Relating the Electronic Structure and Reactivity of the 3d Transition Metal Monoxide Surfaces: Supplemental Information

John Kitchin, Zhongnan Xu[a], John R. Kitchin[a,*]

[a]*Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213*

Contents

*Corresponding author
Email address: `jkitchin@andrew.cmu.edu` (John R. Kitchin)

1. Introduction

The supporting information contains all the code for performing and analyzing the calculations needed to reproduce our results. Where it is feasible we include in tabular form the data generated in this work, as well as the code for generating the figures in the manuscript. We did not include in tabular form the density of states data because they are large tables. We opted not include the data files for the densities of state because they cannot be used directly in the code provided here, and that is not how we used the data. It is presently not possible to share this data in a form that would allow direct use of the code here. The code here relies on the existence of all of the files in a typical VASP calculation. Notably, we are unable to share the POTCAR files as a condition of the VASP license. Without these POTCAR files, the code here that reads data from the VASP calculations cannot run. However, licensed users of VASP should be able to run the codes to generate the data themselves to reproduce these results. This document does not contain the snippets of code used for restarting calculations when they did not converge correctly, crashed, etc...

Initially the study was meant to include Cr and Fe, but after performing adsorption calculations on CrO and FeO (111) surfaces, we found that during relaxation of the surface slabs with and/or without the adsorbate, major deconstruction of the surface happened, and minimization of the forces took unreasonable times to complete. We could not identify the cause for this nor a solution, and calculations on these surfaces are not included in this work.

The document that created this file can be found here: oxide-surfaces-supplemental.org. This file is written in the markup language of org-mode (http://orgmode.org/), which allows narrative text and code to be intermingled using lightweight markup. The code can be executed and the output captured in the document, allowing comprehensive documentation of how calculations were setup, run and analyzed. The file can be readily converted to LaTeX, which can then be converted to PDF. The details of the export can be found in the embedded org file.

2. Modules needed

Below lists some of the modules used that the user will need to reproduce our data.

- `matplotlib`
- `numpy`
- `ase`
- `jasp`
- `mybulk`
- `mysurfaces`

The versions of `matplotlib` and `numpy` used are those found in the enthought python package and downloaded at https://www.enthought.com/products/epd/free/. `jasp` is a python wrapper of VASP developed by Professor John R. Kitchin and can be found at https://github.com/jkitchin/jasp. `ase`, which stands for Atomic Simulation Environment, is a module for performing molecular simulation through python and can be found at https://wiki.fysik.dtu.dk/ase/.

`mybulk` and `mysurfaces` are modules we prepared for this work. `mybulk` and `mysurfaces` store functions for creating bulk and surfaces. The relevant contents needed from these modules are shown below.

2.1. The `mybulk` module

I primarily imported the `rocksalt` function from the mybulk module. The function is shown below.

```
"""Helper function for bulk Atoms objects"""
from __future__ import division
from ase import Atom, Atoms
from ase.constraints import FixAtoms, FixScaled
import numpy as np

def rocksalt(symbols, a=None, mags=(2, 2), vol=None, afm=True):
    '''Returns a spinel atoms object

    Parameters
    ----------
    symbol: tuple
        The atoms in the unit cell.
    a: flt
        The lattice constant.
    mags: flt
        The initial magnetic moment of the cell in the order of symbols
    vol: flt
        This the volume of the unit cell
    NOTE: One must provide either the volume or the lattice
    constant.
    afm: bool
        If true, it returns the primitive cell of rocksalt with the
        anti-ferromagnetic ordering (four atoms). If false, it returns
        the two atom, primitive cell of rocksalt
    '''
```

```
    if a == None and vol == None:
        raise TypeError('Please provide either a lattice or volume')
    elif a != None and vol != None:
        raise TypeError('Cannot provide both lattice constant and volume')
    elif vol != None:
        a = (2 * vol) ** (1/3)
    b = a / 2
    if afm == True:
        bulk = Atoms([Atom(symbols[0], (0.0, 0.0, 0.0), magmom=mags[0]),
                      Atom(symbols[0], (a,a,a), magmom=-mags[0]),
                      Atom(symbols[1], (b,b,b), magmom=mags[1]),
                      Atom(symbols[1], (3*b,3*b,3*b), magmom=mags[1])],
                     cell=[(a,b,b),
                           (b,a,b),
                           (b,b,a)])
        return bulk
    else:
        bulk = Atoms([Atom(symbols[0], (0.0, 0.0, 0.0), magmom=mags[0]),
                      Atom(symbols[1], (b/2., b/2., b/2.), magmom=mags[1])],
                     cell=((0, b, b),
                           (b, 0, b),
                           (b, b, 0)))
        return bulk
```

