

# Modeling materials using density functional theory

John Kitchin

2012-07-11 Wed

## Contents

<b>1</b>	<b>Introduction to this book</b>	<b>11</b>
<b>2</b>	<b>Introduction to DFT</b>	<b>12</b>
2.1	Background . . . . .	13
2.2	Exchange correlation functionals . . . . .	14
2.3	Basis sets . . . . .	15
2.4	Pseudopotentials . . . . .	16
2.5	Fermi Temperature and band occupation numbers . . . . .	17
2.6	Spin polarization and magnetism . . . . .	17
2.7	<b>TODO</b> Recommended reading . . . . .	18
<b>3</b>	<b>Molecules</b>	<b>18</b>
3.1	Defining and visualizing molecules . . . . .	18
3.1.1	From scratch . . . . .	18
3.1.2	Reading other data formats into a calculation . . . . .	21
3.1.3	Predefined molecules . . . . .	23
3.1.4	Combining Atoms objects . . . . .	28
3.2	Simple properties . . . . .	29
3.2.1	Getting cartesian positions . . . . .	29
3.2.2	Molecular weight and molecular formula . . . . .	30
3.2.3	Center of mass . . . . .	31
3.2.4	Moments of inertia . . . . .	31
3.2.5	Computing bond lengths and angles . . . . .	33
	Dihedral angles . . . . .	35
3.3	Simple properties that require single computations . . . . .	36
3.3.1	Energy and forces . . . . .	36
	Convergence with unit cell size . . . . .	36
	Convergence of ENCUT . . . . .	39
3.3.2	Visualizing electron density . . . . .	40
3.3.3	Dipole moments . . . . .	44
3.3.4	The density of states (DOS) . . . . .	45
3.3.5	Atom-projected density of states on molecules . . . . .	46
3.3.6	<b>TODO</b> Electrostatic potential . . . . .	48
3.3.7	Bader analysis . . . . .	50

3.4	Geometry optimization . . . . .	52
3.4.1	Manual determination of a bond length . . . . .	52
3.4.2	Automatic geometry optimization with VASP . . . . .	55
3.4.3	Relaxation of a water molecule . . . . .	56
3.5	Vibrational frequencies . . . . .	57
3.5.1	Manual calculation of vibrational frequency . . . . .	57
3.5.2	Automated vibrational calculations . . . . .	58
	Zero-point energy for multiple modes . . . . .	60
3.6	Simulated infrared spectra . . . . .	61
3.7	Thermochemical properties of molecules . . . . .	65
3.8	Molecular reaction energies . . . . .	67
3.8.1	O <sub>2</sub> dissociation . . . . .	67
	Simple estimate of O <sub>2</sub> dissociation energy . . . . .	68
	Estimating O <sub>2</sub> dissociation energy with spin polarization in triplet ground states . . . . .	68
	Looking at the two spin densities . . . . .	70
	Convergence study of the O <sub>2</sub> dissociation energy . . . . .	71
	Illustration of the effect of SIGMA . . . . .	73
	Estimating triplet oxygen dissociation energy with low symmetry . . . . .	75
	Estimating singlet oxygen dissociation energy . . . . .	78
	Verifying the magnetic moments on each atom . . . . .	79
	Using a different potential . . . . .	80
3.8.2	Water gas shift example . . . . .	81
	Calculation summaries . . . . .	83
	CO . . . . .	83
	CO <sub>2</sub> . . . . .	84
	H <sub>2</sub> . . . . .	85
	H <sub>2</sub> O . . . . .	86
3.8.3	Temperature dependent water gas shift equilibrium constant . . . . .	87
	CO vibrations . . . . .	87
	CO <sub>2</sub> vibrations . . . . .	88
	H <sub>2</sub> vibrations . . . . .	88
	H <sub>2</sub> O vibrations . . . . .	89
	<b>TODO</b> Thermochemistry . . . . .	89
3.9	Molecular reaction barriers . . . . .	92
3.9.1	Get initial and final states . . . . .	92
3.9.2	Run band calculation . . . . .	93
<b>4</b>	<b>Bulk systems</b> . . . . .	<b>94</b>
4.1	Defining and visualizing bulk systems . . . . .	95
4.1.1	Built-in functions in ase . . . . .	95
4.1.2	Using <a href="http://materialsproject.org">http://materialsproject.org</a> . . . . .	100
4.2	Computational parameters that are important for bulk structures . . . . .	103
4.2.1	k-point convergence . . . . .	103
4.2.2	Effect of SIGMA . . . . .	105
4.2.3	The number of bands . . . . .	107

4.3	Determining bulk structures . . . . .	110
4.3.1	fcc/bcc crystal structures . . . . .	111
4.3.2	Optimizing the hcp lattice constant . . . . .	116
4.3.3	Complex structures with internal degrees of freedom . . . . .	117
4.3.4	Effect of XC on bulk properties . . . . .	125
4.4	Cohesive energy . . . . .	127
4.5	<b>TODO</b> Elastic properties . . . . .	129
4.6	Bulk thermodynamics . . . . .	129
4.7	Effect of pressure on phase stability . . . . .	130
4.8	Bulk reaction energies . . . . .	134
4.8.1	Alloy formation energies . . . . .	134
	Basic alloy formation energy . . . . .	135
4.8.2	Metal oxide oxidation energies . . . . .	142
	Cu <sub>2</sub> O calculation . . . . .	142
	CuO calculation . . . . .	144
	Reaction energy calculation . . . . .	145
4.9	Bulk density of states . . . . .	146
4.10	Atom projected density of states . . . . .	148
4.10.1	Effect of RWIGS on ADOS . . . . .	151
4.11	Band structures . . . . .	153
4.11.1	<b>TODO</b> create example showing band dispersion with lattice constant . . . . .	155
4.12	Magnetism . . . . .	155
4.12.1	Determining if a magnetic solution is energetically favorable . . . . .	155
4.12.2	Antiferromagnetic spin states . . . . .	157
4.12.3	<b>TODO</b> NiO-FeO formation energies with magnetism . . . . .	158
4.13	<b>TODO</b> phonons . . . . .	158
4.14	<b>TODO</b> solid state NEB . . . . .	158
<b>5</b>	<b>Surfaces</b> . . . . .	<b>158</b>
5.1	Surface structures . . . . .	158
5.1.1	Simple surfaces . . . . .	158
5.1.2	Vicinal surfaces . . . . .	159
5.2	Surface relaxation . . . . .	162
5.3	Surface reconstruction . . . . .	165
5.3.1	Au(110) missing row reconstruction . . . . .	165
	Clean Au(110) slab . . . . .	165
	Missing row in Au(110) . . . . .	167
	Bulk Au . . . . .	168
	Analysis of energies . . . . .	169
5.3.2	Ag(110) missing row reconstruction . . . . .	169
	Clean Ag(110) slab . . . . .	169
	Missing row in Ag(110) . . . . .	170
	Bulk Ag . . . . .	170
	Analysis of energies . . . . .	170
5.4	Work function . . . . .	171
5.5	Surface energy . . . . .	172

5.5.1	Advanced topics in surface energy . . . . .	175
5.6	Dipole correction . . . . .	176
5.6.1	Slab with no dipole correction . . . . .	176
5.6.2	Slab with a dipole correction . . . . .	176
5.6.3	Comparing no dipole correction with a dipole correction . . . . .	177
5.7	Adsorption energies . . . . .	179
5.7.1	Simple estimate of the adsorption energy . . . . .	179
	Calculations . . . . .	179
	clean slab calculation . . . . .	179
	fcc site . . . . .	179
	bridge site . . . . .	180
	hcp site . . . . .	180
	Analysis of adsorption energies . . . . .	181
	Adsorption on bridge site with constraints . . . . .	184
5.7.2	Coverage dependence . . . . .	185
	clean slab calculation . . . . .	186
	fcc site at 1 ML coverage . . . . .	186
	Adsorption energy at 1ML . . . . .	186
5.7.3	Effect of adsorption on the surface energy . . . . .	187
5.8	Adsorbate vibrations . . . . .	188
5.8.1	<b>TODO</b> Vibrations of the bridge site . . . . .	189
5.9	Surface Diffusion barrier . . . . .	190
5.9.1	Standard nudged elastic band method . . . . .	190
5.9.2	Climbing image NEB . . . . .	192
5.9.3	Using vibrations to confirm a transition state . . . . .	193
5.10	<b>TODO</b> Diffusion rates with transition state theory . . . . .	193
5.11	<b>TODO</b> Effects of electric fields on adsorbates . . . . .	193
5.12	<b>TODO</b> Simulating STM images . . . . .	194
<b>6</b>	<b>Atomistic thermodynamics</b> . . . . .	<b>194</b>
6.1	Bulk phase stability of oxides . . . . .	196
6.2	Effect on adsorption . . . . .	200
6.3	Atomistic therodynamics and multiple reactions . . . . .	202
<b>7</b>	<b>Advanced electronic structure methods</b> . . . . .	<b>203</b>
7.1	DFT+U . . . . .	203
7.1.1	Metal oxide oxidation energies with DFT+U . . . . .	203
	Cu <sub>2</sub> O calculation with U=4.0 . . . . .	203
	CuO calculation with U=4.0 . . . . .	204
	Reaction energy calculation with DFT+U . . . . .	206
	How much does U affect the reaction energy? . . . . .	206
7.2	Hybrid functionals . . . . .	207
7.2.1	FCC Ni DOS . . . . .	207
7.3	<b>TODO</b> DFT+D . . . . .	209
7.3.1	PBE . . . . .	209
	gas-phase benzene . . . . .	209
	clean slab . . . . .	209

7.3.2	benzene on Au(111)	210
7.3.2	DFT-D2	211
	gas-phase benzene	211
	clean slab	211
	benzene on Au(111)	211
7.3.3	Advanced vdW-DF functionals	212
7.4	ELF	212
7.5	<b>TODO</b> Charge partitioning schemes	214
7.6	<b>TODO</b> Modeling Core level shifts	214
<b>8</b>	<b>Acknowledgments</b>	<b>214</b>
<b>9</b>	<b>Appendices</b>	<b>214</b>
9.1	Recipes	214
9.1.1	Modifying Atoms by deleting atoms	214
9.1.2	Advanced tagging	214
9.1.3	Using units in ase	218
9.1.4	Extracting parts of an array	218
9.1.5	Statistics	220
	Confidence intervals	220
9.1.6	Curve fitting	220
	Linear fitting	220
9.1.7	Nonlinear curve fitting	221
9.1.8	Nonlinear curve fitting by direct least squares minimization	222
9.1.9	Nonlinear curve fitting with confidence intervals	224
9.1.10	Interpolation with splines	226
9.1.11	Interpolation in 3D	227
9.1.12	Reading and writing data	232
	Built-in io modules	232
	From scratch	233
9.1.13	Integration	234
9.1.14	Numerical differentiation	234
	Simple loops to define finite difference derivatives	234
	FFT derivatives	238
9.1.15	NetCDF files	240
9.1.16	Python modules	240
9.1.17	Writing and reading Excel files	240
	Writing Excel files	240
	Reading Excel files	241
9.1.18	<b>TODO</b> making movies	241
9.2	Computational geometry	241
9.2.1	Changing coordinate systems	241
9.2.2	Simple distances, angles	244
9.2.3	Unit cell properties	244
9.2.4	d-spacing	245
9.3	Equations of State	247
9.3.1	Birch-Murnaghan	247

9.3.2	Murnaghan . . . . .	247
9.3.3	Birch . . . . .	247
9.3.4	The Anton-Schmidt Equation of state <sup>1</sup> . . . . .	247
9.3.5	Fitting data to these equations of state . . . . .	248
9.4	Miscellaneous <code>jasp</code> /VASP tips . . . . .	250
9.4.1	Installing <code>jasp</code> . . . . .	250
9.4.2	Using a special setup . . . . .	251
9.4.3	Running <code>jasp</code> in parallel . . . . .	252
9.4.4	Running multiple instances of <code>jasp</code> in parallel . . . . .	253
9.4.5	Exporting data json, xml, python, sqlite . . . . .	256
	python . . . . .	256
	json . . . . .	256
	xml . . . . .	261
9.4.6	Recommended values for ENCUT and valence electrons for different POT-CAR files . . . . .	262
<b>10</b>	<b>Python</b>	<b>272</b>
10.1	easy_install as a user . . . . .	272
10.2	Integer division math gotchas . . . . .	272
<b>11</b>	<b>References</b>	<b>273</b>

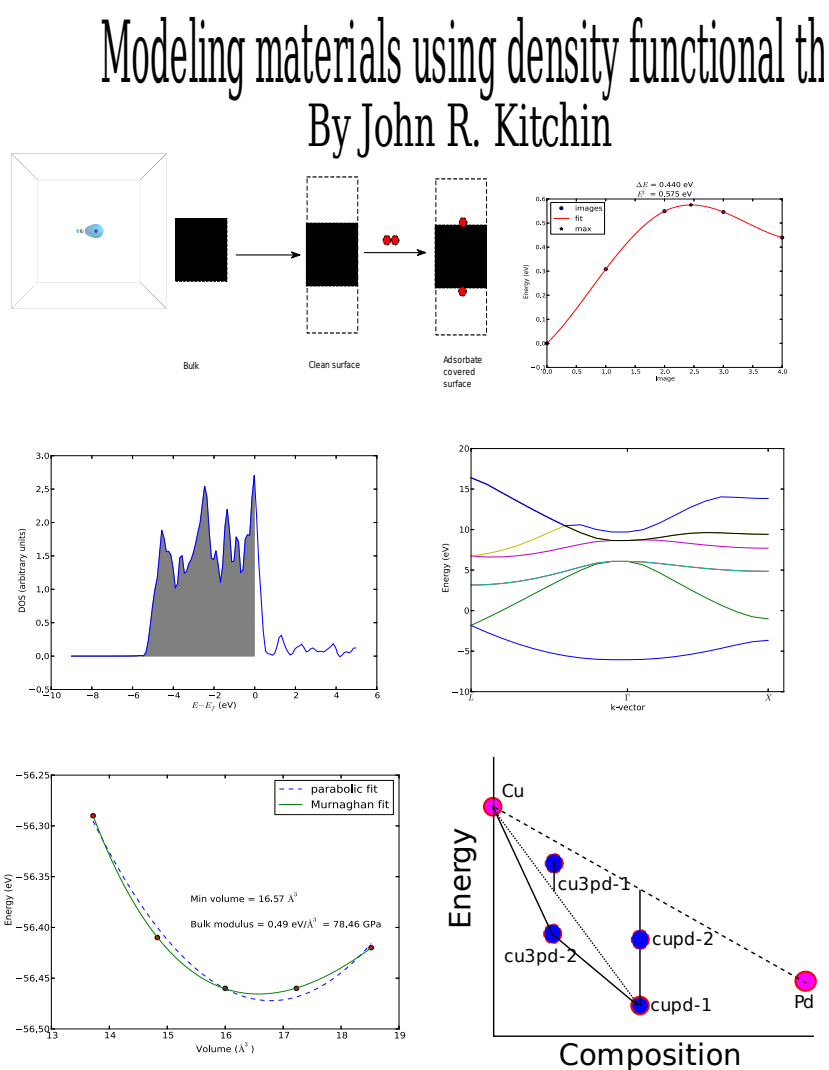


Figure 1: figure:./images/cover-page.svg

## List of Figures

1	figure:./images/cover-page.svg . . . . .	7
2	Image of a CO molecule with the C at the origin. . . . .	19
3	CO in a face-centered cubic unit cell. . . . .	20
4	An isobutane molecule read in from an XYZ formatted data file. . . . .	23
5	A $\text{CH}_3\text{CN}$ molecule in a box. . . . .	26
6	The rotated version of $\text{CH}_3\text{CN}$ . . . . .	27

7	Rotated CH <sub>3</sub> CN molecule . . . . .	28
8	Image featuring ammonia and oxygen molecule in one unit cell. . . . .	29
9	Schematic of the vectors defining the H-N-H angle. . . . .	34
10	Schematic of the calculated ethane dihedral angle. . . . .	35
11	Total energy of a CO molecule as a function of the unit cell length. . . . .	37
12	Ideal gas pressure dependence on temperature and unit cell volume. . . . .	38
13	Dependence of the total energy of CO molecule on ENCUT. . . . .	40
14	Charge density of a CO molecule that is located at the origin. The electron density that is outside the cell is wrapped around to the other corners. . . . .	42
15	Charge density of a CO molecule centered in the unit cell. Now the electron density is centered in the unit cell. . . . .	43
16	Density of states for a CO molecule. . . . .	46
17	Atom-projected DOS for a CO molecule. The total density of states and the <i>s</i> , <i>p</i> and <i>d</i> states on the C and O are shown. . . . .	48
18	Plot of the electrostatic potential of CF <sub>3</sub> Br. TODO: figure out how to do an isosurface of charge, colormapped by the local potential. . . . .	49
19	Energy vs CO bond length. . . . .	55
20	Spin-polarized DOS for the O <sub>2</sub> molecule. . . . .	70
21	Convergence study of the O <sub>2</sub> dissociation energy as a function of ENCUT. . . . .	72
22	Effect of SIGMA on the oxygen dissociation energy. . . . .	75
23	Thermodynamic energies of the water gas shift reaction as a function of temperature. . . . .	91
24	Temperature dependence of the equilibrium constant. . . . .	92
25	Nudged elastic band results for ammonia flipping. . . . .	94
26	A simple fcc Ag bulk structure in the primitive unit cell. . . . .	96
27	A simple fcc Ag bulk structure in the primitive unit cell. . . . .	97
28	A simple fcc Ag bulk structure in the traditional unit cell. . . . .	97
29	A simple Ag <sub>3</sub> Pd bulk structure. . . . .	98
30	A top view of graphite. . . . .	98
31	A side view of graphite. . . . .	99
32	A view of a NaCl crystal structure. . . . .	100
33	An RuO <sub>2</sub> unit cell prepared from a cif file. . . . .	101
34	k-point convergence of the total energy of fcc Ag. . . . .	104
35	Effects of SIGMA on the occupancies of the Cu system. . . . .	107
36	Total energy vs. fcc lattice constant for Cu. It appears the minimum is near 3.65 Å. . . . .	112
37	Total energy vs. volume for fcc Cu with fitted cubic polynomial equation of state. . . . .	113
38	Comparison of energies between fcc and bcc Cu. The fcc structure is lower in energy. . . . .	115
39	Total energy vs. <i>c/a</i> for different values of <i>a</i> . . . . .	117
40	Contour plot of the total energy of hcp Ru for different values of <i>a</i> and <i>c/a</i> . . . . .	118
41	k-point convergence of rutile TiO <sub>2</sub> . . . . .	120
42	Total energy vs. volume for rutile TiO <sub>2</sub> in step 1 of the optimization. . . . .	122
43	Total energy vs. volume for step 2 of the unit cell optimization. . . . .	123
44	Equations of state (E(V)) for anatase and rutile TiO <sub>2</sub> . . . . .	133
45	Illustration of the common tangent that shows the pressure where anatase and rutile coexist before anatase converts to rutile. . . . .	134



46	Conceptual picture of two alloys with exothermic formation energies. The dashed line represents a composition weighted average energy of the parent metals. E4 and E3 are energies associated with two different alloy structures at the same composition. Both structures are more stable than a mixture of pure metals with the same composition, but E3 is more stable than E4. . . . .	138
47	Illustration of of an alloy structure with an exothermic formation energy that is not stable with respect to phase separation. The solid line shows the composition weighted average energy of a mixture of Cu and cupd-2. Since the energy of cu3pd-1 is above the solid line, it is less favorable than a mixture of Cu and cupd-2 with the same composition. . . . .	140
48	Illustration that cu3pd-2 is more stable than cu3pd-1 and that it is more stable than a composition weighted mixture of Cu and cupd-1. The dotted line shows the energy of a composition weighted average energy of a mixture of Cu and cupd-1. Since cu3pd-2 is below the dotted line, it is more stable than the phase-separated mixture. . . . .	142
49	Total DOS for bulk Pd. . . . .	147
50	Total DOS for Pd computed with ISMEAR=-5 . . . . .	148
51	Atom projected <i>d</i> -band for bulk Pd. The shaded area corresponds to the occupied states below the Fermi level. . . . .	151
52	Effect of the RWIGS on the number of occupied <i>d</i> -states. . . . .	153
53	Effect of the RWIGS on the <i>d</i> -band center and width. . . . .	153
54	Calculated band-structure for Si. . . . .	155
55	Total energy vs. total magnetic moment for bcc Fe. . . . .	156
56	An Al(111) slab with three layers and 20 Å of vacuum. . . . .	159
57	An Au(211) surface constructed with ase. . . . .	161
58	The unreconstructed Au(110) surface viewed from the side. . . . .	166
59	Au(110) with the missing row reconstruction. . . . .	168
60	<i>xy</i> averaged local electrostatic potential of an Al(111) slab. . . . .	172
61	Schematic figure illustrating the calculation of a surface energy. . . . .	173
62	Surface energy of an Al(111) slab as a function of thickness. . . . .	175
63	Comparison of the electrostatic potentials with a dipole correction and without it. . . . .	178
64	Final geometry of the fcc site. . . . .	182
65	Final geometry of the hcp site. . . . .	182
66	Final geometry of the bridge site. You can see that the oxygen atom ended up in the fcc site. . . . .	182
67	Initial geometry of the bridge site. It is definitely on the bridge. . . . .	183
68	Final geometry of the bridge site. It has fallen into the fcc site. . . . .	183
69	Initial state of the O atom on the bridge site. . . . .	185
70	Final state of the constrained O atom, still on the bridge site. . . . .	185
71	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. . . . .	187
72	Energy pathway for O diffusion from an fcc to hcp site with a spline fit to determine the barrier. . . . .	192
73	Climbing image NEB. . . . .	193
74	Effect of temperature on the Gibbs free energy of an O <sub>2</sub> molecule at 1 atm. . . . .	195
75	Effects of pressure on the ideal gas Gibbs free energy of O <sub>2</sub> . . . . .	196

76	$\Delta \mu_{O_2}(T,p)$ at different pressures and temperatures. . . . .	198
77	Temperature dependent decomposition pressure for $Ag_2O$ . . . . .	199
78	Dependence of the formation energy on the oxygen chemical potential. . . . .	200
79	Effect of oxygen chemical potential on the adsorption energy. . . . .	202
80	Comparison of DOS from GGA, and two hybrid GGAs (PBE0 ad HSE06). . . .	209
81	ELF for an isosurface of 0.3 for $CF_4$ . . . . .	213
82	ELF for an isosurface of 0.75 for $CF_4$ . . . . .	213
83	The tagged bcc(111) structure created above. Unfortunately, the frozen atoms do not show up in the figure. . . . .	217
84	Example of slicing out part of an array. The solid line represents the whole array, and the symbols are the array between $2 < x < 4$ . . . . .	219
85	Example of linear least-squares curve fitting. . . . .	221
86	Example of least-squares non-linear curve fitting. . . . .	223
87	Fitting a nonlinear function. . . . .	224
88	Nonlinear fit to data. . . . .	226
89	Illustration of a spline fit to data and finding the maximum point. . . . .	227
90	Trilinear interpolation scheme. . . . .	228
91	An example of interpolated charge density of a CO molecule along the axis of molecule. . . . .	230
92	Interpolated charge density for a CO molecule. . . . .	232
93	Comparison of different numerical derivatives. . . . .	236
94	Comparison of different numerical derivatives. . . . .	237
95	Comparison of 2 point and 4 point numerical derivatives. . . . .	238
96	Comparison of FFT numerical derivatives. . . . .	239
97	Fitted equation of state for bulk data. The initial fitted parabola is shown to illustrate how it is useful for making initial guesses of the minimum and bulk modulus. . . . .	250
98	Equation of state for Cu using the multiprocessing module. . . . .	255

## List of Tables

1	Planewave cutoff energies (in eV) required for different convergence precisions for different elements. . . . .	16
2	Bader charges for a water molecule . . . . .	52
3	Symmetry numbers for common point groups . . . . .	65
4	Total energy vs. lattice constant for BCC Cu. . . . .	115
5	Total energy of $TiO_2$ vs. volume. . . . .	121
6	Adsorption site dependence of adsorption energies of oxygen on Pt(111). . . . .	201
7	Meaning of extensions on POTCAR files for special setups. . . . .	251
8	Parameters for POTCAR files. . . . .	263
8	Parameters for POTCAR files. . . . .	264
8	Parameters for POTCAR files. . . . .	265
8	Parameters for POTCAR files. . . . .	266
8	Parameters for POTCAR files. . . . .	267
8	Parameters for POTCAR files. . . . .	268
8	Parameters for POTCAR files. . . . .	269

8	Parameters for POTCAR files. . . . .	270
8	Parameters for POTCAR files. . . . .	271
8	Parameters for POTCAR files. . . . .	272

## 1 Introduction to this book

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

You will read a lot of python code in this book, as the comprehension of code is an integral part of its design. I believe that computational work should always be scripted. As a result, a written record of everything you have done will be available, allowing you to reproduce your code or report the method of its execution exactly at a later time.

This book makes heavy use of many computational tools including:

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

`jasp` is a sophisticated wrapper to the `ase.calculators.vasp` python interface to the VASP calculator. It was written by me to facilitate writing this book and to develop the best possible way to run DFT calculations. The best way to learn to use `jasp` is from this book. It probably requires the latest svn version of `ase` to work since I have been adding new functionality to `ase.calculators.vasp` as `jasp` is developed. `jasp` is available at <https://github.com/jkitchin/jasp>.

The goal in writing `jasp` was to create a computational environment where you write one script to create, run and analyze your calculations. The code is smart, and submits jobs to a queue system when required, does not submit jobs more than once, allows many jobs to run in

parallel, and gives you answers when available. `jasp` takes care of changing into VASP calculation directories, and then changing back to where you started from. `jasp` has an interface to `sqlite`, and to version control (`git`) software. `jasp` has some features that `ase.calculators.vasp` does not have yet. They may get incorporated into `ase` one day.

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

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

---

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

---

with code similar to this:

---

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

---

The DFT code used primarily in this book is [VASP](#).

- [VASP wiki](#)
- [VASP Manual](#)

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

### Exercise 1.1

Review all the hyperlinks in this chapter.

## 2 Introduction to DFT

A comprehensive overview of DFT is beyond the scope of this book, as excellent reviews on these subjects are readily found in the literature, and are suggested reading in the following paragraph. Instead, this chapter is intended to provide a useful starting point for a non-expert to begin learning about and using DFT in the manner used in this book. Much of the information presented here is standard knowledge among experts, but a consequence of this is that it is rarely discussed in current papers in the literature. A secondary goal of this chapter is to provide new users with a path through the extensive literature available and to point out potential difficulties and pitfalls in these calculations.

A fairly standard textbook on DFT is the one written by Parr and Yang.<sup>2</sup> The Chemist’s Guide to DFT<sup>3</sup> is more readable and contains more practical information for running calculations, but both of these books focus on molecular systems. The standard texts in solid state physics are by Kittel<sup>4</sup> and Ashcroft and Mermin.<sup>5</sup> Both have their fine points, the former being more mathematically rigorous and the latter more readable. However, neither of these books is particularly easy to relate to chemistry. For this, one should consult the exceptionally clear writings of Roald Hoffman,<sup>6;7</sup> and follow these with the work of Nørskov and coworkers.<sup>8;9</sup>

In this chapter, only the elements of DFT that are relevant to this work will be discussed. An excellent review on other implementations of DFT can be found in Reference<sup>10</sup>, and details on the various algorithms used in DFT codes can be found in Refs.<sup>11;12</sup>.

One of the most useful sources of information has been the dissertations of other students, perhaps because the difficulties they faced in learning the material are still fresh in their minds. Thomas Bligaard, a coauthor of Dacapo, wrote a particularly relevant thesis on exchange/correlation functionals<sup>13</sup> and a dissertation illustrating the use of DFT to design new alloys with desirable thermal and mechanical properties.<sup>14</sup> The Ph.D. thesis of Ari Seitsonen contains several useful appendices on k-point setups, and convergence tests of calculations, in addition to a thorough description of DFT and analysis of calculation output.<sup>15</sup> Finally, another excellent overview of DFT and its applications to bimetallic alloy phase diagrams and surface reactivity is presented in the PhD thesis of Robin Hirschl.<sup>16</sup>

## 2.1 Background

In 1926, Erwin Schrödinger published the first accounts of his now famous wave equation.<sup>17</sup> He later shared the Nobel prize with Paul A. M. Dirac in 1933 for this discovery. Schrödinger’s wave function seemed extremely promising, as it contains all of the information available about a system. Unfortunately, most practical systems of interest consist of many interacting electrons, and the effort required to find solutions to Schrödinger’s equation increases exponentially with the number of electrons, limiting this approach to systems with a small number of relevant electrons,  $N \lesssim O(10)$ .<sup>18</sup> Even if this rough estimate is off by an order of magnitude, a system with 100 electrons is still very small, for example, two Ru atoms if all the electrons are counted, or perhaps ten Pt atoms if only the valence electrons are counted. Thus, the wave function method, which has been extremely successful in studying the properties of small molecules, is unsuitable for studies of large, extended solids. Interestingly, this difficulty was recognized by Dirac as early as 1929, when he wrote “The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the application of these laws leads to equations much too complicated to be soluble.”<sup>19</sup>

In 1964, Hohenberg and Kohn showed that the ground state total energy of a system of interacting electrons is a unique functional of the electron density.<sup>20</sup> By definition, a function returns a number when given a number. For example, in  $f(x) = x^2$ ,  $f(x)$  is the function, and it equals four when  $x = 2$ . A functional returns a number when given a function. Thus, in  $g(f(x)) = \int_0^\pi f(x)dx$ ,  $g(f(x))$  is the functional, and it is equal to two when  $f(x) = \sin(x)$ . Hohenberg and Kohn further identified a variational principle that appeared to reduce the problem of finding the ground state energy of an electron gas in an external potential (i.e., in the presence of ion cores) to that of the minimization of a functional of the three-dimensional density function. Unfortunately, the definition of the functional involved a set of  $3N$ -dimensional trial wave functions.

In 1965, Kohn and Sham made a significant breakthrough when they showed that the problem of many interacting electrons in an external potential can be mapped exactly to a set of noninteracting electrons in an effective external potential.<sup>21</sup> This led to a set of self-consistent, single particle equations known as the Kohn-Sham (KS) equations:

$$\left(-\frac{1}{2}\nabla^2 + v_{eff}(\mathbf{r}) - \epsilon_j\right)\varphi_j(\mathbf{r}) = 0, \quad (1)$$

with

$$v_{eff}(\mathbf{r}) = v(\mathbf{r}) + \int \frac{n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' + v_{xc}(\mathbf{r}), \quad (2)$$

where  $v(\mathbf{r})$  is the external potential and  $v_{xc}(\mathbf{r})$  is the exchange-correlation potential, which depends on the entire density function. Thus, the density needs to be known in order to define the effective potential so that Eq. (1) can be solved.  $\varphi_j(\mathbf{r})$  corresponds to the  $j^{th}$  KS orbital of energy  $\epsilon_j$ .

The ground state density is given by:

$$n(\mathbf{r}) = \sum_{j=1}^N |\varphi_j(\mathbf{r})|^2 \quad (3)$$

To solve Eq. (1) then, an initial guess is used for  $\varphi_j(r)$  which is used to generate Eq. (3), which is subsequently used in Eq. (2). This equation is then solved for  $\varphi_j(\mathbf{r})$  iteratively until the  $\varphi_j(\mathbf{r})$  that result from the solution are the same as the  $\varphi_j(\mathbf{r})$  that are used to define the equations, that is, the solutions are self-consistent. Finally, the ground state energy is given by:

$$E = \sum_j \epsilon_j + E_{xc}[n(\mathbf{r})] - \int v_{xc}(\mathbf{r})n(\mathbf{r})d\mathbf{r} - \frac{1}{2} \int \frac{n(\mathbf{r})n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}'d\mathbf{r}, \quad (4)$$

where  $E_{xc}[n(\mathbf{r})]$  is the exchange-correlation energy functional. Walter Kohn shared the Nobel prize in Chemistry in 1998 for this work.<sup>18</sup> The other half of the prize went to John Pople for his efforts in wave function based quantum mechanical methods.<sup>22</sup> Provided the exchange-correlation energy functional is known, Eq. (4) is exact. However, the exact form of the exchange-correlation energy functional is not known, thus approximations for this functional must be used.

## 2.2 Exchange correlation functionals

The two main types of exchange/correlation functionals used in DFT are the local density approximation (LDA) and the generalized gradient approximation (GGA). In the LDA, the exchange-correlation functional is defined for an electron in a uniform electron gas of density  $n$ .<sup>21</sup> It is obviously exact for a uniform electron gas, and is anticipated to be valid for slowly varying densities. In molecules and solids, however, the density tends to vary substantially in space. Despite this, the LDA has been very successfully used in many systems. It tends to predict overbonding in both molecular and solid systems,<sup>23</sup> and it tends to make semiconductor systems too metallic (the band gap problem).<sup>24</sup>

The generalized gradient approximation includes corrections for gradients in the electron density, and is often implemented as a corrective function of the LDA. The form of this corrective

function, or “exchange enhancement” function determines which functional it is, e.g. PBE, RPBE, revPBE, etc.<sup>25</sup> In this book the PBE GGA functional is used the most. N{ $\emptyset$ }rskov and coworkers have found that the RPBE functional gives superior chemisorption energies for atomic and molecular bonding to surfaces, but that it gives worse bulk properties, such as lattice constants compared to experimental data.<sup>25</sup>

Finally, there are increasingly new types of functionals in the literature. The so-called hybrid functionals, such as B3LYP, are more popular with gaussian basis sets (e.g. in Gaussian), but they are presently inefficient with planewave basis sets. None of these other types of functionals were used in this work. For more details see Chapter 6 in Ref.<sup>3</sup> and Thomas Bligaard’s thesis on exchange and correlation functionals.<sup>13</sup>

## 2.3 Basis sets

Briefly, VASP utilizes planewaves as the basis set to expand the Kohn-Sham orbitals. In a periodic solid, one can use Bloch’s theorem to show that the wave function for an electron can be expressed as the product of a planewave and a function with the periodicity of the lattice<sup>5</sup> Ch. 8:

$$\psi_{n\mathbf{k}}(\mathbf{r}) = \exp(i\mathbf{k} \cdot \mathbf{r})u_{n\mathbf{k}}(\mathbf{r}) \quad (5)$$

where  $\mathbf{r}$  is a position vector, and  $\mathbf{k}$  is a so-called wave vector that will only have certain allowed values defined by the size of the unit cell. Bloch’s theorem sets the stage for using planewaves as a basis set, because it suggests a planewave character of the wave function. If the periodic function  $u_{n\mathbf{k}}(\mathbf{r})$  is also expanded in terms of planewaves determined by wave vectors of the reciprocal lattice vectors,  $\mathbf{G}$ , then the wave function can be expressed completely in terms of a sum of planewaves:<sup>11</sup>

$$\psi_i(\mathbf{r}) = \sum_{\mathbf{G}} c_{i,\mathbf{k}+\mathbf{G}} \exp(i(\mathbf{k} + \mathbf{G}) \cdot \mathbf{r}). \quad (6)$$

where  $c_{i,\mathbf{k}+\mathbf{G}}$  are now coefficients that can be varied to determine the lowest energy solution. This also converts Eq. (1) from an integral equation to a set of algebraic equations that can readily be solved using matrix algebra.

In aperiodic systems, such as systems with even one defect, or randomly ordered alloys, there is no periodic unit cell. Instead one must represent the portion of the system of interest in a supercell, which is then subjected to the periodic boundary conditions so that a planewave basis set can be used. It then becomes necessary to ensure the supercell is large enough to avoid interactions between the defects in neighboring supercells. The case of the randomly ordered alloy is virtually hopeless as the energy of different configurations will fluctuate statistically about an average value. These systems were not considered in this work, and for more detailed discussions the reader is referred to Ref.<sup>26</sup> Once a supercell is chosen, however, Bloch’s theorem can be applied to the new artificially periodic system.

To get a perfect expansion, one needs an infinite number of planewaves. Luckily, the coefficients of the planewaves must go to zero for high energy planewaves, otherwise the energy of the wave function would go to infinity. This provides justification for truncating the planewave basis set above a cutoff energy. Careful testing of the effect of the cutoff energy on the total energy can be done to determine a suitable cutoff energy. The cutoff energy required to obtain a particular convergence precision is also element dependent, shown in Table 1. It can also vary with the “softness” of the pseudopotential. Thus, careful testing should be done to ensure the desired level of convergence of properties in different systems. Table 1 refers to convergence of



total energies. These energies are rarely considered directly, it is usually differences in energy that are important. These tend to converge with the planewave cutoff energy much more quickly than total energies, due to cancellations of convergence errors. In this work, 350 eV was found to be suitable for the H adsorption calculations, but a cutoff energy of 450 eV was required for O adsorption calculations.

Table 1: Planewave cutoff energies (in eV) required for different convergence precisions for different elements.

Precision	Low	High
Mo	168	293
O	300	520
O <sub>sv</sub>	1066	1847

Bloch’s theorem eliminates the need to calculate an infinite number of wave functions, because there are only a finite number of electrons in the unit (super) cell. However, there are still an infinite number of discrete  $\mathbf{k}$  points that must be considered, and the energy of the unit cell is calculated as an integral over these points. It turns out that wave functions at  $\mathbf{k}$  points that are close together are similar, thus an interpolation scheme can be used with a finite number of  $\mathbf{k}$  points. This also converts the integral used to determine the energy into a sum over the  $\mathbf{k}$  points, which are suitably weighted to account for the finite number of them. There will be errors in the total energy associated with the finite number of  $\mathbf{k}$ , but these can be reduced and tested for convergence by using higher  $\mathbf{k}$ -point densities. An excellent discussion of this for aperiodic systems can be found in Ref. [26](#).

The most common schemes for generating  $\mathbf{k}$  points are the Chadi-Cohen scheme, [27](#) and the Monkhorst-Pack scheme. [28](#) The use of these  $\mathbf{k}$  point setups amounts to an expansion of the periodic function in reciprocal space, which allows a straight-forward interpolation of the function between the points that is more accurate than with other  $\mathbf{k}$  point generation schemes. [28](#)

## 2.4 Pseudopotentials

The core electrons of an atom are computationally expensive with planewave basis sets because they are highly localized. This means that a very large number of planewaves are required to expand their wave functions. Furthermore, the contributions of the core electrons to bonding compared to those of the valence electrons is usually negligible. In fact, the primary role of the core electron wave functions is to ensure proper orthogonality between the valence electrons and core states. Consequently, it is desirable to replace the atomic potential due to the core electrons with a pseudopotential that has the same effect on the valence electrons. [29](#) There are essentially two kinds of pseudopotentials, norm-conserving soft pseudopotentials [29](#) and Vanderbilt ultrasoft pseudopotentials. [30](#) In either case, the pseudopotential function is generated from an all-electron calculation of an atom in some reference state. In norm-conserving pseudopotentials, the charge enclosed in the pseudopotential region is the same as that enclosed by the same space in an all-electron calculation. In ultrasoft pseudopotentials, this requirement is relaxed and charge augmentation functions are used to make up the difference. As its name implies, this allows a “softer” pseudopotential to be generated, which means fewer planewaves are required to expand it.



The pseudopotentials are not unique, and calculated properties depend on them. However, there are standard methods for ensuring the quality and transferability (to different chemical environments) of the pseudopotentials.<sup>31</sup>

TODO: PAW description

VASP provides a database of PAW potentials.<sup>32;33</sup>

## 2.5 Fermi Temperature and band occupation numbers

At absolute zero, the occupancies of the bands of a system are well-defined step functions; all bands up to the Fermi level are occupied, and all bands above the Fermi level are unoccupied. There is a particular difficulty in the calculation of the electronic structures of metals compared to semiconductors and molecules. In molecules and semiconductors, there is a clear energy gap between the occupied states and unoccupied states. Thus, the occupancies are insensitive to changes in the energy that occur during the self-consistency cycles. In metals, however, the density of states is continuous at the Fermi level, and there are typically a substantial number of states that are close in energy to the Fermi level. Consequently, small changes in the energy can dramatically change the occupation numbers, resulting in instabilities that make it difficult to converge to the occupation step function. A related problem is that the Brillouin zone integral (which in practice is performed as a sum over a finite number of  $\mathbf{k}$  points) that defines the band energy converges very slowly with the number of  $\mathbf{k}$  points due to the discontinuity in occupancies in a continuous distribution of states for metals.<sup>12;34</sup> The difficulty arises because the temperature in most DFT calculations is at absolute zero. At higher temperatures, the DOS is smeared across the Fermi level, resulting in a continuous occupation function over the distribution of states. A finite-temperature version of DFT was developed,<sup>35</sup> which is the foundation on which one solution to this problem is based. In this solution, the step function is replaced by a smoothly varying function such as the Fermi-Dirac function at a small, but non-zero temperature.<sup>12</sup> The total energy is then extrapolated back to absolute zero.

## 2.6 Spin polarization and magnetism

There are two final points that need to be discussed about these calculations, spin polarization and dipole corrections. Spin polarization is important for systems that contain net spin. For example, iron, cobalt and nickel are magnetic because they have more electrons with spin “up” than spin “down” (or vice versa). Spin polarization must also be considered in atoms and molecules with unpaired electrons, such as hydrogen and oxygen atoms, oxygen molecules and radicals. For example, there are two spin configurations for an oxygen molecule, the singlet state with no unpaired electrons, and the triplet state with two unpaired electrons. The oxygen triplet state is lower in energy than the oxygen singlet state, and thus it corresponds to the ground state for an oxygen atom. A classically known problem involving spin polarization is the dissociation of a hydrogen molecule. In this case, the molecule starts with no net spin, but it dissociates into two atoms, each of which has an unpaired electron. See section 5.3.5 in Reference<sup>3</sup> for more details on this.

In VASP, spin polarization is not considered by default; it must be turned on, and an initial guess for the magnetic moment of each atom in the unit cell must be provided (typically about one Bohr-magneton per unpaired electron). For Fe, Co, and Ni, the experimental values are 2.22, 1.72, and 0.61 Bohr-magnetons, respectively<sup>4</sup> and are usually good initial guesses. See Reference<sup>31</sup> for a very thorough discussion of the determination of the magnetic properties of these

metals with DFT. For a hydrogen atom, an initial guess of 1.0 Bohr-magnetons (corresponding to one unpaired electron) is usually good. An oxygen atom has two unpaired electrons, thus an initial guess of 2.0 Bohr-magnetons should be used. The spin-polarized solution is sensitive to the initial guess, and typically converges to the closest solution. Thus, a magnetic initial guess usually must be provided to get a magnetic solution. Finally, unless an adsorbate is on a magnetic metal surface, spin polarization typically does not need to be considered, although the gas-phase reference state calculation may need to be done with spin-polarization.

The downside of including spin polarization is that it essentially doubles the calculation time.

## 2.7 TODO Recommended reading

This section needs cleaning up.

Original papers on DFT<sup>20;21</sup>

Kohn's Nobel Lecture,<sup>18</sup> Pople's Nobel Lecture<sup>22</sup>

<sup>7</sup> Hoffman

PAW in GPAW<sup>36</sup>

All calculations were performed using VASP<sup>12;37-39</sup> with the projector augmented wave (PAW) potentials provided in VASP .

Monkhorst-Pack k-points<sup>28</sup>

## 3 Molecules

In this chapter we consider how to construct models of molecules, how to manipulate them, and how to calculate many properties of molecules. For a nice comparison of VASP and Gaussian see<sup>40</sup>.

### 3.1 Defining and visualizing molecules

We start by learning how to define a molecule and visualize it. We will begin with defining molecules from scratch, then reading molecules from data files, and finally using some built-in databases in `ase`.

#### 3.1.1 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 2). We must define the type and position of each atom, and the unit cell the atoms are in.

---

```

1  from ase import Atoms, Atom
2  from ase.io import write
3
4  # define an Atoms object
5  atoms = Atoms([Atom('C',[0., 0.,0.]),
6                  Atom('O',[1.1,0.,0.]),
7                  cell=(10,10,10))
8
9  print 'V = {0:1.0f} Angstrom^3'.format(atoms.get_volume())
10
11 write('images/simple-cubic-cell.png',atoms,show_unit_cell=2)

```

---

$$V = 1000 \text{ \AA}^3$$

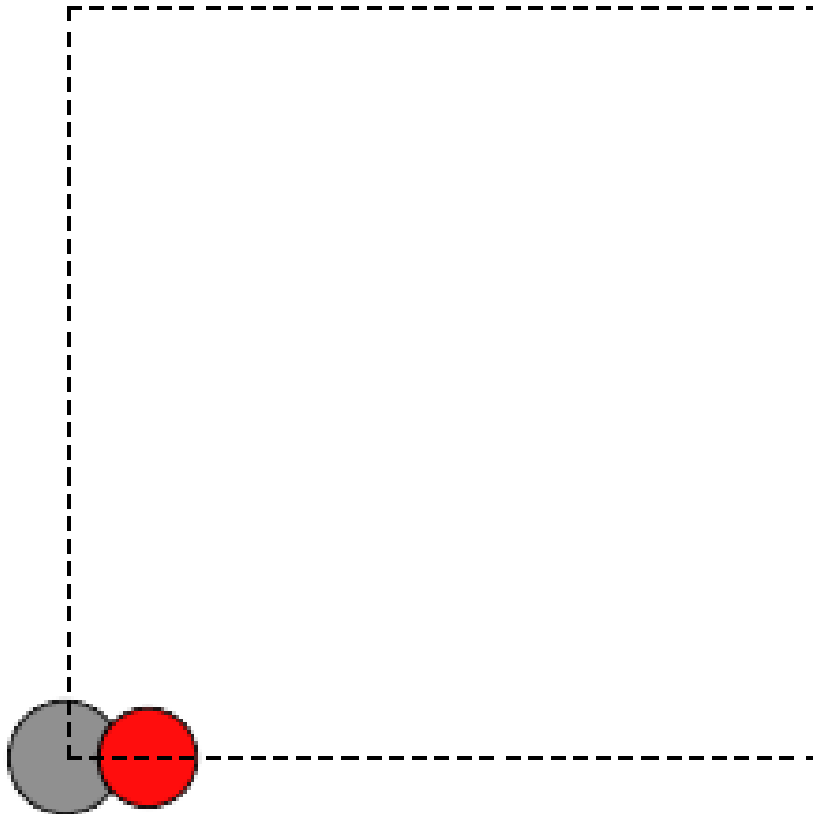


Figure 2: Image of a CO molecule with the C at the origin.

There are two inconvenient features of the simple cubic cell:

1. Since the CO molecule is at the corner, its electron density is spread over the 8 corners of the box, which is not convenient for visualization later (see [Visualizing electron density](#)).
2. Due to the geometry of the cube, you need fairly large cubes to make sure the electron density of the molecule does not overlap with that of its images. Electron-electron interactions are repulsive, and the overlap makes the energy increase significantly. Here, the CO molecule has 6 images due to periodic boundary conditions that are 10 Å away. The volume of the unit cell is 1000 Å<sup>3</sup>.

The first problem is easily solved by centering the atoms in the unit cell. The second problem can be solved by using a face-centered cubic lattice, which is the lattice with the closest packing. We show the results of the centering in Figure 3, where we have guessed values for  $b$  until the CO molecules are on average 10 Å apart. Note the final volume is only about 715 Å<sup>3</sup>, which is smaller than the cube. This will result in less computational time to compute properties.

---

```

1 from ase import Atoms, Atom
2 from ase.io import write

```

```

3
4 b = 7.1
5 atoms = Atoms([Atom('C',[0., 0., 0.]),
6               Atom('O',[1.1, 0., 0.])],
7               cell=[[b, b, 0.],
8                   [b, 0., b],
9                   [0., b, b]])
10 print 'V = {0:1.0f} Ang^3'.format(atoms.get_volume())
11
12 atoms.center() #translate atoms to center of unit cell
13 write('images/fcc-cell.png', atoms, show_unit_cell=2)

```

---

V = 716 Ang<sup>3</sup>

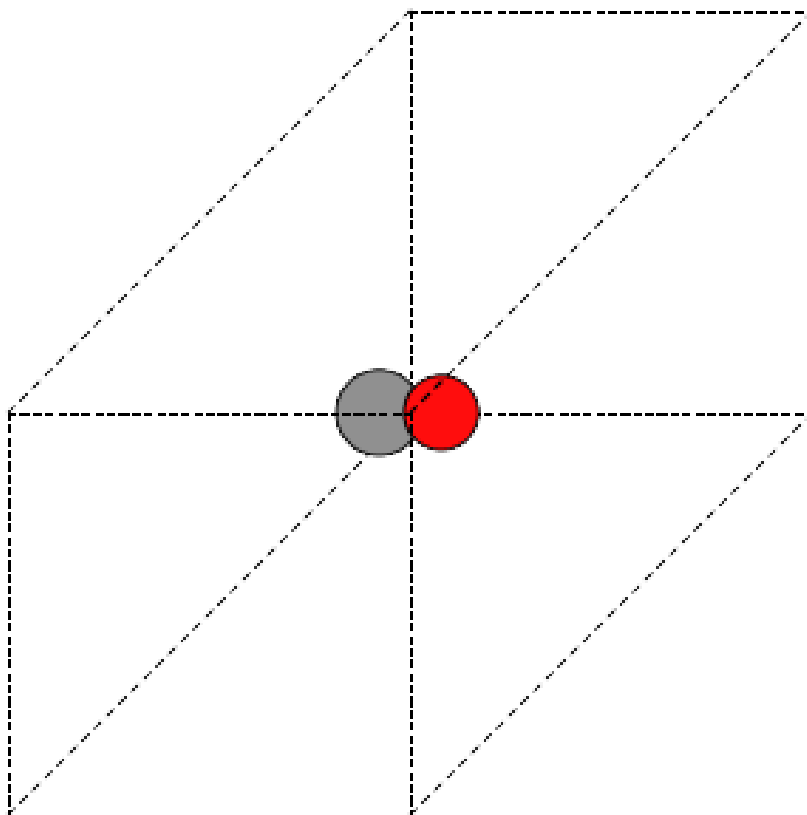


Figure 3: CO in a face-centered cubic unit cell.

At this point you might ask, “How do you know the distance to the neighboring image?” The **ag** viewer lets you compute this graphically, but we can use code to determine this too. All we have to do is figure out the length of each lattice vector, because these are what separate the atoms in the images. We use the **numpy** module to compute the distance of a vector as the square root of the sum of squared elements.

---

```

1 from ase import Atoms, Atom
2 import numpy as np
3

```

```

4  b = 7.1
5  atoms = Atoms([Atom('C',[0., 0.,0.]),
6                Atom('O',[1.1,0.,0.])],
7                cell=[[b, b, 0.],
8                      [b, 0., b],
9                      [0., b, b]])
10
11  # get unit cell vectors and their lengths
12  (a1, a2, a3) = atoms.get_cell()
13  print '|a1| = {0:1.2f} Ang'.format(np.sum(a1**2)**0.5)
14  print '|a2| = {0:1.2f} Ang'.format(np.linalg.norm(a2))
15  print '|a3| = {0:1.2f} Ang'.format(np.sum(a3**2)**0.5)

```

---

```

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

```

### 3.1.2 Reading other data formats into a calculation

ase.io.read supports many different file formats:

Known formats:

format	short name
=====	=====
GPAW restart-file	gpw
Dacapo netCDF output file	dacapo
Old ASE netCDF trajectory	nc
Virtual Nano Lab file	vnl
ASE pickle trajectory	traj
ASE bundle trajectory	bundle
GPAW text output	gpaw-text
CUBE file	cube
XCrySDen Structure File	xsf
Dacapo text output	dacapo-text
XYZ-file	xyz
VASP POSCAR/CONTCAR file	vasp
VASP OUTCAR file	vasp_out
SIESTA STRUCT file	struct_out
ABINIT input file	abinit
V_Sim ascii file	v_sim
Protein Data Bank	pdb
CIF-file	cif
FHI-aims geometry file	aims
FHI-aims output file	aims_out
VTK XML Image Data	vti
VTK XML Structured Grid	vti
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 file format to create `ase.Atoms` objects. Here is what an XYZ file format might look like:

14

```

C      0.0000000000000000      0.0000000000000000      0.3769490000000000
H      0.0000000000000000      0.0000000000000000      1.4752690000000000
C      0.0000000000000000      1.4502900000000000     -0.0962340000000000
H      0.0000000000000000      1.4939970000000000     -1.1908470000000000
H     -0.8854820000000000      1.9846950000000000      0.2612970000000000
H      0.8854820000000000      1.9846950000000000      0.2612970000000000
C      1.2559880000000000     -0.7251450000000000     -0.0962340000000000
H      1.2938390000000000     -0.7469980000000000     -1.1908470000000000
H      2.1615370000000000     -0.2254980000000000      0.2612970000000000
H      1.2760550000000000     -1.7591980000000000      0.2612970000000000
C     -1.2559880000000000     -0.7251450000000000     -0.0962340000000000
H     -1.2938390000000000     -0.7469980000000000     -1.1908470000000000
H     -1.2760550000000000     -1.7591980000000000      0.2612970000000000
H     -2.1615370000000000     -0.2254980000000000      0.2612970000000000

```

The first line is the number of atoms in the file. The second line is often a comment. What follows is one line per atom with the symbol and Cartesian coordinates in Å. Note that the XYZ format does not have unit cell information in it, so you will have to figure out a way to provide it. In this example, we center the atoms in a box with vacuum on all sides (Figure 4).

---

```

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

```

---

None

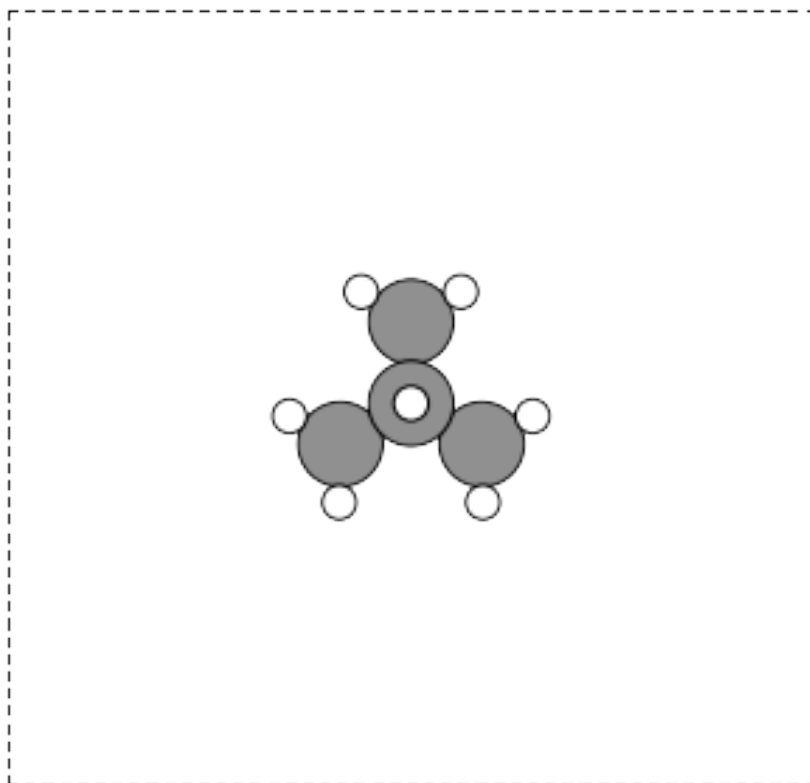


Figure 4: An isobutane molecule read in from an XYZ formatted data file.

### 3.1.3 Predefined molecules

`ase` defines a number of molecular geometries in the `ase.data.molecules` database. For example, the database includes the molecules in the G2/97 database.<sup>41</sup> This database contains a broad set of atoms and molecules for which good experimental data exists, making them useful for benchmarking studies. See [this site](#) for the original files.

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

---

```

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

```

---

isobutene	CH3CH2OH	CH3COOH
COF2	CH3NO2	CF3CN
CH3OH	CCH	CH3CH2NH2
PH3	Si2H6	O3
O2	BCl3	CH2_s1A1d
Be	H2CCl2	C3H9C

C3H9N	CH3CH2OCH3	BF3
CH3	CH4	S2
C2H6CHOH	SiH2_s1A1d	H3CNH2
CH3O	H	BeH
P	C3H4_C3v	C2F4
OH	methylenecyclopropane	F2O
SiCl4	HCF3	HCCl3
C3H7	CH3CH2O	AlF3
CH2NHCH2	SiH2_s3B1d	H2CF2
SiF4	H2CCO	PH2
OCS	HF	NO2
SH2	C3H4_C2v	H2O2
CH3CH2Cl	isobutane	CH3COF
HCOOH	CH3ONO	C5H8
2-butyne	SH	NF3
HOCl	CS2	P2
C	CH3S	O
C4H4S	S	C3H7Cl
H2CCHCl	C2H6	CH3CHO
C2H4	HCN	C2H2
C2Cl4	bicyclobutane	H2
C6H6	N2H4	C4H4NH
H2CCHCN	H2CCHF	cyclobutane
HCl	CH3OCH3	Li2
Na	CH3SiH3	NaCl
CH3CH2SH	OCHCHO	SiH4
C2H5	SiH3	NH
ClO	AlCl3	CCl4
NO	C2H3	ClF
HCO	CH3CONH2	CH2SCH2
CH3COCH3	C3H4_D2d	CH
CO	CN	F
CH3COCl	N	CH3Cl
Si	C3H8	CS
N2	Cl2	NCCN
F2	CO2	Cl
CH2OCH2	H2O	CH3CO
SO	HC(O)CH3	butadiene
ClF3	Li	PF3
B	CH3SH	CF4
C3H6_Cs	C2H6NH	N2O
LiF	H2COH	cyclobutene
LiH	SiO	Si2
C2H6SO	C5H5N	trans-butane
Na2	C4H4O	SO2
NH3	NH2	CH2_s3B1d



C1NO  
CH3SCH3

C3H6\_D3h  
H2CO

A1  
CH3CN

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

Here is an example of getting the geometry of an acetonitrile molecule and writing an image to a file. Note that the default unit cell is a  $1\text{\AA} \times 1\text{\AA} \times 1\text{\AA}$  cubic cell. That is too small to use if your calculator uses periodic boundary conditions. We center the atoms in the unit cell and add vacuum on each side. We will add 6  $\text{\AA}$  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 5.

---

```
1 from ase.data.molecules import molecule
2 from ase.io import write
3
4 atoms = molecule('CH3CN')
5
6 atoms.center(vacuum=6)
7 print 'unit cell'
8 print '-----'
9 print atoms.get_cell()
10
11 write('images/ch3cn.png', atoms, show_unit_cell=2)
```

---

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

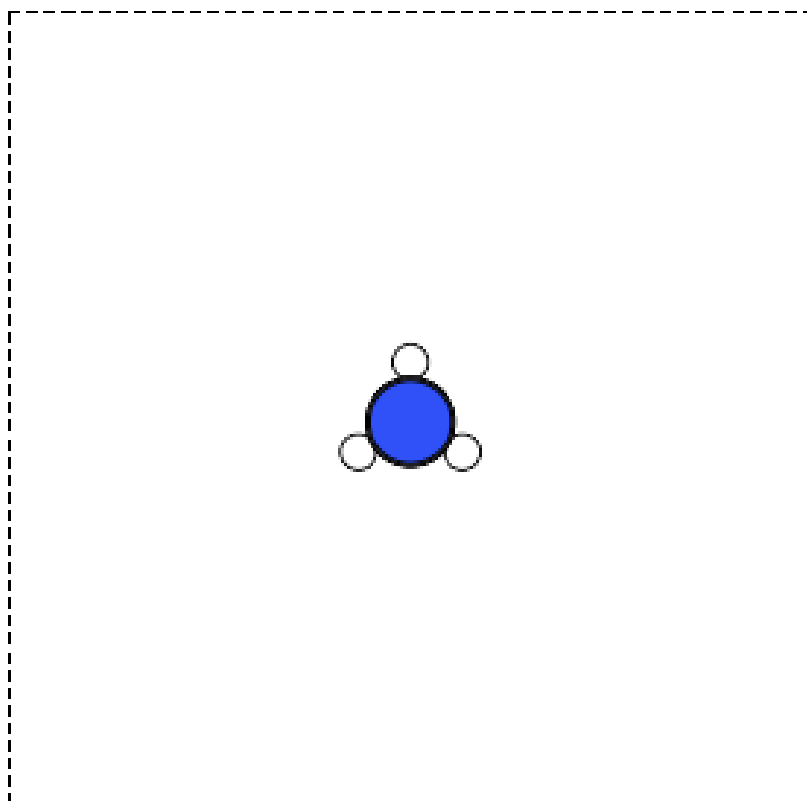


Figure 5: A  $\text{CH}_3\text{CN}$  molecule in a box.

It is possible to rotate the atoms with `ase.io.write` if you wanted to see pictures from another angle. In the next example we rotate 45 degrees about the  $x$ -axis, then 45 degrees about the  $y$ -axis. Note that this only affects the image, not the actual coordinates. See [Figure 6](#)

---

```

1 from ase.data.molecules import molecule
2 from ase.io import write
3
4 atoms = molecule('CH3CN')
5
6 atoms.center(vacuum=6)
7 print 'unit cell'
8 print '-----'
9 print atoms.get_cell()
10
11 write('images/ch3cn-rotated.png', atoms,
12       show_unit_cell=2, rotation='45x,45y,0z')
```

---

```

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

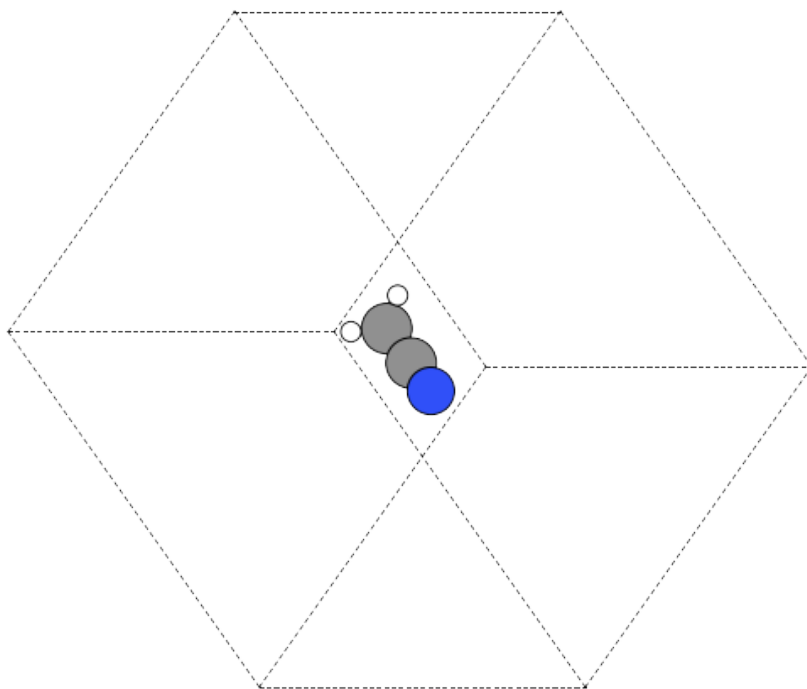


Figure 6: The rotated version of CH<sub>3</sub>CN.

If you actually want to rotate the coordinates, there is a nice way to do that too, with the `ase.Atoms.rotate` method. Actually there are some subtleties in rotation. One rotates the molecule an angle (in radians) around a vector, but you have to choose whether the center of mass should be fixed or not. You also must decide whether or not the unit cell should be rotated. 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.

---

```

1  from ase.data.molecules import molecule
2  from ase.io import write
3  from numpy import pi
4
5  atoms = molecule('CH3CN')
6  atoms.center(vacuum=6)
7  p1 = atoms.get_positions()
8
9  atoms.rotate('x', pi/4, center='COM', rotate_cell=False)
10 atoms.rotate('y', pi/4, center='COM', rotate_cell=False)
11
12 write('images/ch3cn-rotated-2.png', atoms, show_unit_cell=2)
13 print 'difference in positions after rotating'
14 print 'atom      difference vector'
15 print '-----'
16 p2 = atoms.get_positions()
17
18 diff = p2 - p1
19 for i, d in enumerate(diff):
20     print '{0} {1}'.format(i, d)

```

---

difference in positions after rotating

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

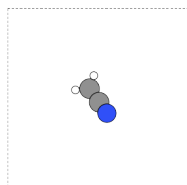


Figure 7: Rotated  $\text{CH}_3\text{CN}$  molecule

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

### 3.1.4 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 8. We set the `Atoms` about three Å apart using the `ase.Atoms.translate` function.

---

```

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

---

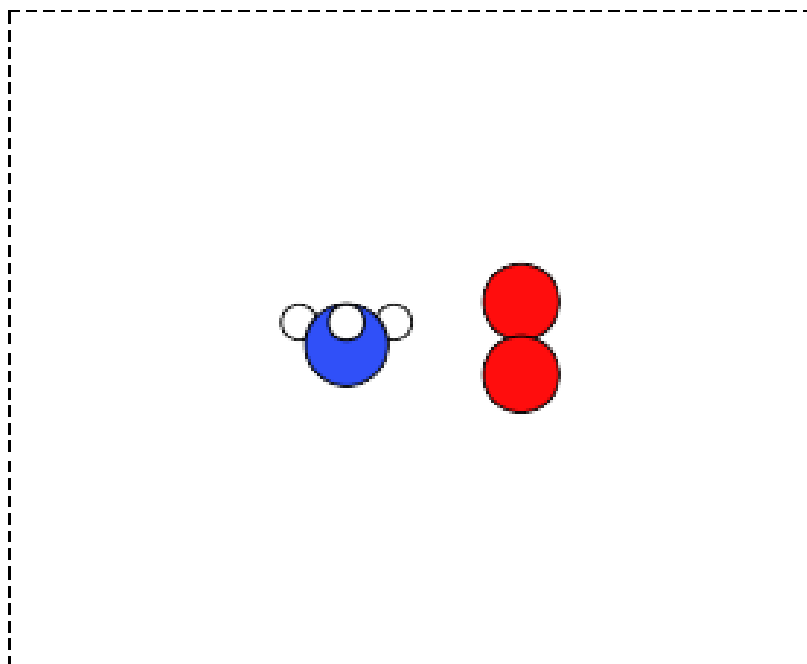


Figure 8: Image featuring ammonia and oxygen molecule in one unit cell.

## 3.2 Simple properties

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

### 3.2.1 Getting cartesian positions

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

---

```

1  from ase.data.molecules import molecule
2  from ase.io import write
3
4  atoms = molecule('C6H6') # benzene
5
6  # access properties on each atom
7  print ' # sym  p_x    p_y    p_z'
8  print '-----'
9  for i, atom in enumerate(atoms):
10     print '{0:3d}{1:^4s}{2:-8.2f}{3:-8.2f}{4:-8.2f}'.format(i,
11                                                             atom.symbol,
12                                                             atom.x,
13                                                             atom.y,
14                                                             atom.z)
15
16  # get all properties in arrays
17  sym = atoms.get_chemical_symbols()
18  pos = atoms.get_positions()
19  num = atoms.get_atomic_numbers()
20
21  atom_indices = range(len(atoms))

```

```

22
23 print
24 print ' # sym      at#      p_x      p_y      p_z'
25 print '-----',
26 for i, s, n, p in zip(atom_indices, sym, num, pos):
27     px, py, pz = p
28     print '{0:3d}{1:>3s}{2:8d}{3:-8.2f}{4:-8.2f}{5:-8.2f}'.format(i, s, n,
29                                                                    px, py, pz)

```

---

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

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

### 3.2.2 Molecular weight and molecular formula

We can quickly compute the molecular weight of a molecule with this recipe. We use `ase.Atoms.get_masses` to get an array of the atomic masses of each atom in the `Atoms` object, and then just sum them up.

---

```

1 from ase.data.molecules import molecule
2
3 atoms = molecule('C6H6')
4 masses = atoms.get_masses()
5

```

```

6 molecular_weight = masses.sum()
7 molecular_formula = atoms.get_chemical_symbols(reduce=True)
8 print 'The molecular weight of {0} is {1:1.2f} gm/mol'.format(molecular_formula,
9                                                                molecular_weight)

```

---

The molecular weight of C6H6 is 78.11 gm/mol

Note that the argument `reduce=True` for `ase.Atoms.get_chemical_symbols` collects all the symbols to provide a molecular formula.

### 3.2.3 Center of mass

The center of mass (COM) is defined as:

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

The center of mass is essentially the average position of the atoms, weighted by the mass of each atom. Here is an example of getting the center of mass from an `Atoms` object using `ase.Atoms.get_center_of_mass`.

