from /home/jkitchin/dft-org/bulk/CuO
      converged: True
     Energy = -19.509498 \text{ eV}
     Unit cell vectors (angstroms)
                                   length
                     У
      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,
                           уу,
                                 ZZ,
                                          yΖ,
                                                  ΧZ,
                  0.008 0.013 0.009 -0.000 0.000 -0.000
     Atom# sym
                      position [x,y,z]
                                                   tag rmsForce
       0
            Cu [1.151
                              0.888
                                          0.023]
                                                        0.00
       1
            Cu [0.770
                             -0.888
                                          2.566]
                                                        0.00
                                                    0
       2
            0 [2.111
                            -0.168
                                          1.318]
                                                    0 0.03
       3
            0
                 [1.730
                             0.168
                                          3.861]
                                                    0
                                                        0.03
```

INCAR Parameters:

nbands: 23 nsw: 30

```
ibrion: 2
isif: 3
encut: 400.0
prec: Normal
kpts: [8 8 8]
reciprocal: False
xc: PBE
txt: -
gamma: False
```

Pseudopotentials used:

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

Reaction energy calculation

```
from jasp import *
1
    # don't forget to normalize your total energy to a formula unit. Cu20 has 3 atoms, so the number of formula units in
3
    with jasp('bulk/Cu20') as calc:
        atoms = calc.get_atoms()
5
        cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
6
    with jasp('bulk/CuO') as calc:
8
9
        atoms = calc.get_atoms()
        cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
10
11
    # make sure to use the same cutoff energy for the O2 molecule!
12
13
    with jasp('molecules/02-sp-triplet-400') as calc:
14
        atoms = calc.get_atoms()
        o2_energy = atoms.get_potential_energy()
15
16
   rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
17
18
    print rxn_energy
```

-1.966168

There are a few reasons why our number does not agree with the experimental reaction energy. One reason is related to errors in the O2 dissociation energy, and another reason is related to localization of electrons in the Cu 3d orbitals [36]. The first error of incorrect O₂ dissociation error is a systematic error that can be corrected empirically [36]. 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
1
2
    from jasp import *
3
    import sys
   from pylab import *
   import numpy as np
6
    from ase.dft import DOS
    a = 3.9 # approximate lattice constant
9
   b = a / 2.
10
    bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
11
                 cell=[(0, b, b),
12
                        (b, 0, b),
13
                        (b, b, 0)])
14
15
    with jasp('bulk/pd-dos',
16
              encut=300,
17
18
              xc='PBE',
              lreal=False,
19
20
              kpts=(8, 8, 8), # this is too low for high quality DOS
              atoms=bulk) as calc:
21
22
        # this runs the calculation
23
        bulk.get_potential_energy()
        dos = DOS(calc, width=0.2)
25
        d = dos.get_dos()
26
27
        e = dos.get_energies()
28
29
    import pylab as plt
    plt.plot(e,d)
30
    plt.xlabel('energy [eV]')
31
    plt.ylabel('DOS')
32
    plt.savefig('images/pd-dos.png')
```

None

This DOS looks roughly like you would expect. The peak between -5 to $0~{\rm eV}$ is the Pd d-band.

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

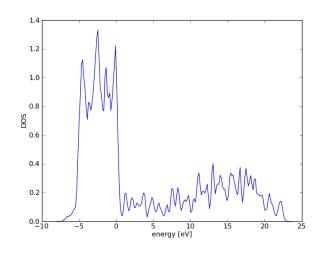


Figure 34: Total DOS for bulk Pd.

Η

```
from jasp import *
    from ase.dft import DOS
2
    with jasp('bulk/pd-dos') as calc:
3
         calc.clone('bulk/pd-dos-ismear-5')
5
6
    with jasp('bulk/pd-dos-ismear-5') as calc:
        bulk = calc.get_atoms()
7
8
         calc.set(ismear=-5)
9
10
        bulk.get_potential_energy()
11
        dos = DOS(calc, width=0.2)
12
13
        d = dos.get_dos()
         e = dos.get_energies()
14
15
    import pylab as plt
16
    plt.plot(e, d)
17
18
    plt.xlabel('energy [eV]')
    plt.ylabel('DOS')
19
    plt.savefig('images/pd-dos-ismear-5.png')
```

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

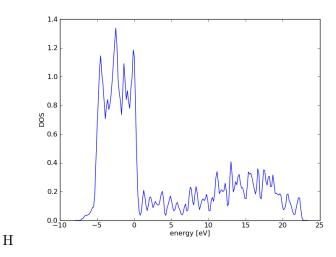


Figure 35: Total DOS for Pd computed with ISMEAR=-5

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
2
    from jasp import *
    from ase.calculators.vasp import VaspDos
3
    import sys
5
6
    from pylab import *
    import numpy as np
7
8
    a = 3.9 # approximate lattice constant
9
10
    b = a / 2.
    bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
11
                  cell=[(0, b, b),
12
                        (b, 0, b),
13
                        (b, b, 0)])
14
```

```
15
    with jasp('bulk/pd-ados',
16
17
              encut=300,
              xc='PBE',
18
19
              lreal=False,
              rwigs=[1.5], # wigner-seitz radii for ados
20
21
              kpts=(8, 8, 8),
              atoms=bulk) as calc:
22
23
24
        # this runs the calculation
        bulk.get_potential_energy()
25
26
        # now get results
27
28
        ados = VaspDos(efermi=calc.get_fermi_level())
29
30
        energies = ados.energy
        dos = ados.site_dos(0, 'd')
31
32
        #we will select energies in the range of -10, 5
33
        ind = (energies < 5) & (energies > -10)
34
35
        energies = energies[ind]
36
        dos = dos[ind]
37
38
39
        Nstates = np.trapz(dos, energies)
        occupied = energies <= 0.0
40
        N_occupied_states = np.trapz(dos[occupied], energies[occupied])
41
        ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
42
43
        wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)
44
45
        print 'Total # states = %1.2f' % Nstates
        print 'number of occupied states = %1.2f' % N_occupied_states
46
        print 'd-band center = %1.2f eV' % ed
47
        print 'd-band width = %1.2f eV' % np.sqrt(wd2)
48
49
        # plot the d-band
50
        plot(energies, dos, label='$d$-orbitals')
51
52
        # plot the occupied states in shaded gray
53
        fill_between(x=energies[occupied],
54
55
                     y1=dos[occupied],
                     y2=zeros(dos[occupied].shape),
56
                     color='gray')
57
58
        xlabel('$E - E_f$ (eV)')
59
60
        ylabel('DOS (arbitrary units)')
61
    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
```

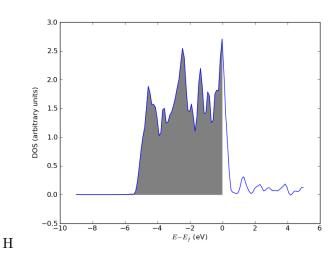


Figure 36: Atom projected d-band for bulk Pd. The shaded area corresponds to the occupied states below the Fermi level.

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 *
1
    JASPRC['queue.walltime'] = '10:00:00'
3
    from ase import Atom, Atoms
4
    from ase.visualize import view
5
6
7
    a = 5.38936
    atoms = Atoms([Atom('Si',[0,0,0]),
8
                    Atom('Si',[0.25, 0.25, 0.25])])
10
11
    {\tt atoms.set\_cell([[a/2.,\ a/2.,\ 0.0],}
12
                      [0.0, a/2., a/2.],
                      [a/2., 0.0, a/2.]],scale_atoms=True)
13
14
    with jasp('bulk/Si-selfconsistent',
15
               xc='PBE',
16
               prec='Medium',
17
               istart=0,
18
19
               icharg=2,
               ediff=0.1e-03,
20
               kpts=(4,4,4),
21
```

```
22 atoms=atoms) as calc:
23 print atoms.get_potential_energy()
```

-10.799983

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

func:ase.calculators.vasp.Vasp.read_kpoints does not support reading in this KPOINTS file yet. mod: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()]
```

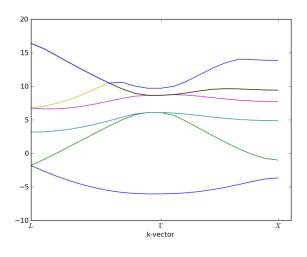


Figure 37: Calculated band-structure for Si.

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

Η

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 *
1
2
    JASPRC['queue.walltime'] = '1:00:00'
    from ase.lattice.cubic import BodyCenteredCubic
5
    atoms = BodyCenteredCubic(directions=[[1,0,0],
                                            [0,1,0],
6
                                            [0,0,1]],
7
8
                                            size=(1,1,1),
9
                                            symbol='Fe')
10
    NUPDOWNS = [0.0, 2.0, 4.0, 5.0, 6.0, 8.0]
11
12
    energies = []
    for B in NUPDOWNS:
13
        with jasp('bulk/Fe-bcc-fixedmagmom-{0:1.2f}'.format(B),
14
15
               xc='PBE',
               encut=300.
16
               kpts=(4,4,4),
17
               ispin=2,
18
               nupdown=B,
19
20
               atoms=atoms) as calc:
21
22
                 e = atoms.get_potential_energy()
23
                 energies.append(e)
             except (VaspSubmitted, VaspQueued):
24
25
                 pass
26
27
    import matplotlib.pyplot as plt
    plt.plot(NUPDOWNS, energies)
28
    plt.xlabel('Total Magnetic Moment')
    plt.ylabel('Energy (eV)')
30
    plt.savefig('images/Fe-fixedmagmom.png')
```

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

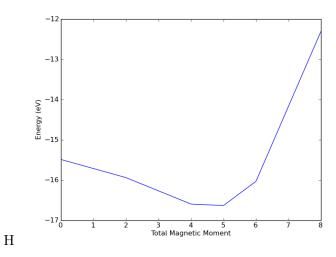


Figure 38: Total energy vs. total magnetic moment for bcc Fe.

```
1
    from jasp import *
    from ase.lattice.cubic import BodyCenteredCubic
2
3
    atoms = BodyCenteredCubic(directions=[[1,0,0],
5
6
                                             [0,1,0],
                                            [0,0,1]],
7
8
                                            size=(1,1,1),
                                            symbol='Fe')
9
10
11
    for atom in atoms:
        atom.magmom = 2.5
12
13
    with jasp('bulk/Fe-bcc-sp-1',
14
15
               xc='PBE',
               encut=300,
16
               kpts=(4,4,4),
17
18
               ispin=2,
               atoms=atoms) as calc:
19
20
             try:
^{21}
                 e = atoms.get_potential_energy()
                 print atoms.get_magnetic_moment()
22
23
                 print atoms.get_magnetic_moments()
             except (VaspSubmitted, VaspQueued):
^{24}
25
                 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 mod:ase.Atom object.

```
1
    from jasp import *
    from ase import Atom, Atoms
2
3
    atoms = Atoms([Atom('Fe', [0.00, 0.00, 0.00], magmom=5),
4
                    Atom('Fe', [4.3, 4.3, 4.3], magmom=-5),
5
                    Atom(^{\circ}0, [2.15, 2.15, 2.15], magmom=0),
6
                    Atom('0', [6.45, 6.45, 6.45], magmom=0)],
7
8
                    cell=[[4.3,
                                  2.15,
                                            2.15],
                          [2.15,
                                   4.3,
                                             2.15],
9
10
                          [2.15,
                                    2.15,
                                             4.3]])
11
12
    with jasp('bulk/afm-feo',
13
               encut=350,
              prec='Normal',
14
15
               nupdown=0, # this forces a non-magnetic solution
16
               lorbit=11,
17
18
              lreal=False.
19
              atoms=atoms) as calc:
20
        print 'Magnetic moments = ',atoms.get_magnetic_moments()
        print 'Total magnetic moment = ',atoms.get_magnetic_moment()
21
```

```
[ 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 [30] recent Henkelman paper

6 Surfaces

6.1 Surface structures

As with molecules and bulk systems mod: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

6.1.2 vicinal surfaces

Many stepped surfaces are not that difficult to make now. The main idea in generating them is described here. mod: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

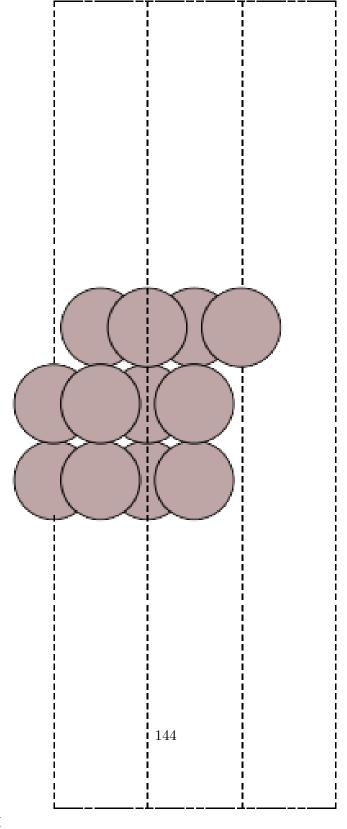
# Au(211) with 9 layers

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



Η

Figure 39: An Al(111) slab with three layers and 20 $\hbox{Å}$ of vacuum.

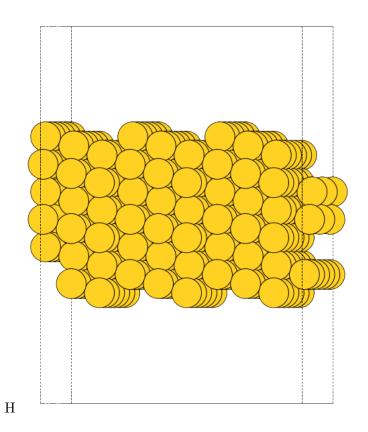


Figure 40: An $\operatorname{Au}(211)$ surface constructed with mod: ase.

6.2 Surface relaxation

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

```
1
   from jasp import *
   from ase.lattice.surface import *
2
   atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
4
   with jasp('surfaces/Al-slab-unrelaxed',
           xc='PBE',
7
8
           kpts=(6,6,1),
9
           encut=350.
10
           atoms=atoms) as calc:
      atoms.get_forces()
11
      print calc
     VASP calculation from /home/jkitchin/dft-org/surfaces/Al-slab-unrelaxed
     converged: True
     Energy = -14.177069 eV
     Unit cell vectors (angstroms)
                                   length
            X
                     У
                            z
     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,
                                  zz,
                           уу,
                                                  ΧZ,
                                          yz,
                                                          ху
                         0.007 0.002 -0.000 -0.000 -0.000
                  0.007
    Atom#
                       position [x,y,z]
            sym
                                                   tag rmsForce
       0
            Al [0.000
                              0.000
                                                         0.01
                                         10.000]
       1
            Al [1.432
                              0.827
                                         12.338]
                                                    3
                                                        0.18
       2
            Al [2.864
                              1.653
                                         14.677]
                                                    2
                                                        0.18
       3
                [0.000
                              0.000
                                         17.015]
            Al
                                                    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 atoms[i].z - atoms[i-1].z
```

- 2.33826859022
- 2.33826859022
- 2.33826859022

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 mod: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 mod:ase.constraints.FixScaled. The other constraints are not written to the POSCAR and are not used by VASP.