2.2. The `mysurfaces` module

The three functions I used from the mysurfaces are rocksalt111 to construct the rocksalt symmetric slab, `add_adsorbate` to add the adsorbate onto both sides of the symmetric slab, and `rotate_111` for rotating the slab. The purpose of rotating the slab will be explained in the section talking about calculating the density of states.

```
"""Module that stores surfaces"""
from __future__ import division

from ase import Atom, Atoms
from ase.constraints import FixAtoms, FixScaled
from myvasp import condense
import numpy as np

def rocksalt111(symbol, a=4.20, b=0, area=(1, 1), layers=5, vacuum=0, mag=2,
                fixlayers=0, base=0, cell=None, afm=True):
    """Returns an atoms object with rocksalt(111) surface


    Parameters
    ----------
    symbol: list
        The atoms used in the cell

    """
    if cell == None:
        a1 = np.array([2**(0.5)/2*(b-a), 6**(0.5)/6*(b-a), 3**(0.5)/3*(2*a+b)])
        a2 = np.array([2**(0.5)/2*(a-b), 6**(0.5)/6*(b-a), 3**(0.5)/3*(2*a+b)])
        a3 = np.array([0               , 6**(0.5)/3*(a-b), 3**(0.5)/3*(2*a+b)])
        h1 = a2 - a1
        h2 = a3 - a1
        h3 = (a1 + a2 + a3)
    else:
        h1 = cell[0]
        h2 = cell[1]
        h3 = cell[2]
    up = mag
    if afm == True:
```

```
        down = -mag
    else:
        down = mag
    slab = Atoms(cell=(h1, h2, h3 / 6 * layers))
    if fixlayers > 0:
        mid_layer = layers // 2 + base
        fixlayers = [mid_layer - fixlayers // 2, mid_layer + fixlayers // 2]
    else:
        fixlayers = [-1, -1]
    # Now add the layers
    for i in range(base, base + layers):
        j = i // 12
        if i >= fixlayers[0] and i <= fixlayers[1]:
            tag = 1
        else:
            tag = 0
        if i % 12 == 0:
            layer = Atoms([Atom(symbol[0], h3*2*j, magmom=up, tag=tag)])
        elif i % 12 == 1:
            layer = Atoms([Atom(symbol[1], h1/3 + h2/3 + h3*(1/6 + 2*j), magmom=0, tag=tag)])
        elif i % 12 == 2:
            layer = Atoms([Atom(symbol[0], 2*h1/3 + 2*h2/3 + h3*(1/3 + 2*j), magmom=down, tag=tag)])
        elif i % 12 == 3:
            layer = Atoms([Atom(symbol[1], h3*(1/2 + 2*j), magmom=0, tag=tag)])
        elif i % 12 == 4:
            layer = Atoms([Atom(symbol[0], h1/3 + h2/3 + h3*(2/3 + 2*j), magmom=up, tag=tag)])
        elif i % 12 == 5:
            layer = Atoms([Atom(symbol[1], 2*h1/3 + 2*h2/3 + h3*(5/6 + 2*j), magmom=0, tag=tag)])
        elif i % 12 == 6:
            layer = Atoms([Atom(symbol[0], h3*(2*j + 1), magmom=down, tag=tag)])
        elif i % 12 == 7:
            layer = Atoms([Atom(symbol[1], h1/3 + h2/3 + h3*(7/6 + 2*j), magmom=0, tag=tag)])
        elif i % 12 == 8:
            layer = Atoms([Atom(symbol[0], 2*h1/3 + 2*h2/3 + h3*(4/3 + 2*j), magmom=up, tag=tag)])
        elif i % 12 == 9:
            layer = Atoms([Atom(symbol[1], h3*(3/2 + 2*j), magmom=0, tag=tag)])
        elif i % 12 == 10:
            layer = Atoms([Atom(symbol[0], h1/3 + h2/3 + h3*(5/3 + 2*j), magmom=down, tag=tag)])
        elif i % 12 == 11:
            layer = Atoms([Atom(symbol[1], 2*h1/3 + 2*h2/3 + h3*(11/6 + 2*j), magmom=0, tag=tag)])
        slab.extend(layer)

    slab = slab * (area[0], area[1], 1)
    if vacuum != 0:
        slab.center(vacuum=vacuum, axis=2)
    if fixlayers > 0:
        c = FixAtoms(indices=[atom.index for atom in slab if atom.tag == 1])
        slab.set_constraint(c)
    return slab

def add_adsorbate(slab_original, adsorbate, height, position=None, mol_index=None,
                  top=True, fix_relax=False):
    """Add an adsorbate to a surface

    This function adds an adsorbate to a slab. If the adsorbate is a molecule,
    the atom indexed by the mol_index optional argument is positioned on top
    the adsorption position on the surface if top is true. If not, then it is
    positioned on the bottom (useful for symmetric slabs).

    Parameters:
    -----------

    slab: Atoms object
        The slab onto which the adsorbate should be added
    adsorbate: Atoms object
        The adsorbate you wish to adsorb on
    height: flt
        Height above (or below) the surface
```

```
    position: tuple
        The x-y position of the adsorbate
    mol_index (default 0): int
        If the adsorbate is a molecule, index of the atom to be positioned
        above the location specified by the position argument. If the argument
        is none, the origin is taken as the place where the adsorbate should
        be attached.

    """
    slab = slab_original.copy()

    # Get the atom on the surface we want to attach the atom to
    if top == True:
        a = slab.positions[:, 2].argmax()
    else:
        a = slab.positions[:, 2].argmin()

    # Get the x,y position to attach the adsorbate to
    if position is None:
        pos = np.array((slab.positions[a,0], slab.positions[a,1]))
    else:
        pos = np.array(position) # (x, y) par

    # Convert the adsorbate to an Atoms object
    if isinstance(adsorbate, Atoms):
        ads = adsorbate.copy()
    elif isinstance(adsorbate, Atom):
        ads = Atoms([adsorbate])
    else:
        raise TypeError('Adsorbate needs to be an Atom or Atoms object')

    # Flip the adsorbate if its supposed to be on the bottom
    if top == False:
        ads.positions = -ads.positions

    # Get the z-coordinate and attach it
    if top == True:
        z = slab.positions[a, 2] + height
    else:
        z = slab.positions[a, 2] - height
    if mol_index is not None:
        ads.translate([pos[0], pos[1], z] - ads.positions[mol_index])
    else:
        ads.translate([pos[0], pos[1], z])
    slab.extend(ads)

    return slab

def rotate_111(atoms):
    '''The point of this function is to rotate rocksalt/spinel surfaces
    in the 111 direction so that the eg and t2g orbitals accurately
    represent the correct projections'''

    scaled_pos = atoms.get_scaled_positions()
    cell = atoms.get_cell()
    a1 = cell[0]
    a2 = cell[1]
    a3 = cell[2]

    r1 = np.array([[-np.sqrt(2)/2, -np.sqrt(2)/2, 0.],
                   [np.sqrt(2)/2,  -np.sqrt(2)/2, 0.],
                   [0         , 0         , 1]])
    r2 = np.array([[1, 0              , 0              ],
                   [0, -1/np.sqrt(3)     , -np.sqrt(2)/np.sqrt(3)],
                   [0, np.sqrt(2)/np.sqrt(3), -1/np.sqrt(3)      ]])

    a1 = np.dot(r1, np.dot(r2, a1))
    a2 = np.dot(r1, np.dot(r2, a2))
```

```
    a3 = np.dot(r1, np.dot(r2, a3))
    atoms.set_cell((a1, a2, a3))
    atoms.set_scaled_positions(scaled_pos)

    return atoms
```

3. Bulk Properties

3.1. Introduction

In addition to calculating the equilibrium lattice constants needed for constructing surface slabs for adsorption energy calculations, we also needed to identify the equilibrium magnetic ordering of the rock salt monoxides. Rocksalt monoxides can have both a ferromagnetic or an anti-ferromagnetic ordering of spins. We compared the relative stability of both of these spin orderings for all our structures.

3.2. Calculation of the bulk metal and rocksalt AFM monoxide structures

We first need to perform bulk relaxations on the metals and their monoxides. For the fcc metals, we will use the primitive cell. For the monoxides, we will first use the anti-ferromagnetic configuration and then calculate the ferromagnetic configuration in the next section. Because rocksalt cells with an anti-ferromagnetic magnetic ordering are known to slightly distort in the (111) direction (normal to the planes of metal with parallel spins), we will allow the relaxations to change shape as well.

The parameters we want to get out of these calculations is the structure. For the fcc metals, we require only a lattice constant 'a'. For the rhombohedrally distorted rocksalt structures, we need the volume and the amount of rhombohedral distortion as well. This is captured through the 'a' and 'b' parameters in the rhombohedral, rocksalt unit cell. If b is 0, then there is no rhombohedral distortion.

$A_1 = (b, a, a)$
$A_2 = (a, b, a)$
$A_3 = (a, a, b)$

```
from jasp import *
from ase import Atom, Atoms
from ase.visualize import view
from mybulk import rocksalt

atoms = ('Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu')
mags = (3, 2, 6, 4, 3, 2)

ready = True

print '|Element|FCC Lattice|'
print '|-----|'

for metal, mag in zip(atoms, mags):
    bulk = Atoms([Atom(metal, (0, 0, 0))],
                 cell=((0, 2, 2),
                       (2, 0, 2),
                       (2, 2, 0)))
```

```
    # view(bulk)
    # continue
    with jasp('bulk/{0}'.format(metal), atoms=bulk,
              xc='PBE', lreal=False,
              encut=520, prec='Accurate', ediff=1e-5,
              kpts=(8, 8, 8), ismear=1, sigma=0.05,
              ibrion=2, isif=7, nsw=50,
              ispin=2, magmom=[mag], lorbit=11,
              lwave=False) as calc:
        try:
            calc.calculate()
            print '|{0}|{1:1.3f}|'.format(metal, bulk.get_cell()[0][1] * 2)
        except (VaspRunning, VaspQueued, VaspSubmitted):
            print calc.vaspdir, 'running'
            ready = False

print '\n|Element|AFM Rocksalt - a|AFM Rocksalt - b|'
print '|---|'

for metal, mag in zip(atoms, mags):
    bulk = rocksalt((metal, 'O'), a=4.3, mags=(mag, 0))
    with jasp('bulk/{0}O'.format(metal), atoms=bulk,
              xc='PBE', lreal=False,
              encut=520, prec='Accurate',
              kpts=(7, 7, 7), ismear=1, sigma=0.05, gamma=True,
              ibrion=1, isif=6, nsw=50, ediffg=-0.05,
              ispin=2, nupdown=0, lorbit=11,
              lwave=False) as calc:
        try:
            calc.calculate()
            a = bulk.get_cell()[0][0]/2
            b = bulk.get_cell()[0][1] - a
            print '|{0}|{1:1.3f}|{2:1.3f}|'.format(metal, a, b)
        except (VaspSubmitted, VaspQueued, VaspRunning):
            print calc.vaspdir, 'running'
            ready = False
```

Table 1: Lattice constants of metals in fcc structure

Element	FCC Lattice
Ti	4.093
V	3.794
Cr	3.614
Mn	3.496
Fe	3.479
Co	3.517
Ni	3.522
Cu	3.634

3.3. Calculation of FM bulk structure properties

Calculating the total energies of the ferromagnetic (FM) bulk structures is similar to the AFM bulk structures. Again, all calculations include a complete cell and ion relaxation.