---

```

1 from ase.structure import molecule
2 from ase.io import write
3 import numpy as np
4
5 # ammonia
6 atoms = molecule('NH3')
7
8 print 'COM1 = {0}'.format(atoms.get_center_of_mass()) # cartesian coordinates
9
10 # compute the center of mass by hand
11 pos = atoms.positions
12 masses = atoms.get_masses()
13
14 COM = np.array([0., 0., 0.])
15 for m,p in zip(masses, pos):
16     COM += m*p
17 COM /= masses.sum()
18
19 print 'COM2 = {0}'.format(COM)
20
21 # one-line linear algebra definition of COM
22 print 'COM3 = {0}'.format(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 that these centers of mass, which are calculated by different methods, are the same.

### 3.2.4 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 others may be degenerate. There is a convenient function to get the moments of inertia: `ase.Atoms.get_moments_of_inertia`. Here are several examples of molecules with different types of symmetry.:

---

```

1  from ase.structure import molecule
2
3  print 'linear rotors: I = [0 Ia Ia]'
4  atoms = molecule('CO2')
5  print '  CO2 moments of inertia: ',atoms.get_moments_of_inertia()
6  print
7
8  print 'symmetric rotors (Ia = Ib) < Ic'
9  atoms = molecule('NH3')
10 print '  NH3 moments of inertia: ',atoms.get_moments_of_inertia()
11
12 atoms = molecule('C6H6')
13 print '  C6H6 moments of inertia: ',atoms.get_moments_of_inertia()
14 print
15
16 print 'symmetric rotors Ia < (Ib = Ic)'
17 atoms = molecule('CH3Cl')
18 print '  CH3Cl moments of inertia: ',atoms.get_moments_of_inertia()
19 print
20
21 print 'spherical rotors Ia = Ib = Ic'
22 atoms = molecule('CH4')
23 print '  CH4 moments of inertia: ',atoms.get_moments_of_inertia()
24 print
25
26 print 'unsymmetric rotors Ia != Ib != Ic'
27 atoms = molecule('C3H7Cl')
28 print '  C3H7Cl moments of inertia: ',atoms.get_moments_of_inertia()

```

---

linear rotors: I = [0 Ia Ia]

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)

CH3Cl moments of inertia: [ 3.20372189 37.97009644 37.97009837]

spherical rotors Ia = Ib = Ic

CH4 moments of inertia: [ 3.19145621 3.19145621 3.19145621]

unsymmetric rotors Ia != Ib != Ic

C3H7Cl moments of inertia: [ 19.41351508 213.18961963 223.16255537]

If you want to know the principle axes of rotation, we simply pass `vectors=True` to the function, and it returns the moments of inertia and the principle axes.

---

```

1  from ase.structure import molecule
2  import numpy as np
3
4  atoms = molecule('CH3Cl')
5  moments, axes = atoms.get_moments_of_inertia(vectors=True)
6  print 'Moments = {0}'.format(moments)
7  print 'axes = {0}'.format(axes)

```

---



```

Moments = [ 3.20372189  37.97009644  37.97009837]
axes = [[ 0.  0.  1.]
        [ 0.  1.  0.]
        [ 1.  0.  0.]]

```

This shows the first moment is about the z-axis, the second moment is about the y-axis, and the third moment is about the x-axis.

### 3.2.5 Computing bond lengths and angles

A typical question we might ask is, “What is the structure of a molecule?” In other words, what are the bond lengths, angles between bonds, and similar properties. The `Atoms` object contains an `ase.Atoms.get_distance` method to make this easy. To calculate the distance between two atoms, you have to specify their indices, remembering that the index starts at 0.

---

```

1 from ase.structure import molecule
2 from ase.io import write
3
4 # ammonia
5 atoms = molecule('NH3')
6
7 print 'atom symbol'
8 print '====='
9 for i, atom in enumerate(atoms):
10     print '{0:2d} {1:3s}'.format(i, atom.symbol)
11
12 # N-H bond length
13 s = 'The N-H distance is {0:1.3f} angstroms'
14 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 two atoms, we could use some simple trigonometry to compute the angle between the vectors:  $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos(\theta)$ . So we can calculate the angle as  $\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}\right)$ , we just have to define our two vectors  $\vec{a}$  and  $\vec{b}$ . We compute these vectors as the difference in positions of two atoms. For example, here we compute the angle H-N-H in an ammonia molecule. This is the angle between N-H<sub>1</sub> and N-H<sub>2</sub>. In the next example, we utilize functions in `numpy` to perform the calculations, specifically the `numpy.arccos` function, the `numpy.dot` function, and `numpy.linalg.norm` functions.

---

```

1 from ase.structure import molecule
2 from ase.io import write
3
4 # ammonia

```

---

```

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

```

---

```

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

```

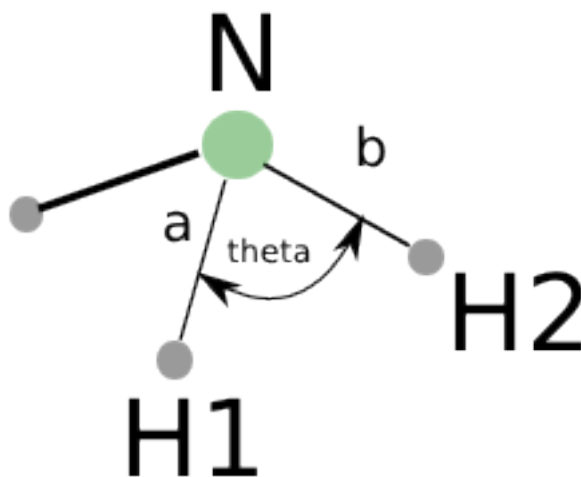


Figure 9: Schematic of the vectors defining the H-N-H angle.

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

---

```

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

```

---

theta = 106.334624232 degrees

**Dihedral angles** There is support in ase for computing [dihedral angles](#). Let us illustrate that for ethane. We will compute the dihedral angle between atoms 5, 1, 0, and 4. That is a H-C-C-H dihedral angle, and one can visually see (although not here) that these atoms have a dihedral angle of 60° (Figure 10).

---

```

1 # calculate an ethane dihedral angle
2 from ase.structure import molecule
3 import numpy as np
4
5 atoms = molecule('C2H6')
6
7 print 'atom symbol'
8 print '======'
9 for i, atom in enumerate(atoms):
10     print '{0:2d} {1:3s}'.format(i,atom.symbol)
11
12 da = atoms.get_dihedral([5,1,0,4])*180./np.pi
13 print 'dihedral angle = {0:1.2f} degrees'.format(da)

```

---

atom symbol

=====

```

0  C
1  C
2  H
3  H
4  H
5  H
6  H
7  H

```

dihedral angle = 60.00 degrees



Figure 10: Schematic of the calculated ethane dihedral angle.

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

### 3.3 Simple properties that require single computations

There are many properties that only require a single DFT calculation to obtain the energy, forces, density of states, electron density and electrostatic potential. This section describes some of these calculations and their analysis.

#### 3.3.1 Energy and forces

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

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

##### Note:

This is your first DFT calculation in the book! See [ISMEAR](#), [SIGMA](#), [NBANDS](#), and [ENCUT](#) to learn more about these VASP keywords.

---

```
1 from ase import Atoms, Atom
2 from jasp import *
3 import numpy as np
4 np.set_printoptions(precision=3, suppress=True)
5
6 co = Atoms([Atom('C',[0, 0, 0]),
7             Atom('O',[1.2, 0, 0])],
8            cell=(6., 6., 6.))
9
10 with jasp('molecules/simple-co', #output dir
11          xc='PBE', # the exchange-correlation functional
12          nbands=6, # number of bands
13          encut=350, # planewave cutoff
14          ismear=1, # Methfessel-Paxton smearing
15          sigma=0.01, # very small smearing factor for a molecule
16          atoms=co) as calc:
17     print 'energy = {0} eV'.format(co.get_potential_energy())
18     print co.get_forces()
```

---

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

**Convergence with unit cell size** There are a number of parameters that affect the energy and forces including the calculation parameters and the unit cell. We will first consider the effect of the unit cell on the total energy and forces. The reason that the unit cell affects the total energy is that it can change the distribution of electrons in the molecule.

---

```
1 from jasp import *
2 from ase import Atoms, Atom
3 import numpy as np
```

```

4  np.set_printoptions(precision=3, suppress=True)
5
6  atoms = Atoms([Atom('C',[0, 0, 0]),
7                Atom('O',[1.2, 0, 0])])
8
9  L = [4, 5, 6, 8, 10]
10
11 volumes, energies = [], []
12 for a in L:
13     atoms.set_cell([a,a,a], scale_atoms=False)
14     atoms.center()
15     with jasp('molecules/co-L-{0}'.format(a),
16             encut=350,
17             xc='PBE',
18             atoms=atoms) as calc:
19         try:
20             energies.append(atoms.get_potential_energy())
21         except (VaspSubmitted, VaspQueued):
22             pass
23 import matplotlib.pyplot as plt
24 plt.plot(L, energies, 'bo-')
25 plt.xlabel('Unit cell length ($\AA$)')
26 plt.ylabel('Total energy (eV)')
27 plt.savefig('images/co-e-v.png')
28 plt.show()

```

---

None

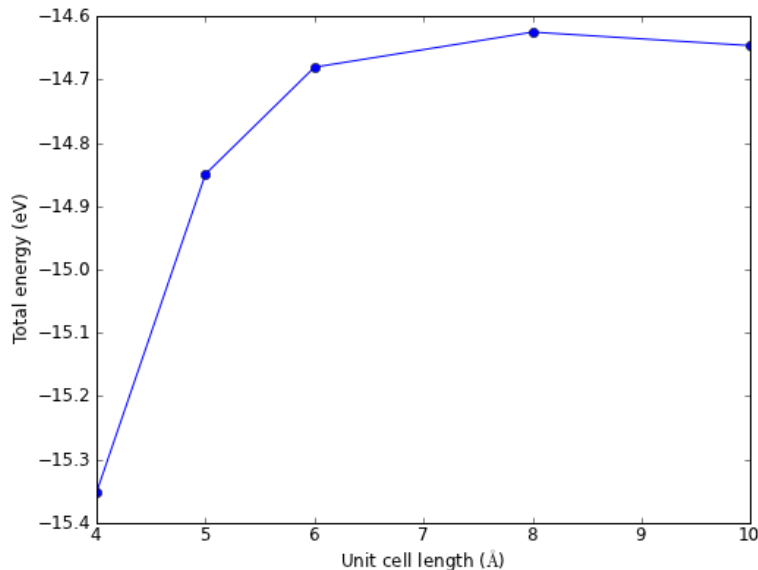


Figure 11: Total energy of a CO molecule as a function of the unit cell length.

Here there are evidently attractive interactions between the CO molecules which lower the total energy for small box sizes. We have to decide what an appropriate volume for our calculation is, and the choice depends on the goal. We may wish to know the total energy of a molecule that is not interacting with any other molecules, e.g. in the ideal gas limit. In that

case we need a large unit cell so the electron density from the molecule does not go outside the unit cell where it would overlap with neighboring images.

Let us consider what the pressure in the unit cell is. In the ideal gas limit we have  $PV = nRT$ , which gives a pressure of zero at absolute zero. At non-zero temperatures, we have  $P = n/VRT$ . Let us consider some examples. In atomic units we use  $k_B$  instead of  $R$ , and it has a value of

---

```

1  from ase.units import kB, Pascal
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  atm = 101325 * Pascal
6
7  L = np.linspace(4,10)
8  V = L**3
9
10 n = 1 # one atom per unit cell
11
12 for T in [298, 600, 1000]:
13     P = n/V*kB*T / atm # convert to atmospheres
14
15     plt.plot(V, P, label='{0}K'.format(T))
16
17 plt.xlabel('Unit cell volume (Å3)')
18 plt.ylabel('Pressure (atm)')
19 plt.legend(loc='best')
20 plt.savefig('images/ideal-gas-pressure.png')
21 plt.show()

```

---

None

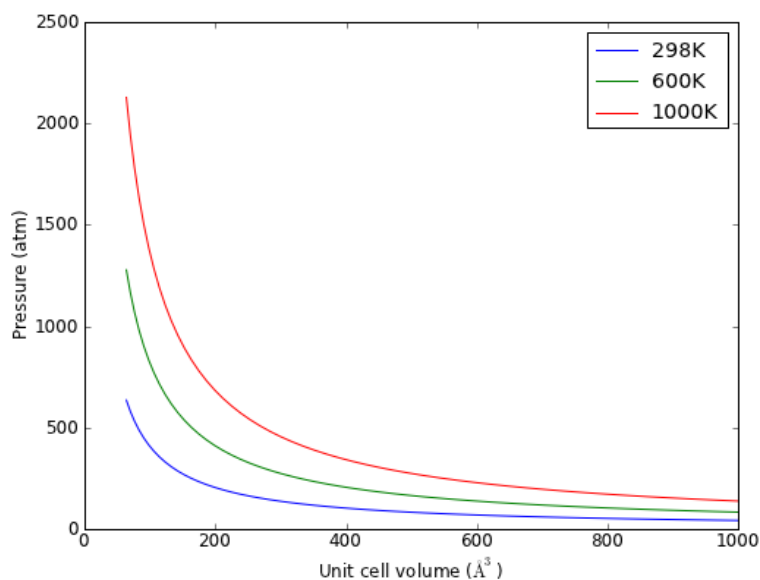


Figure 12: Ideal gas pressure dependence on temperature and unit cell volume.

**Convergence of ENCUT** The total energy and forces also depend on the computational parameters, notably [ENCUT](#).

---

```
1 from ase import Atoms, Atom
2 from jasp import *
3 import numpy as np
4 np.set_printoptions(precision=3, suppress=True)
5
6 atoms = Atoms([Atom('C', [0, 0, 0]),
7               Atom('O', [1.2, 0, 0])],
8               cell=(6,6,6))
9 atoms.center()
10
11 ENCUTS = [250, 300, 350, 400, 450, 500]
12
13 energies = []
14 ready = True
15 for en in ENCUTS:
16     with jasp('molecules/co-en-{0}'.format(en),
17             encut=en,
18             xc='PBE',
19             atoms=atoms) as calc:
20         try:
21             energies.append(atoms.get_potential_energy())
22         except (VaspSubmitted, VaspQueued):
23             ready = False
24
25 if not ready:
26     import sys; sys.exit()
27
28 import matplotlib.pyplot as plt
29 plt.plot(ENCUTS, energies, 'bo-')
30 plt.xlabel('ENCUT (eV)')
31 plt.ylabel('Total energy (eV)')
32 plt.savefig('images/co-encut-v.png')
```

---

None

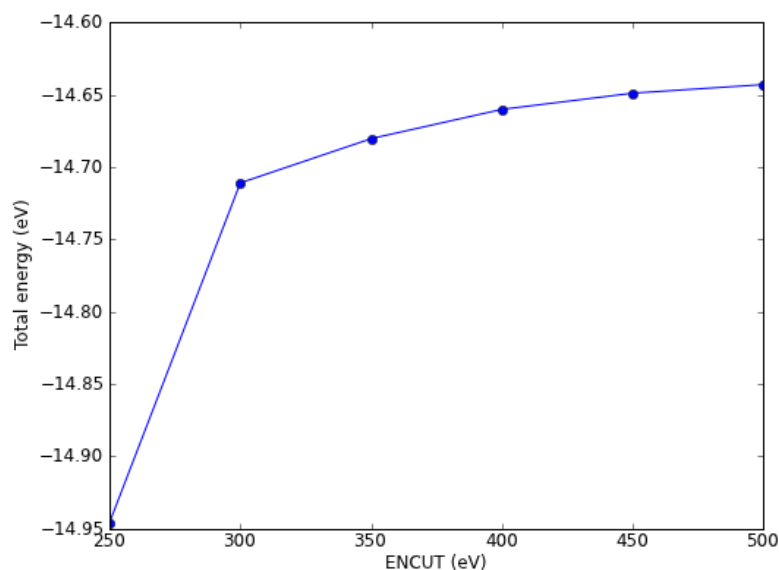


Figure 13: Dependence of the total energy of CO molecule on ENCUT.

You can see in this figure that it takes a cutoff energy of about 400 eV to achieve a convergence level around 10 meV, and that even at 500 meV the energy is still changing slightly. Keep in mind that we are generally interested in differences in total energy, and the differences tend to converge faster than a single total energy. Also it is important to note that it is usually a single element that determines the rate of convergence. The reason we do not just use very high ENCUT all the time is it is expensive.

---

```
1 grep "Elapsed time (sec):" molecules/co-en-*/OUTCAR
```

---

molecules/co-en-250/OUTCAR: Elapsed time (sec): 4.954 molecules/co-en-300/OUTCAR: Elapsed time (sec): 7.662 molecules/co-en-350/OUTCAR: Elapsed time (sec): 8.232 molecules/co-en-400/OUTCAR: Elapsed time (sec): 11.148 molecules/co-en-450/OUTCAR: Elapsed time (sec): 12.592 molecules/co-en-500/OUTCAR: Elapsed time (sec): 15.425

Although here the times do not seem that unreasonable, note we increased the time by a factor of three by simply doubling ENCUT. If the first time was a week, this is a significant increase in time!

### 3.3.2 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 14) of plotting the charge density, and we consider some issues we have to consider when visualizing volumetric data in unit cells with periodic boundary conditions. We will use the results from a previous calculation.

---

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

---



```

3  from ase.data import vdw_radii
4  from ase.data.colors import cpk_colors
5
6  with jasp('molecules/simple-co') as calc:
7      atoms = calc.get_atoms()
8      x, y, z, cd = calc.get_charge_density()
9
10 mlab.figure(1, bgcolor=(1, 1, 1)) # make a white figure
11
12 # plot the atoms as spheres
13 for atom in atoms:
14     mlab.points3d(atom.x,
15                   atom.y,
16                   atom.z,
17                   scale_factor=vdw_radii[atom.number]/5., #this determines the size of the atom
18                   resolution=20,
19                   # a tuple is required for the color
20                   color=tuple(cpk_colors[atom.number]),
21                   scale_mode='none')
22
23 # draw the unit cell - there are 8 corners, and 12 connections
24 a1, a2, a3 = atoms.get_cell()
25 origin = [0, 0, 0]
26 cell_matrix = [[origin, a1],
27                [origin, a2],
28                [origin, a3],
29                [a1, a1 + a2],
30                [a1, a1 + a3],
31                [a2, a2 + a1],
32                [a2, a2 + a3],
33                [a3, a1 + a3],
34                [a3, a2 + a3],
35                [a1 + a2, a1 + a2 + a3],
36                [a2 + a3, a1 + a2 + a3],
37                [a1 + a3, a1 + a3 + a2]]
38
39 for p1, p2 in cell_matrix:
40     mlab.plot3d([p1[0], p2[0]], # x-positions
41                [p1[1], p2[1]], # y-positions
42                [p1[2], p2[2]], # z-positions
43                tube_radius=0.02)
44
45 # Now plot the charge density
46 mlab.contour3d(x, y, z, cd)
47 mlab.view(azimuth=-90, elevation=90, distance='auto')
48
49 mlab.savefig('images/co-cd.png')
50 mlab.show()

```

---

None

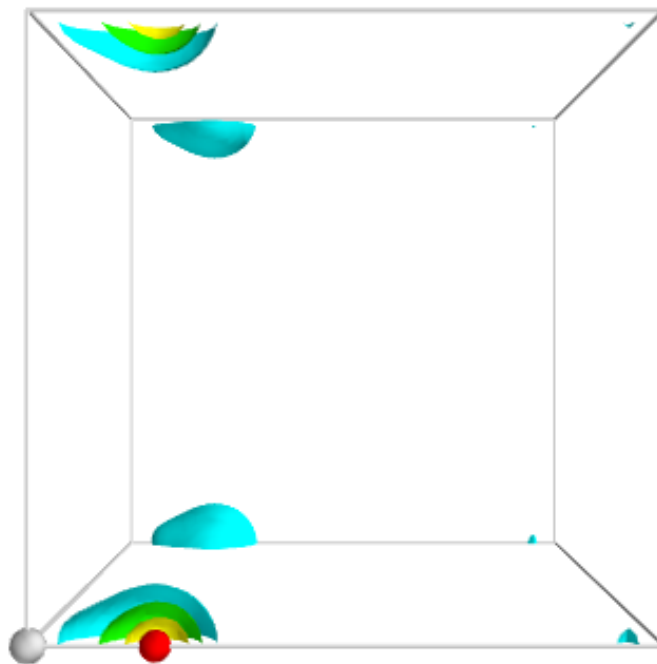


Figure 14: Charge density of a CO molecule that is located at the origin. The electron density that is outside the cell is wrapped around to the other corners.

If we take care to center the CO molecule in the unit cell, we get a nicer looking result.

---

```

1  from jasp import *
2  from enthought.mayavi import mlab
3  from ase.data import vdw_radii
4  from ase.data.colors import cpk_colors
5  from ase import Atom, Atoms
6
7  atoms = Atoms([Atom('C', [2.422, 0.0, 0.0]),
8                  Atom('O', [3.578, 0.0, 0.0])],
9                cell=(10,10,10))
10
11  atoms.center()
12
13  with jasp('molecules/co-centered',
14            encut=350,
15            xc='PBE',
16            atoms=atoms) as calc:
17
18      atoms = calc.get_atoms()
19      x, y, z, cd = calc.get_charge_density()
20
21  mlab.figure(bgcolor=(1, 1, 1))
22
23  # plot the atoms as spheres
24  for atom in atoms:
25      mlab.points3d(atom.x,
26                    atom.y,
27                    atom.z,
28                    scale_factor=vdw_radii[atom.number]/5.,
29                    resolution=20,
30                    # a tuple is required for the color
31                    color=tuple(cpk_colors[atom.number]),

```

```

32         scale_mode='none')
33
34 # draw the unit cell - there are 8 corners, and 12 connections
35 a1, a2, a3 = atoms.get_cell()
36 origin = [0, 0, 0]
37 cell_matrix = [[origin, a1],
38                [origin, a2],
39                [origin, a3],
40                [a1, a1 + a2],
41                [a1, a1 + a3],
42                [a2, a2 + a1],
43                [a2, a2 + a3],
44                [a3, a1 + a3],
45                [a3, a2 + a3],
46                [a1 + a2, a1 + a2 + a3],
47                [a2 + a3, a1 + a2 + a3],
48                [a1 + a3, a1 + a3 + a2]]
49
50 for p1, p2 in cell_matrix:
51     mlab.plot3d([p1[0], p2[0]], # x-positions
52                [p1[1], p2[1]], # y-positions
53                [p1[2], p2[2]], # z-positions
54                tube_radius=0.02)
55
56
57 # Now plot the charge density
58 mlab.contour3d(x, y, z, cd, transparent=True)
59
60 # this view was empirically found by iteration
61 mlab.view(azimuth=-90, elevation=90, distance='auto')
62
63 mlab.savefig('images/co-centered-cd.png')
64 mlab.show()

```

---

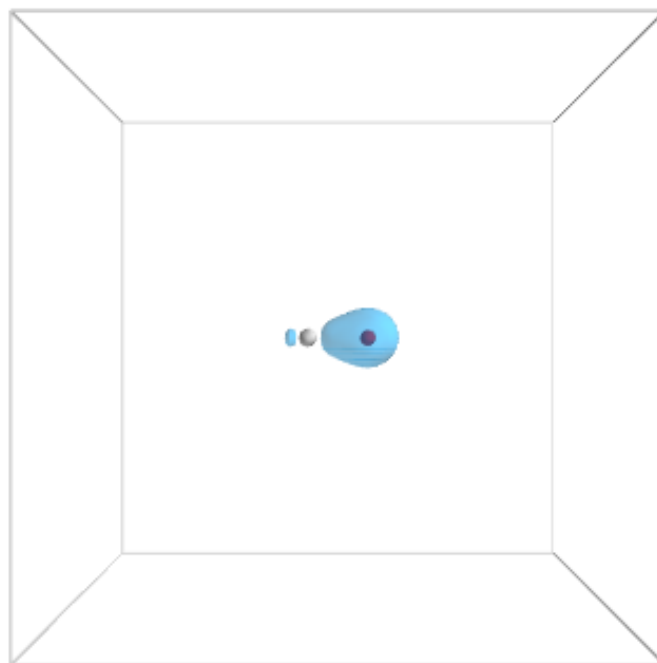


Figure 15: Charge density of a CO molecule centered in the unit cell. Now the electron density is centered in the unit cell.

### 3.3.3 Dipole moments

The **dipole moment** is a vector describing the separation of electrical (negative) and nuclear (positive) charge. The magnitude of this vector is the dipole moment, which has units of Coulomb-meter, or more commonly Debye. The symmetry of a molecule determines if a molecule has a dipole moment or not. Below we compute the dipole moment of CO. We must integrate the electron density to find the center of electrical charge, and compute a sum over the nuclei to find the center of positive charge.

---

```
1 from jasp import *
2 from ase.units import Debye
3
4 with jasp('molecules/co-centered') as calc:
5     atoms.get_potential_energy()
6
7     vcd = VaspChargeDensity()
8
9     cd = np.array(vcd.chg[0])
10    n0, n1, n2 = cd.shape
11
12    s0 = 1.0/n0
13    s1 = 1.0/n1
14    s2 = 1.0/n2
15
16    X, Y, Z = np.mgrid[0.0:1.0:s0,
17                        0.0:1.0:s1,
18                        0.0:1.0:s2]
19
20    C = np.column_stack([X.ravel(),
21                        Y.ravel(),
22                        Z.ravel()])
23
24    atoms = calc.get_atoms()
25    uc = atoms.get_cell()
26    real = np.dot(C, uc)
27
28    #now convert arrays back to unitcell shape
29    x = np.reshape(real[:, 0], (n0, n1, n2))
30    y = np.reshape(real[:, 1], (n0, n1, n2))
31    z = np.reshape(real[:, 2], (n0, n1, n2))
32
33    nelements = n0 * n1 * n2
34    voxel_volume = atoms.get_volume() / nelements
35    total_electron_charge = -cd.sum() * voxel_volume
36
37
38    electron_density_center = np.array([(cd * x).sum(),
39                                       (cd * y).sum(),
40                                       (cd * z).sum()])
41    electron_density_center *= voxel_volume
42    electron_density_center /= total_electron_charge
43
44    electron_dipole_moment = -electron_density_center * total_electron_charge
45
46    # now the ion charge center. We only need the Zval listed in the potcar
47    from jasp.POTCAR import get_ZVAL
48
49    LOP = calc.get_pseudopotentials()
50    ppp = os.environ['VASP_PP_PATH']
51
52    zval = {}
53    for sym, ppath, hash in LOP:
54        fullpath = os.path.join(ppp, ppath)
```

```

55         z = get_ZVAL(fullpath)
56         zval[sym] = z
57     ion_charge_center = np.array([0.0, 0.0, 0.0])
58     total_ion_charge = 0.0
59     for atom in atoms:
60         Z = zval[atom.symbol]
61         total_ion_charge += Z
62         pos = atom.position
63         ion_charge_center += Z*pos
64
65     ion_charge_center /= total_ion_charge
66     ion_dipole_moment = ion_charge_center * total_ion_charge
67
68     dipole_vector = (ion_dipole_moment + electron_dipole_moment)
69
70     dipole_moment = ((dipole_vector**2).sum())**0.5 / Debye
71
72     print 'The dipole moment is {0:1.2f} Debye'.format(dipole_moment)

```

---

The dipole moment is 0.10 Debye

Note that a function using the code above exists in `jasp` which makes it trivial to compute the dipole moment. Here is an example of its usage.

```

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

```

---

The dipole moment is 0.10 Debye

### 3.3.4 The density of states (DOS)

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

#### Warning:

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

```

1  from jasp import *
2  from ase.dft.dos import DOS
3  import matplotlib.pyplot as plt
4
5  with jasp('molecules/simple-co') as calc: # we already ran this!
6      dos = DOS(calc)
7      plt.plot(dos.get_energies(), dos.get_dos())
8      plt.xlabel('Energy - $E_f$ (eV)')
9      plt.ylabel('DOS')
10

```

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

---

None

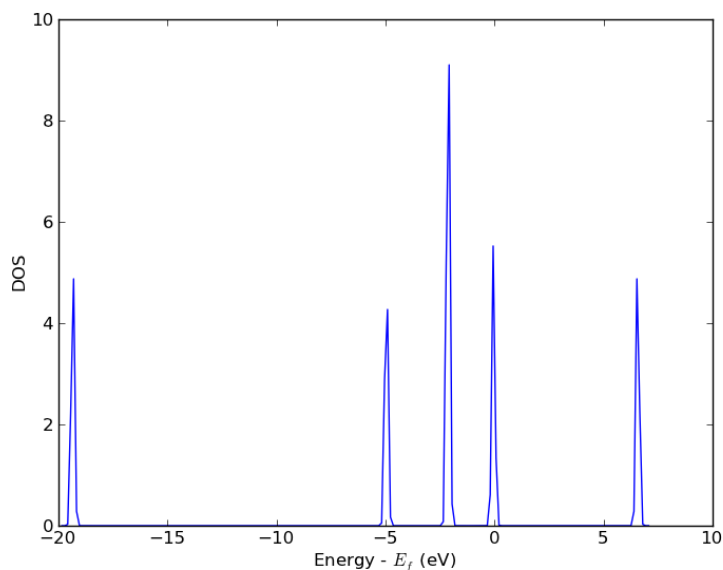


Figure 16: Density of states for a CO molecule.

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

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 you risk losing the results of the first step. That can make it difficult to reproduce a result. We advocate our approach of using multiple directories for the subsequent calculations, because it leaves a clear trail of how the work was done.

**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.

You can also get spin-polarized atom-projected DOS, and magnetization projected DOS. See [http://cms.mpi.univie.ac.at/vasp/vasp/DOSCAR\\_-file.html#doscar](http://cms.mpi.univie.ac.at/vasp/vasp/DOSCAR_-file.html#doscar) for more details.

---

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

---

None

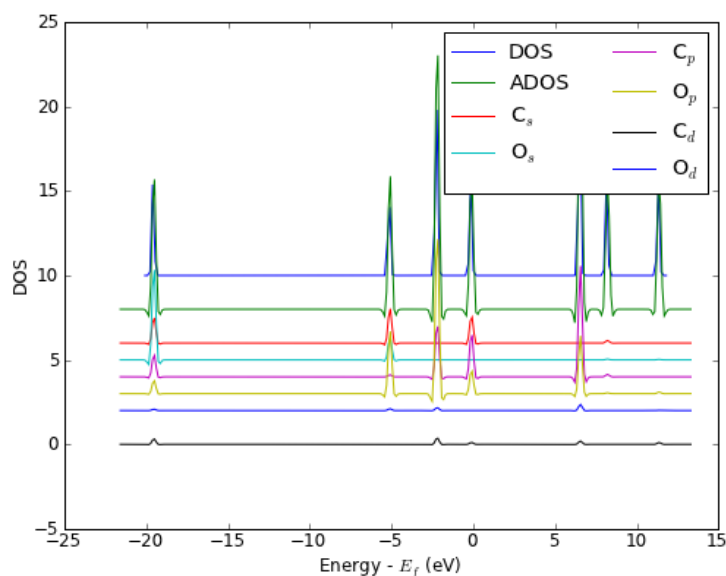


Figure 17: Atom-projected DOS for a CO molecule. The total density of states and the  $s$ ,  $p$  and  $d$  states on the C and O are shown.

### 3.3.6 TODO Electrostatic potential

This is an example of the so-called  $\sigma$  hole in a [halogen bond](http://cccbdb.nist.gov/exp2.asp?casno=75638). The coordinates for the  $\text{CF}_3\text{Br}$  molecule were found at <http://cccbdb.nist.gov/exp2.asp?casno=75638>.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  from ase.io import write
4
5  from enthought.mayavi import mlab
6  from ase.data import vdw_radii
7  from ase.data.colors import cpk_colors
8
9  atoms = Atoms([Atom('C', [ 0.0000, 0.0000, -0.8088]),
10                Atom('Br', [ 0.0000, 0.0000, 1.1146]),
11                Atom('F', [ 0.0000, 1.2455, -1.2651]),
12                Atom('F', [ 1.0787, -0.6228, -1.2651]),
13                Atom('F', [-1.0787, -0.6228, -1.2651])],
14                cell=(10, 10, 10))
15  atoms.center()
16
17  with jasp('molecules/CF3Br',
18            encut=350,
19            xc='PBE',
20            ibrion=1,
21            nsw=50,
22            lvtot=True,
23            lvhar=True,
24            atoms=atoms) as calc:
25      calc.set_nbands(f=2)
26      calc.calculate()
27
28      x, y, z, lp = calc.get_local_potential()
29      x, y, z, cd = calc.get_charge_density()

```



```

30
31 mlab.figure(1, bgcolor=(1, 1, 1)) # make a white figure
32
33 # plot the atoms as spheres
34 for atom in atoms:
35     mlab.points3d(atom.x,
36                   atom.y,
37                   atom.z,
38                   scale_factor=vdw_radii[atom.number]/5.,
39                   resolution=20,
40                   # a tuple is required for the color
41                   color=tuple(cpk_colors[atom.number]),
42                   scale_mode='none')
43 # plot the bonds. We want a line from C-Br, C-F, etc...
44 # We create a bond matrix showing which atoms are connected.
45 bond_matrix = [[0, 1],
46               [0, 2],
47               [0, 3],
48               [0, 4]]
49
50 for a1, a2 in bond_matrix:
51     mlab.plot3d(atoms.positions[[a1,a2], 0], # x-positions
52               atoms.positions[[a1,a2], 1], # y-positions
53               atoms.positions[[a1,a2], 2], # z-positions
54               [2, 2],
55               tube_radius=0.02,
56               colormap='Reds')
57
58 mlab.contour3d(x, y, z, lp)
59 mlab.savefig('images/halogen-ep.png')
60 mlab.show()

```

---

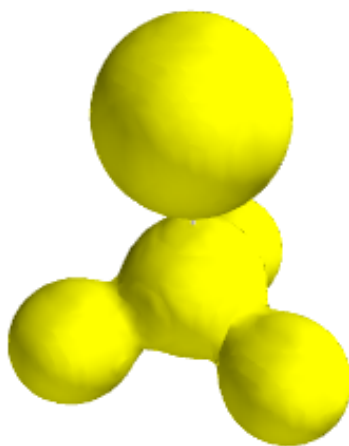


Figure 18: Plot of the electrostatic potential of  $\text{CF}_3\text{Br}$ . TODO: figure out how to do an isosurface of charge, colormapped by the local potential.

See <http://www.uni-due.de/~hp0058/?file=manual03.html&dir=vmdplugins> for examples of using **VMD** for visualization.

### 3.3.7 Bader analysis

Bader analysis is a charge partitioning scheme where charge is divided by surfaces of zero flux that define atomic basins of charge. The most modern way of calculating the Bader charges is using the **bader** program from Graeme Henkelmen's group.<sup>42,43</sup> Let us consider a water molecule, centered in a box. The strategy is first to run the calculation, then run the bader program on the results.

---

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

---

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

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

RUN TIME: 0.73 SECONDS

CALCULATING BADER CHARGE DISTRIBUTION

0 10 25 50 75 100

PERCENT DONE: \*\*\*\*\*

REFINING AUTOMATICALLY

ITERATION: 1

EDGE POINTS: 1400048

REASSIGNED POINTS: 111249

ITERATION: 2

CHECKED POINTS: 913184

REASSIGNED POINTS: 1424

ITERATION: 3

CHECKED POINTS: 15784

REASSIGNED POINTS: 362

ITERATION: 4

```

CHECKED POINTS:      3190
REASSIGNED POINTS:   176
ITERATION: 5
CHECKED POINTS:      1228
REASSIGNED POINTS:    99
ITERATION: 6
CHECKED POINTS:      688
REASSIGNED POINTS:    21
ITERATION: 7
CHECKED POINTS:      267
REASSIGNED POINTS:     1
ITERATION: 8
CHECKED POINTS:      26
REASSIGNED POINTS:     0

```

RUN TIME: 27.04 SECONDS

CALCULATING MINIMUM DISTANCES TO ATOMS

0 10 25 50 75 100

PERCENT DONE: \*\*\*\*\*

RUN TIME: 1.30 SECONDS

WRITING BADER ATOMIC CHARGES TO ACF.dat

WRITING BADER VOLUME CHARGES TO BCF.dat

NUMBER OF BADER MAXIMA FOUND: 5407

SIGNIFICANT MAXIMA FOUND: 10

NUMBER OF ELECTRONS: 7.99999

WRITING ATOMIC VOLUMES

0 10 25 50 75 100

PERCENT DONE: \*\*\*\*\*

RUN TIME: 4.06 SECONDS

WRITING BADER ATOMIC INDEX TO AtIndex.dat

1 cat ACF.dat

#	X	Y	Z	CHARGE	MIN DIST	ATOMIC VOL
1	6.0000	7.5265	6.0000	0.0007	1.6422	759.8654
2	6.0000	6.0000	6.0000	0.0003	1.6422	640.1559
3	6.0000	6.7632	6.5963	7.9990	2.2193	644.5831

```
VACUUM CHARGE:          0.0000
VACUUM VOLUME:          0.0000
NUMBER OF ELECTRONS:    8.0000
```

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

---

```
1 from jasp import *
2 from ase.io.bader import attach_charges
3 from ase.units import Bohr
4
5 with jasp('molecules/h2o-bader') as calc:
6     atoms = calc.get_atoms()
7
8     symbols = np.array(atoms.get_chemical_symbols())[calc.sort]
9     pos = atoms.positions[calc.sort] * Bohr
10    newatoms = Atoms(symbols, positions=pos, cell=atoms.get_cell())
11
12    attach_charges(newatoms, 'ACF.dat')
13
14    print '#+tblname: bader'
15    print '#+caption: Bader charges for a water molecule'
16    print '| atom | Bader charge|'
17    print '|-'
18    for atom in newatoms:
19        print '|{0} | {1} |'.format(atom.symbol, atom.charge)
```

---

Table 2: Bader charges for a water molecule

atom	Bader charge
H	0.9993
H	0.9997
O	0.001

TODO: figure out these charges

## 3.4 Geometry optimization

### 3.4.1 Manual determination of a bond length

The equilibrium bond length of a CO molecule is approximately the bond length that minimizes the total energy. We can find that by computing the total energy as a function of bond length, and noting where the minimum is. Here is an example in VASP. There are a few features to point out here. We want to compute 5 bond lengths, and each calculation is independent of all the others. `jasp` is set up 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 the following examples.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  import numpy as np
4  np.set_printoptions(precision=3,suppress=True)
5
6  bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
7  energies = []
8
9  for d in bond_lengths: #possible bond lengths
10
11      co = Atoms([Atom('C', [0, 0, 0]),
12                  Atom('O', [d, 0, 0])],
13                  cell=(6, 6, 6))
14
15      with jasp('molecules/co-{0}'.format(d), #output dir
16                xc='PBE',
17                nbands=6,
18                encut=350,
19                ismear=1,
20                sigma=0.01,
21                atoms=co):
22          try:
23              e = co.get_potential_energy()
24              energies.append(e)
25              print 'd = {0:1.2f} ang'.format(d)
26              print 'energy = {0:1.3f} eV'.format(e)
27              print 'forces = (eV/ang)\n {0}'.format(co.get_forces())
28              print '' #blank line
29          except (VaspSubmitted, VaspQueued):
30              energies.append(None)
31              pass
32
33  if not None in energies:
34      import matplotlib.pyplot as plt
35      plt.plot(bond_lengths, energies, 'bo-')
36      plt.xlabel('Bond length ($\AA$)')
37      plt.ylabel('Total energy (eV)')
38      plt.savefig('images/co-bondlengths.png')

```

---

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

To find the minimum we could run more calculations, but a simpler and faster way is to fit a polynomial to the data, and find the analytical minimum. The results are shown in Figure 19.

---

```

1  from jasp import *
2  import numpy as np
3  import matplotlib.pyplot as plt
4  np.set_printoptions(precision=3,suppress=True)
5
6  bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
7  energies = []
8
9  for d in bond_lengths: #possible bond lengths
10
11      with jasp('molecules/co-{0}'.format(d)) as calc:
12          atoms = calc.get_atoms()
13          energies.append(atoms.get_potential_energy())
14
15  # Now we fit an equation - cubic polynomial
16  pp = np.polyfit(bond_lengths, energies, 3)
17  dp = np.polyder(pp) # first derivative - quadratic
18
19  # we expect two roots from the quadratic eqn. These are where the
20  # first derivative is equal to zero.
21  roots = np.roots(dp)
22
23  # The minimum is where the second derivative is positive.
24  dpp = np.polyder(dp) # second derivative - line
25  secd = np.polyval(dpp, roots)
26
27  minV = roots[secd > 0]
28  minE = np.polyval(pp, minV)
29
30  print 'The minimum energy is {0} eV at V = {1} Ang^3'.format(minE,minV)
31
32  # plot the fit
33  x = np.linspace(1.05, 1.25)
34  fit = np.polyval(pp, x)
35
36  plt.plot(bond_lengths, energies, 'bo ')
37  plt.plot(x,fit, 'r-')
38  plt.plot(minV, minE, 'm* ')
39  plt.legend(['DFT','fit','minimum'], numpoints=1)
40  plt.xlabel('Bond length ($AA$)')
41  plt.ylabel('Total energy (eV)')
42  plt.savefig('images/co-bondlengths.png')
43  plt.show()

```

---

The minimum energy is  $[-14.843]$  eV at  $V = [1.144]$  Ang<sup>3</sup>

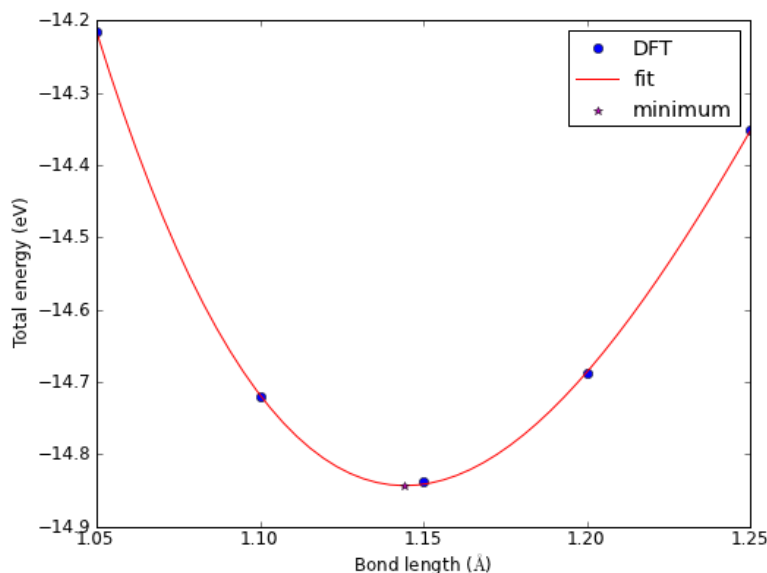


Figure 19: Energy vs CO bond length.

### 3.4.2 Automatic geometry optimization with VASP

It is generally the case that the equilibrium geometry of a system is the one that minimizes the total energy and forces. Since each atom has three degrees of freedom, you can quickly get a high dimensional optimization problem. Luckily, 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.

Here are the most common choices for IBRION.

IBRION value	algorithm
1	quasi-Newton (use if initial guess is good)
2	conjugate gradient

#### Note:

VASP applies a criteria for stopping a geometry optimization. When the change in energy between two steps is less than 0.001 eV (or  $10 \times \text{EDIFF}$ ), the relaxation is stopped. This criteria is controlled by the [EDIFFG](#) tag. If you prefer to stop based on forces, set  $\text{EDIFFG} = -0.05$ , i.e. to a negative number. The units of force is eV/Å. For most work, a force tolerance of 0.05 eV/Å is usually sufficient.

```

1 from ase import Atom, Atoms
2 from jasp import *
3 import numpy as np
4

```

```

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

```

---

```

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

```

### 3.4.3 Relaxation of a water molecule

It is not more complicated to relax more atoms, it just may take longer because there are more electrons and degrees of freedom. Here we relax a water molecule which has three atoms.

```

1  from ase import Atoms, Atom
2  from jasp import *
3
4  atoms = Atoms([Atom('H', [0.5960812, -0.7677068, 0.0000000]),
5                  Atom('O', [0.0000000, 0.0000000, 0.0000000]),
6                  Atom('H', [0.5960812, 0.7677068, 0.0000000])],
7                  cell=(8, 8, 8))
8
9  with jasp('molecules/h2o_relax',
10           xc='PBE',
11           encut=400,
12           ismear=0, # Gaussian smearing
13           ibrion=2,
14           ediff=1e-8,
15           nsw=10,
16           atoms=atoms) as calc:
17
18     print "forces"
19     print '======'
20     print atoms.get_forces()

```

---

```

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

```



## 3.5 Vibrational frequencies

### 3.5.1 Manual calculation of vibrational frequency

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

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

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

---

```
1  from jasp import *
2  from ase.units import *
3
4  bond_lengths = [1.05, 1.1, 1.15, 1.2, 1.25]
5  energies = []
6
7  for d in bond_lengths:
8      with jasp('molecules/co-{}'.format(d)) as calc:
9          atoms = calc.get_atoms()
10         energies.append(atoms.get_potential_energy())
11
12  # fit the data
13  pars = np.polyfit(bond_lengths, energies, 3)
14  xfit = np.linspace(1.05, 1.25)
15  efit = np.polyval(pars, xfit)
16
17  # first derivative
18  dpars = np.polyder(pars)
19  # find where the minimum is. chose the second one because it is the
20  # minimum we need.
21  print 'roots of first derivative are {}'.format(np.roots(dpars))
22
23  d_min = 1.14425395 # we manually copy this from the output to here
24  # second derivative
25  ddpars = np.polyder(dpars)
26
27  #curvature at minimum = force constant
28  k = np.polyval(ddpars, d_min) / kg * s**2
29
30  # reduced mass
31  from ase.data import atomic_masses
32  C_mass = atomic_masses[6] / kg
33  O_mass = atomic_masses[8] / kg
34
35  mu = (C_mass*O_mass)/(C_mass + O_mass)
36
37  frequency = 1. / (2. * np.pi) * np.sqrt(k / mu)
38  print 'The CO vibrational frequency is {} cm^-1'.format(frequency/3e10)
39
40  import matplotlib.pyplot as plt
```

---

```

41 plt.plot(bond_lengths, energies, 'bo ')
42 plt.plot(xfit, efit, 'b-')
43 plt.xlabel('Bond length (Å)')
44 plt.ylabel('Total energy (eV)')
45 plt.show()

```

---

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 described above is basically how many vibrational calculations are performed. With more atoms, you have to determine a force constant matrix and diagonalize it. For more details, see. <sup>44</sup> In practice, we usually allow a packaged code to automate this, which we cover in [Automated vibrational calculations](#)

We now consider how much energy is in this vibration. This is commonly called zero-point energy (ZPE) and it is defined as  $E_{ZPE} = \frac{1}{2}h\nu$  for a single mode, and  $h$  is Planck's constant (4.135667516e-15 eV/s).

---

```

1 c = 3e10 # speed of light cm/s
2 h = 4.135667516e-15 # eV/s
3
4 nu = 2143.6076625*c # 1/s
5
6 E_zpe = 0.5*h*nu
7
8 print 'E_ZPE = {0:1.3f} eV'.format(E_zpe)

```

---

E\_ZPE = 0.133 eV

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

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

### 3.5.2 Automated vibrational calculations

VASP has built-in capability for performing vibrational calculations. We access the capability by using a new value for [IBRION](#). The values of 5 and 6 calculated the Hessian matrix using finite differences. For IBRION=5, all atoms that are not constrained are displaced. For IBRION=6, only symmetry inequivalent displacements are considered, which makes the calculations slightly cheaper. You can specify the number of displacements with [NFREE](#). The default number of displacements is 2. You can also specify the size of the displacement with [POTIM](#) (the default is 0.015 Å).

---

```

1 #adapted from http://cms.mpi.univie.ac.at/wiki/index.php/H2O_vibration
2 from ase import Atoms, Atom
3 from jasp import *
4 import ase.units
5
6 atoms = Atoms([Atom('H', [0.5960812, -0.7677068, 0.0000000]),
7                 Atom('O', [0.0000000, 0.0000000, 0.0000000]),
8                 Atom('H', [0.5960812, 0.7677068, 0.0000000])],

```

```

9         cell=(8, 8, 8))
10 atoms.center()
11
12 with jasp('molecules/h2o_vib',
13         xc='PBE',
14         encut=400,
15         ismear=0, # Gaussian smearing
16         ibrion=6, # finite differences with symmetry
17         nfree=2, # central differences (default)
18         potim=0.015, # default as well
19         ediff=1e-8, # for vibrations you need precise energies
20         nsw=1,
21         atoms=atoms) as calc:
22
23     print 'Forces'
24     print '====='
25     print atoms.get_forces()
26     print
27     # vibrational energies are in eV
28     energies, modes = calc.get_vibrational_modes()
29     print 'energies\n====='
30     for i, e in enumerate(energies):
31         print '{0:02d}: {1} eV'.format(i, e)

```

---

Forces

=====

```

[[ 0.018 -0.032 -0.001]
 [-0.035  0.    0.003]
 [ 0.018  0.032 -0.001]]

```

energies

=====

```

00: 0.475824705 eV
01: 0.461734211 eV
02: 0.19618999 eV
03: 0.007093876 eV
04: 0.002459836 eV
05: (0.000291856+0j) eV
06: (0.012642767+0j) eV
07: (0.012959493+0j) eV
08: (0.015883929+0j) eV

```

Note we get 9 frequencies here. Water has 3 atoms, with three degrees of freedom each, leading to 9 possible combinations of collective motions. Three of those collective motions are translations, i.e. where all atoms move in the same direction (either  $x$ ,  $y$  or  $z$ ) and there is no change in the total energy of the molecule. Another three of those motions are rotations, which also do not change the total energy of the molecule. That leaves  $3N-6 = 3$  degrees of vibrational freedom where some or all of the bonds are stretched, resulting in a change in the total energy. The modes of water vibration are (with our calculated values in parentheses):

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

3. and a bending mode at  $1595\text{ cm}^{-1}$  (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.

Let us briefly discuss how to determine which vectors are vibrations and which are rotations or translations. One way is to visualize the modes. The vibrations are easy to spot. The rotations/translations are not always cleanly separable. This is an issue of accuracy and convergence. We usually do not worry about this because these modes are usually not important.

1. mode 0 is an asymmetric stretch
2. mode 1 is a symmetric stretch
3. mode 2 is a bending mode
4. mode 3 is a mixed translation/rotation
5. mode 4 is a rotation
6. mode 5 is a translation
7. mode 6 is a rotation
8. mode 7 is a partial translation
9. mode 8 is a rotation

---

```
1 from jasp import *
2 import numpy as np
3 with jasp('molecules/h2o_vib') as calc:
4     energies, modes = calc.get_vibrational_modes(mode=3, massweighted=True,
5                                                  show=True)
```

---

See [http://www.gaussian.com/g\\_whitepap/vib.htm](http://www.gaussian.com/g_whitepap/vib.htm) for a more quantitative discussion of these modes, identifying them, and a method to project the rotations and translations out of the Hessian matrix.

**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).

---

```
1 from jasp import *
2 import numpy as np
3 c = 3e10 # speed of light cm/s
4 h = 4.135667516e-15 # eV/s
5
6 # first, get the frequencies.
7 with jasp('molecules/h2o_vib') as calc:
```

---

```

8     freq = calc.get_vibrational_frequencies()
9
10    ZPE = 0.0
11    for f in freq:
12        if not isinstance(f, float):
13            continue #skip complex numbers
14        nu = f*c # convert to frequency
15        ZPE += 0.5*h*nu
16
17    print 'The ZPE of water is {0:1.3f} eV'.format(ZPE)
18
19    # one liner
20    ZPE = np.sum([0.5*h*f*c for f in freq if isinstance(f, float)])
21    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.

### 3.6 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 *
3
4  atoms = Atoms([Atom('H', [0.5960812, -0.7677068, 0.0000000]),
5                 Atom('O', [0.0000000, 0.0000000, 0.0000000]),
6                 Atom('H', [0.5960812, 0.7677068, 0.0000000])],
7               cell=(8, 8, 8))
8
9  with jasp('molecules/h2o_relax',
10           xc='PBE',
11           encut=400,
12           ismear=0, # Gaussian smearing
13           ibrion=2,
14           ediff=1e-8,
15           nsw=10,
16           atoms=atoms) as calc:
17      print 'Forces'
18      print '=====',
19      print atoms.get_forces()

```

---

Forces

```

=====
[[ 0.    -0.    0.   ]
 [-0.001  0.    0.   ]
 [ 0.    0.    0.   ]]

```

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

---

```

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

```

---

To analyze the results, this [shell script](#) was provided to extract the results.

---

```

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

```

---

```

37 done
38 for s in `seq 1 $EIG_NIONS` ; do
39   let sd=s-1 ; bo=`printf "%03u" $s` ; bod=`printf "%03u" $sd` ; mv temp.bo.$bod borncs.$bo
40 done
41 printf " ..done\n"
42
43 # parse deviation vectors (eig)
44 printf "..parsing eigenvectors"
45 let sad=$EIG_NIONS+1
46 for nu in `seq 1 $EIG_NVIBS` ; do
47   nuu=`printf "%03u" $nu`
48   tail -n $sad eigens.vib.$nuu | head -n $EIG_NIONS | awk '{print $4,$5,$6}' > e.vib.$nuu.allions
49   split -a 3 -d -l 1 e.vib.$nuu.allions temp.e.vib.$nuu.ion.
50   for s in `seq 1 $EIG_NIONS` ; do
51     let sd=s-1; bo=`printf "%03u" $s`; bod=`printf "%03u" $sd`; mv temp.e.vib.$nuu.ion.$bod e.vib.$nuu.ion.$bo
52   done
53 done
54 printf " ..done\n"
55
56 # parse born effective charge matrices (born)
57 printf "..parsing eff.charges"
58 for s in `seq 1 $EIG_NIONS` ; do
59   ss=`printf "%03u" $s`
60   awk '{print $2,$3,$4}' borncs.$ss | tail -3 > bornch.$ss
61 done
62 mkdir temps02 ; mv eigens.* borncs.* temps02/
63 printf " ..done\n"
64
65 # parse matrices, multiply them and collect squares (giving intensities)
66 printf "..multiplying matrices, summing "
67 for nu in `seq 1 $EIG_NVIBS` ; do
68   nuu=`printf "%03u" $nu`
69   int=0.0
70   for alpha in 1 2 3 ; do # summing over alpha coordinates
71     sumpol=0.0
72     for s in `seq 1 $EIG_NIONS` ; do # summing over atoms
73       ss=`printf "%03u" $s`
74       awk -v a="$alpha" '(NR==a){print}' bornch.$ss > z.ion.$ss.alpha.$alpha
75       # summing over beta coordinates and multiplying Z(s,alpha)*e(s) done by the following awk script
76       paste z.ion.$ss.alpha.$alpha e.vib.$nuu.ion.$ss | \
77       awk '{pol=$1*$4+$2*$5+$3*$6; print $0," ",pol}' > matr-vib-${nuu}-alpha-${alpha}-ion-${ss}
78     done
79     sumpol=`cat matr-vib-${nuu}-alpha-${alpha}-ion-* | awk '{sum+=$7} END {print sum}'`
80     int=`echo "$int+(sumpol)^2" | sed 's/[eE]/*10~/g' | bc -l`
81   done
82   freq=`awk '(NR==1){print $8}' temps02/eigens.vib.$nuu`
83   echo "$nuu $freq $int">> exact.res.txt
84   printf "."
85 done
86 printf " ..done\n"
87
88 # format results, normalize intensities
89 printf "..normalizing intensities"
90 max=`awk '(NR==1){max=$3} $3>max {max=$3} END {print max}' exact.res.txt`
91 awk -v max="$max" '{printf "%03u %6.1f %5.3f\n",$1,$2,$3/max}' exact.res.txt > results.txt
92 printf " ..done\n"
93
94 # clean up, display results
95 printf "..finalizing:\n"
96 mkdir temps03; mv bornch.* e.vib.*.allions temps03/
97 mkdir temps04; mv z.ion.* e.vib.*.ion.* temps04/
98 mkdir temps05; mv matr-* temps05/
99 mkdir results; mv *res.txt results/
100 let NMATRIX=$EIG_NVIBS**2
101 printf "%5u atoms found\n%5u vibrations found\n%5u matrices evaluated" \
102       $EIG_NIONS $EIG_NVIBS $NMATRIX > results/statistics.txt

```

```

103  # fast switch to clean up all temporary files
104  rm -r temps*
105  cat results/results.txt

```

---

```

..reading OUTCAR ..done
..splitting files ..done
..parsing eigenvectors ..done
..parsing eff.charges ..done
..multiplying matrices, summing ..... ..done
..normalizing intensities ..done
..finalizing:
001 3827.3 0.227
002 3713.0 0.006
003 1587.2 0.312
004 235.5 1.000
005 19.1 0.006
006 2.3 0.000
007 16.6 0.005
008 45.0 0.000
009 136.1 0.345

```

Note the results above include the rotational and vibrational modes (modes 4-9). The following [shell script](#) removes those, and recalculates the intensities. Note that it appears to just remove the last 6 modes and req compute the intensities. It is not obvious that will always be the right way to do it as the order of the eigenvectors is not guaranteed.

---

```

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

```

---

```

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

```

The interpretation of these results is that the mode at  $3713\text{ cm}^{-1}$  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. <sup>45</sup> For HREELS simulations see. <sup>46</sup>

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



---

```

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

```

---

```

mode Relative intensity
00      0.227
01      0.006
02      0.312
03      1.000
04      0.006
05      0.000
06      0.005
07      0.000
08      0.345

```

### 3.7 Thermochemical properties of molecules

`ase.thermochemistry` can be used to estimate thermodynamic properties of gases in the ideal gas limit. The module needs as input the geometry, the total energy, the vibrational energies, and some information about the molecular symmetry. We first consider an  $\text{N}_2$  molecule.

The symmetry numbers are determined by the molecular point group.<sup>47</sup> Here is a table of the most common ones.

Table 3: Symmetry numbers for common point groups

point group	$\sigma$	examples
$C_{\infty v}$	1	
$C_{\infty h}$	1	
$C_{2v}$	2	$\text{H}_2\text{O}$
$C_{3v}$	3	$\text{NH}_3$
$C_{4v}$	4	
$C_{6v}$	6	
$D_{\infty h}$	1	$\text{CO}_2$ , $\text{H}_2$
$D_{2h}$	2	
$D_{3h}$	6	
$D_{4h}$	10	
$D_{6h}$	12	
$T_d$	12	$\text{CH}_4$
$O_h$	24	

---

```

1 from ase.structure import molecule
2 from ase.thermochemistry import IdealGasThermo
3 from jasp import *
4

```

```

5  atoms = molecule('N2')
6  atoms.set_cell((10,10,10), scale_atoms=False)
7
8  # first we relax a molecule
9  with jasp('molecules/n2-relax',
10           xc='PBE',
11           encut=300,
12           ibrion=2,
13           nsw=5,
14           atoms=atoms) as calc:
15     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          ibrion=6,
22          nfree=2,
23          potim=0.15,
24          nsw=1,
25          atoms=atoms) as calc:
26     calc.calculate()
27     vib_freq = calc.get_vibrational_frequencies() # in cm-1
28
29     #convert wavenumbers to energy
30     h = 4.1356675e-15 # eV*s
31     c = 3.0e10 #cm/s
32     vib_energies = [h*c*nu for nu in vib_freq]
33     print 'vibrational energies\n=====',
34     for i,e in enumerate(vib_energies):
35         print '{0:02d}: {1} eV'.format(i,e)
36
37 # # now we can get some properties. Note we only need one vibrational
38 # energy since there is only one mode. This example does not work if
39 # you give all the energies because one energy is zero.
40 thermo = IdealGasThermo(vib_energies=vib_energies[0:0],
41                         electronicenergy=electronicenergy, atoms=atoms,
42                         geometry='linear', symmetrynumber=2, spin=0)
43
44 # temperature in K, pressure in Pa, G in eV
45 G = thermo.get_free_energy(temperature=298.15, pressure=101325.)

```

#### vibrational energies

=====

```

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

```

Enthalpy components at T = 298.15 K:

=====

E_elec	-16.478 eV
E_ZPE	0.000 eV
Cv_trans (0->T)	0.039 eV
Cv_rot (0->T)	0.026 eV
Cv_vib (0->T)	0.000 eV
(C_v -> C_p)	0.026 eV

```
-----
H                -16.388 eV
=====
```

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

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

Free energy components at T = 298.15 K and P = 101325.0 Pa:

```
=====
H                -16.388 eV
-T*S            -0.699 eV
-----
G                -17.087 eV
=====
```

Let us compare this to what is in the [Nist webbook](#) via the Shomate equations.

---

```
1  import numpy as np
2  A = 28.98641
3  B = 1.853978
4  C = -9.647459
5  D = 16.63537
6  E = 0.000117
7  F = -8.671914
8  G = 226.4168
9  H = 0.0
10
11  T = 298.15
12  t = T/1000.
13
14  S = A*np.log(t) + B*t + C*t**2/2 + D*t**3/3 - E/(2*t**2) + G
15  print '-T*S = {0:1.3f} eV'.format(-T*S/1000/96.4853)
```

---

```
0.00028578754105
-T*S = -0.592 eV
```

This is reasonable agreement for the entropy.

## 3.8 Molecular reaction energies

### 3.8.1 O<sub>2</sub> dissociation

The first reaction we consider is a simple dissociation of oxygen molecule into two oxygen atoms:  $\text{O}_2 \rightarrow 2\text{O}$ . The dissociation energy is pretty straightforward to define: it is the energy of the

products minus the energy of the reactant.  $D = 2 * E_O - E_{O_2}$ . It would appear that we simply calculate the energy of an oxygen atom, and the energy of an oxygen molecule and evaluate the formula. Let us do that.

### Simple estimate of $O_2$ dissociation energy

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('O',[5,5,5])),
5                cell=(10,10,10))
6
7  with jasp('molecules/O',
8            xc='PBE',
9            encut=400,
10           ismear=0,
11           atoms=atoms) as calc:
12     try:
13         E_O = atoms.get_potential_energy()
14     except (VaspSubmitted, VaspQueued):
15         E_O = None
16
17     # now relaxed O2 dimer
18     atoms = Atoms([Atom('O',[5,5,5]),
19                   Atom('O',[6.22, 5,5])),
20                   cell=(10,10,10))
21
22     with jasp('molecules/O2',
23              xc='PBE',
24              encut=400,
25              ismear=0,
26              ibrion=2,
27              nsw=10,
28              atoms=atoms) as calc:
29         try:
30             E_O2 = atoms.get_potential_energy()
31         except (VaspSubmitted, VaspQueued):
32             E_O2 = None
33
34     if None not in (E_O, E_O2):
35         print 'O2 -> 2O  D = {0:1.3f} eV'.format(2*E_O - E_O2)

```

---

O2 -> 2O D = 8.521 eV

The answer we have obtained is way too high! Experimentally the dissociation energy is about 5.2 eV (need reference), which is **very** different than what we calculated! Let us consider some factors that contribute to this error.

We implicitly neglected spin-polarization in the example above. That could be a problem, since the  $O_2$  molecule can be in one of two spin states, a singlet or a triplet, and these should have different energies. Furthermore, the oxygen atom can be a singlet or a triplet, and these would have different energies. To account for spin polarization, we have to tell VASP to use spin-polarization, and give initial guesses for the magnetic moments of the atoms. Let us try again with spin polarization.

### Estimating $O_2$ dissociation energy with spin polarization in triplet ground states

To tell VASP to use spin-polarization we use `ISPIN=2`, and we set initial guesses for magnetic

moments on the atoms with the `magmom` keyword. In a triplet state there are two electrons with spins of the same sign.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('O', [5, 5, 5], magmom=2)],
5                cell=(10, 10, 10))
6
7  with jasp('molecules/O-sp-triplet',
8            xc='PBE',
9            encut=400,
10           ismear=0,
11           ispin=2,
12           atoms=atoms) as calc:
13      try:
14          E_0 = atoms.get_potential_energy()
15      except (VaspSubmitted, VaspQueued):
16          E_0 = None
17
18  print 'Magnetic moment on 0 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20  # now relaxed O2 dimer
21  atoms = Atoms([Atom('O', [5, 5, 5], magmom=1),
22                Atom('O', [6.22, 5, 5], magmom=1)],
23                cell=(10,10,10))
24
25  with jasp('molecules/O2-sp-triplet',
26            xc='PBE',
27            encut=400,
28            ismear=0,
29            ispin=2, # turn spin-polarization on
30            ibrion=2, # make sure we relax the geometry
31            nsw=10,
32            atoms=atoms) as calc:
33      try:
34          E_02 = atoms.get_potential_energy()
35      except (VaspSubmitted, VaspQueued):
36          E_02 = None
37
38  # verify magnetic moment
39  print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
40
41  if None not in (E_0, E_02):
42      print 'O2 -> 20 D = {0:1.3f} eV'.format(2*E_0 - E_02)

```

---

```

Magnetic moment on 0 = 2.0000072 Bohr magnetons
Magnetic moment on O2 = 2.0000083 Bohr magnetons
O2 -> 20 D = 6.668 eV

```

This is much closer to accepted literature values for the DFT-GGA O<sub>2</sub> 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.

TODO: show what “hard or soft” is in the pseudopotential.

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

---

```

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

```

---

```

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

```

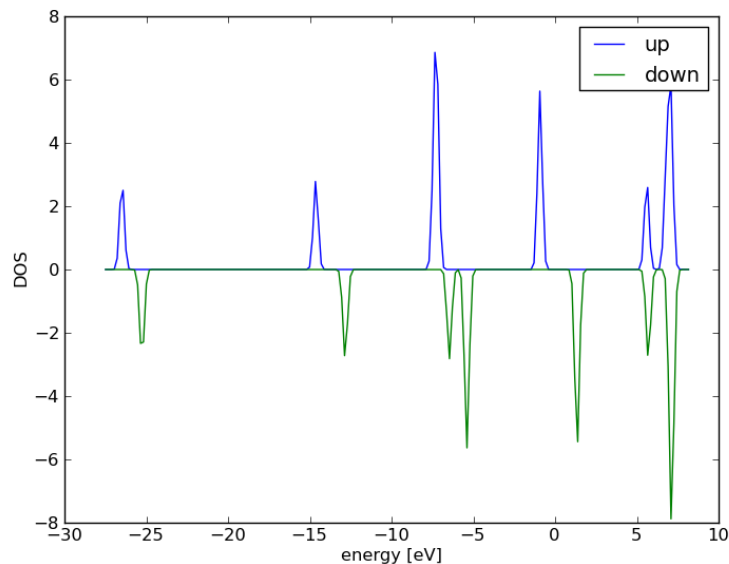


Figure 20: Spin-polarized DOS for the O<sub>2</sub> molecule.

You can see in Figure 20 that there are two different densities of states for the two spins. One has 7 electrons in it (the blue lines), and the other has 5 electrons in it (the green line).

The difference of two electrons leads to the magnetic moment of 2 which we calculated earlier. Remember that only peaks in the DOS below the Fermi level are occupied. It is customary to set the Fermi level to 0 eV in DOS plots. The peaks roughly correspond to electrons. For example, the blue peak between -25 and -30 eV corresponds to one electron, in a 1s orbital, where as the blue peak between -5 and -10 eV corresponds to three electrons.

## Convergence study of the O<sub>2</sub> dissociation energy

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  encuts = [250, 300, 350, 400, 450, 500, 550]
4
5  D = []
6  for encut in encuts:
7      atoms = Atoms([Atom('O', [5, 5, 5], magmom=2)],
8                    cell=(10,10,10))
9
10     with jasp('molecules/O-sp-triplet-{0}'.format(encut),
11              xc='PBE',
12              encut=encut,
13              ismear=0,
14              ispin=2,
15              atoms=atoms) as calc:
16         try:
17             E_0 = atoms.get_potential_energy()
18         except (VaspSubmitted, VaspQueued):
19             E_0 = None
20
21     # now relaxed O2 dimer
22     atoms = Atoms([Atom('O', [5, 5, 5], magmom=1),
23                   Atom('O', [6.22, 5, 5], magmom=1)],
24                   cell=(10,10,10))
25
26     with jasp('molecules/O2-sp-triplet-{0}'.format(encut),
27              xc='PBE',
28              encut=encut,
29              ismear=0,
30              ispin=2, # turn spin-polarization on
31              ibrion=2, # this turns relaxation on
32              nsw=10,
33              atoms=atoms) as calc:
34         try:
35             E_02 = atoms.get_potential_energy()
36         except (VaspSubmitted, VaspQueued):
37             E_02 = None
38
39     if None not in (E_0, E_02):
40         d = 2*E_0 - E_02
41         D.append(d)
42         print 'O2 -> 2O encut = {0} D = {1:1.3f} eV'.format(encut, d)
43
44 import matplotlib.pyplot as plt
45 plt.plot(encuts, D)
46 plt.xlabel('ENCUT (eV)')
47 plt.ylabel('O2 -> 2O dissociation energy (eV)')
48 plt.savefig('images/O2-dissociation-convergence.png')

```

---

```

O2 -> 2O encut = 250 D = 6.696 eV
O2 -> 2O encut = 300 D = 6.727 eV
O2 -> 2O encut = 350 D = 6.708 eV

```

```

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

```

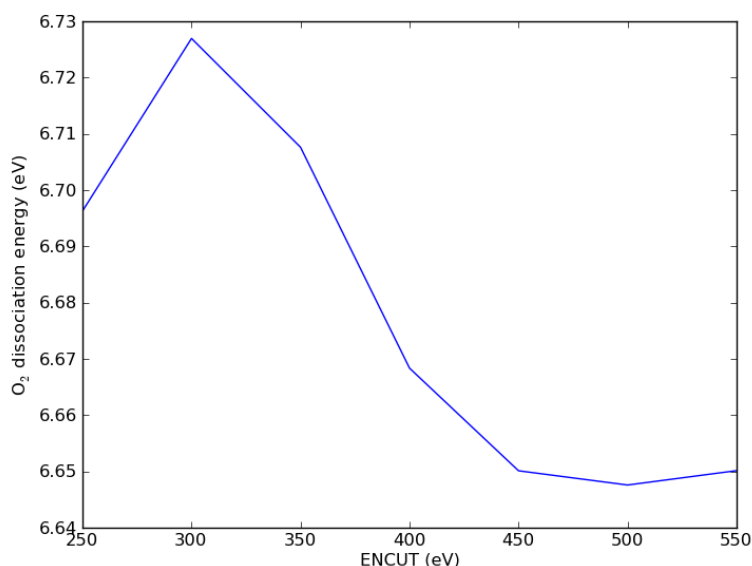


Figure 21: Convergence study of the O<sub>2</sub> dissociation energy as a function of ENCUT.

Based on these results (Figure 21), 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]
2 print 'encut (eV)          Total CPU time'
3 print '-----'
4
5 for encut in encuts:
6     OUTCAR = 'molecules/O2-sp-triplet-{0}/OUTCAR'.format(encut)
7     f = open(OUTCAR, 'r')
8     for line in f:
9         if 'Total CPU time used (sec)' in line:
10             print '{0} eV: {1}'.format(encut, line)
11 f.close()

```

---

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

```
300:                               Total CPU time used (sec):          121.296
```



350:	Total CPU time used (sec):	150.808
400:	Total CPU time used (sec):	166.956
450:	Total CPU time used (sec):	199.381
500:	Total CPU time used (sec):	240.705
550:	Total CPU time used (sec):	266.318

**Illustration of the effect of SIGMA** The methodology for extrapolation of the total energy to absolute zero is only valid for a continuous density of states at the Fermi level.<sup>12</sup> Consequently, it should not be used for semiconductors, molecules or atoms. In VASP, this means a very small Fermi temperature (SIGMA) should be used. The O<sub>2</sub> dissociation energy as a function of SIGMA is shown in Figure 22. A variation of nearly 0.2 eV is seen from the default Fermi temperature of  $k_bT = 0.2$  eV and the value of  $k_bT = 0.0001$  eV. However, virtually no change was observed for a hydrogen atom or molecule or for an oxygen molecule as a function of the Fermi temperature. It is recommended that the total energy be calculated at several values of the Fermi temperature to make sure the total energy is converged with respect to the Fermi temperature.