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

```
1 from jasp import *
2 from ase.lattice.surface import *
3 from ase.constraints import FixAtoms
  atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
   with jasp('surfaces/Al-slab-relaxed',
           xc='PBE',
8
           kpts=(6,6,1),
9
10
           encut=350,
           ibrion=2,
11
12
           isif=2,
           nsw=10,
13
14
           atoms=atoms) as calc:
15
16
      constraint = FixAtoms(mask=[atom.tag >= 3 for atom in atoms])
17
      atoms.set_constraint(constraint)
18
19
      atoms.get_forces()
      print calc
20
   : ------
     VASP calculation from /home/jkitchin/dft-org/surfaces/Al-slab-relaxed
     converged: True
     Energy = -14.181417 eV
     Unit cell vectors (angstroms)
                           z
                                   length
                    V
     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,
                                 zz,
                                                 xz,
                          уу,
                                         yz,
                 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.

```
d_{-}(1,0) = 2.33826859022

d_{-}(2,1) = 2.29891157834

d_{-}(3,2) = 2.34530960922
```

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.

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

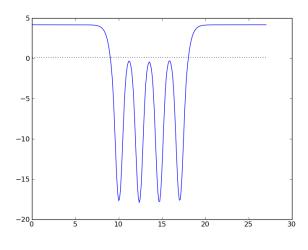


Figure 41: xy averaged local electrostatic potential of an Al(111) slab.

6.4 Surface energy

Η

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

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

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

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

Η

Figure 42: Schematic figure illustrating the calculation of a surface energy.

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

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

a function of the slab thickness.

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

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

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

from jasp import * from ase.units import *

```
from ase.lattice.surface import *
3
    import matplotlib.pyplot as plt
    Nlayers = [3, 4, 5, 6, 7, 8, 9, 10, 11]
6
    energies = []
    sigmas = []
8
    for n in Nlayers:
10
11
        slab = fcc111('Cu', size=(1,1,n), vacuum=10.0)
12
        slab.center()
13
14
        with jasp('bulk/Cu-layers/{0}'.format(n),
15
                  xc='PBE',
16
17
                  encut=350,
                  kpts=(8,8,1),
18
19
                  atoms=slab) as calc:
            calc.set_nbands(f=2) # the default nbands in Vasp is too low for Al
20
21
                energies.append(slab.get_potential_energy())
22
23
            except (VaspSubmitted, VaspQueued):
24
                pass
25
    for i in range(len(Nlayers)-1):
26
        N = Nlayers[i]
27
        DeltaE_N = energies[i+1] - energies[i]
28
        sigma = 0.5*(-N*energies[i+1] + (N+1)*energies[i])
29
        sigmas.append(sigma)
30
31
        print 'sigma = {0} eV/atom'.format(sigma)
32
    plt.plot(Nlayers[0:-1], sigmas,'bo-')
33
    plt.xlabel('Number of layers')
34
    plt.ylabel('Surface energy (eV/atom)')
35
    plt.savefig('images/Cu-unrelaxed-surface-energy.png')
     sigma = 0.5522845 \text{ eV/atom}
      sigma = 0.3979905 eV/atom
     sigma = 0.5903805 eV/atom
```

```
sigma = 0.5522845 eV/atom

sigma = 0.3979905 eV/atom

sigma = 0.5903805 eV/atom

sigma = 0.3177165 eV/atom

sigma = 0.579716 eV/atom

sigma = 0.321148 eV/atom

sigma = 0.593281 eV/atom

sigma = 0.401526 eV/atom
```

One reason for the oscillations may be quantum size effects [9]. In [32] the surface energy of Cu(111) is reported as 0.48 eV/atom, or 1.36 J/m². 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.

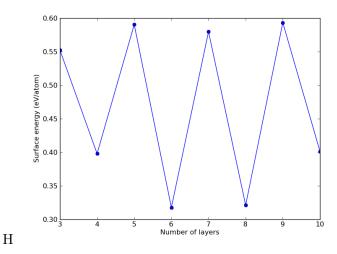


Figure 43: Surface energy of an Al(111) slab as a function of thickness.

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

 $sigma = 1.3628146074 J/m^2$

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 [12] for an example of computing Mo\$_2\$C surface energies and particle shapes, and [37] 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 [35] .

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 [27] (see [3] for a small correction) to minimize the effect.

Here we will just illustrate the effect.

6.5.1 TODO slab with no dipole correction

label plot and save

```
# compute local potential of slab with no dipole
    from ase.lattice.surface import *
2
    from jasp import *
3
    from jasp.volumetric_data import *
5
6
    slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
    add_adsorbate(slab,'Na',height=1.2,position='fcc')
7
8
    slab.center()
9
10
    with jasp('surfaces/Al-Na-nodip',
11
12
              xc='PBE',
13
              encut=340,
              kpts=(2,2,1),
14
              lvtot=True, # write out local potential
15
              lvhar=True, # write out only electrostatic potential, not xc pot
16
               atoms=slab) as calc:
17
18
         calc.calculate()
19
20
        x,y,z,lp = calc.get_local_potential()
21
        nx, ny, nz = lp.shape
22
        axy = [np.average(lp[:,:,z]) for z in range(nz)]
23
        # setup the x-axis in realspace
24
25
        uc = atoms.get_cell()
        xaxis = np.linspace(0,uc[2][2],nz)
26
27
        plot(xaxis, axy)
28
29
        show()
```

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 *
5
    slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
6
    add_adsorbate(slab,'Na',height=1.2,position='fcc')
    slab.center()
9
10
    with jasp('surfaces/Al-Na-dip',
              xc='PBE',
11
12
              encut=340,
13
              kpts=(2,2,1),
              14
              lvtot=True, # write out local potential
15
              lvhar=True, # write out only electrostatic potential, not xc pot
16
              atoms=slab) as calc:
17
18
        calc.calculate()
19
20
21
        x,y,z,cd = calc.get_charge_density()
        n0, n1, n2 = cd.shape
23
        nelements = n0*n1*n2
        voxel_volume = slab.get_volume()/nelements
24
25
        total_electron_charge = cd.sum()*voxel_volume
26
27
        electron_density_center = np.array([(cd*x).sum(),
                                             (cd*y).sum(),
28
29
                                             (cd*z).sum()])
30
        electron_density_center *= voxel_volume
        electron_density_center /= total_electron_charge
31
32
        print electron_density_center
33
        uc = slab.get_cell()
34
35
        # get scaled electron charge density center
36
37
        sedc = np.dot(np.linalg.inv(uc.T),electron_density_center.T).T
38
        ## # now turn on potential calculation: violates jasp workflow!
39
        calc.set(ldipol=True, dipol=sedc)
40
41
42
        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 *
1
    from pylab import *
2
3
4
    with jasp('surfaces/Al-Na-nodip') as calc:
         atoms = calc.get_atoms()
5
6
        x,y,z,lp = calc.get_local_potential()
7
8
        nx, ny, nz = lp.shape
9
        axy_1 = [np.average(lp[:,:,z]) for z in range(nz)]
10
         # setup the x-axis in realspace
11
        uc = atoms.get_cell()
12
        xaxis_1 = np.linspace(0,uc[2][2],nz)
13
14
         e1 = atoms.get_potential_energy()
15
16
    with jasp('surfaces/Al-Na-dip') as calc:
17
        atoms = calc.get_atoms()
18
19
        x,y,z,lp = calc.get_local_potential()
20
^{21}
        nx, ny, nz = lp.shape
22
23
        axy_2 = [np.average(lp[:,:,z]) for z in range(nz)]
        \# setup the x-axis in realspace
24
25
        uc = atoms.get_cell()
        xaxis_2 = np.linspace(0,uc[2][2],nz)
26
27
28
         ef2 = calc.get_fermi_level()
         e2 = atoms.get_potential_energy()
29
30
    print 'The difference in energy is {0} eV.'.format(e2-e1)
31
32
    plot(xaxis_1, axy_1, label='no dipole correction')
33
    plot(xaxis_2, axy_2, label='dipole correction')
34
    plot([min(xaxis_2), max(xaxis_2)],[ef2,ef2], 'k:', label='Fermi level')
35
    xlabel('z ($\AA$)')
36
37
    ylabel('xy-averaged electrostatic potential')
    legend(loc='best')
38
    savefig('images/dip-vs-nodip-esp.png')
```

The difference in energy is 0.006389 eV.

The key points to notice in this figure are:

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

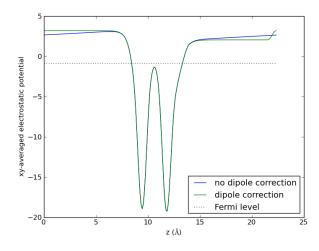


Figure 44: comparison of the electrostatic potentials with a dipole correction and without it.

- 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

Η

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

 $slab + gas-phase molecule \rightarrow slab_adsorbate + products$

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 *
1
2
    from ase.lattice.surface import *
    from ase.constraints import FixAtoms
5
    atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
    constraint = FixAtoms(mask=[True for atom in atoms])
6
    atoms.set_constraint(constraint)
9
    with jasp('surfaces/Pt-slab',
              xc='PBE',
10
11
               kpts=(4,4,1),
12
               encut=350.
              atoms=atoms) as calc:
13
14
         slab_e = atoms.get_potential_energy()
```

fcc site

```
1
    from jasp import *
2
3
    from ase.lattice.surface import *
    from ase.constraints import FixAtoms
4
5
6
    atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
    # note this function only works when atoms are created by the surface module.
8
    add_adsorbate(atoms, '0', height=1.2, position='fcc')
9
10
    constraint = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
11
    atoms.set_constraint(constraint)
12
13
    with jasp('surfaces/Pt-slab-0-fcc',
14
              xc='PBE',
15
              kpts=(4,4,1),
16
               encut=350,
17
               ibrion=2,
```

```
19     nsw=25,
20     atoms=atoms) as calc:
21     slab_o_fcc_e = atoms.get_potential_energy()
```

bridge site

```
1
    from jasp import *
2
    from ase.lattice.surface import *
3
    from ase.constraints import FixAtoms
    atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
5
6
    # note this function only works when atoms are created by the surface module.
7
8
    add_adsorbate(atoms, '0', height=1.2, position='bridge')
9
    constraint = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
10
    atoms.set_constraint(constraint)
11
12
    with jasp('surfaces/Pt-slab-O-bridge',
13
              xc='PBE',
14
               kpts=(4,4,1),
15
               encut=350,
16
17
               ibrion=2,
18
              nsw=25,
19
               atoms=atoms) as calc:
20
         atoms.get_potential_energy()
```

hcp site

```
from jasp import *
1
2
    from ase.lattice.surface import *
    from ase.constraints import FixAtoms
3
    atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
5
6
    \mbox{\it\#} note this function only works when atoms are created by the surface module.
7
    add_adsorbate(atoms, '0', height=1.2, position='hcp')
8
9
    constraint = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
10
11
    atoms.set_constraint(constraint)
12
13
    with jasp('surfaces/Pt-slab-0-hcp',
               xc='PBE',
14
               kpts=(4,4,1),
15
16
               encut=350,
               ibrion=2,
17
               nsw=25,
18
               atoms=atoms) as calc:
19
20
         atoms.get_potential_energy()
```

Analysis of adsorption energies

```
from jasp import *
1
2
    from ase.io import write
3
    with jasp('surfaces/Pt-slab') as calc:
4
        atoms = calc.get_atoms()
        e_slab = atoms.get_potential_energy()
6
    write('images/pt-slab.png',atoms,show_unit_cell=2)
7
8
9
    with jasp('surfaces/Pt-slab-0-fcc') as calc:
10
        atoms = calc.get_atoms()
        e_slab_o_fcc = atoms.get_potential_energy()
11
12
    write('images/pt-slab-fcc-o.png',atoms,show_unit_cell=2)
13
14
    with jasp('surfaces/Pt-slab-O-hcp') as calc:
        atoms = calc.get_atoms()
15
16
        e_slab_o_hcp = atoms.get_potential_energy()
17
    write('images/pt-slab-hcp-o.png',atoms,show_unit_cell=2)
18
    with jasp('surfaces/Pt-slab-O-bridge') as calc:
19
        atoms = calc.get_atoms()
20
        e_slab_o_bridge = atoms.get_potential_energy()
21
22
    write('images/pt-slab-bridge-o.png',atoms,show_unit_cell=2)
23
24
    with jasp('molecules/02-sp-triplet-350') as calc:
25
        atoms = calc.get_atoms()
        e_02 = atoms.get_potential_energy()
26
27
    Hads_fcc = e_slab_o_fcc - e_slab - 0.5*e_02
28
29
    Hads_hcp = e_slab_o_hcp - e_slab - 0.5*e_02
    Hads_bridge = e_slab_o_bridge - e_slab - 0.5*e_02
30
31
    print 'Hads (fcc) = {0} eV/0'.format(Hads_fcc)
32
    print 'Hads (hcp) = {0} eV/O'.format(Hads_hcp)
33
    print 'Hads (bridge) = {0} eV/0'.format(Hads_bridge)
      Hads (fcc) = -1.0385455 \text{ eV/O}
```

```
Hads (fcc) = -1.0385455 \text{ eV/O}

Hads (hcp) = -0.5986655 \text{ eV/O}

Hads (bridge) = -1.0384925 \text{ 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 45 and hcp (Figure 46 sites, which look like we expect.

The bridge site (Figure 47, however, is clearly not at a bride 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

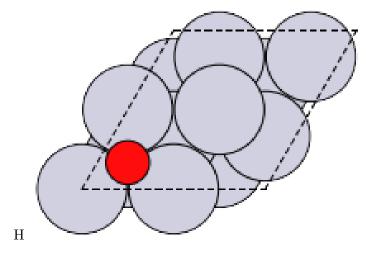


Figure 45: Final geometry of the fcc site.

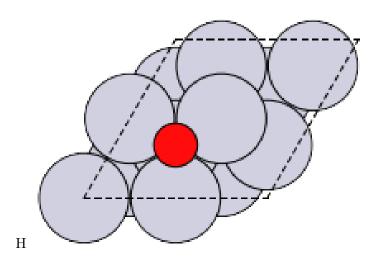


Figure 46: Final geometry of the hcp site.

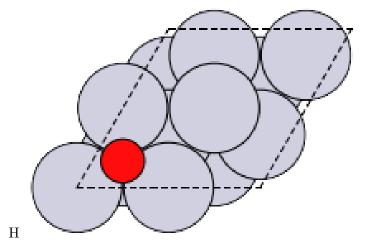


Figure 47: Final geometry of the bridge site. You can see that the oxygen atom ended up in the fcc site.