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

Table 2: Unit cell parameters of rocksalt structures

Element	AFM Rocksalt - a	AFM Rocksalt - b
Ti	2.145	0.002
V	2.007	-0.249
Mn	2.195	-0.048
Co	2.209	0.213
Ni	2.095	-0.009
Cu	2.121	-0.003

```
3   from mybulk import rocksalt
4
5   atoms = ('Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu')
6   mags = (3, 4, 5, 6, 5, 4, 3, 2)
7
8   print '|Element|FM Rocksalt - a|FM Rocksalt - b|'
9   print '|---|'
10
11  for metal, mag in zip(atoms, mags):
12      bulk = rocksalt((metal, 'O'), a=4.3, mags=(mag, 0), afm=False)
13      with jasp('bulk/{0}O-FM'.format(metal), atoms=bulk,
14                xc='PBE', lreal=False,
15                encut=520, prec='Accurate',
16                kpts=(7, 7, 7), ismear=1, sigma=0.05, gamma=True,
17                ibrion=1, isif=6, nsw=50, ediffg=-0.05,
18                ispin=2, lorbit=11,
19                lwave=False) as calc:
20          try:
21              calc.calculate()
22              a = bulk.get_cell()[0][1]
23              b = 0
24              print '|{0}|{1:1.3f}|{2:1.3f}|'.format(metal, a, b)
25          except (VaspSubmitted, VaspQueued, VaspRunning):
26              print calc.vaspdir, 'running'
27              ready = False
```

Table 3: List of ferromagnetic structural properties

Element	FM Rocksalt - a	FM Rocksalt - b
Ti	2.321	0.000
V	2.281	0.000
Cr	2.263	0.000
Mn	2.387	0.000
Fe	2.296	0.000
Co	2.248	0.000
Ni	2.239	0.000
Cu	2.263	0.000

3.4. Comparison of AFM and FM magnetic orderings

We now wanted to compare the energy between the FM and AFM magnetic orderings for all the bulk monoxides we tested. The goal of this was to determine

which magnetic ordering was more stable and therefore to be used to model surfaces.

```
from jasp import *

atoms = ('Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu')

print '|Rocksalt|AFM Energy (eV/MO)|FM Energy (eV/MO)|'
print '|--|'

for atom in atoms:
    with jasp('bulk/{atom}O'.format(**locals())) as calc:
        afm_energy = calc.read_energy()[0] / 2
    with jasp('bulk/{atom}O-FM'.format(**locals())) as calc:
        fm_energy = calc.read_energy()[0]
    print '|{atom}O|{afm_energy}|{fm_energy}|'.format(**locals())
```

Table 4: Comparison between bulk AFM and FM stabilities

Rocksalt	AFM Energy (eV/MO)	FM Energy (eV/MO)
TiO	-17.289	-16.993
VO	-17.157	-17.027
MnO	-16.624	-16.427
CoO	-13.307	-13.380
NiO	-11.746	-11.661
CuO	-9.497	-9.527

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

atoms, afm, fm = zip(*data)
electrons = (2, 3, 5, 7, 8, 9)

fig = plt.figure(1, (5.5, 4.5))
ax = fig.add_axes((0.15, 0.15, 0.7, 0.7))
ax.plot(electrons, fm, marker='o', label='Ferromagnetic')
ax.plot(electrons, afm, marker='o', label='Anti-ferromagnetic')
ax.set_xlim((1, 10))
ax.set_xlabel('Number of electrons')
ax.set_ylabel('Total energy (eV)')
ax.set_xticks(np.linspace(1, 10, 10))
ax.annotate('', xy=(7.7, -10), xytext=(8.7, -10), arrowprops=dict(arrowstyle='->'))
ax.legend(loc=2, prop={'size':'small'})
ax1 = ax.twiny()
ax1.set_xlabel('Species')
ax1.set_xlim(1, 10)
ax1.set_xticks(electrons)
ax1.set_xticklabels(atoms)
ax2 = ax.twinx()
ax2.axhline(0, ls='--', c='k')
ax2.plot(electrons, np.array(afm) - np.array(fm), marker='o', c='r',
         label=r'$E_{AFM} - E_{FM}$')
ax.annotate('', xy=(5, -15.5), xytext=(4, -15.5), arrowprops=dict(arrowstyle='->'))
ax2.set_ylabel('Total energy (eV)')
ax2.set_ylim(-0.35, 0.35)
ax2.set_xlim(2, 9)
ax2.legend(loc=4, prop={'size':'small'})
fig.savefig('figures/fm-vs-afm-bulk-energies.pdf')
plt.show()
```

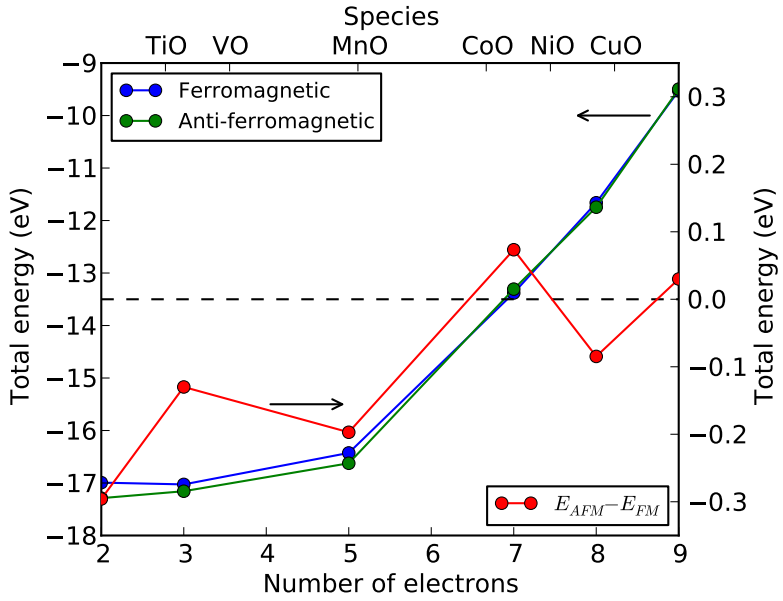

Figure 1: Comparison between ferromagnetic and anti-ferromagnetic bulk stabilities of monoxides

3.5. Summary

We decided to model all rocksalt monoxide (111) surfaces with an AFM ordering and alternating (111) FM planes with opposite magnetic spins. Most equilibrium volumes are within 10% of experimental values, with bulk Mn and VO being notable exceptions. We note that we forced the crystal structure of all metals to be FCC in order to perform our structural perturbations, which is not always the most stable structure of the metal (Ti, V, Mn, Co). However, we still attain good agreement between the experimental and the calculated equilibrium volume. We also compared the total energies of oxides with both ferromagnetic (FM) and anti-ferromagnetic (AFM) ordering. We found that for a majority of the calculations, the AFM ordering was energetically more stable by as much as 0.3 eV per metal atom, and for those materials where the FM orderings were more stable, they more stable by less than 0.1 eV per metal atom. Recent studies on the surface stability of corundum with the (111) crystal facet have shown that the overall AFM magnetic ordering composed with alternating (111) FM planes with opposite magnetic spins was most energetically favorable out of many magnetic orderings [7, 8].

Table 5: Comparison between calculated and experimentally measured equilibrium volumes. All volumes are in $\AA^3/metalatom$. Experimental crystal structures and lattice constants for the metal were attained from reference [1].

Element	Exp Crystal Structure	$V_{calc,M}$	$V_{exp,M}$	$V_{calc,MO}$	$V_{exp,MO}$
Ti	hcp	17.145	17.668	19.718	18.272 (Ref. [2])
V	bcc	13.654	13.850	19.180	16.768 (Ref. [3])
Mn	cubic complex	10.691	12.225	21.860	21.966 (Ref. [4])
Co	hcp	10.880	11.148	18.467	19.256 (Ref. [5])
Ni	fcc	10.922	10.941	18.502	18.141 (Ref. [6])
Cu	fcc	11.993	11.834	19.130	N/A

4. Adsorption Energies

The adsorption energies were all calculated on symmetric slabs that contained six metal layers. For rocksalt monoxides, there were also five layers of oxygen in between the metal layers. In this sense, the surfaces can be seen as reduced surfaces. Below is all the code needed to both perform and analyze these calculations

4.1. Calculating adsorption energies

All of the lattice parameters used in these calculations are taken from the tables shown above

4.1.1. Calculating the adsorption energy on metals