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 is 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 *
2  from ase import Atom, Atoms
3
4  sigmas = [0.2, 0.1, 0.05, 0.02, 0.01, 0.001]
5
6  D = []
7  for sigma in sigmas:
8      atoms = Atoms([Atom('O',[5, 5, 5], magmom=2)],
9                    cell=(10,10,10))
10
11     with jasp('molecules/O-sp-triplet-sigma-{}'.format(sigma),
12              xc='PBE',
13              encut=400,
14              ismear=0,
15              sigma=sigma,
16              ispin=2,
17              atoms=atoms) as calc:
18         try:
19             E_0 = atoms.get_potential_energy()
20         except (VaspSubmitted, VaspQueued):
21             E_0 = None
22
23     # now relaxed O2 dimer
24     atoms = Atoms([Atom('O',[5, 5, 5],magmom=1),
```

```

25         Atom('O',[6.22, 5, 5],magmom=1)],
26         cell=(10,10,10))
27
28     with jasp('molecules/O2-sp-triplet-sigma-{}'.format(sigma),
29             xc='PBE',
30             encut=400,
31             ismear=0,
32             sigma=sigma,
33             ispin=2, # turn spin-polarization on
34             ibrion=2, # make sure we relax the geometry
35             nsw=10,
36             atoms=atoms) as calc:
37         try:
38             E_O2 = atoms.get_potential_energy()
39         except (VaspSubmitted, VaspQueued):
40             E_O2 = None
41
42     if None not in (E_O, E_O2):
43         d = 2*E_O - E_O2
44         D.append(d)
45         print 'O2 -> 2O sigma = {} D = {:.1.3f} eV'.format(sigma, d)
46
47 import matplotlib.pyplot as plt
48 plt.plot(sigmas, D, 'bo-')
49 plt.xlabel('SIGMA (eV)')
50 plt.ylabel('O2 -> 2O dissociation energy (eV)')
51 plt.savefig('images/O2-dissociation-sigma-convergence.png')

```

---

```

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

```

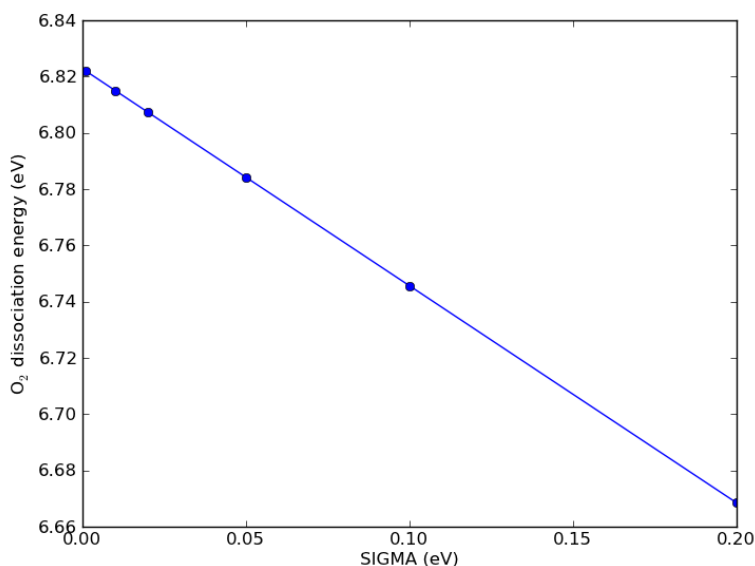


Figure 22: Effect of SIGMA on the oxygen dissociation energy.

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

**Estimating triplet oxygen dissociation energy with low symmetry** It has been suggested that breaking spherical symmetry of the atom can result in lower energy of the atom. The symmetry is broken by putting the atom off-center in a box. We will examine the total energy of an oxygen atom in a few geometries. First, let us consider variations of a square box.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  # square box origin
5  atoms = Atoms([Atom('O', [0, 0, 0], magmom=2)],
6                cell=(10, 10, 10))
7
8  with jasp('molecules/0-square-box-origin',
9            xc='PBE',
10           encut=400,
11           ismear=0,
12           sigma=0.01,
13           ispin=2,
14           atoms=atoms) as calc:
15     try:
16         print 'Square box (origin): E = {0} eV'.format(atoms.get_potential_energy())
17     except (VaspSubmitted, VaspQueued):
18         pass
19
20 # square box center
21 atoms = Atoms([Atom('O', [5, 5, 5], magmom=2)],
22               cell=(10, 10, 10))
23
24 with jasp('molecules/0-square-box-center',
25           xc='PBE',

```

```

26         encut=400,
27         ismear=0,
28         sigma=0.01,
29         ispin=2,
30         atoms=atoms) as calc:
31     try:
32         print 'Square box (center): E = {0} eV'.format(atoms.get_potential_energy())
33     except (VaspSubmitted, VaspQueued):
34         pass
35
36 # square box random
37 atoms = Atoms([Atom('O', [2.13, 7.32, 1.11], magmom=2)],
38               cell=(10, 10, 10))
39
40 with jasp('molecules/0-square-box-random',
41          xc='PBE',
42          encut=400,
43          ismear=0,
44          sigma=0.01,
45          ispin=2,
46          atoms=atoms) as calc:
47     try:
48         print 'Square box (random): E = {0} eV'.format(atoms.get_potential_energy())
49     except (VaspSubmitted, VaspQueued):
50         pass

```

---

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

There is no significant difference in these energies. The origin and center calculations are identical in energy. The meV variation in the random calculation is negligible. Now, let us consider some non-square boxes.

---

```

1  # calculate O atom energy in orthorhombic boxes
2  from jasp import *
3  from ase import Atom, Atoms
4
5  # orthorhombic box origin
6  atoms = Atoms([Atom('O', [0, 0, 0], magmom=2)],
7                cell=(8, 9, 10))
8
9  with jasp('molecules/0-orthorhombic-box-origin',
10          xc='PBE',
11          encut=400,
12          ismear=0,
13          sigma=0.01,
14          ispin=2,
15          atoms=atoms) as calc:
16     try:
17         print 'Orthorhombic box (origin): E = {0} eV'.format(atoms.get_potential_energy())
18     except (VaspSubmitted, VaspQueued):
19         pass
20
21 # orthorhombic box center
22 atoms = Atoms([Atom('O', [4, 4.5, 5], magmom=2)],
23               cell=(8, 9, 10))
24
25 with jasp('molecules/0-orthorhombic-box-center',
26          xc='PBE',
27          encut=400,
28          ismear=0,

```

```

29         sigma=0.01,
30         ispin=2,
31         atoms=atoms) as calc:
32     try:
33         print 'Orthorhombic box (center): E = {0} eV'.format(atoms.get_potential_energy())
34     except (VaspSubmitted, VaspQueued):
35         pass
36
37     # orthorhombic box random
38     atoms = Atoms([Atom('O', [2.13, 7.32, 1.11], magmom=2)],
39                   cell=(8, 9, 10))
40
41     with jasp('molecules/0-orthorhombic-box-random',
42             xc='PBE',
43             encut=400,
44             ismear=0,
45             sigma=0.01,
46             ispin=2,
47             atoms=atoms) as calc:
48     try:
49         print 'Orthorhombic box (random): E = {0} eV'.format(atoms.get_potential_energy())
50     except (VaspSubmitted, VaspQueued):
51         pass

```

---

```

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

```

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

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('O', [5.1, 4.2, 6.1], magmom=2)],
5                cell=(8, 9, 10))
6
7  with jasp('molecules/0-sp-triplet-lowsym',
8          xc='PBE',
9          encut=400,
10         ismear=0,
11         sigma=0.01,
12         ispin=2,
13         atoms=atoms) as calc:
14     try:
15         E_0 = atoms.get_potential_energy()
16         print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
17     except (VaspSubmitted, VaspQueued):
18         E_0 = None
19
20     # now relaxed O2 dimer
21     atoms = Atoms([Atom('O', [5, 5, 5], magmom=1),
22                   Atom('O', [6.22, 5, 5], magmom=1)],
23                   cell=(10, 10, 10))
24
25     with jasp('molecules/O2-sp-triplet',
26             xc='PBE',
27             encut=400,
28             ismear=0,

```

```

29         sigma=0.01,
30         ispin=2, # turn spin-polarization on
31         ibrion=2, # make sure we relax the geometry
32         nsw=10,
33         atoms=atoms) as calc:
34     try:
35         E_O2 = atoms.get_potential_energy()
36         # verify magnetic moment
37         print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
38
39     except (VaspSubmitted, VaspQueued):
40         E_O2 = None
41
42 if None not in (E_O, E_O2):
43     print 'E_O: ', E_O
44     print 'O2 -> 2O   D = {0:1.3f} eV'.format(2*E_O - E_O2)

```

---

```

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

```

This actually agrees within 30-50 meV of reported literature values, although still nearly an eV greater than the experimental dissociation energy. 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 do not need the dissociation energy 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.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('O', [5, 5, 5], magmom=0)],
5                cell=(10, 10, 10))
6
7  with jasp('molecules/O-sp-singlet',
8            xc='PBE',
9            encut=400,
10           ismear=0,
11           ispin=2,
12           atoms=atoms) as calc:
13     try:
14         E_O = atoms.get_potential_energy()
15     except (VaspSubmitted, VaspQueued):
16         E_O = None
17
18     print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
19
20 # now relaxed O2 dimer
21 atoms = Atoms([Atom('O', [5, 5, 5], magmom=1),
22               Atom('O', [6.22, 5, 5], magmom=-1)],
23               cell=(10, 10, 10))
24
25 with jasp('molecules/O2-sp-singlet',
26           xc='PBE',

```

```

27         encut=400,
28         ismear=0,
29         ispin=2, # turn spin-polarization on
30         ibrion=2, # make sure we relax the geometry
31         nsw=10,
32         atoms=atoms) as calc:
33     try:
34         E_02 = atoms.get_potential_energy()
35     except (VaspSubmitted, VaspQueued):
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):
42     print '02 -> 20   D = {0:1.3f} eV'.format(2*E_0 - E_02)

```

---

```

Magnetic moment on 0 = 1.9998232 Bohr magnetons
0.0
02 -> 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:

```

1 from jasp import *
2
3 with jasp('molecules/O2-sp-singlet') as calc:
4     print 'singlet: {0} eV'.format(calc.get_atoms().get_potential_energy())
5
6 with jasp('molecules/O2-sp-triplet') as calc:
7     print 'triplet: {0} eV'.format(calc.get_atoms().get_potential_energy())

```

---

```

singlet:  -8.830101
triplet:  -9.848238

```

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

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

```

1 from jasp import *
2 from ase import Atom, Atoms
3
4 with jasp('molecules/O2-sp-singlet') as calc:
5     calc.clone('molecules/O2-sp-singlet-magmoms')
6
7 with jasp('molecules/O2-sp-singlet-magmoms') as calc:

```

```

8     calc.set(lorbit=11)
9     atoms = calc.get_atoms()
10    magmoms = atoms.get_magnetic_moments()
11
12    print 'singlet ground state'
13    for i,atom in enumerate(atoms):
14        print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
15    print atoms.get_magnetic_moment()
16
17    with jasp('molecules/O2-sp-triplet') as calc:
18        calc.clone('molecules/O2-sp-triplet-magmoms')
19
20    with jasp('molecules/O2-sp-triplet-magmoms') as calc:
21        calc.set(lorbit=11)
22        atoms = calc.get_atoms()
23        magmoms = atoms.get_magnetic_moments()
24        print
25        print 'triplet ground state'
26        for i,atom in enumerate(atoms):
27            print 'atom {0}: magmom = {1}'.format(i, magmoms[i])
28        print atoms.get_magnetic_moment()

```

---

```

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

```

```

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

```

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

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

---

```

1  grep ENMIN $VASP_PP_PATH/potpaw_PBE/O_sv/POTCAR

```

---

```

ENMAX  = 1421.493; ENMIN  = 1066.119 eV

```

In the following calculation, we let VASP select an appropriate ENCUT value.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('O', [4, 4.5, 5], magmom=2)],
5               cell=(8, 9, 10))
6
7  with jasp('molecules/O-sp-triplet-lowsym-sv',

```



```

8         xc='PBE',
9         ismear=0,
10        ispin=2,
11        sigma=0.01,
12        setups={'O': '_sv'},
13        atoms=atoms) as calc:
14    try:
15        E_0 = atoms.get_potential_energy()
16    except (VaspSubmitted, VaspQueued):
17        E_0 = None
18
19    print 'Magnetic moment on O = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
20
21    # now relaxed O2 dimer
22    atoms = Atoms([Atom('O', [5, 5, 5], magmom=1),
23                  Atom('O', [6.22, 5, 5], magmom=1)],
24                  cell=(10, 10, 10))
25
26    with jasp('molecules/O2-sp-triplet-sv',
27             xc='PBE',
28             ismear=0,
29             sigma=0.01,
30             ispin=2, # turn spin-polarization on
31             ibrion=2, # make sure we relax the geometry
32             nsw=10,
33             setups={'O': '_sv'},
34             atoms=atoms) as calc:
35        try:
36            E_O2 = atoms.get_potential_energy()
37        except (VaspSubmitted, VaspQueued):
38            E_O2 = None
39
40    # verify magnetic moment
41    print 'Magnetic moment on O2 = {0} Bohr magnetons'.format(atoms.get_magnetic_moment())
42
43    if None not in (E_0, E_O2):
44        print 'O2 -> 2O   D = {0:1.3f} eV'.format(2*E_0 - E_O2)

```

---

```

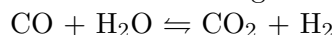
Magnetic moment on O = 2.0000001 Bohr magnetons
Magnetic moment on O2 = 1.9999996 Bohr magnetons
O2 -> 2O   D = 6.299 eV

```

This result is close to other reported values. It is possibly not converged, since we let VASP choose the ENCUT value, and that value is the ENMIN value in the POTCAR. Nevertheless, the point is that a harder potential does not fix the problem of overbinding in the O<sub>2</sub> molecule. That is a fundamental flaw in the GGA exchange-correlation functional.

### 3.8.2 Water gas shift example

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



We define the reaction energy as the difference in energy between the products and reactants.

$$\Delta E = E_{\text{CO}_2} + E_{\text{H}_2} - E_{\text{CO}} - E_{\text{H}_2\text{O}}$$

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

---

```

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

```

```

66
67 if None in (eC02, eH2, eC0, eH2O):
68     pass
69 else:
70     dE = eC02 + eH2 - eC0 - eH2O
71     print 'Delta E = {0:1.3f} eV'.format(dE)
72     print 'Delta E = {0:1.3f} kcal/mol'.format(dE*23.06035)
73     print 'Delta E = {0:1.3f} kJ/mol'.format(dE*96.485)

```

---

```

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

```

We [estimated](#) the enthalpy of this reaction at standard conditions to be -41 kJ/mol using data from the NIST webbook, which is a fair bit lower than we calculated here. 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, i.e., the forces on each atom are less than 0.05 eV/Å.

---

```

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

```

---

## Calculation summaries

### CO

```

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

```

```

Unit cell vectors (angstroms)
      x      y      z      length
a0 [ 8.000  0.000  0.000] 8.000
a1 [ 0.000  8.000  0.000] 8.000
a2 [ 0.000  0.000  8.000] 8.000
a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0
Unit cell volume = 512.000 Ang^3
Stress (GPa):xx,   yy,   zz,   yz,   xz,   xy
              0.005  0.005  0.005 -0.000 -0.000 -0.000
Atom#  sym      position [x,y,z]      tag  rmsForce constraints
   0    O  [0.000      0.000      0.490]    0   0.01      T T T
   1    C  [0.000      0.000      7.346]    0   0.01      T T T
-----

```

#### INCAR Parameters:

```

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

```

#### Pseudopotentials used:

```

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

```

#### CO<sub>2</sub>

```

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

```

```

Unit cell vectors (angstroms)
      x      y      z      length
a0 [ 8.000  0.000  0.000] 8.000
a1 [ 0.000  8.000  0.000] 8.000
a2 [ 0.000  0.000  8.000] 8.000
a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0

```

```

Unit cell volume = 512.000 Ang^3
Stress (GPa):xx,   yy,   zz,   yz,   xz,   xy
                0.009  0.009  0.008 -0.000 -0.000 -0.000
Atom#  sym      position [x,y,z]      tag  rmsForce constraints
  0    C  [0.000    0.000    0.000]    0   0.00      T T T
  1    O  [0.000    0.000    1.177]    0   0.01      T T T
  2    O  [0.000    0.000    6.823]    0   0.01      T T T
-----

```

#### INCAR Parameters:

```

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

```

#### Pseudopotentials used:

```

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

```

#### H<sub>2</sub>

```

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

```

```

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

```

-----

INCAR Parameters:

-----

```

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

```

Pseudopotentials used:

-----

H: potpaw\_PBE/H/POTCAR (git-hash: fbc0773b08b32f553234b0b50cc6ad6f5085c816)

**H<sub>2</sub>O**

: -----

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

Unit cell vectors (angstroms)

	x	y	z	length
a0	[ 8.000	0.000	0.000]	8.000
a1	[ 0.000	8.000	0.000]	8.000
a2	[ 0.000	0.000	8.000]	8.000

a,b,c,alpha,beta,gamma (deg): 8.000 8.000 8.000 90.0 90.0 90.0

Unit cell volume = 512.000 Ang<sup>3</sup>

Stress (GPa):xx, yy, zz, yz, xz, xy  
0.005 0.004 0.005 -0.000 -0.000 -0.000

Atom#	sym	position [x,y,z]	tag	rmsForce	constraints
0	O	[0.000 0.000 0.122]	0	0.06	T T T
1	H	[0.000 0.766 7.522]	0	0.04	T T T
2	H	[0.000 7.234 7.522]	0	0.04	T T T

-----

INCAR Parameters:

-----

```

nbands: 8
ismear: 0
nsw: 10

```

```

ibrion: 2
encut: 350.0
prec: Normal
kpts: [1 1 1]
reciprocal: False
xc: PBE
txt: -
gamma: False

```

Pseudopotentials used:

```

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

```

### 3.8.3 Temperature dependent water gas shift equilibrium constant

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

#### CO vibrations

---

```

1  from jasp import *
2
3  # get relaxed geometry
4  with jasp('molecules/wgs/CO') as calc:
5      CO = calc.get_atoms()
6
7  # now do the vibrations
8  with jasp('molecules/wgs/CO-vib',
9          xc='PBE',
10         encut=350,
11         ismear=0,
12         ibrion=6,
13         nfree=2,
14         potim=0.02,
15         nsw=1,
16         atoms=CO) as calc:
17      calc.calculate()
18      vib_freq = calc.get_vibrational_frequencies()
19      for i,f in enumerate(vib_freq):
20          print '{0:02d}: {1} cm^(-1)'.format(i,f)

```

---

```

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

```

CO has only one vibrational mode ( $3N-5 = 6 - 5 = 1$ ). The other 5 modes are 3 translations and 2 rotations.

## CO<sub>2</sub> vibrations

---

```
1 from jasp import *
2
3 # get relaxed geometry
4 with jasp('molecules/wgs/CO2') as calc:
5     CO2 = calc.get_atoms()
6
7 # now do the vibrations
8 with jasp('molecules/wgs/CO2-vib',
9         xc='PBE',
10        encut=350,
11        ismear=0,
12        ibrion=6,
13        nfree=2,
14        potim=0.02,
15        nsw=1,
16        atoms=CO2) as calc:
17     calc.calculate()
18     vib_freq = calc.get_vibrational_frequencies()
19     for i,f in enumerate(vib_freq):
20         print '{0:02d}: {1} cm-1'.format(i,f)
```

---

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

CO<sub>2</sub> is a linear molecule with  $3N-5 = 4$  vibrational modes. They are the first four frequencies in the output above.

## H<sub>2</sub> vibrations

---

```
1 from jasp import *
2
3 # get relaxed geometry
4 with jasp('molecules/wgs/H2') as calc:
5     H2 = calc.get_atoms()
6
7 # now do the vibrations
8 with jasp('molecules/wgs/H2-vib',
9         xc='PBE',
10        encut=350,
11        ismear=0,
12        ibrion=6,
13        nfree=2,
14        potim=0.02,
15        nsw=1,
16        atoms=H2) as calc:
```

---



```

17     calc.calculate()
18     vib_freq = calc.get_vibrational_frequencies()
19     for i,f in enumerate(vib_freq):
20         print '{0:02d}: {1} cm(-1)'.format(i,f)

```

---

```

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

```

There is only one frequency of importance (the one at 4281 cm<sup>-1</sup>) for the linear H<sub>2</sub> molecule.

## H<sub>2</sub>O vibrations

---

```

1  from jasp import *
2
3  # get relaxed geometry
4  with jasp('molecules/wgs/H2O') as calc:
5      H2O = calc.get_atoms()
6
7  # now do the vibrations
8  with jasp('molecules/wgs/H2O-vib',
9          xc='PBE',
10         encut=350,
11         ismear=0,
12         ibrion=6,
13         nfree=2,
14         potim=0.02,
15         nsw=1,
16         atoms=H2O) as calc:
17      calc.calculate()
18      vib_freq = calc.get_vibrational_frequencies()
19      for i,f in enumerate(vib_freq):
20          print '{0:02d}: {1} cm(-1)'.format(i,f)

```

---

```

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

```

Water has  $3N-6 = 3$  vibrational modes.

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

---

```

1  from ase.thermochemistry import IdealGasThermo
2  from jasp import *
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  # first we get the electronic energies
7  with jasp('molecules/wgs/CO') as calc:
8      CO = calc.get_atoms()
9      E_CO = CO.get_potential_energy()
10
11  with jasp('molecules/wgs/CO2') as calc:
12      CO2 = calc.get_atoms()
13      E_CO2 = CO2.get_potential_energy()
14
15  with jasp('molecules/wgs/H2') as calc:
16      H2 = calc.get_atoms()
17      E_H2 = H2.get_potential_energy()
18
19  with jasp('molecules/wgs/H2O') as calc:
20      H2O = calc.get_atoms()
21      E_H2O = H2O.get_potential_energy()
22
23  # now we get the vibrational energies
24  h = 4.1356675e-15 # eV*s
25  c = 3.0e10 #cm/s
26
27  with jasp('molecules/wgs/CO-vib') as calc:
28      vib_freq = calc.get_vibrational_frequencies()
29      CO_vib_energies = [h*c*nu for nu in vib_freq]
30
31  with jasp('molecules/wgs/CO2-vib') as calc:
32      vib_freq = calc.get_vibrational_frequencies()
33      CO2_vib_energies = [h*c*nu for nu in vib_freq]
34
35  with jasp('molecules/wgs/H2-vib') as calc:
36      vib_freq = calc.get_vibrational_frequencies()
37      H2_vib_energies = [h*c*nu for nu in vib_freq]
38
39  with jasp('molecules/wgs/H2O-vib') as calc:
40      vib_freq = calc.get_vibrational_frequencies()
41      H2O_vib_energies = [h*c*nu for nu in vib_freq]
42
43  # now we make a thermo object for each molecule
44  CO_t = IdealGasThermo(vib_energies=CO_vib_energies[0:0],
45                        electronicenergy=E_CO, atoms=CO,
46                        geometry='linear', symmetrynumber=1,
47                        spin=0)
48
49  CO2_t = IdealGasThermo(vib_energies=CO2_vib_energies[0:4],
50                        electronicenergy=E_CO2, atoms=CO2,
51                        geometry='linear', symmetrynumber=2,
52                        spin=0)
53
54  H2_t = IdealGasThermo(vib_energies=H2_vib_energies[0:0],
55                        electronicenergy=E_H2, atoms=H2,
56                        geometry='linear', symmetrynumber=2,
57                        spin=0)
58
59  H2O_t = IdealGasThermo(vib_energies=H2O_vib_energies[0:3],
60                        electronicenergy=E_H2O, atoms=H2O,
61                        geometry='nonlinear', symmetrynumber=2,
62                        spin=0)
63
64  # now we can compute G_rxn for a range of temperatures from 298 to 1000 K
65  Trange = np.linspace(298,1000,20) #K

```

```

66 P = 101325. # Pa
67 Grxn = np.array([(CO2_t.get_free_energy(temperature=T, pressure=P)
68                  + H2_t.get_free_energy(temperature=T, pressure=P)
69                  - H2O_t.get_free_energy(temperature=T, pressure=P)
70                  - CO_t.get_free_energy(temperature=T, pressure=P))*96.485 for T in Trange])
71
72 Hrxn = np.array([(CO2_t.get_enthalpy(temperature=T)
73                  + H2_t.get_enthalpy(temperature=T)
74                  - H2O_t.get_enthalpy(temperature=T)
75                  - CO_t.get_enthalpy(temperature=T))*96.485 for T in Trange])
76
77 plt.plot(Trange, Grxn, 'bo-', label='$\Delta G_{rxn}$')
78 plt.plot(Trange, Hrxn, 'ro:', label='$\Delta H_{rxn}$')
79 plt.xlabel('Temperature (K)')
80 plt.ylabel('$\Delta G_{rxn}$ (kJ/mol)')
81 plt.legend(loc='best')
82 plt.savefig('images/wgs-dG-T.png')
83
84 plt.figure()
85 R = 8.314e-3 # gas constant in kJ/mol/K
86
87 Keq = np.exp(-Grxn/R/Trange)
88 plt.plot(Trange, Keq)
89 plt.ylim([0, 100])
90 plt.xlabel('Temperature (K)')
91 plt.ylabel('$K_{eq}$')
92 plt.savefig('images/wgs-Keq.png')
93 plt.show()

```

None

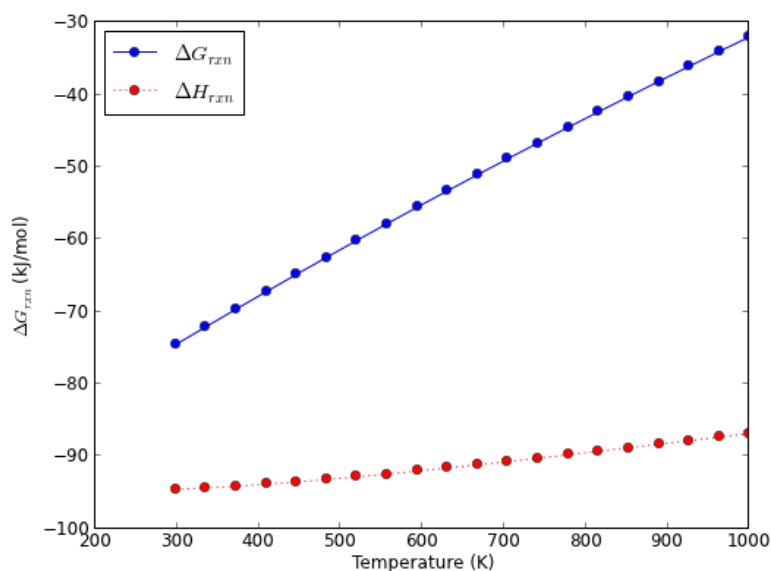


Figure 23: Thermodynamic energies of the water gas shift reaction as a function of temperature.

You can see a few things here. One is that at near 298K, the Gibbs free energy is about -75 kJ/mol. This is too negative compared to the experimental standard 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. The error in energy has a significant effect on the calculated equilibrium constant, significantly overestimating it.

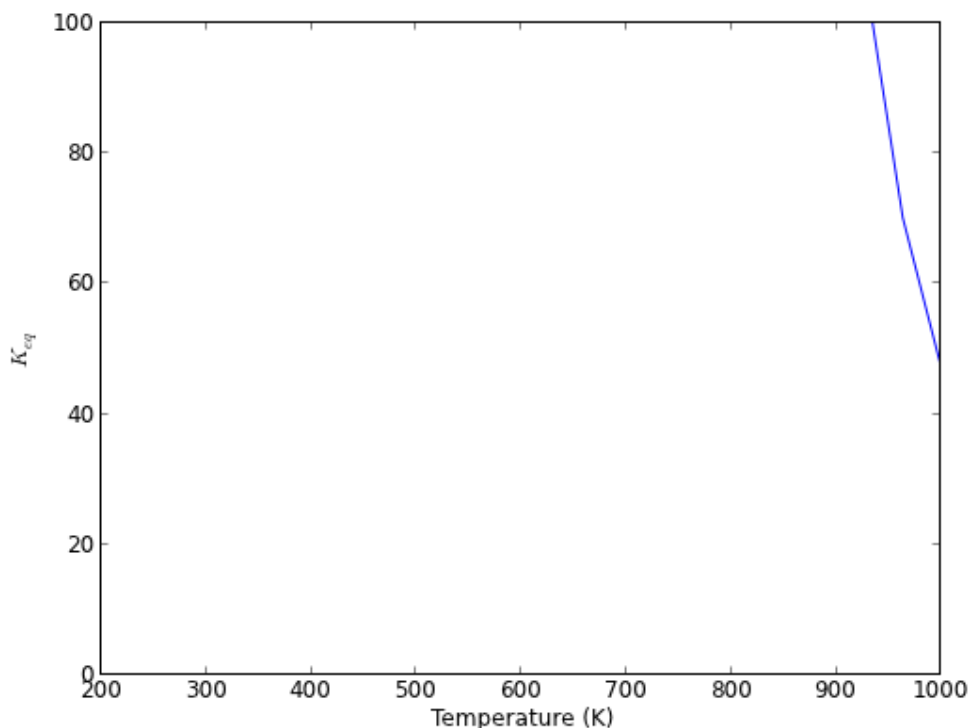


Figure 24: Temperature dependence of the equilibrium constant.

### 3.9 Molecular reaction barriers

We will consider a simple example of the barrier for  $\text{NH}_3$  inversion. We have to create an  $\text{NH}_3$  molecule in the initial and inverted state (these have exactly the same energy), and then interpolate a band of images. Then, we use the NEB method to compute the barrier to inversion.

#### 3.9.1 Get initial and final states

---

```

1  # compute initial and final states
2  from ase import Atoms
3  from ase.data.molecules import molecule
4  import numpy as np
5  from jasp import *
6  from ase.constraints import FixAtoms
7
8  atoms = molecule('NH3')
9  constraint = FixAtoms(mask=[atom.symbol == 'N' for atom in atoms])
10 atoms.set_constraint(constraint)
11
12 Npos = atoms.positions[0]
```

```

13
14 # move N to origin
15 atoms.translate(-Npos)
16 atoms.set_cell((10, 10, 10), scale_atoms=False)
17
18 atoms2 = atoms.copy()
19 pos2 = atoms2.positions
20
21 for i,atom in enumerate(atoms2):
22     if atom.symbol == 'H':
23         # reflect through z
24         pos2[i] *= np.array([1, 1, -1])
25 atoms2.positions = pos2
26
27 #now move N to center of box
28 atoms.translate([5, 5, 5])
29 atoms2.translate([5, 5, 5])
30
31 with jasp('molecules/nh3-initial',
32          xc='PBE',
33          encut=350,
34          ibrion=1,
35          nsw=10,
36          atoms=atoms) as calc:
37     try:
38         calc.calculate()
39     except (VaspSubmitted, VaspQueued):
40         pass
41
42 with jasp('molecules/nh3-final',
43          xc='PBE',
44          encut=350,
45          ibrion=1,
46          nsw=10,
47          atoms=atoms2) as calc:
48     try:
49         calc.calculate()
50     except (VaspSubmitted, VaspQueued):
51         pass

```

---

### 3.9.2 Run band calculation

Now we do the band calculation.

---

```

1 # Run NH3 NEB calculations
2 from jasp import *
3 from ase.neb import NEB
4
5 with jasp('molecules/nh3-initial') as calc:
6     atoms = calc.get_atoms()
7
8 with jasp('molecules/nh3-final') as calc:
9     atoms2 = calc.get_atoms()
10
11 # 5 images including endpoints
12 images = [atoms]
13 images += [atoms.copy() for i in range(3)]
14 images += [atoms2]
15
16 neb = NEB(images)
17 neb.interpolate()
18
19 with jasp('molecules/nh3-neb',

```

```

20         xc='PBE',
21         ibrion=1,
22         nsw=90,
23         spring=-5,
24         atoms=images) as calc:
25     images,energies = calc.get_neb()
26
27     calc.plot_neb(show=False)
28     import matplotlib.pyplot as plt
29     plt.savefig('images/nh3-neb.png')

```

---

None

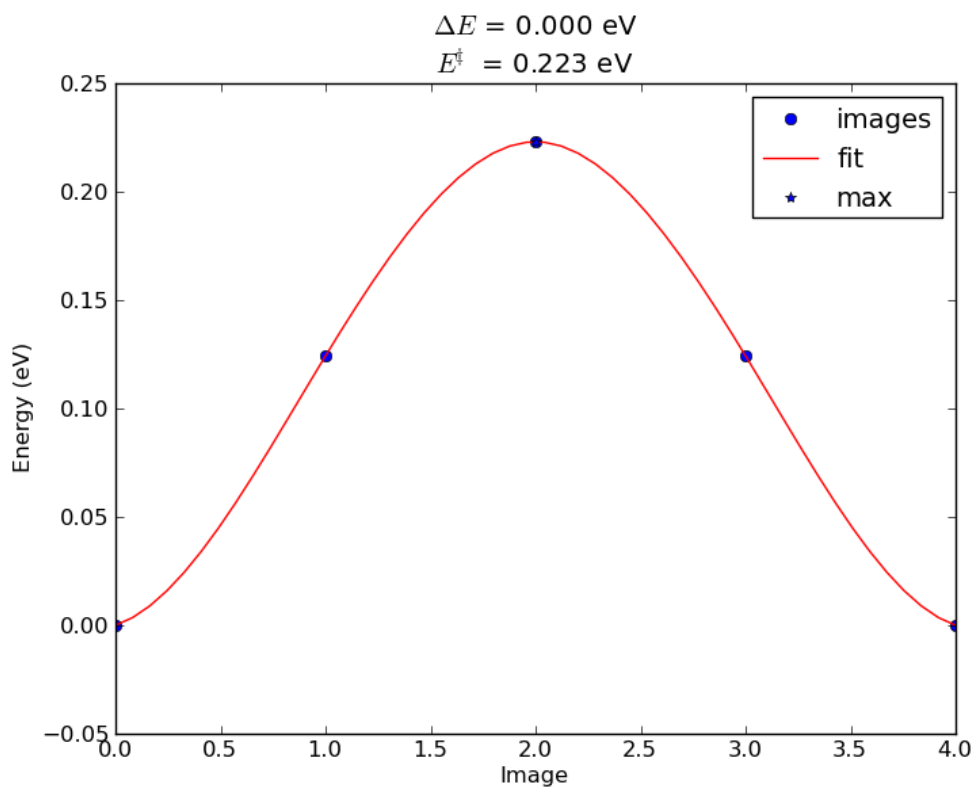


Figure 25: Nudged elastic band results for ammonia flipping.

## 4 Bulk systems

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

## 4.1 Defining and visualizing bulk systems

### 4.1.1 Built-in functions in ase

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

- `ase.lattice.cubic.FaceCenteredCubic`
- `ase.lattice.cubic.BodyCenteredCubic`
- `ase.lattice.hexagonal.Graphite`
- `ase.lattice.compounds.NaCl`

For others, see <https://wiki.fysik.dtu.dk/ase/ase/lattice.html>

We start with a simple example, fcc Ag. By default, `ase` knows Ag is an fcc metal, and knows the experimental lattice constant. We have to specify the directions (vectors along each axis) to get something other than the default output. Here, the default fcc cell contains four atoms.

---

```
1 from ase.io import write
2 from ase.lattice.cubic import FaceCenteredCubic
3
4 atoms = FaceCenteredCubic('Ag')
5
6 write('images/Ag-fcc.png', atoms, show_unit_cell=2)
7
8 print atoms
```

---

```
Lattice(symbols='Ag4', positions=..., cell=[4.09, 4.09, 4.09], pbc=[True, True, True])
```

#### Note:

A `ase.lattice.bravais.Lattice` object is returned! This is practically the same as as an `ase.atoms.Atoms` object.

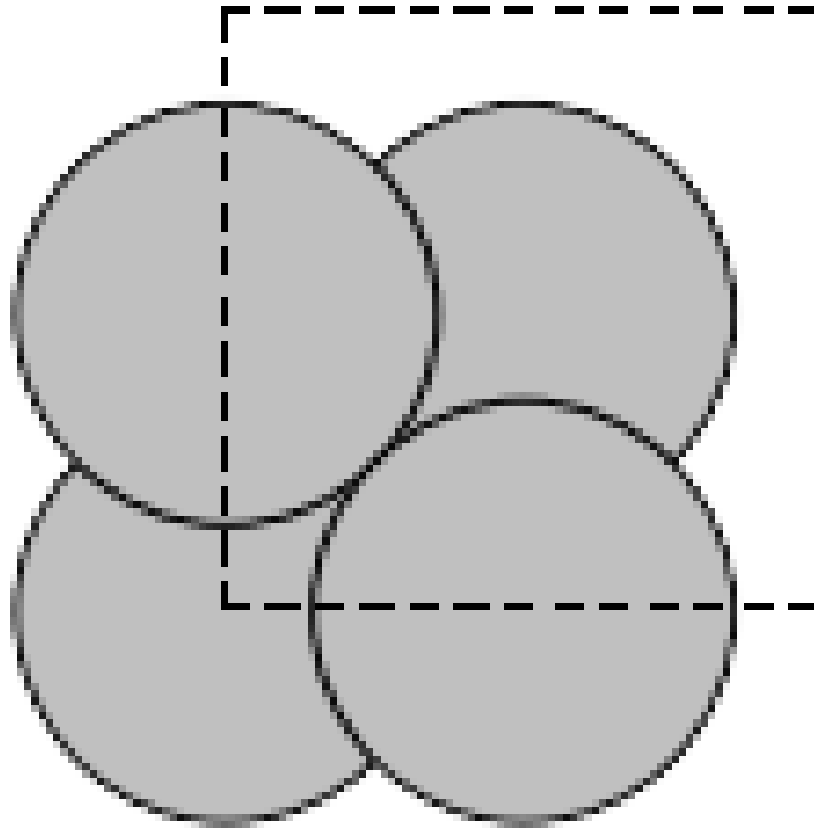


Figure 26: A simple fcc Ag bulk structure in the primitive unit cell.

Here we specify the primitive unit cell, which only has one atom in it.

---

```

1  from ase.io import write
2  from ase.lattice.cubic import FaceCenteredCubic
3
4  atoms = FaceCenteredCubic('Ag', directions=[[0, 1, 1],
5                                               [1, 0, 1],
6                                               [1, 1, 0]])
7
8  write('images/Ag-fcc-primitive.png',atoms,show_unit_cell=2)
9
10 print atoms

```

---

```
Lattice(symbols='Ag', positions=..., cell=[[2.892066735052979, 0.0, 0.0], [1.4460333675264
```



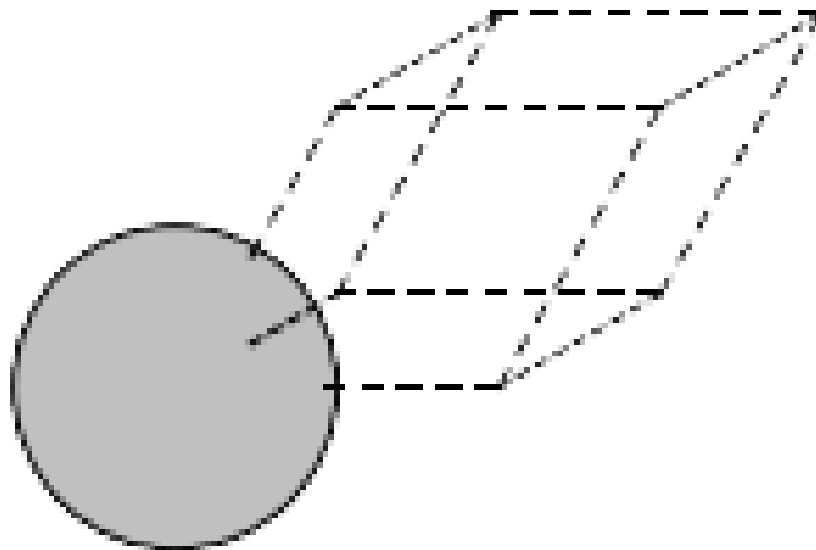


Figure 27: A simple fcc Ag bulk structure in the primitive unit cell.

We can use these modules to build alloy unit cells. The basic strategy is to create the base unit cell in one element and then selectively change some atoms to different chemical symbols. Here we examine an  $\text{Ag}_3\text{Pd}$  alloy structure.

---

```

1  from ase.io import write
2  from ase.lattice.cubic import FaceCenteredCubic
3
4  atoms = FaceCenteredCubic(directions=[[1,0,0],
5                                         [0,1,0],
6                                         [0,0,1]],
7                               size=(1,1,1),
8                               symbol='Ag',
9                               latticeconstant=4.0)
10
11 write('images/Ag-bulk.png',atoms,show_unit_cell=2)
12
13 # to make an alloy, we can replace one atom with another kind
14 atoms[0].symbol = 'Pd'
15 write('images/AgPd-bulk.png', atoms, show_unit_cell=2)

```

---

None

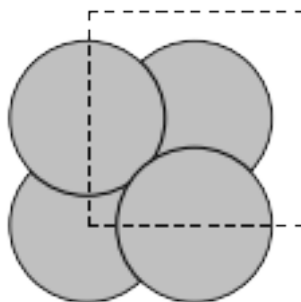


Figure 28: A simple fcc Ag bulk structure in the traditional unit cell.

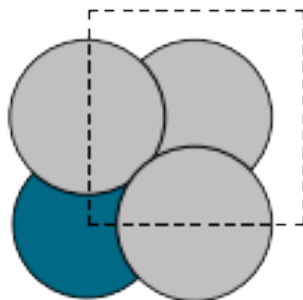


Figure 29: A simple Ag<sub>3</sub>Pd bulk structure.

To create a graphite structure we use the following code. Note that we have to specify the lattice constants (taken from <http://www.phy.ohiou.edu/~smith/NewATOMS/HOPG.pdf>) because `ase` has C in the diamond structure by default. We show two views, because the top view does not show the spacing between the layers.

---

```

1 from ase.lattice.hexagonal import Graphite
2 from ase.io import write
3
4 atoms = Graphite('C', latticeconstant={'a':2.4612, 'c':6.7079})
5 write('images/graphite.png', atoms.repeat((2,2,1)),rotation='115x', show_unit_cell=2)
6 write('images/graphite-top.png', atoms.repeat((2,2,1)), show_unit_cell=2)

```

---

None

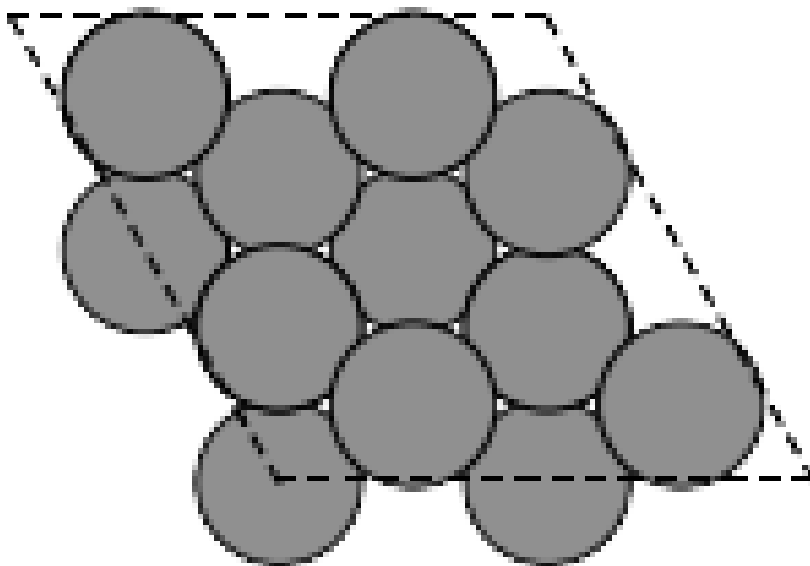


Figure 30: A top view of graphite.

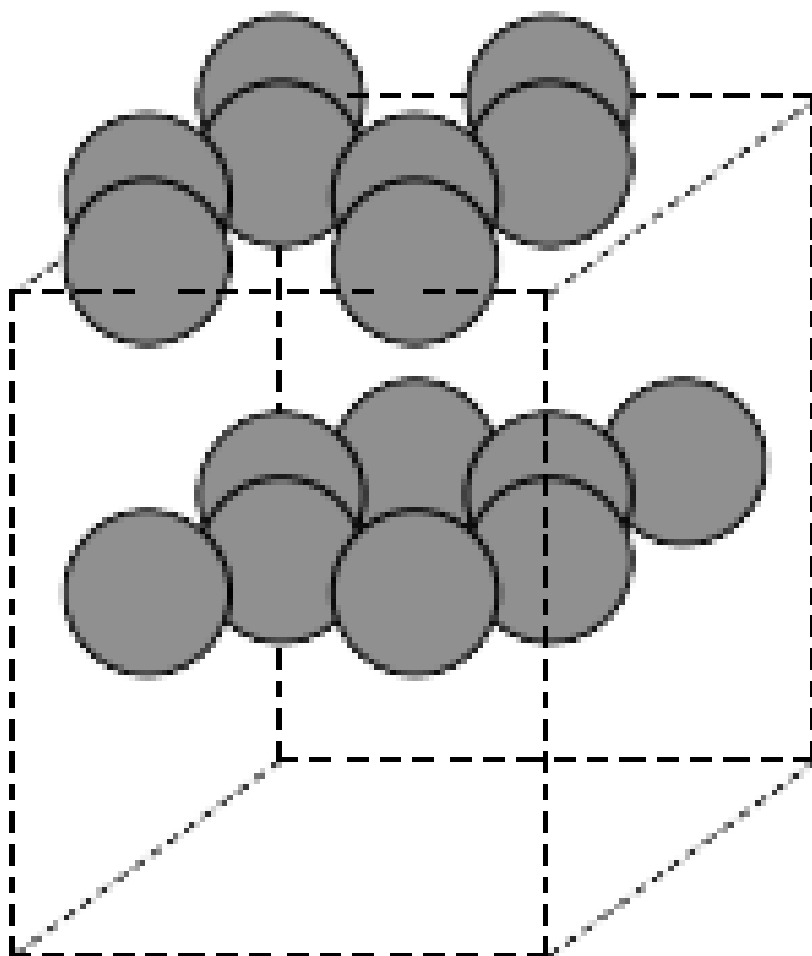


Figure 31: A side view of graphite.

To get a compound, we use the following code. We have to specify the basis atoms to the function generating the compound, and the lattice constant. For NaCl we use the lattice constant at ([http://en.wikipedia.org/wiki/Sodium\\_chloride](http://en.wikipedia.org/wiki/Sodium_chloride)).

---

```

1  from ase.lattice.compounds import NaCl
2  from ase.io import write
3
4  atoms = NaCl(['Na','Cl'], latticeconstant=5.65)
5  write('images/NaCl.png', atoms, show_unit_cell=2, rotation='45x,45y,45z')
```

---

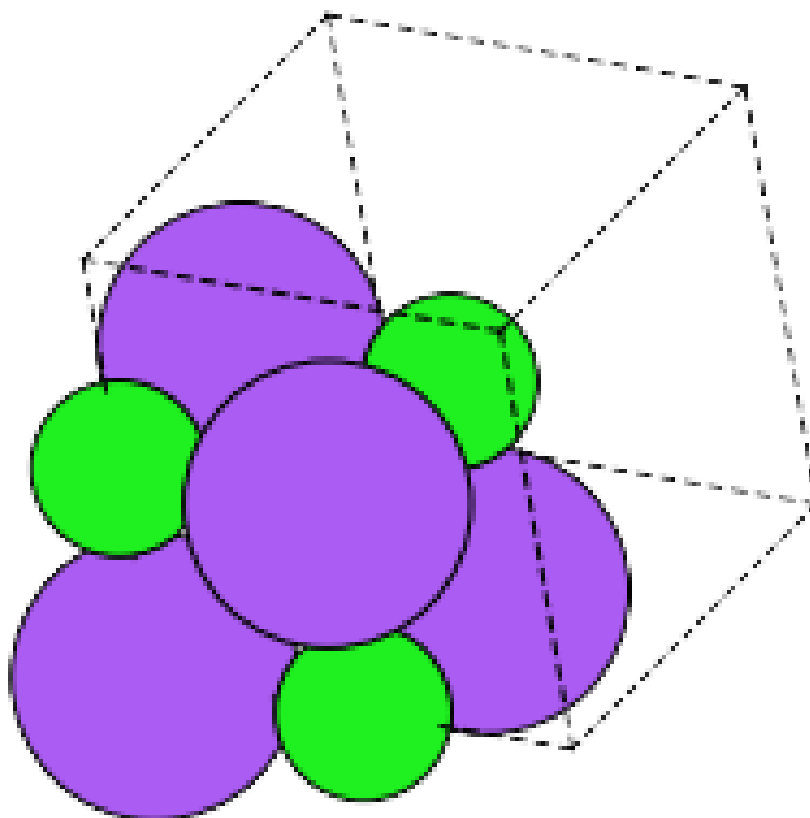


Figure 32: A view of a NaCl crystal structure.

#### 4.1.2 Using <http://materialsproject.org>

The [Materials Project](http://materialsproject.org) 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](http://pymatgen.org) project (which requires python 2.7+).

For example, I downloaded this cif file for a RuO<sub>2</sub> structure (Material ID 825).

---

```

1  #\#CIF1.1
2  #####
3  #           Crystallographic Information Format file
4  #           Produced by PyCifRW module
5  #
6  # This is a CIF file. CIF has been adopted by the International
7  # Union of Crystallography as the standard for data archiving and
8  # transmission.
9  #
10 # For information on this file format, follow the CIF links at
11 # http://www.iucr.org
12 #####
13
14 data_RuO2
15 _symmetry_space_group_name_H-M          'P 1'
```

```

16 _cell_length_a          3.13970109
17 _cell_length_b          4.5436378
18 _cell_length_c          4.5436378
19 _cell_angle_alpha       90.0
20 _cell_angle_beta        90.0
21 _cell_angle_gamma       90.0
22 _chemical_name_systematic 'Generated by pymatgen'
23 _symmetry_Int_Tables_number 1
24 _chemical_formula_structural RuO2
25 _chemical_formula_sum      'Ru2 O4'
26 _cell_volume             64.8180127062
27 _cell_formula_units_Z     2
28 loop_
29   _symmetry_equiv_pos_site_id
30   _symmetry_equiv_pos_as_xyz
31   1 'x, y, z'
32
33 loop_
34   _atom_site_type_symbol
35   _atom_site_label
36   _atom_site_symmetry_multiplicity
37   _atom_site_fract_x
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
43   O 01 1 0.000000 0.694330 0.694330 0 . 1
44   O 02 1 0.500000 0.805670 0.194330 0 . 1
45   O 03 1 0.000000 0.305670 0.305670 0 . 1
46   O 04 1 0.500000 0.194330 0.805670 0 . 1
47   Ru Ru5 1 0.500000 0.500000 0.500000 0 . 1
48   Ru Ru6 1 0.000000 0.000000 0.000000 0 . 1

```

---

We can read this file in with `ase.io.read`. That function automatically recognizes the file type by the extension.

```

1 from ase.io import read, write
2
3 atoms = read('bulk/Ru2O4_1.cif')
4
5 write('images/Ru2O4.png', atoms, show_unit_cell=2)

```

---

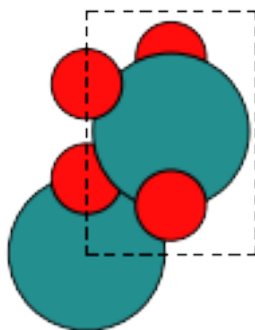


Figure 33: An RuO<sub>2</sub> unit cell prepared from a cif file.

You can also download the VASP files. I have copied these files (INCAR, POSCAR, KPOINTS) to a directory (bulk/Ru2O4), and now we can run a calculation like this:

---

```

1 from jasp import *
2
3 with jasp('bulk/Ru204',
4          xc='PBE',
5          setups={'Ru': '_pv'}) as calc:
6     calc.calculate()
7     print calc

```

---

```

: -----
VASP calculation from /home/jkitchin/dft-org/bulk/Ru204
converged: True
Energy = -44.302690 eV

Unit cell vectors (angstroms)
      x      y      z      length
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,  yy,  zz,  yz,  xz,  xy
              -0.002 -0.000 -0.000 -0.000 -0.000 -0.000
Atom#  sym      position [x,y,z]      tag  rmsForce constraints
  0    Ru  [1.569      2.270      2.270]    0   0.00      T T T
  1    Ru  [0.000      0.000      0.000]    0   0.00      T T T
  2     O  [0.000      3.153      3.153]    0   0.00      T T T
  3     O  [1.569      3.659      0.882]    0   0.00      T T T
  4     O  [0.000      1.388      1.388]    0   0.00      T T T
  5     O  [1.569      0.882      3.659]    0   0.00      T T T
-----

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: [0.6, 0.6, 0.6, 0.6, 0.6, 0.6]
prec: Accurate

```

```

    algo: Fast
    lreal: Auto
    kpts: [8, 6, 6]
    reciprocal: False
    setups: {'Ru': '_pv'}
    xc: PBE
    txt: -
    gamma: False

```

Pseudopotentials used:

-----

```

Ru: potpaw_PBE/Ru_pv/POTCAR (git-hash: c29610ef9b7bfa353e710b09dfadcd2b0fb0d274)
O: 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.

## 4.2 Computaional parameters that are important for bulk structures

### 4.2.1 k-point convergence

In the section on molecules, we learned that the total energy is a function of the planewave cutoff energy (ENCUT) used. In bulk systems that is true also. There is also another calculation parameter you must consider, the k-point grid. The k-point grid is a computational tool used to approximate integrals of some property, e.g. the electron density, over the entire unit cell. The integration is performed in reciprocal space (i.e. in the Brillouin zone) for convenience and efficiency, and the k-point grid is where the property is sampled for the integration. The higher the number of sampled points, the more accurately the integrals are approximated.

We will typically use a Monkhorst-Pack<sup>28</sup> k-point grid, which is essentially a uniformly spaced grid in the Brillouin zone. Another less commonly used scheme is the Chadi-Cohen k-point grid.<sup>27</sup> The Monkhorst-Pack grids are specified as  $n1 \times n2 \times n3$  grids, and the total number of k-points is  $n1 \cdot n2 \cdot n3$ . The computational cost is linear in the total number of k-points, so a calculation on a  $4 \times 4 \times 4$  grid will be roughly 8 times more expensive than on a  $2 \times 2 \times 2$  grid. Hence, one seeks again to balance convergence with computational tractability. Below we consider the k-point convergence of fcc Ag.

---

```

1 from ase.lattice.cubic import FaceCenteredCubic
2 from jasp import *
3
4 atoms = FaceCenteredCubic('Ag')
5
6 KPTS = [2, 3, 4, 5, 6, 8, 10]
7
8 TE = []
9

```

```

10 ready = True
11 for k in KPTS:
12     with jasp('bulk/Ag-kpts-{}'.format(k),
13             xc='PBE',
14             kpts=(k, k, k), #specifies the Monkhorst-Pack grid
15             encut=300,
16             atoms=atoms) as calc:
17         try:
18             TE.append(atoms.get_potential_energy())
19         except (VaspSubmitted, VaspQueued):
20             ready = False
21
22 if not ready:
23     import sys; sys.exit()
24
25 import matplotlib.pyplot as plt
26
27 # consider the change in energy from lowest energy state
28 TE = np.array(TE)
29 TE -= TE.min()
30
31 plt.plot(KPTS, TE)
32 plt.xlabel('number of k-points in each dimension')
33 plt.ylabel('Total Energy (eV)')
34 plt.savefig('images/Ag-kpt-convergence.png')
35 plt.show()

```

---

None

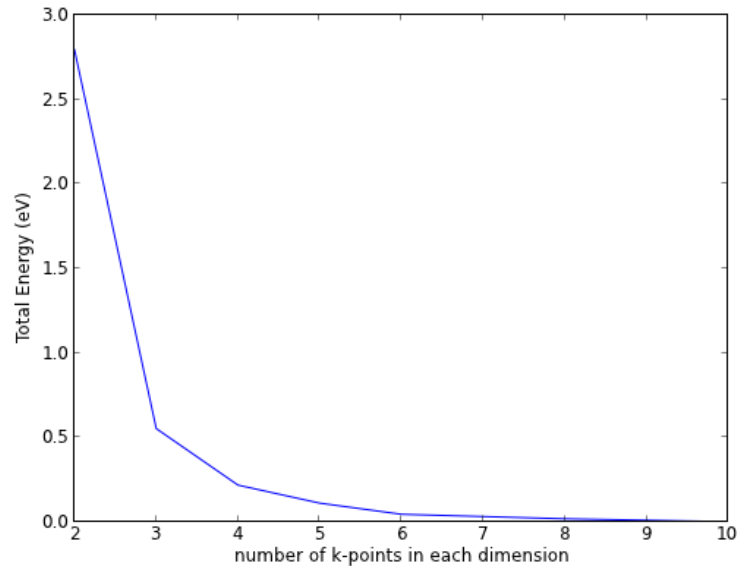


Figure 34: k-point convergence of the total energy of fcc Ag.

Based on this figure, we need at least a  $6 \times 6 \times 6$  k-point grid to achieve a convergence level of at least 50 meV. Note: the k-point convergence is not always monotonic like it is in this example, and sometimes very dense grids (e.g. up to  $20 \times 20 \times 20$ ) are needed for highly converged properties such as the density of states in smaller unit cells. Oscillations in the total



energy are typical, and it can be difficult to get high levels of convergence. The best practices are to use the same k-point sampling grid in energy differences where possible, and dense (high numbers of k-points) otherwise. It is important to check for convergence in these cases.

As unit cells get larger, the number of k-points required becomes smaller. For example, if a  $1 \times 1 \times 1$  fcc unit cell shows converged energies in a  $12 \times 12 \times 12$  k-point grid, then a  $2 \times 2 \times 2$  fcc unit cell would show the same level of convergence with a  $6 \times 6 \times 6$  k-point grid. In other words, doubling the unit cell vectors results in a halving of the number of k-points.

Sometimes you may see k-points described as k-points per reciprocal atom. For example, a  $12 \times 12 \times 12$  k-point grid for a primitive fcc unit cell would be 1728 k-points per reciprocal atom. A  $2 \times 2 \times 2$  fcc unit cell has eight atoms in it, or 0.125 reciprocal atoms, so a  $6 \times 6 \times 6$  k-point grid has 216 k-points in it, or  $216/0.125 = 1728$  k-points per reciprocal atom, the same as we discussed before.

In the k-point convergence example above, we used a  $6 \times 6 \times 6$  k-point grid on a unit cell with four atoms in it, leading to 864 k-points per reciprocal atom. If we had instead used the primitive unit cell, we would need either a  $9 \times 9 \times 9$  or  $10 \times 10 \times 10$  k-point grid to get a similar level of accuracy. In this case, there is no exact matching of k-point grids due to the difference in shape of the cells.

#### 4.2.2 Effect of SIGMA

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

Some rules to keep in mind:

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

The consequence of this finite temperature is that additional bands must be included in the calculation to allow for the partially occupied states above the Fermi level; the number of extra bands depends on the temperature used. An example of the maximum occupancies of the bands for an Al(111) slab as a function of the Fermi temperature is shown in Figure [35](#). Obviously, as the Fermi temperature approaches 0 K, the occupancy approaches a step function. It is preferable that the occupancy of several of the highest bands be zero (or at least of order  $1 \times 10^{-8}$ ) to ensure enough variational freedom was available in the calculation. Consequently, it is suggested that fifteen to twenty extra bands be used for a Fermi temperature of  $k_b T = 0.20$  eV, or that the Fermi temperature be decreased to  $k_b T = 0.10$  eV. In any case, it should be determined that enough bands were used by examination of the occupancies. It is undesirable to have too many extra bands, as this will add computational time.

Below we show the effect of SIGMA on the band occupancies.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  import matplotlib.pyplot as plt
4
5  a = 3.61
6  atoms = Atoms([Atom('Cu', (0, 0, 0))],
7                cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
8                                          [0.0, 1.0, 1.0],
9                                          [1.0, 0.0, 1.0]]).repeat((2,2,2))
10
11  SIGMA = [0.001, 0.05, 0.1, 0.2, 0.5]
12
13  for sigma in SIGMA:
14
15      with jasp('bulk/Cu-sigma-{}'.format(sigma),
16              xc='PBE',
17              encut=350,
18              kpts=(4, 4, 4),
19              ismear=-1,
20              sigma=sigma,
21              nbands=9 * 8,
22              atoms=atoms) as calc:
23          e = atoms.get_potential_energy()
24
25          nbands = calc.nbands
26          nkpts = len(calc.get_ibz_k_points())
27
28          occ = np.zeros((nkpts, nbands))
29          for i in range(nkpts):
30              occ[i,:] = calc.get_occupation_numbers(kpt=i)
31
32          max_occ = np.max(occ,axis=0) #axis 0 is columns
33
34          plt.plot(range(nbands), max_occ, label='$\sigma = {}'.format(sigma))
35
36  plt.xlabel('band number')
37  plt.ylabel('maximum occupancy (electrons)')
38  plt.ylim([-0.1, 2.1])
39  plt.legend(loc='best')
40  plt.savefig('images/occ-sigma.png')
41  plt.show()

```

---

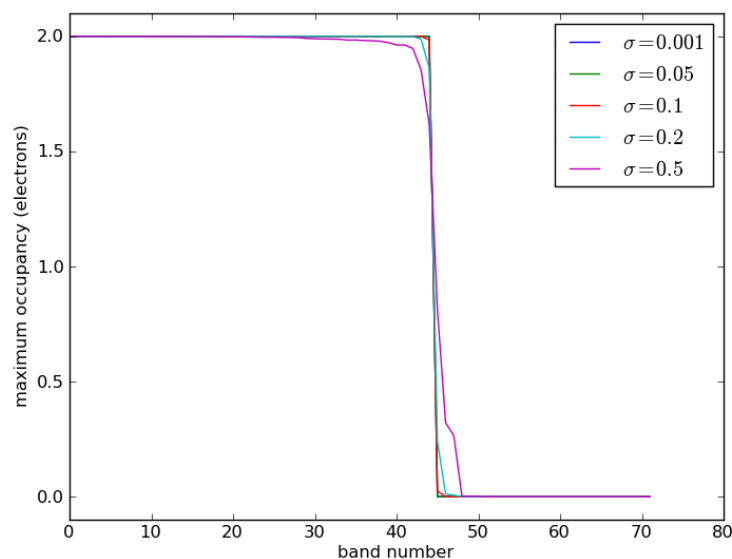


Figure 35: Effects of SIGMA on the occupancies of the Cu system.

### 4.2.3 The number of bands

In the last figure, it is evident that due to the smearing of the electronic states you need to have extra bands to accommodate the electrons above the Fermi level, and the higher the [SIGMA](#) value is, the more bands you need. You need enough bands so that the highest energy bands are unoccupied, and VASP will give you a warning that looks like this:

```
-----
|
|  ADVICE TO THIS USER RUNNING 'VASP/VAMP'   (HEAR YOUR MASTER'S VOICE ...):
|
|  Your highest band is occupied at some k-points! Unless you are
|  performing a calculation for an insulator or semiconductor, without
|  unoccupied bands, you have included TOO FEW BANDS!! Please increase
|  the parameter NBANDS in file 'INCAR' to ensure that the highest band
|  is unoccupied at all k-points. It is always recommended to
|  include a few unoccupied bands to accelerate the convergence of
|  molecular dynamics runs (even for insulators or semiconductors).
|  Because the presence of unoccupied bands improves wavefunction
|  prediction, and helps to suppress 'band-crossings.'
|  Following all k-points will be listed (with the Fermi weights of
|  the highest band given in paranthesis) ... :
|
|           6      (-0.01472)
|           8      (-0.01413)
|          13      (-0.01733)
|          14      (-0.01838)
|
```

```
|
| The total occupancy of band no. 49 is -0.00932 electrons ...
|
```

---

We tell VASP the number of bands to use with the `NBANDS` keyword. VASP will set the `NBANDS` automatically if you do not provide a value, but this is in general bad practice (even though it is often done in this book!). There are a few general guidelines for setting `NBANDS`. First we recognize that a band can only have two electrons in it (one spin up, and one spin down) in an calculation without spin-polarization, or one electron per band for a spin-polarized calculation (note that spin-polarization doubles the number of bands). There absolutely must be enough bands to accommodate all the electrons, so the minimum number of bands is `int(ceil(nelectrons/2))`.

---

```
1 import numpy as np
2
3 print int(np.ceil(50/2.))
4 print int(np.ceil(51/2.))
```

---

```
25
26
```

However, due to the smearing, the minimum number of bands is almost never enough, and we always add more bands. The default behavior in vasp is:

non-spin polarized	$\text{NELECT}/2 + \text{NIONS}/2$
spin-polarized	$0.6 * \text{NELECT} + \text{NMAGIONS}$

These do not always work, especially for small molecular systems where `NIONS/2` may be only 1, or transition metals where it may be necessary to add up to  $2 * \text{NIONS}$  extra bands.

To figure out how many bands you need, it is necessary to know how many electrons are in your calculation. The `jasp.get_valence_electrons` provides this for you. Alternatively, you can look in the [Appendix](#) for a table listing the number of valence electrons for each POTCAR file. Armed with this information you can set `NBANDS` the way you want.

---

```
1 from jasp import *
2
3 from ase import Atom, Atoms
4
5 atoms = Atoms([Atom('Cu', [0.000, 0.000, 0.000])],
6               cell= [[ 1.818, 0.000, 1.818],
7                      [ 1.818, 1.818, 0.000],
8                      [ 0.000, 1.818, 1.818]])
9
10 with jasp('bulk/alloy/cu',
11          xc='PBE',
12          encut=350,
13          kpts=(13,13,13),
14          nbands=9,
15          ibrion=2,
16          isif=4,
17          nsw=10,
18          atoms=atoms) as calc:
19     print calc.get_valence_electrons()
```

---

## 11.0

For this calculation we need at least 6 bands ( $11/2=5.5$  which is rounded up to 6) and we need to include some extra bands. The default rule would only add half a band, which is not enough. We add three additional bands. This system is so small it does not substantially increase the computational cost.

If you are too trifling to do that much work, you can use the `jasp.set_nbands` to automatically set the number of bands. This function takes an argument `N` to set the number of bands to `N`, **or** an argument `f` to set the NBANDS according to the formula  $nbands = \text{int}(nelectrons/2 + \text{len}(atoms) * f)$ . The default value of `f` is 1.5. If you want the default VASP behavior, set `f=0.5`. For transition metals, it may be required that `f=2`. This function does not consider whether the calculation is spin-polarized or not. Here is an example of using `jasp.set_nbands`.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('Cu', [0.000, 0.000, 0.000])],
5                cell= [[ 1.818, 0.000, 1.818],
6                       [ 1.818, 1.818, 0.000],
7                       [ 0.000, 1.818, 1.818]])
8
9  with jasp('bulk/alloy/cu',
10           xc='PBE',
11           encut=350,
12           kpts=(13,13,13),
13           ibrion=2,
14           isif=4,
15           nsw=10,
16           atoms=atoms) as calc:
17     print calc.set_nbands(9)

```

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('Cu', [0.000, 0.000, 0.000])],
5                cell= [[ 1.818, 0.000, 1.818],
6                       [ 1.818, 1.818, 0.000],
7                       [ 0.000, 1.818, 1.818]])
8
9  with jasp('bulk/alloy/cu-setnbands',
10           xc='PBE',
11           encut=350,
12           kpts=(13,13,13),
13           ibrion=2,
14           isif=4,
15           nsw=10,
16           atoms=atoms) as calc:
17     print calc.set_nbands(f=3)
18     print calc

```

---

None

False

:

```

-----
VASP calculation from /home/jkitchin/dft-org/bulk/alloy/cu-setnbands
converged: None

```

Energy = nan eV

Unit cell vectors (angstroms)

	x	y	z	length
a0	[ 1.818	0.000	1.818]	2.571
a1	[ 1.818	1.818	0.000]	2.571
a2	[ 0.000	1.818	1.818]	2.571

a,b,c,alpha,beta,gamma (deg): 2.571 2.571 2.571 60.0 60.0 60.0

Unit cell volume = 12.017 Ang<sup>3</sup>

Stress was not computed

Atom#	sym	position [x,y,z]	tag	rmsForce	constraints
0	Cu	[0.000 0.000 0.000]	0	nan	T T T

-----  
INCAR Parameters:

-----  
nbands: 9  
nsw: 10  
ibrion: 2  
isif: 4  
encut: 350  
magmom: None  
prec: Normal  
kpts: (13, 13, 13)  
reciprocal: False  
xc: PBE  
txt: -  
gamma: False

Pseudopotentials used:

-----  
None: None (git-hash: None)

You are, of course, free to use any formula you want to set the number of bands. Some formulas I have used in the past include:

1.  $\text{NBANDS} = 0.65 \cdot \text{NELECT} + 10$
2.  $\text{NBANDS} = 0.5 \cdot \text{NELECT} + 15$
3. etc...