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)
```

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.

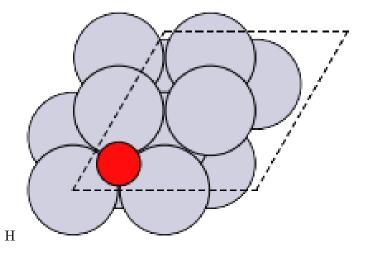


Figure 48: Initial geometry of the bridge site. It is definitely on the bridge.

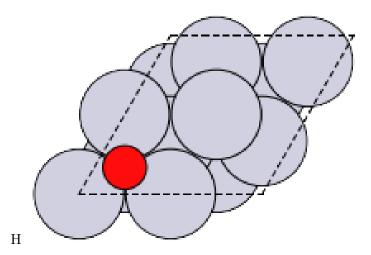


Figure 49: Final geometry of the bridge site. It has fallen into the fcc site.

- 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 func: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 *
1
    JASPRC['mode'] = None
   from ase.lattice.surface import *
    from ase.constraints import FixAtoms, FixScaled
    from ase.io import write
5
    atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
7
    # note this function only works when atoms are created by the surface module.
9
    add_adsorbate(atoms, '0', height=1.2, position='bridge')
10
    constraint1 = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
11
    # fix in xy-direction, free in z. actually, freeze movement in surface
12
    # unit cell, and free along 3rd lattice vector
    constraint2 = FixScaled(atoms.get_cell(), 12, [True, True, False])
14
15
16
    atoms.set_constraint([constraint1, constraint2])
    write('images/Pt-0-bridge-constrained-initial.png', atoms, show_unit_cell=2)
17
    print 'Initial O position: {0}'.format(atoms.positions[-1])
18
19
20
    with jasp('surfaces/Pt-slab-O-bridge-xy-constrained',
21
              xc='PBE'.
              kpts=(4,4,1)
22
23
              encut=350,
              ibrion=2.
24
25
              nsw=25,
              atoms=atoms) as calc:
26
27
        e_bridge = atoms.get_potential_energy()
28
    write('images/Pt-O-bridge-constrained-final.png', atoms, show_unit_cell=2)
29
    print 'Final O position : {0}'.format(atoms.positions[-1])
```

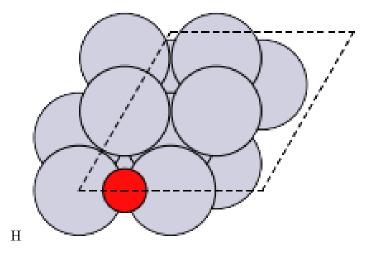


Figure 50: Initial state of the O atom on the bridge site.

```
31
32
    # now compute Hads
    with jasp('surfaces/Pt-slab') as calc:
33
        atoms = calc.get_atoms()
34
35
         e_slab = atoms.get_potential_energy()
36
37
    with jasp('molecules/02-sp-triplet-350') as calc:
38
39
        atoms = calc.get_atoms()
         e_02 = atoms.get_potential_energy()
40
41
    Hads_bridge = e_bridge - e_slab - 0.5*e_02
42
43
    print 'Hads (bridge) = {0:1.3f} eV/O'.format(Hads_bridge)
```

```
Initial O position: [ 1.386 0. 15.726]
Final O position : [ 1.386 0. 15.977]
Hads (bridge) = -0.488 eV/O
```

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.

6.6.2 Coverage dependence

The adsorbates on the surface can interact with each other which results in coverage dependent adsorption energies [15]. Coverage dependence is not difficult to model; we simply compute adsorption energies in different size

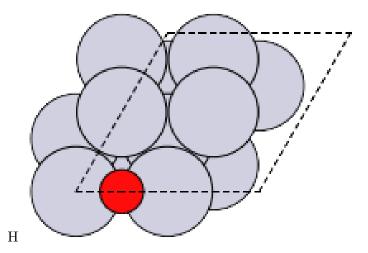


Figure 51: Final state of the constrained O atom, still on the bridge site.

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 [24, 25]
- Coverage effects of atomic adsorbates on Pd(111) [18]
- Simple model for estimating coverage dependence [15]
- Coverage effects on alloys [19]

clean slab calculation

```
1
    from jasp import *
2
    from ase.lattice.surface import *
    from ase.constraints import FixAtoms
3
    atoms = fcc111('Pt', size=(1,1,3), vacuum=10.0)
    constraint = FixAtoms(mask=[True for atom in atoms])
6
    atoms.set_constraint(constraint)
8
    with jasp('surfaces/Pt-slab-1x1',
              xc='PBE',
10
              kpts=(8,8,1),
11
               encut=350,
12
```

```
atoms=atoms) as calc:
14 slab_e = atoms.get_potential_energy()
```

fcc site at 1 ML coverage

```
from jasp import *
1
    from ase.lattice.surface import *
3
    from ase.constraints import FixAtoms
5
    atoms = fcc111('Pt', size=(1,1,3), vacuum=10.0)
6
8
    # note this function only works when atoms are created by the surface module.
9
    add_adsorbate(atoms, '0', height=1.2, position='fcc')
10
11
    constraint = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
    atoms.set_constraint(constraint)
12
13
14
    with jasp('surfaces/Pt-slab-1x1-0-fcc',
              xc='PBE',
15
              kpts=(8,8,1),
16
               encut=350,
17
               ibrion=2,
18
19
              nsw=25,
              atoms=atoms) as calc:
20
^{21}
         slab_o_fcc_e = atoms.get_potential_energy()
        print slab_o_fcc_e
22
```

-21.941036

Adsorption energy at 1ML

```
from jasp import *
1
    with jasp('surfaces/Pt-slab-1x1-0-fcc') as calc:
3
        atoms = calc.get_atoms()
        e_slab_o = atoms.get_potential_energy()
5
6
7
    # clean slab
    with jasp('surfaces/Pt-slab-1x1') as calc:
8
        atoms = calc.get_atoms()
        e_slab = atoms.get_potential_energy()
10
11
    with jasp('molecules/02-sp-triplet-350') as calc:
12
        atoms = calc.get_atoms()
13
14
        e_02 = atoms.get_potential_energy()
15
    hads = e_slab_o - e_slab - 0.5*e_02
16
    print 'Hads (1ML) = {0:1.3f} eV'.format(hads)
17
```

Hads (1ML) = -0.099 eV

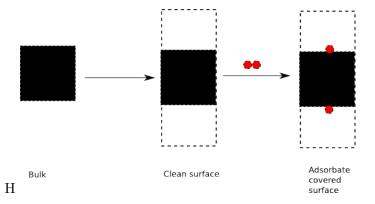


Figure 52: Schematic of forming a surface with adsorbates. First we form two clean surfaces by cleaving the bulk, then allow adsorption to occur on the surfaces.

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:

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 $H_{ads} = E_{slab,ads} - E_{slab,ads}$ $E_{slab} - E_{ads}$ 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 $\frac{\Delta E}{2A}=\gamma=\gamma_{clean}+\frac{H_{ads}}{A}$

$$\frac{\Delta E}{2A} = \gamma = \gamma_{clean} + \frac{H_{ads}}{A}$$

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 (Fix-Atoms or FixScaled) and you use IBRION=5. The other settings (6, 7, 8) do not respect the selective dynamics constraints.

```
from jasp import *
2
    with jasp('surfaces/Pt-slab-O-fcc') as calc:
3
        calc.clone('surfaces/Pt-slab-0-fcc-vib')
4
5
    with jasp('surfaces/Pt-slab-O-fcc-vib') as calc:
6
        calc.set(ibrion=5,# finite differences with selective dynamics
7
8
                nfree=2, # central differences (default)
                potim=0.015,# default as well
9
                ediff=1e-8,
10
                nsw=1)
11
12
        atoms = calc.get_atoms()
13
        f,v = calc.get_vibrational_modes(0)
        print calc.get_vibrational_modes()[0]
14
15
   from ase.units import *
16
    c = 3e10 \# cm/s
17
    h = 4.135667516e-15 # eV*s
18
19
   print 'vibrational energy = {0} eV'.format(f)
20
    print 'vibrational energy = {0} meV'.format(f/meV)
21
    print 'vibrational freq = {0} 1/s'.format(f/h)
    print 'vibrational freq
                            = {0} cm^(-1)'.format(f/(h*c))
     [ 0.062 0.044 0.044]
     vibrational energy = 0.061606647 eV
     vibrational energy = 61.606647 meV
     vibrational freq
                              = 1.48964216204e+13 1/s
     vibrational freq
                              = 496.547387346 \text{ 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.

```
1
    from jasp import *
2
3
    with jasp('surfaces/Pt-slab-O-fcc') as calc:
        calc.clone('Pt-slab-0-fcc-vib-ibrion=6')
4
5
    with jasp('surfaces/Pt-slab-O-fcc-vib-ibrion=6') as calc:
6
        calc.set(ibrion=6,# finite differences with symmetry
7
                nfree=2, # central differences (default)
8
9
                potim=0.015,# default as well
                ediff=1e-8,
10
11
                nsw=1)
12
       atoms = calc.get_atoms()
        #print calc.get_vibrational_frequencies()
13
14
       f,m = calc.get_vibrational_modes(0)
       allfreq = calc.get_vibrational_modes()[0]
15
16
       print len(allfreq), allfreq
    from ase.units import *
17
    c = 3e10 \# cm/s
18
   h = 4.135667516e-15 # eV*s
19
20
   print 'vibrational energy = {0} eV'.format(f)
^{21}
    print 'vibrational energy = {0} meV'.format(f/meV)
22
    print 'vibrational freq = {0} 1/s'.format(f/h)
23
    print 'vibrational freq
                            = \{0\} cm^(-1)'.format(f/(h*c))
                                                                       0.023 0.022 0.022
     39 [ 0.064  0.046  0.046  0.024  0.024
                                                       0.023
                                                                0.023
                                                   0.015
        0.02
                0.02
                         0.02
                                  0.015
                                          0.015
                                                            0.015
                                                                     0.015 0.014
                                                                     0.011 0.011 0.01
        0.013
                0.012
                         0.012
                                  0.012
                                          0.012
                                                   0.012
                                                            0.011
                                                                                               0.01
                0.009 0.009 0.004 0.004
                                                            0.
                                                                     0.
                                                                           ]
     vibrational energy = 0.063537929 eV
     vibrational energy = 63.537929 meV
     vibrational freq
                             = 1.53634035507e+13 1/s
```

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.

 $= 512.113451691 \text{ cm}^{-1}$

vibrational freq

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
2
3
    with jasp('surfaces/Pt-slab-O-bridge-xy-constrained') as calc:
4
       calc.clone('surfaces/Pt-slab-0-bridge-vib')
5
6
       atoms = calc.get_atoms()
7
    del atoms.constraints
8
    constraint = FixAtoms(mask=[atom.symbol != '0' for atom in atoms])
9
    atoms.set_constraint([constraint])
10
11
   with jasp('surfaces/Pt-slab-O-bridge-vib') as calc:
12
13
        calc.set(ibrion=5,# finite differences with selective dynamics
                nfree=2, # central differences (default)
14
                potim=0.015,# default as well
15
                ediff=1e-8.
16
17
                nsw=1)
       atoms.set_calculator(calc) # reset atoms
18
       atoms.get_potential_energy()
19
20
        #print calc.get_vibrational_frequencies()
       f,v = calc.get_vibrational_modes(2)
21
22
       print calc.get_vibrational_modes()[0]
23
24 from ase.units import *
  c = 3e10 \# cm/s
25
   h = 4.135667516e-15 # eV*s
26
28 print 'vibrational energy = {0} eV'.format(f)
29 print 'vibrational energy = {0} meV'.format(f/meV)
30 print 'vibrational freq = \{0\} 1/s'.format(f/h)
    print 'vibrational freq = {0} cm^(-1)'.format(f/(h*c))
      [0.065486063999999997, 0.045079206000000004, (0.019955987000000001+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 [28] 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
7
    with jasp('surfaces/Pt-slab-O-fcc') as calc:
        initial_atoms = calc.get_atoms()
8
9
    with jasp('surfaces/Pt-slab-O-hcp') as calc:
10
        final_atoms = calc.get_atoms()
11
12
    # here is our estimated transition state. we use vector geometry to
13
    # define the bridge position, and add 1.451 Ang to z based on our
14
    # previous bridge calculation.
15
16
    ts = initial_atoms.copy()
    ts.positions[-1] = 0.5*(ts.positions[9] + ts.positions[10]) + [0,0,1.451]
17
18
    # construct the band
19
    images = [initial_atoms]
    images += [initial_atoms.copy()]
21
22
    images += [ts.copy()] # this is the TS
23
24
    neb = NEB(images)
25
    # Interpolate linearly the positions of these images:
    neb.interpolate()
26
27
    # now add the second half
28
    images2 = [ts.copy()]
29
    images2 += [ts.copy()]
    images2 += [final_atoms]
31
    neb2 = NEB(images2)
33
    neb2.interpolate()
34
35
    # collect final band. Note we do not repeat the TS in the second half
36
37
    final_images = images + images2[1:]
38
    with jasp('surfaces/Pt-O-fcc-hcp-neb',
39
40
              ibrion=1.
41
              nsw=90,
42
               spring=-5,
               atoms=final_images) as calc:
43
44
        try:
45
```

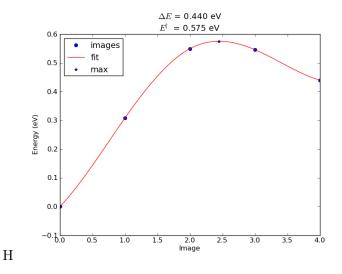


Figure 53: Energy pathway for O diffusion from an fcc to hcp site with a spline fit to determine the barrier

```
images, energies = calc.get_neb()
46
47
             p = calc.plot_neb(show=False)
48
49
             # remember you are in surfaces/Pt-O-fcc-hcp-neb, so to save in
             # the images directory you need ../../ in the path to get you
50
             # back up.
51
            plt.savefig('../../images/pt-o-fcc-hcp-neb.png')
52
         except (VaspSubmitted, VaspQueued):
53
            pass
```

Optimization terminated successfully.