```
from jasp import * # JASP module to interface python with VASP
JASPRC['queue.ppn'] = 16 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '16GB' # Amount of RAM used by each calculation
from ase import Atom, Atoms
from ase.lattice.surface import fcc111
from ase.visualize import view
from ase.constraints import FixScaled, FixAtoms
from mysurfaces import add_adsorbate # My own helper function for adding adsorbates

# Construct clean slabs using the ase.lattice.surface.fcc111 helper function
Ti_slab = fcc111('Ti', size=(2, 2, 6), vacuum=10.0, a=4.093)
V_slab = fcc111('V', size=(2, 2, 6), vacuum=10.0, a=3.794)
Cr_slab = fcc111('Cr', size=(2, 2, 6), vacuum=10.0, a=3.614)
Mn_slab = fcc111('Mn', size=(2, 2, 6), vacuum=10.0, a=3.496)
Fe_slab = fcc111('Fe', size=(2, 2, 6), vacuum=10.0, a=3.479)
Co_slab = fcc111('Co', size=(2, 2, 6), vacuum=10.0, a=3.517)
Ni_slab = fcc111('Ni', size=(2, 2, 6), vacuum=10.0, a=3.522)
Cu_slab = fcc111('Cu', size=(2, 2, 6), vacuum=10.0, a=3.634)

# Create the adsorbate object
ads = Atom('O')

# These are the fractional coordinates of the adsorbate sites on both the top and bottom
# of the symmetric slab
top_ontop = (0, 0)
bot_ontop = (2./3., 2./3.)
top_fcc = (1./6., 1./6.)
bot_fcc = (1./2., 1./2.)
top_hcp = (1./3., 1./3.)
bot_hcp = (1./3., 1./3.)

```

```
# List of site names needed for naming the calculations
sites = ('ontop', 'fcc', 'hcp')

for atom in ('Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu'):
    # Copy the clean slab from the original slab so the original won't change
    # during the calculation
    slab = eval('{0}_slab'.format(atom)).copy()

    # Let the top layers relax and fix the middle two layers
    constraint = FixAtoms(mask=[(element.tag == 3) or (element.tag == 4)
                                for element in slab])
    slab.set_constraint(constraint)

    # Perform the calculation of the clean slab
    with jasp('sym-surface-relax-metal-6lay/{0}-clean'.format(atom), atoms=slab,
              xc='PBE', lreal=False,
              encut=520, nelm=100, nelmin=6,
              prec='Accurate', ediff=1e-6, addgrid=True,
              kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
              ibrion=2, isif=2, ediffg=-0.05, nsw=50,
              amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
              ispin=2, lorbit=11,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            calc.calculate()
        except (VaspRunning, VaspQueued, VaspSubmitted):
            print calc.vaspdir, 'running'

    # Perform the calculation with the adsorbate on each site
    for pos, site in zip(((top_ontop, bot_ontop),
                          (top_fcc, bot_fcc),
                          (top_hcp, bot_hcp)), sites):
        slab = eval('{0}_slab'.format(atom)).copy()
        cell = slab.get_cell()

        # Depending on the site, there should be a different starting height
        # of the adsorbate for a faster relaxation
        if site in ('fcc', 'hcp'):
            h = 1.1
        else:
            h = 1.7

        # Make the atomic coordinates of the adsorption sites
        h1_top = pos[0][0] * cell[0]
        h2_top = pos[0][1] * cell[1]
        h1_bot = pos[1][0] * cell[0]
        h2_bot = pos[1][1] * cell[1]
        slab = add_adsorbate(slab, ads, height=h, position=h1_top + h2_top, top=True)
        slab = add_adsorbate(slab, ads, height=h, position=h1_bot + h2_bot, top=False)

        # We have to set a new constraint to ensure the adsorbate will relax too
        constraint = FixAtoms(mask=[(element.tag == 3) or (element.tag == 4)
                                    for element in slab])
        slab.set_constraint(constraint)

        # Now perform the calculation
        with jasp('sym-surface-relax-metal-6lay/{0}-{1}'.format(atom, site), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=520, nelm=100, nelmin=6,
                  prec='Accurate', ediff=1e-6, addgrid=True,
                  kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
                  ibrion=1, nsw=50, isif=2, ediffg=-0.05,
                  amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
                  ispin=2, lorbit=11,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
```

```
                calc.calculate()
            except (VaspRunning, VaspQueued, VaspSubmitted):
                print calc.vaspdir, 'running'
```

4.1.2. Calculating the adsorption energy on oxides

```
from jasp import * # JASP module to interface python with VASP
JASPRC['queue.ppn'] = 16 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '16GB' # Amount of RAM used by each calculation
from mysurfaces import rocksalt111 # Needed to create the slab
from ase.visualize import view
from mysurfaces import add_adsorbate
from ase.constraints import FixScaled

atoms = ('Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu')
structs = ((2.145, 2.007, 2.101, 2.195, 2.46, 2.209, 2.095, 2.121),
           (0.002, -0.249, -0.135, -0.048, 0.212, 0.213, -0.009, -0.003))

# Magnetic moments are one more than the number of unpaired d-electrons per atom
mags = (3, 4, 5, 6, 5, 4, 3, 2)

# Create the adsorbate object
ads = Atom('O', magmom=0)

# These are the fractional coordinates of the adsorbate sites on both the top and bottom
# of the symmetric slab
top_ontop = (0, 0)
bot_ontop = (5./6., 5./6.)
top_fcc = (1./3., 1./3.)
bot_fcc = (1./2., 1./2.)
top_hcp = (1./6., 1./6.)
bot_hcp = (2./3., 2./3.)

# List of site names needed for naming the calculations
sites = ('ontop', 'fcc', 'hcp')

for metal, a, b, m in zip(atoms, structs[0], structs[1], mags):
    # Create the clean slab needed for each calculation
    slab_main = rocksalt111((metal, 'O'), a, b, vacuum=10, mag=m, layers=11,
                            fixlayers=3, base=2, area=(2,2))

    # Create a copy of the original slab
    slab = slab_main.copy()

    # Perform the calculation
    with jasp('sym-surface-relax-z-6lay/{0}O-clean'.format(metal), atoms=slab,
              xc='PBE', lreal=False,
              encut=520, nelm=100, nelmin=6,
              prec='Accurate', ediff=1e-6, addgrid=True,
              kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
              ispin=2, lorbit=11, nupdown=0,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05,
              amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            calc.calculate()
        except (VaspSubmitted, VaspQueued, VaspRunning):
            print calc.vaspdir, 'running'
            ready = False

    for name, pos in zip(('ontop', 'fcc', 'hcp'),
                         ((top_ontop, bot_ontop),
                          (top_fcc, bot_fcc),
                          (top_hcp, bot_hcp))):

        # Create another copy of the original clean slab for adding adsorbates on
```

```
        slab = slab_main.copy()

        # Make atomic coordinates of the adsorption sites
        cell = slab.get_cell()
        h1_top = pos[0][0] * cell[0]
        h2_top = pos[0][1] * cell[1]
        h1_bot = pos[1][0] * cell[0]
        h2_bot = pos[1][1] * cell[1]

        # Depending on the site, there should be a different starting height
        # of the adsorbate for a faster relaxation
        if name == 'ontop':
            h = 1.6
        else:
            h = 1.0
        slab = add_adsorbate(slab, ads, height=h, position=h1_top + h2_top, top=True)
        slab = add_adsorbate(slab, ads, height=h, position=h1_bot + h2_bot, top=False)