### 4.3 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  $\text{TiO}_2$ ?

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

### 4.3.1 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<sup>4</sup> (Chapter 1).

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  # fcc
5  LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
6  fcc_energies = []
7  ready = True
8  for a in LC:
9      atoms = Atoms([Atom('Cu', (0, 0, 0))],
10                    cell=0.5 * a*np.array([[1.0, 1.0, 0.0],
11                                           [0.0, 1.0, 1.0],
12                                           [1.0, 0.0, 1.0]]))
13
14      with jasp('bulk/Cu-{}'.format(a),
15                xc='PBE',
16                encut=350,
17                kpts=(8,8,8),
18                atoms=atoms) as calc:
19
20          try:
21              e = atoms.get_potential_energy()
22              fcc_energies.append(e)
23          except (VaspSubmitted, VaspQueued):
24              ready = False
25
26  if not ready:
27      import sys; sys.exit()
28
29  import matplotlib.pyplot as plt
30  plt.plot(LC, fcc_energies)
31  plt.xlabel('Lattice constant (Å)')
32  plt.ylabel('Total energy (eV)')
33  plt.savefig('images/Cu-fcc.png')
34
35  print '#+tblname: cu-fcc-energies'
36  print '| lattice constant (Å) | Total Energy (eV) |'
37  for lc, e in zip(LC, fcc_energies):
38      print '| {} | {}'.format(lc, e)

```

---

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

### Exercise 4.2

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

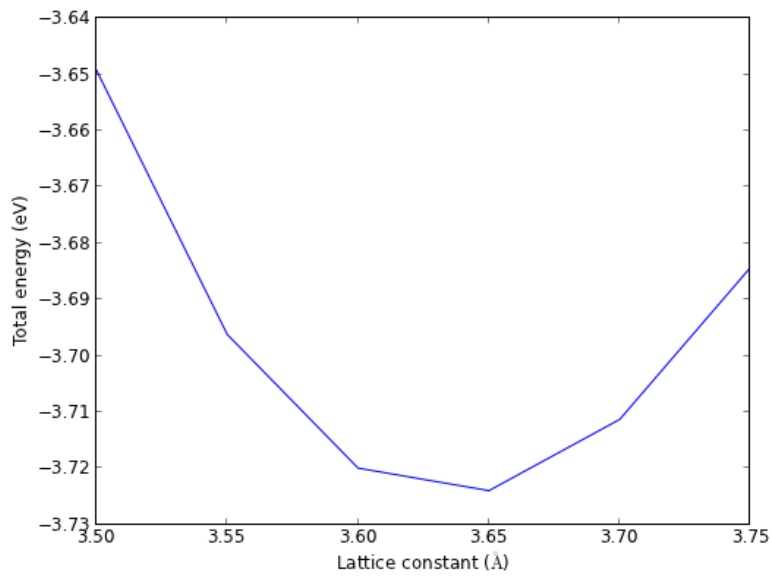


Figure 36: Total energy vs. fcc lattice constant for Cu. It appears the minimum is near 3.65 Å.

If you want to know the lattice constant that gives the lowest energy, you would fit an [equation of state](#) to the data. Here is an example using `ase.utils.eos`. See also the [appendix equations of state](#).

---

```
1 from jasp import *
2 from ase.utils.eos import EquationOfState
3 LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
4 energies = []
5 volumes = []
6 for a in LC:
7     with jasp('bulk/Cu-{}'.format(a)) as calc:
8         atoms = calc.get_atoms()
9         volumes.append(atoms.get_volume())
10        energies.append(atoms.get_potential_energy())
11
12 eos = EquationOfState(volumes, energies)
13 v0, e0, B = eos.fit()
14
15 print '''
16 v0 = {} A^3
17 E0 = {} eV
18 B = {} eV/A^3'''.format(v0, e0, B)
19
20 eos.plot('images/Cu-fcc-eos.png')
```

---



```

v0 = 12.0167118546 Å3
E0 = -3.7246811445 eV
B = 0.859009528171 eV/Å3

```

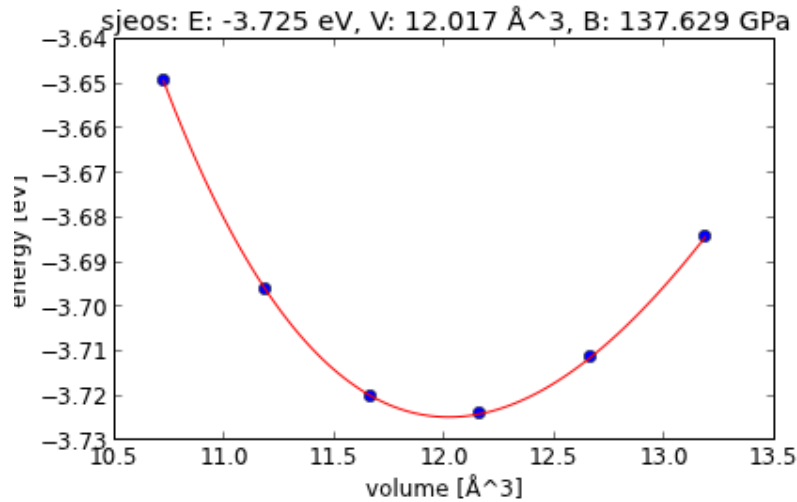


Figure 37: Total energy vs. volume for fcc Cu with fitted cubic polynomial equation of state.

Before we jump into the bcc calculations, let us consider what range of lattice constants we should choose. The fcc lattice is close-packed, and the volume of the primitive cell is  $V = 1/4a^3$  or about  $11.8 \text{ Å}^3/\text{atom}$ . 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 import numpy as np
3 a = 3.61 # lattice constant
4
5 atoms = Atoms([Atom('Cu', [0,0,0])),
6               cell=0.5 * a*np.array([[ 1.0,  1.0, -1.0],
7                                     [-1.0,  1.0,  1.0],
8                                     [ 1.0, -1.0,  1.0]]))
9
10 print 'BCC lattice constant = {0} Ang'.format(a*(11.8/atoms.get_volume())**(1./3.))

```

---

BCC lattice constant = 2.86838428403 Ang

Now we run the equation of state calculations.

---

```

1 from jasp import *
2 from ase import Atom, Atoms
3
4 LC = [2.75, 2.8, 2.85, 2.9, 2.95, 3.0]
5
6 for a in LC:

```

---

```

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

```

---

Finally, we will compare the two crystal structures.

```

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

```

---

### Exercise 4.3

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

Table 4: Total energy vs. lattice constant for BCC Cu.

Lattice constant ( $\text{\AA}^3$ )	Total energy (eV)
2.75	-3.587523
2.8	-3.668112
2.85	-3.708699
2.9	-3.716496
2.95	-3.697781
3.0	-3.657913

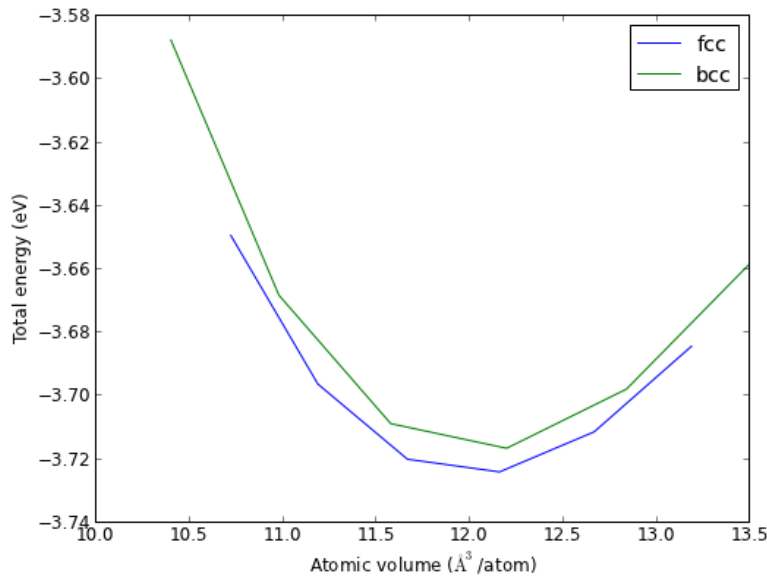


Figure 38: Comparison of energies between fcc and bcc Cu. The fcc structure is lower in energy.

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

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

#### Exercise 4.4

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

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

### 4.3.2 Optimizing the hcp lattice constant

The hcp lattice is more complicated than the fcc/bcc lattices 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 39.

---

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

---

It looks like there is a minimum in the  $a=2.7$  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 40).

---

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

---

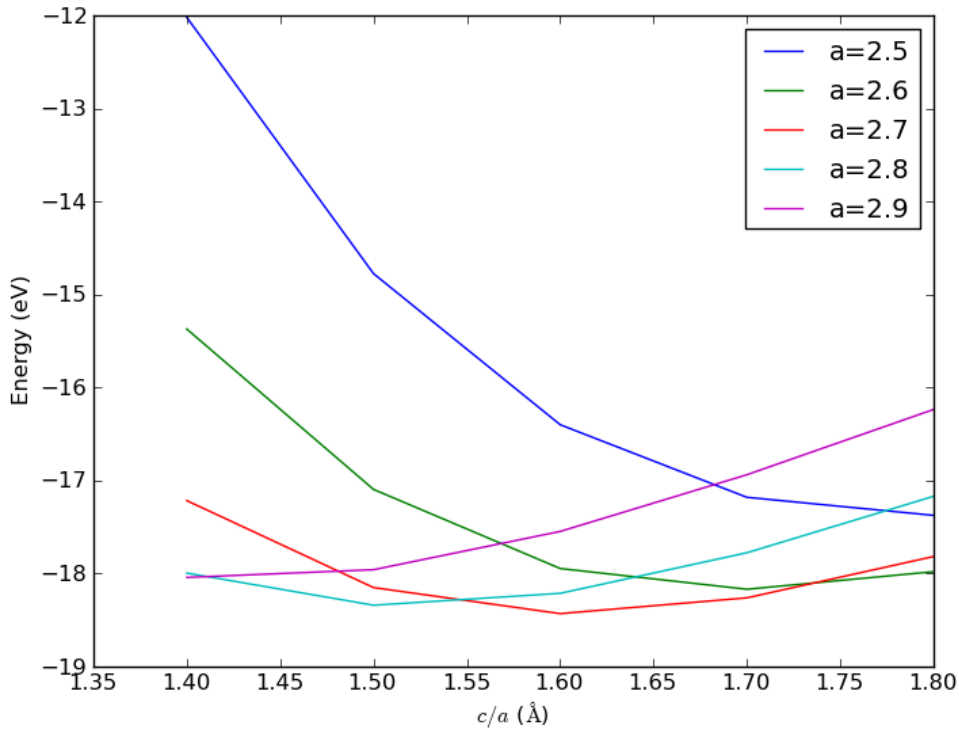


Figure 39: Total energy vs.  $c/a$  for different values of  $a$ .

```

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

```

### 4.3.3 Complex structures with internal degrees of freedom

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

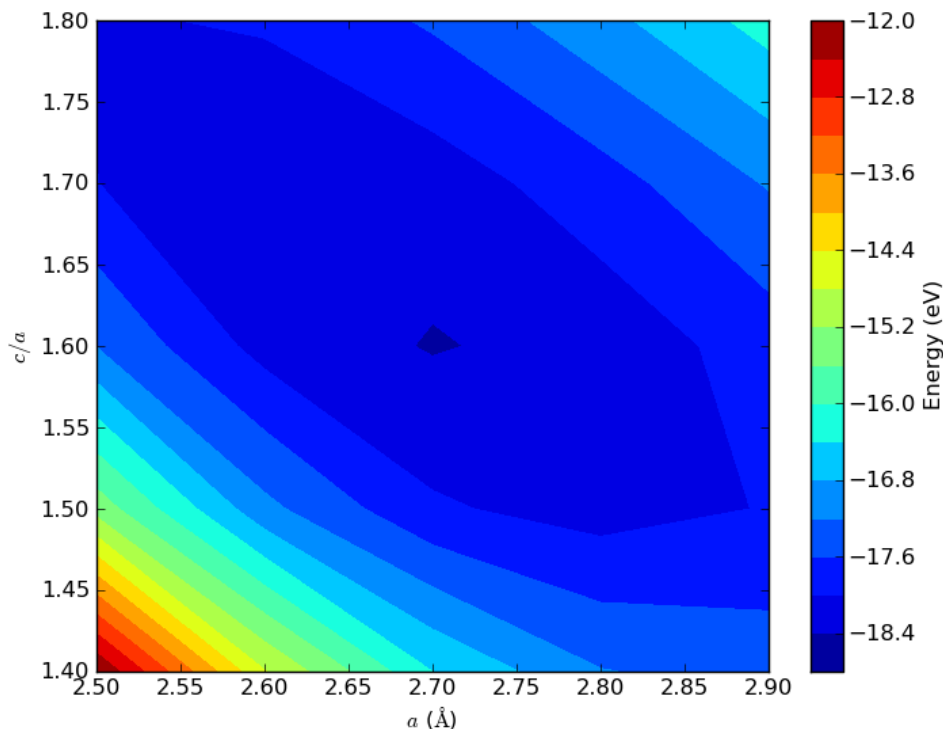


Figure 40: Contour plot of the total energy of hcp Ru for different values of  $a$  and  $c/a$ .

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

1. With the atoms frozen find the volume (at constant shape) that minimizes the total energy (INCAR:ISIF=2). The goal here is to just get an idea of where the right volume is.
2. Using the results from step 1 as a starting point, perform a set of calculations at constant volume around the minimum from step 1, but the shape and internal atom positions are allowed to change (ISIF=4).
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  $\text{TiO}_2$  which has two lattice parameters and one internal degree of freedom.

There are a few subtle points to mention in doing these calculations. The VASP [manual](#) recommends that you set `PREC` to ‘high’, and that `ENCUT` be set to  $1.3 \cdot \max(\text{ENMAX})$  of the pseudopotentials. This is necessary to avoid problems caused by small basis sets when the volume changes, and Pulay stress. It is important to ensure that the energies are reasonably converged with respect to k-point grids. Hence, it can be a significant amount of work to do this right! Let us start with determining the `ENCUT` value that is appropriate for  $\text{TiO}_2$

---

```

1 grep ENMAX $VASP_PP_PATH/POTPAW_PBE/Ti/POTCAR
2 grep ENMAX $VASP_PP_PATH/POTPAW_PBE/O/POTCAR

```

---

```

ENMAX = 178.330; ENMIN = 133.747 eV
ENMAX = 400.000; ENMIN = 300.000 eV

```

According to the manual, we should use  $ENCUT = 1.3 \times 400$  for good results.

Now we consider the k-point convergence. The lattice vectors of the rutile  $TiO_2$  structure are not all the same length, which means it is not essential that we use the same number of k-points in each direction. For simplicity, however, we do that here.

---

```

1  # step 1 frozen atoms and shape at different volumes
2  from ase import Atom, Atoms
3  import numpy as np
4  from jasp import *
5  import matplotlib.pyplot as plt
6
7  '''
8  create a TiO2 structure from the lattice vectors at
9  http://cst-www.nrl.navy.mil/lattice/struk/c4.html
10  '''
11  a = 4.59 # experimental degrees of freedom.
12  c = 2.96
13  u = 0.3 #internal degree of freedom!
14
15  #primitive vectors
16  a1 = a*np.array([1.0, 0.0, 0.0])
17  a2 = a*np.array([0.0, 1.0, 0.0])
18  a3 = c*np.array([0.0, 0.0, 1.0])
19
20  atoms = Atoms([Atom('Ti', [0., 0., 0.]),
21                Atom('Ti', 0.5 * a1 + 0.5 * a2 + 0.5 * a3),
22                Atom('O', u * a1 + u * a2),
23                Atom('O', -u * a1 - u * a2),
24                Atom('O', (0.5 + u) * a1 + (0.5 - u) * a2 + 0.5 * a3),
25                Atom('O', (0.5 - u) * a1 + (0.5 + u) * a2 + 0.5 * a3)],
26               cell=[a1, a2, a3])
27
28  KPOINTS = [2, 3, 4, 5, 6, 7, 8]
29  energies = []
30
31  ready = True
32  for k in KPOINTS:
33      with jasp('bulk/tio2/kpts-{}'.format(k),
34              encut=520,
35              kpts=(k, k, k),
36              xc='PBE',
37              sigma=0.05,
38              atoms=atoms) as calc:
39          try:
40              energies.append(atoms.get_potential_energy())
41          except (VaspSubmitted, VaspQueued):
42              ready = False
43
44  if not ready:
45      import sys; sys.exit()
46
47  plt.plot(KPOINTS, energies)
48  plt.xlabel('number of k-points in each vector')
49  plt.ylabel('Total energy (eV)')

```

```

50 plt.savefig('images/tio2-kpt-convergence.png')
51 plt.show()

```

---

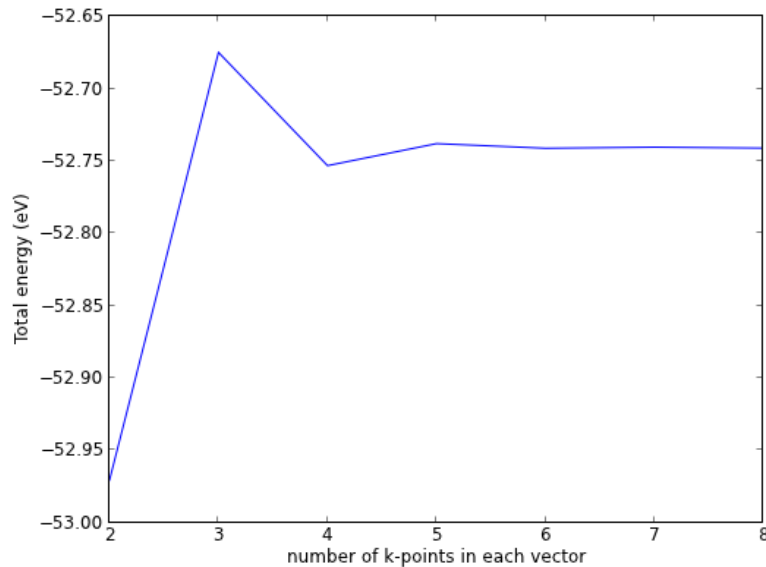


Figure 41: k-point convergence of rutile  $\text{TiO}_2$ .

A k-point grid of  $5 \times 5 \times 5$  appears suitable for reasonably converged results. Now we proceed with step 1: Compute the total energy of the unit cell allowing internal degrees of freedom to relax, but keeping a constant cell shape.

---

```

1  # step 1 frozen atoms and shape at different volumes
2  from ase import Atom, Atoms
3  import numpy as np
4  from jasp import *
5  import matplotlib.pyplot as plt
6
7  '''
8  create a TiO2 structure from the lattice vectors at
9  http://cst-www.nrl.navy.mil/lattice/struk/c4.html
10  '''
11  a = 4.59 # experimental degrees of freedom.
12  c = 2.96
13  u = 0.3 #internal degree of freedom!
14
15  #primitive vectors
16  a1 = a*np.array([1.0, 0.0, 0.0])
17  a2 = a*np.array([0.0, 1.0, 0.0])
18  a3 = c*np.array([0.0, 0.0, 1.0])
19
20  atoms = Atoms([Atom('Ti', [0., 0., 0.]),
21                Atom('Ti', 0.5 * a1 + 0.5 * a2 + 0.5 * a3),
22                Atom('O', u * a1 + u * a2),
23                Atom('O', -u * a1 - u * a2),
24                Atom('O', (0.5 + u) * a1 + (0.5 - u) * a2 + 0.5 * a3),
25                Atom('O', (0.5 - u) * a1 + (0.5 + u) * a2 + 0.5 * a3)],
26                cell=[a1, a2, a3])

```



```

27
28 v0 = atoms.get_volume()
29 cell0 = atoms.get_cell()
30
31 factors = [0.9, 0.95, 1.0, 1.05, 1.1] #to change volume by
32
33 energies, volumes = [], []
34
35 ready = True
36 for f in factors:
37     v1 = f*v0
38     cell_factor = (v1 / v0)**(1. / 3.)
39
40     atoms.set_cell(cell0 * cell_factor, scale_atoms=True)
41
42     with jasp('bulk/tio2/step1-{:0:1.2f}'.format(f),
43             encut=520,
44             kpts=(5,5,5),
45             isif=2, # relax internal degrees of freedom
46             ibrion=1,
47             nsw=50,
48             xc='PBE',
49             sigma=0.05,
50             atoms=atoms) as calc:
51         try:
52             energies.append(atoms.get_potential_energy())
53             volumes.append(atoms.get_volume())
54         except (VaspSubmitted, VaspQueued):
55             ready = False
56
57 if not ready:
58     import sys; sys.exit()
59
60 plt.plot(volumes, energies)
61 plt.xlabel('Vol. ( $\text{\AA}^3$ )')
62 plt.ylabel('Total energy (eV)')
63 plt.savefig('images/tio2-step1.png')
64
65 print '#+tblname: tio2-vol-ene'
66 print '#+caption: Total energy of TiO2 vs. volume.'
67 print '| Volume ( $\text{\AA}^3$ ) | Energy (eV) |'
68 print '|-'
69 for v, e in zip(volumes, energies):
70     print '| {0} | {1} |'.format(v, e)

```

---

Table 5: Total energy of TiO<sub>2</sub> vs. volume.

Volume ( $\text{\AA}^3$ )	Energy (eV)
56.1254184	-51.820427
59.2434972	-52.462241
62.361576	-52.761097
65.4796548	-52.801329
68.5977336	-52.64715

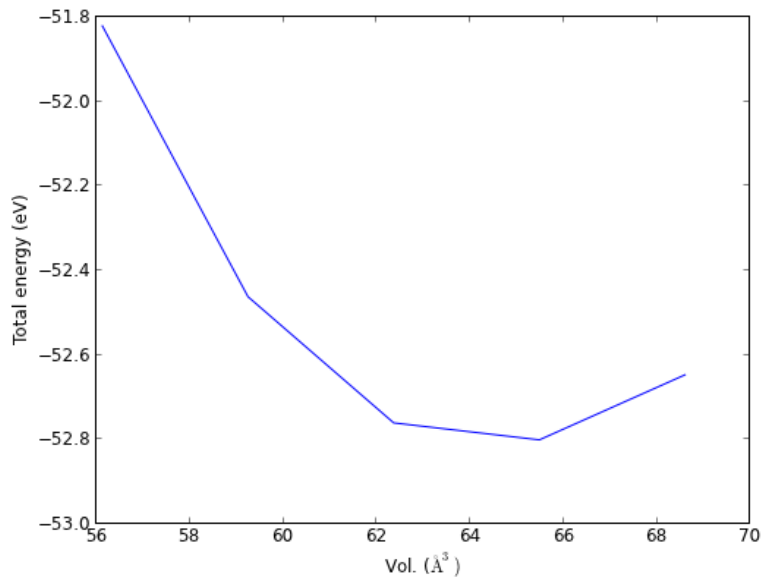


Figure 42: Total energy vs. volume for rutile  $\text{TiO}_2$  in step 1 of the optimization.

Now, we know the minimum volume is near  $64 \text{ \AA}^3$ . You could at this point fit an equation of state to find that minimum. However, we now want to use these initial starting points for a second round of optimization where we allow the unit cell shape to change, at constant volume: ISIF=4.

---

```

1  from jasp import *
2
3  factors = [0.9, 0.95, 1.0, 1.05, 1.1] #to change volume by
4
5  energies1, volumes1 = [], [] # from step 1
6  energies, volumes = [], [] # for step 2
7  ready = True
8  for f in factors:
9      with jasp('bulk/tio2/step1-{0:1.2f}'.format(f)) as calc:
10         atoms = calc.get_atoms()
11         energies1.append(atoms.get_potential_energy())
12         volumes1.append(atoms.get_volume())
13         calc.clone('bulk/tio2/step2-{0:1.2f}'.format(f))
14
15     # now set ISIF=4 and run
16     with jasp('bulk/tio2/step2-{0:1.2f}'.format(f),
17             isif=4) as calc:
18         atoms = calc.get_atoms()
19         try:
20             energies.append(atoms.get_potential_energy())
21             volumes.append(atoms.get_volume())
22         except (VaspSubmitted, VaspQueued):
23             ready = False
24
25 if not ready:
26     import sys; sys.exit()
27
28 import matplotlib.pyplot as plt
29 plt.plot(volumes1, energies1, volumes, energies)
30 plt.xlabel('Vol. ($\text{\AA}^3$)')
```

---

```

31 plt.ylabel('Total energy (eV)')
32 plt.legend(['step 1', 'step 2'], loc='best')
33 plt.savefig('images/tio2-step2.png')
34 plt.show()

```

---

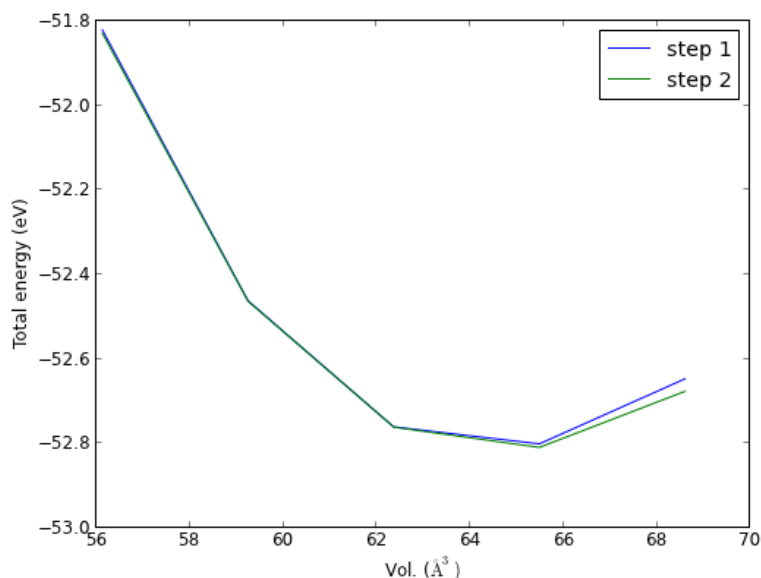


Figure 43: Total energy vs. volume for step 2 of the unit cell optimization.

The take away message here is that the total energy slightly decreases when we allow the unit cell shape to change, especially for the larger unit cell deformation. This has little effect on the minimum volume, but would have an effect on the bulk modulus, which is related to the curvature of the equation of state. At this point, you could fit an equation of state to the step 2 data, and estimate the volume at the minimum volume, and recalculate the

An alternative is a final calculation with ISIF=3, which optimizes the unit cell volume, shape and internal coordinates. It looks like the calculation at bulk/tio2/step2-1.05 is close to the minimum, so we will use that as a starting point for the final calculation.

---

```

1 from jasp import *
2
3 with jasp('bulk/tio2/step2-1.05') as calc:
4     calc.clone('bulk/tio2/step3')
5
6 with jasp('bulk/tio2/step3',
7     isif=3) as calc:
8     calc.calculate()
9     atoms = calc.get_atoms()
10    print calc
11
12 from pyspglib import spglib
13 print '\n\nThe spacegroup is {0}'.format(spglib.get_spacegroup(atoms))

```

---

: -----

VASP calculation from /home/jkitchin/dft-org/bulk/tio2/step3  
 converged: True  
 Energy = -52.818610 eV

Unit cell vectors (angstroms)

	x	y	z	length
a0	4.661	0.000	0.000	4.661
a1	0.000	4.661	0.000	4.661
a2	0.000	0.000	2.970	2.970

a,b,c,alpha,beta,gamma (deg): 4.661 4.661 2.970 90.0 90.0 90.0

Unit cell volume = 64.515 Ang<sup>3</sup>

Stress (GPa):xx, yy, zz, yz, xz, xy  
 -0.000 -0.000 -0.000 -0.000 -0.000 -0.000

Atom#	sym	position [x,y,z]	tag	rmsForce	constraints
0	Ti	[0.000 0.000 0.000]	0	0.00	T T T
1	Ti	[2.331 2.331 1.485]	0	0.00	T T T
2	O	[1.420 1.420 0.000]	0	0.00	T T T
3	O	[3.241 3.241 0.000]	0	0.00	T T T
4	O	[3.751 0.910 1.485]	0	0.00	T T T
5	O	[0.910 3.751 1.485]	0	0.00	T T T

-----  
 INCAR Parameters:

```

nbands: 20
nsw: 50
ibrion: 1
isif: 3
encut: 520.0
sigma: 0.05
magmom: None
prec: Normal
kpts: [5, 5, 5]
reciprocal: False
xc: PBE
txt: -
gamma: False
  
```

Pseudopotentials used:

-----  
 O: potpaw\_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5fbe3a3b1dfa)  
 Ti: potpaw\_PBE/Ti/POTCAR (git-hash: a29a4f0f14083cf90544a3cf1ae7fc220b31829c)

The spacegroup is P4<sub>2</sub>/mm (136)

This is the final result. You can see that the forces on all the atoms are less than 0.01 eV/Å, and the stress is also very small. The final volume is close to where we expect it to be based

on steps 1 and 2. The space group is still correct. The lattice vectors are close in length to the experimentally known values, and the angles between the vectors has not changed much. Looks good!

As a final note, the VASP [manual](#) recommends you do not use the final energy directly from the calculation, but rather run a final calculation with `ISMEAR` set to -5. Here we examine the effect.

---

```

1  from jasp import *
2
3  with jasp('bulk/tio2/step3') as calc:
4      atoms = calc.get_atoms()
5      print 'default ismear: ', atoms.get_potential_energy()
6      calc.clone('bulk/tio2/step4')
7
8  with jasp('bulk/tio2/step4',
9          ismear=-5,
10         nsw=0) as calc:
11      atoms = calc.get_atoms()
12      print 'ismear=-5:      ', atoms.get_potential_energy()

```

---

```

default ismear:  -52.81861
ismear=-5:      -52.817345

```

The difference here is on the order of a meV. That does not seem significant here. I suspect the recommended practice stems from early days when much smaller ENCUT values were used and changes in the number of basis functions were more significant.

#### 4.3.4 Effect of XC on bulk properties

The exchange correlation functional can significantly affect computed bulk properties. Here, we examine the effect on the bulk lattice constant of Pd (exp. 3.881). An excellent review of this can be found in.<sup>48</sup> We examine several functionals. The `xc` keyword in `jasp` is used to select the POTCARs. The value of `xc` can be 'LDA', 'PW91' or 'PBE'. Let us consider the LDA functional first.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  from ase.utils.eos import EquationOfState
4
5  LC = [3.75, 3.80, 3.85, 3.90, 3.95, 4.0, 4.05, 4.1]
6
7  volumes, energies = [], []
8  for a in LC:
9      atoms = Atoms([Atom('Pd', (0, 0, 0))],
10                  cell=0.5 * a*np.array([[1.0, 1.0, 0.0],
11                                         [0.0, 1.0, 1.0],
12                                         [1.0, 0.0, 1.0]]))
13      with jasp('bulk/Pd-LDA-{0}'.format(a),
14              encut=350,
15              kpts=(12,12,12),
16              xc='LDA',
17              atoms=atoms):
18          try:
19              e = atoms.get_potential_energy()
20              energies.append(e)
21              volumes.append(atoms.get_volume())

```

---

```

22         except (VaspSubmitted, VaspQueued):
23             pass
24
25 if len(energies) == len(LC):
26     eos = EquationOfState(volumes, energies)
27     v0, e0, B = eos.fit()
28     print 'LDA lattice constant is {0:1.3f} Ang^3'.format((4*v0)**(1./3.))

```

---

LDA lattice constant is 3.855 Ang<sup>3</sup>

For a GGA calculation, it is possible to specify which functional you want via the [GGA](#) tag. This tag was designed to use the LDA POTCAR files, but with a GGA functional. We will consider four different functionals here.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3  from ase.utils.eos import EquationOfState
4
5  LC = [3.75, 3.80, 3.85, 3.90, 3.95, 4.0, 4.05, 4.1]
6
7  GGA = {'AM': 'AM05',
8         'PE': 'PBE',
9         'PS': 'PBEsol',
10        'RP': 'RPBE'}
11
12 for key in GGA:
13     volumes, energies = [], []
14     for a in LC:
15         atoms = Atoms([Atom('Pd', (0, 0, 0))],
16                        cell=0.5 * a*np.array([[1.0, 1.0, 0.0],
17                                                [0.0, 1.0, 1.0],
18                                                [1.0, 0.0, 1.0]]))
19         with jasp('bulk/Pd-GGA-{}-{}'.format(a, key),
20                  encut=350,
21                  kpts=(12,12,12),
22                  xc='LDA',
23                  gga=key,
24                  atoms=atoms):
25             try:
26                 e = atoms.get_potential_energy()
27                 energies.append(e)
28                 volumes.append(atoms.get_volume())
29             except (VaspSubmitted, VaspQueued):
30                 pass
31
32 if len(energies) == len(LC):
33     eos = EquationOfState(volumes, energies)
34     v0, e0, B = eos.fit()
35     print '{1:6s} lattice constant is {0:1.3f} Ang^3'.format((4*v0)**(1./3.),
36                                                             GGA[key])
37 else:
38     print energies, LC
39     print '{0} is not ready'.format(GGA[key])

```

---

PBEsol lattice constant is 3.885 Ang<sup>3</sup>

AM05 lattice constant is 3.882 Ang<sup>3</sup>

RPBE lattice constant is 3.990 Ang<sup>3</sup>

PBE lattice constant is 3.952 Ang<sup>3</sup>

These results compare very favorably to those in.<sup>48</sup> It is typical that LDA functionals underestimate the lattice constants, and that GGAs tend to overestimate the lattice constants. PBEsol and AM05 were designed specifically for solids, and for Pd, these functionals do an exceptional job of reproducing the lattice constants. RPBE is particularly bad at the lattice constant, but it has been reported to be a superior functional for reactivity.<sup>25</sup>

## 4.4 Cohesive energy

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

---

```

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

```

---

The cohesive energy is 6.187 eV

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

1. The k-point grid used in the bulk state is not very dense. However, you can see below that the total energy is pretty converged by a  $6 \times 6 \times 6$   $k$ -point grid.
2. We did not check for convergence with the planewave cutoff.
3. We neglected spin on the atomic state. Rh in the atomic state has this electronic structure: [Kr] 4d8 5s1 and is a doublet.

First we consider the k-point convergence.

---

```

1 from jasp import *
2
3 with jasp('bulk/atomic-rh') as calc:
4     atoms = calc.get_atoms()
5     atomic_energy = atoms.get_potential_energy()
6
7 with jasp('bulk/bulk-rh') as calc:
8     atoms = calc.get_atoms()
9
10 kpts = [3, 4, 6, 9, 12, 15, 18]
11
12 for k in kpts:
13     with jasp('bulk/bulk-rh-kpts-{}'.format(k),
14             xc='PBE',
15             encut=350,
16             kpts=(k,k,k),
17             atoms=atoms) as calc:
18         e = atoms.get_potential_energy()
19
20 print '{0:2d}, {0:2d}, {0:2d}: cohesive energy = {1} eV'.format(k,e-atomic_energy)

```

---

```

( 3,  3,  3): cohesive energy = -4.837472 eV
( 4,  4,  4): cohesive energy = -6.189573 eV
( 6,  6,  6): cohesive energy = -6.210668 eV
( 9,  9,  9): cohesive energy = -6.208576 eV
(12, 12, 12): cohesive energy = -6.215367 eV
(15, 15, 15): cohesive energy = -6.215319 eV
(18, 18, 18): cohesive energy = -6.216032 eV

```

Using only 1 k-point for the bulk energy is a terrible approximation! It takes at least a  $6 \times 6 \times 6$  grid to get the total energy converged to less than 10 meV. Note we do not need to check the k-point convergence of the atomic state because it is surrounded by vacuum on all sides, and so there should not be any dispersion in the bands.

We will examine the magnetic state next.

---

```

1 from jasp import *
2 from ase.lattice.cubic import FaceCenteredCubic
3 from ase import Atoms, Atom
4 # bulk system
5 atoms = FaceCenteredCubic(directions=[[0,1,1],
6                                     [1,0,1],
7                                     [1,1,0]],
8                           size=(1,1,1),
9                           symbol='Rh')
10
11 with jasp('bulk/bulk-rh',
12         xc='PBE',
13         encut=350,
14         kpts=(4,4,4),
15         isif=3,
16         ibrion=2,
17         nsw=10,
18         atoms=atoms) as calc:
19     bulk_energy = atoms.get_potential_energy()
20
21 # atomic system

```

---



---

```

22 atoms = Atoms([Atom('Rh',[5,5,5],magmom=1)],
23               cell=(7,8,9))
24
25 with jasp('bulk/atomic-rh-sp',
26          xc='PBE',
27          encut=350,
28          kpts=(1,1,1),
29          ispin=2,
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<sup>4</sup> 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](#)).

## 4.5 TODO Elastic properties

See this reference.<sup>49</sup>

## 4.6 Bulk thermodynamics

We can predict temperature dependent thermodynamic properties of bulk materials without too much effort. As with the thermochemical properties of ideal gases, we must use some simple models that we parameterize by DFT. Here we follow the example in Reference<sup>50</sup> for computing the thermal coefficient of expansion, heat capacity, enthalpy and entropy for Ni as a function of temperature.

We start by computing the equation of state for fcc Ni.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  # fcc
5  LC = [3.5, 3.55, 3.6, 3.65, 3.7, 3.75]
6  volumes, energies = [], []
7  for a in LC:
8      atoms = Atoms([Atom('Ni', (0, 0, 0), magmom=2.5)],
9                    cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
10                                           [0.0, 1.0, 1.0],
11                                           [1.0, 0.0, 1.0]]))
12
13     with jasp('bulk/Ni-{0}'.format(a),
14            xc='PBE',
15            encut=350,
16            kpts=(12,12,12),
17            ispin=2,
18            atoms=atoms) as calc:
19         try:
20             e = atoms.get_potential_energy()
21             energies.append(e)
22             volumes.append(atoms.get_volume())
23         except:
24             pass
25

```

---

```

26 if len(energies) != len(LC):
27     import sys; sys.exit()
28
29 import matplotlib.pyplot as plt
30 plt.plot(LC, fcc_energies)
31 plt.xlabel('Lattice constant ($\AA$)')
32 plt.ylabel('Total energy (eV)')
33 plt.savefig('images/Ni-fcc.png')

```

---

## 4.7 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  $\text{TiO}_2$  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  $E_{\text{rutile}}(V)$  and  $E_{\text{anatase}}(V)$ .

---

```

1  # run the rutile calculations
2  from jasp import *
3  from ase import Atom, Atoms
4
5  B='Ti'; X='O'; a=4.59; c=2.958; u=0.305;
6  '''
7  create a rutile structure from the lattice vectors at
8  http://cst-www.nrl.navy.mil/lattice/struc/c4.html
9
10 spacegroup: 136 P4_2/mmm
11 '''
12 a1 = a*np.array([1.0, 0.0, 0.0])
13 a2 = a*np.array([0.0, 1.0, 0.0])
14 a3 = c*np.array([0.0, 0.0, 1.0])
15
16 atoms = Atoms([Atom(B, [0., 0., 0.]),
17               Atom(B, 0.5*a1 + 0.5*a2 + 0.5*a3),
18               Atom(X, u*a1 + u*a2),
19               Atom(X, -u*a1 - u*a2),
20               Atom(X, (0.5+u)*a1 + (0.5-u)*a2 + 0.5*a3),
21               Atom(X, (0.5-u)*a1 + (0.5+u)*a2 + 0.5*a3)],
22              cell=[a1, a2, a3])
23
24 nTiO2 = len(atoms)/3.
25 v0 = atoms.get_volume()
26 cell0 = atoms.get_cell()
27
28 volumes = [28., 30., 32., 34., 36.] #vol of one TiO2
29
30 for v in volumes:
31     atoms.set_cell(cell0*((nTiO2*v/v0)**(1./3.)), scale_atoms=True)
32
33     with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v),
34              encut=350,
35              kpts=(6,6,6),
36              xc='PBE',
37              ismear=0,
38              sigma=0.001,
39              isif=2,

```

```

40         ibrion=2,
41         nsw=20,
42         atoms=atoms) as calc:
43     try:
44         calc.calculate()
45     except (VaspSubmitted, VaspQueued):
46         pass

```

---

```

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

```

---

Now we will fit cubic polynomials to the data.

---

```

1  # fit cubic polynomials to E(V) for rutile and anatase
2  from jasp import *
3  import matplotlib.pyplot as plt
4  import numpy as np
5  np.set_printoptions(precision=2)
6
7  # anatase equation of stte
8  volumes = [30., 33., 35., 37., 39.] #vol of one TiO2
9  a_volumes, a_energies = [], []

```

```

10 for v in volumes:
11     with jasp('bulk/TiO2/anatase/anatase-{0}'.format(v)) as calc:
12         atoms = calc.get_atoms()
13         nTiO2 = len(atoms)/3.0
14         a_volumes.append(atoms.get_volume()/nTiO2)
15         a_energies.append(atoms.get_potential_energy()/nTiO2)
16
17 # rutile equation of state
18 volumes = [28., 30., 32., 34., 36.] #vol of one TiO2
19 r_volumes, r_energies = [], []
20 for v in volumes:
21     with jasp('bulk/TiO2/rutile/rutile-{0}'.format(v)) as calc:
22         atoms = calc.get_atoms()
23         nTiO2 = len(atoms)/3.0
24         r_volumes.append(atoms.get_volume()/nTiO2)
25         r_energies.append(atoms.get_potential_energy()/nTiO2)
26
27 # cubic polynomial fit to equation of state  $E(V) = \text{pars}[V^3 V^2 V^1 V^0]$ 
28 apars = np.polyfit(a_volumes, a_energies, 3)
29 rpars = np.polyfit(r_volumes, r_energies, 3)
30
31 print 'E_anatase(V) = {0:1.2f}*V^3 + {1:1.2f}*V^2 + {2:1.2f}*V + {3:1.2f}'.format(*apars)
32 print 'E_rutile(V) = {0:1.2f}*V^3 + {1:1.2f}*V^2 + {2:1.2f}*V + {3:1.2f}'.format(*rpars)
33 print 'anatase epars: {0!r}'.format(apars)
34 print 'rutile epars: {0!r}'.format(rpars)
35 # get pressure parameters  $P(V) = -dE/dV$ 
36 dapars = -np.polyder(apars)
37 drpars = -np.polyder(rpars)
38
39 print 'anatase ppars: {0!r}'.format(dapars)
40 print 'rutile ppars: {0!r}'.format(drpars)
41
42 print
43 print 'P_anatase(V) = {0:1.2f}*V^2 + {1:1.2f}*V + {2:1.2f}'.format(*dapars)
44 print 'P_rutile(V) = {0:1.2f}*V^2 + {1:1.2f}*V + {2:1.2f}'.format(*drpars)
45
46 vfit = np.linspace(28,40)
47
48 # plot the equations of state
49 plt.plot(a_volumes, a_energies, 'bo ', label='Anatase')
50 plt.plot(vfit, np.polyval(apars, vfit), 'b-')
51
52 plt.plot(r_volumes, r_energies, 'gs ', label='Rutile')
53 plt.plot(vfit, np.polyval(rpars, vfit), 'g-')
54
55 plt.xlabel('Volume ($\text{\AA}^3/\text{f.u.})')
56 plt.ylabel('Total energy (eV/f.u.)')
57 plt.legend()
58 plt.xlim([25,40])
59 plt.ylim([-27, -26])
60 plt.savefig('images/rutile-anatase-eos.png')

```

---

```

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

```

```

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

```

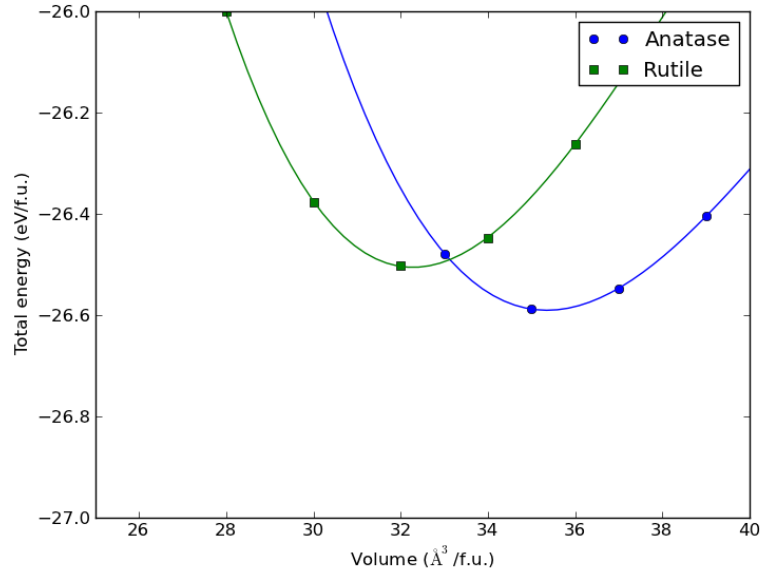


Figure 44: Equations of state ( $E(V)$ ) for anatase and rutile  $\text{TiO}_2$ .

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

$$(E_{\text{anatase}}(V_1) - E_{\text{rutile}}(V_2))/(V_1 - V_2) = P_{\text{anatase}}(V_1)$$

$$(E_{\text{anatase}}(V_1) - E_{\text{rutile}}(V_2))/(V_1 - V_2) = P_{\text{rutile}}(V_2)$$

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

---

```

1  from ase.units import GPa
2  from numpy import array, linspace, polyval
3
4  anatase_epars = array([-1.06049246e-03,  1.30279404e-01, -5.23520055e+00,
5                        4.25202869e+01])
6  rutile_epars = array([-1.24680208e-03,  1.42966536e-01, -5.33239733e+00,
7                        3.85903670e+01])
8
9  anatase_ppars = array([ 3.18147737e-03, -2.60558808e-01,  5.23520055e+00])
10 rutile_ppars = array([ 3.74040625e-03, -2.85933071e-01,  5.33239733e+00])
11
12 def func(V):
13     V1 = V[0] # rutile volume
14     V2 = V[1] # anatase volume
15
16     E_rutile = polyval(rutile_epars,V1)
17     E_anatase = polyval(anatase_epars,V2)
18
19     P_rutile = polyval(rutile_ppars,V1)
20     P_anatase = polyval(anatase_ppars,V2)
21
22     return [(E_anatase - E_rutile)/(V1-V2) - P_anatase,
23            (E_anatase - E_rutile)/(V1-V2) - P_rutile]
24
25 from scipy.optimize import fsolve
26 x0 = fsolve(func,[28,34])
27 print 'The solutions are at V = {0}'.format(x0)

```

```

28 print 'Anatase pressure: {0} GPa'.format(polyval(anatase_ppars,x0[1])/GPa)
29 print 'Rutile pressure: {0} GPa'.format(polyval(rutile_ppars,x0[0])/GPa)
30
31 # illustrate the common tangent
32 import matplotlib.pyplot as plt
33
34 vfit = linspace(28,40)
35 plt.plot(vfit, polyval(anatase_epars,vfit),label='anatase')
36 plt.plot(vfit, polyval(rutile_epars,vfit),label='rutile')
37 plt.plot(x0, [polyval(rutile_epars,x0[0]),
38               polyval(anatase_epars,x0[1])], 'ko-', label='common tangent')
39 plt.legend()
40 plt.xlabel('Volume ( $\text{\AA}^3/\text{f.u.}$ )')
41 plt.ylabel('Total energy (eV/f.u.)')
42 plt.savefig('images/eos-common-tangent.png')

```

---

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

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

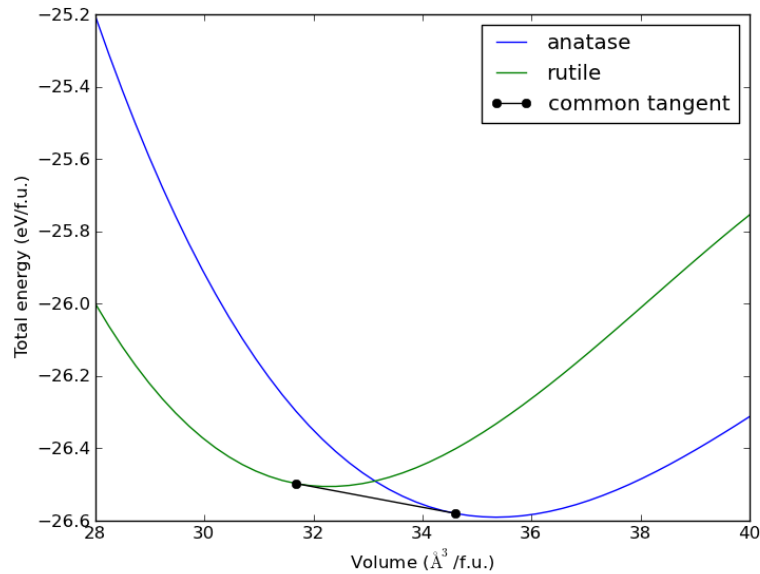


Figure 45: Illustration of the common tangent that shows the pressure where anatase and rutile coexist before anatase converts to rutile.

## 4.8 Bulk reaction energies

### 4.8.1 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

---

```
1  # get bulk Cu and Pd energies.
2  from jasp import *
3
4  from ase import Atom, Atoms
5
6  atoms = Atoms([Atom('Cu', [0.000, 0.000, 0.000])),
7                cell= [[ 1.818, 0.000, 1.818],
8                      [ 1.818, 1.818, 0.000],
9                      [ 0.000, 1.818, 1.818]])
10
11  with jasp('bulk/alloy/cu',
12            xc='PBE',
13            encut=350,
14            kpts=(13,13,13),
15            nbands=9,
16            ibrion=2,
17            isif=4,
18            nsw=10,
19            atoms=atoms) as calc:
20      cu = atoms.get_potential_energy()
21
22  atoms = Atoms([Atom('Pd', [0.000, 0.000, 0.000])),
23                cell= [[ 1.978, 0.000, 1.978],
24                      [ 1.978, 1.978, 0.000],
25                      [ 0.000, 1.978, 1.978]])
26
27  with jasp('bulk/alloy/pd',
28            xc='PBE',
29            encut=350,
30            kpts=(13,13,13),
31            nbands=9,
32            ibrion=2,
33            isif=4,
34            nsw=10,
35            atoms=atoms) as calc:
36      pd = atoms.get_potential_energy()
37
38  print 'Cu energy = {0} eV'.format(cu)
39  print 'Pd energy = {0} eV'.format(pd)
```

---

Cu energy = -3.723305 eV

Pd energy = -5.183823 eV

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

---

```
1  from jasp import *
2  from ase import Atom, Atoms
3
4  atoms = Atoms([Atom('Cu', [0.000, 0.000, 0.000]),
5                Atom('Pd', [-1.652, 0.000, 2.039])),
6                cell= [[ 0.000, -2.039, 2.039],
7                      [ 0.000, 2.039, 2.039],
8                      [-3.303, 0.000, 0.000]])
9
10  with jasp('bulk/alloy/cupd-1',
11            xc='PBE',
12            encut=350,
```

```

13         kpts=(12,12,8),
14         nbands=17,
15         ibrion=2,
16         isif=4,
17         nsw=10,
18         atoms=atoms) as calc:
19     cupd1 = atoms.get_potential_energy()
20
21
22     atoms = Atoms([Atom('Cu', [-0.049, 0.049, 0.049]),
23                    Atom('Cu', [-11.170, 11.170, 11.170]),
24                    Atom('Pd', [-7.415, 7.415, 7.415]),
25                    Atom('Pd', [-3.804, 3.804, 3.804])],
26                    cell=[[-5.629, 3.701, 5.629 ],
27                          [-3.701, 5.629, 5.629 ],
28                          [-5.629, 5.629, 3.701 ]])
29
30     with jasp('bulk/alloy/cupd-2',
31              xc='PBE',
32              encut=350,
33              kpts=(8,8,8),
34              nbands=34,
35              ibrion=2,
36              isif=4,
37              nsw=10,
38              atoms=atoms) as calc:
39         cupd2 = atoms.get_potential_energy()
40
41     print 'cupd-1 = {0} eV'.format(cupd1)
42     print 'cupd-2 = {0} eV'.format(cupd2)

```

---

```

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

```

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

```

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

```

---

```

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:



$2\text{Cu} + 2\text{Pd} \rightarrow \text{Cu}_2\text{Pd}_2$  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()
5      cu = atoms.get_potential_energy()/len(atoms)
6
7  with jasp('bulk/alloy/pd') as calc:
8      atoms = calc.get_atoms()
9      pd = atoms.get_potential_energy()/len(atoms)
10
11 with jasp('bulk/alloy/cupd-1') as calc:
12     atoms = calc.get_atoms()
13     e1 = atoms.get_potential_energy()
14     for atom in atoms:
15         if atom.symbol == 'Cu':
16             e1 -= cu
17         else:
18             e1 -= pd
19     e1 /= len(atoms)
20
21 with jasp('bulk/alloy/cupd-2') as calc:
22     atoms = calc.get_atoms()
23     e2 = atoms.get_potential_energy()
24     for atom in atoms:
25         if atom.symbol == 'Cu':
26             e2 -= cu
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 46.

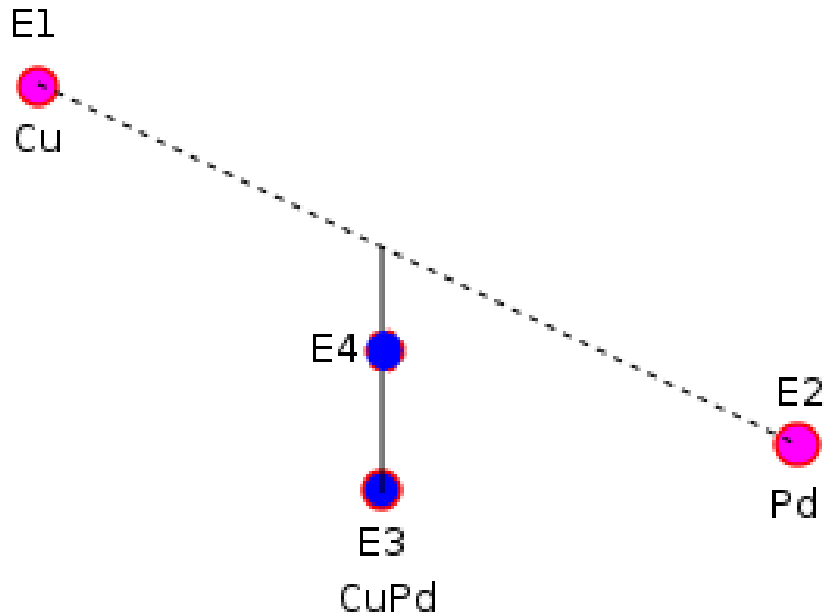


Figure 46: Conceptual picture of two alloys with exothermic formation energies. The dashed line represents a composition weighted average energy of the parent metals. E4 and E3 are energies associated with two different alloy structures at the same composition. Both structures are more stable than a mixture of pure metals with the same composition, but E3 is more stable than E4.

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

---

```

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

```

```

21 with jasp('bulk/alloy/cu3pd-1',
22           xc='PBE',
23           encut=350,
24           kpts=(8,8,8),
25           nbands=34,
26           ibrion=2,
27           isif=4,
28           nsw=10,
29           atoms=atoms) as calc:
30     e3 = atoms.get_potential_energy()
31     for atom in atoms:
32       if atom.symbol == 'Cu':
33         e3 -= cu
34       else:
35         e3 -= pd
36     e3 /= len(atoms)
37 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
4 print 'Composition weighted average = {0} eV'.format(Hf1 + (x0-x)/(x0-x3)*(Hf3 - Hf1))

```

---

Composition weighted average = -0.06 eV

We find the weighted composition formation energy of pure Cu and cupd-1 is more favorable than the formation energy of cu3pd-1. Therefore, we could expect that structure to phase separate into a mixture of pure Cu and cupd-1. Schematically what we are seeing is shown in Figure 47.

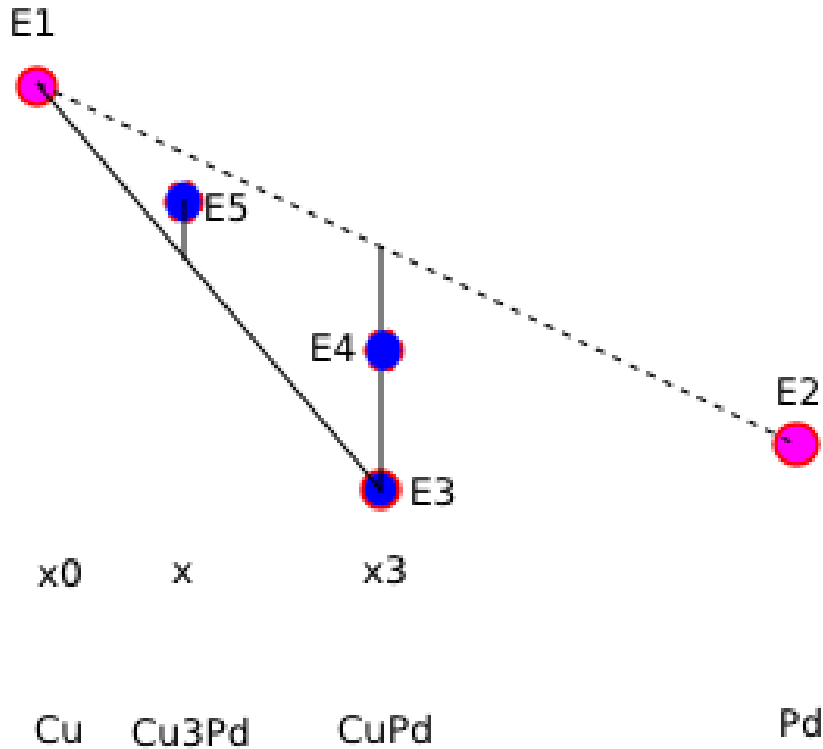


Figure 47: Illustration of of an alloy structure with an exothermic formation energy that is not stable with respect to phase separation. The solid line shows the composition weighted average energy of a mixture of Cu and cupd-2. Since the energy of cu3pd-1 is above the solid line, it is less favorable than a mixture of Cu and cupd-2 with the same composition.

Finally, let us consider one more structure with the  $\text{Cu}_3\text{Pd}$  stoichiometry.

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  # parent metals
5  with jasp('bulk/alloy/cu') as calc:
6      atoms = calc.get_atoms()
7      cu = atoms.get_potential_energy()/len(atoms)
8
9  with jasp('bulk/alloy/pd') as calc:
10     atoms = calc.get_atoms()
11     pd = atoms.get_potential_energy()/len(atoms)
12
13  atoms = Atoms([Atom('Cu', [-1.867, 1.867, 0.000]),
14                Atom('Cu', [0.000, 0.000, 0.000]),
15                Atom('Cu', [0.000, 1.867, 1.867]),
16                Atom('Pd', [-1.867, 0.000, 1.867])],
17                cell=[[-3.735, 0.000, 0.000],
18                    [0.000, 0.000, 3.735],
19                    [0.000, 3.735, 0.000]])
20
21  with jasp('bulk/alloy/cu3pd-2',
22            xc='PBE',

```

```

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

```

---

Delta Hf cu3pd-2 = -0.10 eV/atom

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

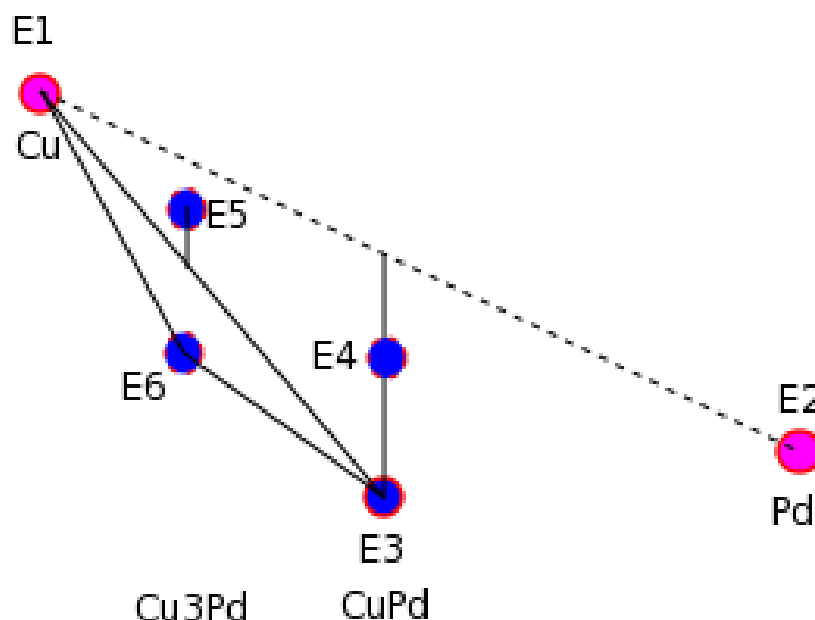


Figure 48: Illustration that cu3pd-2 is more stable than cu3pd-1 and that it is more stable than a composition weighted mixture of Cu and cupd-1. The dotted line shows the energy of a composition weighted average energy of a mixture of Cu and cupd-1. Since cu3pd-2 is below the dotted line, it is more stable than the phase-separated mixture.

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)<sup>51;52</sup> which uses a cluster expansion methodology.

#### 4.8.2 Metal oxide oxidation energies

We will consider here the reaction  $2 \text{Cu}_2\text{O} + \text{O}_2 \rightleftharpoons 4 \text{CuO}$ . The reaction energy is:

$$\Delta E = 4E_{\text{CuO}} - 2E_{\text{Cu}_2\text{O}} - E_{\text{O}_2}.$$

We need to compute the energy of each species.

##### Cu<sub>2</sub>O calculation

---

```

1 # run Cu2O calculation
2 from jasp import *
3 from ase import Atom, Atoms
4
5 #http://phycomp.technion.ac.il/~ira/types.html#Cu2O
6 a = 4.27

```

```

7
8 atoms = Atoms([Atom('Cu',[0,0,0]),
9               Atom('Cu',[0.5, 0.5, 0.0]),
10              Atom('Cu',[0.5, 0.0, 0.5]),
11              Atom('Cu',[0.0, 0.5, 0.5]),
12              Atom('O',[0.25, 0.25, 0.25]),
13              Atom('O',[0.75, 0.75, 0.75])])
14
15 atoms.set_cell((a,a,a), scale_atoms=True)
16
17 with jasp('bulk/Cu2O',
18          encut=400,
19          kpts=(8,8,8),
20          ibrion=2,
21          isif=3,
22          nsw=30,
23          xc='PBE',
24          atoms=atoms) as calc:
25     calc.set_nbands()
26     calc.calculate()
27     print calc

```

---

: -----

VASP calculation from /home/jkitchin/dft-org/bulk/Cu2O

converged: True

Energy = -27.204590 eV

Unit cell vectors (angstroms)

	x	y	z	length
a0	4.270	0.000	0.000	4.270
a1	0.000	4.270	0.000	4.270
a2	0.000	0.000	4.270	4.270

a,b,c,alpha,beta,gamma (deg): 4.270 4.270 4.270 90.0 90.0 90.0

Unit cell volume = 77.854 Ang<sup>3</sup>

Stress (GPa):xx, yy, zz, yz, xz, xy  
0.000 0.000 0.000 -0.000 -0.000 -0.000

Atom#	sym	position [x,y,z]	tag	rmsForce	constraints
0	Cu	[0.000 0.000 0.000]	0	0.00	T T T
1	Cu	[2.135 2.135 0.000]	0	0.00	T T T
2	Cu	[2.135 0.000 2.135]	0	0.00	T T T
3	Cu	[0.000 2.135 2.135]	0	0.00	T T T
4	O	[1.067 1.067 1.067]	0	0.00	T T T
5	O	[3.202 3.202 3.202]	0	0.00	T T T

-----

INCAR Parameters:

-----

nbands: 37

nsw: 30

ibrion: 2

isif: 3

encut: 400.0

```

    magmom: None
    prec: Normal
    kpts: (8, 8, 8)
    reciprocal: False
    xc: PBE
    txt: -
    gamma: False

```

Pseudopotentials used:

-----

```

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

```

## CuO calculation

---

```

1  # run CuO calculation
2  from jasp import *
3  from ase import Atom, Atoms
4
5  # CuO
6  # http://cst-www.nrl.navy.mil/lattice/struk/b26.html
7  # http://www.springermaterials.com/docs/info/10681727_51.html
8  a = 4.6837
9  b = 3.4226
10 c = 5.1288
11 beta = 99.54/180*np.pi
12 y = 0.5819
13
14 a1 = np.array([0.5*a, -0.5*b, 0.0])
15 a2 = np.array([0.5*a, 0.5*b, 0.0])
16 a3 = np.array([c*np.cos(beta), 0.0, c*np.sin(beta)])
17
18 atoms = Atoms([Atom('Cu', 0.5*a2),
19                Atom('Cu', 0.5*a1 + 0.5*a3),
20                Atom('O', -y*a1 + y*a2 + 0.25*a3),
21                Atom('O', y*a1 - y*a2 - 0.25*a3)],
22               cell=(a1, a2, a3))
23
24 with jasp('bulk/CuO',
25           encut=400,
26           kpts=(8,8,8),
27           ibrion=2,
28           isif=3,
29           nsw=30,
30           xc='PBE',
31           atoms=atoms) as calc:
32     calc.set_nbands()
33     calc.calculate()
34     print calc

```

---

: -----

```

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

```

Unit cell vectors (angstroms)



```

      x      y      z      length
a0 [ 2.302 -1.776  0.046] 2.908
a1 [ 2.302  1.776  0.046] 2.908
a2 [-0.762  0.000  5.087] 5.144
a,b,c,alpha,beta,gamma (deg): 2.908 2.908 5.144 95.8 95.8 95.8
Unit cell volume = 41.730 Ang^3
Stress (GPa):xx,   yy,   zz,   yz,   xz,   xy
              0.008  0.013  0.009 -0.000  0.000 -0.000
Atom#  sym      position [x,y,z]      tag  rmsForce constraints
  0    Cu  [1.151      0.888      0.023]  0  0.00      T T T
  1    Cu  [0.770     -0.888      2.566]  0  0.00      T T T
  2     O  [2.111     -0.168      1.318]  0  0.03      T T T
  3     O  [1.730      0.168      3.861]  0  0.03      T T T
-----

```

#### INCAR Parameters:

```

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

```

#### Pseudopotentials used:

```

-----
O: potpaw_PBE/O/POTCAR (git-hash: 9a0489b46120b0cad515d935f44b5f5be3a3b1dfa)
Cu: potpaw_PBE/Cu/POTCAR (git-hash: a44c591415026f53deb16a99ca3f06b1e69be10b)

```

#### Reaction energy calculation

---

```

1  from jasp import *
2
3  # don't forget to normalize your total energy to a formula unit. Cu2O
4  # has 3 atoms, so the number of formula units in an atoms is
5  # len(atoms)/3.
6  with jasp('bulk/Cu2O') as calc:
7      atoms = calc.get_atoms()
8      cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
9
10 with jasp('bulk/CuO') as calc:
11     atoms = calc.get_atoms()
12     cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
13

```

```

14 # make sure to use the same cutoff energy for the O2 molecule!
15 with jasp('molecules/O2-sp-triplet-400') as calc:
16     atoms = calc.get_atoms()
17     o2_energy = atoms.get_potential_energy()
18
19 rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
20 print 'Reaction energy = {0} eV'.format(rxn_energy)

```

---

Reaction energy = -1.966168 eV

There are a few reasons why our number does not agree with the experimental reaction energy. One reason is related to errors in the O<sub>2</sub> dissociation energy, and another reason is related to localization of electrons in the Cu 3d orbitals.<sup>53</sup> The first error of incorrect O<sub>2</sub> dissociation error is a systematic error that can be corrected empirically.<sup>53</sup> Fixing the second error requires the application of DFT+U (see [DFT+U](#))

## 4.9 Bulk density of states

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

The solution to Eq. (1) yields a set of Kohn-Sham (K-S) orbitals and an associated set of eigenvalues that correspond to the energies of these orbitals, neither of which have any known directly observable meaning.<sup>18</sup> The sum of the squared K-S orbitals, however, is equal to the electron density (Eq. (3)), and the sum of the eigenvalues is a significant part of the total energy (Eq. (4)). Thus, it seems reasonable to suppose these quantities have other significant relationships to physical observables. Perdew et al. showed that the highest occupied eigenvalue is equal to the ionization energy of a system within an exact density functional theory,<sup>24</sup> but their interpretation has been vigorously debated in the literature,<sup>54–56</sup> and is only true for the exact exchange/correlation functional, not the approximate ones used in practice<sup>3</sup> Section 5.3.3. Stowasser and Hoffmann discussed an approach to using the K-S orbitals in more traditional molecular orbital interpretations, but the results were primarily qualitative.<sup>57</sup> More recently, a DFT analog of Koopmans' theorem has been developed that formally identifies the eigenvalues with vertical ionization potentials, which can be measured with photoelectron spectroscopy.<sup>58</sup>

Despite the arguments against ascribing physical meaning to the K-S orbitals and eigenvalues, it has become fairly standard, especially for solids, to use them to calculate the density of states (DOS)<sup>59</sup> Sec. VI. B. This has been found to yield reasonable results for the valence bands in metals, but poor results for tightly bound orbitals and band gaps.<sup>24</sup> A highly technical discussion of this issue can be found in Ref.<sup>60</sup>. The density of states can be calculated by a sum over the k-points:<sup>15</sup>

$$\rho(\epsilon) = \sum_{\mathbf{k}} \omega_{\mathbf{k}} \sum_i \beta(\epsilon - \epsilon_{i\mathbf{k}}) \quad (7)$$

where  $\omega_{\mathbf{k}}$  is the weight associated with the k-point, and  $\beta$  is a broadening function, typically a gaussian function, to account for the finite number of k-points used in the calculations. The amount of broadening is arbitrary, and should tend to zero as the number of k-points approaches infinity.