Current function value: -0.575389

Iterations: 12

Function evaluations: 24

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
1
    from jasp import *
    with jasp('surfaces/Pt-O-fcc-hcp-neb') as calc:
3
4
        calc.clone('surfaces/Pt-O-fcc-hcp-cineb')
    with jasp('surfaces/Pt-O-fcc-hcp-cineb',debug=logging.DEBUG) as calc:
6
        calc.set(ichain=0,lclimb=True)
8
9
        images, energies = calc.get_neb(npi=4)
10
        calc.plot_neb(show=False)
    import matplotlib.pyplot as plt
11
    plt.savefig('images/pt-o-cineb.png')
```

Optimization terminated successfully.

Current function value: -0.575288

Iterations: 12

Function evaluations: 24

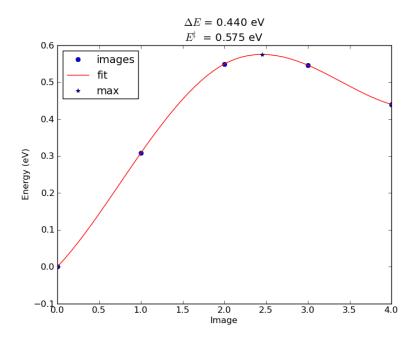


Figure 54: Climbing image NEB

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's 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
6
    A = 31.32234; B = -20.23531; C = 57.86644
    D = -36.50624; E = -0.007374; F = -8.903471
8
    G = 246.7945; H = 0.0
9
10
    def entropy(T):
11
         ''', entropy returned as eV/K
12
        T in K
13
14
        t = T/1000.
15
        s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
16
^{17}
        \texttt{return } \texttt{s}*\texttt{J/mol/K}
18
19
    def enthalpy(T):
        ''' H - H(298.15) returned as eV/molecule'''
20
21
        h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
22
        return h*kJ/mol
23
^{24}
    T = np.linspace(100,700)
^{25}
26
    G = enthalpy(T) - T*entropy(T)
27
28
    plt.plot(T,G)
29
30
    plt.xlabel('Temperature (K)')
   plt.ylabel('$\Delta G^\circ$ (eV)')
31
   plt.savefig('images/02-mu.png')
```

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 *
```

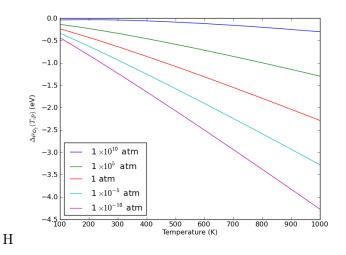


Figure 55: Effect of temperature on the Gibbs free energy of an O_2 molecule at 1 atm.

```
atm = 101325*Pascal #atm is not defined in units
5
6
    K = 1
7
    # examine range over 10^-10 to 10^10 atm
8
9
    P = np.logspace(-10,10)*atm
10
11
    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')
12
    plt.xlabel('Pressure (atm)')
13
    plt.ylabel('$\Delta G$ (eV)')
    plt.legend(loc='best')
15
    plt.savefig('images/02-g-p.png')
```

None

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 oxygen: $\mu_{O_2}=E_{O_2}^{DFT}+E_{O_2}^{ZPE}+\Delta\mu(T)+kT\ln(P/P_0)$

$$\mu_{O_2} = E_{O_2}^{DFT} + E_{O_2}^{ZPE} + \Delta\mu(T) + kT\ln(P/P_0)$$

bulk phase stability of oxides

We will consider the effects of oxygen pressure and temperature on the formation energy of Ag\$_2\$O and Cu\$_2\$O.

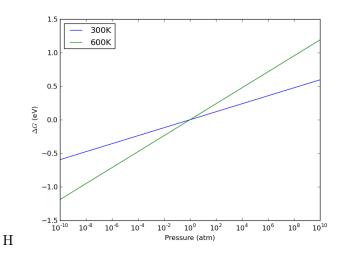


Figure 56: Effects of pressure on the ideal gas Gibbs free energy of O_2 .

$$2Cu + 1/2O_2 \rightarrow Cu_2O$$

In atomistic thermodynamics, we define the free energy of formation as: We define the formation energy 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 $\Delta\mu_{O_2}(P,T) = \delta\mu(T) + k\tilde{T}\ln(P/P_0)$. 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 formation 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\$_2\$O is -1.9521 eV/formula unit. The formation energy for Ag\$_2\$O 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
1
    import matplotlib.pyplot as plt
    from ase.units import *
    from scipy.optimize import fsolve
    K = 1. #not defined in ase.units!
6
    atm = 101325*Pascal
7
    # Shomate parameters valid from 100-700K
9
10
   A = 31.32234; B = -20.23531; C = 57.86644
   D = -36.50624; E = -0.007374; F = -8.903471
11
    G = 246.7945; H = 0.0
12
13
    def entropy(T):
14
        ''', entropy returned as eV/K
15
        T in K
16
17
        t = T/1000.
18
        s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
19
20
        return s*J/mol/K
21
22
    def enthalpy(T):
        ''' H - H(298.15) returned as eV/molecule'''
23
24
        h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
25
        return h*kJ/mol
26
27
    def DeltaMu(T,P):
28
29
        T in K
30
        P in atm
31
32
        return enthalpy(T) - T*entropy(T) + kB*T*np.log(P/atm)
33
    T = np.linspace(100,1000)
35
    P = 1e-10*atm
36
37
38
    def func(T):
39
        'Cu20'
        return -1.95 - 0.5*DeltaMu(T,P)
40
41
    print 'Cu20 decomposition temperature is {0} K'.format(fsolve(func, 900))
42
43
44
    def func(T):
45
        'Ag20'
46
        return -0.99 - 0.5*DeltaMu(T,P)
47
48
    print 'Ag20 decomposition temperature is {0} K'.format(fsolve(func, 470))
49
    # you have use \\times to escape the first \ in pyplot
50
    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')
```

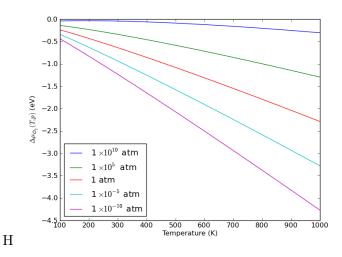


Figure 57: $\Delta\mu_{O_2}(T,p)$ at different pressures and temperatures.

```
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('$\Delta \mu_{0_2}(T,p)$ (eV)')

plt.legend(loc='best')

plt.savefig('images/02-mu.png')
```

Cu20 decomposition temperature is [916.90220698] K Ag20 decomposition temperature is [478.25748939] K

```
import numpy as np
    import matplotlib.pyplot as plt
2
3
    from ase.units import *
4
    from scipy.optimize import fsolve
    K = 1. #not defined in ase.units!
6
    atm = 101325*Pascal
7
8
    \# Shomate parameters valid from 100-700K
9
    A = 31.32234; B = -20.23531; C = 57.86644
10
    D = -36.50624; E = -0.007374; F = -8.903471
11
    G = 246.7945; H = 0.0
12
13
14
    def entropy(T):
         ''', entropy returned as eV/K
15
         T in K
16
```

```
,,,
17
18
        t = T/1000.
        s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
19
        return s*J/mol/K
20
^{21}
    def enthalpy(T):
22
         ''' H - H(298.15) returned as eV/molecule'''
23
        t = T/1000.
24
        h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
25
        return h*kJ/mol
26
27
28
    def DeltaMu(T,P):
29
30
        T in K
31
        P in atm
32
33
        return enthalpy(T) - T*entropy(T) + kB*T*np.log(P/atm)
34
   P = np.logspace(-11,1,10)*atm
35
    T = []
36
    for p in P:
37
38
        def func(T):
39
40
           return -0.99 - 0.5*DeltaMu(T,p)
        T.append(fsolve(func, 450)[0])
41
42
    plt.semilogy(T,P/atm)
43
    plt.xlabel('Temperature (K)')
44
   plt.ylabel('Pressure (atm)')
    plt.text(800,1e-7,'Ag')
46
    plt.text(600,1e-3,'Ag$_2$0')
    plt.savefig('images/Ag20-decomposition.png')
```

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
1
    import matplotlib.pyplot as plt
    from ase.units import *
3
5
    K = 1. #not defined in ase.units!
    atm = 101325*Pascal
6
    Hf = -0.99
8
9
10
    Dmu = np.linspace(-4,0)
11
12
    Hf = -0.99 - 0.5*Dmu
13
14
    plt.plot(Dmu, Hf, label='Ag$_2$0')
15
plt.plot(Dmu, np.zeros(Hf.shape), label='Ag')
17 plt.xlabel('$\Delta \mu_{0_2}$ (eV)')
```

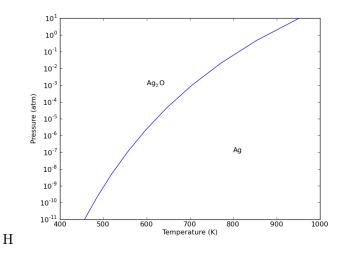


Figure 58: Temperature dependent decomposition pressure for Ag\$.2\$O.

```
18 plt.ylabel('$H_f$ (eV)')
19 plt.show()
```

7.1.1 TODO save graph

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 57): About 500K at 1e-10 atm, 600K at 1e-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 \mu_{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:

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

```
1
    import numpy as np
    import matplotlib.pyplot as plt
    fcc25 = -1.04
   hcp25 = -0.60
    bridge25 = -0.49
6
    fcc1 = -0.10
7
    Dmu = np.linspace(-4,0)
9
10
    plt.plot(Dmu, np.zeros(Dmu.shape), label='Pt(111)')
11
    plt.plot(Dmu, fcc25 - 0.5*Dmu, label='fcc - 0.25 ML')
12
    plt.plot(Dmu, hcp25 - 0.5*Dmu, label='hcp - 0.25 ML')
13
    plt.plot(Dmu, bridge25 - 0.5*Dmu, label='bridge - 0.25 ML')
14
15
    plt.plot(Dmu, fcc1 - 0.5*Dmu, label='fcc - 1.0 ML')
16
17
    plt.xlabel('$\Delta \mu O_2$ (eV)')
    plt.ylabel('$\Delta G_{ads}$ (eV/0)')
18
    plt.legend(loc='best')
19
    plt.savefig('images/atomistic-thermo-adsorption.png')
```

None

7.3 Atomistic therodynamics and multiple reactions

In [37] 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.

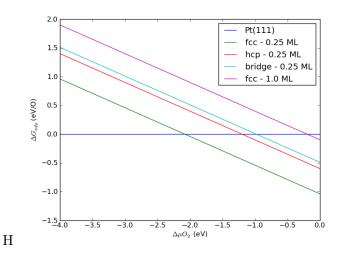


Figure 59: Effect of oxygen chemical potential on the adsorption energy.

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:

 $\mu_{gas}(T,p) = E_{gas}(0K) + \delta\mu + kT \ln(p/p^0)$, 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 [36] they use a U parameter of 4 eV for Cu which gave the best agreement with the experimental value. We will also try that.

Cu2O calculation with U=4.0

```
1
    from jasp import *
2
3
    from ase.visualize import *
    from ase import Atom, Atoms
4
    with jasp('bulk/Cu20') as calc:
6
7
         calc.clone('bulk/Cu20-U=4.0')
8
    with jasp('bulk/Cu20-U=4.0') as calc:
9
         calc.set(ldau=True, # turn DFT+U on
    ldautype=2, # select simplified rotationally invariant option
10
11
                   ldau_luj={'Cu':{'L':2, 'U':4.0, 'J':0.0},
'0':{'L':-1, 'U':0.0, 'J':0.0}},
12
13
                  ldauprint=1,
14
15
                   ibrion=-1, #do not rerelax
16
                  nsw=0)
         atoms = calc.get_atoms()
17
         print atoms.get_potential_energy()
18
19
         print calc
     -22.228203
       VASP calculation from /home/jkitchin/dft-org/bulk/Cu20-U=4.0
```

```
VASP calculation from /home/jkitchin/dft-org/bulk/Cu20-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
Unit cell volume = 77.854 Ang^3
Stress (GPa):xx,
                 уу,
                        zz,
                                    ΧZ,
                              yz,
                                          хy
          Atom#
               position [x,y,z]
                                     tag rmsForce
     sym
      Cu [0.000
 0
                                         0.00
                    0.000
                              0.000]
 1
      Cu [2.135
                    2.135
                              0.000]
                                         0.00
 2
      Cu [2.135
                    0.000
                              2.135]
                                         0.00
 3
      Cu [0.000
                    2.135
                              2.135]
                                      0.00
 4
         [1.067
                   1.067
                              1.067]
                                         0.00
      0
                                      0
 5
      0
         [3.202
                    3.202
                              3.202]
                                      0
                                         0.00
```

INCAR Parameters:

```
_____
```

nbands: 37 nsw: 0 ibrion: -1 ldautype: 2 isif: 3 ldauprint: 1 encut: 400.0 ldau: True ldaul: [-1.0, 2.0]

ldauj: [0.0, 0.0] ldauu: [0.0, 4.0]

ldau_luj: {'0': {'J': 0.0, 'U': 0.0, 'L': -1}, 'Cu': {'J': 0.0, 'U': 4.0, 'L':

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)

grep -A 3 "LDA+U is selected, type is set to LDAUTYPE" bulk/Cu20-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

```
from jasp import *
1
   from ase.visualize import *
   from ase import Atom, Atoms
   with jasp('bulk/CuO') as calc:
6
7
       calc.clone('bulk/Cu0-U=4.0')
8
   with jasp('bulk/Cu0-U=4.0') as calc:
       calc.set(ldau=True, # turn DFT+U on
    ldautype=2, # select simplified rotationally invariant option
10
11
               ldau_luj={'Cu':{'L':2, 'U':4.0, 'J':0.0},
12
                         '0':{'L':-1, 'U':0.0, 'J':0.0}},
13
               ldauprint=1,
               \verb|ibrion=-1|, & \textit{\#do not rerelax}|
15
               nsw=0)
16
       atoms = calc.get_atoms()
17
       print atoms.get_potential_energy()
18
       print calc
   -16.87013
    : ------
      VASP calculation from /home/jkitchin/dft-org/bulk/CuO-U=4.0
      converged: True
      Energy = -16.870130 \text{ eV}
      Unit cell vectors (angstroms)
                                       length
             х
                              z
                       У
      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,
                             уу,
                                     zz,
                                              yz,
                                                      ΧZ,
                                                               ху
                   0.030 0.018 0.027 -0.000 -0.007 -0.000
                         position [x,y,z]
     Atom# sym
                                                       tag rmsForce
```

```
Cu [1.151
               0.888
                         0.023]
                                0.00
1
   Cu [0.770
              -0.888
                         2.566]
                                0.00
2
   0 [2.111
              -0.168
                         1.318]
                                0
                                    0.04
3
   0 [1.730
               0.168
                         3.861]
                                0 0.04
```

INCAR Parameters:

```
nbands: 23
nsw: 0
ibrion: -1
ldautype: 2
isif: 3
ldauprint: 1
encut: 400.0
```

ldau: True 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':

prec: Normal
 kpts: [8 8 8]
reciprocal: False

xc: PBE txt: gamma: False

Pseudopotentials used:

0: 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. Cu20
# has 3 atoms, so the number of formula units in an atoms is
# len(atoms)/3.

with jasp('bulk/Cu20-U=4.0') as calc:
atoms = calc.get_atoms()
cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
```

```
9
    with jasp('bulk/Cu0-U=4.0') as calc:
10
        atoms = calc.get_atoms()
11
        cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
12
13
    # make sure to use the same cutoff energy for the O2 molecule!
14
    with jasp('molecules/02-sp-triplet-400') as calc:
15
        atoms = calc.get_atoms()
16
17
        o2_energy = atoms.get_potential_energy()
18
    rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
19
20
    print rxn_energy
    print rxn_energy - 1.36
21
```

-1.663819 -3.023819

This is still not in quantitative agreement with the result in [36], which at U=4 eV is about -3.14 eV (estimated from a graph). We have not applied the O₂ correction here yet. In that paper, they apply a constant shift of -1.36 eV per O₂. 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]:
2
        with jasp('bulk/Cu20') as calc:
4
            calc.clone('bulk/Cu20-U={0}'.format(U))
5
6
       with jasp('bulk/Cu20-U={0}'.format(U)) as calc:
7
            calc.set(ldau=True, # turn DFT+U on
8
                    {\tt ldautype=2, \# select \ simplified \ rotationally \ invariant \ option}
9
                    ldau_luj={'Cu':{'L':2, 'U':U, 'J':0.0},
'0':{'L':-1, 'U':0.0, 'J':0.0}},
10
11
                   ldauprint=1,
12
                   ibrion=-1, #do not rerelax
13
                   nsw=0)
14
15
            atoms = calc.get_atoms()
16
            cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
17
        18
        with jasp('bulk/Cu0') as calc:
19
            calc.clone('bulk/Cu0-U={0}'.format(U))
20
21
        with jasp('bulk/CuO-U={0}'.format(U)) as calc:
22
23
            calc.set(ldau=True, # turn DFT+U on
                    ldautype=2, # select simplified rotationally invariant option
24
```

```
ldau_luj={'Cu':{'L':2, 'U':U, 'J':0.0},
25
                              'O':{'L':-1, 'U':0.0, 'J':0.0}},
26
27
                   ldauprint=1,
                   ibrion=-1, #do not rerelax
28
29
                   nsw=0)
30
           atoms = calc.get_atoms()
31
           cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
32
        33
34
        # make sure to use the same cutoff energy for the O2 molecule!
       with jasp('molecules/02-sp-triplet-400') as calc:
35
36
           atoms = calc.get_atoms()
           o2_energy = atoms.get_potential_energy()
37
38
39
        rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
       print 'U = {0} reaction energy = {1}'.format(U,rxn_energy
40
```

```
U = 2.0 reaction energy = -3.876906

U = 4.0 reaction energy = -3.653819

U = 6.0 reaction energy = -3.397605
```

In [36], 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 $\rm O_2$ binding energy. In [36] the authors tried to get the $\rm O_2$ 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 [16] 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

```
1 from jasp import *
2 from ase.lattice.cubic import FaceCenteredCubic
```

```
from ase.dft import DOS
3
    atoms = FaceCenteredCubic(directions=[[0,1,1],
                                            [1,0,1],
 6
 7
                                            [1,1,0]],
                                            size=(1,1,1),
8
9
                                            symbol='Ni')
    atoms[0].magmom = 1
10
11
    with jasp('bulk/Ni-PBE',
12
               ismear=-5,
13
14
               kpts=(5,5,5),
               xc='PBE',
15
16
               ispin=2,lorbit=11,
17
               atoms=atoms) as calc:
18
         print 'PBE energy: ',atoms.get_potential_energy()
19
         dos = DOS(calc,width=0.2)
         e_pbe = dos.get_energies()
20
         d_pbe = dos.get_dos()
^{21}
22
         calc.clone('bulk/Ni-PBE0')
23
         calc.clone('bulk/Ni-HSE06')
24
25
26
    with jasp('bulk/Ni-PBEO') as calc:
         calc.set(lhfcalc=True,
27
                   algo='D',
28
                   time=0.4)
29
          atoms = calc.get_atoms()
30
31
          print 'PBEO energy: ',atoms.get_potential_energy()
          dos = DOS(calc,width=0.2)
32
33
          e_pbe0 = dos.get_energies()
          d_pbe0 = dos.get_dos()
34
35
    with jasp('bulk/Ni-HSE06') as calc:
36
37
         calc.set(lhfcalc=True,
38
                   hfscreen=0.2,
                   algo='D', time=0.4)
39
          atoms = calc.get_atoms()
40
          print 'HSE06 energy: ', atoms.get_potential_energy()
41
          dos = DOS(calc,width=0.2)
42
43
          e_hse06 = dos.get_energies()
         d_hse06 = dos.get_dos()
44
45
46
    import pylab as plt
    plt.plot(e_pbe, d_pbe, label='PBE')
47
    plt.plot(e_pbe0, d_pbe0, label='PBE0')
48
    plt.plot(e_hse06, d_hse06, label='HSE06')
49
50 plt.xlabel('energy [eV]')
51 plt.ylabel('DOS')
52
    plt.legend()
    plt.savefig('images/ni-dos-pbe-pbe0-hse06.png')
```

PBE energy: -5.530247 PBEO energy: -6.848931 HSE06 energy -6.293369

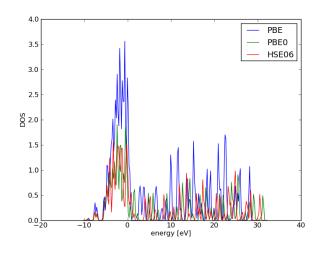


Figure 60: Comparison of DOS from GGA, and two hybrid GGAs (PBE0 ad HSE06).

8.3 TODO vdW

Η

 $http://cms.mpi.univie.ac.at/vasp/vasp/vdW_DF_functional_Langreth_Lundqvist_et_al.html$

8.4 TODO DFT+D

http://cms.mpi.univie.ac.at/vasp/vasp/DFT_D2_method_Grimme.html [21]

8.5 ELF

Need better intro here.

```
# compute ELF for CF4
1
    from jasp import *
    from ase.data.molecules import molecule
3
    from enthought.mayavi import mlab
    atoms = molecule('CF4')
6
7
    atoms.center(vacuum=5)
8
    with jasp('molecules/cf4-elf',
9
10
              encut=350,
              prec='high',
11
12
               ismear=0,
              sigma=0.01,
13
               xc='PBE',
14
              lelf=True,
15
```

```
atoms=atoms) as calc:
16
17
          calc.calculate()
18
          x,y,z,elf = calc.get_elf()
19
          mlab.contour3d(x,y,z,elf,contours=[0.3])
20
          mlab.savefig('../../images/cf4-elf-3.png')
21
22
          mlab.figure()
23
          mlab.contour3d(x,y,z,elf,contours=[0.75])
mlab.savefig('../../images/cf4-elf-75.png')
24
25
```

None

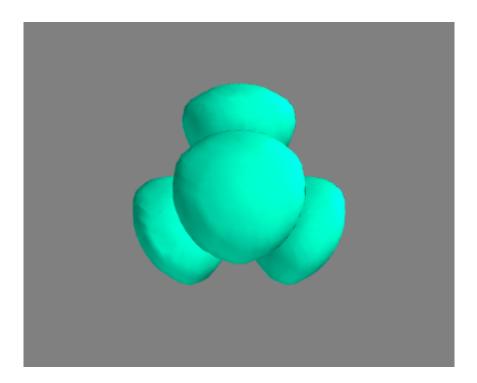


Figure 61: ELF for an isosurface of 0.3 for CF_4 .

These images (Figure 61 and 62) are basically consistent with those in Reference [31].

8.6 TODO Charge partitioning schemes

8.7 TODO Modeling Core level shifts

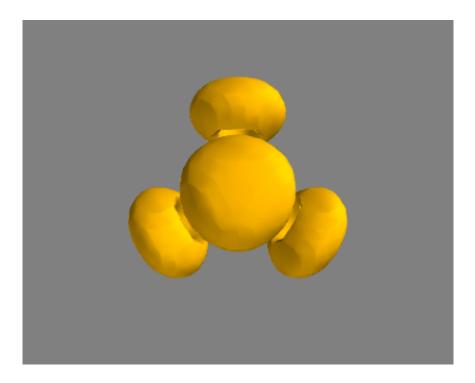


Figure 62: ELF for an isosurface of 0.75 for CF₄.

9 Acknowledgments

I would like to thank Zhongnan Xu for sending me some examples on magnetism. Alan McGaughey for sending me some NEB examples.

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')

g print atoms

```
4
5    ind2del = [atom.index for atom in atoms if atom.symbol=='H']
6    print ind2del
7
8    del atoms[ind2del]
9    print atoms
```

```
Atoms(symbols='C20H6', positions=..., cell=[1.0, 1.0, 1.0], pbc=[False, False, [3, 4, 5, 6, 7, 8]
Atoms(symbols='C20', 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] 

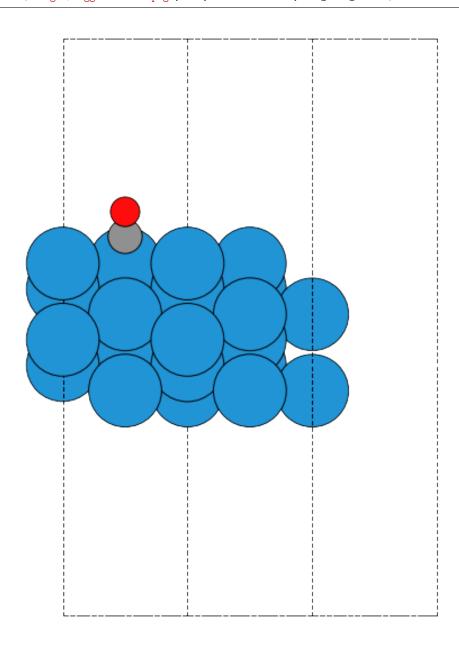
8 = [0 0 0 1]
```

so, an atom tagged with 1 and 2 would have a tag of $[1\ 1\ 0\ 0]$ or a tag of 3.

```
1
    adapted from https://listserv.fysik.dtu.dk/pipermail/campos/2004-September/001155.html
2
3
5
    from ase import *
6
    from ase.io import write
    from ase.lattice.surface import *
7
    from ase.constraints import FixAtoms
8
    # the bcc111 function automatically tags atoms
10
11
    slab = bcc111('W',
                   a=3.92.
                                 # W lattice constant
12
13
                   size=(2,2,6), #6-layer slab in 2x2 configuration
14
                   vacuum=10.0)
15
    #reset tags to be powers of two
```

```
slab.set_tags([2**a.get_tag() for a in slab])
17
18
    # we had 6 layers, so we create new tags starting at 7
19
   # Note you must use powers of two for all the tags!
20
21 LAYER1 = 2
22 ADSORBATE = 2**7
    FREE = 2**8
23
    NEARADSORBATE = 2**9
24
25
26
   # let's tag LAYER1 atoms to be FREE too. we can address it by LAYER1 or FREE
27
28
    tags = slab.get_tags()
    for i,tag in enumerate(tags):
29
30
        if tag == LAYER1:
31
            tags[i] += FREE
32
    slab.set_tags(tags)
33
    #create a CO molecule
34
    co= Atoms([Atom('C',[0., 0., 0.], tag=ADSORBATE),
35
               Atom('0',[0., 0., 1.1], tag=ADSORBATE+FREE)]) #we will relax only O
36
37
    add_adsorbate(slab,co,height=1.2,position='hollow')
38
39
   #the adsorbate is centered between atoms 20, 21 and 22 (use
40
    #view(slab)) and over atom12 lets label those atoms, so it is easy to
41
    #do electronic structure analysis on them later.
42
    tags = slab.get_tags() # len(tags) changed, so we reget them.
43
    tags[12]+=NEARADSORBATE
44
45 tags[20]+=NEARADSORBATE
    tags[21]+=NEARADSORBATE
46
    tags[22]+=NEARADSORBATE
47
    slab.set_tags(tags)
48
49
    #update the tags
50
   slab.set_tags(tags)
51
    #extract pieces of the slab based on tags
52
    #atoms in the adsorbate
53
    ads = slab[(slab.get_tags() & ADSORBATE) == ADSORBATE]
54
55
    #atoms in LAYER1
56
57
    layer1 = slab[(slab.get_tags() & LAYER1) == LAYER1]
58
    #atoms defined as near the adsorbate
59
    nearads = slab[(slab.get_tags() & NEARADSORBATE) == NEARADSORBATE]
60
61
62
    #atoms that are free
    free = slab[(slab.get_tags() & FREE) == FREE]
63
    #atoms that are FREE and part of the ADSORBATE
65
    freeads = slab[(slab.get_tags() & FREE+ADSORBATE) == FREE+ADSORBATE]
66
67
   #atoms that are NOT FREE
68
   notfree = slab[(slab.get_tags() & FREE) != FREE]
69
70
    constraint = FixAtoms(mask = (slab.get_tags() & FREE) != FREE)
71
72
    slab.set_constraint(constraint)
```





10.1.3 Using units in ase

mod: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 *
1
2
    d = 1*Angstrom
3
    print d/nm
4
6
    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))
10
    print '1 Hartree = %f kcal/mol' % (1*Hartree/(kcal/mol))
11
    print '1 Rydberg = %f eV' % (1*Rydberg/eV)
12
13
    # derived units
14
    minute = 60*s
15
16
    hour = 60*minute
17
   #convert 10 hours to minutes
18
    print 10*hour/minute
```

```
0.1

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

600.0
```

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 $2 \mid x \mid 4$, 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
5
    from pylab import *
    x = np.linspace(0,6,100)
7
    y = np.cos(x)
9
    plot(x,y,label='full')
10
11
    ind = (x>2) & (x<4)
12
13
    subx = x[ind]
14
    suby = y[ind]
15
16
    plot(subx,suby,'bo',label='sliced')
17
    xlabel('x')
18
19
    ylabel('cos(x)')
    legend(loc='lower right')
20
    savefig('images/np-array-slice.png')
```

None

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

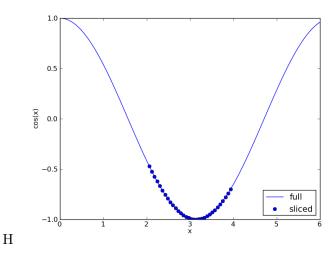


Figure 63: Example of slicing out part of an array. The solid line represents the whole array, and the symbols are the array between 2 < x < 4.