        # Now we want to set constraints so everything only relaxes in one direction
        # When constructing the slabs, the atoms that are told to be fixed are tagged as 1
        # The rest of the atoms in the surface are 2, and the added adsorbate is 0
        c = []
        for atom in slab:
            if atom.tag == 1:
                c.append(FixScaled(cell=slab.cell, a=atom.index, mask=(1, 1, 1)))
            elif (atom.tag == 0 or atom.tag == 2):
                c.append(FixScaled(cell=slab.cell, a=atom.index, mask=(1, 1, 0)))
            else:
                c.append(FixScaled(cell=slab.cell, a=atom.index, mask=(0, 0, 0)))
        slab.set_constraint(c)

        # Perform the calculation
        with jasp('sym-surface-relax-z-6lay/{0}O-{1}'.format(metal, name), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=520, nelm=100, nelmin=6,
                  prec='Accurate', ediff=1e-6, addgrid=True,
                  kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
                  ibrion=2, isif=2, nsw=50, ediffg=-0.05,
                  amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
                  ispin=2, lorbit=11, nupdown=0,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspSubmitted, VaspQueued, VaspRunning):
                print calc.vaspdir, 'running'
                ready = False
```

4.1.3. Calculating the adsorption energy on expanded metals

```
from jasp import * # JASP module to interface python with VASP
from ase.visualize import view
from ase.constraints import FixAtoms
JASPRC['queue.ppn'] = 16 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '36GB' # Amount of RAM used by each calculation

atoms = ('Ti', 'V', 'Cr', 'Mn', 'Co', 'Ni', 'Cu')
sites = ('ontop', 'fcc', 'hcp')
print '|Element|Adsorption Site|Total Energy|'
print '|--|'

for atom in atoms:
    # First perform the total energy calculation of the clean slab.
    # We do this by first copying the clean oxide slab and then removing
    # all of the surface oxygen
    with jasp('sym-surface-relax-z-6lay/{atom}O-clean'.format(**locals())) as calc:
        slab = calc.get_atoms()
```

```
    for iatom, a in reversed(list(enumerate(slab))):
        if (a.symbol == 'O'):
            slab.pop(iatom)
    magmoms = 2 * np.ones(len(slab))
    slab.set_initial_magnetic_moments(magmoms)

    # Perform the calculation on the clean slab
    with jasp('sym-surface-expanded/{atom}O-clean'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False, lmaxmix=4,
              encut=520, nelm=500, nelmin=6,
              prec='Accurate', ediff=1e-4,
              kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
              ispin=2, lorbit=11,
              ibrion=-1,
              amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
              npar=2, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            energy = slab.get_potential_energy()
            print '|{atom}|clean|{energy}|'.format(**locals())
        except (VaspSubmitted, VaspQueued, VaspRunning):
            print '|{atom}|clean|running|'.format(**locals())

    # Now perform the calculation on the expanded metal lattice with the adsorbate
    for site in sites:
        # We first want to get a copy of the surface slab with the adsorbate of the oxides
        with jasp('sym-surface-relax-z-6lay/{atom}O-{site}'.format(**locals())) as calc:
            slab = calc.get_atoms()

        # This removes the constraints so we can remove atoms on it
        slab.set_constraint()
        l = len(slab)

        # Now we remove all of the oxygen atoms in the slab
        for iatom, a in reversed(list(enumerate(slab))):
            if (a.symbol == 'O' and iatom not in (l - 2, l - 1)):
                slab.pop(iatom)

        # We add magnetic moments onto the metal elements.
        # We assume ferromagnetic ordering
        magmoms = []
        for a in slab:
            if a.symbol == 'O':
                magmoms.append(0)
            else:
                magmoms.append(2)
        slab.set_initial_magnetic_moments()

        # We constrain all of the metal atoms
        constraint = FixAtoms(mask=[element.symbol == atom for element in slab])
        slab.set_constraint(constraint)

        # Perform the calculation
        with jasp('sym-surface-expanded-relax-ads/{atom}O-{site}'.format(**locals()), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=520, nelm=500, nelmin=6, lmaxmix=4,
                  prec='Accurate', ediff=1e-5, addgrid=True,
                  kpts=(7, 7, 1), ismear=1, sigma=0.05, gamma=True,
                  ispin=2, lorbit=11,
                  ibrion=1, nsw=50, isif=2, ediffg=-0.05,
                  amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                energy = slab.get_potential_energy()
                print '|{atom}|{site}|{energy}|'.format(**locals())
            except (VaspSubmitted, VaspQueued, VaspRunning):
                print '|{atom}|{site}|running|'.format(**locals())
```

4.2. Tables of adsorption energies

Table 6: Metal Adsorption Energies

Atom	Site	Adsorption Energy
Ti	ontop	-3.735
Ti	fcc	-6.407
Ti	hcp	-6.197
V	ontop	-4.124
V	fcc	-6.736
V	hcp	-6.634
Cr	ontop	-4.134
Cr	fcc	-5.719
Cr	hcp	-5.903
Mn	ontop	-3.034
Mn	fcc	-4.437
Mn	hcp	-4.716
Fe	ontop	-3.397
Fe	fcc	-4.071
Fe	hcp	-4.508
Co	ontop	-1.851
Co	fcc	-4.153
Co	hcp	-3.682
Ni	ontop	-1.082
Ni	fcc	-3.007
Ni	hcp	-2.907
Cu	ontop	-0.429
Cu	fcc	-2.346
Cu	hcp	-2.196

4.3. Note on adsorption energies on FeO and CrO

After starting calculations of the adsorption energies on oxides, we found that the surface structures of FeO and CrO with and without adsorbates would undergo a large amount of surface reconstruction. This behavior included subsurface oxygen molecules rising to the surface and, in the case of some FeO structures, a complete switching of the top two layers of the surface. We could not identify neither the source of this behavior nor a solution. We therefore did not include any of the data on Fe and Cr surfaces in our analysis.

4.4. Graphing the adsorption energies

This is the code for creating the graph of the adsorption energies (Figure 2) found in the paper.

Table 7: Six layer oxide adsorption energies

Atom	Site	Adsorption Energy
Ti	ontop	-3.874
Ti	fcc	-5.858
Ti	hcp	-6.187
V	ontop	-3.958
V	fcc	-4.182
V	hcp	-5.339
Mn	ontop	-2.400
Mn	fcc	-4.253
Mn	hcp	-5.312
Co	ontop	-2.106
Co	fcc	-3.732
Co	hcp	-3.864
Ni	ontop	-1.565
Ni	fcc	-2.916
Ni	hcp	-3.843
Cu	ontop	-0.612
Cu	fcc	-2.556
Cu	hcp	-3.779

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

# First, graph all of the adsorption energies together
group = (4, 5, 7, 9, 10, 11)

ontop_metals, ontop_oxides, ontop_expanded = [], [], []
fcc_metals, fcc_oxides, fcc_expanded = [], [], []
hcp_metals, hcp_oxides, hcp_expanded = [], [], []
atoms = ['Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu']

for atom, site, energy in metal:
    if atom in ('Cr', 'Fe'):
        continue
    if site == 'ontop':
        ontop_metals.append(energy)
    elif site == 'fcc':
        fcc_metals.append(energy)
    elif site == 'hcp':
        hcp_metals.append(energy)

for atom, site, energy in oxide:
    if site == 'ontop':
        ontop_oxides.append(energy)
    elif site == 'fcc':
        fcc_oxides.append(energy)
    elif site == 'hcp':
        hcp_oxides.append(energy)

for atom, site, energy in expanded:
    if site == 'ontop':
        ontop_expanded.append(energy)
    elif site == 'fcc':
        fcc_expanded.append(energy)
```