---

```

1 from ase import Atoms, Atom
2 from jasp import *
3 import sys
4

```

```

5  import matplotlib.pyplot as plt
6  import numpy as np
7  from ase.dft import DOS
8
9  a = 3.9 # approximate lattice constant
10 b = a / 2.
11 bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
12              cell=[(0, b, b),
13                    (b, 0, b),
14                    (b, b, 0)])
15
16 with jasp('bulk/pd-dos',
17          encut=300,
18          xc='PBE',
19          lreal=False,
20          kpts=(8, 8, 8), # this is too low for high quality DOS
21          atoms=bulk) as calc:
22
23     # this runs the calculation
24     bulk.get_potential_energy()
25     dos = DOS(calc, width=0.2)
26     d = dos.get_dos()
27     e = dos.get_energies()
28
29 import pylab as plt
30 plt.plot(e,d)
31 plt.xlabel('energy (eV)')
32 plt.ylabel('DOS')
33 plt.savefig('images/pd-dos.png')

```

---

None

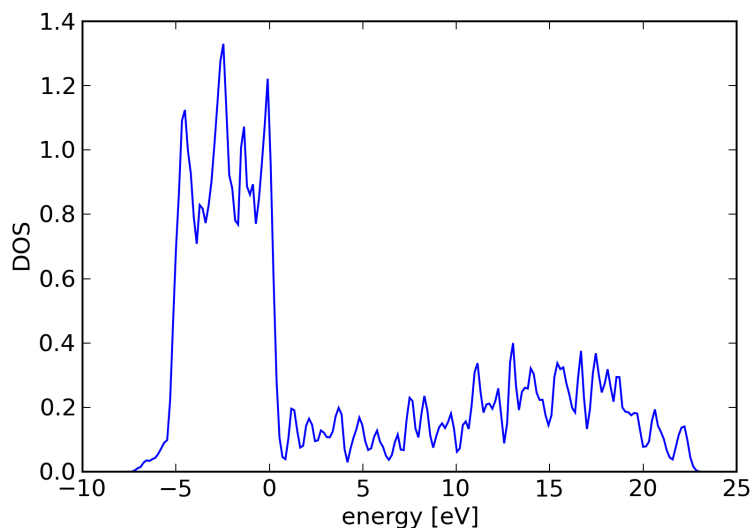


Figure 49: Total DOS for bulk Pd.

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

The VASP manual [recommends](#) a final run be made with ISMEAR=-5, which uses the tetrahedron method with Blöchl corrections.

---

```

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

```

---

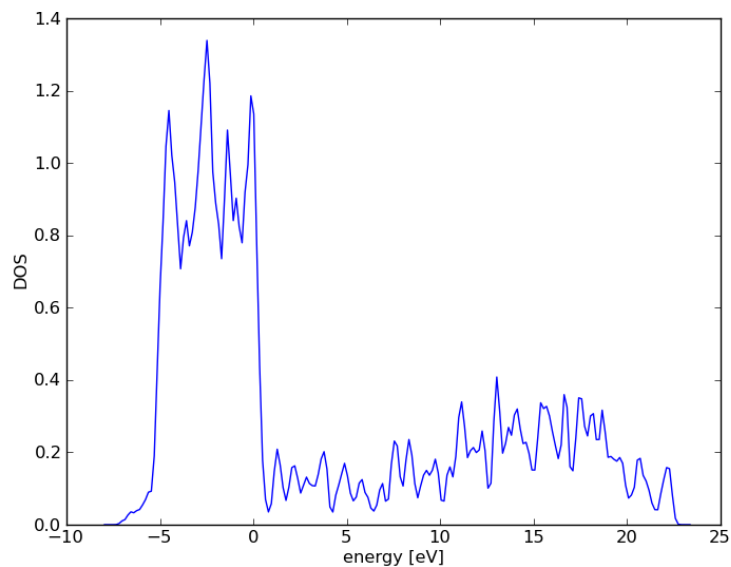


Figure 50: Total DOS for Pd computed with ISMEAR=-5

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

#### 4.10 Atom projected density of states

One major disadvantage of a planewave basis set is that it is difficult to relate the completely delocalized planewaves to localized phenomena such as bonding. Much insight into bonding has been gained by atomic/molecular orbital theory, which has carried over to the solid-state

arena.<sup>7</sup> Consequently, several schemes have been developed to project the one-electron Kohn-Sham wave functions onto atomic wave functions.<sup>61–63</sup> In VASP, the one electron wave functions can be projected onto spherical harmonic orbitals. The radial component of the atomic orbitals extends to infinity. In a solid, this means that the projection on one atom may overlap with the projection on a neighboring atom, resulting in double counting of electrons. Consequently, a cutoff radius was introduced, beyond which no contributions are included. It is not obvious what the best cutoff radius is. If the radius is too small, it might not capture all of the electrons associated with the atom. However, if it is too large, it may include electrons from neighboring atoms. One might want to use different cutoff radii for different atoms, which have different sizes. Furthermore, the ideal cutoff radius for an atom may change in different environments, thus it would require an iterative procedure to determine it. This difficulty arises because the orbital-band occupations are not observable, thus how the electrons are divided up between atoms is arbitrary and, as will be seen later, is sensitive to the cutoff radius (and in other DFT implementations, the basis set). However, Mulliken orbital populations have been used successfully for many years to examine the qualitative differences between similar systems, and that is precisely what these quantities are used for here. Thus, a discussion of the analysis and results is warranted.

The *s* and *p* states in a metal are typically delocalized in space and more like free-electrons, whereas the *d*-orbitals are fairly localized in space and have been treated successfully with tight-binding theories such as extended Hückel theory,<sup>7</sup> and linear muffin tin orbital theory.<sup>64</sup> Consequently, the remaining discussion will be focused on the properties of the projected *d*-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. By integrating the projected *d*-band up to the Fermi level, the *d*-band filling can be determined. It is not obvious what the electron count in the *d*-band should be for an atom in a metal. For a gas-phase, neutral metal atom in the ground state, however, the *d*-orbital electron count is well defined, so it will be used as an initial reference point for comparison.<sup>4</sup>

A powerful method for characterizing distributions is to examine various moments of the distribution (see Chapter 4 in Ref.<sup>65</sup> and Chapter 6 in Refs.<sup>66</sup> and<sup>67</sup>). The  $n^{th}$  order moment,  $\mu_n$ , of a distribution of states  $\rho(\epsilon)$  with respect to a reference  $\epsilon_o$  is defined by

$$\mu_n = \frac{\int_{-\infty}^{\infty} \epsilon^n \rho(\epsilon - \epsilon_o) d\epsilon}{\int_{-\infty}^{\infty} \rho(\epsilon - \epsilon_o) d\epsilon} \quad (8)$$

In this work, the reference energy is always the Fermi level. The zeroth moment is just the total number of states, in this case it will be normalized to unity. The first moment is the average energy of distribution, analogous to the center of mass for a mass density distribution. The second moment is the mean squared width of the distribution. The third moment is a measure of skewness and the fourth moment is related to kurtosis, but these moments are rarely used, and only the first and second moments are considered in this work.

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.

---

```

1  from ase import Atoms, Atom
2  from jasp import *
3  from ase.calculators.vasp import VaspDos
4  import sys
5
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  a = 3.9 # approximate lattice constant
10 b = a / 2.
11 bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
12              cell=[(0, b, b),
13                    (b, 0, b),
14                    (b, b, 0)])
15
16 with jasp('bulk/pd-ados',
17          encut=300,
18          xc='PBE',
19          lreal=False,
20          rwigs=[1.5], # wigner-seitz radii for ados
21          kpts=(8, 8, 8),
22          atoms=bulk) as calc:
23
24     # this runs the calculation
25     bulk.get_potential_energy()
26
27     # now get results
28     ados = VaspDos(efermi=calc.get_fermi_level())
29
30     energies = ados.energy
31     dos = ados.site_dos(0, 'd')
32
33     #we will select energies in the range of -10, 5
34     ind = (energies < 5) & (energies > -10)
35
36     energies = energies[ind]
37     dos = dos[ind]
38
39     Nstates = np.trapz(dos, energies)
40     occupied = energies <= 0.0
41     N_occupied_states = np.trapz(dos[occupied], energies[occupied])
42     ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
43     wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)
44
45     print 'Total # states = {0:1.2f}'.format(Nstates)
46     print 'number of occupied states = {0:1.2f}'.format(N_occupied_states)
47     print 'd-band center = {0:1.2f} eV'.format(ed)
48     print 'd-band width = {0:1.2f} eV'.format(np.sqrt(wd2))
49
50     # plot the d-band
51     plt.plot(energies, dos, label='$d$-orbitals')
52
53     # plot the occupied states in shaded gray
54     plt.fill_between(x=energies[occupied],
55                     y1=dos[occupied],
56                     y2=np.zeros(dos[occupied].shape),
57                     color='gray')
58
59     plt.xlabel('$E - E_f$ (eV)')
60     plt.ylabel('DOS (arbitrary units)')
61
62 plt.savefig('images/pd-ados.png')
63 plt.show()

```

---

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

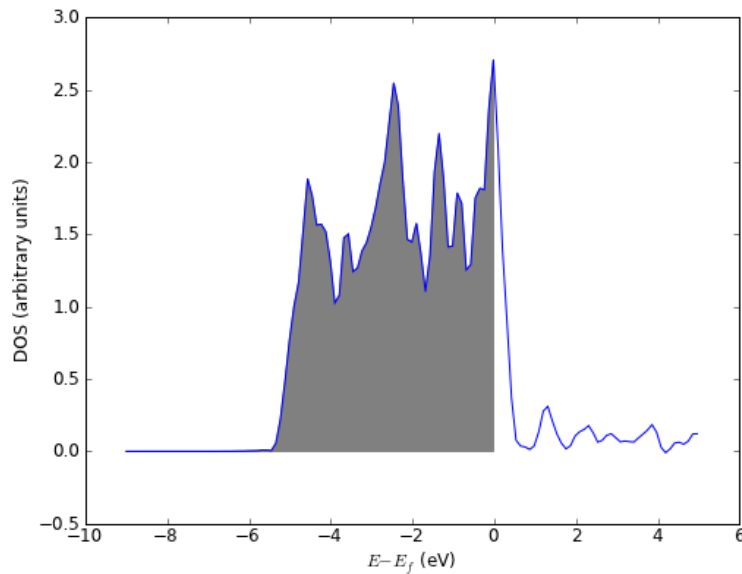


Figure 51: Atom projected *d*-band for bulk Pd. The shaded area corresponds to the occupied states below the Fermi level.

#### 4.10.1 Effect of RWIGS on ADOS

---

```

1  from ase import Atoms, Atom
2  from jasp import *
3  from ase.calculators.vasp import VaspDos
4  import sys
5
6  import matplotlib.pyplot as plt
7  import numpy as np
8
9  a = 3.9 # approximate lattice constant
10 b = a / 2.
11 bulk = Atoms([Atom('Pd', (0.0, 0.0, 0.0))],
12             cell=[(0, b, b),
13                  (b, 0, b),
14                  (b, b, 0)])
15
16 RWIGS = [1.0, 1.1, 1.25, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0 ]
17
18 ED, WD, N = [], [], []
19
20 for rwigs in RWIGS:
21     with jasp('bulk/pd-ados') as calc:
22         calc.clone('bulk/pd-ados-rwigs-{}'.format(rwigs))
23
24     with jasp('bulk/pd-ados-rwigs-{}'.format(rwigs)) as calc:
25         calc.set(rwigs=[rwigs])

```

```

26
27     try:
28         calc.calculate()
29     except (VaspSubmitted, VaspQueued):
30         continue
31
32     # now get results
33     ados = VaspDos(efermi=calc.get_fermi_level())
34
35     energies = ados.energy
36     dos = ados.site_dos(0, 'd')
37
38     #we will select energies in the range of -10, 5
39     ind = (energies < 5) & (energies > -10)
40
41     energies = energies[ind]
42     dos = dos[ind]
43
44     Nstates = np.trapz(dos, energies)
45     occupied = energies <= 0.0
46     N_occupied_states = np.trapz(dos[occupied], energies[occupied])
47     ed = np.trapz(energies * dos, energies) / np.trapz(dos, energies)
48     wd2 = np.trapz(energies**2 * dos, energies) / np.trapz(dos, energies)
49
50     N.append(N_occupied_states)
51     ED.append(ed)
52     WD.append(wd2**0.5)
53
54     plt.plot(RWIGS, N, 'bo', label='N. occupied states')
55     plt.legend(loc='best')
56     plt.xlabel('RWIGS ($\AA$)')
57     plt.ylabel('# occupied states')
58     plt.savefig('images/ados-rwigs-occupation.png')
59
60     plt.figure()
61     plt.plot(RWIGS, ED, 'bo', label='d-band center (eV)')
62     plt.plot(RWIGS, WD, 'gs', label='d-band width (eV)')
63     plt.xlabel('RWIGS ($\AA$)')
64     plt.legend(loc='best')
65     plt.savefig('images/ados-rwigs-moments.png')

```

---



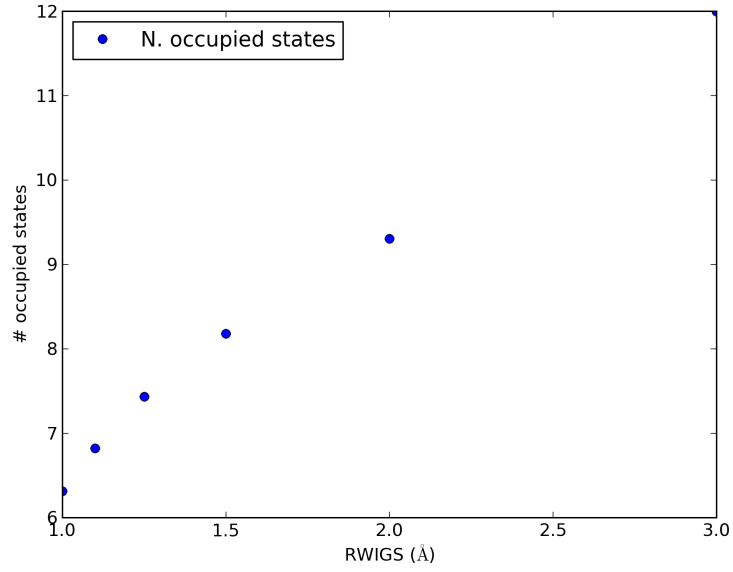


Figure 52: Effect of the RWIGS on the number of occupied  $d$ -states.

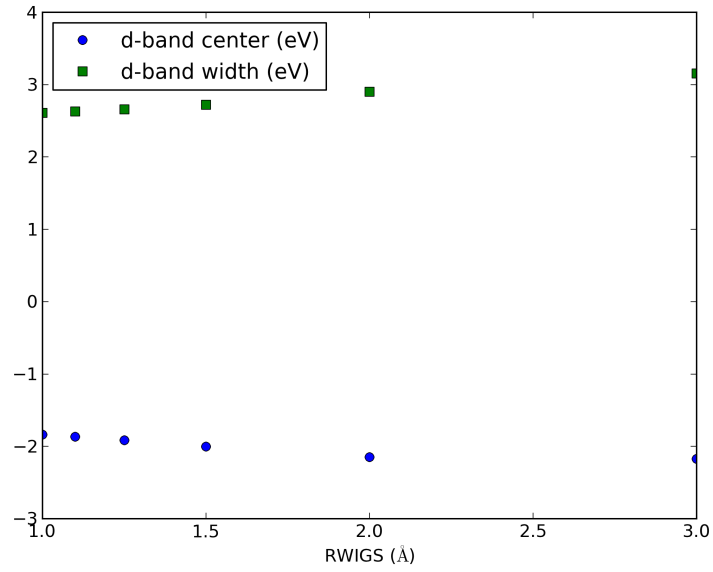


Figure 53: Effect of the RWIGS on the  $d$ -band center and width.

#### 4.11 Band structures

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

---

```

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

```

---

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

---

```

1  from jasp import *
2
3  wd = 'bulk/Si-bandstructure'
4
5  with jasp('bulk/Si-selfconsistent') as calc:
6      calc.clone(wd)
7
8  kpts = [[0.5,0.5,0.0], # L
9          [0,0,0],      # Gamma
10         [0,0,0],
11         [0.5, 0.5, 0.5]] # X
12
13  with jasp(wd,
14            kpts=kpts,
15            reciprocal=True,
16            kpts_nintersections=10,
17            icharg=11) as calc:
18      calc.calculate()

```

---

`jasp` does not have a fancy way of extracting band structures yet, so we will learn how to manually parse the EIGENVAL file here to generate the band structure.

---

```

1  f = open('bulk/Si-bandstructure/EIGENVAL', 'r')
2
3  line1 = f.readline()
4  line2 = f.readline()
5  line3 = f.readline()
6  line4 = f.readline()
7  comment = f.readline()
8  unknown, npoints, nbands = [int(x) for x in f.readline().split()]
9
10 blankline = f.readline()
11

```

---

```

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

```

---

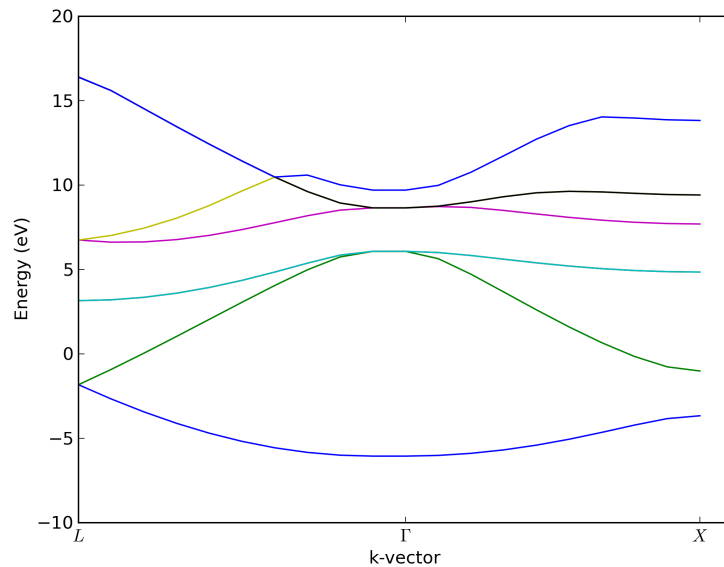


Figure 54: Calculated band-structure for Si.

#### 4.11.1 TODO create example showing band dispersion with lattice constant

### 4.12 Magnetism

#### 4.12.1 Determining if a magnetic solution is energetically favorable

We can force a total magnetic moment onto a unit cell and compute the total energy as function of the total magnetic moment. If there is a minimum in the energy, then we know there is a

lower energy magnetic solution than a non-magnetic solution.

---

```
1 from jasp import *
2 from ase.lattice.cubic import BodyCenteredCubic
3
4 atoms = BodyCenteredCubic(directions=[[1,0,0],
5                                     [0,1,0],
6                                     [0,0,1]],
7                               size=(1,1,1),
8                               symbol='Fe')
9
10 NUPDOWNS = [0.0, 2.0, 4.0, 5.0, 6.0, 8.0]
11 energies = []
12 for B in NUPDOWNS:
13     with jasp('bulk/Fe-bcc-fixedmagmom-{0:1.2f}'.format(B),
14               xc='PBE',
15               encut=300,
16               kpts=(4,4,4),
17               ispin=2,
18               nupdown=B,
19               atoms=atoms) as calc:
20         try:
21             e = atoms.get_potential_energy()
22             energies.append(e)
23         except (VaspSubmitted, VaspQueued):
24             pass
25
26 import matplotlib.pyplot as plt
27 plt.plot(NUPDOWNS, energies)
28 plt.xlabel('Total Magnetic Moment')
29 plt.ylabel('Energy (eV)')
30 plt.savefig('images/Fe-fixedmagmom.png')
```

---

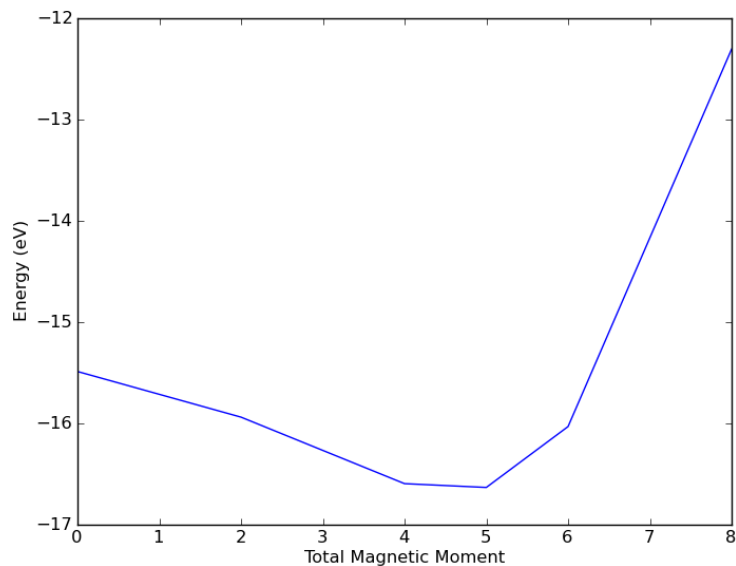


Figure 55: Total energy vs. total magnetic moment for bcc Fe.

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

---

```

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

```

---

#### 4.12.2 Antiferromagnetic spin states

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

---

```

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

```

---

```
Magnetic moments = [ 2.652 -2.652  0.    0.  ]
Total magnetic moment = 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. Hence, the sum of spins is zero, and this arrangement is called anti-ferromagnetic.

#### 4.12.3 TODO NiO-FeO formation energies with magnetism

#### 4.13 TODO phonons

68

#### 4.14 TODO solid state NEB

69 Carter paper<sup>70</sup> recent Henkelman paper

## 5 Surfaces

### 5.1 Surface structures

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

#### 5.1.1 Simple surfaces

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

---

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

---

None

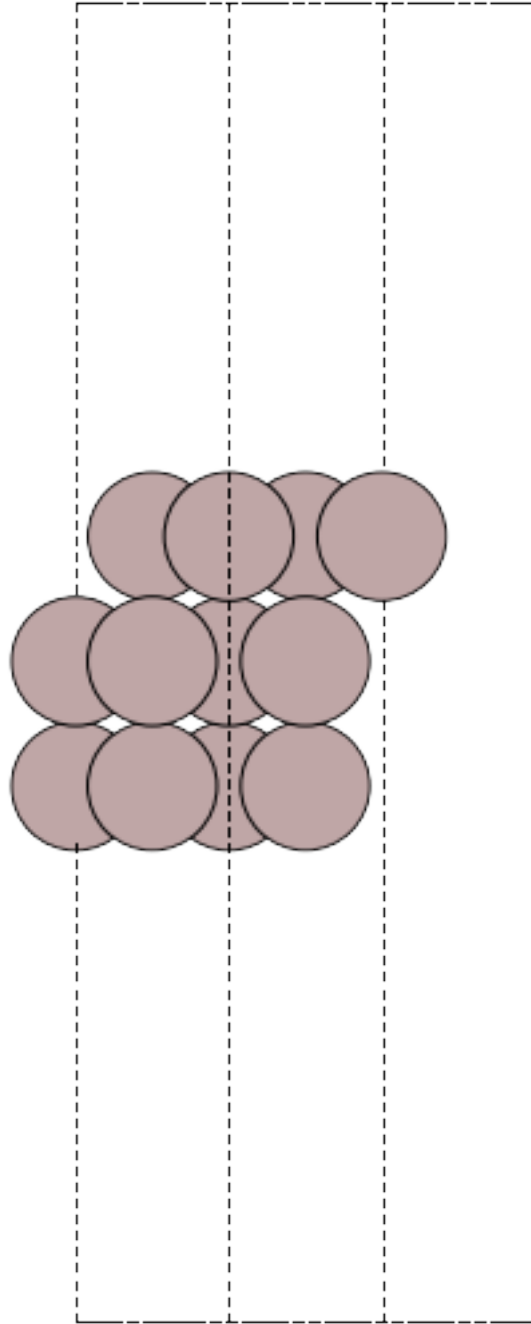


Figure 56: An Al(111) slab with three layers and 20 Å of vacuum.

### 5.1.2 Vicinal surfaces

The vast majority of surface calculations are performed on flat surfaces. This is partially because these surfaces tend to have the lowest surface energies, and thus are likely to be experimentally observed. The flat surfaces, also known as low Miller index surfaces, also have small unit cells,

which tends to make them computationally affordable. There are, however, many reasons to model the properties of surfaces that are not flat. You may be interested in the reactivity of a step edge, for example, or you may use the lower coordination of steps as a proxy for nanoparticle reactivity. Many stepped surfaces are not that difficult to make now. The main idea in generating them is described [here](#). `ase` provides a general function for making vicinal surfaces. Here is an example of a (211) surface.

---

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

---

None



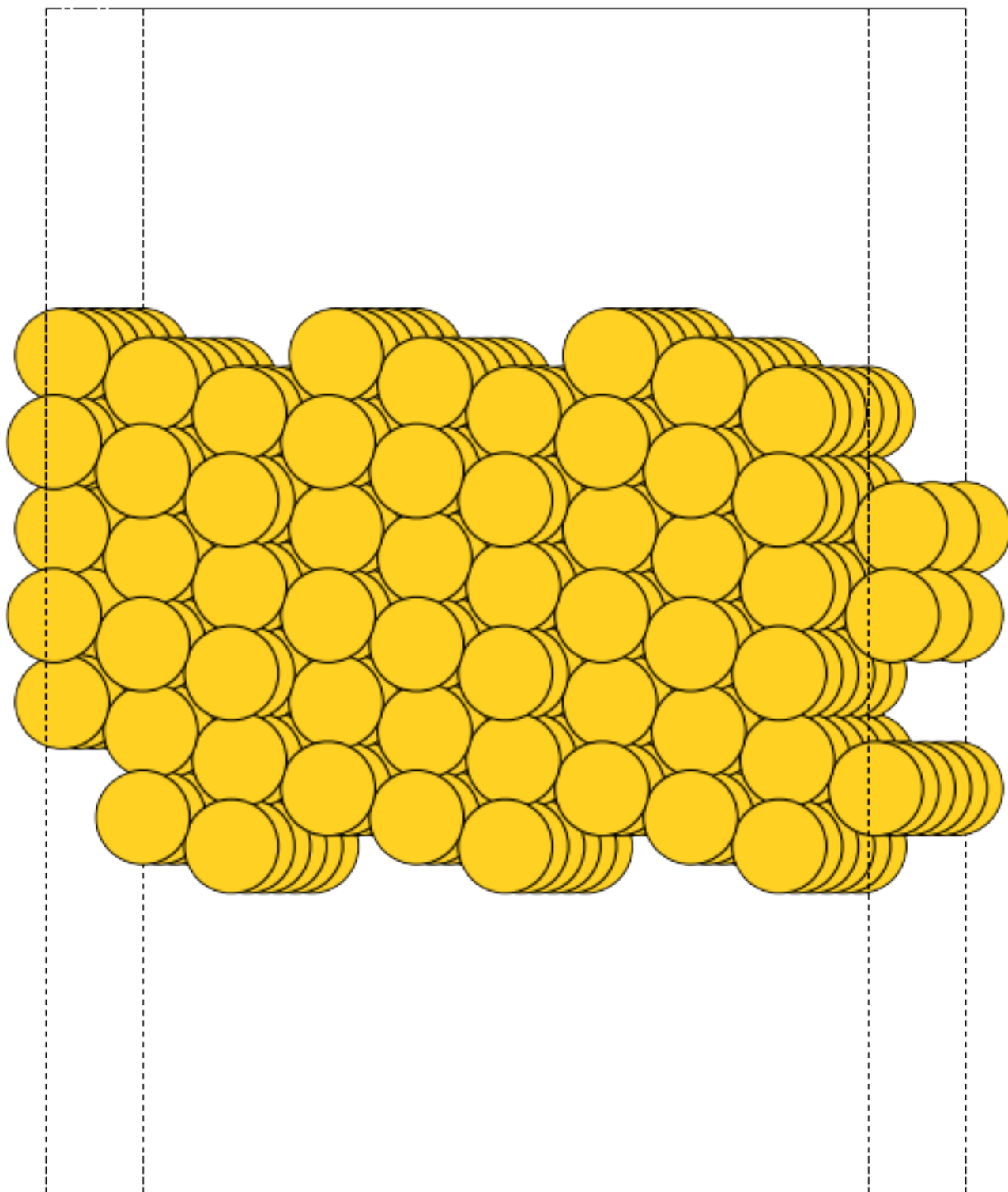


Figure 57: An Au(211) surface constructed with `ase`.

## 5.2 Surface relaxation

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

---

```
1 from jasp import *
2 from ase.lattice.surface import fcc111
3
4 atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
5
6 with jasp('surfaces/Al-slab-unrelaxed',
7           xc='PBE',
8           kpts=(6,6,1),
9           encut=350,
10          atoms=atoms) as calc:
11     atoms.get_forces()
12     print calc
```

---

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

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

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

Pseudopotentials used:

-----  
Al: potpaw\_PBE/Al/POTCAR (git-hash: c8d9ecb0b6ebec0256c5f5072cee4de6a046dac2)

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

---

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

---

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

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

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

---

```
1 from jasp import *
2 from ase.lattice.surface import fcc111
3 from ase.constraints import FixAtoms
4
5 atoms = fcc111('Al', size=(1,1,4), vacuum=10.0)
6
7 with jasp('surfaces/Al-slab-relaxed',
8         xc='PBE',
9         kpts=(6,6,1),
10        encut=350,
11        ibrion=2,
12        isif=2,
13        nsw=10,
14        atoms=atoms) as calc:
15
16    constraint = FixAtoms(mask=[atom.tag >= 3 for atom in atoms])
17    atoms.set_constraint(constraint)
18
19    atoms.get_forces()
20    print calc
```

---

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

Unit cell vectors (angstroms)

	x	y	z	length
a0	2.864	0.000	0.000	2.864
a1	1.432	2.480	0.000	2.864
a2	0.000	0.000	27.015	27.015

a,b,c,alpha,beta,gamma (deg): 2.864 2.864 27.015 90.0 90.0 90.0

Unit cell volume = 191.872 Ang<sup>3</sup>

Stress (GPa):xx, yy, zz, yz, xz, xy  
0.006 0.006 0.001 -0.000 -0.000 -0.000

Atom#	sym	position [x,y,z]	tag	rmsForce
0	Al	[0.000 0.000 10.000]	4	0.00
1	Al	[1.432 0.827 12.338]	3	0.00
2	Al	[2.864 1.653 14.637]	2	0.05
3	Al	[0.000 0.000 16.982]	1	0.02

-----  
INCAR Parameters:

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

Pseudopotentials used:

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

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

---

```
1 from jasp import *
2 from ase.lattice.surface import fcc111
```

```

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

```

---

```

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

```

### 5.3 Surface reconstruction

We previously considered how relaxation can lower the surface energy. For some surfaces, a more extreme effect can reduce the surface energy: reconstruction. In a simple surface relaxation, the basic structure of a surface is preserved. However, sometimes there is a different surface structure that may have a lower surface energy.

TODO: some citations for famous reconstructions Si- $\sqrt{7} \times \sqrt{7}$  Pt(100) hex reconstruction Au(111) herringbone reconstruction

We will consider the (110) missing row reconstruction.<sup>71</sup> For some metals, especially Pt and Au, it is energetically favorable to form the so-called missing row reconstruction where every other row in the surface is “missing”. It is favorable because it lowers the surface energy. Let us consider how we might calculate and predict that. It is straightforward to compute the energy of a (110) slab, and of a (110) slab with one row missing. However, these slabs contain different numbers of atoms, so we cannot directly compare the total energies to determine which energy is lower.

We have to consider where the missing row atoms have gone, so we can account for their energy. We will consider that they have gone into the bulk, and so we to consider the energy associated with the following transformation:

slab<sub>110</sub> → slab<sub>missingrow</sub> + bulk

Thus, if this change in energy:  $E_{\text{bulk}} + E_{\text{slab}_{\text{missingrow}}} - E_{\text{slab}_{110}}$  is negative, then the formation of the missing row is expected to be favorable.

#### 5.3.1 Au(110) missing row reconstruction

We first consider the Au(110) case, where the reconstruction is known to be favorable.

##### Clean Au(110) slab

---

```

1 from jasp import *
2 from ase.lattice.surface import fcc110
3 from ase.io import write
4 from ase.constraints import FixAtoms
5
6 atoms = fcc110('Au', size=(2,1,6), vacuum=10.0)
7 constraint = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
8 atoms.set_constraint(constraint)
9
10 write('images/Au-110.png', atoms.repeat((2,2,1)), rotation='-90x', show_unit_cell=2)

```

```

11
12 with jasp('surfaces/Au-110',
13           xc='PBE',
14           kpts=(6,6,1),
15           encut=350,
16           ibrion=2,
17           isif=2,
18           nsw=10,
19           atoms=atoms) as calc:
20     calc.calculate()

```

---

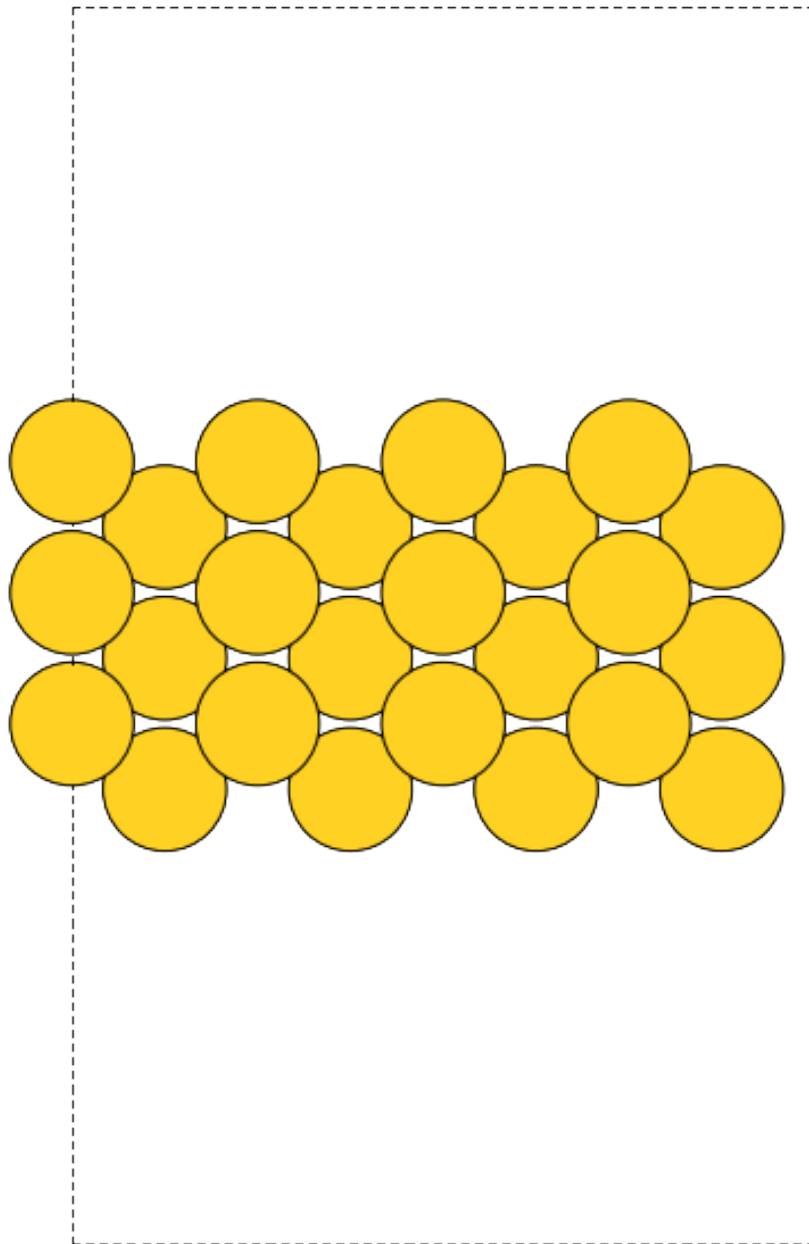


Figure 58: The unreconstructed Au(110) surface viewed from the side.

## Missing row in Au(110)

---

```
1 from jasp import *
2 from ase.lattice.surface import fcc110
3 from ase.io import write
4 from ase.constraints import FixAtoms
5
6 atoms = fcc110('Au', size=(2,1,6), vacuum=10.0)
7 del atoms[11] # delete surface row
8
9 constraint = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
10 atoms.set_constraint(constraint)
11
12 write('images/Au-110-missing-row.png', atoms.repeat((2,2,1)), rotation='-90x', show_unit_cell=2)
13
14 with jasp('surfaces/Au-110-missing-row',
15         xc='PBE',
16         kpts=(6,6,1),
17         encut=350,
18         ibrion=2,
19         isif=2,
20         nsw=10,
21         atoms=atoms) as calc:
22     calc.calculate()
```

---

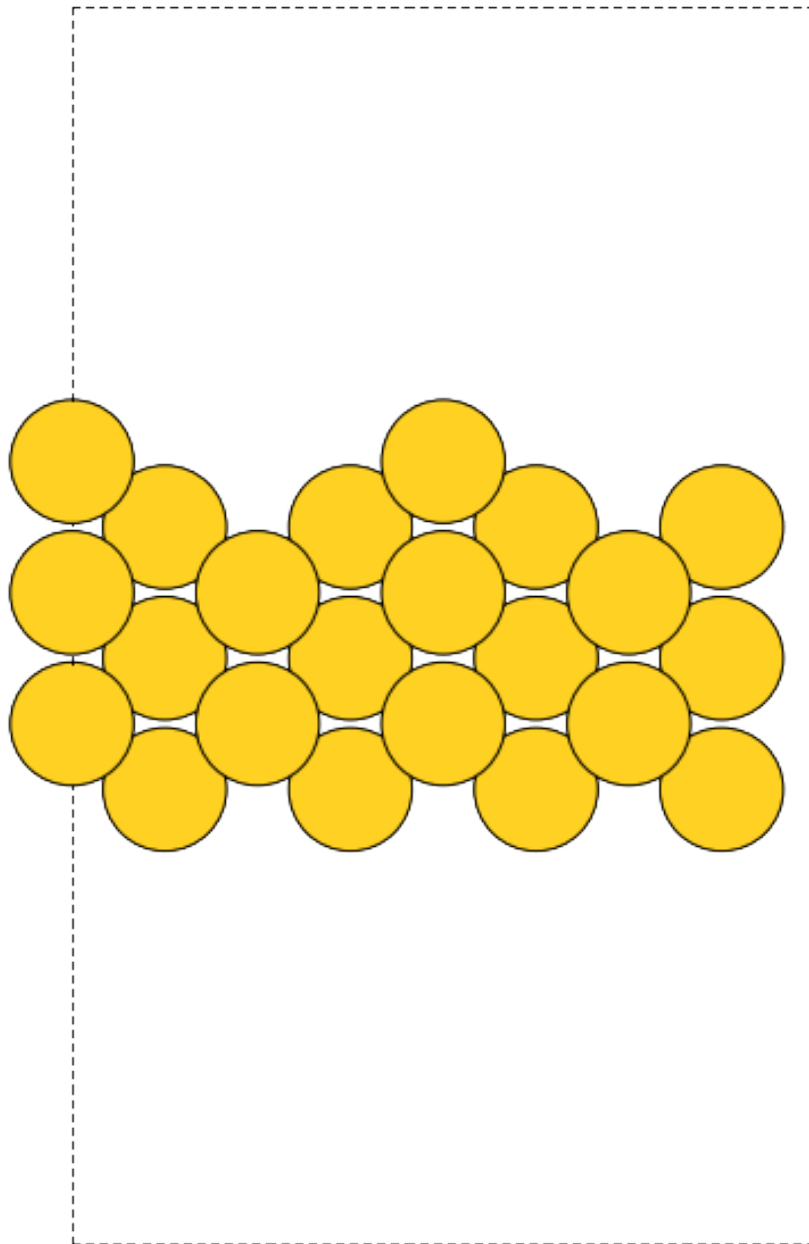


Figure 59: Au(110) with the missing row reconstruction.

## Bulk Au

---

```

1 from jasp import *
2 from ase.visualize import view
3 from ase.lattice.cubic import FaceCenteredCubic
4
5 atoms = FaceCenteredCubic(directions=[[0,1,1],
6                                     [1,0,1],
7                                     [1,1,0]],
8                               size=(1,1,1),

```



```

9                                     symbol='Au')
10
11 with jasp('bulk/Au-fcc',
12          xc='PBE',
13          encut=350,
14          kpts=(12,12,12),
15          atoms=atoms) as calc:
16     calc.calculate()

```

---

## Analysis of energies

```

1 from jasp import *
2
3 with jasp('surfaces/Au-110') as calc:
4     slab = calc.get_atoms()
5     eslab = slab.get_potential_energy()
6
7 with jasp('surfaces/Au-110-missing-row') as calc:
8     missingrow = calc.get_atoms()
9     emissingrow = missingrow.get_potential_energy()
10
11 with jasp('bulk/Au-fcc') as calc:
12     bulk = calc.get_atoms()
13     ebulk = bulk.get_potential_energy()
14
15 print 'natoms slab          = {0}'.format(len(slab))
16 print 'natoms missing row = {0}'.format(len(missingrow))
17 print 'natoms bulk         = {0}'.format(len(bulk))
18
19 print 'dE = {0:1.3f} eV'.format(emissingrow + ebulk - eslab)

```

---

```

natoms slab          = 12
natoms missing row = 11
natoms bulk         = 1
dE = -0.070 eV

```

The missing row formation energy is slightly negative. The magnitude of the formation energy is pretty small, but just slightly bigger than the typical convergence errors observed, so we should cautiously conclude that the reconstruction is favorable for Au(110). We made a lot of shortcuts in computing this quantity, including using the experimental lattice constant of Au, not checking for convergence in k-points or planewave cutoff, and not checking for convergence with respect to slab thickness or number of relaxed layers.

### 5.3.2 Ag(110) missing row reconstruction

#### Clean Ag(110) slab

```

1 from jasp import *
2 from ase.lattice.surface import fcc110
3 from ase.io import write
4 from ase.constraints import FixAtoms
5
6 atoms = fcc110('Ag', size=(2,1,6), vacuum=10.0)
7 constraint = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
8 atoms.set_constraint(constraint)
9
10 with jasp('surfaces/Ag-110',

```

```

11         xc='PBE',
12         kpts=(6,6,1),
13         encut=350,
14         ibrion=2,
15         isif=2,
16         nsw=10,
17         atoms=atoms) as calc:
18     calc.calculate()

```

---

## Missing row in Ag(110)

```

1  from jasp import *
2  from ase.lattice.surface import fcc110
3  from ase.io import write
4  from ase.constraints import FixAtoms
5
6  atoms = fcc110('Ag', size=(2,1,6), vacuum=10.0)
7  del atoms[11] # delete surface row
8
9  constraint = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
10 atoms.set_constraint(constraint)
11
12 with jasp('surfaces/Ag-110-missing-row',
13         xc='PBE',
14         kpts=(6,6,1),
15         encut=350,
16         ibrion=2,
17         isif=2,
18         nsw=10,
19         atoms=atoms) as calc:
20     calc.calculate()

```

---

## Bulk Ag

```

1  from jasp import *
2  from ase.visualize import view
3  from ase.lattice.cubic import FaceCenteredCubic
4
5  atoms = FaceCenteredCubic(directions=[[0,1,1],
6                                       [1,0,1],
7                                       [1,1,0]],
8                               size=(1,1,1),
9                               symbol='Ag')
10
11 with jasp('bulk/Ag-fcc',
12         xc='PBE',
13         encut=350,
14         kpts=(12,12,12),
15         atoms=atoms) as calc:
16     calc.calculate()

```

---

## Analysis of energies

```

1  from jasp import *
2
3  with jasp('surfaces/Ag-110') as calc:
4      slab = calc.get_atoms()
5      eslab = slab.get_potential_energy()
6

```

---

```

7  with jasp('surfaces/Ag-110-missing-row') as calc:
8      missingrow = calc.get_atoms()
9      emissingrow = missingrow.get_potential_energy()
10
11  with jasp('bulk/Ag-fcc') as calc:
12      bulk = calc.get_atoms()
13      ebulk = bulk.get_potential_energy()
14
15  print 'natoms slab          = {}'.format(len(slab))
16  print 'natoms missing row = {}'.format(len(missingrow))
17  print 'natoms bulk         = {}'.format(len(bulk))
18
19  print 'dE = {:.13f} eV'.format(emissingrow + ebulk - eslab)

```

---

```

natoms slab          = 12
natoms missing row = 11
natoms bulk         = 1
dE = -0.008 eV

```

For Ag(110), the missing row formation energy is practically thermoneutral, i.e. not that favorable. This energy is so close to 0eV, that we cannot confidently say whether the reconstruction is favorable or not. Experimentally, the reconstruction is not seen on very clean Ag(110) although it is reported that some adsorbates may induce the reconstruction.<sup>72</sup>

## 5.4 Work function

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

---

```

1  from jasp import *
2  import matplotlib.pyplot as plt
3
4  with jasp('surfaces/Al-slab-relaxed') as calc:
5      atoms = calc.get_atoms()
6
7  with jasp('surfaces/Al-slab-locpot',
8          xc='PBE',
9          kpts=(6,6,1),
10         encut=350,
11         lvtot=True, # write out local potential
12         lvhar=True, # write out only electrostatic potential, not xc pot
13         atoms=atoms) as calc:
14
15     calc.calculate()
16     ef = calc.get_fermi_level()
17
18     atoms = calc.get_atoms()
19
20     x,y,z,lp = calc.get_local_potential()
21
22     nx, ny, nz = lp.shape
23
24     axy = np.array([np.average(lp[:, :, z]) for z in range(nz)])
25     # setup the x-axis in realspace
26     uc = atoms.get_cell()
27     xaxis = np.linspace(0, uc[2][2], nz)
28
29     plt.plot(xaxis, axy)
30     plt.plot([min(xaxis), max(xaxis)], [ef, ef], 'k:')

```

```

31 plt.savefig('images/Al-wf.png')
32
33 ind = (xaxis > 0) & (xaxis < 5)
34 wf = np.average(axy[ind]) - ef
35 print ' The workfunction is {0:1.2f} eV'.format(wf)

```

---

The workfunction is 4.04 eV

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

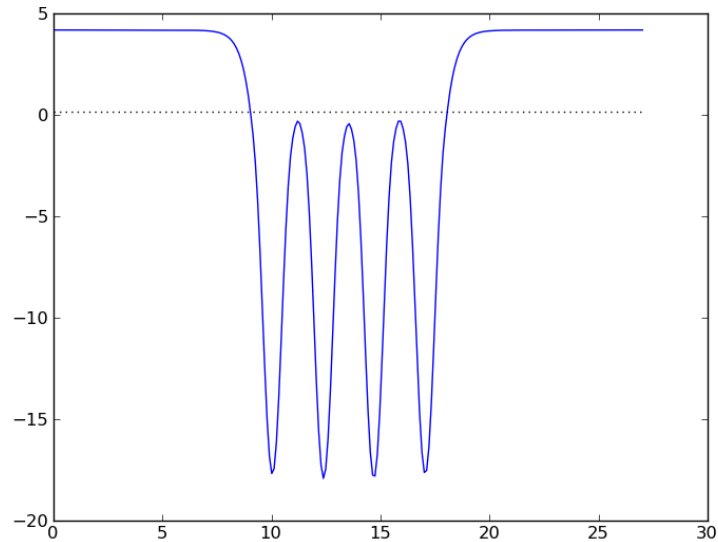


Figure 60:  $xy$  averaged local electrostatic potential of an Al(111) slab.

## 5.5 Surface energy

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

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

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  $\sigma$  converges.

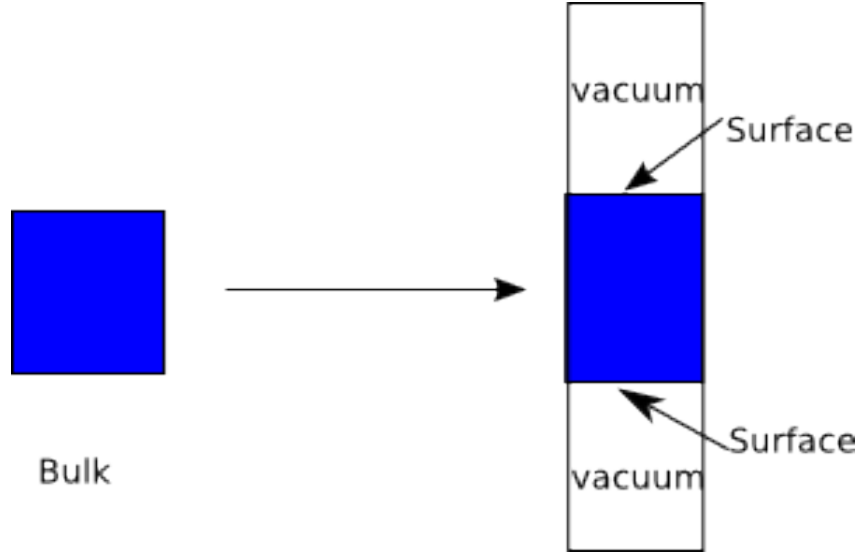


Figure 61: Schematic figure illustrating the calculation of a surface energy.

Unfortunately, this approach does not always work. The bulk system is treated subtly different than the slab system, particularly in the  $z$ -direction where the vacuum is (where typically only one  $k$ -point is used in slabs). Consequently, the  $k$ -point sampling is not equivalent in the two systems, and one can in general expect some errors due to this, with the best case being cancellation of the errors due to total  $k$ -point convergence. In the worst case, one can get a linear divergence in the surface energy with slab thickness.<sup>73</sup>

A variation of this method that usually results in better  $k$ -point error cancellation is to calculate the bulk unit cell energy using the slab unit cell with no vacuum space, with the same  $k$ -point mesh in the  $x$  and  $y$  directions, but with increased  $k$ -points in the  $z$ -direction. Thus, the bulk system and slab system have the same Brillouin zone in at least two dimensions. This maximizes the cancellation of  $k$ -point errors, but still does not guarantee convergence of the surface energy, as discussed in.<sup>73;74</sup>

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.<sup>74</sup> The method follows from equation (ref{eq:se}) where for a  $N$ -atom slab, in the limit of  $N \rightarrow \infty$ ,

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

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

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

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

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

```

1  from jasp import *
2  from ase.lattice.surface import fcc111
3  import matplotlib.pyplot as plt
4
5  Nlayers = [3, 4, 5, 6, 7, 8, 9, 10, 11]
6  energies = []
7  sigmas = []
8
9  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              encut=350,
17              kpts=(8,8,1),
18              atoms=slab) as calc:
19          calc.set_nbands(f=2) # the default nbands in VASP is too low for Al
20          try:
21              energies.append(slab.get_potential_energy())
22          except (VaspSubmitted, VaspQueued):
23              pass
24
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      print 'nlayers = {1:2d} sigma = {0:1.3f} eV/atom'.format(sigma, N)
31
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')

```

---

```

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

```

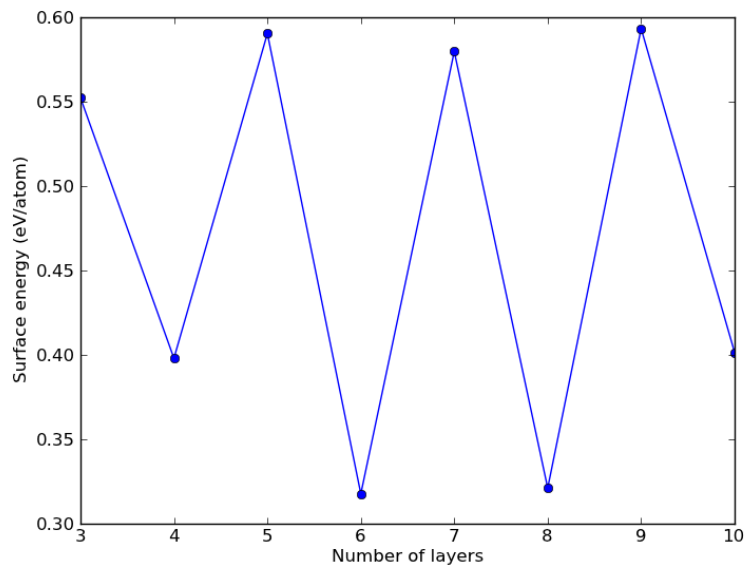


Figure 62: Surface energy of an Al(111) slab as a function of thickness.

One reason for the oscillations may be quantum size effects.<sup>75</sup> In<sup>76</sup> the surface energy of Cu(111) is reported as 0.48 eV/atom, or 1.36 J/m<sup>2</sup>. Here is an example showing a conversion between these two units. We use ase to compute the area of the unit cell from the norm of the cross-product of the vectors defining the surface unit cell.

---

```

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

```

---

sigma = 1.3628146074 J/m^2

### 5.5.1 Advanced topics in surface energy

The surface energies can be used to estimate the shapes of nanoparticles using a Wulff construction. See<sup>77</sup> for an example of computing Mo<sub>2</sub>C surface energies and particle shapes, and<sup>78</sup> 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<sup>79</sup>.

## 5.6 Dipole correction

A subtle problem can arise when an adsorbate is placed on one side of a slab with periodic boundary conditions, which is currently the common practice. The problem is that this gives the slab a dipole moment. The array of dipole moments created by the periodic boundary conditions generates an electric field that can distort the electron density of the slab and change the energy. The existence of this field in the vacuum also makes the zero-potential in the vacuum ill-defined, thus the work function is not well-defined. One solution to this problem is to use slabs with adsorbates on both sides, but then very thick (eight to ten layers) slabs must be used to ensure the adsorbates do not interact through the slab. An alternative solution, the dipole correction scheme, was developed by Neugebauer and Scheffler<sup>80</sup> and later corrected by Bengtsson.<sup>81</sup> In this technique, an external field is imposed in the vacuum region that exactly cancels the artificial field caused by the slab dipole moment. The advantage of this approach is that thinner slabs with adsorbates on only one side can be used.

There are also literature reports that the correction is small.<sup>82</sup> Nevertheless, in the literature the use of this correction is fairly standard, and it is typical to at least consider the correction.

Here we will just illustrate the effect.

### 5.6.1 Slab with no dipole correction

---

```
1  # compute local potential of slab with no dipole
2  from ase.lattice.surface import fcc111, add_adsorbate
3  from jasp import *
4  import matplotlib.pyplot as plt
5  slab = fcc111('Al', size=(2,2,2), vacuum=10.0)
6  add_adsorbate(slab, 'Na', height=1.2, position='fcc')
7
8  slab.center()
9
10 with jasp('surfaces/Al-Na-nodip',
11          xc='PBE',
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      calc.calculate()
```

---

None

### 5.6.2 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.



---

```

1  # compute local potential with dipole calculation on
2  from ase.lattice.surface import fcc111, add_adsorbate
3  from jasp import *
4
5  slab = fcc111('Al', size=(2, 2, 2), vacuum=10.0)
6  add_adsorbate(slab, 'Na', height=1.2, position='fcc')
7
8  slab.center()
9
10 with jasp('surfaces/Al-Na-dip',
11          xc='PBE',
12          encut=340,
13          kpts=(2, 2, 1),
14          idipol=3, # only along z-axis
15          lvtot=True, # write out local potential
16          lvhar=True, # write out only electrostatic potential, not xc pot
17          atoms=slab) as calc:
18
19     calc.calculate()
20
21     x, y, z, cd = calc.get_charge_density()
22     n0, n1, n2 = cd.shape
23     nelements = n0 * n1 * n2
24     voxel_volume = slab.get_volume() / nelements
25     total_electron_charge = cd.sum() * voxel_volume
26
27     electron_density_center = np.array([(cd * x).sum(),
28                                         (cd * y).sum(),
29                                         (cd * z).sum()])
30     electron_density_center *= voxel_volume
31     electron_density_center /= total_electron_charge
32
33     print 'electron-density center = {0}'.format(electron_density_center)
34     uc = slab.get_cell()
35
36     # get scaled electron charge density center
37     sedc = np.dot(np.linalg.inv(uc.T), electron_density_center.T).T
38
39     calc.clone('surfaces/Al-Na-dip-step2')
40
41 with jasp('surfaces/Al-Na-dip-step2',
42          ldipol=True, dipol=sedc) as calc:
43     calc.calculate()

```

---

### 5.6.3 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.

---

```

1  from jasp import *
2  import matplotlib.pyplot as plt
3
4  with jasp('surfaces/Al-Na-nodip') as calc:
5      atoms = calc.get_atoms()
6
7      x, y, z, lp = calc.get_local_potential()
8      nx, ny, nz = lp.shape
9
10     axy_1 = [np.average(lp[:, :, z]) for z in range(nz)]
11     # setup the x-axis in realspace
12     uc = atoms.get_cell()
13     xaxis_1 = np.linspace(0, uc[2][2], nz)

```

```

14
15     e1 = atoms.get_potential_energy()
16
17 with jasp('surfaces/Al-Na-dip-step2') as calc:
18     atoms = calc.get_atoms()
19
20     x, y, z, lp = calc.get_local_potential()
21     nx, ny, nz = lp.shape
22
23     axy_2 = [np.average(lp[:, :, z]) for z in range(nz)]
24     # setup the x-axis in realspace
25     uc = atoms.get_cell()
26     xaxis_2 = np.linspace(0, uc[2][2], nz)
27
28     ef2 = calc.get_fermi_level()
29     e2 = atoms.get_potential_energy()
30
31 print 'The difference in energy is {0} eV.'.format(e2-e1)
32
33 plt.plot(xaxis_1, axy_1, label='no dipole correction')
34 plt.plot(xaxis_2, axy_2, label='dipole correction')
35 plt.plot([min(xaxis_2), max(xaxis_2)], [ef2, ef2], 'k:', label='Fermi level')
36 plt.xlabel('z ($\text{\AA}$)')
37 plt.ylabel('xy-averaged electrostatic potential')
38 plt.legend(loc='best')
39 plt.savefig('images/dip-vs-nodip-esp.png')

```

---

The difference in energy is 0.006391 eV.

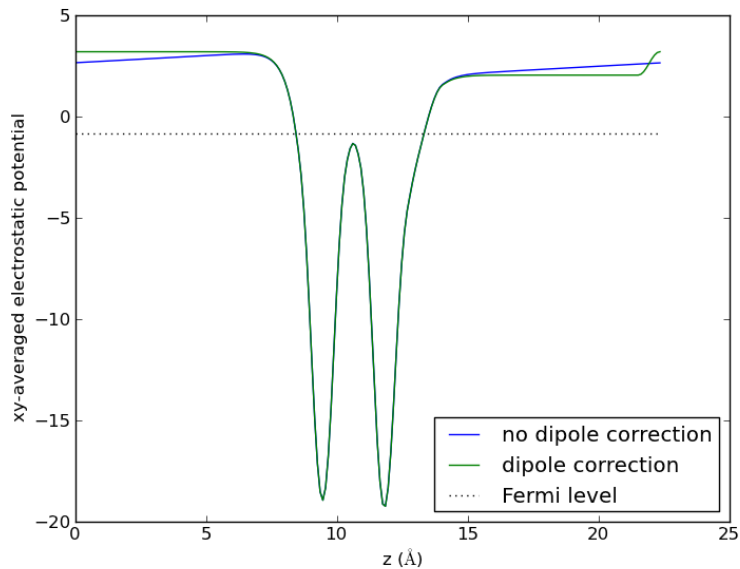


Figure 63: Comparison of the electrostatic potentials with a dipole correction and without it.

The key points to notice in this figure are:

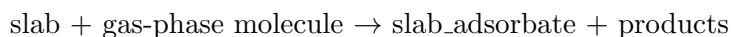
1. The two deep dips are where the atoms are.

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

## 5.7 Adsorption energies

### 5.7.1 Simple estimate of the adsorption energy

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



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<sub>2</sub> 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

---

```

1  from jasp import *
2  from ase.lattice.surface import fcc111
3  from ase.constraints import FixAtoms
4
5  atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
6  constraint = FixAtoms(mask=[True for atom in atoms])
7  atoms.set_constraint(constraint)
8
9  with jasp('surfaces/Pt-slab',
10          xc='PBE',
11          kpts=(4,4,1),
12          encut=350,
13          atoms=atoms) as calc:
14      slab_e = atoms.get_potential_energy()
```

---

### fcc site

---

```

1  from jasp import *
2
3  from ase.lattice.surface import fcc111, add_adsorbate
4  from ase.constraints import FixAtoms
5
```

---

```

6  atoms = fcc111('Pt', size=(2, 2, 3), vacuum=10.0)
7
8  # note this function only works when atoms are created by the surface module.
9  add_adsorbate(atoms, 'O', height=1.2, position='fcc')
10
11  constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
12  atoms.set_constraint(constraint)
13
14  with jasp('surfaces/Pt-slab-0-fcc',
15           xc='PBE',
16           kpts=[4, 4, 1],
17           encut=350,
18           ibrion=2,
19           nsw=25,
20           atoms=atoms) as calc:
21
22      calc.calculate()

```

---

## bridge site

```

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

```

---

## hcp site

```

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

```

```
19         atoms=atoms) as calc:
20     calc.calculate()
```

---

## Analysis of adsorption energies

---

```
1  from jasp import *
2  from ase.io import write
3
4  with jasp('surfaces/Pt-slab') as calc:
5      atoms = calc.get_atoms()
6      e_slab = atoms.get_potential_energy()
7      write('images/pt-slab.png',atoms,show_unit_cell=2)
8
9  with jasp('surfaces/Pt-slab-0-fcc') as calc:
10     atoms = calc.get_atoms()
11     e_slab_o_fcc = atoms.get_potential_energy()
12     write('images/pt-slab-fcc-o.png',atoms,show_unit_cell=2)
13
14  with jasp('surfaces/Pt-slab-0-hcp') as calc:
15     atoms = calc.get_atoms()
16     e_slab_o_hcp = atoms.get_potential_energy()
17     write('images/pt-slab-hcp-o.png',atoms,show_unit_cell=2)
18
19  with jasp('surfaces/Pt-slab-0-bridge') as calc:
20     atoms = calc.get_atoms()
21     e_slab_o_bridge = atoms.get_potential_energy()
22     write('images/pt-slab-bridge-o.png',atoms,show_unit_cell=2)
23
24  with jasp('molecules/O2-sp-triplet-350') as calc:
25     atoms = calc.get_atoms()
26     e_O2 = atoms.get_potential_energy()
27
28  Hads_fcc = e_slab_o_fcc - e_slab - 0.5*e_O2
29  Hads_hcp = e_slab_o_hcp - e_slab - 0.5*e_O2
30  Hads_bridge = e_slab_o_bridge - e_slab - 0.5*e_O2
31
32  print 'Hads (fcc)      = {0} eV/0'.format(Hads_fcc)
33  print 'Hads (hcp)      = {0} eV/0'.format(Hads_hcp)
34  print 'Hads (bridge)   = {0} eV/0'.format(Hads_bridge)
```

---

```
Hads (fcc)      = -1.0384925 eV/0
Hads (hcp)      = -0.5986145 eV/0
Hads (bridge)   = -1.0384575 eV/0
```

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 64 and hcp (Figure 65 sites, which look like we expect.

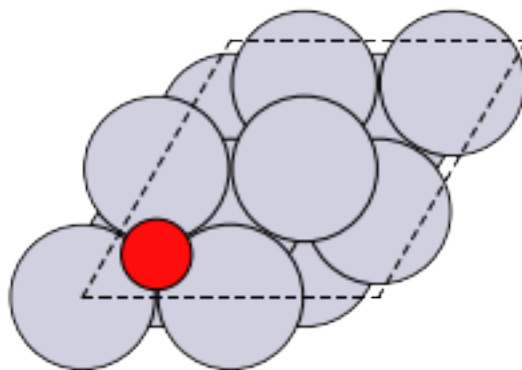


Figure 64: Final geometry of the fcc site.

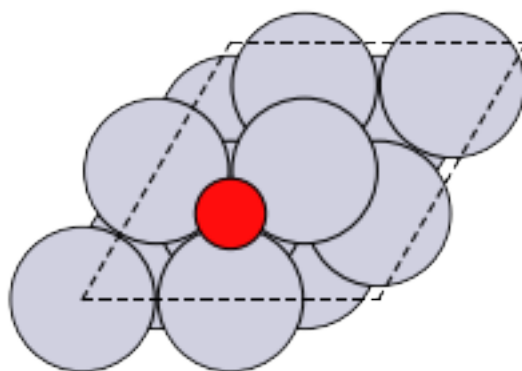


Figure 65: Final geometry of the hcp site.

The bridge site (Figure 66, however, is clearly not at a bridge site!

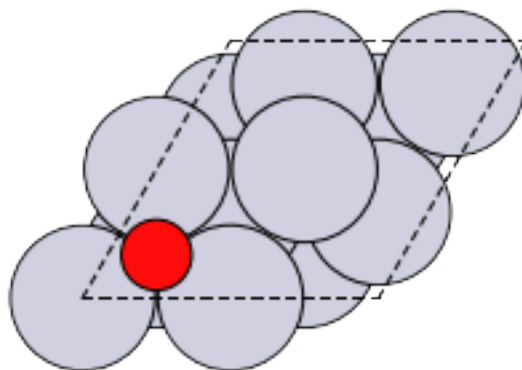


Figure 66: Final geometry of the bridge site. You can see that the oxygen atom ended up in the fcc site.

Let us see what the original geometry and final geometry for the bridge site were. The POSCAR contains the initial geometry (as long as you haven't copied CONTCAR to POSCAR), and the CONTCAR contains the final geometry.

---

```

1 from ase.io import read, write
2
3 atoms = read('surfaces/Pt-slab-0-bridge/POSCAR')
4 write('images/Pt-o-bridge-ori.png', atoms, show_unit_cell=2)
5
6 atoms = read('surfaces/Pt-slab-0-bridge/CONTCAR')
7 write('images/Pt-o-bridge-final.png', atoms, show_unit_cell=2)

```

---

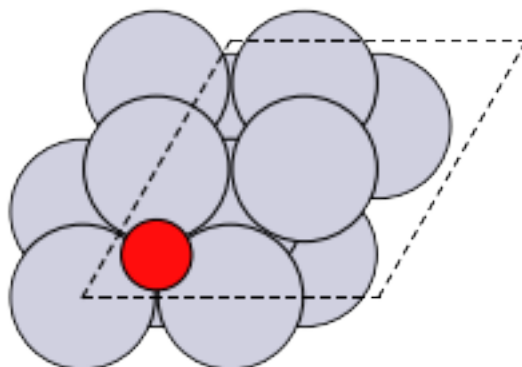


Figure 67: Initial geometry of the bridge site. It is definitely on the bridge.

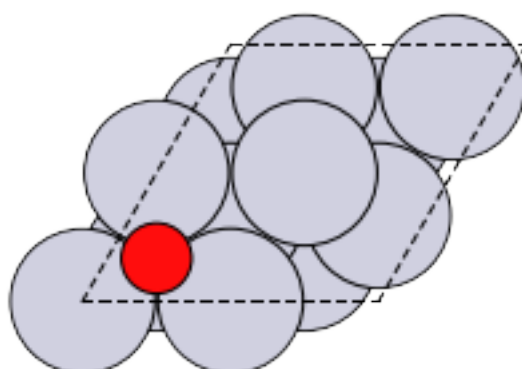


Figure 68: Final geometry of the bridge site. It has fallen into the fcc site.

You can see the problem. We should not call the adsorption energy from this calculation a bridge site adsorption energy because the O atom is actually in an fcc site! This kind of result can happen with relaxation, and you should always check that the result you get makes sense. Next, we consider how to get a bridge site adsorption energy by using constraints.

Some final notes:

1. We did not let the slabs relax in these examples, and allowing them to relax is likely to have a big effect on the adsorption energies. You have to decide how many layers to relax, and check for convergence with respect to the number of layers.
2. The slabs were pretty thin. It is typical these days to see slabs that are 4-5 or more layers thick.

3. We did not consider how well converged the calculations were with respect to  $k$ -points or [ENCUT](#).
4. We did not consider the effect of the error in  $O_2$  dissociation energy on the adsorption energies.
5. We did not consider coverage effects (see [Coverage dependence](#)).

**Adsorption on bridge site with constraints** To prevent the oxygen atom from sliding down into the fcc site, we have to constrain it so that it only moves in the  $z$ -direction. This is an artificial constraint; the bridge site is only metastable. But there are lots of reasons you might want to do this anyway. One is the bridge site is a transition state for diffusion between the fcc and hcp sites. Another is to understand the role of coordination in the adsorption energies. We use a `ase.constraints.FixScaled` constraint in ase to constrain the O atom so it can only move in the  $z$ -direction (actually so it can only move in the direction of the third unit cell vector, which only has a  $z$ -component).

---

```

1  from jasp import *
2
3  from ase.lattice.surface import fcc111, add_adsorbate
4  from ase.constraints import FixAtoms, FixScaled
5  from ase.io import write
6
7  atoms = fcc111('Pt', size=(2,2,3), vacuum=10.0)
8
9  # note this function only works when atoms are created by the surface module.
10 add_adsorbate(atoms, 'O', height=1.2, position='bridge')
11 constraint1 = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
12 # fix in xy-direction, free in z. actually, freeze movement in surface
13 # unit cell, and free along 3rd lattice vector
14 constraint2 = FixScaled(atoms.get_cell(), 12, [True, True, False])
15
16 atoms.set_constraint([constraint1, constraint2])
17 write('images/Pt-O-bridge-constrained-initial.png', atoms, show_unit_cell=2)
18 print 'Initial O position: {0}'.format(atoms.positions[-1])
19
20 with jasp('surfaces/Pt-slab-O-bridge-xy-constrained',
21          xc='PBE',
22          kpts=(4,4,1),
23          encut=350,
24          ibrion=2,
25          nsw=25,
26          atoms=atoms) as calc:
27     e_bridge = atoms.get_potential_energy()
28
29 write('images/Pt-O-bridge-constrained-final.png', atoms, show_unit_cell=2)
30 print 'Final O position : {0}'.format(atoms.positions[-1])
31
32 # now compute Hads
33 with jasp('surfaces/Pt-slab') as calc:
34     atoms = calc.get_atoms()
35     e_slab = atoms.get_potential_energy()
36
37
38 with jasp('molecules/O2-sp-triplet-350') as calc:
39     atoms = calc.get_atoms()
40     e_O2 = atoms.get_potential_energy()
41
42 Hads_bridge = e_bridge - e_slab - 0.5*e_O2

```