```
from numpy import *
1
    from scipy.stats.distributions import t
2
3
    n = 10 #number of measurements
    dof = n - 1 #degrees of freedom
5
    avg_x = 16.1 #average measurement
6
    std_x = 0.01 #standard deviation of measurements
7
8
    #Find 95% prediction interval for next measurement
9
10
11
    alpha = 1.0 - 0.95
12
    pred_interval = t.ppf(1-alpha/2.,dof)*std_x*sqrt(1.+1./n)
13
14
    s = ['We are 95\%' confident the next measurement',
15
16
           ' will be between %1.3f and %1.3f']
    print ''.join(s) % (avg_x - pred_interval, avg_x + pred_interval)
17
```

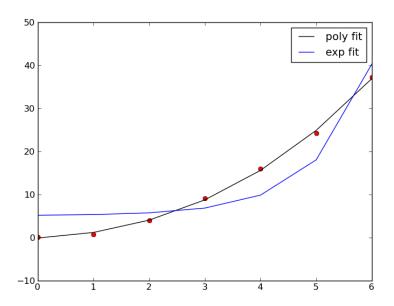
We are 95% confident the next measurement will be between 16.076 and 16.124

10.1.6 Curve fitting

Linear fitting

```
#examples of linear curve fitting using least squares
import numpy as np
```

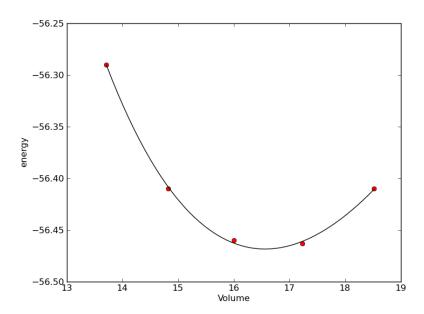
```
3
    xdata = np.array([0.,1.,2.,3.,4.,5.,6.])
4
    ydata = np.array([0.1, 0.81, 4.03, 9.1, 15.99, 24.2, 37.2])
   #fit a third order polynomial
   from pylab import polyfit, plot, show, legend, savefig
8
9
    pars = polyfit(xdata,ydata,3)
10
    print pars
11
12 ## numpy method returns more data
A = np.column_stack([xdata**3,xdata**2,xdata,np.ones(len(xdata),np.float)])
14
    pars_np,resids,rank,s = np.linalg.lstsq(A,ydata)
    print pars_np
15
16
17
18
    we are trying to solve Ax = b for x in the least squares sense. There
19
    are more rows in A than elements in x so, we can left multiply each
20 side by A^T, and then solve for x with an inverse.
22 \qquad A^TAx = A^Tb
    x = (A^TA)^{-1} A^T b
23
24
25 # not as pretty but equivalent!
pars_man= np.dot(np.linalg.inv(np.dot(A.T,A)), np.dot(A.T,ydata))
27 print pars_man
28
29 #but, it is easy to fit an exponential function to it!
30 	 # y = a*exp(x)+b
31 Aexp = np.column_stack([np.exp(xdata), np.ones(len(xdata),np.float)])
    pars_exp=np.dot(np.linalg.inv(np.dot(Aexp.T,Aexp)), np.dot(Aexp.T,ydata))
32
33
    plot(xdata,ydata,'ro')
34
35 fity = np.dot(A,pars)
36 plot(xdata,fity,'k-',label='poly fit')
plot(xdata,np.dot(Aexp,pars_exp),'b-',label='exp fit')
legend()
39 savefig('images/curve-fit-1.png')
      [ 0.04861111  0.63440476  0.61365079  -0.08928571]
```



10.1.7 Nonlinear curve fitting

```
1
    from scipy.optimize import leastsq
    import numpy as np
2
3
    vols = np.array([13.71, 14.82, 16.0, 17.23, 18.52])
5
    energies = np.array([-56.29, -56.41, -56.46, -56.463,-56.41])
6
    def Murnaghan(parameters,vol):
8
         'From PRB 28,5480 (1983'
9
         E0 = parameters[0]
10
        B0 = parameters[1]
BP = parameters[2]
11
12
13
        V0 = parameters[3]
14
15
         E = E0 + B0*vol/BP*(((VO/vol)**BP)/(BP-1)+1) - V0*BO/(BP-1.)
16
        return E
17
18
    def objective(pars,y,x):
19
         #we will minimize this function
20
         err = y - Murnaghan(pars,x)
^{21}
        return err
22
23
    x0 = [-56., 0.54, 2., 16.5] #initial guess of parameters
24
25
    plsq = leastsq(objective, x0, args=(energies,vols))
26
```

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



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]
10
        B0 = parameters[1]
11
       BP = parameters[2]
12
        V0 = parameters[3]
13
14
        E = E0 + B0*vol/BP*(((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
15
16
        return E
17
18
19
    def objective(pars,vol):
        #we will minimize this function
20
21
        err = energies - Murnaghan(pars,vol)
        return np.sum(err**2) #we return the summed squared error directly
22
23
    x0 = [-56., 0.54, 2., 16.5] #initial guess of parameters
24
25
26
    plsq = fmin(objective,x0,args=(volumes,)) #note args is a tuple
27
    print plsq
28
29
    from pylab import *
30
31
    plot(volumes, energies, 'ro')
32
   #plot the fitted curve on top
   x = np.linspace(min(volumes), max(volumes), 50)
34
    y = Murnaghan(plsq,x)
35
    plot(x,y,'k-')
36
    savefig('images/nonlinear-fitting-lsq.png')
     Optimization terminated successfully.
                 Current function value: 0.000020
                 Iterations: 137
                 Function evaluations: 240
      [-56.46932645
                          0.59141447
                                           1.9044796
                                                           16.593413037
```

10.1.9 Nonlinear curve fitting with confidence intervals

```
# Nonlinear curve fit with confidence interval
2 from numpy import *
    from scipy.optimize import curve_fit
3
    from scipy.stats.distributions import t
5
6
    fit this equation to data
7
    y = c1 exp(-x) + c2*x
8
10
    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.
12
    from Matlab
```

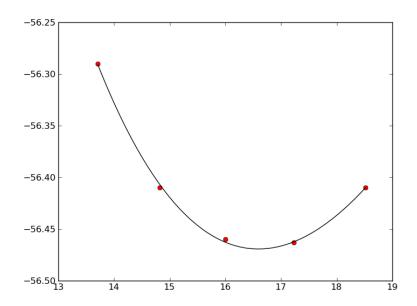


Figure 64: Fitting a nonlinear function.

```
b =
15
16
       4.9671
^{17}
       2.1100
18
19
20
21
    bint =
22
23
       4.6267
                5.3075
                2.4528
       1.7671
^{24}
25
26
   27
28
29
30
    \# this is the function we want to fit to our data
31
    def func(x,c0, c1):
32
33
       return c0*exp(-x) + c1*x
34
   pars, pcov = curve_fit(func, x, y, p0=[4.96, 2.11])
35
36
   alpha = 0.05 # 95% confidence interval
37
38
   n = len(y)
              # number of data points
39
40
   p = len(pars) # number of parameters
```

```
41
42
    dof = max(0, n-p) # number of degrees of freedom
43
    tval = t.ppf(1.0-alpha/2., dof) # student-t value for the dof and confidence level
44
45
    for i, p,var in zip(range(n), pars, diag(pcov)):
46
47
         sigma = var**0.5
         print 'c{0}: {1} [{2} {3}]'.format(i, p,
48
                                         p - sigma*tval,
49
                                         p + sigma*tval)
50
51
52
    import matplotlib.pyplot as plt
    plt.plot(x,y,'bo ')
53
54
    xfit = linspace(0,1)
    yfit = func(xfit, pars[0], pars[1])
55
    plt.plot(xfit,yfit,'b-')
plt.legend(['data','fit'],loc='best')
56
57
    plt.savefig('images/nonlin-fit-ci.png')
```

c0: 4.96713966556 [4.62674477087 5.30753456025] c1: 2.10995112466 [1.76711622383 2.45278602549]

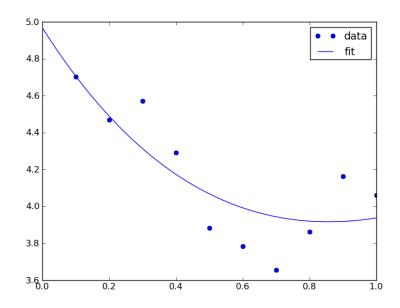


Figure 65: Nonlinear fit to data.

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
1
    from scipy.interpolate import interp1d
    from scipy.optimize import fmin
    import numpy as np
    import matplotlib.pyplot as plt
6
    x = np.array([ 0,
8
                           1,
                                    2,
                                            3,
                           0.308, 0.55,
                                          0.546, 0.44])
9
    y = np.array([0.,
10
    # create the interpolating function
11
12
    f = interp1d(x, y, kind='cubic', bounds_error=False)
13
    # to find the maximum, we minimize the negative of the function. We cannot just multiply f by -1, so we create a new
14
    f2 = interp1d(x, -y, kind='cubic')
15
16
    xmax = fmin(f2, 2.5)
17
    xfit = np.linspace(0,4)
18
19
    plt.plot(x,y,'bo')
20
    plt.plot(xfit, f(xfit),'r-')
21
22
    plt.plot(xmax, f(xmax),'g*')
    plt.legend(['data','fit','max'], loc='best', numpoints=1)
23
    plt.xlabel('x data')
    plt.ylabel('y data')
25
26
    plt.title('Max point = ({0:1.2f}, {1:1.2f})'.format(float(xmax),
                                                         float(f(xmax))))
27
    plt.savefig('images/splinefit.png')
28
```

There are other good examples at http://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.htm

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

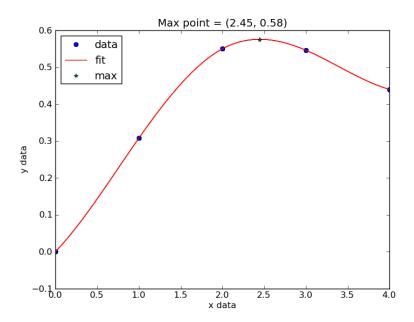


Figure 66: Illustration of a spline fit to data and finding the maximum point.

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 67.

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

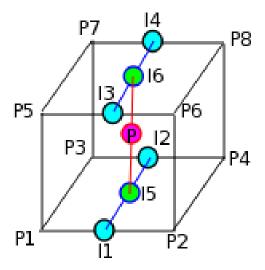


Figure 67: Trilinear interpolation scheme.

evaluated at z=c to get the final value of the scalar field at P=(a,b,c).

```
from jasp import *
1
2
    from ase.calculators.vasp import *
    with jasp('molecules/co-centered') as calc:
4
        atoms = calc.get_atoms()
6
        vcd = VaspChargeDensity()
7
8
9
        cd = np.array(vcd.chg[0])
        n0, n1, n2 = cd.shape
10
11
         s0 = 1.0/n0
12
        s1 = 1.0/n1
13
        s2 = 1.0/n2
14
15
```

```
X, Y, Z = np.mgrid[0.0:1.0:s0,
16
                             0.0:1.0:s1,
17
                             0.0:1.0:s2]
18
19
         C = np.column_stack([X.ravel(),
20
                               Y.ravel(),
21
22
                               Z.ravel()])
23
         atoms = calc.get_atoms()
24
25
         uc = atoms.get_cell()
        real = np.dot(C, uc)
26
27
         #now convert arrays back to unitcell shape
28
29
        x = np.reshape(real[:, 0], (n0, n1, n2))
         y = np.reshape(real[:, 1], (n0, n1, n2))
30
         z = np.reshape(real[:, 2], (n0, n1, n2))
31
32
    def interp3d(x,y,z,cd,xi,yi,zi):
33
34
         interpolate a cubic 3D grid defined by x,y,z,cd at the point
35
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):
                 if val > value:
45
46
                     return i-1
             return None
47
48
         xv = x[:,0,0]
49
50
        yv = y[0,:,0]
51
        zv = z[0,0,:]
52
        a,b,c = xi, yi, zi
53
54
        i = get_index(a,xv)
55
        j = get_index(b,yv)
k = get_index(c,zv)
56
57
58
59
         x1 = x[i,j,k]
60
        x2 = x[i+1,j,k]
61
        y1 = y[i,j,k]
        y2 = y[i,j+1,k]
62
        z1 = z[i,j,k]
63
        z2 = z[i,j,k+1]
64
65
         u1 = cd[i, j, k]
66
        u2 = cd[i+1, j, k]
67
         u3 = cd[i, j+1, k]
        u4 = cd[i+1, j+1, k]
69
70
        u5 = cd[i, j, k+1]
        u6 = cd[i+1, j, k+1]
71
```

```
u7 = cd[i, j+1, k+1]
72
         u8 = cd[i+1, j+1, k+1]
73
74
         w1 = u2 + (u2-u1)/(x2-x1)*(a-x2)
75
76
         w2 = u4 + (u4-u3)/(x2-x1)*(a-x2)
         w3 = w2 + (w2-w1)/(y2-y1)*(b-y2)
77
         w4 = u5 + (u6-u5)/(x2-x1)*(a-x1)
78
         w5 = u7 + (u8-u7)/(x2-x1)*(a-x1)
79
         w6 = w4 + (w5-w4)/(y2-y1)*(b-y1)
80
81
         w7 = w3 + (w6-w3)/(z2-z1)*(c-z1)
82
83
         return u
84
85
86
     pos = atoms.get_positions()
87
88
     P1 = np.array([0.0, 5.0, 5.0])
     P2 = np.array([9.0, 5.0, 5.0])
89
90
     npoints = 60
91
92
     points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]
93
94
     R = [np.linalg.norm(p-P1) for p in points]
95
96
97
     icd = [interp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]
98
99
100
     from pylab import *
101
     plot(R,icd)
102
     cR = np.linalg.norm(pos[0]-P1)
103
     oR = np.linalg.norm(pos[1]-P1)
104
     plot([cR,cR],[0,2],'r-') #markers for where the nuclei are
105
     plot([oR,oR],[0,8],'r-')
106
     xlabel('|R| ($\AA$)')
107
     ylabel('Charge density (e/$\AA^3$)')
108
     savefig('images/CO-charge-density.png')
109
110
```

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.

^{1 &#}x27;''2 3D vector interpolation in non-cubic unit cells with vector

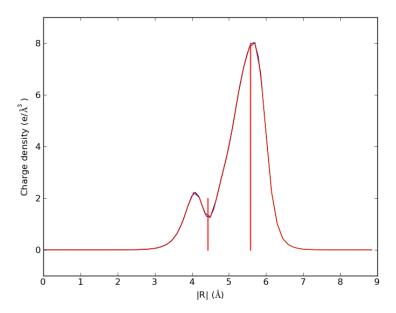


Figure 68: An example of interpolated charge density of A CO molecule along the axis of molecule.