Table 8: Six layer expanded metal adsorption energies

Atom	Site	Adsorption Energy
Ti	ontop	-3.898
Ti	fcc	-6.354
Ti	hcp	-6.275
V	ontop	-4.160
V	fcc	-4.945
V	hcp	-5.686
Mn	ontop	-1.594
Mn	fcc	0.803
Mn	hcp	-1.119
Co	ontop	-1.404
Co	fcc	-1.583
Co	hcp	-1.552
Ni	ontop	-1.505
Ni	fcc	-3.166
Ni	hcp	-3.509
Cu	ontop	-0.265
Cu	fcc	-2.206
Cu	hcp	-1.745

```
    elif site == 'hcp':
        hcp_expanded.append(energy)

atoms = ['Ti', 'V', '', 'Mn', '', 'Co', 'Ni', 'Cu']

fig = plt.figure(1, (9, 9))
ax11 = fig.add_axes((0.1, 0.51, 0.39, 0.39))
ax11.plot(group, ontop_metals, marker='o', c='b', label='Metal top')
ax11.plot(group, fcc_metals, marker='o', c='c', label='Metal fcc')
ax11.plot(group, hcp_metals, marker='o', c='m', label='Metal hcp')
ax11.plot(group, ontop_oxides, marker='^', c='b', ls='--', label='Oxide top')
ax11.plot(group, fcc_oxides, marker='^', c='c', ls='--', label='Oxide fcc')
ax11.plot(group, hcp_oxides, marker='^', c='m', ls='--', label='Oxide hcp')
ax11.text(0.9, 0.9, 'a)', fontsize=20, transform=ax11.transAxes)
ax11.set_ylabel(r'E$_{ads}$ (eV/O)')
ax11.set_ylim(-7, 2)
ax11.set_yticks((-6, -5, -4, -3, -2, -1, 0, 1))
ax11.set_xlim(3, 12)
ax11.set_xticklabels([], [])
ax11x = ax11.twiny()
ax11x.set_yticks((-6, -5, -4, -3, -2, -1, 0, 1))
# ax11x.set_xlabel(r'Number of £d£-electrons')
ax11x.set_xlim(ax11.get_xlim())
ax11x.set_xticks((4, 5, 6, 7, 8, 9, 10, 11))
ax11x.set_xticklabels(atoms)
ax11.legend(loc=2, prop={'size':'small'}, ncol=2, numpoints=1)

ax121 = fig.add_axes((0.51, 0.705, 0.39, 0.195))
ax121.plot(group, ontop_metals, marker='o', label='Metal top', color='b')
ax121.plot(group, ontop_expanded, marker='s', label='Expanded top', color='b', ls=':')
ax121.plot(group, ontop_oxides, marker='^', label='Oxide top', color='b', ls='--')
ax121.set_xlim((3, 12))
ax121.set_ylim((-7, 4))
ax121.text(0.9, 0.82, 'b)', fontsize=20, transform=ax121.transAxes)
ax121.set_yticklabels([], [])
```

```
ax121.set_xticklabels([], [])
ax121.legend(loc=2, prop={'size':'small'}, numpoints=1)
ax121x = ax121.twiny()
ax121x.set_xlim((3, 12))
ax121x.set_yticklabels([], [])
ax121x.set_yticks((2, 0, -2, -4, -6))
# ax121x.set_xlabel(r'Number of £d£-electrons')
ax121x.set_xticklabels(atoms)
ax121x.set_xticks((4, 5, 6, 7, 8, 9, 10, 11))
ax121y = ax121.twinx()
ax121y.set_xticklabels([], [])
ax121y.set_ylim(ax121.get_ylim())
ax121y.set_ylabel((r'E$_{ads}$ (eV/O)'))
ax121y.set_yticks((2, 0, -2, -4, -6))

ax122 = fig.add_axes((0.51, 0.51, 0.39, 0.195))
ax122.plot(group, np.array(ontop_expanded) - np.array(ontop_metals),
           marker='o', label='Expansion', color='g')
ax122.plot(group, np.array(ontop_oxides) - np.array(ontop_expanded),
           marker='o', label='Oxidation', color='r')
ax122.axhline(0, ls='--', c='k')
ax122.set_xlim((3, 12))
ax122.set_xticklabels([], [])
ax122.set_yticklabels([], [])
ax122.set_ylim((-6, 6))
ax122.legend(loc=3, prop={'size':'small'}, numpoints=1)
ax122y = ax122.twinx()
ax122y.set_ylim(ax122.get_ylim())
ax122y.set_yticks((-4, -2, 0, 2, 4))
ax122y.set_ylabel(r'$\Delta$ E$_{ads}$ (eV/O)')
ax122y.set_xticklabels([], [])

ax211 = fig.add_axes((0.1, 0.295, 0.39, 0.195))
ax211.plot(group, fcc_metals, marker='o', label='Metal fcc', color='c')
ax211.plot(group, fcc_expanded, marker='s', label='Expanded fcc', color='c', ls=':')
ax211.plot(group, fcc_oxides, marker='^', label='Oxide fcc', color='c', ls='--')
ax211.set_ylabel((r'E$_{ads}$ (eV/O)'))
ax211.set_ylim((-7, 4))
ax211.set_yticks((-6, -4, -2, 0, 2))
ax211.text(0.9, 0.82, 'c)', fontsize=20, transform=ax211.transAxes)
ax211.set_xlim(3, 12)
ax211.set_xticklabels([], [])
ax211.legend(loc=2, prop={'size':'small'}, numpoints=1)

ax212 = fig.add_axes((0.1, 0.1, 0.39, 0.195))
ax212.plot(group, np.array(fcc_expanded) - np.array(fcc_metals),
           marker='o', label='Expansion', color='g')
ax212.plot(group, np.array(fcc_oxides) - np.array(fcc_expanded),
           marker='o', label='Oxidation', color='r')
ax212.axhline(0, ls='--', c='k')
ax212.set_xlim((3, 12))
ax212.set_xticks((4, 5, 6, 7, 8, 9, 10, 11))
ax212.set_ylim((-6, 6))
ax212.set_yticks((-4, -2, 0, 2, 4))
ax212.set_ylabel(r'$\Delta$ E$_{ads}$ (eV/O)')
ax212.set_xlabel(r'Group on periodic table')
ax212.legend(loc=3, prop={'size':'small'}, numpoints=1)

ax221 = fig.add_axes((0.51, 0.295, 0.39, 0.195))
ax221.plot(group, hcp_metals, marker='o', label='Metal hcp', color='m')
ax221.plot(group, hcp_expanded, marker='s', label='Expanded hcp', color='m', ls=':')
ax221.plot(group, hcp_oxides, marker='^', label='Oxide hcp', color='m', ls='--')
ax221.set_xlim((3, 12))
ax221.set_yticklabels([], [])
ax221.set_xticklabels([], [])
ax221.set_ylim((-7, 4))
ax221.text(0.9, 0.82, 'd)', fontsize=20, transform=ax221.transAxes)
ax221.set_yticks((-6, -4, -2, 0, 2))
```

```
ax221.legend(loc=2, prop={'size':'small'}, numpoints=1)
ax221y = ax221.twinx()
ax221y.set_xticklabels([], [])
ax221y.set_ylim(ax221.get_ylim())
ax221y.set_ylabel((r'E$_{ads}$ (eV/O)'))
ax221y.set_ylim((-7, 4))
ax221y.set_yticks((-6, -4, -2, 0, 2))

ax222 = fig.add_axes((0.51, 0.1, 0.39, 0.195))
ax222.plot(group, np.array(hcp_expanded) - np.array(hcp_metals),
          marker='o', label='Expansion', color='g')
ax222.plot(group, np.array(hcp_oxides) - np.array(hcp_expanded),
          marker='o', label='Oxidation', color='r')
ax222.axhline(0, ls='--', c='k')
ax222.set_xlim((3, 12))
ax222.set_xticks((4, 5, 6, 7, 8, 9, 10, 11))
ax222.set_xlabel(r'Group on periodic table')
ax222.set_yticklabels([], [])
ax222.set_ylim((-6, 6))
ax222.legend(loc=3, prop={'size':'small'}, numpoints=1)
ax222y = ax222.twinx()
ax222y.set_xticks((4, 5, 6, 7, 8, 9, 10, 11))
ax222y.set_ylim(ax222.get_ylim())
ax222y.set_yticks((-4, -2, 0, 2, 4))
ax222y.set_ylabel(r'$\Delta$ E$_{ads}$ (eV/O)')

fig.savefig('figures/FIG2.pdf')
fig.savefig('figures/adsorption-results-final.png')
plt.show()
```

5. Density of states (DOS)

5.1. Calculate the DOS of the surfaces

5.1.1. Introduction

All calculations on the density of states on slabs were done with a higher sampling of k-points. For metals and expanded metals, the sampling was a 12 12 1 k-point grid, while for oxides we used a 10 10 1 k point grid. We also did a fixed rotation of all surfaces so that the orientation of the projected *t2g* and *eg* orbitals would correspond to where the bonds where. In oxides, the *eg* orbitals are characterized as directly overlapping with the oxygen p orbitals, and we performed a fixed rotation of the unit cell to capture this behavior. We note that after projecting the density of states onto the *t2g* and *eg* orbitals, we could find no meaningful correlation between adsorption energy and those aspects of the electronic structure that would fit in the scope of this paper.