```

43
44 print 'Hads (bridge) = {0:1.3f} eV/0'.format(Hads_bridge)

```

---

```

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

```

You can see that only the  $z$ -position of the O atom changed. Also, the adsorption energy of O on the bridge site is **much** less favorable than on the fcc or hcp sites.

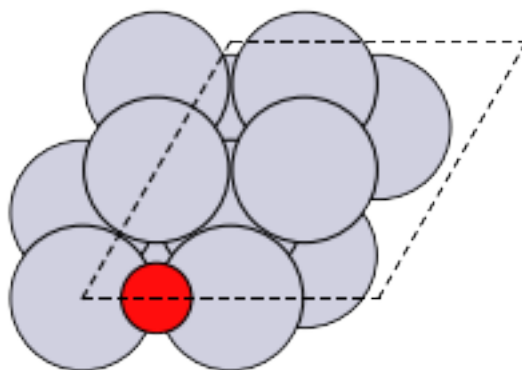


Figure 69: Initial state of the O atom on the bridge site.

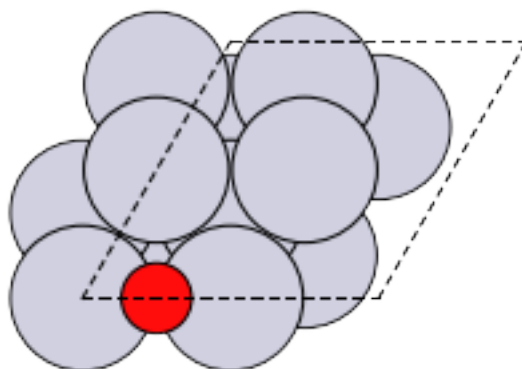


Figure 70: Final state of the constrained O atom, still on the bridge site.

### 5.7.2 Coverage dependence

The adsorbates on the surface can interact with each other which results in coverage dependent adsorption energies.<sup>83</sup> Coverage dependence is not difficult to model; we simply compute adsorption energies in different size unit cells, and/or with different adsorbate configurations. Here we consider dissociative oxygen adsorption at 1ML on Pt(111) in an fcc site, which is one oxygen atom in a  $1 \times 1$  unit cell.

For additional reading, see these references from our work:

- Correlations of coverage dependence of oxygen adsorption on different metals<sup>84;85</sup>

- Coverage effects of atomic adsorbates on Pd(111)<sup>86</sup>
- Simple model for estimating coverage dependence<sup>83</sup>
- Coverage effects on alloys<sup>87</sup>

## clean slab calculation

---

```

1 from jasp import *
2 from ase.lattice.surface import fcc111
3 from ase.constraints import FixAtoms
4
5 atoms = fcc111('Pt', size=(1, 1, 3), vacuum=10.0)
6 constraint = FixAtoms(mask=[True for atom in atoms])
7 atoms.set_constraint(constraint)
8
9 with jasp('surfaces/Pt-slab-1x1',
10         xc='PBE',
11         kpts=(8, 8, 1),
12         encut=350,
13         atoms=atoms) as calc:
14     slab_e = atoms.get_potential_energy()

```

---

## fcc site at 1 ML coverage

---

```

1 from jasp import *
2
3 from ase.lattice.surface import fcc111, add_adsorbate
4 from ase.constraints import FixAtoms
5
6 atoms = fcc111('Pt', size=(1, 1, 3), vacuum=10.0)
7
8 # note this function only works when atoms are created by the surface module.
9 add_adsorbate(atoms, 'O', height=1.2, position='fcc')
10
11 constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
12 atoms.set_constraint(constraint)
13
14 with jasp('surfaces/Pt-slab-1x1-0-fcc',
15         xc='PBE',
16         kpts=(8, 8, 1),
17         encut=350,
18         ibrion=2,
19         nsw=25,
20         atoms=atoms) as calc:
21     calc.calculate()

```

---

## Adsorption energy at 1ML

---

```

1 from jasp import *
2
3 with jasp('surfaces/Pt-slab-1x1-0-fcc') as calc:
4     atoms = calc.get_atoms()
5     e_slab_o = atoms.get_potential_energy()
6
7 # clean slab
8 with jasp('surfaces/Pt-slab-1x1') as calc:
9     atoms = calc.get_atoms()
10    e_slab = atoms.get_potential_energy()

```

---

```

11
12 with jasp('molecules/O2-sp-triplet-350') as calc:
13     atoms = calc.get_atoms()
14     e_O2 = atoms.get_potential_energy()
15
16 hads = e_slab_o - e_slab - 0.5*e_O2
17 print 'Hads (1ML) = {0:1.3f} eV'.format(hads)

```

---

Hads (1ML) = -0.099 eV

The adsorption energy is **much** less favorable at 1ML coverage than at 0.25 ML coverage! We will return what this means in [Atomistic thermodynamics effect on adsorption](#).

### 5.7.3 Effect of adsorption on the surface energy

There is a small point to make here about what adsorption does to surface energies. Let us define a general surface formation energy scheme like this:

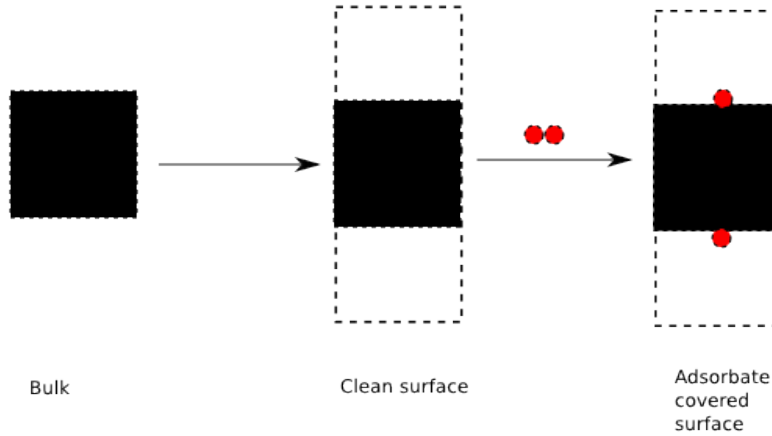


Figure 71: Schematic of forming a surface with adsorbates. First we form two clean surfaces by cleaving the bulk, then allow adsorption to occur on the surfaces.

Let us presume the surfaces are symmetric, and that each surface contributes half of the energy change. The overall change in energy:

$$\Delta E = E_{slab,ads} - E_{ads} - E_{bulk}$$

where the the energies are appropriately normalized for the stoichiometry. Let us rearrange the terms, and add and subtract a constant term  $E_{slab}$ .

$$\Delta E = E_{slab,ads} - E_{slab} - E_{ads} - E_{bulk} + E_{slab}$$

We defined  $\gamma_{clean} = \frac{1}{2A}(E_{slab} - E_{bulk})$ , and we defined  $H_{ads} = E_{slab,ads} - E_{slab} - E_{ads}$  for adsorption on a single side of a slab. In this case, there are adsorbates on both sides of the slab, so  $E_{slab,ads} - E_{slab} - E_{ads} = 2\Delta H_{ads}$ . If we normalize by  $2A$ , the area for both sides of the slab, we get

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

You can see here that the adsorption energy serves to stabilize, or reduce the surface energy, provided that the adsorption energy is negative.

Some final notes about the equations above:

- We were not careful about stoichiometry. As written, it is assumed there are the same number of atoms (not including the adsorbates) in the slabs and bulk, and the same number of adsorbate atoms in the slab and  $E_{ads}$ . Appropriate normalization factors must be included if that is not true.
- It is not necessary to perform a symmetric slab calculation to determine the effect of adsorption on the surface energy! You can examine  $\gamma - \gamma_{clean}$  with knowledge of only the adsorption energies!

## 5.8 Adsorbate vibrations

Adsorbates also have vibrational modes. Unlike a free molecule, the translational and rotational modes of an adsorbate may actually have real frequencies. Sometimes they are called frustrated translations or rotations. For metal surfaces with adsorbates, it is common to only compute vibrational modes of the adsorbate on a frozen metal slab. The rationale is that the metal atoms are so much heavier than the adsorbate that there will be little coupling between the surface and adsorbates. You can limit the number of modes calculated with constraints (`ase.constraints.FixAtoms` or `ase.constraints.FixScaled`) if you use `IBRION=5`. The other `IBRION` settings (6, 7, 8) do not respect the selective dynamics constraints. Below we consider the vibrational modes of an oxygen atom in an fcc site on Pt(111).

---

```

1  from jasp import *
2
3  with jasp('surfaces/Pt-slab-0-fcc') as calc:
4      calc.clone('surfaces/Pt-slab-0-fcc-vib')
5
6  with jasp('surfaces/Pt-slab-0-fcc-vib') as calc:
7      calc.set(ibrion=5, # finite differences with selective dynamics
8              nfree=2, # central differences (default)
9              potim=0.015, # default as well
10             ediff=1e-8,
11             nsw=1)
12     atoms = calc.get_atoms()
13     f,v = calc.get_vibrational_modes(0)
14
15 from ase.units import meV
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)
22 print 'vibrational freq  = {0} cm^(-1)'.format(f/(h*c))

```

---

```

vibrational energy = 0.061606647 eV
vibrational energy = 61.606647 meV
vibrational freq   = 1.48964216204e+13 1/s
vibrational freq   = 496.547387346 cm^(-1)

```

There are three modes for the free oxygen atom. One of them is a mode normal to the surface (the one with highest frequency). The other two are called frustrated translations. Note that we did not include the surface Pt atoms in the calculation, and this will have an effect on the result because the O atom could be coupled to the surface modes. It is typical to neglect this coupling because of the large difference in mass between O and Pt. Next we look at the difference in results when we calculate all the modes.

---

```

1 from jasp import *
2
3 with jasp('surfaces/Pt-slab-0-fcc') as calc:
4     calc.clone('Pt-slab-0-fcc-vib-ibrion=6')
5
6 with jasp('surfaces/Pt-slab-0-fcc-vib-ibrion=6') as calc:
7     calc.set(ibrion=6, # finite differences with symmetry
8             nfree=2, # central differences (default)
9             potim=0.015, # default as well
10            ediff=1e-8,
11            nsw=1)
12     atoms = calc.get_atoms()
13
14     f,m = calc.get_vibrational_modes(0)
15     allfreq = calc.get_vibrational_modes()[0]
16
17 from ase.units import meV
18 c = 3e10 # cm/s
19 h = 4.135667516e-15 # eV*s
20
21 print 'For mode 0:'
22 print 'vibrational energy = {0} eV'.format(f)
23 print 'vibrational energy = {0} meV'.format(f/meV)
24 print 'vibrational freq  = {0} 1/s'.format(f/h)
25 print 'vibrational freq  = {0} cm-1'.format(f/(h*c))

```

---

```

For mode 0:
vibrational energy = 0.063537929 eV
vibrational energy = 63.537929 meV
vibrational freq  = 1.53634035507e+13 1/s
vibrational freq  = 512.113451691 cm-1

```

Note that now there are 39 modes, which is  $3*N$  where  $N=13$  atoms in the unit cell. Many of the modes are low in frequency, which correspond to slab modes that are essentially phonons. The O frequencies are not that different from the previous calculation (497 vs 512  $\text{cm}^{-1}$ ). This is why it is common to keep the slab atoms frozen.

Calculating these results took  $39*2$  finite differences. It took about a day to get these results on a single CPU. It pays to use constraints to minimize the number of these calculations.

### 5.8.1 TODO Vibrations of the bridge site

Here we consider the vibrations of an O atom in a bridge site, which we saw earlier is a metastable saddle point.

---

```

1 from jasp import *
2 from ase.constraints import FixAtoms
3
4 # clone calculation so we do not overwrite previous results

```

---

---

```

5  with jasp('surfaces/Pt-slab-0-bridge-xy-constrained') as calc:
6      calc.clone('surfaces/Pt-slab-0-bridge-vib')
7
8  with jasp('surfaces/Pt-slab-0-bridge-vib') as calc:
9      calc.set(ibrion=5, # finite differences with selective dynamics
10             nfree=2, # central differences (default)
11             potim=0.015, # default as well
12             ediff=1e-8,
13             nsw=1)
14      atoms = calc.get_atoms()
15      del atoms.constraints
16      constraint = FixAtoms(mask=[atom.symbol != 'O' for atom in atoms])
17      atoms.set_constraint([constraint])
18
19      f,v = calc.get_vibrational_modes(2)
20      print calc.get_vibrational_modes()[0]
21
22  from ase.units import meV
23  c = 3e10 # cm/s
24  h = 4.135667516e-15 # eV*s
25
26  print 'vibrational energy = {0} eV'.format(f)
27  print 'vibrational energy = {0} meV'.format(f/meV)
28  print 'vibrational freq  = {0} 1/s'.format(f/h)
29  print 'vibrational freq  = {0} cm-1'.format(f/(h*c))

```

---

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.

## 5.9 Surface Diffusion barrier

See this review<sup>88</sup> of diffusion on transition metal surfaces.

### 5.9.1 Standard nudged elastic band method

Here we illustrate a standard NEB method. You need an initial and final state to start with. We will use the results from previous calculations of oxygen atoms in an fcc and hcp site. then we will construct a band of images connecting these two sites. Finally, we let VASP optimize the band and analyze the results to get the barrier.

---

```

1  from jasp import *
2  from ase.neb import NEB
3  import matplotlib.pyplot as plt
4  from scipy import interpolate
5  import numpy as np
6
7  with jasp('surfaces/Pt-slab-0-fcc') as calc:
8      initial_atoms = calc.get_atoms()
9
10  with jasp('surfaces/Pt-slab-0-hcp') as calc:
11      final_atoms = calc.get_atoms()
12
13  # here is our estimated transition state. we use vector geometry to
14  # define the bridge position, and add 1.451 Ang to z based on our
15  # previous bridge calculation. The bridge position is half way between
16  # atoms 9 and 10.
17  ts = initial_atoms.copy()

```

```

18  ts.positions[-1] = 0.5*(ts.positions[9] + ts.positions[10]) + [0,0,1.451]
19
20  # construct the band
21  images = [initial_atoms]
22  images += [initial_atoms.copy()]
23  images += [ts.copy()] # this is the TS
24
25  neb = NEB(images)
26  # Interpolate linearly the positions of these images:
27  neb.interpolate()
28
29  # now add the second half
30  images2 = [ts.copy()]
31  images2 += [ts.copy()]
32  images2 += [final_atoms]
33
34  neb2 = NEB(images2)
35  neb2.interpolate()
36
37  # collect final band. Note we do not repeat the TS in the second half
38  final_images = images + images2[1:]
39
40  with jasp('surfaces/Pt-0-fcc-hcp-neb',
41          ibrion=1,
42          nsw=90,
43          spring=-5,
44          atoms=final_images) as calc:
45
46      try:
47          images, energies = calc.get_neb()
48          p = calc.plot_neb(show=False)
49
50          # remember you are in surfaces/Pt-0-fcc-hcp-neb, so to save in
51          # the images directory you need ../../ in the path to get you
52          # back up.
53          plt.savefig('../../images/pt-o-fcc-hcp-neb.png')
54      except (VaspSubmitted, VaspQueued):
55          pass

```

---

Optimization terminated successfully.  
 Current function value: -0.575224  
 Iterations: 12  
 Function evaluations: 24

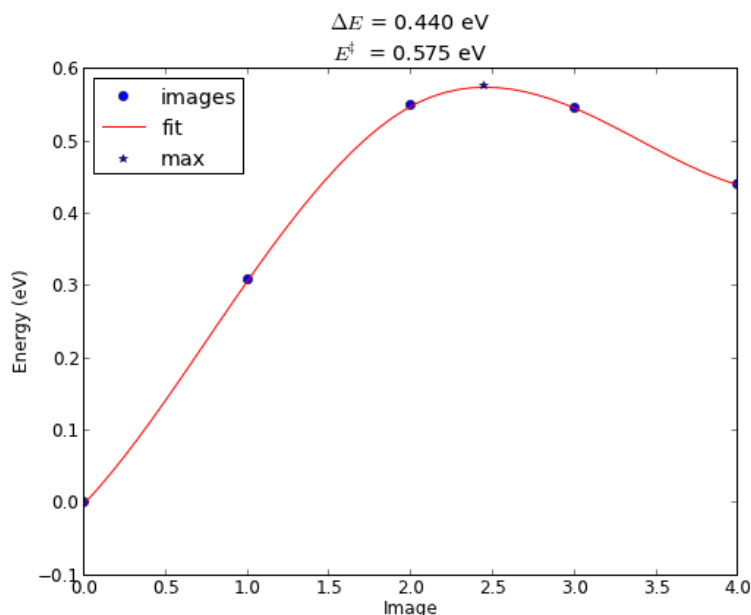


Figure 72: Energy pathway for O diffusion from an fcc to hcp site with a spline fit to determine the barrier.

We should compare this barrier to what we could estimate from the simple adsorption energies in the fcc and bridge sites. The adsorption energy in the fcc site was -1.04 eV, and in the bridge site was -0.49 eV. The difference between these two is 0.55 eV, which is very close to the calculated barrier from the NEB calculation. In cases where you can determine what the transition state is, e.g. by symmetry, or other means, it is much faster to directly compute the energy of the initial and transition states for barrier determinations. This is not usually possible though.

### 5.9.2 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.

---

```

1  # perform a climbing image NEB calculation
2  from jasp import *
3  with jasp('surfaces/Pt-O-fcc-hcp-neb') as calc:
4      calc.clone('surfaces/Pt-O-fcc-hcp-cineb')
5
6  with jasp('surfaces/Pt-O-fcc-hcp-cineb') as calc:
7      calc.set(ichain=0, lclimb=True)
8
9      images, energies = calc.get_neb(npi=4)
10     calc.plot_neb(show=False)
11 import matplotlib.pyplot as plt

```



```

12 plt.savefig('images/pt-o-cineb.svg')
13 plt.show()

```

---

Optimization terminated successfully.

Current function value: -0.575288

Iterations: 12

Function evaluations: 24

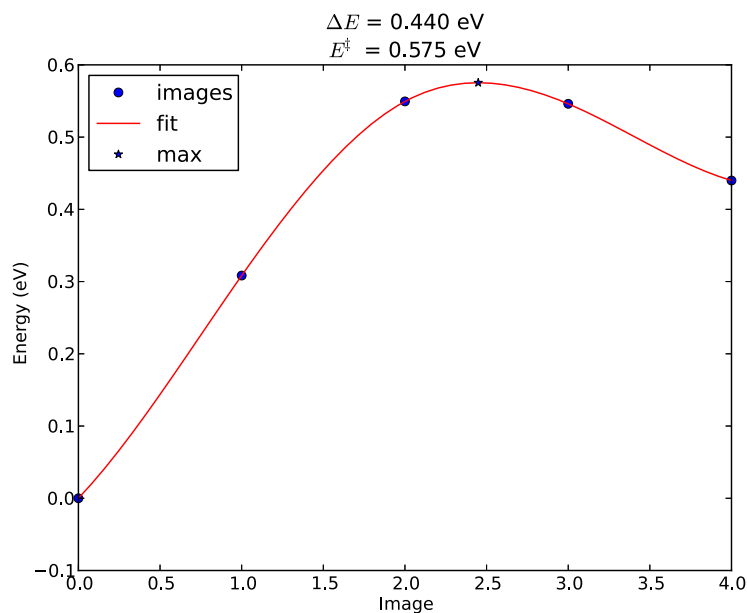


Figure 73: Climbing image NEB.

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, or the top is within the tolerance of the top two points. More band points would probably clarify that.

### 5.9.3 Using vibrations to confirm a transition state

A transition state should have exactly one imaginary degree of freedom which corresponds to the mode that takes reactants to products. See [Vibrations of the bridge site](#) for an example.

## 5.10 TODO Diffusion rates with transition state theory

### 5.11 TODO Effects of electric fields on adsorbates

Electric fields can polarize adsorbates, i.e. shift their electron density around, which changes their stability. We can model this to some extent in VASP. We have to set the [EFIELD](#) parameter to specify the field strength (in V/Å), and [LDIPOL](#) to `True`, and finally specify which lattice vector the field should be applied to with [IDIPOL](#). Usually, [IDIPOL](#) will be set to 3 for surfaces so it points in the direction of the third lattice vector, which is conventionally in the z-direction and normal to the surface.

We will illustrate the effect using an example from<sup>89</sup> for the effect of electric field on the adsorption energy of a CO molecule on Pt(111) at 0.25 ML. For simplicity, we will use a frozen slab, with the CO molecule in the geometry reported in that reference.

---

```

1  from jasp import *
2  from ase.lattice.surface import fcc111
3
4  atoms = fcc111('Pt', size=(2, 2, 4), vacuum=10.0, a=3.986)
5  atoms.append(Atom('C', [atoms[12].x, atoms[12].y, atoms[12].z + 1.851]))
6  atoms.append(Atom('O', [atoms[-1].x, atoms[-1].y, atoms[-1].z + 1.157]))
7
8  for field in [-0.5, 0.0, 0.5]:
9      with jasp('surfaces/Pt-co-field-{0}'.format(field),
10              xc='PBE',
11              encut=350,
12              kpts=(6,6,1),
13              efield=field, # set the field
14              ldipol=True, # turn dipole correction on
15              idipol=3,    # set field in z-direction
16              atoms=atoms) as calc:
17          try:
18              print '{0}: {1:1.3f}'.format(field, atoms.get_potential_energy())
19          except (VaspSubmitted, VaspQueued):
20              pass

```

---

0.0: -108.077

## 5.12 TODO Simulating STM images

[http://cst-www.nrl.navy.mil/users/sullivan/stm\\_backup/stm2.html](http://cst-www.nrl.navy.mil/users/sullivan/stm_backup/stm2.html)

## 6 Atomistic thermodynamics

Let us consider how much the Gibbs free energy of an O<sub>2</sub> 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<sub>2</sub>.

---

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from ase.units import *
4  K = 1. #not defined in ase.units!
5
6  # Shomate parameters
7  A = 31.32234; B = -20.23531; C = 57.86644
8  D = -36.50624; E = -0.007374; F = -8.903471
9  G = 246.7945; H = 0.0
10
11 def entropy(T):
12     '''entropy returned as eV/K
13     T in K
14     '''
15     t = T/1000.
16     s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
17     return s*J/mol/K
18
19 def enthalpy(T):
20     '''H - H(298.15) returned as eV/molecule'''
21     t = T/1000.

```

```

22     h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
23     return h*kJ/mol
24
25 T = np.linspace(100,700)
26
27 G = enthalpy(T) - T*entropy(T)
28
29 plt.plot(T,G)
30 plt.xlabel('Temperature (K)')
31 plt.ylabel('$\Delta G^\circ$ (eV)')
32 plt.savefig('images/O2-mu.png')

```

---

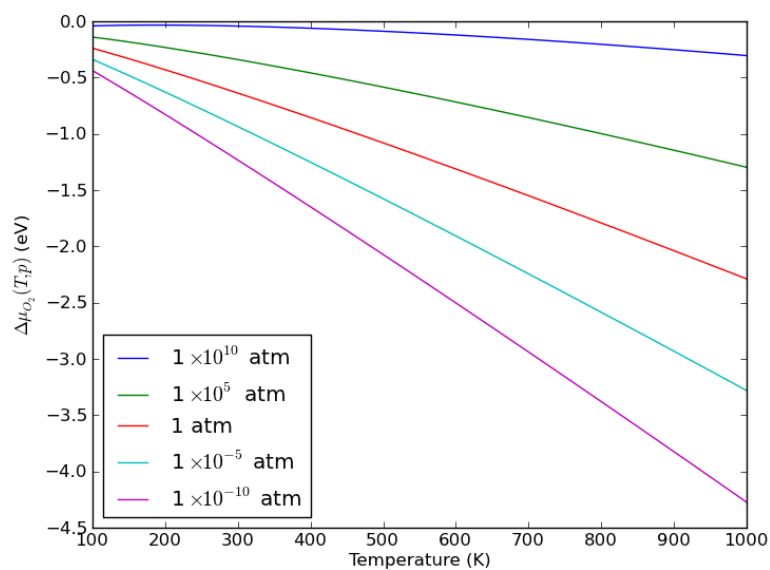


Figure 74: Effect of temperature on the Gibbs free energy of an O<sub>2</sub> molecule at 1 atm.

This is clearly a big effect! Between 500-600K, the energy has dropped by nearly 1 eV.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  from ase.units import *
4
5  atm = 101325*Pascal #atm is not defined in units
6  K = 1
7
8  # examine range over 10^-10 to 10^10 atm
9  P = np.logspace(-10,10)*atm
10
11 plt.semilogx(P/atm, kB*(300*K)*np.log(P/(1*atm)),label='300K')
12 plt.semilogx(P/atm, kB*(600*K)*np.log(P/(1*atm)),label='600K')
13 plt.xlabel('Pressure (atm)')
14 plt.ylabel('$\Delta G$ (eV)')
15 plt.legend(loc='best')
16 plt.savefig('images/O2-g-p.png')

```

---

None

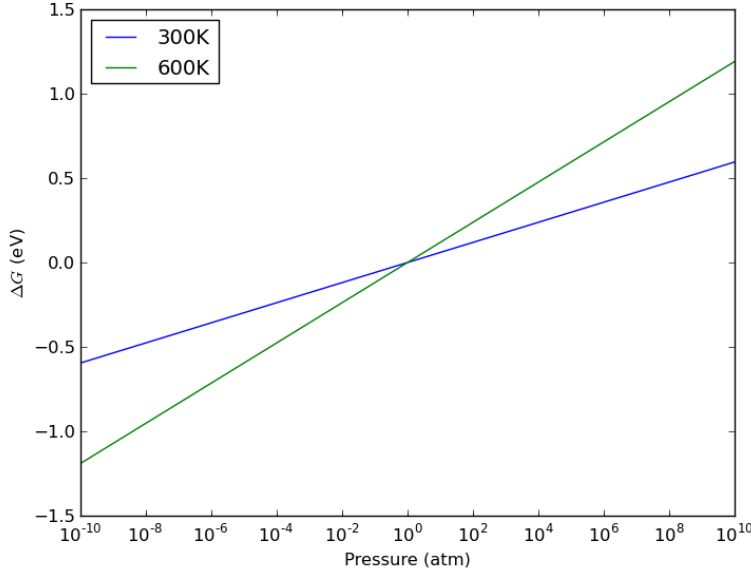


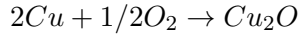
Figure 75: Effects of pressure on the ideal gas Gibbs free energy of O<sub>2</sub>.

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:

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

## 6.1 Bulk phase stability of oxides

We will consider the effects of oxygen pressure and temperature on the formation energy of Ag<sub>2</sub>O and Cu<sub>2</sub>O.



In atomistic thermodynamics, we define the free energy of formation as:

$$G_f = G_{Cu_2O} - 2G_{Cu} - 0.5G_{O_2}$$

We will at this point assume that the solids are incompressible so that  $p\Delta V \approx 0$ , and that  $S_{Cu_2O} - 2S_{Cu} \approx 0$ , which leads to  $G_{Cu_2O} - 2G_{Cu} \approx E_{Cu_2O} - 2E_{Cu}$ , which we directly compute from DFT. We express  $G_{O_2} = \mu_{O_2} = E_{O_2}^{DFT} + E_{O_2}^{ZPE} + \Delta\mu(T) + kT \ln(P/P_0)$ . In this example we neglect the zero-point energy of the oxygen molecule, and finally arrive at:

$$G_f \approx E_{Cu_2O} - 2E_{Cu} - 0.5(E_{O_2}^{DFT} + \delta\mu(T) + kT \ln(P/P_0))$$

Which, after grouping terms is:

$$G_f \approx E_{Cu_2O} - 2E_{Cu} - 0.5(E_{O_2}^{DFT}) - 0.5 * \Delta\mu_{O_2}(P, T)$$

with  $\Delta\mu_{O_2}(P, T) = \delta\mu(T) + kT \ln(P/P_0)$ . We get  $\delta\mu(T)$  from the Janaf Tables, or the NIST Webbook.

You will recognize in this equation the standard formation energy we calculated in [Metal oxide oxidation energies](#) plus a correction for the non standard state pressure and temperature ( $\Delta\mu_{O_2}(P, T) = 0$  at standard state).

$$G_f \approx H_f - 0.5 * \Delta\mu_{O_2}(P, T)$$

The formation energy of Cu<sub>2</sub>O is -1.9521 eV/formula unit. The formation energy for Ag<sub>2</sub>O is -0.99 eV/formula unit. Let us consider what temperature the oxides decompose at a fixed oxygen pressure of  $1 \times 10^{-10}$  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.

---

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from ase.units import *
4  from scipy.optimize import fsolve
5
6  K = 1. #not defined in ase.units!
7  atm = 101325*Pascal
8
9  # Shomate parameters valid from 100-700K
10 A = 31.32234; B = -20.23531; C = 57.86644
11 D = -36.50624; E = -0.007374; F = -8.903471
12 G = 246.7945; H = 0.0
13
14 def entropy(T):
15     '''entropy returned as eV/K
16     T in K
17     '''
18     t = T/1000.
19     s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
20     return s*J/mol/K
21
22 def enthalpy(T):
23     ''' H - H(298.15) returned as eV/molecule'''
24     t = T/1000.
25     h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
26     return h*kJ/mol
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
35 T = np.linspace(100,1000)
36 P = 1e-10*atm
37
38 def func(T):
39     'Cu2O'
40     return -1.95 - 0.5*DeltaMu(T,P)
41
42 print 'Cu2O decomposition temperature is {0:1.0f} K'.format(fsolve(func, 900)[0])
43
44 def func(T):
45     'Ag2O'
46     return -0.99 - 0.5*DeltaMu(T,P)
47
48 print 'Ag2O decomposition temperature is {0:1.0f} K'.format(fsolve(func, 470)[0])
49
50 # you have use \\times to escape the first \ in pyplot
51 plt.plot(T, DeltaMu(T,1e10*atm),label='1$\\times 10^{10}$ atm')
52 plt.plot(T, DeltaMu(T,1e5*atm),label='1$\\times 10^5$ atm')
53 plt.plot(T, DeltaMu(T,1*atm),label='1 atm')
54 plt.plot(T, DeltaMu(T,1e-5*atm),label='1$\\times 10^{-5}$ atm')
55 plt.plot(T, DeltaMu(T,1e-10*atm),label='1$\\times 10^{-10}$ atm')
56
57 plt.xlabel('Temperature (K)')
58 plt.ylabel('$\Delta \mu_{O_2}(T,p)$ (eV)')

```

```

59 plt.legend(loc='best')
60 plt.savefig('images/02-mu.png')

```

---

Cu<sub>2</sub>O decomposition temperature is 917 K  
 Ag<sub>2</sub>O decomposition temperature is 478 K

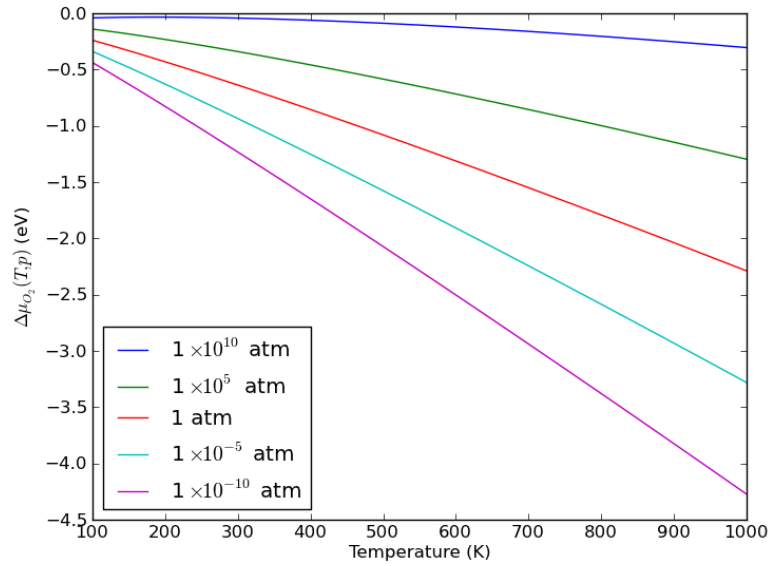


Figure 76:  $\Delta \mu_{O_2}(T,p)$  at different pressures and temperatures.

---

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from ase.units import *
4  from scipy.optimize import fsolve
5
6  K = 1. #not defined in ase.units!
7  atm = 101325*Pascal
8
9  # Shomate parameters valid from 100-700K
10 A = 31.32234; B = -20.23531; C = 57.86644
11 D = -36.50624; E = -0.007374; F = -8.903471
12 G = 246.7945; H = 0.0
13
14 def entropy(T):
15     '''entropy returned as eV/K
16     T in K
17     '''
18     t = T/1000.
19     s = A*np.log(t) + B*t + C*(t**2)/2. + D*(t**3)/3. - E/(2.*t**2) + G
20     return s*kJ/mol/K
21
22 def enthalpy(T):
23     '''H - H(298.15) returned as eV/molecule'''
24     t = T/1000.
25     h = A*t + B*(t**2)/2. + C*(t**3)/3. + D*(t**4)/4. - E/t + F - H
26     return h*kJ/mol
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
35 P = np.logspace(-11,1,10)*atm
36 T = []
37 for p in P:
38
39     def func(T):
40         return -0.99 - 0.5*DeltaMu(T,p)
41     T.append(fsolve(func, 450)[0])
42
43 plt.semilogy(T,P/atm)
44 plt.xlabel('Temperature (K)')
45 plt.ylabel('Pressure (atm)')
46 plt.text(800,1e-7,'Ag')
47 plt.text(600,1e-3,'Ag2O')
48 plt.savefig('images/Ag2O-decomposition.png')

```

---

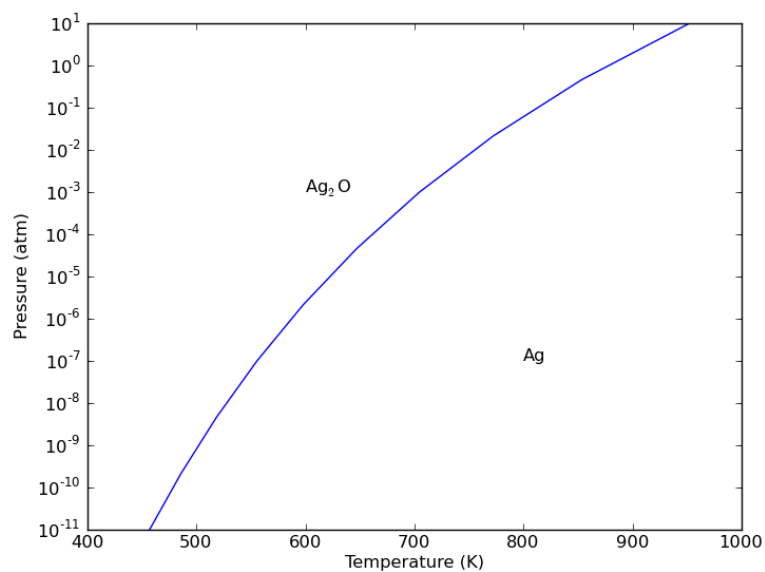


Figure 77: Temperature dependent decomposition pressure for Ag<sub>2</sub>O.

This shows that at high temperature and low  $p_{\text{O}_2}$  metallic silver is stable, but if the  $p_{\text{O}_2}$  gets high enough, the oxide becomes thermodynamically favorable. Here is another way to look at it.

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from ase.units import *
4
5 K = 1. #not defined in ase.units!
6 atm = 101325*Pascal
7 Hf = -0.99

```

```

8
9 P = 1*atm
10
11 Dmu = np.linspace(-4,0)
12
13 Hf = -0.99 - 0.5*Dmu
14
15 plt.plot(Dmu, Hf, label='Ag$_{20}$')
16 plt.plot(Dmu, np.zeros(Hf.shape), label='Ag')
17 plt.xlabel('$\Delta \mu_{O_2}$ (eV)')
18 plt.ylabel('$H_f$ (eV)')
19 plt.savefig('images/atomistic-thermo-hf-mu.png')

```

---

None

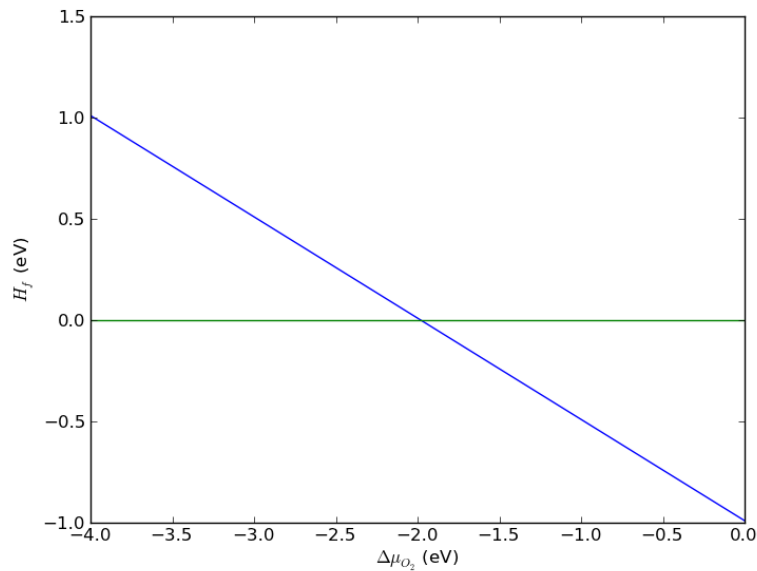


Figure 78: Dependence of the formation energy on the oxygen chemical potential.

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 76): About 500K at  $1 \times 10^{-10}$  atm, 600K at  $1 \times 10^{-5}$  atm, 900K at 1atm, etc. . .

## 6.2 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 also 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  $\Delta H_{ads}$  is the adsorption energy we defined earlier. Now we can examine the effect of  $\Delta\mu(T, p)$  on the adsorption energies. We will use the adsorption energies for the oxygen on Pt(111) system we computed earlier:

Table 6: Adsorption site dependence of adsorption energies of oxygen on Pt(111).

system	$\Delta H(eV/O)$
fcc (0.25 ML)	-1.04
hcp (0.25 ML)	-0.60
bridge (0.25 ML)	-0.49
fcc(1ML)	-0.10

---

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  fcc25 = -1.04
5  hcp25 = -0.60
6  bridge25 = -0.49
7  fcc1 = -0.10
8
9  Dmu = np.linspace(-4,0)
10
11  plt.plot(Dmu, np.zeros(Dmu.shape), label='Pt(111)')
12  plt.plot(Dmu, fcc25 - 0.5*Dmu, label='fcc - 0.25 ML')
13  plt.plot(Dmu, hcp25 - 0.5*Dmu, label='hcp - 0.25 ML')
14  plt.plot(Dmu, bridge25 - 0.5*Dmu, label='bridge - 0.25 ML')
15  plt.plot(Dmu, fcc1 - 0.5*Dmu, label='fcc - 1.0 ML')
16
17  plt.xlabel('$\Delta \mu_{0.25}$ (eV)')
18  plt.ylabel('$\Delta G_{ads}$ (eV/O)')
19  plt.legend(loc='best')
20  plt.savefig('images/atomistic-thermo-adsorption.png')

```

---

None

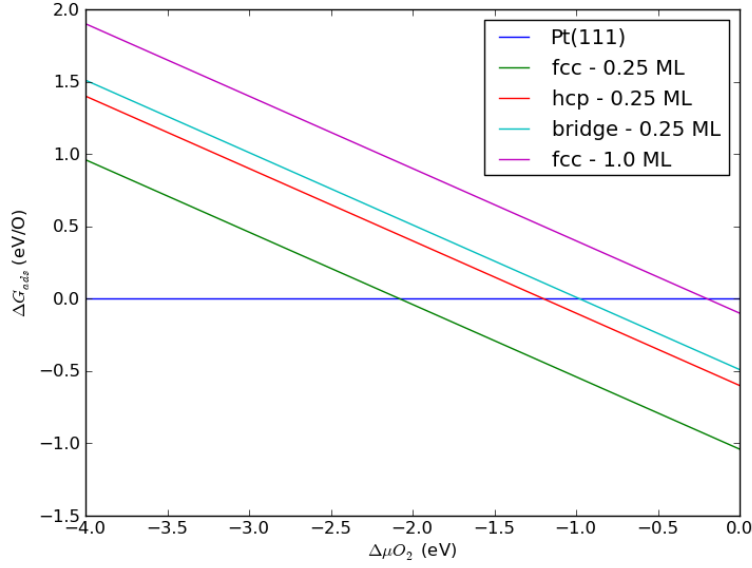


Figure 79: Effect of oxygen chemical potential on the adsorption energy.

### 6.3 Atomistic therodynamics and multiple reactions

In <sup>78</sup> we considered multiple reactions in an atomistic thermodynamic framework. Let us consider these three reactions of dissociative adsorption of hydrogen and hydrogen disulfide, and consider how to compute the reaction energy for the third reaction.

1.  $H_2 + 2* \rightleftharpoons 2H*$
2.  $H_2S + 2* \rightleftharpoons H* + SH*$
3.  $SH* + * \rightleftharpoons S* + H*$

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, preferably 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:

$$\mu_{H*} + \mu_{SH*} = \mu_{H_2S} \text{ and } \mu_{S*} + \mu_{S*} = \mu_{SH*}.$$

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.

## 7 Advanced electronic structure methods

### 7.1 DFT+U

[VASP manual on DFT+U](#)

#### 7.1.1 Metal oxide oxidation energies with DFT+U

We will reconsider here the reaction (see [Metal oxide oxidation energies](#))  $2 \text{Cu}_2\text{O} + \text{O}_2 \rightleftharpoons 4 \text{CuO}$ . We need to compute the energy of each species, now with DFT+U. In <sup>53</sup> they use a U parameter of 4 eV for Cu which gave the best agreement with the experimental value. We will also try that.

#### Cu<sub>2</sub>O calculation with U=4.0

---

```
1 from jasp import *
2 from ase import Atom, Atoms
3
4 with jasp('bulk/Cu2O') as calc:
5     calc.clone('bulk/Cu2O-U=4.0')
6
7 with jasp('bulk/Cu2O-U=4.0') as calc:
8     calc.set(ldau=True, # turn DFT+U on
9             ldautype=2, # select simplified rotationally invariant option
10            ldauluj={'Cu':{'L':2, 'U':4.0, 'J':0.0},
11                   'O':{'L':-1, 'U':0.0, 'J':0.0}},
12            ldauprint=1,
13            ibrion=-1, #do not rerelax
14            nsw=0)
15     calc.calculate()
16     print calc
```

---

```
: -----
VASP calculation from /home/jkitchin/dft-org/bulk/Cu2O-U=4.0
converged: True
Energy = -22.228203 eV

Unit cell vectors (angstroms)
      x      y      z      length
a0 [ 4.270  0.000  0.000] 4.270
a1 [ 0.000  4.270  0.000] 4.270
a2 [ 0.000  0.000  4.270] 4.270
a,b,c,alpha,beta,gamma (deg): 4.270 4.270 4.270 90.0 90.0 90.0
Unit cell volume = 77.854 Ang^3
Stress (GPa):xx,   yy,   zz,   yz,   xz,   xy
              0.018  0.018  0.018 -0.000 -0.000 -0.000
Atom#  sym      position [x,y,z]      tag  rmsForce constraints
  0    Cu  [0.000      0.000      0.000]  0  0.00      T T T
  1    Cu  [2.135      2.135      0.000]  0  0.00      T T T
  2    Cu  [2.135      0.000      2.135]  0  0.00      T T T
  3    Cu  [0.000      2.135      2.135]  0  0.00      T T T
  4     O  [1.067      1.067      1.067]  0  0.00      T T T
```

```

5      0      [3.202      3.202      3.202]      0      0.00      T T T
-----

```

#### INCAR Parameters:

```

-----
      nbands: 37
      nsw: 0
      ibrion: -1
      ldautype: 2
      isif: 3
      ldauprint: 1
      encut: 400.0
      ldau: True
      magmom: None
      ldaul: [-1.0, 2.0]
      ldauj: [0.0, 0.0]
      ldauu: [0.0, 4.0]
      ldau_luj: {'O': {'J': 0.0, 'U': 0.0, 'L': -1}, 'Cu': {'J': 0.0,
      'U': 4.0, 'L': 2}}
      prec: Normal
      kpts: [8, 8, 8]
      reciprocal: False
      xc: PBE
      txt: -
      gamma: False

```

#### Pseudopotentials used:

```

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

```

---

```

1  grep -A 3 "LDA+U is selected, type is set to LDAUTYPE" bulk/Cu2O-U=4.0/OUTCAR

```

---

```

LDA+U is selected, type is set to LDAUTYPE = 2
angular momentum for each species LDAUL =    -1    2
U (eV)           for each species LDAUU =    0.0  4.0
J (eV)           for each species LDAUJ =    0.0  0.0

```

#### CuO calculation with U=4.0

---

```

1  from jasp import *
2  from ase import Atom, Atoms
3
4  with jasp('bulk/CuO') as calc:
5      calc.clone('bulk/CuO-U=4.0')
6
7  with jasp('bulk/CuO-U=4.0') as calc:
8      calc.set(ldau=True, # turn DFT+U on

```

```

9         ldautype=2, # select simplified rotationally invariant option
10        ldauluj={'Cu':{'L':2, 'U':4.0, 'J':0.0},
11                'O':{'L':-1, 'U':0.0, 'J':0.0}},
12        ldauprint=1,
13        ibrion=-1, #do not rerelease
14        nsw=0)
15    calc.calculate()
16    print calc

```

---

: -----

VASP calculation from /home/jkitchin/dft-org/bulk/CuO-U=4.0

converged: True

Energy = -16.870130 eV

Unit cell vectors (angstroms)

	x	y	z	length
a0	2.302	-1.776	0.046	2.908
a1	2.302	1.776	0.046	2.908
a2	-0.762	0.000	5.087	5.144

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

a0	2.302	-1.776	0.046	2.908
a1	2.302	1.776	0.046	2.908
a2	-0.762	0.000	5.087	5.144

a,b,c,alpha,beta,gamma (deg): 2.908 2.908 5.144 95.8 95.8 95.8

Unit cell volume = 41.730 Ang^3

Stress (GPa):	xx	yy	zz	yz	xz	xy
	0.030	0.018	0.027	-0.000	-0.007	-0.000

Atom#	sym	position [x,y,z]	tag	rmsForce	constraints
0	Cu	[1.151 0.888 0.023]	0	0.00	T T T
1	Cu	[0.770 -0.888 2.566]	0	0.00	T T T
2	O	[2.111 -0.168 1.318]	0	0.04	T T T
3	O	[1.730 0.168 3.861]	0	0.04	T T T

-----

INCAR Parameters:

-----

```

nbands: 23
nsw: 0
ibrion: -1
ldautype: 2
isif: 3
ldauprint: 1
encut: 400.0
ldau: True
magmom: None
ldaul: [-1.0, 2.0]
ldauj: [0.0, 0.0]
ldauu: [0.0, 4.0]
ldauluj: {'O': {'J': 0.0, 'U': 0.0, 'L': -1}, 'Cu': {'J': 0.0,
    'U': 4.0, 'L': 2}}
prec: Normal
kpts: [8, 8, 8]

```

```

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

```

Pseudopotentials used:

-----

```

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

```

## Reaction energy calculation with DFT+U

---

```

1 from jasp import *
2
3 # don't forget to normalize your total energy to a formula unit. Cu2O
4 # has 3 atoms, so the number of formula units in an atoms is
5 # len(atoms)/3.
6 with jasp('bulk/Cu2O-U=4.0') as calc:
7     atoms = calc.get_atoms()
8     cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
9
10 with jasp('bulk/CuO-U=4.0') as calc:
11     atoms = calc.get_atoms()
12     cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
13
14 # make sure to use the same cutoff energy for the O2 molecule!
15 with jasp('molecules/O2-sp-triplet-400') as calc:
16     atoms = calc.get_atoms()
17     o2_energy = atoms.get_potential_energy()
18
19 rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
20 print 'Reaction energy = {0} eV'.format(rxn_energy)
21 print 'Corrected energy = {0} eV'.format(rxn_energy - 1.36)

```

---

```

Reaction energy = -1.663819 eV
Corrected energy = -3.023819 eV

```

This is still not in quantitative agreement with the result in,<sup>53</sup> which at  $U=4$  eV is about -3.14 eV (estimated from a graph). We have not applied the  $O_2$  correction here yet. In that paper, they apply a constant shift of -1.36 eV per  $O_2$ . After we apply that correction, we agree within 0.12 eV, which is pretty good considering we have not checked for convergence.

**How much does  $U$  affect the reaction energy?** It is reasonable to consider how sensitive our results are to the  $U$  parameter. We do that here.

---

```

1 from jasp import *
2 for U in [2.0, 4.0, 6.0]:
3     ## Cu2O #####
4     with jasp('bulk/Cu2O') as calc:
5         calc.clone('bulk/Cu2O-U={0}'.format(U))
6
7     with jasp('bulk/Cu2O-U={0}'.format(U)) as calc:
8         calc.set(ldau=True, # turn DFT+U on
9                 ldautype=2, # select simplified rotationally invariant option
10                ldau_luj={'Cu':{'L':2, 'U':U, 'J':0.0},

```

```

11         'O':{'L':-1, 'U':0.0, 'J':0.0}},
12         ldauprint=1,
13         ibrion=-1, #do not rerelease
14         nsw=0)
15     atoms = calc.get_atoms()
16     cu2o_energy = atoms.get_potential_energy()/(len(atoms)/3)
17
18     ## CuO #####
19     with jasp('bulk/CuO') as calc:
20         calc.clone('bulk/CuO-U={0}'.format(U))
21
22     with jasp('bulk/CuO-U={0}'.format(U)) as calc:
23         calc.set(ldau=True, # turn DFT+U on
24                 ldautype=2, # select simplified rotationally invariant option
25                 ldau_luj={'Cu':{'L':2, 'U':U, 'J':0.0},
26                          'O':{'L':-1, 'U':0.0, 'J':0.0}},
27                 ldauprint=1,
28                 ibrion=-1, #do not rerelease
29                 nsw=0)
30     atoms = calc.get_atoms()
31     cuo_energy = atoms.get_potential_energy()/(len(atoms)/2)
32
33     ## O2 #####
34     # make sure to use the same cutoff energy for the O2 molecule!
35     with jasp('molecules/O2-sp-triplet-400') as calc:
36         atoms = calc.get_atoms()
37         o2_energy = atoms.get_potential_energy()
38
39     rxn_energy = 4.0*cuo_energy - o2_energy - 2.0*cu2o_energy
40     print 'U = {0} reaction energy = {1}'.format(U,rxn_energy - 1.99)

```

---

```

U = 2.0 reaction energy = -3.876906
U = 4.0 reaction energy = -3.653819
U = 6.0 reaction energy = -3.397605

```

In,<sup>53</sup> the difference in reaction energy from U=2 eV to U=4 eV was about 0.5 eV (estimated from graph). Here we see a range of 0.48 eV from U=2 eV to U=4 eV. Note that for U=0 eV, we had a (corrected reaction energy of -3.96 eV). Overall, the effect of adding U decreases this reaction energy.

This example highlights the challenge of using an approach like DFT+U. On one hand, U has a clear effect of changing the reaction energy. On the other hand, so does the correction factor for the O<sub>2</sub> binding energy. In<sup>53</sup> the authors tried to get the O<sub>2</sub> 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<sup>90</sup> 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.

## 7.2 Hybrid functionals

### 7.2.1 FCC Ni DOS

This example is adapted from [http://cms.mpi.univie.ac.at/wiki/index.php/FccNi\\_DOS](http://cms.mpi.univie.ac.at/wiki/index.php/FccNi_DOS)

---

```

1 from jasp import *
2 from ase.lattice.cubic import FaceCenteredCubic
3 from ase.dft import DOS
4

```

```

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

```

---

```

PBE energy:      -5.530247
PBE0 energy:     -6.848931
HSE06 energy:    -6.293369

```



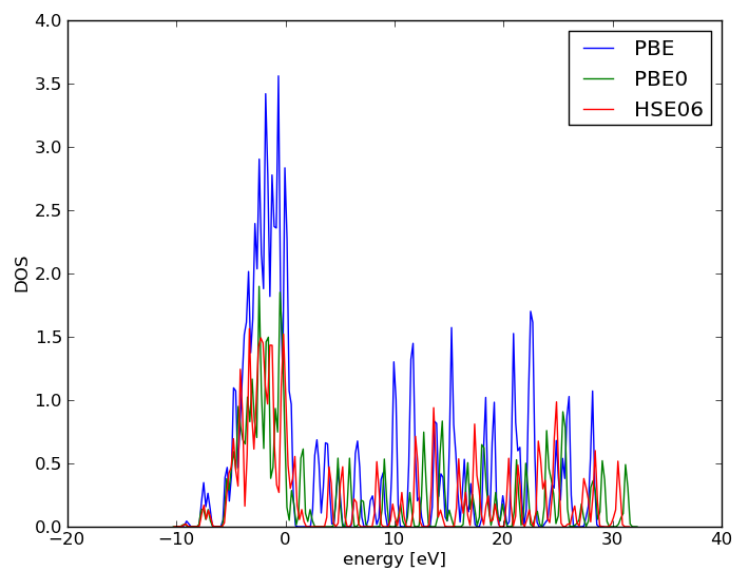


Figure 80: Comparison of DOS from GGA, and two hybrid GGAs (PBE0 and HSE06).

## 7.3 TODO DFT+D

[http://cms.mpi.univie.ac.at/vasp/vasp/DFT\\_D2\\_method\\_Grimme.html](http://cms.mpi.univie.ac.at/vasp/vasp/DFT_D2_method_Grimme.html)<sup>91</sup>

Van der Waal forces can play a considerable role in binding of aromatic molecules to metal surfaces (ref). Here we consider the effects of these forces on the adsorption energy of benzene on an Au(111) surface. First, we consider the regular PBE functional.

### 7.3.1 PBE

#### gas-phase benzene

---

```

1 from jasp import *
2 from ase.data.molecules import molecule
3
4 benzene = molecule('C6H6')
5 benzene.center(vacuum=5)
6
7 with jasp('molecules/benzene-pbe',
8         xc='PBE',
9         encut=350,
10        kpts=(1,1,1),
11        ibrion=1,
12        nsw=100,
13        atoms=benzene) as calc:
14     print benzene.get_potential_energy()
```

---

-76.011625

#### clean slab

---

```

1  # the clean gold slab
2  from jasp import *
3  from ase.lattice.surface import fcc111, add_adsorbate
4  from ase.constraints import FixAtoms
5
6  atoms = fcc111('Au', size=(3,3,3), vacuum=10)
7
8  # now we constrain the slab
9  c = FixAtoms(mask=[atom.symbol=='Au' for atom in atoms])
10 atoms.set_constraint(c)
11
12 #from ase.visualize import view; view(atoms)
13
14 with jasp('surfaces/Au-pbe',
15         xc='PBE',
16         encut=350,
17         kpts=(4,4,1),
18         ibrion=1,
19         nsw=100,
20         atoms=atoms) as calc:
21     print atoms.get_potential_energy()

```

---

-82.419513

## benzene on Au(111)

---

```

1  # Benzene on the slab
2  from jasp import *
3  from ase.lattice.surface import fcc111, add_adsorbate
4  from ase.data.molecules import molecule
5  from ase.constraints import FixAtoms
6
7  atoms = fcc111('Au', size=(3,3,3), vacuum=10)
8  benzene = molecule('C6H6')
9  benzene.translate(-benzene.get_center_of_mass())
10
11 # I want the benzene centered on the position in the middle of atoms
12 # 20, 22, 23 and 25
13 p = (atoms.positions[20] +
14     atoms.positions[22] +
15     atoms.positions[23] +
16     atoms.positions[25])/4.0 + [0.0, 0.0, 3.05]
17
18 benzene.translate(p)
19 atoms += benzene
20
21 # now we constrain the slab
22 c = FixAtoms(mask=[atom.symbol=='Au' for atom in atoms])
23 atoms.set_constraint(c)
24
25 #from ase.visualize import view; view(atoms)
26
27 with jasp('surfaces/Au-benzene-pbe',
28         xc='PBE',
29         encut=350,
30         kpts=(4,4,1),
31         ibrion=1,
32         nsw=100,
33         atoms=atoms) as calc:
34     print atoms.get_potential_energy()

```

---

### 7.3.2 DFT-D2

To turn on the van der Waals corrections<sup>92</sup> we set `LVDW` to `True`.

#### gas-phase benzene

---

```
1 from jasp import *
2 from ase.data.molecules import molecule
3
4 benzene = molecule('C6H6')
5 benzene.center(vacuum=5)
6
7 with jasp('molecules/benzene-pbe-d2',
8          xc='PBE',
9          encut=350,
10         kpts=(1,1,1),
11         ibrion=1,
12         nsw=100,
13         lvdw=True,
14         atoms=benzene) as calc:
15     print benzene.get_potential_energy()
```

---

#### clean slab

---

```
1 # the clean gold slab
2 from jasp import *
3 from ase.lattice.surface import fcc111, add_adsorbate
4 from ase.constraints import FixAtoms
5
6 atoms = fcc111('Au', size=(3,3,3), vacuum=10)
7
8 # now we constrain the slab
9 c = FixAtoms(mask=[atom.symbol=='Au' for atom in atoms])
10 atoms.set_constraint(c)
11
12 #from ase.visualize import view; view(atoms)
13
14 with jasp('surfaces/Au-pbe-d2',
15         xc='PBE',
16         encut=350,
17         kpts=(4,4,1),
18         ibrion=1,
19         nsw=100,
20         lvdw=True,
21         atoms=atoms) as calc:
22     print atoms.get_potential_energy()
```

---

#### benzene on Au(111)

---

```
1 # Benzene on the slab
2 from jasp import *
3 from ase.lattice.surface import fcc111, add_adsorbate
4 from ase.data.molecules import molecule
5 from ase.constraints import FixAtoms
6
7 atoms = fcc111('Au', size=(3,3,3), vacuum=10)
8 benzene = molecule('C6H6')
9 benzene.translate(-benzene.get_center_of_mass())
10
11 # I want the benzene centered on the position in the middle of atoms
```

```

12 # 20, 22, 23 and 25
13 p = (atoms.positions[20] +
14      atoms.positions[22] +
15      atoms.positions[23] +
16      atoms.positions[25])/4.0 + [0.0, 0.0, 3.05]
17
18 benzene.translate(p)
19 atoms += benzene
20
21 # now we constrain the slab
22 c = FixAtoms(mask=[atom.symbol=='Au' for atom in atoms])
23 atoms.set_constraint(c)
24
25 #from ase.visualize import view; view(atoms)
26
27 with jasp('surfaces/Au-benzene-pbe-d2',
28          xc='PBE',
29          encut=350,
30          kpts=(4,4,1),
31          ibrion=1,
32          nsw=100,
33          lvdw=True,
34          atoms=atoms) as calc:
35     print atoms.get_potential_energy()

```

---

### 7.3.3 Advanced vdW-DF functionals

There is a more sophisticated (i.e. more parameters) treatment of van der Waal forces in VASP ([http://cms.mpi.univie.ac.at/vasp/vasp/vdW\\_DF\\_functional\\_Langreth\\_Lundqvist\\_et\\_al.html](http://cms.mpi.univie.ac.at/vasp/vasp/vdW_DF_functional_Langreth_Lundqvist_et_al.html)).

## 7.4 ELF

Need better intro here.

---

```

1 # compute ELF for CF4
2 from jasp import *
3 from ase.data.molecules import molecule
4 from enthought.mayavi import mlab
5
6 atoms = molecule('CF4')
7 atoms.center(vacuum=5)
8
9 with jasp('molecules/cf4-elf',
10         encut=350,
11         prec='high',
12         ismear=0,
13         sigma=0.01,
14         xc='PBE',
15         lelf=True,
16         atoms=atoms) as calc:
17     calc.calculate()
18
19     x,y,z,elf = calc.get_elf()
20     mlab.contour3d(x,y,z,elf,contours=[0.3])
21     mlab.savefig('../..//images/cf4-elf-3.png')
22
23     mlab.figure()
24     mlab.contour3d(x,y,z,elf,contours=[0.75])
25     mlab.savefig('../..//images/cf4-elf-75.png')

```

---

None

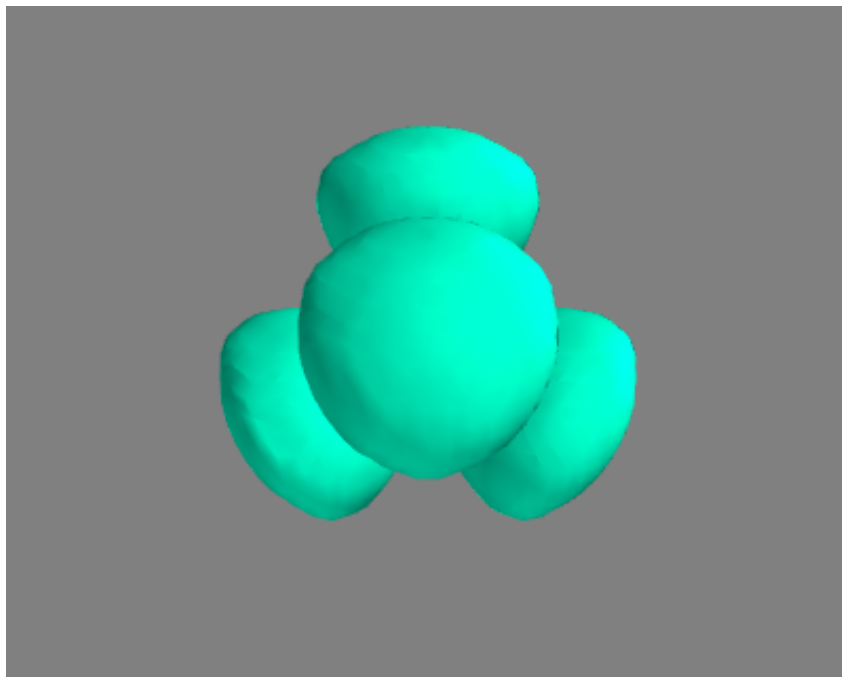


Figure 81: ELF for an isosurface of 0.3 for  $\text{CF}_4$ .

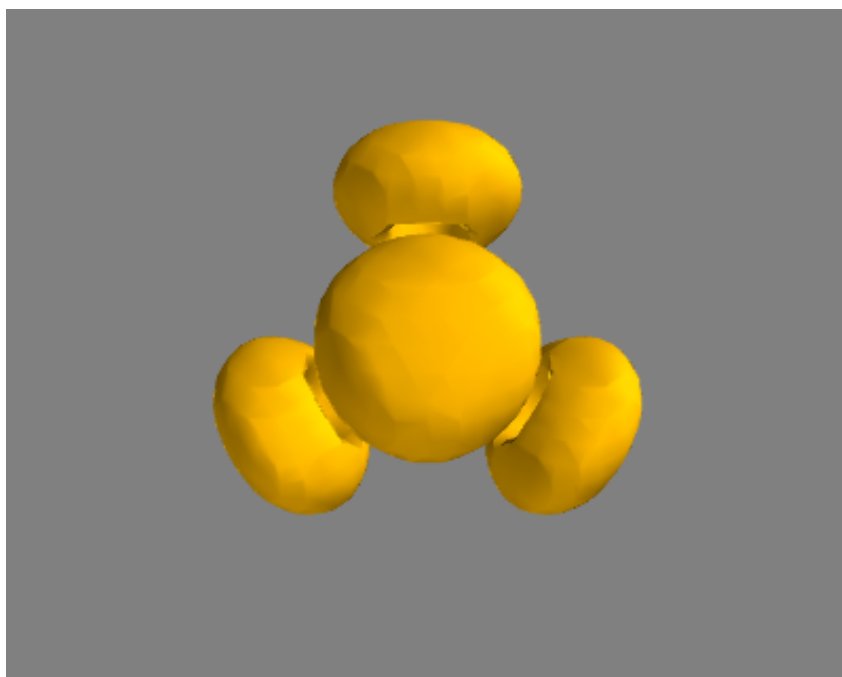


Figure 82: ELF for an isosurface of 0.75 for  $\text{CF}_4$ .

These images (Figure 81 and 82) are basically consistent with those in Reference.<sup>93</sup>

## 7.5 TODO Charge partitioning schemes

## 7.6 TODO Modeling Core level shifts

# 8 Acknowledgments

I would like to thank Zhongnan Xu for sending me some examples on magnetism. Alan McGaughey and Lars Grabow for sending me some NEB examples. Matt Curnan for examples of phonons.

Many thanks to students in my class who have pointed out typos, places of confusion, etc. . . These include Bruno Calfa, Matt Curnan, Charlie Janini, Feng Cao, Gamze Gumuslu, Nicholas Chisholm, Prateek Mehta, Qiyang Duan, Shubhaditya Majumdar, Steven Illes, Wee-Liat Ong, Ye Wang, Yichun Sun, Yubing Lu, and Zhongnan Xu.

# 9 Appendices

## 9.1 Recipes

### 9.1.1 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.

---

```
1 import textwrap
2 from ase.data.molecules import molecule
3
4 atoms = molecule('CH3CH2OH')
5 print atoms
6
7 #delete all the hydrogens
8 ind2del = [atom.index for atom in atoms if atom.symbol=='H']
9 print 'Indices to delete: ',ind2del
10
11 del atoms[ind2del]
12
13 # now print what is left
14 print atoms
```

---

```
Atoms(symbols='C2OH6', positions=..., cell=[1.0, 1.0, 1.0], pbc=[False, False, False])
Indices to delete:  [3, 4, 5, 6, 7, 8]
Atoms(symbols='C2O', positions=..., cell=[1.0, 1.0, 1.0], pbc=[False, False, False])
```

### 9.1.2 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 equivalently in decimal numbers, a tag of 3.

---

```

1  '''
2  adapted from https://listserv.fysik.dtu.dk/pipermail/campos/2004-September/001155.html
3  '''
4
5  from ase import *
6  from ase.io import write
7  from ase.lattice.surface import bcc111, add_adsorbate
8  from ase.constraints import FixAtoms
9
10 # the bcc111 function automatically tags atoms
11 slab = bcc111('W',
12               a=3.92,          # W lattice constant
13               size=(2,2,6),    # 6-layer slab in 2x2 configuration
14               vacuum=10.0)
15
16 #reset tags to be powers of two so we can use binary math
17 slab.set_tags([2**a.get_tag() for a in slab])
18
19 # we had 6 layers, so we create new tags starting at 7
20 # Note you must use powers of two for all the tags!
21 LAYER1 = 2
22 ADSORBATE = 2**7
23 FREE = 2**8
24 NEARADSORBATE = 2**9
25
26 # let us tag LAYER1 atoms to be FREE too. we can address it by LAYER1 or FREE
27 tags = slab.get_tags()
28 for i,tag in enumerate(tags):
29     if tag == LAYER1:
30         tags[i] += FREE
31 slab.set_tags(tags)
32
33 #create a CO molecule
34 co= Atoms([Atom('C',[0., 0., 0. ], tag=ADSORBATE),
35           Atom('O',[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 let us 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 tags[20]+=NEARADSORBATE
45 tags[21]+=NEARADSORBATE
46 tags[22]+=NEARADSORBATE
47 slab.set_tags(tags)
48 #update the tags
49 slab.set_tags(tags)
50
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 layer1 = slab[(slab.get_tags() & LAYER1) == LAYER1]
57
58 #atoms defined as near the adsorbate
59 nearads = slab[(slab.get_tags() & NEARADSORBATE) == NEARADSORBATE]
60
61 #atoms that are free
62 free = slab[(slab.get_tags() & FREE) == FREE]
63
64 #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 slab.set_constraint(constraint)
72 write('images/tagged-bcc111.png', slab, rotation='-90x', show_unit_cell=2)
73 from ase.visualize import view; view(slab)

```

---



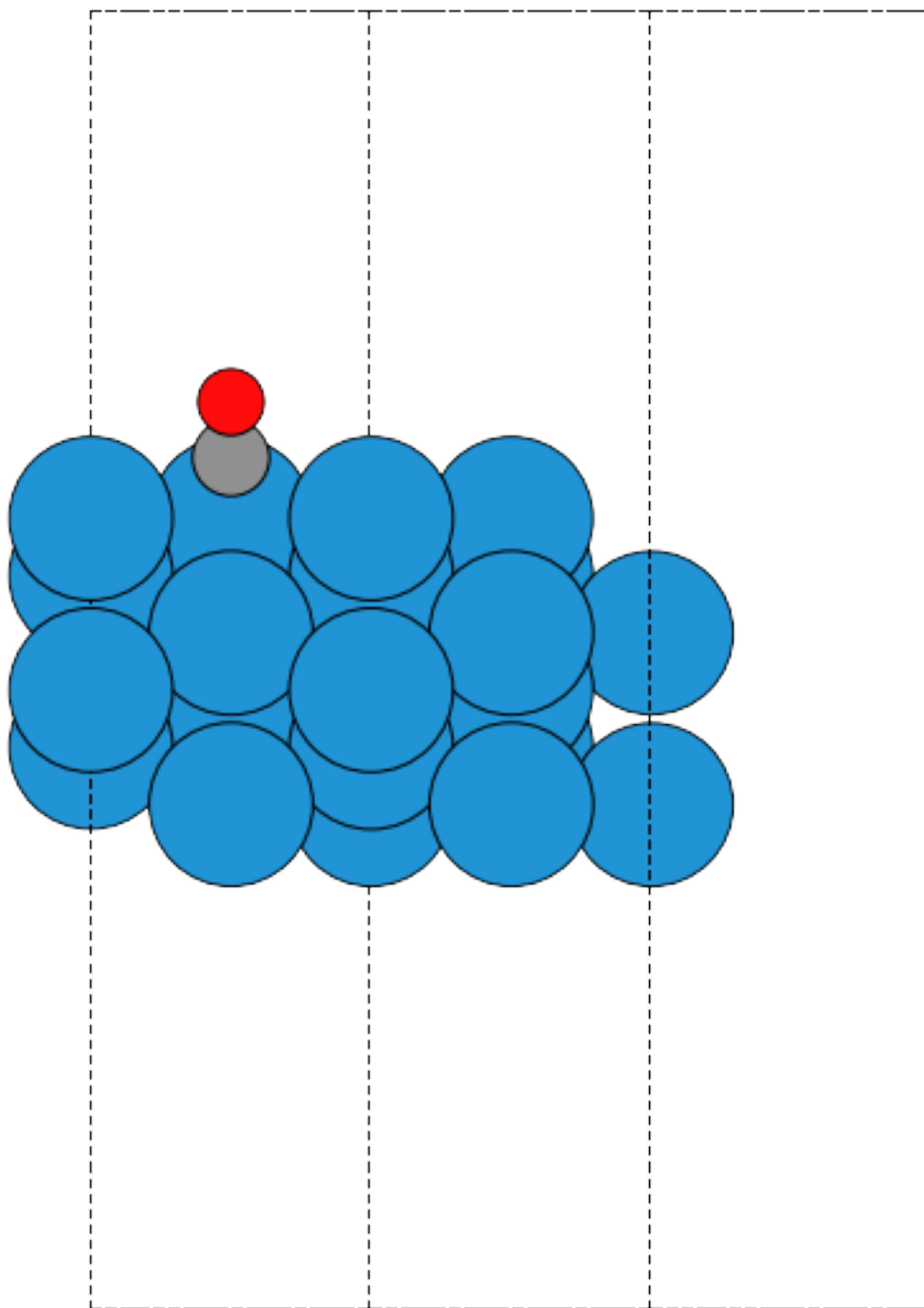


Figure 83: The tagged bcc(111) structure created above. Unfortunately, the frozen atoms do not show up in the figure.

### 9.1.3 Using units in ase

ase uses a base set of atomic units. These are eV for energy, Å for distance, seconds for time, and amu for mass. 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 units algebra is enforced (i.e. it will be ok to add a m and a kg)!

---