```
interpolation.
3
4
    This function should work for any shape unit cell.
5
6
7
    from ase.calculators.jacapo import Jacapo
    import bisect
8
9
    import numpy as np
10
    atoms = Jacapo.read_atoms('/home/jkitchin/www/dft/source/jacapo/1-molecules/1.2-calculating-simple-properties/1.2.6
11
12
    calc = atoms.get_calculator()
    x,y,z,cd = calc.get_charge_density()
13
14
    def vinterp3d(x,y,z,u,xi,yi,zi):
15
16
         p = np.array([xi,yi,zi])
^{17}
18
         #1D arrays of cooridinates
19
        xv = x[:,0,0]
20
        yv = y[0,:,0]
21
        zv = z[0,0,:]
22
23
24
         # we subtract 1 because bisect tells us where to insert the
25
         # element to maintain an ordered list, so we want the index to the
26
         # left of that point
         i = bisect.bisect_right(xv,xi) - 1
27
```

```
j = bisect.bisect_right(yv,yi) - 1
28
         k = bisect.bisect_right(zv,zi) - 1
29
30
         #points at edge of cell. We only need P1, P2, P3, and P5
31
        P1 = np.array([x[i,j,k],y[i,j,k],z[i,j,k]])
32
        P2 = np.array([x[i+1,j,k],y[i+1,j,k],z[i+1,j,k]])
33
34
        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]])
35
36
37
         #values of u at edge of cell
        u1 = u[i,j,k]
38
39
        u2 = u[i+1,j,k]
        u3 = u[i,j+\bar{1},k]
40
41
        u4 = u[i+1, j+1, k]
42
        u5 = u[i,j,k+1]
        u6 = u[i+1,j,k+1]
43
44
        u7 = u[i,j+1,k+1]
        u8 = u[i+1,j+1,k+1]
45
46
         #cell basis vectors, not the unit cell, but the voxel cell containing the point
47
         cbasis = np.array([P2-P1,
48
49
                            P3-P1,
                            P5-P1])
50
51
         #now get interpolated point in terms of the cell basis
52
         s = np.dot(np.linalg.inv(cbasis.T),np.array([xi,yi,zi])-P1)
53
54
         #now s = (sa, sb, sc) which are fractional coordinates in the vector space
55
56
         #next we do the interpolations
        ui1 = u1 + s[0]*(u2-u1)
57
        ui2 = u3 + s[0]*(u4-u3)
58
59
        ui3 = u5 + s[0]*(u6-u5)
60
        ui4 = u7 + s[0]*(u8-u7)
61
62
63
        ui5 = ui1 + s[1]*(ui2-ui1)
        ui6 = ui3 + s[1]*(ui4-ui3)
64
65
        ui7 = ui5 + s[2]*(ui6-ui5)
66
67
68
         return ui7
69
70
    # compute a line with 60 points in it through these two points
71
    P1 = np.array([0.0, 3.0, 3.0])
72
    P2 = np.array([5.0, 3.0, 3.0])
73
74
    npoints = 60
75
76
    points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]
77
78
    # compute the distance along the line
79
80
    R = [np.linalg.norm(p-P1) for p in points]
81
    icd = [vinterp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]
```

83

```
from pylab import plot, xlabel, ylabel, show
84
85
86
    plot(R,icd)
87
88
   pos = atoms.get_positions()
   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
92 plot([oR,oR],[0,8],'r-')
93 xlabel('|R| ($\AA$)')
    ylabel('Charge density (e/$\AA^3$)')
94
```

None

10.1.12 Reading and writing data

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

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

and later you can read these arrays back in with:

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

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

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

Reading a datafile with

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

A standard approach would be to read in all the lines, skip the first two lines, split each line (remember each line is a string) at the ',', and append the first field to one variable, and append the second field to another variable as an integer. For example:

```
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: func: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 moments 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 func: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
1
   from pylab import *
    import time
3
5
    These are the brainless way to calculate numerical derivatives. They
6
    work well for very smooth data. they are surprisingly fast even up to
    10000 points in the vector.
8
9
10
    x = np.linspace(0.78, 0.79, 100)
11
12
    y = np.sin(x)
    dy_analytical = np.cos(x)
13
14
    lets use a forward difference method:
15
16
    that works up until the last point, where there is not
17
    a forward difference to use. there, we use a backward difference.
18
19
    tf1 = time.time()
20
    dyf = [0.0]*len(x)
21
22
    for i in range(len(y)-1):
        dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
23
24
    #set last element by backwards difference
    dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
25
26
    print 'Forward difference took %1.1f seconds' % (time.time() - tf1)
27
28
29
    '''and now a backwards difference'''
   tb1 = time.time()
30
    dyb = [0.0]*len(x)
   #set first element by forward difference
32
33
    dyb[0] = (y[0] - y[1])/(x[0] - x[1])
34
    for i in range(1,len(y)):
        dyb[i] = (y[i] - y[i-1])/(x[i]-x[i-1])
35
36
    print 'Backward difference took %1.1f seconds' % (time.time() - tb1)
37
38
    '''and now, a centered formula'''
39
40
    tc1 = time.time()
41
    dyc = [0.0]*len(x)
    dyc[0] = (y[0] - y[1])/(x[0] - x[1])
42
    for i in range(1,len(y)-1):
43
        dyc[i] = (y[i+1] - y[i-1])/(x[i+1]-x[i-1])
44
    dyc[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
45
46
    print 'Centered difference took %1.1f seconds' % (time.time() - tc1)
47
48
49
50
    the centered formula is the most accurate formula here
51
52
53
    plot(x,y)
    plot(x,dy_analytical,label='analytical derivative')
54
55 plot(x,dyf,'--',label='forward')
56 plot(x,dyb,'--',label='backward')
```

```
57 plot(x,dyc,'--',label='centered')
58
59 legend(loc='lower left')
60 savefig('images/simple-diffs.png')
61 show()
```

Forward difference took 0.0 seconds Backward difference took 0.0 seconds Centered difference took 0.0 seconds

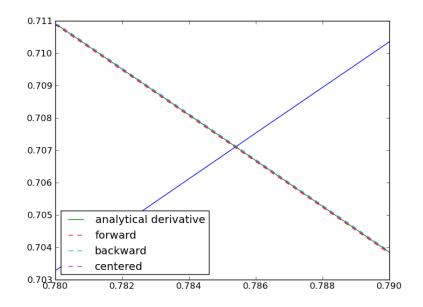


Figure 69: Comparison of different numerical derivatives.

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:

```
import numpy as np
from pylab import *

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

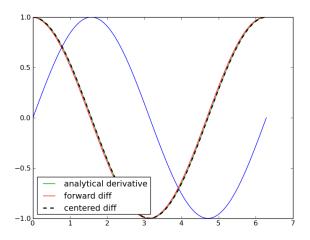


Figure 70: Comparison of different numerical derivatives.

```
# we need to specify the size of dy ahead because diff returns
8
    #an array of n-1 elements
9
    dy = np.zeros(y.shape,np.float) #we know it will be this size
10
11
    dy[0:-1] = np.diff(y)/np.diff(x)
    dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
12
13
14
15
16
    calculate dy by center differencing using array slices
17
18
    dy2 = np.zeros(y.shape,np.float) #we know it will be this size
19
20
    dy2[1:-1] = (y[2:] - y[0:-2])/(x[2:] - x[0:-2])
21
    dy2[0] = (y[1]-y[0])/(x[1]-x[0])
    dy2[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
22
23
    plot(x,y)
24
    plot(x,dy_analytical,label='analytical derivative')
25
    plot(x,dy,label='forward diff')
26
    plot(x,dy2,'k--',lw=2,label='centered diff')
27
28
    legend(loc='lower left')
    savefig('images/vectorized-diffs.png')
29
    show()
```

None

Η

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 *
2
    x = np.linspace(0,2*np.pi,100)
4
    y = np.sin(x) + 0.1*np.random.random(size=x.shape)
5
    dy_analytical = np.cos(x)
    #2-point formula
    dyf = [0.0]*len(x)
9
    for i in range(len(y)-1):
10
        dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
11
    #set last element by backwards difference
12
    dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
13
14
15
    calculate dy by 4-point center differencing using array slices
16
17
18
    \frac{y[i-2] - 8y[i-1] + 8[i+1] - y[i+2]}{12h}
19
20
    y[0] and y[1] must be defined by lower order methods
    and y[-1] and y[-2] must be defined by lower order methods
21
22
23
    dy = np.zeros(y.shape,np.float) #we know it will be this size
24
25
    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)
26
    dy[0] = (y[1]-y[0])/(x[1]-x[0])
28
29
    dy[1] = (y[2]-y[1])/(x[2]-x[1])
    dy[-2] = (y[-2] - y[-3])/(x[-2] - x[-3])
30
    dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
31
33
    plot(x,y)
    plot(x,dy_analytical,label='analytical derivative')
34
    plot(x,dyf,'r-',label='2pt-forward diff')
35
    plot(x,dy,'k--',lw=2,label='4pt-centered diff')
36
    legend(loc='lower left')
    savefig('images/multipt-diff.png')
38
    show()
```

None

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:

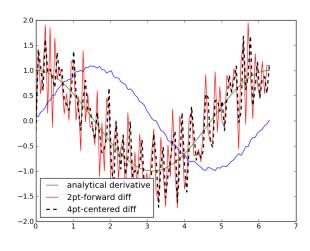


Figure 71: Comparison of 2 point and 4 point numerical derivatives.

Η

```
import numpy as np
1
    from pylab import *
2
3
    N = 101 #number of points
4
    L = 2*np.pi #interval of data
5
    x = np.arange(0.0,L,L/float(N)) #this does not include the endpoint
7
8
    #add some random noise
9
    y = np.sin(x) + 0.05*np.random.random(size=x.shape)
10
11
    dy_analytical = np.cos(x)
12
13
    http://sci.tech-archive.net/Archive/sci.math/2008-05/msg00401.html
14
15
    you can use fft to calculate derivatives!
16
17
18
    if N % 2 == 0:
19
20
        k = np.asarray(range(0,N/2)+[0] + range(-N/2+1,0))
21
        k = np.asarray(range(0,(N-1)/2) + [0] + range(-(N-1)/2,0))
22
23
    k = 2*np.pi/L
24
25
    fd = np.fft.ifft(1.j*k * np.fft.fft(y))
26
27
    plot(x,y)
^{28}
    plot(x,dy_analytical,label='analytical der')
29
    plot(x,fd,label='fft der')
    legend(loc='lower left')
```

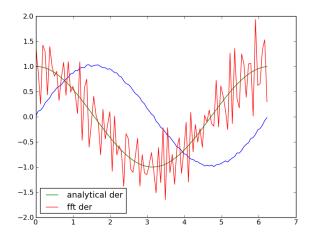


Figure 72: Comparison of FFT numerical derivatives.

```
32
33 savefig('images/fft-der.png')
34 show()
```

None

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

Η

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

```
(http://dirac.cnrs-orleans.fr/plone/software/scientificpython/)
:mod: 'netCDF3' (http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF3-module.html)
```

:mod:'pycdf' (http://pysclint.sourceforge.net/pycdf/) this is a very low level module modelled after the C-api. I am not sure it is completely bug-free (I have problems with character variables) I suggest you read through the jacapo.py code for many examples of reading and writing NetCDF files.

10.1.16 python modules

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

csv writing:

```
import numpy as np

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

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

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

csv reading:

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

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

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

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

10.1.17 TODO reading and writing excel files

10.1.18 TODO making movies

1. using animate

2. using swftools (png2swf, pdf2swf)

http://wiki.swftools.org/wiki/Main_Page#SWF_Tools_0.9.2_.28_Current_Stable_Version_.29_Document

10.2 Computational geometry

10.2.1 Changing coordinate systems

Let A, B, C be the unit cell vectors

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

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

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

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

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

where

1.
$$P = [p1x + p2y + p3z]$$

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

•
$$s2 B1 x + s2 B2 y + s2 B3 z$$

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

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

$$A1 B1 C1$$
 $s1$ $p1$
 $A2 B2 C2$ $s2$ $=$ $p2$
 $A3 B3 C3$ $s3$ $p3$

or, in standard form:

transpose(A) s = p

and we need to solve for s as:

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

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

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

Here we implement that in code:

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

```
1
    from ase.lattice.cubic import BodyCenteredCubic
    from ase.calculators.jacapo import *
2
    bulk = BodyCenteredCubic(directions=[[1,0,0],
4
                                           [0,1,0],
                                           [0,0,1]],
6
7
                              size=(2,2,2),
8
                              latticeconstant=2.87,
9
                              symbol='Fe')
10
11
    newbasis = 2.87*np.array([[-0.5, 0.5, 0.5],
12
                                [0.5, -0.5, 0.5],
13
                               [0.5, 0.5, -0.5]])
14
15
16
    pos = bulk.get_positions()
^{17}
18
    s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
    print 'atom positions in primitive basis
19
20
    print s
21
22
    #let's see the unit cell in terms of the primitive basis too
    print 'unit cell in terms of the primitive basis'
23
    print np.dot(np.linalg.inv(newbasis.T),bulk.get_cell().T).T
```

atom positions in primitive basis

```
[[ 0.
       0.
            0.]
 [ 1.
       1.
            1.]
 Γ0.
       1.
            1.]
 Г1.
       2.
            2.1
 [ 1.
       0.
            1.]
 [ 2.
       1.
            2.]
 Г1.
       1.
            2.]
 [ 2.
       2.
            3.]
 Г1.
       1.
            0.1
 [ 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.

```
1 import numpy as np
   from Scientific.Geometry import *
   A = Vector([1,1,1]) #Scientfic
4
   a = np.array([1,1,1]) #numpy
   B = Vector([0.0, 1.0, 0.0])
   print '|A| = ',A.length()
9
                               #Scientific Python way
   print '|a| = ',np.sum(a**2)**0.5 #numpy way
10
   print '|a| = ',np.linalg.norm(a) #numpy way 2
11
   print 'ScientificPython angle = ',A.angle(B) #in radians
13
14
   print 'numpy angle =
                                ',np.arccos(np.dot(a/np.linalg.norm(a),B/np.linalg.norm(B)))
15
  #cross products
16
   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. :mod:'ase' implements a convenient function to get the volume of an :mod:'Atoms' object: :func:'ase.atoms.get_volume'.

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

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

20

20.0

20.0

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.