5.1.2. Calculating the DOS of surface metals

```
from jasp import * # JASP module to interface python with VASP
JASPRC['queue.ppn'] = 8 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '16GB' # Amount of RAM used by each calculation
from ase import Atom, Atoms
from ase.lattice.surface import fcc111
from ase.visualize import view
from ase.constraints import FixScaled, FixAtoms
from mysurfaces import add_adsorbate, rotate_111

sites = ('ontop', 'fcc', 'hcp')

```

```
for atom in ('Ti', 'V',  'Mn', 'Co', 'Ni', 'Cu'):
    slab = eval('{0}_slab'.format(atom)).copy()
    # Copy the atoms object from the completed calculation
    with jasp('sym-surface-relax-metal-6lay/{atom}-clean'.format(**locals())) as calc:
        slab_clean = calc.get_atoms()
    slab_clean = rotate_111(slab_clean)

    # Perform the calculation with higher k-points and a fixed cell
    with jasp('dos/surf-{0}-clean'.format(atom), atoms=slab_clean,
              xc='PBE', lreal=False,
              encut=520, nelm=300, nelmin=6,
              prec='Accurate', ediff=1e-4,
              kpts=(12, 12, 1), ismear=1, sigma=0.05, gamma=True,
              ibrion=-1,
              amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
              ispin=2, lorbit=11,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            calc.calculate()
        except (VaspRunning, VaspQueued, VaspSubmitted):
            print calc.vaspdir, 'running'
    for site in sites:
        # Copy the atoms object from the completed calculation
        with jasp('sym-surface-relax-metal-6lay/{0}-{1}'.format(atom, site)) as calc:
            slab = calc.get_atoms()
        slab = rotate_111(slab)

        # Perform the calculation with higher k-points and a fixed cell
        with jasp('dos/surf-{0}-{1}'.format(atom, site), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=520, nelm=300, nelmin=6,
                  prec='Accurate', ediff=1e-4,
                  kpts=(12, 12, 1), ismear=1, sigma=0.05, gamma=True,
                  ibrion=-1,
                  amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
                  ispin=2, lorbit=11,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspRunning, VaspQueued, VaspSubmitted):
                print calc.vaspdir, 'running'
```

5.1.3. Calculating the DOS surface expanded Metals

```
from jasp import * # JASP module to interface python with VASP
JASPRC['queue.ppn'] = 8 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '12GB' # Amount of RAM used by each calculation
from mysurfaces import rotate_111

atoms = ('Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu')
sites = ('ontop', 'fcc', 'hcp')
print '|Element|Adsorption Site|Total Energy|'
print '|--|'

for atom in atoms:
    # Copy the atoms object from the completed calculation
    with jasp('sym-surface-relax-z-6lay/{atom}O-clean'.format(**locals())) as calc:
        slab = calc.get_atoms()
    for iatom, a in reversed(list(enumerate(slab))):
        if (a.symbol == 'O'):
            slab.pop(iatom)
    magmoms = 2 * np.ones(len(slab))
    slab.set_initial_magnetic_moments(magmoms)
    slab = rotate_111(slab)

```

```
    # Perform the calculation
    with jasp('dos/surf-{atom}-expanded-clean'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False, lmaxmix=4,
              encut=520, nelm=500, nelmin=6,
              prec='Accurate',
              kpts=(12, 12, 1), ismear=1, sigma=0.05, gamma=True,
              ispin=2, lorbit=11,
              ibrion=-1,
              amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
              npar=2, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            energy = slab.get_potential_energy()
            print '|{atom}|clean|{energy}|'.format(**locals())
        except (VaspSubmitted, VaspQueued, VaspRunning):
            print '|{atom}|clean|running|'.format(**locals())
    for site in sites:
        # Copy the atoms object from a completed calculation
        with jasp('sym-surface-expanded-relax-ads/{atom}O-{site}'.format(**locals())) as calc:
            slab = calc.get_atoms()
        magmoms = []
        for a in slab:
            if a.symbol == 'O':
                magmoms.append(0)
            else:
                magmoms.append(2)
        slab.set_initial_magnetic_moments(magmoms)
        slab = rotate_111(slab)

        # Perform the calculation
        with jasp('dos/surf-{atom}-expanded-{site}-relax-low-kpts'.format(**locals()), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=520, nelm=500, nelmin=6, lmaxmix=4,
                  prec='Accurate',
                  kpts=(12, 12, 1), ismear=1, sigma=0.05, gamma=True,
                  ispin=2, lorbit=11,
                  ibrion=-1,
                  amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
                  npar=2, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                energy = slab.get_potential_energy()
                print '|{atom}|{site}|{energy}|'.format(**locals())
            except (VaspSubmitted, VaspQueued, VaspRunning):
                print '|{atom}|{site}|running|'.format(**locals())
```