```
1 from ase.units import *
2
3 d = 1*Angstrom
4 print ' d = {0} nm'.format(d/nm)
5
6 print '1 eV = {0} Hartrees'.format(eV/Hartree)
7 print '1 eV = {0} Rydbergs'.format(eV/Rydberg)
8 print '1 eV = {0} kJ/mol'.format(eV/(kJ/mol))
9 print '1 eV = {0} kcal/mol'.format(eV/(kcal/mol))
10
11 print '1 Hartree = {0} kcal/mol'.format(1*Hartree/(kcal/mol))
12 print '1 Rydberg = {0} eV'.format(1*Rydberg/eV)
13
14 # derived units
15 minute = 60*s
16 hour = 60*minute
17
18 #convert 10 hours to minutes
19 print '10 hours = {0} minutes'.format(10*hour/minute)
```

---

```

d = 0.1 nm
1 eV = 0.036749 Hartrees
1 eV = 0.073499 Rydbergs
1 eV = 96.485309 kJ/mol
1 eV = 23.060542 kcal/mol
1 Hartree = 627.509541 kcal/mol
1 Rydberg = 13.605698 eV
10 hours = 600.0 minutes
```

### 9.1.4 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 data between a particular time interval. You can slice a numpy array to extract parts of it. See <http://www.scipy.org/Cookbook/Indexing> for several examples of this.

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

```

1  import numpy as np
2  import matplotlib as mpl
3  #http://matplotlib.sourceforge.net/users/customizing.html
4  mpl.rcParams['legend.numpoints'] = 1 #default is 2
5  import matplotlib.pyplot as plt
6
7  x = np.linspace(0,6,100)
8  y = np.cos(x)
9
10 plt.plot(x,y,label='full')
11
12 ind = (x>2) & (x<4)
13
14 subx = x[ind]
15 suby = y[ind]
16
17 plt.plot(subx,suby,'bo',label='sliced')
18 xlabel('x')
19 ylabel('cos(x)')
20 plt.legend(loc='lower right')
21 plt.savefig('images/np-array-slice.png')

```

---

None

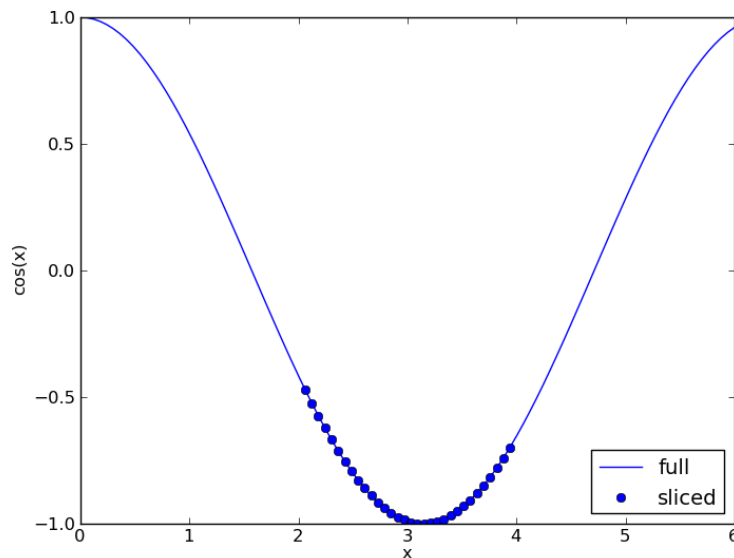


Figure 84: Example of slicing out part of an array. The solid line represents the whole array, and the symbols are the array between  $2 < x < 4$ .

The expression  $x > 2$  returns an array of booleans (True where the element of  $x$  is greater than 2, and False where it is not) equal in size to  $x$ . Similarly  $x < 4$  returns a boolean array where  $x$  is less than 4. We take the logical **and** of these two boolean arrays to get another boolean array where both conditions are True (i.e.  $x < 2$  and  $x > 4$ ). This final boolean array is **True** for the part of the arrays we are interested in, and we can use it to extract the subarrays we want.

### 9.1.5 Statistics

**Confidence intervals** `scipy` has a statistical package available for getting statistical distributions. This is useful for computing confidence intervals using the student-t tables. Here is an example of computing a 95% confidence interval on an average.

---

```
1 import numpy as np
2 from scipy.stats.distributions import t
3
4 n = 10 #number of measurements
5 dof = n - 1 #degrees of freedom
6 avg_x = 16.1 #average measurement
7 std_x = 0.01 #standard deviation of measurements
8
9 #Find 95% prediction interval for next measurement
10
11 alpha = 1.0 - 0.95
12
13 pred_interval = t.ppf(1-alpha/2., dof) * std_x * np.sqrt(1.+1./n)
14
15 s = ['We are 95% confident the next measurement',
16      ' will be between {0:1.3f} and {1:1.3f}']
17 print ''.join(s).format(avg_x - pred_interval, avg_x + pred_interval)
```

---

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

### 9.1.6 Curve fitting

#### Linear fitting

---

```
1 #examples of linear curve fitting using least squares
2 import numpy as np
3
4 xdata = np.array([0.,1.,2.,3.,4.,5.,6.])
5 ydata = np.array([0.1, 0.81, 4.03, 9.1, 15.99, 24.2, 37.2])
6
7 #fit a third order polynomial
8 from pylab import polyfit, plot, xlabel, ylabel, show, legend, savefig
9 pars = polyfit(xdata,ydata,3)
10 print 'pars from polyfit: {0}'.format(pars)
11
12 ## numpy method returns more data
13 A = np.column_stack([xdata**3,
14                     xdata**2,
15                     xdata,
16                     np.ones(len(xdata),np.float)])
17
18 pars_np,resids,rank,s = np.linalg.lstsq(A, ydata)
19 print 'pars from np.linalg.lstsq: {0}'.format(pars_np)
20
21 '''
22 we are trying to solve Ax = b for x in the least squares sense. There
23 are more rows in A than elements in x so, we can left multiply each
24 side by A^T, and then solve for x with an inverse.
25
26 A^T Ax = A^T b
27 x = (A^T A)^-1 A^T b
28 '''
29 # not as pretty but equivalent!
30 pars_man= np.dot(np.linalg.inv(np.dot(A.T,A)), np.dot(A.T,ydata))
31 print 'pars from linear algebra: {0}'.format(pars_man)
```

---

```

32
33 #but, it is easy to fit an exponential function to it!
34 # y = a*exp(x)+b
35 Aexp = np.column_stack([np.exp(xdata), np.ones(len(xdata), np.float)])
36 pars_exp=np.dot(np.linalg.inv(np.dot(Aexp.T, Aexp)), np.dot(Aexp.T, ydata))
37
38 plot(xdata, ydata, 'ro')
39 fity = np.dot(A, pars)
40 plot(xdata, fity, 'k-', label='poly fit')
41 plot(xdata, np.dot(Aexp, pars_exp), 'b-', label='exp fit')
42 xlabel('x')
43 ylabel('y')
44 legend()
45 savefig('images/curve-fit-1.png')

```

---

```

pars from polyfit: [ 0.04861111  0.63440476  0.61365079 -0.08928571]
pars from np.linalg.lstsq: [ 0.04861111  0.63440476  0.61365079 -0.08928571]
pars from linear algebra: [ 0.04861111  0.63440476  0.61365079 -0.08928571]

```

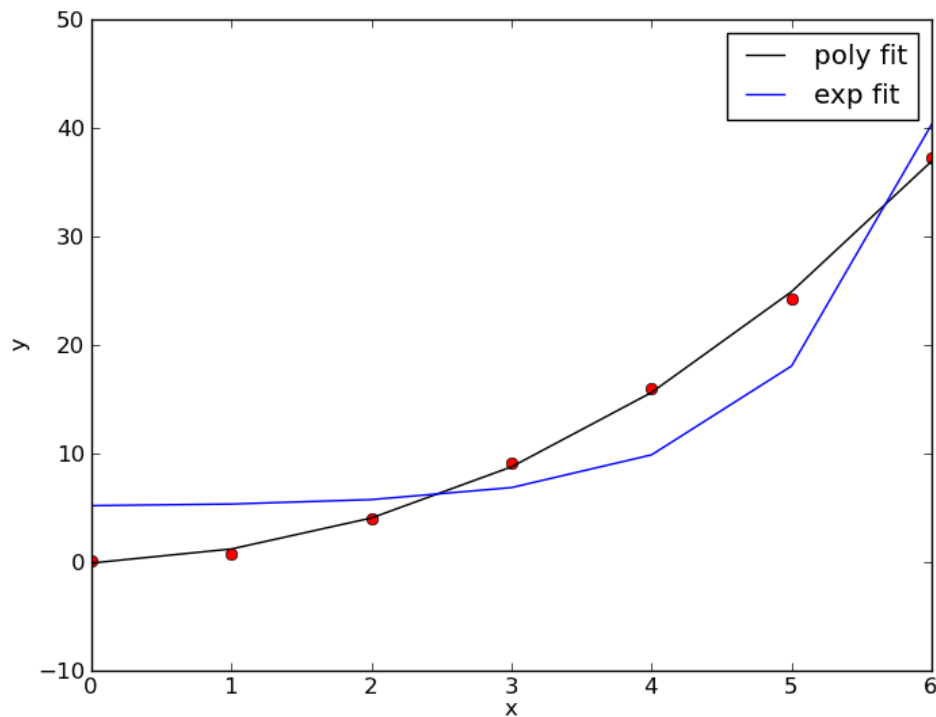


Figure 85: Example of linear least-squares curve fitting.

### 9.1.7 Nonlinear curve fitting

```

1 from scipy.optimize import leastsq
2 import numpy as np
3

```

```

4 vols = np.array([13.71, 14.82, 16.0, 17.23, 18.52])
5
6 energies = np.array([-56.29, -56.41, -56.46, -56.463,-56.41])
7
8 def Murnaghan(parameters,vol):
9     'From Phys. Rev. B 28, 5480 (1983)'
10    E0 = parameters[0]
11    B0 = parameters[1]
12    BP = parameters[2]
13    V0 = parameters[3]
14
15    E = E0 + B0*vol/BP*((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
16
17    return E
18
19 def objective(pars,y,x):
20     #we will minimize this function
21     err = y - Murnaghan(pars,x)
22     return err
23
24 x0 = [ -56., 0.54, 2., 16.5] #initial guess of parameters
25
26 plsq = leastsq(objective, x0, args=(energies,vols))
27
28 print 'Fitted parameters = {0}'.format(plsq[0])
29
30 import matplotlib.pyplot as plt
31 plt.plot(vols,energies,'ro')
32
33 #plot the fitted curve on top
34 x = np.linspace(min(vols),max(vols),50)
35 y = Murnaghan(plsq[0],x)
36 plt.plot(x,y,'k-')
37 plt.xlabel('Volume')
38 plt.ylabel('energy')
39 plt.savefig('images/nonlinear-curve-fitting.png')

```

---

Fitted parameters = (array([-56.46839641, 0.57233217, 2.7407944 , 16.55905648]), 1)

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

## 9.1.8 Nonlinear curve fitting by direct least squares minimization

---

```

1 from scipy.optimize import fmin
2 import numpy as np
3
4 volumes = np.array([13.71, 14.82, 16.0, 17.23, 18.52])
5
6 energies = np.array([-56.29, -56.41, -56.46, -56.463,-56.41])
7
8 def Murnaghan(parameters,vol):
9     'From PRB 28,5480 (1983)'
10    E0 = parameters[0]
11    B0 = parameters[1]
12    BP = parameters[2]
13    V0 = parameters[3]
14
15    E = E0 + B0*vol/BP*((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
16
17    return E

```

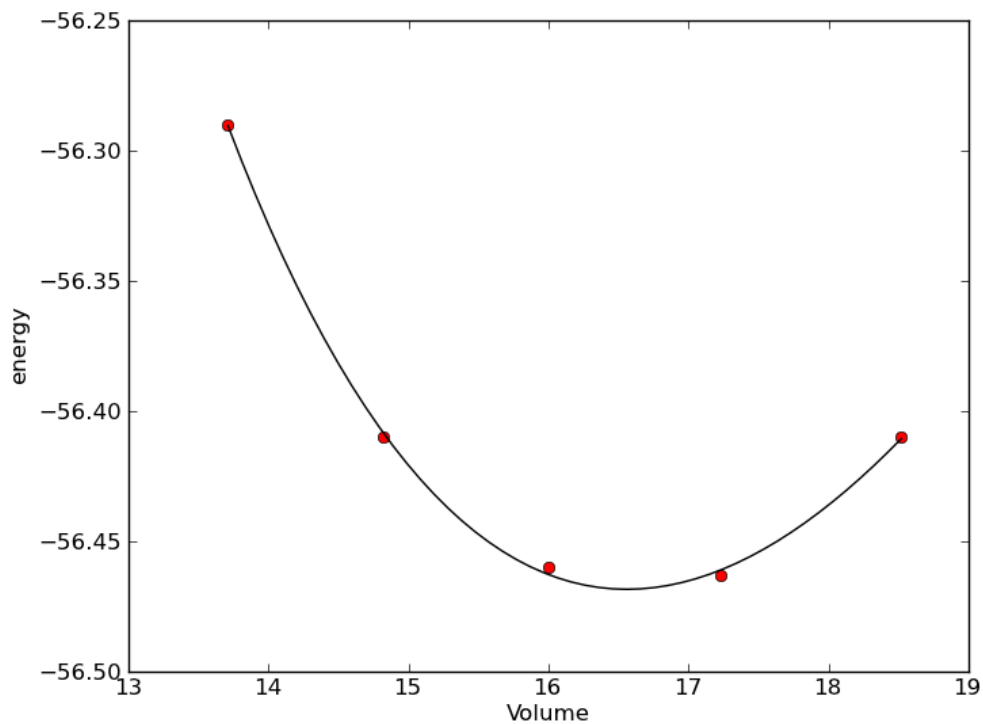


Figure 86: Example of least-squares non-linear curve fitting.

```

18
19 def objective(pars,vol):
20     #we will minimize this function
21     err = energies - Murnaghan(pars,vol)
22     return np.sum(err**2) #we return the summed squared error directly
23
24 x0 = [ -56., 0.54, 2., 16.5] #initial guess of parameters
25
26 plsq = fmin(objective,x0,args=(volumes,)) #note args is a tuple
27
28 print 'parameters = {0}'.format(plsq)
29
30 import matplotlib.pyplot as plt
31 plt.plot(volumes,energies,'ro')
32
33 #plot the fitted curve on top
34 x = np.linspace(min(volumes),max(volumes),50)
35 y = Murnaghan(plsq,x)
36 plt.plot(x,y,'k-')
37 plt.xlabel('Volume ($\AA^3$)')
38 plt.ylabel('Total energy (eV)')
39 plt.savefig('images/nonlinear-fitting-lsq.png')

```

---

Optimization terminated successfully.  
 Current function value: 0.000020  
 Iterations: 137  
 Function evaluations: 240

```
parameters = [-56.46932645    0.59141447    1.9044796    16.59341303]
```

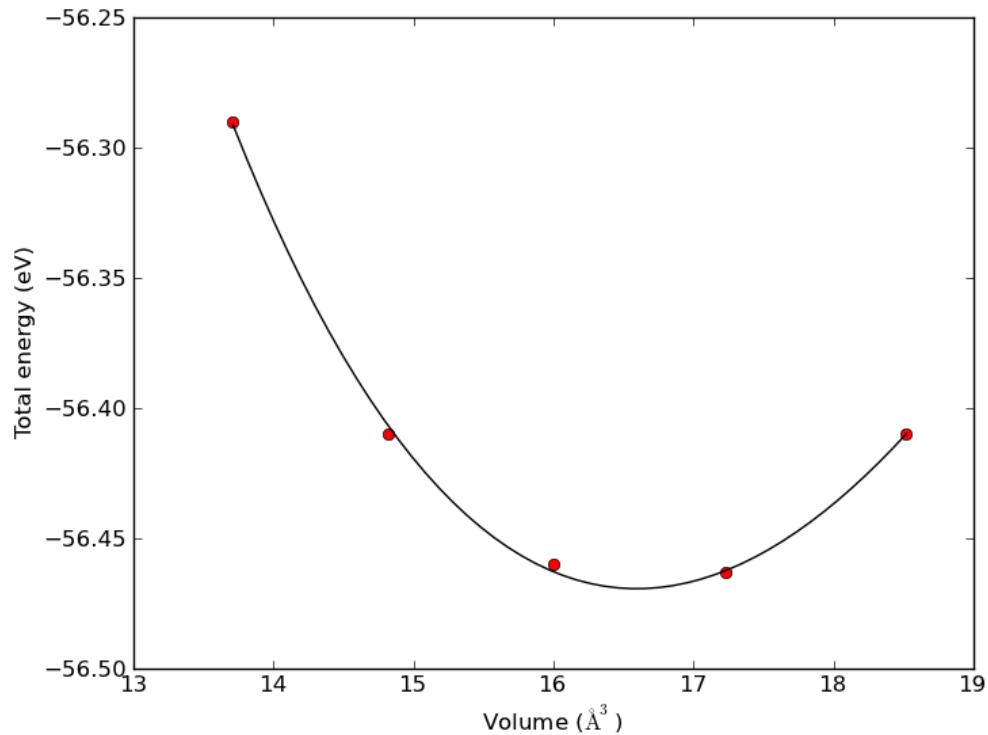


Figure 87: Fitting a nonlinear function.

### 9.1.9 Nonlinear curve fitting with confidence intervals

```
1  # Nonlinear curve fit with confidence interval
2  import numpy as np
3  from scipy.optimize import curve_fit
4  from scipy.stats.distributions import t
5
6  '''
7  fit this equation to data
8   $y = c1 \exp(-x) + c2*x$ 
9
10 this is actually a linear regression problem, but it is convenient to
11 use the nonlinear fitting routine because it makes it easy to get
12 confidence intervals. The downside is you need an initial guess.
13
14 from Matlab
15 b =
16
17     4.9671
18     2.1100
19
20
21 bint =
22
```



```

23     4.6267    5.3075
24     1.7671    2.4528
25     '''
26
27 x = np.array([ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
28 y = np.array([ 4.70192769, 4.46826356, 4.57021389, 4.29240134, 3.88155125,
29               3.78382253, 3.65454727, 3.86379487, 4.16428541, 4.06079909])
30
31 # this is the function we want to fit to our data
32 def func(x,c0, c1):
33     return c0 * np.exp(-x) + c1*x
34
35 pars, pcov = curve_fit(func, x, y, p0=[4.96, 2.11])
36
37 alpha = 0.05 # 95% confidence interval
38
39 n = len(y)    # number of data points
40 p = len(pars) # number of parameters
41
42 dof = max(0, n-p) # number of degrees of freedom
43
44 tval = t.ppf(1.0-alpha/2., dof) # student-t value for the dof and confidence level
45
46 for i, p,var in zip(range(n), pars, np.diag(pcov)):
47     sigma = var**0.5
48     print 'c{0}: {1} [{2} {3}]'.format(i, p,
49                                         p - sigma*tval,
50                                         p + sigma*tval)
51
52 import matplotlib.pyplot as plt
53 plt.plot(x,y,'bo ')
54 xfit = np.linspace(0,1)
55 yfit = func(xfit, pars[0], pars[1])
56 plt.plot(xfit,yfit,'b-')
57 plt.legend(['data', 'fit'],loc='best')
58 plt.savefig('images/nonlin-fit-ci.png')

```

---

```

c0: 4.96713966439 [4.62674476321  5.30753456558]
c1: 2.10995112628 [1.76711622067  2.45278603188]

```

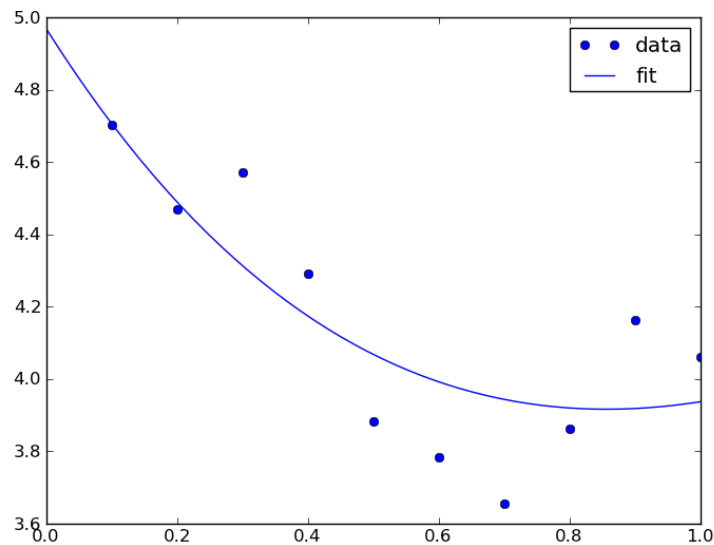


Figure 88: Nonlinear fit to data.

### 9.1.10 Interpolation with splines

When you do not know the functional form of data to fit an equation, you can still fit/interpolate with splines.

---

```

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

```

---

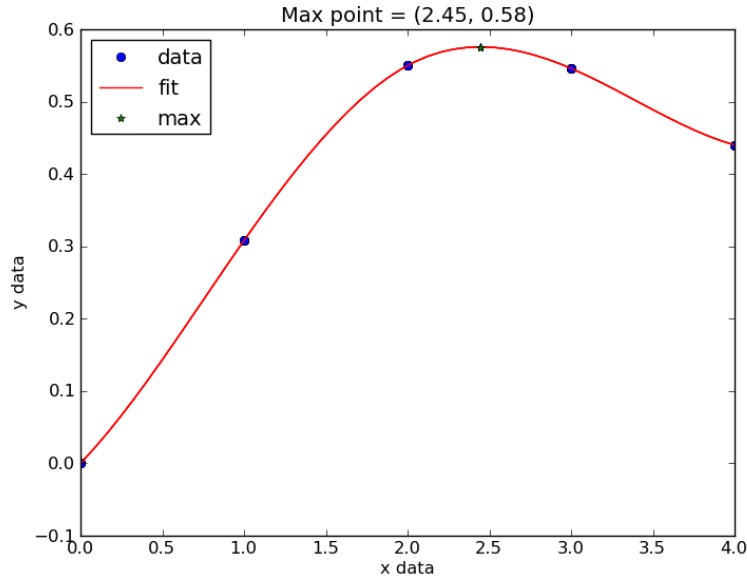


Figure 89: Illustration of a spline fit to data and finding the maximum point.

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

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

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

Solution: Find the cube that contains the point, and is defined by points P1-P8 as shown in Figure 90.

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 I6 is at  $(a,b,z2)$ , so the final interpolation is again a 1D interpolation along z evaluated at  $z=c$  to get the final value of the scalar field at  $P=(a,b,c)$ .

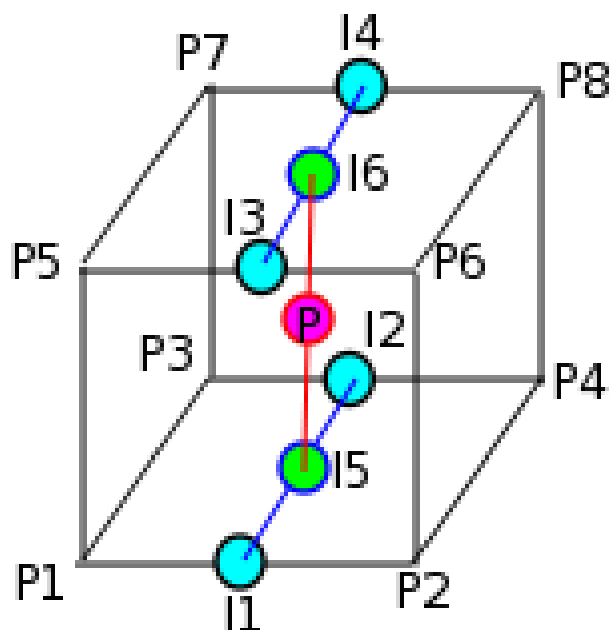


Figure 90: Trilinear interpolation scheme.

```

1  from jasp import jasp
2  import numpy as np
3
4  with jasp('molecules/co-centered') as calc:
5      atoms = calc.get_atoms()
6      x, y, z, cd = calc.get_charge_density()
7
8  def interp3d(x,y,z,cd,xi,yi,zi):
9      """
10     interpolate a cubic 3D grid defined by x,y,z,cd at the point
11     (xi,yi,zi)
12     """
13
14  def get_index(value,vector):
15      """
16     assumes vector ordered decreasing to increasing. A bisection
17     search would be faster.
18     """

```

```

19         for i,val in enumerate(vector):
20             if val > value:
21                 return i-1
22         return None
23
24     xv = x[:,0,0]
25     yv = y[0,:,0]
26     zv = z[0,0,:]
27
28     a,b,c = xi, yi, zi
29
30     i = get_index(a,xv)
31     j = get_index(b,yv)
32     k = get_index(c,zv)
33
34     x1 = x[i,j,k]
35     x2 = x[i+1,j,k]
36     y1 = y[i,j,k]
37     y2 = y[i,j+1,k]
38     z1 = z[i,j,k]
39     z2 = z[i,j,k+1]
40
41     u1 = cd[i, j, k]
42     u2 = cd[i+1, j, k]
43     u3 = cd[i, j+1, k]
44     u4 = cd[i+1, j+1, k]
45     u5 = cd[i, j, k+1]
46     u6 = cd[i+1, j, k+1]
47     u7 = cd[i, j+1, k+1]
48     u8 = cd[i+1, j+1, k+1]
49
50     w1 = u2 + (u2-u1)/(x2-x1)*(a-x2)
51     w2 = u4 + (u4-u3)/(x2-x1)*(a-x2)
52     w3 = w2 + (w2-w1)/(y2-y1)*(b-y2)
53     w4 = u5 + (u6-u5)/(x2-x1)*(a-x1)
54     w5 = u7 + (u8-u7)/(x2-x1)*(a-x1)
55     w6 = w4 + (w5-w4)/(y2-y1)*(b-y1)
56     w7 = w3 + (w6-w3)/(z2-z1)*(c-z1)
57     u = w7
58
59     return u
60
61 pos = atoms.get_positions()
62
63 P1 = np.array([0.0, 5.0, 5.0])
64 P2 = np.array([9.0, 5.0, 5.0])
65
66 npoints = 60
67
68 points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]
69
70 R = [np.linalg.norm(p-P1) for p in points]
71
72 # interpolated line
73 icd = [interp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]
74
75 import matplotlib.pyplot as plt
76
77 plt.plot(R, icd)
78 cR = np.linalg.norm(pos[0] - P1)
79 oR = np.linalg.norm(pos[1] - P1)
80 plt.plot([cR, cR], [0, 2], 'r-') #markers for where the nuclei are
81 plt.plot([oR, oR], [0, 8], 'r-')
82 plt.xlabel('|R| ($\AA$)')
83 plt.ylabel('Charge density (e/$\AA^3$)')
84 plt.savefig('images/C0-charge-density.png')

```

85 `plt.show()`

---

None

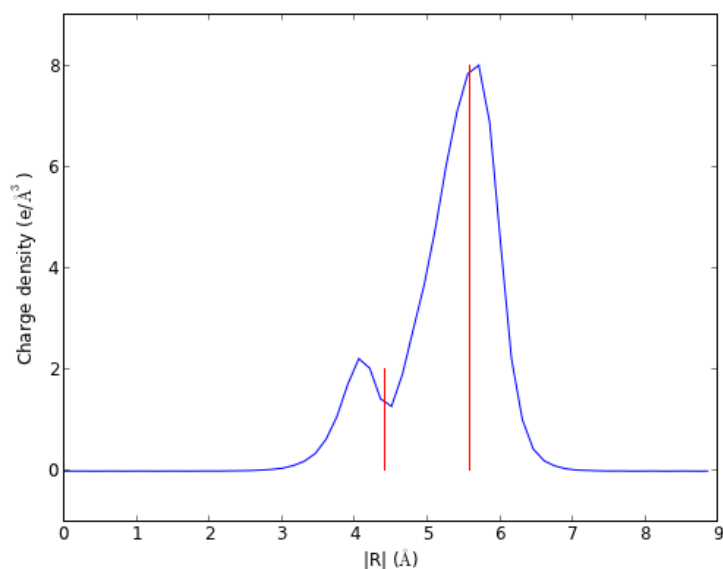


Figure 91: An example of interpolated charge density of a CO molecule along the axis of molecule.

To generalize this to non-cubic cells, we need to do interpolation along arbitrary vectors. The overall strategy is the same:

Find the cell that contains the point  $(a,b,c)$ . compute the scaled coordinates  $(s_a,s_b,s_c)$  of the point inside the cell. Do the interpolations along the basis vectors. Given  $u_1$  at  $P_1(x_1,y_1,z_1)$  and  $u_2$  at  $P_2(x_2,y_2,z_2)$  where  $(P_2-P_1)$  is a cell basis vector  $a$ ,  $u = u_1 + s_a*(u_2-u_1)$ . 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  '''
2  3D vector interpolation in non-cubic unit cells with vector
3  interpolation.
4
5  This function should work for any shape unit cell.
6  '''
7  from jasp import *
8  import bisect
9  import numpy as np
10 from pylab import plot, xlabel, ylabel, savefig, show
11
12 with jasp('molecules/co-centered') as calc:
13     atoms = calc.get_atoms()
14     x,y,z,cd = calc.get_charge_density()
15
16 def vinterp3d(x,y,z,u,xi,yi,zi):
17
```

```

18     p = np.array([xi,yi,zi])
19
20     #1D arrays of coordinates
21     xv = x[:,0,0]
22     yv = y[0,:,0]
23     zv = z[0,0,:]
24
25     # we subtract 1 because bisect tells us where to insert the
26     # element to maintain an ordered list, so we want the index to the
27     # left of that point
28     i = bisect.bisect_right(xv,xi) - 1
29     j = bisect.bisect_right(yv,yi) - 1
30     k = bisect.bisect_right(zv,zi) - 1
31
32     #points at edge of cell. We only need P1, P2, P3, and P5
33     P1 = np.array([x[i,j,k],y[i,j,k],z[i,j,k]])
34     P2 = np.array([x[i+1,j,k],y[i+1,j,k],z[i+1,j,k]])
35     P3 = np.array([x[i,j+1,k],y[i,j+1,k],z[i,j+1,k]])
36     P5 = np.array([x[i,j,k+1],y[i,j,k+1],z[i,j,k+1]])
37
38     #values of u at edge of cell
39     u1 = u[i,j,k]
40     u2 = u[i+1,j,k]
41     u3 = u[i,j+1,k]
42     u4 = u[i+1,j+1,k]
43     u5 = u[i,j,k+1]
44     u6 = u[i+1,j,k+1]
45     u7 = u[i,j+1,k+1]
46     u8 = u[i+1,j+1,k+1]
47
48     #cell basis vectors, not the unit cell, but the voxel cell containing the point
49     cbasis = np.array([P2-P1,
50                        P3-P1,
51                        P5-P1])
52
53     #now get interpolated point in terms of the cell basis
54     s = np.dot(np.linalg.inv(cbasis.T),np.array([xi,yi,zi])-P1)
55
56     #now s = (sa, sb, sc) which are fractional coordinates in the vector space
57     #next we do the interpolations
58     ui1 = u1 + s[0]*(u2-u1)
59     ui2 = u3 + s[0]*(u4-u3)
60
61     ui3 = u5 + s[0]*(u6-u5)
62     ui4 = u7 + s[0]*(u8-u7)
63
64     ui5 = ui1 + s[1]*(ui2-ui1)
65     ui6 = ui3 + s[1]*(ui4-ui3)
66
67     ui7 = ui5 + s[2]*(ui6-ui5)
68
69     return ui7
70
71     # compute a line with 60 points in it through these two points
72     P1 = np.array([0.0, 5.0, 5.0])
73     P2 = np.array([10.0, 5.0, 5.0])
74
75     npoints = 60
76
77     points = [P1 + n*(P2-P1)/npoints for n in range(npoints)]
78
79     # compute the distance along the line
80     R = [np.linalg.norm(p-P1) for p in points]
81
82     icd = [vinterp3d(x,y,z,cd,p[0],p[1],p[2]) for p in points]
83

```

---

```

84 plot(R,icd)
85 pos = atoms.get_positions()
86 cR = np.linalg.norm(pos[0]-P1)
87 oR = np.linalg.norm(pos[1]-P1)
88 plot([cR,cR],[0,2], 'r-') #markers for where the nuclei are
89 plot([oR,oR],[0,8], 'r-')
90 xlabel(' |R| (Å)')
91 ylabel('Charge density (e/Å3)')
92 savefig('images/interpolated-charge-density.png')
93 show()

```

---

None

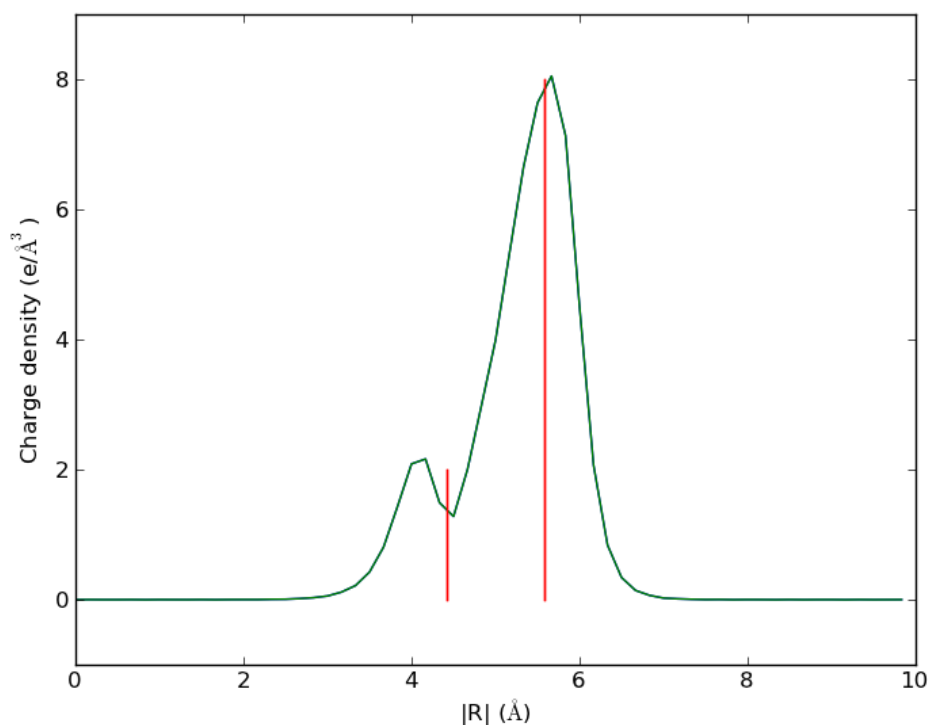


Figure 92: Interpolated charge density for a CO molecule.

### 9.1.12 Reading and writing data

**Built-in io modules** `pylab` has two convenient and powerful functions for saving and reading data, `pylab.save` and `pylab.load`.

---

```

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

```

---

and later you can read these arrays back in with:



---

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

---

see also `pylab.csv2rec` and `pylab.loadtxt` and `pylab.savetxt`.

See <http://www.scipy.org/Cookbook/InputOutput> for examples of numpy io.

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

Let us consider a datafile with these contents:

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

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

---

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

---

Writing datafiles is easy too.

---

```
1 v1 = ['john', 'robert', 'terry']
2 v2 = [4,5,6]
3 f = open('somefile', 'w') #note 'w' = write mode
4 f.write('#header\n')
5 f.write('#ignore these lines\n')
6 for a,b in zip(v1,v2):
7     f.write('{0}, {1}\n'.format(a,b))
8 f.close()
```

---

Some notes:

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

1. when writing to a file you have to add a carriage return to each line.

2. Manually writing and reading files is pretty tedious. Whenever possible you should use the built-in methods of `numpy` or `pylab`.

### 9.1.13 Integration

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

Integration can be used to calculate average properties of continuous distributions. Suppose for example, we have a density of states,  $\rho$  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:

---

```
1 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:

---

```
1 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  $n$ th 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::

---

```
1 n = 2
2 mom_2 = np.trapz(rho*E**n,E)/np.trapz(rho,E)
```

---

### 9.1.14 Numerical differentiation

`numpy` has a function called `numpy.diff` that is similar to the one found in Matlab. It calculates the differences between the elements in your list, and returns a list that is one element shorter, which makes it unsuitable for plotting the derivative of a function.

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

---

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import time
4
5 '''
6 These are the brainless way to calculate numerical derivatives. They
```

```

7  work well for very smooth data. they are surprisingly fast even up to
8  10000 points in the vector.
9  '''
10
11 x = np.linspace(0.78, 0.79, 100) # 100 points between 0.78 and 0.79
12 y = np.sin(x)
13 dy_analytical = np.cos(x)
14 '''
15 let us use a forward difference method:
16 that works up until the last point, where there is not
17 a forward difference to use. there, we use a backward difference.
18 '''
19
20 tf1 = time.time()
21 dyf = [0.0]*len(x)
22 for i in range(len(y)-1):
23     dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
24 #set last element by backwards difference
25 dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
26
27 print ' Forward difference took {0:1.1f} seconds'.format(time.time() - tf1)
28
29 '''and now a backwards difference'''
30 tb1 = time.time()
31 dyb = [0.0]*len(x)
32 #set first element by forward difference
33 dyb[0] = (y[0] - y[1])/(x[0] - x[1])
34 for i in range(1,len(y)):
35     dyb[i] = (y[i] - y[i-1])/(x[i]-x[i-1])
36
37 print ' Backward difference took {0:1.1f} seconds'.format(time.time() - tb1)
38
39 '''and now, a centered formula'''
40 tc1 = time.time()
41 dyc = [0.0]*len(x)
42 dyc[0] = (y[0] - y[1])/(x[0] - x[1])
43 for i in range(1,len(y)-1):
44     dyc[i] = (y[i+1] - y[i-1])/(x[i+1]-x[i-1])
45 dyc[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
46
47 print ' Centered difference took {0:1.1f} seconds'.format(time.time() - tc1)
48
49 '''
50 the centered formula is the most accurate formula here
51 '''
52
53 plt.plot(x,dy_analytical,label='analytical derivative')
54 plt.plot(x,dyf,'--',label='forward')
55 plt.plot(x,dyb,'--',label='backward')
56 plt.plot(x,dyc,'--',label='centered')
57
58 plt.legend(loc='lower left')
59 plt.savefig('images/simple-diffs.png')

```

---

```

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

```

Obviously, all of these evaluations are very fast.

Loops are usually not great for performance. Numpy offers some vectorized methods that allow us to compute derivatives without loops, although this comes at the mental cost of harder to understand syntax:

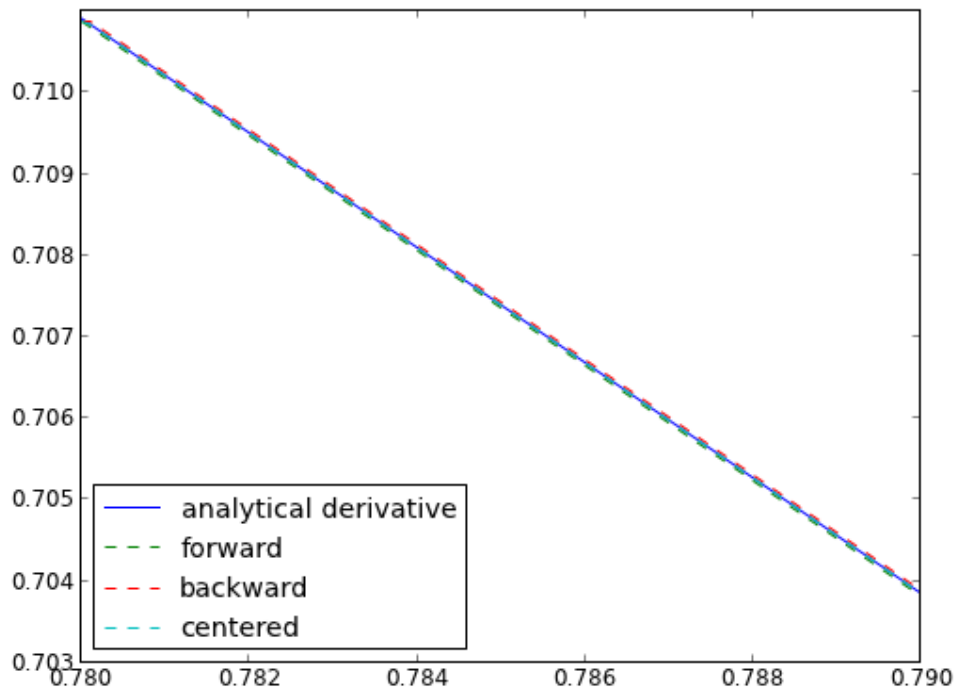


Figure 93: Comparison of different numerical derivatives.

---

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(0,2*np.pi,100)
5  y = np.sin(x)
6  dy_analytical = np.cos(x)
7
8  # we need to specify the size of dy ahead because diff returns
9  #an array of n-1 elements
10 dy = np.zeros(y.shape,np.float) #we know it will be this size
11 dy[0:-1] = np.diff(y)/np.diff(x)
12 dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
13
14
15 '''
16 calculate dy by center differencing using array slices
17 '''
18
19 dy2 = np.zeros(y.shape,np.float) #we know it will be this size
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])
22 dy2[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
23
24 plt.plot(x,y)
25 plt.plot(x,dy_analytical,label='analytical derivative')
26 plt.plot(x,dy,label='forward diff')
27 plt.plot(x,dy2,'k--',lw=2,label='centered diff')
28 plt.legend(loc='lower left')

```

```
29 plt.savefig('images/vectorized-diffs.png')
```

---

None

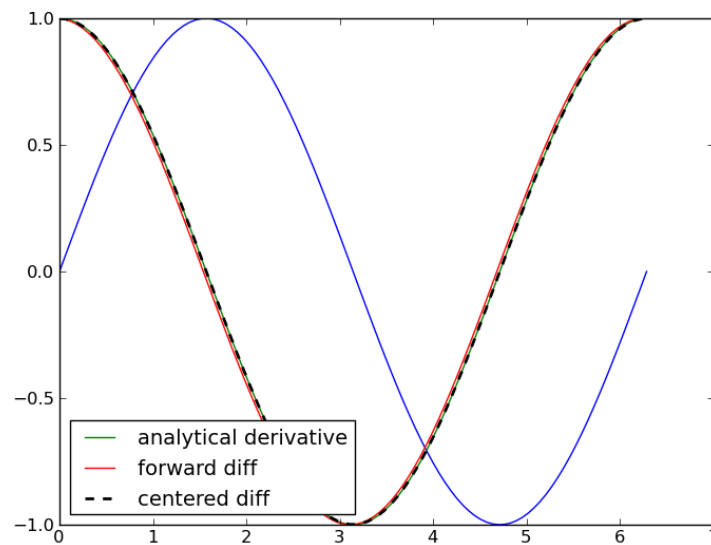


Figure 94: Comparison of different numerical derivatives.

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

Here is an example of a 4-point centered difference of some noisy data:

---

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0,2*np.pi,100)
5 y = np.sin(x) + 0.1*np.random.random(size=x.shape)
6 dy_analytical = np.cos(x)
7
8 #2-point formula
9 dyf = [0.0]*len(x)
10 for i in range(len(y)-1):
11     dyf[i] = (y[i+1] - y[i])/(x[i+1]-x[i])
12 #set last element by backwards difference
13 dyf[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
14
15 '''
16 calculate dy by 4-point center differencing using array slices
17
18  $\frac{y[i-2] - 8y[i-1] + 8y[i+1] - y[i+2]}{12h}$ 
19
20 y[0] and y[1] must be defined by lower order methods
```

```

21 and y[-1] and y[-2] must be defined by lower order methods
22 '''
23
24 dy = np.zeros(y.shape,np.float) #we know it will be this size
25 h = x[1]-x[0] #this assumes the points are evenly spaced!
26 dy[2:-2] = (y[0:-4] - 8*y[1:-3] + 8*y[3:-1] - y[4:])/ (12.*h)
27
28 dy[0] = (y[1]-y[0])/(x[1]-x[0])
29 dy[1] = (y[2]-y[1])/(x[2]-x[1])
30 dy[-2] = (y[-2] - y[-3])/(x[-2] - x[-3])
31 dy[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])
32
33 plt.plot(x,y)
34 plt.plot(x,dy_analytical,label='analytical derivative')
35 plt.plot(x,dyf,'r-',label='2pt-forward diff')
36 plt.plot(x,dy,'k--',lw=2,label='4pt-centered diff')
37 plt.legend(loc='lower left')
38 plt.savefig('images/multipt-diff.png')

```

---

None

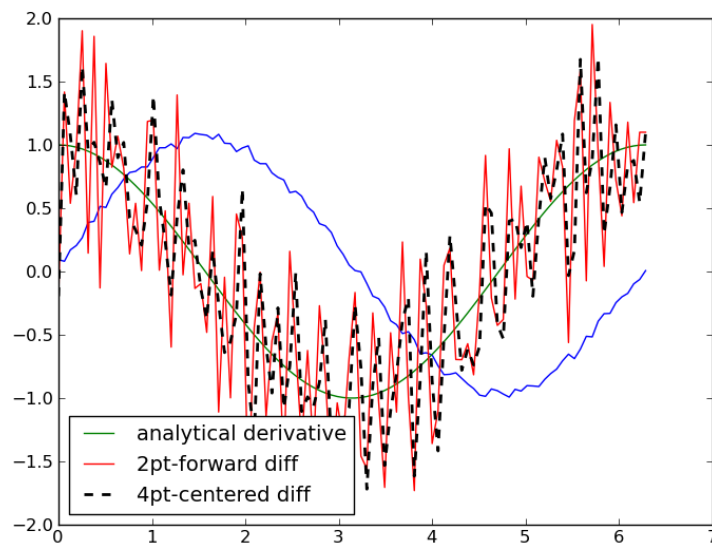


Figure 95: Comparison of 2 point and 4 point numerical derivatives.

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

**FFT derivatives** It is possible to perform derivatives using fast fourier transforms (FFT):

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 N = 101 #number of points
5 L = 2*np.pi #interval of data
6

```

```

7  x = np.arange(0.0,L,L/float(N)) #this does not include the endpoint
8
9  #add some random noise
10 y = np.sin(x) + 0.05*np.random.random(size=x.shape)
11 dy_analytical = np.cos(x)
12
13 '''
14 http://sci.tech-archive.net/Archive/sci.math/2008-05/msg00401.html
15
16 you can use fft to calculate derivatives!
17 '''
18
19 if N % 2 == 0:
20     k = np.asarray(range(0,N/2)+[0] + range(-N/2+1,0))
21 else:
22     k = np.asarray(range(0,(N-1)/2) + [0] + range(-(N-1)/2,0))
23
24 k *= 2*np.pi/L
25
26 fd = np.fft.ifft(1.j*k * np.fft.fft(y))
27
28 plt.plot(x,y)
29 plt.plot(x,dy_analytical,label='analytical der')
30 plt.plot(x,fd,label='fft der')
31 plt.legend(loc='lower left')
32
33 plt.savefig('images/fft-der.png')

```

---

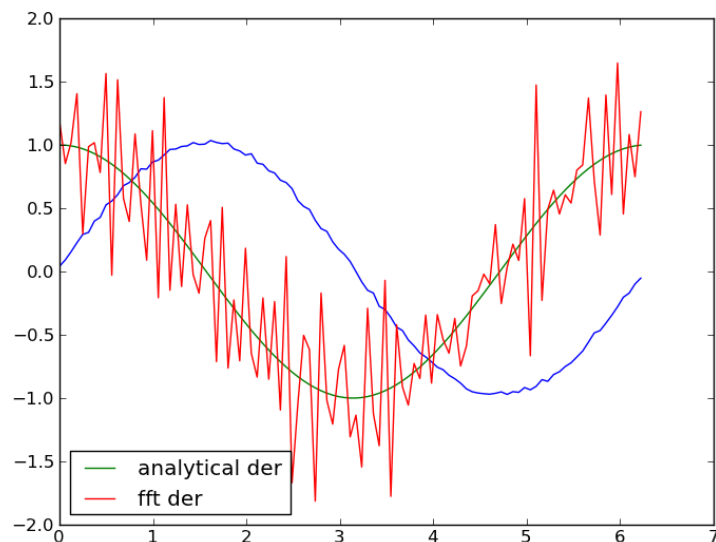


Figure 96: Comparison of FFT numerical derivatives.

This example does not show any major advantage in the quality of the derivative, and it is almost certain I would never remember how to do this off the top of my head.

### 9.1.15 NetCDF files

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

**Scientific.IO.NetCDF** (<http://dirac.cnrs-orleans.fr/plone/software/scientificpython/>)  
**netCDF3** (<http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF3-module.html>)  
**pycdf** (<http://pysclint.sourceforge.net/pycdf/>) this is a very low level module modelled after the C-api. I am not sure it is completely bug-free (I have problems with character variables)

### 9.1.16 Python modules

The comma separated values (**csv**) module in python allows you to easily create datafiles:  
csv writing:

---

```
1 import numpy as np
2
3 x = np.linspace(0.0,6.0,100)
4 y = np.cos(x)
5
6 import csv
7 writer = csv.writer(open("some.csv", "w"))
8 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:

---

```
1 import csv
2 reader = csv.reader(open("some.csv", 'r'), delimiter=',')
3
4 x,y = [],[]
5 for row in reader:
6     #csv returns strings that must be cast as floats
7     a,b = [float(z) for z in row]
8     x.append(a)
9     y.append(b)
```

---

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

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

### 9.1.17 Writing and reading Excel files

**Writing Excel files** It is sometimes convenient to do some analysis in Excel. We can create Excel files in python with **xlwt**. Google this module if you need to do this a lot.

---

```
1 import numpy as np
2 import xlwt
3
```



```

4  wbk = xlwt.Workbook()
5  sheet = wbk.add_sheet('sheet 1')
6
7  volumes = np.array([13.72, 14.83, 16.0, 17.23, 18.52])
8  energies = np.array([-56.29, -56.41, -56.46, -56.46, -56.42])
9
10 for i, pair in enumerate(zip(volumes, energies)):
11     vol = pair[0]
12     energy = pair[1]
13     sheet.write(i,0,vol)
14     sheet.write(i,1,energy)
15 wbk.save('images/test-write.xls')

```

---

**Reading Excel files** We can also read Excel files (even on Linux!) with `xlrd`. Let us read in the data we just wrote. We wrote 5 volumes to column 0, and 5 energies to column 1.

```

1  import xlrd
2  wbk = xlrd.open_workbook('images/test-write.xls')
3  sheet1 = wbk.sheet_by_name('sheet 1')
4  print sheet1.col_values(0)
5  print sheet1.col_values(1)

```

---

```

[13.72, 14.83, 16.0, 17.23, 18.52]
[-56.29, -56.41, -56.46, -56.46, -56.42]

```

### 9.1.18 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\\_Documentation](http://wiki.swftools.org/wiki/Main_Page#SWF_Tools_0.9.2..28_Current_Stable_Version..29_Documentation)

## 9.2 Computational geometry

### 9.2.1 Changing coordinate systems

Let  $A$ ,  $B$ ,  $C$  be the unit cell vectors

$$A = A_1x + A_2y + A_3z \quad (9)$$

$$B = B_1x + B_2y + B_3z \quad (10)$$

$$C = C_1x + C_2y + C_3z \quad (11)$$

and we want to find the vector  $[s_1, s_2, s_3]$  so that  $P = s_1A + s_2B + s_3C$   
if we expand this, we get:

$$\begin{aligned}
 & s_1A_1x + s_1A_2y + s_1A_3z \\
 & + s_2B_1x + s_2B_2y + s_2B_3z \\
 & + s_3C_1x + s_3C_2y + s_3C_3z = p_1x + p_2y + p_3z
 \end{aligned}$$

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

$$\begin{bmatrix} A1 & B1 & C1 \\ A2 & B2 & C2 \\ A3 & B3 & C3 \end{bmatrix} \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} = \begin{bmatrix} p1 \\ p2 \\ p3 \end{bmatrix} \quad (12)$$

or, in standard form:

$$A^T s = p$$

and we need to solve for s as:

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

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

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

Here we implement that in code:

---

```

1  from ase.lattice.surface import fcc111
2  import numpy as np
3  np.set_printoptions(precision=3,suppress=True)
4
5  slab = fcc111('Pd',
6               a=3.92,          # Pd lattice constant
7               size=(2,2,3),    # 3-layer slab in 1x1 configuration
8               vacuum=10.0)
9
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 newbasis = slab.get_cell()
14
15 s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
16 print 'Coordinates in new basis are: \n',s
17
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 basis are:

```

[[ 0.167  0.167  0.408]
 [ 0.667  0.167  0.408]
 [ 0.167  0.667  0.408]
 [ 0.667  0.667  0.408]
 [-0.167  0.333  0.5  ]
 [ 0.333  0.333  0.5  ]
 [-0.167  0.833  0.5  ]
 [ 0.333  0.833  0.5  ]
 [ 0.     0.     0.592]
 [ 0.5    0.     0.592]
 [ 0.     0.5    0.592]
 [ 0.5    0.5    0.592]]

```

Scaled coordinates from ase are:

```

[[ 0.167  0.167  0.408]
 [ 0.667  0.167  0.408]
 [ 0.167  0.667  0.408]

```

```

[ 0.667  0.667  0.408]
[ 0.833  0.333  0.5  ]
[ 0.333  0.333  0.5  ]
[ 0.833  0.833  0.5  ]
[ 0.333  0.833  0.5  ]
[ 0.      0.      0.592]
[ 0.5     0.      0.592]
[ 0.      0.5     0.592]
[ 0.5     0.5     0.592]]

```

The method shown above is general to all basis set transformations. We examine another case next. Sometimes it is nice if all the coordinates are integers. For this example, we will use the bcc primitive lattice vectors and express the positions of each atom in terms of them. By definition each atomic position should be an integer combination of the primitive lattice vectors (before relaxation, and assuming one atom is at the origin, and the unit cell is aligned with the primitive basis!)

---

```

1  from ase.lattice.cubic import BodyCenteredCubic
2  import numpy as np
3  bulk = BodyCenteredCubic(directions=[[1,0,0],
4                                     [0,1,0],
5                                     [0,0,1]],
6                             size=(2,2,2),
7                             latticeconstant=2.87,
8                             symbol='Fe')
9
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 pos = bulk.get_positions()
16
17 s = np.dot(np.linalg.inv(newbasis.T),pos.T).T
18 print 'atom positions in primitive basis'
19 print s
20
21 #let us see the unit cell in terms of the primitive basis too
22 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.]
 [ 0.  1.  1.]
 [ 1.  2.  2.]
 [ 1.  0.  1.]
 [ 2.  1.  2.]
 [ 1.  1.  2.]
 [ 2.  2.  3.]
 [ 1.  1.  0.]
 [ 2.  2.  1.]
 [ 1.  2.  1.]

```

```

[ 2.  3.  2.]
[ 2.  1.  1.]
[ 3.  2.  2.]
[ 2.  2.  2.]
[ 3.  3.  3.]]
unit cell in terms of the primitive basis
[[ 0.  2.  2.]
 [ 2.  0.  2.]
 [ 2.  2.  0.]]

```

## 9.2.2 Simple distances, angles

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

---

```

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

```

---

## 9.2.3 Unit cell properties

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

$$V = |\det(ucell)|$$

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

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

---

```

1  import numpy as np
2
3  a1 = [2, 0, 0]
4  a2 = [1, 1, 0]
5  a3 = [0, 0, 10]

```

---

```

6
7 uc = np.array([a1, a2, a3])
8
9 print 'V = {0} ang^3 from dot/cross'.format(np.dot(np.cross(a1,a2),a3))
10 print 'V = {0} ang^3 from det'.format(np.linalg.det(uc))
11
12 from ase import Atoms
13
14 atoms = Atoms([],cell=uc) #empty list of atoms
15 print 'V = {0} ang^3 from get_volume'.format(atoms.get_volume())

```

---

```

V = 20 ang^3 from dot/cross
V = 20.0 ang^3 from det
V = 20.0 ang^3 from get_volume

```

### 9.2.4 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 (which is the spacing between parallel planes of atoms) for the hkl plane you are using. The script below shows several ways to accomplish that.

---

```

1 import numpy as np
2 from ase.lattice.cubic import FaceCenteredCubic
3
4 ag = FaceCenteredCubic(directions=[[1,0,0],
5                                   [0,1,0],
6                                   [0,0,1]],
7                         size=(1,1,1),
8                         symbol='Ag',
9                         latticeconstant=4.0)
10
11 # these are the reciprocal lattice vectors
12 b1,b2,b3 = np.linalg.inv(ag.get_cell())
13
14 '''
15 g(111) = 1*b1 + 1*b2 + 1*b3
16
17 and |g(111)| = 1/d_111
18 '''
19 h,k,l = (1,1,1)
20 d = 1./np.linalg.norm(h*b1 + k*b2 + l*b3)
21
22 print 'd_111 spacing (method 1) = {0:1.3f} Angstroms'.format(d)
23
24 #method #2
25 hkl = np.array([h,k,l])
26 G = np.array([b1,b2,b3]) #reciprocal unit cell
27
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)],
33                  [dot(b1,b3), dot(b2,b3), dot(b3,b3)]])
34
35 but I prefer the notationally more compact:
36 Gstar = G .dot. transpose(G)
37
38 then, 1/d_hkl^2 = hkl .dot. Gstar .dot. hkl
39 '''
40

```

```

41  Gstar = np.dot(G,G.T)
42
43  id2 = np.dot(hkl,np.dot(Gstar,hkl))
44
45  print 'd_111 spacing (method 2) =',np.sqrt(1/id2)
46
47  # http://books.google.com/books?id=nJHSqEseuIUC&pg=PA118&ots=YA9TBldoVH
48  # &dq=reciprocal%20metric%20tensor&pg=PA119#v=onepage
49  # &q=reciprocal%20metric%20tensor&f=false
50
51  '''Finally, many text books on crystallography use long algebraic
52  formulas for computing the d-spacing with sin and cos, vector lengths,
53  and angles. Below we compute these and use them in the general
54  triclinic structure formula which applies to all the structures.
55  '''
56  from Scientific.Geometry import Vector
57  import math
58
59  unitcell = ag.get_cell()
60  A = Vector(unitcell[0])
61  B = Vector(unitcell[1])
62  C = Vector(unitcell[2])
63
64  # lengths of the vectors
65  a = A.length()*angstroms2bohr
66  b = B.length()*angstroms2bohr
67  c = C.length()*angstroms2bohr
68
69  # angles between the vectors in radians
70  alpha = B.angle(C)
71  beta = A.angle(C)
72  gamma = A.angle(B)
73
74  print
75  print 'a   b   c   alpha beta gamma'
76  print '{0:1.3f} {1:1.3f} {2:1.3f} {3:1.3f} {4:1.3f} {5:1.3f}\n'.format(a,b,c,
77  alpha,beta,gamma)
78
79  h,k,l = (1,1,1)
80
81  from math import sin, cos
82
83  id2 = ((h**2/a**2*sin(alpha)**2
84  + k**2/b**2*sin(beta)**2
85  + l**2/c**2*sin(gamma)**2
86  +2*k*l/b/c*(cos(beta)*cos(gamma)-cos(alpha))
87  +2*h*l/a/c*(cos(alpha)*cos(gamma)-cos(beta))
88  +2*h*k/a/b*(cos(alpha)*cos(beta)-cos(gamma)))
89  /(1-cos(alpha)**2-cos(beta)**2 - cos(gamma)**2
90  +2*cos(alpha)*cos(beta)*cos(gamma)))
91
92  d = 1/math.sqrt(id2)
93
94  print 'd_111 spacing (method 3) = {0}'.format(d)

```

---

```

a   b   c   alpha beta gamma
4.000 4.000 4.000 1.571 1.571 1.571

```

```

d_111 spacing (method 3) = 2.30940107676

```

### 9.3 Equations of State

The module `ase.util.eos` uses a simple polynomial equation of state to find bulk unit cell equilibrium volumes and bulk modulus. There are several other choices you could use that are more standard in the literature. Here we summarize them and provide references to the relevant literature. In each of these cases we show equations for the energy as a function of volume, although sometimes the volume is transformed or normalized.

#### 9.3.1 Birch-Murnaghan

This is probably the most common equation of state used most often, and is a modification of the original Murnaghan EOS described below. A current description of the equation is in reference.<sup>94</sup> You can also find the equations for the Vinet and Poirier-Tarantola equations of state in that reference.

Birch-Murnaghan EOS:

$$E(\eta) = E_0 + \frac{9B_0V_0}{16}(\eta^2 - 1)^2(6 + B'_0(\eta^2 - 1) - 4\eta^2)$$

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

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

$$E(\eta) = E_0 + \frac{2B_0V_0}{(B'_0-1)^2}(2 - (5 + 3B'_0(\eta - 1)e^{-3(B'_0-1)(\eta-1)/2}))$$

and the Poirier-Tarantola EOS:

$$E(\varrho) = E_0 + \frac{B_0V_0\varrho^2}{6}(3 + \varrho(B'_0 - 2))$$

with  $\varrho = -3\ln(\eta)$ .

#### 9.3.2 Murnaghan

The equation most often used in the Murnaghan<sup>95</sup> equation of state is described in.<sup>96</sup>

$$E = E_T + \frac{B_0V}{B'_0} \left[ \frac{(V_0/V)^{B'_0}}{B'_0-1} + 1 \right] - \frac{V_0B_0}{B'_0-1}$$

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

#### 9.3.3 Birch

The original Birch equation<sup>97</sup> is:

$$E = E_0 + \frac{9}{8}B_0V_0 \left( \left( \frac{V_0}{V} \right)^{\frac{2}{3}} - 1 \right)^2 + \frac{9}{16}B_0V_0(B'_0 - 4) \left( \left( \frac{V_0}{V} \right)^{\frac{2}{3}} - 1 \right)^3$$

#### 9.3.4 The Anton-Schmidt Equation of state<sup>1</sup>

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

where  $E_\infty$  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 typically about -2.

### 9.3.5 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:

---

```
1  '''Example of fitting the Birch-Murnaghan EOS to data'''
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.optimize import leastsq
6
7  # raw data from 2.2.3-al-analyze-eos.py
8  v = np.array([13.72, 14.83, 16.0, 17.23, 18.52])
9  e = np.array([-56.29, -56.41, -56.46, -56.46, -56.42])
10
11 #make a vector to evaluate fits on with a lot of points so it looks smooth
12 vfit = np.linspace(min(v),max(v),100)
13
14 ### fit a parabola to the data
15 # y = ax^2 + bx + c
16 a,b,c = np.polyfit(v,e,2) #this is from pylab
17
18 '''
19 the parabola does not fit the data very well, but we can use it to get
20 some analytical guesses for other parameters.
21
22 V0 = minimum energy volume, or where dE/dV=0
23 E = aV^2 + bV + c
24 dE/dV = 2aV + b = 0
25 V0 = -b/2a
26
27 E0 is the minimum energy, which is:
28 E0 = aV0^2 + bV0 + c
29
30 B is equal to V0*d^2E/dV^2, which is just 2a*V0
31
32 and from experience we know Bprime_0 is usually a small number like 4
33 '''
34
35 #now here are our initial guesses.
36 v0 = -b/(2*a)
37 e0 = a*v0**2 + b*v0 + c
38 b0 = 2*a*v0
39 bP = 4
40
41 #now we have to create the equation of state function
42 def Murnaghan(parameters,vol):
43     '''
44     given a vector of parameters and volumes, return a vector of energies.
45     equation From PRB 28,5480 (1983)
46     '''
47     E0 = parameters[0]
48     B0 = parameters[1]
```



```

49     BP = parameters[2]
50     V0 = parameters[3]
51
52     E = E0 + B0*vol/BP*((V0/vol)**BP)/(BP-1)+1) - V0*B0/(BP-1.)
53
54     return E
55
56     # and we define an objective function that will be minimized
57     def objective(pars,y,x):
58         #we will minimize this function
59         err = y - Murnaghan(pars,x)
60         return err
61
62     x0 = [e0, b0, bP, v0] #initial guesses in the same order used in the Murnaghan function
63
64     murnpars, ier = leastsq(objective, x0, args=(e,v)) #this is from scipy
65
66     #now we make a figure summarizing the results
67     plt.plot(v,e,'ro')
68     plt.plot(vfit, a*vfit**2 + b*vfit + c,'--',label='parabolic fit')
69     plt.plot(vfit, Murnaghan(murnpars,vfit), label='Murnaghan fit')
70     plt.xlabel('Volume ($\AA^3$)')
71     plt.ylabel('Energy (eV)')
72     plt.legend(loc='best')
73
74     #add some text to the figure in figure coordinates
75     ax = plt.gca()
76     plt.text(0.4, 0.5, 'Min volume = {0:1.2f} $\AA^3$'.format(murnpars[3]),
77             transform = ax.transAxes)
78     plt.text(0.4, 0.4, 'Bulk modulus = {0:1.2f} eV/$\AA^3$ = {1:1.2f} GPa'.format(murnpars[1],
79                                     murnpars[1]*160.21773),
80             transform = ax.transAxes)
81     plt.savefig('images/a-eos.png')
82
83     np.set_printoptions(precision=3)
84     print 'initial guesses  : ', np.array(x0) #array for easy printing
85     print 'fitted parameters: ', murnpars

```

---

```

initial guesses  :  [-56.472   0.631   4.    16.79 ]
fitted parameters:  [-56.466   0.49   4.753  16.573]

```

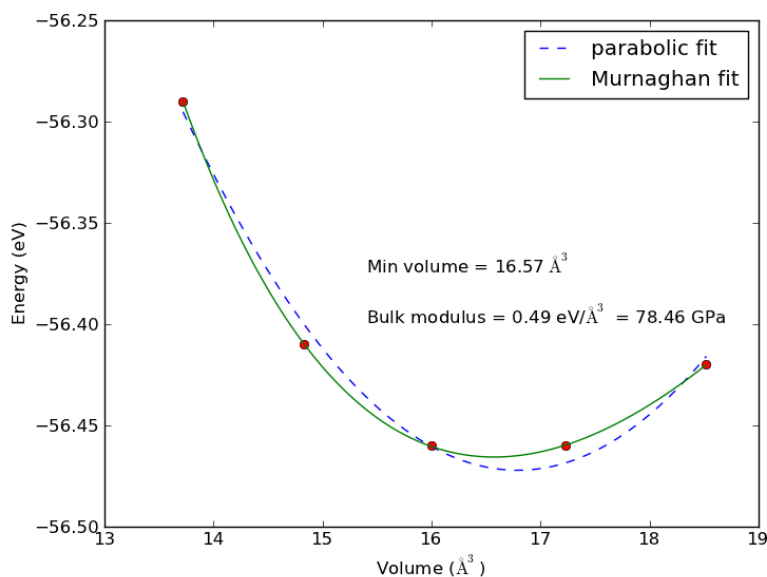


Figure 97: Fitted equation of state for bulk data. The initial fitted parabola is shown to illustrate how it is useful for making initial guesses of the minimum and bulk modulus.

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.

## 9.4 Miscellaneous jasp/VASP tips

### 9.4.1 Installing jasp

You need to create an executable script named `runvasp.py` on your executable path. Here is an example script that works for both serial and parallel versions of VASP. This script is located in `jasp/bin`.

---

```

1  #!/usr/bin/env python
2  import os
3
4  serial_vasp = '/home/jkitchin/src/vasp/bin/vasp_serial_intel_mkl'
5  parallel_vasp = '/home/jkitchin/src/vasp/bin/vasp_openmpi_intel_mkl'
6
7  if 'PBS_NODEFILE' in os.environ:
8      NPROCS = len(open(os.environ['PBS_NODEFILE']).readlines())
9
10     if NPROCS == 1:
11         print 'NPROCS = ',NPROCS
12         exitcode = os.system(serial_vasp)
13     else:
14         print 'NPROCS = ',NPROCS
15         parcmd = 'mpirun -np %i %s' % (NPROCS,parallel_vasp)
16         exitcode = os.system(parcmd)
17 else:

```

```

18     exitcode = os.system(serial_vasp)
19 #end

```

---

Then create an environment variable that points to this script. For example in `.bashrc`:

```

1 export VASP_SCRIPT=$HOME/kitchinpython/jasp/bin/run_vasp.py

```

---

`jasp` expects to find the POTCAR potentials in specific directories. You have two options:

1. Install the potentials in a directory and name the potential directories like this:

```

my_vasp_potentials
  potpaw
  potpaw_GGA
  potpaw_PBE

```

The `potpaw` directory contains the LDA potentials, `potpaw_GGA` contains PW91 potentials, and `potpaw_PBE` contains the PBE potentials. Then, in your `.bashrc` file set an environment variable that points to this directory. For example:

```

1 export VASP_PP_PATH=$HOME/vasp/my_vasp_potentials

```

---

If you already have potential directories setup with different names, you can always make symlinks with the names above that point to your directories.

### 9.4.2 Using a special setup

VASP provides [special setups](#) for some elements. The following guidelines tell you what is in a potential:

No extension means the standard potential. The following extensions mean:

Table 7: Meaning of extensions on POTCAR files for special setups.

extension	
<code>_h</code>	means the potential is harder than the standard (i.e. needs a higher cutoff energy)
<code>_s</code>	means the potential is softer than the standard (i.e. needs a lower cutoff energy)
<code>_sv</code>	<i>s</i> and <i>p</i> semi-core states are treated as valence states
<code>_pv</code>	<i>p</i> semi-core states are treated as valence states
<code>_d</code>	<i>d</i> semi-core states are treated as valence states

Here are some links to information in the VASP manual for the setups.