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

```
4
    ag = FaceCenteredCubic(directions=[[1,0,0],
5
6
                                       [0,1,0],
                                       [0,0,1]],
7
                           size=(1,1,1),
 8
                           symbol='Ag',
9
10
                           latticeconstant=4.0)
11
    # these are the reciprocal lattice vectors
12
13
   b1,b2,b3 = np.linalg.inv(ag.get_cell())
14
15
    g(111) = 1*b1 + 1*b2 + 1*b3
16
17
    and |g(111)| = 1/d_1111
18
19
20
    h,k,l = (1,1,1)
    d = 1./np.linalg.norm(h*b1 + k*b2 + 1*b3)
21
    print 'd_111 spacing = %1.3f Angstroms' % d
23
24
    #method #2
25
    hkl = np.array([h,k,1])
26
27
    G = np.array([b1,b2,b3]) #reciprocal unit cell
28
29
    Gstar is usually defined as this matrix of dot products:
30
31
    Gstar = np.array([[dot(b1,b1), dot(b1,b2), dot(b1,b3)],
32
                      [dot(b1,b2), dot(b2,b2), dot(b2,b3)],
[dot(b1,b3), dot(b2,b3), dot(b3,b3)]])
33
34
35
36
    but I prefer the notationally more compact:
    Gstar = G .dot. transpose(G)
37
38
39
    then, 1/d_hkl^2 = hkl \cdot dot \cdot Gstar \cdot dot \cdot hkl
40
41
    Gstar = np.dot(G,G.T)
42
43
44
    id2 = np.dot(hkl,np.dot(Gstar,hkl))
45
    print np.sqrt(1/id2)
46
^{47}
    48
49
50
    '''Finally, many text books on crystallography use long algebraic
51
    formulas for computing the d-spacing with sin and cos, vector lengths,
52
    and angles. Below we compute these and use them in the general
53
    triclinic structure formula which applies to all the structures.
54
55
56
   from Scientific.Geometry import Vector
   import math
57
59
    unitcell = ag.get_cell()
```

```
A = Vector(unitcell[0])
60
    B = Vector(unitcell[1])
61
62 C = Vector(unitcell[2])
63
64
    # lengths of the vectors
    a = A.length()#*angstroms2bohr
65
66
    b = B.length()#*angstroms2bohr
67
    c = C.length()#*angstroms2bohr
68
69
    # angles between the vectors in radians
    alpha = B.angle(C)
70
71
    beta = A.angle(C)
    gamma = A.angle(B)
72
73
    print '%1.3f %1.3f %1.3f %1.3f %1.3f \n' % (a,b,c,
74
75
                                                        alpha, beta, gamma)
76
    h,k,l = (1,1,1)
77
78
    from math import sin, cos
79
80
    id2 = ((h**2/a**2*sin(alpha)**2
81
           + k**2/b**2*sin(beta)**2
82
            + 1**2/c**2*sin(gamma)**2
            +2*k*1/b/c*(cos(\overline{beta})*cos(gamma)-cos(alpha))
84
            +2*h*1/a/c*(cos(alpha)*cos(gamma)-cos(beta))
85
            +2*h*k/a/b*(cos(alpha)*cos(beta)-cos(gamma)))
86
            /(1-\cos(alpha)**2-\cos(beta)**2 - \cos(gamma)**2
87
88
              +2*cos(alpha)*cos(beta)*cos(gamma)))
89
90
    d = 1/math.sqrt(id2)
91
    print d
92
```

```
d_111 spacing = 2.309 Angstroms
2.30940107676
4.000 4.000 4.000 1.571 1.571 1.571
```

2.30940107676

10.2.5 Center of mass

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

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

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

10.2.6 Moments of inertia

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

```
from ase import *
1
   from ase.structure import molecule
3 from ase.units import *
5 print 'linear rotors: I = [Ia Ia 0]'
6 \quad mlc = 'CO2'
   atoms = molecule(mlc)
  print ' %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
9
10 print
11
12
   print 'symmetric rotors (IA = Ib) < Ic'</pre>
   mlc = 'NH3'
13
  atoms = molecule(mlc)
14
print ' %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
16
   mlc = 'C6H6'
17
18 atoms = molecule(mlc)
19 print ' %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
20 print
21
22 print 'spherical rotors Ia = Ib = Ic'
23 mlc = 'CH4'
24 atoms = molecule(mlc)
25 print ' %s moments of inertia: ' %(mlc), atoms.get_moments_of_inertia()
   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 ]
    spherical rotors Ia = Ib = Ic
      CH4 moments of inertia: [ 3.19145621 3.19145621 3.19145621]
```

10.3 Equations of State

The module :mod: 'ase.util.eos' uses a simple polynomial equation of state to find bulk unit cell equilibrium volumes and bulk modulus. There are

several other choices you could use that are more standard in the literature. Here we summarize them and provide references to the relevant literature.

10.3.1Birch-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 [13]. You can also find the equations for the Vinet and Pourier-Tarantola equations of state in that reference.

Birch-Murnaghan EOS:

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

where $\eta = (V/V_0)^{1/3}$, B_0 and B'_0 are the bulk modulus and its pressure derivative at the equilibrium volume V_0 . You may find other derivations of this equation in the literature too.

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

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

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

10.3.2Murnaghan

The equation most often used in the Murnaghan [26] equation of state is described in [10].

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

where V is the volume, B_0 and B'_0 are the bulk modulus and its pressure derivative at the equilibrium volume V_0 . All of these are parameters that are fitted to energy vs. unit cell volume (V) data. When fitting data to this equation a guess of 2-4 for B'_0 is usually a good start.

10.3.3 Birch

The original Birch equation [1] is:

$$E = E_0 + \frac{9}{8}B_0V_0 \left(\left(\frac{V_0}{V} \right)^{\frac{2}{3}} - 1 \right)^2 + \frac{9}{16}B_0V_0(B_0' - 4) \left(\left(\frac{V}{V_0} \right)^{2/3} - 1 \right)^3$$

The Anton-Schmidt Equation of state [23]

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

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

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.

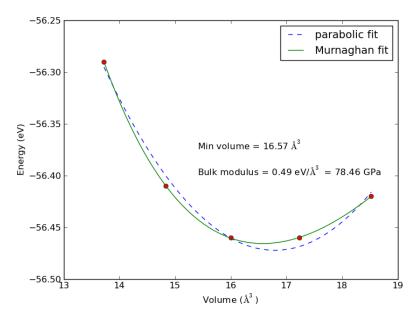
Here we describe a strategy for getting some estimates of the parameters using a linear least squares fitting of a parabola to the data to estimate E_0 , V_0 , B and B'_0 which are used as initial guess for a non-linear least squares fit of the equation of state to the data.

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

```
''', Example of fitting the Birch-Murnaghan EOS to data'''
1
2
    from pylab import * #this includes numpy as np!
3
    from scipy.optimize import leastsq
5
6
    # raw data from 2.2.3-al-analyze-eos.py
    v = np.array([13.72, 14.83, 16.0, 17.23, 18.52])
7
    e = np.array([-56.29, -56.41, -56.46, -56.46, -56.42])
8
    #make a vector to evaluate fits on with a lot of points so it looks smooth
10
    vfit = np.linspace(min(v),max(v),100)
11
12
    ### fit a parabola to the data
13
    # y = ax^2 + bx + c
    a,b,c = polyfit(v,e,2) #this is from pylab
15
16
17
    the parabola does not fit the data very well, but we can use it to get
18
19
    some analytical guesses for other parameters.
20
    VO = minimum energy volume, or where dE/dV=0
21
   E = aV^2 + bV + c
22
    dE/dV = 2aV + b = 0
23
    VO = -b/2a
24
25
    EO is the minimum energy, which is:
26
27
    E0 = aV0^2 + bV0 + c
    B is equal to VO*d^2E/dV^2, which is just 2a*VO
```

```
30
    and from experience we know Bprime_O is usually a small number like 4
31
32
33
34
    #now here are our initial guesses.
    v0 = -b/(2*a)
35
36
    e0 = a*v0**2 + b*v0 + c
    b0 = 2*a*v0
37
    bP = 4
38
39
    #now we have to create the equation of state function
40
41
    def Murnaghan(parameters, vol):
42
43
         given a vector of parameters and volumes, return a vector of energies.
         equation From PRB 28,5480 (1983)
44
45
        E0 = parameters[0]
46
        B0 = parameters[1]
47
        BP = parameters[2]
48
        V0 = parameters[3]
49
50
        E = EO + B0*vol/BP*(((VO/vol)**BP)/(BP-1)+1) - VO*BO/(BP-1.)
51
52
        return E
53
54
    # and we define an objective function that will be minimized
55
    def objective(pars,y,x):
56
         #we will minimize this function
57
58
         err = y - Murnaghan(pars,x)
        return err
59
60
    x0 = [e0, b0, bP, v0] #initial guesses in the same order used in the Murnaghan function
61
62
    murnpars, ier = leastsq(objective, x0, args=(e,v)) #this is from scipy
63
64
65
    #now we make a figure summarizing the results
    plot(v,e,'ro')
66
    plot(vfit, a*vfit**2 + b*vfit + c,'--',label='parabolic fit')
67
    plot(vfit, Murnaghan(murnpars,vfit), label='Murnaghan fit')
68
    xlabel('Volume ($\AA^3$)')
69
70
    ylabel('Energy (eV)')
    legend(loc='best')
71
72
    #add some text to the figure in figure coordinates
73
    ax = gca()
74
    text(0.4,0.5,'Min volume = %1.2f $\AA^3$' % murnpars[3],
75
         transform = ax.transAxes)
76
    text(0.4,0.4,'Bulk modulus = %1.2f eV/\LambdaAA^3$ = %1.2f GPa' % (murnpars[1],
77
                                                                     murnpars[1]*160.21773)
78
79
          , transform = ax.transAxes)
80
    savefig('images/a-eos.png')
    show()
81
82
    print 'initial guesses : ',x0
83
    print 'fitted parameters: ', murnpars
```

initial guesses : [-56.472134949694016, 0.63066383554976546, 4, 16.789583121285862 fitted parameters: [-56.46560371 0.48969488 4.75292243 16.57279459]



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.

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 _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 are treated as valence states

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

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

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

-1578.61345

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.

```
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.

```
from jasp import *
1
JASPRC['queue.nodes']=4
3
    from ase import Atom, Atoms
    atoms = Atoms([Atom('0', [5,5,5], magmom=1)],
                 cell=(6,6,6))
7
    with jasp('molecules/0_sv-4nodes',
8
              encut=300,
9
              xc='PBE',
10
              ispin=2,
              ismear=0,
11
              sigma=0.001,
12
```

-1578.613448

How do you know it ran on four nodes?

```
head molecules/0_sv-4nodes/OUTCAR
```

from jasp import *

```
vasp.5.2.12 11Nov11 complex
```

```
executed on LinuxIFC date 2012.07.19 21:00:24 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.

```
from jasp import *

with jasp('bulk/alloy/cu') as calc:
print calc.python

from numpy import array
from ase import Atom, Atoms
```

```
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 *
1
  with jasp('bulk/alloy/cu') as calc:
      print calc.json
    {"INCAR": {"ldau": null, "eint": null, "lclimb": null, "iband": null, "timestep": nu
  xml This relies on the pyxser module.
  from jasp import *
  with jasp('bulk/alloy/cu') as calc:
3
     print calc.xml
  <?xml version="1.0" encoding="utf-8"?>
   <pyxs:obj xmlns:pyxs="http://projects.coder.cl/pyxser/model/" version="1.0" type="vas</pre>
     <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">
```

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.

```
1    nPd = 4
2    nCu = 5
3    x_Cu = nCu/(nPd + nCu)
4    print x_Cu
5
6    # now cast as floats
7    x_Cu = float(nCu)/float(nPd + nCu)
8    print x_Cu
```

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
```

0.5555555556

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
```

0.5555555556

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] Jr. E. Bright Wilson, J.C. Decius, and Paul C. Cross. *Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra*. Dover Publications, 1955.
- [9] Carlos Fiolhais, L.M. Almeida, and C. Henriques. Extraction of aluminium surface energies from slab calculations: perturbative and non-perturbative approaches. *Progress in Surface Science*, 74(1-8):209 217, 2003. ¡ce:title¿Maria Steslicka Memorial Volume¡/ce:title¿. URL: http://www.sciencedirect.com/science/article/pii/S0079681603000777, doi:10.1016/j.progsurf.2003.08.017.
- [10] 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.

- [11] Paolo Giannozzi and Stefano Baroni. Vibrational and dielectric properties of c[sub 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.
- [12] Jeong Woo Han, Liwei Li, and David S. Sholl. Density functional theory study of h and co adsorption on alkali-promoted mo2c surfaces. The Journal of Physical Chemistry C, 115(14):6870–6876, 2011. URL: http://pubs.acs.org/doi/abs/10.1021/jp200950a, arXiv: http://pubs.acs.org/doi/pdf/10.1021/jp200950a, doi:10.1021/jp200950a.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.

- [18] 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.
- [19] John R. Kitchin, Karsten Reuter, and Matthias Scheffler. Alloy surface segregation in reactive environments: First-principles atomistic thermodynamics study of ag₃Pd(111) in oxygen atmospheres. *Phys. Rev. B*, 77:075437, Feb 2008. URL: http://link.aps.org/doi/10.1103/PhysRevB.77.075437.
- [20] Charles Kittel. *Introduction to Solid State Physics*. Wiley, 8th edition, 2005.
- [21] 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.
- [22] Thomas A. Manz and David S. Sholl. Chemically meaningful atomic charges that reproduce the electrostatic potential in periodic and nonperiodic materials. *Journal of Chemical Theory and Computation*, 6(8):2455–2468, 2010. URL: http://pubs.acs.org/doi/abs/10.1021/ct100125x, arXiv:http://pubs.acs.org/doi/pdf/10.1021/ct100125x, doi:10.1021/ct100125x.
- [23] B. Mayer, H. Anton, E. Bott, M. Methfessel, J. Sticht, J. Harris, and P.C. Schmidt. Ab-initio calculation of the elastic constants and thermal expansion coefficients of laves phases. *Intermetallics*, 11(1):23 32, 2003. URL: http://www.sciencedirect.com/science/article/pii/S0966979502001279, doi:10.1016/S0966-9795(02)00127-9.
- [24] 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.
- [25] 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.
- [26] F. D. Murnaghan. The compressibility of media under extreme pressures. Proceedings of the National Academy of Sciences of the United States of America, 30(9):pp. 244-247, 1944. URL: http://www.jstor.org/stable/87468.
- [27] Jörg Neugebauer and Matthias Scheffler. Adsorbate-substrate and adsorbate-adsorbate interactions of na and k adlayers on al(111). *Phys. Rev. B*, 46:16067–16080, Dec 1992. URL: http://link.aps.org/doi/10.1103/PhysRevB.46.16067, doi:10.1103/PhysRevB.46.16067.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] Axel van de Walle. Multicomponent multisublattice alloys, non-configurational entropy and other additions to the alloy theoretic automated toolkit. Calphad, 33(2):266 278, 2009. ¡ce:title¿Tools for Computational Thermodynamics¡/ce:title¿. URL: http://www.sciencedirect.com/science/article/pii/S0364591608001314, doi:10.1016/j.calphad.2008.12.005.
- [35] 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.
- [36] 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.
- [37] 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

\mathbf{Index}

```
convergence
ENCUT, 53
convergence:KPOINTS, 106
DFT+U, 178
HSE06, 183
ISMEAR, 104
SIGMA, 104
```