- [1st row elements](#)
- [Alkali and alkali-earth metals](#)

- d-elements
- p-elements
- f-elements

Here we show how to select the O<sub>sv</sub> potential in a calculation.

---

```

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

```

---

Total energy = -1578.61345 eV

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

---

```

1  head -n 1 molecules/O_sv/POTCAR

```

---

PAW\_PBE O\_sv 05Jul2007

This shows we indeed used the O<sub>sv</sub> setup.

### 9.4.3 Running jasp in parallel

jasp is smart. If you ask for more than one node, it will automatically try to run in parallel.

---

```

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

```

---

None

How do you know it ran on four nodes?

---

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

---

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

```
executed on          LinuxIFC date 2012.08.31  09:10:10
running on    4 nodes
distr:  one band on    1 nodes,    4 groups
```

---

#### 9.4.4 Running multiple instances of jasp in parallel

`jasp` was designed to enable asynchronous, parallel running processes through a queuing system. This is ideal for submitting large numbers of independent calculations in one script. The design uses exceptions to exit the script if the results are not available for subsequent analysis. The design expects that you run the script often, and the results are analyzed only when they are finally available.

Sometimes it is convenient to run a set of calculations and then wait for them to finish so that a second set of calculations that depend on the first results can be run. In this scenario, it is inconvenient to have to rerun your script again after the first set of calculations is done. The challenge is how to tell the computer to run a set of calculations in parallel, **and** wait for the calculations to finish. This can be achieved using the `multiprocessing` module in python.

The principle idea is to set up the calculations you want to run, and use `multiprocessing` to handle running them and waiting for you. To do this, you must instruct `jasp` to use a “run mode”, and construct a script with a function that runs a calculation, and a section that only runs in the “`__main__`” script.

---

```
1 import multiprocessing
2 from jasp import *
3 from ase import Atom, Atoms
4 from ase.utils.eos import EquationOfState
5 import numpy as np
6
7 JASPRC['mode'] = 'run'
8
9 # this is the function that runs a calculation
10 def do_calculation(calculator):
11     'function to run a calculation through multiprocessing'
12     with calculator as calc:
13         atoms = calc.get_atoms()
14         e = atoms.get_potential_energy()
15         v = atoms.get_volume()
16     return v, e
17
18 # this only runs in the main script, not in processes on other cores
19 if __name__ == '__main__':
20     NCORES = 6 # number of cores to run processes on
21
```

---

```

22     # setup an atoms object
23     a = 3.6
24     atoms = Atoms([Atom('Cu',(0, 0, 0))],
25                   cell=0.5 * a*np.array([[1.0, 1.0, 0.0],
26                                           [0.0, 1.0, 1.0],
27                                           [1.0, 0.0, 1.0]]))
28     v0 = atoms.get_volume()
29
30     # Step 1
31     COUNTER = 0
32     calculators = [] # list of calculators to be run
33     factors = [-0.1, 0.05, 0.0, 0.05, 0.1]
34     for f in factors:
35         newatoms = atoms.copy()
36         newatoms.set_volume(v0*(1 + f))
37         label = 'bulk/cu-mp/step1-{0}'.format(COUNTER)
38         COUNTER += 1
39
40         calc = jasp(label,
41                   xc='PBE',
42                   encut=350,
43                   kpts=(6,6,6),
44                   isym=2,
45                   atoms=newatoms)
46
47         calculators.append(calc)
48
49     # now we set up the Pool of processes
50     pool = multiprocessing.Pool(processes=NCORES)
51
52     # get the output from running each calculation
53     out = pool.map(do_calculation, calculators)
54     pool.close()
55     pool.join() # this makes the script wait here until all jobs are done
56
57     # now proceed with analysis
58     V = [x[0] for x in out]
59     E = [x[1] for x in out]
60
61     eos = EquationOfState(V, E)
62     v1, e1, B = eos.fit()
63     print 'step1: v1 = {v1}'.format(**locals())
64
65     ### #####
66     ## STEP 2, eos around the minimum
67     ## #####
68     factors = [-0.06, -0.04, -0.02,
69               0.0,
70               0.02, 0.04, 0.06]
71
72     calculators = [] # reset list
73     for f in factors:
74         newatoms = atoms.copy()
75         newatoms.set_volume(v1*(1 + f))
76         label = 'bulk/cu-mp/step2-{0}'.format(COUNTER)
77         COUNTER += 1
78
79         calc = jasp(label,
80                   xc='PBE',
81                   encut=350,
82                   kpts=(6,6,6),
83                   isym=2,
84                   atoms=newatoms)
85         calculators.append(calc)
86
87     pool = multiprocessing.Pool(processes=NCORES)

```

```

88
89 out = pool.map(do_calculation, calculators)
90 pool.close()
91 pool.join() # wait here for calculations to finish
92
93 # proceed with analysis
94 V += [x[0] for x in out]
95 E += [x[1] for x in out]
96
97 V = np.array(V)
98 E = np.array(E)
99
100 f = np.array(V)/v1
101
102 # only take points within +- 10% of the minimum
103 ind = (f >=0.90) & (f <= 1.1)
104
105 eos = EquationOfState(V[ind], E[ind])
106 v2, e2, B = eos.fit()
107 print 'step2: v2 = {v2}'.format(**locals())
108 eos.plot('images/cu-mp-eos.png')

```

---

step1: v1 = 12.0218897111

step2: v2 = 12.0216094217

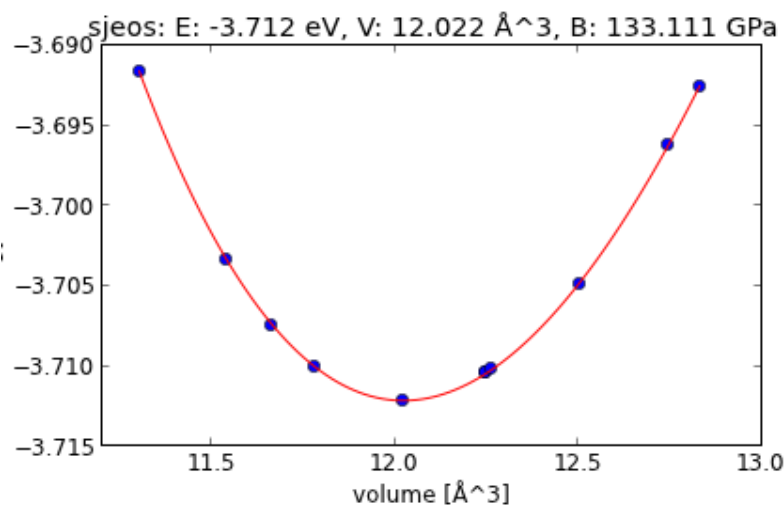


Figure 98: Equation of state for Cu using the multiprocessing module.

**Note:**

The first time you run this you will get all the VASP output. The second time you get the smaller output above.

Also, I have not figured out how to integrate this method with the queue system. At the moment, the runjasp.py script which ultimately runs VASP will run VASP in parallel, i.e. one process on multiple nodes/cores instead of a single job that runs multiple processes simultaneously on multiple nodes/cores.

### 9.4.5 Exporting data json, xml, python, sqlite

jasp has some capability for representing a calculation result in an archival format. The formats currently under development are json, xml, python and sqlite. The main point of these methods is to make it easy to create archive files that are machine readable for supplementary information in publications. These are under development.

**python** This is code that should reconstruct the python code needed to run a particular calculation. There are some limitations, e.g. it does not currently get magnetic moments on the atoms.

---

```
1 from jasp import *
2
3 with jasp('bulk/alloy/cu') as calc:
4     print calc.python
```

---

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

atoms = Atoms([Atom('Cu',[0.0, 0.0, 0.0]),
               cell = [[1.818, 0.0, 1.818],
                       [1.818, 1.818, 0.0],
                       [0.0, 1.818, 1.818]])
```

```
with jasp('bulk/alloy/cu',
          nbands = 9,
          nsw = 10,
          ibrion = 2,
          isif = 4,
          encut = 350.0,
          prec = 'Normal',
          kpts = array([13, 13, 13]),
          reciprocal = False,
          xc = 'PBE',
          txt = '-',
          gamma = False,
          atoms=atoms) as calc:
    # your code here
```

#### json

---

```
1 from jasp import *
2
3 with jasp('bulk/alloy/cu') as calc:
4     print calc.pretty_json
```

---

```
{
```



```

"INCAR": {
  "addgrid": null,
  "aexx": null,
  "aggac": null,
  "aggax": null,
  "aldac": null,
  "algo": null,
  "amin": null,
  "amix": null,
  "amix_mag": null,
  "bmix": null,
  "bmix_mag": null,
  "ddr": null,
  "deper": null,
  "dfnmax": null,
  "dfnmin": null,
  "dipol": null,
  "drotmax": null,
  "ebreak": null,
  "ediff": null,
  "ediffg": null,
  "eint": null,
  "emax": null,
  "emin": null,
  "enaug": null,
  "encut": 350.0,
  "encutfock": null,
  "encutgw": null,
  "falpha": null,
  "falphadec": null,
  "fdstep": null,
  "ferdo": null,
  "ferwe": null,
  "fnmin": null,
  "ftimedec": null,
  "ftimeinc": null,
  "ftimemax": null,
  "gga": null,
  "hfscreen": null,
  "ialgo": null,
  "iband": null,
  "ibrion": 2,
  "ichain": null,
  "icharg": null,
  "idipol": null,
  "images": null,

```

"iniwav": null,  
"invcurve": null,  
"iopt": null,  
"isif": 4,  
"ismear": null,  
"ispin": null,  
"istart": null,  
"isym": null,  
"iwavpr": null,  
"jacobian": null,  
"kgamma": null,  
"kpuse": null,  
"kspacing": null,  
"laechg": null,  
"lasph": null,  
"lasync": null,  
"lbfgsmem": null,  
"lcharg": null,  
"lclimb": null,  
"lcorr": null,  
"ldau": null,  
"ldau\_luj": null,  
"ldauj": null,  
"ldaul": null,  
"ldauprint": null,  
"ldautype": null,  
"ldauu": null,  
"ldiag": null,  
"ldipol": null,  
"ldneb": null,  
"lelf": null,  
"lepsilon": null,  
"lglobal": null,  
"lhfcalc": null,  
"llineopt": null,  
"lmaxmix": null,  
"lnebcell": null,  
"loptics": null,  
"lorbit": null,  
"lpard": null,  
"lplane": null,  
"lscalapack": null,  
"lscalu": null,  
"lsepb": null,  
"lsepk": null,  
"ltangentold": null,

"lthomas": null,  
"luse\_vdw": null,  
"lvdw": null,  
"lvhar": null,  
"lvtot": null,  
"lwave": null,  
"magmom": null,  
"maxmix": null,  
"maxmove": null,  
"nbands": 9,  
"nblk": null,  
"nbmod": null,  
"nelect": null,  
"nelm": null,  
"nelmdl": null,  
"nelmin": null,  
"nfree": null,  
"ngx": null,  
"ngxf": null,  
"ngy": null,  
"ngyf": null,  
"ngz": null,  
"ngzf": null,  
"nkred": null,  
"nkredx": null,  
"nkredy": null,  
"nkredz": null,  
"nomega": null,  
"nomegar": null,  
"npar": null,  
"nsim": null,  
"nsw": 10,  
"nupdown": null,  
"nwrite": null,  
"param1": null,  
"param2": null,  
"pomass": null,  
"potim": null,  
"prec": "Normal",  
"precfock": null,  
"ropt": null,  
"rwigs": null,  
"sdalpha": null,  
"sdr": null,  
"sigma": null,  
"smass": null,

```

    "snl": null,
    "spring": null,
    "stol": null,
    "symprec": null,
    "system": null,
    "tebeg": null,
    "teend": null,
    "time": null,
    "timestep": null,
    "vdwgr": null,
    "vdwrn": null,
    "voskown": null,
    "weimin": null,
    "zab_vdw": null,
    "zval": null
  },
  "atoms": {
    "cell": [
      [
        1.818,
        0.0,
        1.818
      ],
      [
        1.818,
        1.818,
        0.0
      ],
      [
        0.0,
        1.818,
        1.818
      ]
    ],
    "pbc": [
      true,
      true,
      true
    ],
    "positions": [
      [
        0.0,
        0.0,
        0.0
      ]
    ]
  },
],

```

```

        "symbols": [
            "Cu"
        ],
        "tags": [
            0
        ]
    },
    "input": {
        "gamma": false,
        "kpts": [
            13,
            13,
            13
        ],
        "kpts_nintersections": null,
        "reciprocal": false,
        "setups": null,
        "txt": "-",
        "xc": "PBE"
    }
}

```

**xml** This relies on the pyxser module.

---

```

1 from jasp import *
2
3 with jasp('bulk/alloy/cu') as calc:
4     print calc.xml

```

---

```

<?xml version="1.0" encoding="utf-8"?>
<pyx:obj xmlns:pyx="http://projects.coder.cl/pyxser/model/" version="1.0" type="vasp" mod
  <pyx:col type="dict" name="d">
    <pyx:col type="dict" name="INCAR">
      <pyx:prop type="str" name="prec">Normal</pyx:prop>
      <pyx:prop type="float" name="encut">350.0</pyx:prop>
      <pyx:prop type="int" name="nbands">9</pyx:prop>
      <pyx:prop type="int" name="isif">4</pyx:prop>
      <pyx:prop type="int" name="nsw">10</pyx:prop>
      <pyx:prop type="int" name="ibrion">2</pyx:prop>
    </pyx:col>
    <pyx:col type="dict" name="input">
      <pyx:col type="list" name="kpts">
        <pyx:prop type="int" name="kpts">13</pyx:prop>
        <pyx:prop type="int" name="kpts">13</pyx:prop>
        <pyx:prop type="int" name="kpts">13</pyx:prop>
      </pyx:col>
      <pyx:prop type="bool" name="reciprocal">False</pyx:prop>

```

```

    <pyx:prop type="str" name="xc">PBE</pyx:prop>
    <pyx:prop type="str" name="txt">-</pyx:prop>
    <pyx:prop type="bool" name="gamma">False</pyx:prop>
  </pyx:col>
  <pyx:col type="dict" name="atoms">
    <pyx:col type="list" name="cell">
      <pyx:col type="list" name="cell">
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
        <pyx:prop type="float" name="cell">0.0</pyx:prop>
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
      </pyx:col>
      <pyx:col type="list" name="cell">
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
        <pyx:prop type="float" name="cell">0.0</pyx:prop>
      </pyx:col>
      <pyx:col type="list" name="cell">
        <pyx:prop type="float" name="cell">0.0</pyx:prop>
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
        <pyx:prop type="float" name="cell">1.818</pyx:prop>
      </pyx:col>
    </pyx:col>
    <pyx:col type="list" name="symbols">
      <pyx:prop type="str" name="symbols">Cu</pyx:prop>
    </pyx:col>
    <pyx:col type="list" name="pbc">
      <pyx:prop type="bool" name="pbc">True</pyx:prop>
      <pyx:prop type="bool" name="pbc">True</pyx:prop>
      <pyx:prop type="bool" name="pbc">True</pyx:prop>
    </pyx:col>
    <pyx:col type="list" name="positions">
      <pyx:col type="list" name="positions">
        <pyx:prop type="float" name="positions">0.0</pyx:prop>
        <pyx:prop type="float" name="positions">0.0</pyx:prop>
        <pyx:prop type="float" name="positions">0.0</pyx:prop>
      </pyx:col>
    </pyx:col>
  </pyx:col>
</pyx:obj>

```

#### 9.4.6 Recommended values for ENCUT and valence electrons for different POT-CAR files

The [ENCUT](#) tag and [PREC](#) tag affect the accuracy/convergence of your calculations.

```

1  from jasp.POTCAR import *
2  from ase.data import chemical_symbols
3  import glob, os
4
5  print '#+tblname: POTCAR'
6  print '#+caption: Parameters for POTCAR files.'
7  print '#+ATTR_LaTeX: longtable'
8  print '| POTCAR | ENMIN (eV) | ENMAX (eV) | ENCUT prec=high (eV) | # valence electrons |'
9  print '|-'
10
11  chemical_symbols.sort()
12  for symbol in chemical_symbols:
13
14      potcars = glob.glob('{0}/POTPAW_PBE/{1}*/POTCAR'.format(os.environ['VASP_PP_PATH'],
15                                                                symbol))
16
17      for potcar in potcars:
18
19          POTCAR = os.path.relpath(potcar,
20                                  os.environ['VASP_PP_PATH'][:-7])
21          ENMIN = get_ENMIN(potcar)
22          ENMAX = get_ENMAX(potcar)
23          HIGH = 1.3*ENMAX
24          ZVAL = get_ZVAL(potcar)
25
26      print '|{POTCAR:30s}|{ENMIN}|{ENMAX}|{HIGH:1.3f}|{ZVAL}|'.format(**locals())

```

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Ac	129.178	172.237	223.908	1
POTPAW_PBE/Ag_new	187.383	249.844	324.797	1
POTPAW_PBE/Ag	187.385	249.846	324.800	1
POTPAW_PBE/Ag_pv	223.399	297.865	387.225	1
POTPAW_PBE/Al	180.225	240.3	312.390	1
POTPAW_PBE/Am	191.906	255.875	332.637	1
POTPAW_PBE/Ar	199.795	266.393	346.311	1
POTPAW_PBE/As_d_GW	259.629	346.172	450.024	1
POTPAW_PBE/As_d	216.488	288.651	375.246	1
POTPAW_PBE/As_GW	156.526	208.702	271.313	1
POTPAW_PBE/As	156.51	208.68	271.284	1
POTPAW_PBE/At_d	199.688	266.251	346.126	1
POTPAW_PBE/At	121.073	161.43	209.859	1
POTPAW_PBE/Au_new	172.457	229.943	298.926	1
POTPAW_PBE/Au	172.461	229.948	298.932	1
POTPAW_PBE/Ba_sv	140.408	187.21	243.373	1
POTPAW_PBE/Be	185.658	247.544	321.807	1
POTPAW_PBE/Be_sv	231.563	308.75	401.375	1
POTPAW_PBE/B_h	500.0	700.0	910.000	1
POTPAW_PBE/Bi_d	182.138	242.851	315.706	1
POTPAW_PBE/Bi	78.777	105.037	136.548	1
POTPAW_PBE/Bi_pv	231.89	309.187	401.943	2
POTPAW_PBE/B	238.954	318.606	414.188	2

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Br	162.198	216.264	281.143	
POTPAW_PBE/B_s	201.934	269.245	350.019	
POTPAW_PBE/Ba_sv	140.408	187.21	243.373	1
POTPAW_PBE/Be	185.658	247.544	321.807	
POTPAW_PBE/Be_sv	231.563	308.75	401.375	
POTPAW_PBE/Bi_d	182.138	242.851	315.706	1
POTPAW_PBE/Bi	78.777	105.037	136.548	
POTPAW_PBE/Bi_pv	231.89	309.187	401.943	2
POTPAW_PBE/Br	162.198	216.264	281.143	
POTPAW_PBE/Ca	77.067	102.755	133.582	
POTPAW_PBE/Ca_pv	89.665	119.554	155.420	
POTPAW_PBE/Ca_sv	199.939	266.586	346.562	1
POTPAW_PBE/C_d	310.494	413.992	538.190	
POTPAW_PBE/Cd	205.757	274.342	356.645	1
POTPAW_PBE/Ce_3	135.964	181.286	235.672	1
POTPAW_PBE/Ce_h	224.925	299.9	389.870	1
POTPAW_PBE/Ce	204.781	273.042	354.955	1
POTPAW_PBE/C_GW	310.494	413.992	538.190	
POTPAW_PBE/C_h_nr	556.263	741.684	964.189	
POTPAW_PBE/C_h	500.0	700.0	910.000	
POTPAW_PBE/Cl_h	306.852	409.136	531.877	
POTPAW_PBE/Cl	196.854	280.0	364.000	
POTPAW_PBE/Co_new	200.976	267.968	348.358	
POTPAW_PBE/Co	200.977	267.969	348.360	
POTPAW_PBE/Co_sv	292.771	390.362	507.471	1
POTPAW_PBE/C	300.0	400.0	520.000	
POTPAW_PBE/Cr	170.311	227.082	295.207	
POTPAW_PBE/Cr_pv_new	199.261	265.681	345.385	1
POTPAW_PBE/Cr_pv	199.262	265.683	345.388	1
POTPAW_PBE/Cr_sv_new	296.603	395.471	514.112	1
POTPAW_PBE/Cr_sv	296.603	395.471	514.112	1
POTPAW_PBE/C_s	205.426	273.901	356.071	
POTPAW_PBE/Cs_sv	165.238	220.318	286.413	
POTPAW_PBE/Cu_f	221.585	295.446	384.080	1
POTPAW_PBE/Cu_new	221.585	295.446	384.080	1
POTPAW_PBE/Cu	204.91	273.214	355.178	1
POTPAW_PBE/Cu_pv_f	276.486	368.648	479.242	1
POTPAW_PBE/Cu_pv	276.454	368.605	479.187	1
POTPAW_PBE/Ca	77.067	102.755	133.582	
POTPAW_PBE/Ca_pv	89.665	119.554	155.420	
POTPAW_PBE/Ca_sv	199.939	266.586	346.562	1
POTPAW_PBE/Cd	205.757	274.342	356.645	1
POTPAW_PBE/Ce_3	135.964	181.286	235.672	1

Continued on next page



Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Ce_h	224.925	299.9	389.870	1
POTPAW_PBE/Ce	204.781	273.042	354.955	1
POTPAW_PBE/Cl_h	306.852	409.136	531.877	
POTPAW_PBE/Cl	196.854	280.0	364.000	
POTPAW_PBE/Co_new	200.976	267.968	348.358	
POTPAW_PBE/Co	200.977	267.969	348.360	
POTPAW_PBE/Co_sv	292.771	390.362	507.471	1
POTPAW_PBE/Cr	170.311	227.082	295.207	
POTPAW_PBE/Cr_pv_new	199.261	265.681	345.385	1
POTPAW_PBE/Cr_pv	199.262	265.683	345.388	1
POTPAW_PBE/Cr_sv_new	296.603	395.471	514.112	1
POTPAW_PBE/Cr_sv	296.603	395.471	514.112	1
POTPAW_PBE/Cs_sv	165.238	220.318	286.413	
POTPAW_PBE/Cu_f	221.585	295.446	384.080	1
POTPAW_PBE/Cu_new	221.585	295.446	384.080	1
POTPAW_PBE/Cu	204.91	273.214	355.178	1
POTPAW_PBE/Cu_pvf	276.486	368.648	479.242	1
POTPAW_PBE/Cu_pv	276.454	368.605	479.187	1
POTPAW_PBE/Dy_3	116.797	155.729	202.448	
POTPAW_PBE/Dy	191.601	255.467	332.107	2
POTPAW_PBE/Er_2	89.813	119.75	155.675	
POTPAW_PBE/Er_3	116.29	155.053	201.569	
POTPAW_PBE/Er	223.587	298.116	387.551	2
POTPAW_PBE/Eu_2	74.478	99.304	129.095	
POTPAW_PBE/Eu_3	96.793	129.057	167.774	
POTPAW_PBE/Eu_GW	452.441	603.254	784.230	1
POTPAW_PBE/Eu	187.251	249.668	324.568	1
POTPAW_PBE/F_d_GW	365.773	487.698	634.007	
POTPAW_PBE/Fe	200.912	267.883	348.248	
POTPAW_PBE/Fe_pv_new	219.928	293.238	381.209	1
POTPAW_PBE/Fe_pv	219.929	293.238	381.209	1
POTPAW_PBE/Fe_sv_h	410.523	547.365	711.575	1
POTPAW_PBE/Fe_sv	292.918	390.558	507.725	1
POTPAW_PBE/F_h	500.0	700.0	910.000	
POTPAW_PBE/F	300.0	400.0	520.000	
POTPAW_PBE/Fr_sv	160.905	214.54	278.902	
POTPAW_PBE/F_s	217.369	289.825	376.772	
POTPAW_PBE/Fe	200.912	267.883	348.248	
POTPAW_PBE/Fe_pv_new	219.928	293.238	381.209	1
POTPAW_PBE/Fe_pv	219.929	293.238	381.209	1
POTPAW_PBE/Fe_sv_h	410.523	547.365	711.575	1
POTPAW_PBE/Fe_sv	292.918	390.558	507.725	1
POTPAW_PBE/Fr_sv	160.905	214.54	278.902	

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Ga_d_GW	277.386	369.848	480.802	1
POTPAW_PBE/Ga_d	212.022	282.697	367.506	1
POTPAW_PBE/Ga_h	303.451	404.601	525.981	1
POTPAW_PBE/Ga	101.009	134.678	175.081	
POTPAW_PBE/Ga_s	62.877	83.836	108.987	
POTPAW_PBE/Ga_sv_GW	377.564	503.418	654.443	2
POTPAW_PBE/Gd_3	115.761	154.348	200.652	
POTPAW_PBE/Gd	192.354	256.472	333.414	1
POTPAW_PBE/Ge_d3	169.58	226.106	293.938	1
POTPAW_PBE/Ge_d_GW2	254.348	339.13	440.869	1
POTPAW_PBE/Ge_d_GW	232.72	310.294	403.382	1
POTPAW_PBE/Ge_d_GW_ref	429.008	572.01	743.613	1
POTPAW_PBE/Ge_d	232.72	310.294	403.382	1
POTPAW_PBE/Ge_h	307.818	410.425	533.553	1
POTPAW_PBE/Ge	130.355	173.807	225.949	
POTPAW_PBE/H1.25	200.0	250.0	325.000	1
POTPAW_PBE/H1.5	200.0	250.0	325.000	
POTPAW_PBE/H.5	200.0	250.0	325.000	
POTPAW_PBE/H.75	200.0	250.0	325.000	0
POTPAW_PBE/He	359.172	478.896	622.565	
POTPAW_PBE/Hf	165.25	220.333	286.433	
POTPAW_PBE/Hf_pv	165.256	220.342	286.445	1
POTPAW_PBE/Hf_sv_GW	238.045	317.394	412.612	1
POTPAW_PBE/Hf_sv	178.083	237.444	308.677	1
POTPAW_PBE/Hg	174.911	233.214	303.178	1
POTPAW_PBE/H.h	500.0	700.0	910.000	
POTPAW_PBE/Ho_3	115.615	154.153	200.399	
POTPAW_PBE/Ho	192.876	257.168	334.318	2
POTPAW_PBE/H	200.0	250.0	325.000	
POTPAW_PBE/He	359.172	478.896	622.565	
POTPAW_PBE/Hf	165.25	220.333	286.433	
POTPAW_PBE/Hf_pv	165.256	220.342	286.445	1
POTPAW_PBE/Hf_sv_GW	238.045	317.394	412.612	1
POTPAW_PBE/Hf_sv	178.083	237.444	308.677	1
POTPAW_PBE/Hg	174.911	233.214	303.178	1
POTPAW_PBE/Ho_3	115.615	154.153	200.399	
POTPAW_PBE/Ho	192.876	257.168	334.318	2
POTPAW_PBE/In_d	179.413	239.218	310.983	1
POTPAW_PBE/In	71.951	95.934	124.714	
POTPAW_PBE/I	131.735	175.647	228.341	
POTPAW_PBE/Ir	158.153	210.87	274.131	
POTPAW_PBE/In_d	179.413	239.218	310.983	1
POTPAW_PBE/In	71.951	95.934	124.714	

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Ir	158.153	210.87	274.131	
POTPAW_PBE/K_pv	87.548	116.731	151.750	
POTPAW_PBE/Kr	138.945	185.26	240.838	
POTPAW_PBE/K_sv	194.412	259.216	336.981	
POTPAW_PBE/Kr	138.945	185.26	240.838	
POTPAW_PBE/La	164.485	219.313	285.107	1
POTPAW_PBE/La_s	102.414	136.552	177.518	
POTPAW_PBE/Li	100.0	140.0	182.000	
POTPAW_PBE/Li_sv2	416.33	555.106	721.638	
POTPAW_PBE/Li_sv	374.276	499.034	648.744	
POTPAW_PBE/Lu_3	116.257	155.009	201.512	
POTPAW_PBE/Lu	191.771	255.695	332.404	2
POTPAW_PBE/Mg_new	94.607	126.143	163.986	
POTPAW_PBE/Mg	157.509	210.012	273.016	
POTPAW_PBE/Mg_pv_GW	302.947	403.929	525.108	
POTPAW_PBE/Mg_pv.old	199.18	265.574	345.246	
POTPAW_PBE/Mg_pv	302.947	403.929	525.108	
POTPAW_PBE/Mg_sv	371.417	495.223	643.790	1
POTPAW_PBE/Mn	202.399	269.865	350.825	
POTPAW_PBE/Mn_pv_new	202.398	269.864	350.823	1
POTPAW_PBE/Mn_pv	202.399	269.865	350.825	1
POTPAW_PBE/Mn_sv	290.39	387.187	503.343	1
POTPAW_PBE/Mo	168.438	224.584	291.959	
POTPAW_PBE/Mo_pv_new	168.438	224.584	291.959	1
POTPAW_PBE/Mo_pv	168.438	224.584	291.959	1
POTPAW_PBE/Mo_sv	182.007	242.676	315.479	1
POTPAW_PBE/Na	76.476	101.968	132.558	
POTPAW_PBE/Na_pv	194.671	259.561	337.429	
POTPAW_PBE/Na_sv	484.23	700.0	910.000	
POTPAW_PBE/Nb_pv	156.456	208.608	271.190	1
POTPAW_PBE/Nb_sv_new	219.927	293.235	381.206	1
POTPAW_PBE/Nb_sv	219.927	293.235	381.206	1
POTPAW_PBE/Nd_3	136.909	182.546	237.310	1
POTPAW_PBE/Nd	189.892	253.189	329.146	1
POTPAW_PBE/Ne	257.704	343.606	446.688	
POTPAW_PBE/N_h	500.0	700.0	910.000	
POTPAW_PBE/Ni_new	202.149	269.532	350.392	1
POTPAW_PBE/Ni	202.15	269.533	350.393	1
POTPAW_PBE/Ni_pv	275.959	367.945	478.329	1
POTPAW_PBE/N	300.0	400.0	520.000	
POTPAW_PBE/Np	190.776	254.369	330.680	1
POTPAW_PBE/Np_s	158.138	210.851	274.106	1
POTPAW_PBE/N_s_GW	222.371	296.495	385.444	

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/N_s	209.76	279.68	363.584	
POTPAW_PBE/N_vs	209.76	279.68	363.584	
POTPAW_PBE/Na	76.476	101.968	132.558	
POTPAW_PBE/Na_pv	194.671	259.561	337.429	
POTPAW_PBE/Na_sv	484.23	700.0	910.000	
POTPAW_PBE/Nb_pv	156.456	208.608	271.190	1
POTPAW_PBE/Nb_sv_new	219.927	293.235	381.206	1
POTPAW_PBE/Nb_sv	219.927	293.235	381.206	1
POTPAW_PBE/Nd_3	136.909	182.546	237.310	1
POTPAW_PBE/Nd	189.892	253.189	329.146	1
POTPAW_PBE/Ne	257.704	343.606	446.688	
POTPAW_PBE/Ni_new	202.149	269.532	350.392	1
POTPAW_PBE/Ni	202.15	269.533	350.393	1
POTPAW_PBE/Ni_pv	275.959	367.945	478.329	1
POTPAW_PBE/Np	190.776	254.369	330.680	1
POTPAW_PBE/Np_s	158.138	210.851	274.106	1
POTPAW_PBE/O_GW	310.976	414.635	539.025	
POTPAW_PBE/O_h	500.0	700.0	910.000	
POTPAW_PBE/O	300.0	400.0	520.000	
POTPAW_PBE/O_s_GW	225.516	300.688	390.894	
POTPAW_PBE/O_s	212.131	282.841	367.693	
POTPAW_PBE/Os	171.017	228.022	296.429	
POTPAW_PBE/Os_pv	171.017	228.022	296.429	1
POTPAW_PBE/O_sv	1066.119	1421.493	1847.941	
POTPAW_PBE/Os	171.017	228.022	296.429	
POTPAW_PBE/Os_pv	171.017	228.022	296.429	1
POTPAW_PBE/Pa	189.237	252.316	328.011	1
POTPAW_PBE/Pa_s	145.182	193.576	251.649	1
POTPAW_PBE/Pb_d	178.384	237.846	309.200	1
POTPAW_PBE/Pb_d_rel2	178.357	237.809	309.152	1
POTPAW_PBE/Pb_d_rel	178.357	237.809	309.152	1
POTPAW_PBE/Pb	73.48	97.973	127.365	
POTPAW_PBE/Pd_new	188.194	250.925	326.203	1
POTPAW_PBE/Pd	188.194	250.925	326.203	1
POTPAW_PBE/Pd_pv_new	188.194	250.925	326.203	1
POTPAW_PBE/Pd_pv	203.323	271.098	352.427	1
POTPAW_PBE/Pd_vnew	188.194	250.925	326.203	1
POTPAW_PBE/P_h	292.651	390.202	507.263	
POTPAW_PBE/Pm_3	137.931	183.908	239.080	1
POTPAW_PBE/Pm	193.97	258.627	336.215	1
POTPAW_PBE/Po_d	198.424	264.565	343.935	1
POTPAW_PBE/Po	119.78	159.707	207.619	
POTPAW_PBE/P	191.28	270.0	351.000	

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Pr_3	136.734	182.312	237.006	1
POTPAW_PBE/Pr	204.706	272.941	354.823	1
POTPAW_PBE/Pt_new	172.712	230.283	299.368	1
POTPAW_PBE/Pt	172.712	230.283	299.368	1
POTPAW_PBE/Pt_pv	220.955	294.607	382.989	1
POTPAW_PBE/Pt_pv_ZORA	220.953	294.604	382.985	1
POTPAW_PBE/Pt_ZORA	172.711	230.281	299.365	1
POTPAW_PBE/Pu_h	333.587	444.783	578.218	1
POTPAW_PBE/Pu	190.844	254.458	330.795	1
POTPAW_PBE/Pu_s	158.508	211.344	274.747	1
POTPAW_PBE/Pa	189.237	252.316	328.011	1
POTPAW_PBE/Pa_s	145.182	193.576	251.649	1
POTPAW_PBE/Pb_d	178.384	237.846	309.200	1
POTPAW_PBE/Pb_d_rel2	178.357	237.809	309.152	1
POTPAW_PBE/Pb_d_rel	178.357	237.809	309.152	1
POTPAW_PBE/Pb	73.48	97.973	127.365	
POTPAW_PBE/Pd_new	188.194	250.925	326.203	1
POTPAW_PBE/Pd	188.194	250.925	326.203	1
POTPAW_PBE/Pd_pv_new	188.194	250.925	326.203	1
POTPAW_PBE/Pd_pv	203.323	271.098	352.427	1
POTPAW_PBE/Pd_vnew	188.194	250.925	326.203	1
POTPAW_PBE/Pm_3	137.931	183.908	239.080	1
POTPAW_PBE/Pm	193.97	258.627	336.215	1
POTPAW_PBE/Po_d	198.424	264.565	343.935	1
POTPAW_PBE/Po	119.78	159.707	207.619	
POTPAW_PBE/Pr_3	136.734	182.312	237.006	1
POTPAW_PBE/Pr	204.706	272.941	354.823	1
POTPAW_PBE/Pt_new	172.712	230.283	299.368	1
POTPAW_PBE/Pt	172.712	230.283	299.368	1
POTPAW_PBE/Pt_pv	220.955	294.607	382.989	1
POTPAW_PBE/Pt_pv_ZORA	220.953	294.604	382.985	1
POTPAW_PBE/Pt_ZORA	172.711	230.281	299.365	1
POTPAW_PBE/Pu_h	333.587	444.783	578.218	1
POTPAW_PBE/Pu	190.844	254.458	330.795	1
POTPAW_PBE/Pu_s	158.508	211.344	274.747	1
POTPAW_PBE/Ra_sv	178.025	237.367	308.577	1
POTPAW_PBE/Rb_pv	91.439	121.919	158.495	
POTPAW_PBE/Rb_sv	165.017	220.022	286.029	
POTPAW_PBE/Re	169.662	226.216	294.081	
POTPAW_PBE/Re_pv	169.667	226.223	294.090	1
POTPAW_PBE/Rh_new	171.747	228.996	297.695	
POTPAW_PBE/Rh	171.75	229.0	297.700	
POTPAW_PBE/Rh_pv_new	185.556	247.408	321.630	1

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Rh_pv	203.602	271.47	352.911	1
POTPAW_PBE/Rn	114.091	152.121	197.757	
POTPAW_PBE/Ru_new	159.953	213.271	277.252	
POTPAW_PBE/Ru	159.957	213.276	277.259	
POTPAW_PBE/Ru_pv_new	180.037	240.049	312.064	1
POTPAW_PBE/Ru_pv	172.822	230.429	299.558	1
POTPAW_PBE/Ru_sv	239.141	318.855	414.512	1
POTPAW_PBE/Sb	129.028	172.037	223.648	
POTPAW_PBE/Sc	116.072	154.763	201.192	
POTPAW_PBE/Sc_sv_h	285.522	380.696	494.905	1
POTPAW_PBE/Sc_sv	166.998	222.664	289.463	1
POTPAW_PBE/Se_GW	158.666	211.555	275.022	
POTPAW_PBE/Se	158.651	211.534	274.994	
POTPAW_PBE/S_h	301.827	402.436	523.167	
POTPAW_PBE/Si_d_GW_nr	184.004	245.338	318.939	
POTPAW_PBE/Si_d_GW	184.009	245.345	318.949	
POTPAW_PBE/Si_h_old	285.109	380.146	494.190	
POTPAW_PBE/Si_h	285.109	380.146	494.190	
POTPAW_PBE/Si_nopc	184.009	245.345	318.949	
POTPAW_PBE/Si	184.009	245.345	318.949	
POTPAW_PBE/Si_pv_GW	356.322	475.096	617.625	1
POTPAW_PBE/Si_sv_GW_nr	356.326	475.101	617.631	1
POTPAW_PBE/Si_sv_GW	356.322	475.096	617.625	1
POTPAW_PBE/Sm_3	132.815	177.087	230.213	1
POTPAW_PBE/Sm	193.136	257.515	334.769	1
POTPAW_PBE/Sn_d	180.817	241.09	313.417	1
POTPAW_PBE/Sn_GW	77.427	103.236	134.207	
POTPAW_PBE/Sn	77.427	103.236	134.207	
POTPAW_PBE/S	194.016	280.0	364.000	
POTPAW_PBE/Sr_sv	171.961	229.282	298.067	1
POTPAW_PBE/Sr	171.961	229.282	298.067	1
POTPAW_PBE/Sb	129.028	172.037	223.648	
POTPAW_PBE/Sc	116.072	154.763	201.192	
POTPAW_PBE/Sc_sv_h	285.522	380.696	494.905	1
POTPAW_PBE/Sc_sv	166.998	222.664	289.463	1
POTPAW_PBE/Se_GW	158.666	211.555	275.022	
POTPAW_PBE/Se	158.651	211.534	274.994	
POTPAW_PBE/Si_d_GW_nr	184.004	245.338	318.939	
POTPAW_PBE/Si_d_GW	184.009	245.345	318.949	
POTPAW_PBE/Si_h_old	285.109	380.146	494.190	
POTPAW_PBE/Si_h	285.109	380.146	494.190	
POTPAW_PBE/Si_nopc	184.009	245.345	318.949	
POTPAW_PBE/Si	184.009	245.345	318.949	

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Si_pv_GW	356.322	475.096	617.625	1
POTPAW_PBE/Si_sv_GW_nr	356.326	475.101	617.631	1
POTPAW_PBE/Si_sv_GW	356.322	475.096	617.625	1
POTPAW_PBE/Sm_3	132.815	177.087	230.213	1
POTPAW_PBE/Sm	193.136	257.515	334.769	1
POTPAW_PBE/Sn_d	180.817	241.09	313.417	1
POTPAW_PBE/Sn_GW	77.427	103.236	134.207	
POTPAW_PBE/Sn	77.427	103.236	134.207	
POTPAW_PBE/Sr_sv	171.961	229.282	298.067	1
POTPAW_PBE/Sr	171.961	229.282	298.067	1
POTPAW_PBE/Ta	167.75	223.667	290.767	
POTPAW_PBE/Ta_pv	167.756	223.675	290.778	1
POTPAW_PBE/Tb_3	116.721	155.628	202.316	
POTPAW_PBE/Tb	198.618	264.824	344.271	1
POTPAW_PBE/Tc_new	171.521	228.694	297.302	
POTPAW_PBE/Tc	171.521	228.694	297.302	
POTPAW_PBE/Tc_pv_new	197.642	263.523	342.580	1
POTPAW_PBE/Tc_pv	171.524	228.699	297.309	1
POTPAW_PBE/Te	131.236	174.982	227.477	
POTPAW_PBE/Te_rel	131.234	174.979	227.473	
POTPAW_PBE/Th	185.587	247.449	321.684	1
POTPAW_PBE/Th_s	127.119	169.492	220.340	1
POTPAW_PBE/Ti	133.747	178.33	231.829	
POTPAW_PBE/Ti_pv	166.753	222.338	289.039	1
POTPAW_PBE/Ti_sv_GW	234.428	312.571	406.342	1
POTPAW_PBE/Ti_sv_h	291.524	388.698	505.307	1
POTPAW_PBE/Ti_sv_new2	205.957	274.61	356.993	1
POTPAW_PBE/Ti_sv_new	205.957	274.61	356.993	1
POTPAW_PBE/Ti_sv	205.93	274.574	356.946	1
POTPAW_PBE/Tl_d	177.797	237.063	308.182	1
POTPAW_PBE/Tl	67.605	90.14	117.182	
POTPAW_PBE/Tm_3	111.916	149.221	193.987	
POTPAW_PBE/Tm	193.065	257.419	334.645	2
POTPAW_PBE/U	189.462	252.616	328.401	1
POTPAW_PBE/U_s	156.802	209.069	271.790	1
POTPAW_PBE/V	144.408	192.543	250.306	
POTPAW_PBE/V_pv	197.756	263.675	342.778	1
POTPAW_PBE/V_sv_h	292.998	390.664	507.863	1
POTPAW_PBE/V_sv_new	197.755	263.673	342.775	1
POTPAW_PBE/V_sv	197.756	263.675	342.778	1
POTPAW_PBE/W	167.293	223.057	289.974	
POTPAW_PBE/W_pv_new	167.293	223.057	289.974	1
POTPAW_PBE/W_pv	167.299	223.065	289.985	1

Continued on next page

Table 8: Parameters for POTCAR files.

POTCAR	ENMIN (eV)	ENMAX (eV)	ENCUT prec=high (eV)	# valence electrons
POTPAW_PBE/Xe	114.823	153.098	199.027	
POTPAW_PBE/Xe	114.823	153.098	199.027	
POTPAW_PBE/Yb_2_n	108.514	144.685	188.091	1
POTPAW_PBE/Yb_2	84.407	112.543	146.306	
POTPAW_PBE/Yb	189.771	253.028	328.936	2
POTPAW_PBE/Y_sv	158.731	211.641	275.133	1
POTPAW_PBE/Yb_2_n	108.514	144.685	188.091	1
POTPAW_PBE/Yb_2	84.407	112.543	146.306	
POTPAW_PBE/Yb	189.771	253.028	328.936	2
POTPAW_PBE/Zn	207.545	276.727	359.745	1
POTPAW_PBE/Zn_pv	282.455	376.607	489.589	1
POTPAW_PBE/Zr	115.974	154.632	201.022	
POTPAW_PBE/Zr_sv_GW	230.877	307.836	400.187	1
POTPAW_PBE/Zr_sv_new	172.424	229.898	298.867	1
POTPAW_PBE/Zr_sv	172.379	229.839	298.791	1

## 10 Python

### 10.1 easy\_install as a user

`easy_install` is a python command-line utility that automatically installs python packages. Usually you need root access to install a python package, but you can also tell `easy_install` where to install a package. This usually works if the directory is on your PYTHONPATH

```
easy_install -d ~/lib/python2.6/site-packages/ pymatgen
```

### 10.2 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 = {0} (integer division)'.format(x_Cu)
5
6 # now cast as floats
7 x_Cu = float(nCu)/float(nPd + nCu)
8 print 'x_cu = {0} (float division)'.format(x_Cu)

```

---

```

x_cu = 0 (integer division)
x_cu = 0.555555555556 (float division)

```



Note that if one of the numbers is a float, python will automatically cast the integer as a float, and return a float.

---

```
1 nPd = 4
2 nCu = 5
3
4 # now cast as floats
5 x_Cu = float(nCu)/(nPd + nCu)
6 print 'x_cu = {0}'.format(x_Cu)
```

---

x\_cu = 0.555555555556

Finally, you can tell python a number is a float by adding a decimal to it. You do not need to put a 0 after the decimal, but you can.

---

```
1 nPd = 4. # this is a float
2 nCu = 5
3
4 x_Cu = nCu/(nPd + nCu)
5 print 'x_cu = {0}'.format(x_Cu)
```

---

x\_cu = 0.555555555556

## 11 References

### References

- [1] 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. ISSN 0966-9795. doi: 10.1016/S0966-9795(02)00127-9. URL <http://www.sciencedirect.com/science/article/pii/S0966979502001279>.
- [2] R. G. Parr and W. Yang. *Density-Functional Theory of Atoms and Molecules*. Oxford Science Publications, 1989.
- [3] W. Koch and M. C. Holthausen. *A Chemist's Guide to Density Functional Theory*. Wiley-VCH, 2 edition, 2001.
- [4] Charles Kittel. *Introduction to Solid State Physics*. Wiley, 8th edition, 2005.
- [5] N. W. Ashcroft and N. David Mermin. *Solid State Physics*. Saunders College Publishing, 1976.
- [6] Roald Hoffmann. How chemistry and physics meet in the solid state. *Angewandte Chemie International Edition in English*, 26(9):846–878, 1987. ISSN 1521-3773. doi: 10.1002/anie.198708461. URL <http://dx.doi.org/10.1002/anie.198708461>.
- [7] Roald Hoffmann. A chemical and theoretical way to look at bonding on surfaces. *Rev. Mod. Phys.*, 60:601–628, Jul 1988. doi: 10.1103/RevModPhys.60.601. URL <http://link.aps.org/doi/10.1103/RevModPhys.60.601>.

- [8] B. Hammer and J.K. Nørskov. Theoretical surface science and catalysis calculations and concepts. In Helmut Knozinger Bruce C. Gates, editor, *Impact of Surface Science on Catalysis*, volume 45 of *Advances in Catalysis*, pages 71 – 129. Academic Press, 2000. doi: 10.1016/S0360-0564(02)45013-4. URL <http://www.sciencedirect.com/science/article/pii/S0360056402450134>.
- [9] Jeff Greeley, Jens K. Nørskov, and Manos Mavrikakis. Electronic structure and catalysis on metal surfaces. *Annual Review of Physical Chemistry*, 53(1):319–348, 2002. doi: 10.1146/annurev.physchem.53.100301.131630. URL <http://www.annualreviews.org/doi/abs/10.1146/annurev.physchem.53.100301.131630>.
- [10] A J Freeman and E Wimmer. Density functional theory as a major tool in computational materials science. *Annual Review of Materials Science*, 25(1):7–36, 1995. doi: 10.1146/annurev.ms.25.080195.000255.
- [11] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. Iterative minimization techniques for *ab initio* total-energy calculations: molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64:1045–1097, Oct 1992. doi: 10.1103/RevModPhys.64.1045. URL <http://link.aps.org/doi/10.1103/RevModPhys.64.1045>.
- [12] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Computational Materials Science*, 6(1):15 – 50, 1996. ISSN 0927-0256. doi: 10.1016/0927-0256(96)00008-0. URL <http://www.sciencedirect.com/science/article/pii/0927025696000080>.
- [13] T. Bligaard. Exchange and Correlation Functionals - a study toward improving the precision of electron density functional calculations of atomistic systems. Master’s thesis, Technical University of Denmark, 2000. <http://www.fysik.dtu.dk/~{}bligaard/masterthesis/masterdirectory/project/project.pdf>.
- [14] T. Bligaard. *Understanding Materials Properties on the Basis of Density Functional Theory Calculations*. PhD thesis, Technical University of Denmark, 2003. <http://www.fysik.dtu.dk/~{}bligaard/phdthesis/phdproject.pdf>.
- [15] A. P. Seitsonen. *Theoretical Investigations into adsorption and co-adsorption on transition-metal surfaces as models to heterogeneous catalysis*. PhD thesis, Technical University of Berlin, School of Mathematics and Natural Sciences, 2000. [http://edocs.tu-berlin.de/diss/2000/seitsonen\\_ari.pdf](http://edocs.tu-berlin.de/diss/2000/seitsonen_ari.pdf).
- [16] R. Hirschl. *Binary Transition Metal Alloys and Their Surfaces*. PhD thesis, Institut für Materialphysik, University of Vienna, 2002. [http://www.hirschl.at/download/diss\\_part1.pdf](http://www.hirschl.at/download/diss_part1.pdf) and [http://www.hirschl.at/download/diss\\_part2.pdf](http://www.hirschl.at/download/diss_part2.pdf).
- [17] L. Pauling and E. B. Wilson, Jr. *Introduction to Quantum Mechanics with Applications to Chemistry*. Dover Publications, Inc., 1963.
- [18] W. Kohn. Nobel lecture: Electronic structure of matter-wave functions and density functionals. *Rev. Mod. Phys.*, 71:1253–1266, Oct 1999. doi: 10.1103/RevModPhys.71.1253. URL <http://link.aps.org/doi/10.1103/RevModPhys.71.1253>.

- [19] P. A. M. Dirac. Quantum mechanics of many-electron systems. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 123(792):pp. 714–733, 1929. ISSN 09501207. URL <http://www.jstor.org/stable/95222>.
- [20] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964. doi: 10.1103/PhysRev.136.B864. URL <http://link.aps.org/doi/10.1103/PhysRev.136.B864>.
- [21] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965. doi: 10.1103/PhysRev.140.A1133. URL <http://link.aps.org/doi/10.1103/PhysRev.140.A1133>.
- [22] John A. Pople. Nobel lecture: Quantum chemical models. *Rev. Mod. Phys.*, 71:1267–1274, Oct 1999. doi: 10.1103/RevModPhys.71.1267. URL <http://link.aps.org/doi/10.1103/RevModPhys.71.1267>.
- [23] M. Fuchs, M. Bockstedte, E. Pehlke, and M. Scheffler. Pseudopotential study of binding properties of solids within generalized gradient approximations: The role of core-valence exchange correlation. *Phys. Rev. B*, 57:2134–2145, Jan 1998. doi: 10.1103/PhysRevB.57.2134. URL <http://link.aps.org/doi/10.1103/PhysRevB.57.2134>.
- [24] John P. Perdew, Robert G. Parr, Mel Levy, and Jose L. Balduz. Density-functional theory for fractional particle number: Derivative discontinuities of the energy. *Phys. Rev. Lett.*, 49:1691–1694, Dec 1982. doi: 10.1103/PhysRevLett.49.1691. URL <http://link.aps.org/doi/10.1103/PhysRevLett.49.1691>.
- [25] B. Hammer, L. B. Hansen, and J. K. Nørskov. Improved adsorption energetics within density-functional theory using revised perdew-burke-ernzerhof functionals. *Phys. Rev. B*, 59:7413–7421, Mar 1999. doi: 10.1103/PhysRevB.59.7413. URL <http://link.aps.org/doi/10.1103/PhysRevB.59.7413>.
- [26] G. Makov and M. C. Payne. Periodic boundary conditions in *ab initio* calculations. *Phys. Rev. B*, 51:4014–4022, Feb 1995. doi: 10.1103/PhysRevB.51.4014. URL <http://link.aps.org/doi/10.1103/PhysRevB.51.4014>.
- [27] D. J. Chadi and Marvin L. Cohen. Special points in the brillouin zone. *Phys. Rev. B*, 8: 5747–5753, Dec 1973. doi: 10.1103/PhysRevB.8.5747. URL <http://link.aps.org/doi/10.1103/PhysRevB.8.5747>.
- [28] Hendrik J. Monkhorst and James D. Pack. Special points for brillouin-zone integrations. *Phys. Rev. B*, 13:5188–5192, Jun 1976. doi: 10.1103/PhysRevB.13.5188. URL <http://link.aps.org/doi/10.1103/PhysRevB.13.5188>.
- [29] N. Troullier and José Luriaas Martins. Efficient pseudopotentials for plane-wave calculations. *Phys. Rev. B*, 43:1993–2006, Jan 1991. doi: 10.1103/PhysRevB.43.1993. URL <http://link.aps.org/doi/10.1103/PhysRevB.43.1993>.
- [30] David Vanderbilt. Soft self-consistent pseudopotentials in a generalized eigenvalue formalism. *Phys. Rev. B*, 41:7892–7895, Apr 1990. doi: 10.1103/PhysRevB.41.7892. URL <http://link.aps.org/doi/10.1103/PhysRevB.41.7892>.

- [31] E. G. Moroni, G. Kresse, J. Hafner, and J. Furthmüller. Ultrasoft pseudopotentials applied to magnetic fe, co, and ni: From atoms to solids. *Phys. Rev. B*, 56:15629–15646, Dec 1997. doi: 10.1103/PhysRevB.56.15629. URL <http://link.aps.org/doi/10.1103/PhysRevB.56.15629>.
- [32] P. E. Blöchl. Projector augmented-wave method. *Phys. Rev. B*, 50:17953–17979, Dec 1994. doi: 10.1103/PhysRevB.50.17953. URL <http://link.aps.org/doi/10.1103/PhysRevB.50.17953>.
- [33] G. Kresse and D. Joubert. From ultrasoft pseudopotentials to the projector augmented-wave method. *Phys. Rev. B*, 59:1758–1775, Jan 1999. doi: 10.1103/PhysRevB.59.1758. URL <http://link.aps.org/doi/10.1103/PhysRevB.59.1758>.
- [34] M J Gillan. Calculation of the vacancy formation energy in aluminium. *Journal of Physics: Condensed Matter*, 1(4):689, 1989. URL <http://stacks.iop.org/0953-8984/1/i=4/a=005>.
- [35] N. David Mermin. Thermal properties of the inhomogeneous electron gas. *Phys. Rev.*, 137:A1441–A1443, Mar 1965. doi: 10.1103/PhysRev.137.A1441. URL <http://link.aps.org/doi/10.1103/PhysRev.137.A1441>.
- [36] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71:035109, Jan 2005. doi: 10.1103/PhysRevB.71.035109. URL <http://link.aps.org/doi/10.1103/PhysRevB.71.035109>.
- [37] G. Kresse and J. Furthmüller. Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996. doi: 10.1103/PhysRevB.54.11169. URL <http://link.aps.org/doi/10.1103/PhysRevB.54.11169>.
- [38] G. Kresse and J. Hafner. *Ab initio* molecular-dynamics simulation of the liquid-metal/amorphous-semiconductor transition in germanium. *Phys. Rev. B*, 49:14251–14269, May 1994. doi: 10.1103/PhysRevB.49.14251. URL <http://link.aps.org/doi/10.1103/PhysRevB.49.14251>.
- [39] G. Kresse and J. Hafner. *Ab initio* molecular dynamics for liquid metals. *Phys. Rev. B*, 47:558–561, Jan 1993. doi: 10.1103/PhysRevB.47.558. URL <http://link.aps.org/doi/10.1103/PhysRevB.47.558>.
- [40] 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. doi: 10.1063/1.1926272. URL <http://link.aip.org/link/?JCP/122/234102/1>.
- [41] Larry A. Curtiss, Krishnan Raghavachari, Paul C. Redfern, and John A. Pople. Assessment of gaussian-2 and density functional theories for the computation of enthalpies of formation. *The Journal of Chemical Physics*, 106(3):1063–1079, 1997. doi: 10.1063/1.473182. URL <http://link.aip.org/link/?JCP/106/1063/1>.
- [42] 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. ISSN 0927-0256. doi: 10.1016/j.commatsci.2005.04.010. URL <http://www.sciencedirect.com/science/article/pii/S0927025605001849>.
- [43] 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. doi: 10.1021/ct100125x. URL <http://pubs.acs.org/doi/abs/10.1021/ct100125x>.
- [44] Jr. E. Bright Wilson, J.C. Decius, and Paul C. Cross. *Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra*. Dover Publications, 1955.
- [45] Paolo Giannozzi and Stefano Baroni. Vibrational and dielectric properties of  $c_{60}$  from density-functional perturbation theory. *The Journal of Chemical Physics*, 100(11):8537–8539, 1994. doi: 10.1063/1.466753. URL <http://link.aip.org/link/?JCP/100/8537/1>.
- [46] 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>.
- [47] Antonio Fernández-Ramos, Benjamin Ellingson, Rubén Meana-Pañeda, Jorge Marques, and Donald Truhlar. Symmetry numbers and chemical reaction rates. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 118:813–826, 2007. ISSN 1432-881X. URL <http://dx.doi.org/10.1007/s00214-007-0328-0>. 10.1007/s00214-007-0328-0.
- [48] Ann E. Mattsson, Rickard Armiento, Joachim Paier, Georg Kresse, John M. Wills, and Thomas R. Mattsson. The am05 density functional applied to solids. *The Journal of Chemical Physics*, 128(8):084714, 2008. doi: 10.1063/1.2835596. URL <http://link.aip.org/link/?JCP/128/084714/1>.
- [49] Yvon Le Page and Paul Saxe. Symmetry-general least-squares extraction of elastic data for strained materials from *ab initio* calculations of stress. *Phys. Rev. B*, 65:104104, Feb 2002. doi: 10.1103/PhysRevB.65.104104. URL <http://link.aps.org/doi/10.1103/PhysRevB.65.104104>.
- [50] Shun-Li Shang, Yi Wang, DongEung Kim, and Zi-Kui Liu. First-principles thermodynamics from phonon and debye model: Application to ni and ni3al. *Computational Materials Science*, 47(4):1040 – 1048, 2010. ISSN 0927-0256. doi: 10.1016/j.commatsci.2009.12.006. URL <http://www.sciencedirect.com/science/article/pii/S0927025609004558>.
- [51] A. van de Walle, M. Asta, and G. Ceder. The alloy theoretic automated toolkit: A user guide. *Calphad*, 26(4):539 – 553, 2002. ISSN 0364-5916. doi: 10.1016/S0364-5916(02)80006-2. URL <http://www.sciencedirect.com/science/article/pii/S0364591602800062>.
- [52] Axel van de Walle. Multicomponent multisublattice alloys, nonconfigurational entropy and other additions to the alloy theoretic automated toolkit. *Calphad*, 33(2):266 – 278, 2009. ISSN 0364-5916. doi: 10.1016/j.calphad.2008.12.005. URL <http://www.sciencedirect.com/science/article/pii/S0364591608001314>. jce:title;Tools for Computational Thermodynamicsj/ce:title;.

- [53] 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. doi: 10.1103/PhysRevB.73.195107. URL <http://link.aps.org/doi/10.1103/PhysRevB.73.195107>.
- [54] Leonard Kleinman. Significance of the highest occupied Kohn-Sham eigenvalue. *Phys. Rev. B*, 56:12042–12045, Nov 1997. doi: 10.1103/PhysRevB.56.12042. URL <http://link.aps.org/doi/10.1103/PhysRevB.56.12042>.
- [55] John P. Perdew and Mel Levy. Comment on "Significance of the highest occupied Kohn-Sham eigenvalue". *Phys. Rev. B*, 56:16021–16028, Dec 1997. doi: 10.1103/PhysRevB.56.16021. URL <http://link.aps.org/doi/10.1103/PhysRevB.56.16021>.
- [56] Leonard Kleinman. Reply to "Comment on 'Significance of the highest occupied Kohn-Sham eigenvalue' ". *Phys. Rev. B*, 56:16029–16030, Dec 1997. doi: 10.1103/PhysRevB.56.16029. URL <http://link.aps.org/doi/10.1103/PhysRevB.56.16029>.
- [57] Ralf Stowasser and Roald Hoffmann. What do the Kohn-Sham Orbitals and Eigenvalues Mean? *Journal of the American Chemical Society*, 121(14):3414–3420, 1999. doi: 10.1021/ja9826892. URL <http://pubs.acs.org/doi/abs/10.1021/ja9826892>.
- [58] O. V. Gritsenko and E. J. Baerends. The analog of Koopmans' theorem in spin-density functional theory. *The Journal of Chemical Physics*, 117(20):9154–9159, 2002. doi: 10.1063/1.1516800. URL <http://link.aip.org/link/?JCP/117/9154/1>.
- [59] R. O. Jones and O. Gunnarsson. The density functional formalism, its applications and prospects. *Rev. Mod. Phys.*, 61:689–746, Jul 1989. doi: 10.1103/RevModPhys.61.689. URL <http://link.aps.org/doi/10.1103/RevModPhys.61.689>.
- [60] J. P. Perdew and Alex Zunger. Self-interaction correction to density-functional approximations for many-electron systems. *Phys. Rev. B*, 23:5048–5079, May 1981. doi: 10.1103/PhysRevB.23.5048. URL <http://link.aps.org/doi/10.1103/PhysRevB.23.5048>.
- [61] Daniel Sanchez-Portal, Emilio Artacho, and Jose M Soler. Projection of plane-wave calculations into atomic orbitals. *Solid State Communications*, 95(10):685 – 690, 1995. ISSN 0038-1098. doi: 10.1016/0038-1098(95)00341-X. URL <http://www.sciencedirect.com/science/article/pii/003810989500341X>.
- [62] M. D. Segall, R. Shah, C. J. Pickard, and M. C. Payne. Population analysis of plane-wave electronic structure calculations of bulk materials. *Phys. Rev. B*, 54:16317–16320, Dec 1996. doi: 10.1103/PhysRevB.54.16317. URL <http://link.aps.org/doi/10.1103/PhysRevB.54.16317>.
- [63] M. D. Segall, C. J. Pickard, R. Shah, and M. C. Payne. Population analysis in plane wave electronic structure calculations. *Mol. Phys.*, 89(2):571–577, 1996.
- [64] A Ruban, B Hammer, P Stoltze, H.L Skriver, and J.K Nørskov. Surface electronic structure and reactivity of transition and noble metals. *Journal of Molecular Catalysis A: Chemical*, 115(3):421 – 429, 1997. ISSN 1381-1169. doi: 10.1016/S1381-1169(96)00348-2. URL <http://www.sciencedirect.com/science/article/pii/S1381116996003482>.
- [65] A. Cottrell. *Introduction to the Modern Theory of Metals*. The Institute of Metals, 1988.



- [66] F. Ducastelle. *Order and Phase Stability in Alloys*. Elsevier Science Publishers, 1991.
- [67] D. G. Pettifor and A. H. Cottrell, editors. *Electron Theory in Alloy Design*. The Institute of Materials, 1992.
- [68] 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. doi: 10.1103/RevModPhys.73.515. URL <http://link.aps.org/doi/10.1103/RevModPhys.73.515>.
- [69] Kyle J. Caspersen and Emily A. Carter. Finding transition states for crystalline solid/solid phase transformations. *Proceedings of the National Academy of Sciences of the United States of America*, 102(19):6738–6743, 2005. doi: 10.1073/pnas.0408127102. URL <http://www.pnas.org/content/102/19/6738.abstract>.
- [70] 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. doi: 10.1063/1.3684549. URL <http://link.aip.org/link/?JCP/136/074103/1>.
- [71] 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. doi: 10.1103/PhysRevB.83.075415. URL <http://link.aps.org/doi/10.1103/PhysRevB.83.075415>.
- [72] J. W. M. Frenken, R. L. Krams, J. F. van der Veen, E. Holub-Krappe, and K. Horn. Missing-row surface reconstruction of ag(110) induced by potassium adsorption. *Phys. Rev. Lett.*, 59:2307–2310, Nov 1987. doi: 10.1103/PhysRevLett.59.2307. URL <http://link.aps.org/doi/10.1103/PhysRevLett.59.2307>.
- [73] J. C. Boettger. Nonconvergence of surface energies obtained from thin-film calculations. *Phys. Rev. B*, 49:16798–16800, Jun 1994. doi: 10.1103/PhysRevB.49.16798. URL <http://link.aps.org/doi/10.1103/PhysRevB.49.16798>.
- [74] 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>.
- [75] 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. ISSN 0079-6816. doi: 10.1016/j.progsurf.2003.08.017. URL <http://www.sciencedirect.com/science/article/pii/S0079681603000777>.
- [76] 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. doi: 10.1103/PhysRevB.75.115131. URL <http://link.aps.org/doi/10.1103/PhysRevB.75.115131>.

- [77] Jeong Woo Han, Liwei Li, and David S. Sholl. Density functional theory study of H and CO adsorption on alkali-promoted Mo<sub>2</sub>C surfaces. *The Journal of Physical Chemistry C*, 115(14):6870–6876, 2011. doi: 10.1021/jp200950a. URL <http://pubs.acs.org/doi/abs/10.1021/jp200950a>.
- [78] 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. ISSN 0021-9517. doi: 10.1016/j.jcat.2008.11.020. URL <http://www.sciencedirect.com/science/article/pii/S0021951708004314>.
- [79] L. Vitos, A.V. Ruban, H.L. Skriver, and J. Kollár. The surface energy of metals. *Surface Science*, 411(12):186 – 202, 1998. ISSN 0039-6028. doi: 10.1016/S0039-6028(98)00363-X. URL <http://www.sciencedirect.com/science/article/pii/S003960289800363X>.
- [80] 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. doi: 10.1103/PhysRevB.46.16067. URL <http://link.aps.org/doi/10.1103/PhysRevB.46.16067>.
- [81] Lennart Bengtsson. Dipole correction for surface supercell calculations. *Phys. Rev. B*, 59:12301–12304, May 1999. doi: 10.1103/PhysRevB.59.12301. URL <http://link.aps.org/doi/10.1103/PhysRevB.59.12301>.
- [82] Yoshitada Morikawa. Adsorption geometries and vibrational modes of C<sub>2</sub>H<sub>2</sub> on the si(001) surface. *Phys. Rev. B*, 63:033405, Jan 2001. doi: 10.1103/PhysRevB.63.033405. URL <http://link.aps.org/doi/10.1103/PhysRevB.63.033405>.
- [83] 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. doi: 10.1103/PhysRevB.82.045414. URL <http://link.aps.org/doi/10.1103/PhysRevB.82.045414>.
- [84] 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. doi: 10.1063/1.3561287. URL <http://link.aip.org/link/?JCP/134/104709/1>.
- [85] 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. ISSN 0039-6028. doi: 10.1016/j.susc.2009.01.021. URL <http://www.sciencedirect.com/science/article/pii/S0039602809001186>.
- [86] John R. Kitchin. Correlations in coverage-dependent atomic adsorption energies on pd(111). *Phys. Rev. B*, 79:205412, May 2009. doi: 10.1103/PhysRevB.79.205412. URL <http://link.aps.org/doi/10.1103/PhysRevB.79.205412>.
- [87] John R. Kitchin, Karsten Reuter, and Matthias Scheffler. Alloy surface segregation in reactive environments: First-principles atomistic thermodynamics study of Ag<sub>3</sub>Pd(111) in oxygen atmospheres. *Phys. Rev. B*, 77:075437, Feb 2008. doi: 10.1103/PhysRevB.77.075437. URL <http://link.aps.org/doi/10.1103/PhysRevB.77.075437>.



- [88] 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. ISSN 1521-3773. doi: 10.1002/anie.200602223. URL <http://dx.doi.org/10.1002/anie.200602223>.
- [89] P. Deshlahra, E. E. Wolf, and W. F. Schneider. A periodic density functional theory analysis of co chemisorption on pt(111) in the presence of uniform electric fields†. *The Journal of Physical Chemistry A*, 113(16):4125–4133, 2009. doi: 10.1021/jp810518x. URL <http://pubs.acs.org/doi/abs/10.1021/jp810518x>.
- [90] 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. doi: 10.1103/PhysRevB.84.045115. URL <http://link.aps.org/doi/10.1103/PhysRevB.84.045115>.
- [91] 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. doi: 10.1103/PhysRevB.82.081101. URL <http://link.aps.org/doi/10.1103/PhysRevB.82.081101>.
- [92] Stefan Grimme. Semiempirical gga-type density functional constructed with a long-range dispersion correction. *Journal of Computational Chemistry*, 27(15):1787–1799, 2006. ISSN 1096-987X. doi: 10.1002/jcc.20495. URL <http://dx.doi.org/10.1002/jcc.20495>.
- [93] 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>.
- [94] 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. doi: 10.1103/PhysRevB.70.224107. URL <http://link.aps.org/doi/10.1103/PhysRevB.70.224107>.
- [95] 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. ISSN 00278424. URL <http://www.jstor.org/stable/87468>.
- [96] 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. doi: 10.1103/PhysRevB.28.5480. URL <http://link.aps.org/doi/10.1103/PhysRevB.28.5480>.
- [97] Michael J. Mehl, Barry M. Klein, and Dimitri A. Papaconstantopoulos. *Intermetallic Compounds: Principles and Principles, Volume I: Principles*, volume I, chapter First principles calculations of elastic properties of metals, pages 195–210. John Wiley and Sons, 1995. URL <http://cst-www.nrl.navy.mil/users/mehl/papers/cij453.pdf>.

## 12 Index

## Index

adsorption energy, [179](#)  
atomistic thermodynamics, [194](#)  
  
bader, [50](#)  
band structure, [153](#)  
  
center of mass, [31](#)  
cohesive energy, [127](#)  
convergence  
    ENCUT, [71](#)  
    KPOINTS, [128](#)  
  
DFT+U, [203](#)  
dipole correction, [176](#)  
dipole moment, [44](#)  
  
HSE06, [207](#)  
  
infrared intensity, [61](#)  
ISMEAR, [105](#)  
  
jasp, [11](#)  
  
molecular weight, [30](#)  
moment of inertia, [31](#)  
  
nudged elastic band, [93](#)  
  
reconstruction, [165](#)  
  
SIGMA, [105](#)  
  
thermochemistry, [65](#)  
  
vibrations, [57](#)  
  
work function, [171](#)