5.1.4. Calculating the DOS of surface rocksalts

```
from jasp import * # JASP module to interface python with VASP
JASPRC['queue.ppn'] = 8 # Number of processors to run each calculation on
JASPRC['queue.mem'] = '16GB' # Amount of RAM used by each calculation
from mysurfaces import rocksalt111
from ase.visualize import view
from mysurfaces import add_adsorbate, rotate_111
from ase.constraints import FixScaled
from shutil import rmtree

# In order to construct the arrays that hold the correct initial magnetic
# moments, we need to re-construct the slabs and then copy their magnetic
# moments

atoms = ('Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu')
structs = ((2.145, 2.007, 2.195, 2.209, 2.095, 2.121),
           (0.002, -0.249, -0.048, 0.213, -0.009, -0.003))
mags = (3, 4, 6, 4, 3, 2)
ads = Atom('O', magmom=0)
cwd = os.getcwd() + '/'
```

```
20
21  top_ontop = (0, 0)
22  bot_ontop = (5./6., 5./6.)
23  top_fcc = (1./3., 1./3.)
24  bot_fcc = (1./2., 1./2.)
25  top_hcp = (1./6., 1./6.)
26  bot_hcp = (2./3., 2./3.)
27
28  sites = ('ontop', 'fcc', 'hcp')
29
30  for metal, a, b, m in zip(atoms, structs[0], structs[1], mags):
31      # Get the initial magnetic moments
32      slab_main = rocksalt111((metal, 'O'), a, b, vacuum=10, mag=m, layers=11,
33                              fixlayers=3, base=2, area=(2,2))
34      slab = slab_main.copy()
35      magmoms = slab.get_initial_magnetic_moments()
36
37      # Get the correct relaxed structure of the clean slab
38      with jasp('sym-surface-relax-z-6lay/{0}O-clean'.format(metal)) as calc:
39          slab_clean = calc.get_atoms()
40      slab_clean = rotate_111(slab_clean)
41      slab_clean.set_initial_magnetic_moments(magmoms)
42      with jasp('dos/surf-{0}O-clean'.format(metal), atoms=slab_clean,
43                xc='PBE', lreal=False,
44                encut=520, nelm=200, nelmin=6,
45                prec='Accurate', ediff=1e-4,
46                kpts=(10, 10, 1), ismear=1, sigma=0.05, gamma=True,
47                ispin=2, lorbit=11, nupdown=0,
48                ibrion=-1,
49                amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
50                npar=4, nsim=4, lplane=True, lscalu=False,
51                lwave=False) as calc:
52          try:
53              calc.calculate()
54          except (VaspSubmitted, VaspQueued, VaspRunning):
55              print calc.vaspdir, 'running'
56      for name, pos in zip(('ontop', 'fcc', 'hcp'),
57                           ((top_ontop, bot_ontop),
58                            (top_fcc, bot_fcc),
59                            (top_hcp, bot_hcp))):
60          # First get the initial magnetic moments
61          slab = slab_main.copy()
62          cell = slab.get_cell()
63          h1_top = pos[0][0] * cell[0]
64          h2_top = pos[0][1] * cell[1]
65          h1_bot = pos[1][0] * cell[0]
66          h2_bot = pos[1][1] * cell[1]
67          if name == 'ontop':
68              h = 1.6
69          else:
70              h = 1.0
71          slab = add_adsorbate(slab, ads, height=h, position=h1_top + h2_top, top=True)
72          slab = add_adsorbate(slab, ads, height=h, position=h1_bot + h2_bot, top=False)
73          magmoms = slab.get_initial_magnetic_moments()
74
75          # Now get the relaxed structure of the slab with adsorbates on it
76          with jasp('sym-surface-relax-z-6lay/{0}O-{1}'.format(metal, name)) as calc:
77              slab = calc.get_atoms()
78          slab = rotate_111(slab)
79          slab.set_initial_magnetic_moments(magmoms)
80          with jasp('dos/surf-{0}O-{1}'.format(metal, name), atoms=slab,
81                    xc='PBE', lreal=False,
82                    encut=520, nelm=200, nelmin=6,
83                    prec='Accurate', ediff=1e-6, addgrid=True,
84                    kpts=(10, 10, 1), ismear=1, sigma=0.05, gamma=True,
85                    ibrion=-1,
86                    amix=0.2, amix_mag=0.8, bmix=0.0001, bmix_mag=0.0001,
87                    ispin=2, lorbit=11, nupdown=0,
```

```
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspSubmitted, VaspQueued, VaspRunning):
                print calc.vaspdir, 'running'
                ready = False
```

5.2. Analyze the DOS

5.2.1. Introduction

Below consists of the code to analyze the density of states and also the tables which contain the data. The centers of the bands are calculated using Equation 1, and all band widths are calculated using Equation 2.

$$E$$

$$_d = \int \rho E dE \overline{\int EdE(1)}$$

$$W$$

$$^2_d = \int \rho(E - E_d)^2 dE \overline{\int \rho dE(2)}$$

All band centers and widths of the surface projected metals are done for only the clean surface.

5.2.2. Analyzing the DOS of metals

```
from jasp import *
from ase.calculators.vasp import Vasp, VaspDos
import matplotlib.pyplot as plt

atoms, sites, ads_energies = zip(*data)

# These are the indexes of the surface atoms needed to be analyzed.
# I found these by visualizing the atom using the ase.visualize.view function
atoms = ('Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu')
a_index = (0, 1, 2, 3, 20, 21, 22, 23)
t2g_index = (8, 9, 10, 11, 14, 15)
eg_index = (12, 13, 16, 17)
op_index = (2, 3, 4, 5, 6, 7)
o_index = (24, 25)

sites = ('ontop', 'fcc', 'hcp')

print '|Atom|Ads site|d center|d width|dstates|eg center|t2g center|op center|Ads Energy|'
print '|--|'

i = 0

for atom in atoms:
    fig = plt.figure(1, (5, 6))
    # Calculate properties of the electronic structure of the clean surface
    with jasp('dos/surf-{atom}-clean'.format(**locals())) as calc:
        ados = VaspDos(efermi=calc.get_fermi_level()) # Get the density of states
        energies_clean = ados.energy # Get the list of energies and the density at each energy
        ind = (energies_clean < 5) & (energies_clean > -10) # The range of energies we're interested in
        energies_clean = energies_clean[ind] # The range of densities we're interested in
        eg = np.zeros(len(energies_clean)) # Create an array for the density
        t2g = np.zeros(len(energies_clean)) # Create an array for the density
        # Loop over each surface metal atom and each part of the d-band
```

```
        for ai in a_index:
            for t2gi in t2g_index:
                t2g += np.array((ados.site_dos(calc.resort[ai], t2gi)[ind]))
            for egi in eg_index:
                eg += np.array((ados.site_dos(calc.resort[ai], egi)[ind]))
        eg_states = np.trapz(eg, energies_clean) # Number of eg states
        t2g_states = np.trapz(t2g, energies_clean) # Number of t2g states
        eg_center = np.trapz(energies_clean * eg, energies_clean) / eg_states # eg center
        t2g_center = np.trapz(energies_clean * t2g, energies_clean) / t2g_states # t2g center
        d = t2g + eg
        d_states = np.trapz(d, energies_clean) # Number of d-states
        d_center = np.trapz(energies_clean * d, energies_clean) / d_states # d center
        d_centers = d_center * np.ones(len(energies_clean)) # We need this to calculate the width
        # Calculate the d-width
        d_width = np.sqrt(np.trapz((energies_clean-d_centers) ** 2 * d, energies_clean) / d_states)

    for site in sites:
        # We perform calculations on the adsorbate p-band center below
        with jasp('dos/surf-{atom}-{site}'.format(**locals())) as calc:
            ados = VaspDos(efermi=calc.get_fermi_level())
            energies = ados.energy
            ind = (energies < 5) & (energies > -10)
            energies = energies[ind]
            op = np.zeros(len(energies))
            for ai in o_index:
                for opi in op_index:
                    op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
        op_states = np.trapz(op, energies)
        op_center = np.trapz(energies * op, energies) / op_states
        energy = ads_energies[i]
        i += 1
        s = '|{0}|{1}|{2:1.3f}|{3:1.3f}|{4:1.3f}|{5:1.3f}|{6:1.3f}|{7:1.3f}|{8:1.3f}|'
        print s.format(atom, site, d_center, d_width, d_states, eg_center, t2g_center,
                       op_center, energy)
```

5.2.3. Analyzing the DOS of expanded metals

```
from jasp import *
from ase.calculators.vasp import Vasp, VaspDos
import matplotlib.pyplot as plt

atoms, sites, ads_energies = zip(*data)

# These are the indexes of the surface atoms needed to be analyzed.
# I found these by visualizing the atom using the ase.visualize.view function
atoms = ('Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu')
a_index = (0, 6, 12, 18, 5, 11, 17, 23)
o_index = (24, 25)
t2g_index = (8, 9, 10, 11, 14, 15)
eg_index = (12, 13, 16, 17)
op_index = (2, 3, 4, 5, 6, 7)

sites = ('ontop', 'fcc', 'hcp')

print '|Atom|Ads site|d center|d width|d states|eg center|t2g center|op center|Ads Energy|'
print '|--|'

i = 0

for atom in atoms:
    fig = plt.figure(1, (5, 6))
    # Calculate properties of the electronic structure of the clean surface
    with jasp('dos/surf-{atom}-expanded-clean'.format(**locals())) as calc:
        ados = VaspDos(efermi=calc.get_fermi_level()) # Get the density of states
        energies_clean = ados.energy # Get the list of energies and the density at each energy
```

Table 9: Metal adsorption DOS data

Atom	Ads site	d center	d width	d states	eg center	$t2g$ center	oxygen p center	Ads Energy
Ti	ontop	-0.276	1.036	36.599	-0.209	-0.326	-2.740	-3.735
Ti	fcc	-0.276	1.036	36.599	-0.209	-0.326	-4.622	-6.407
Ti	hcp	-0.276	1.036	36.599	-0.209	-0.326	-4.748	-6.197
V	ontop	-0.476	1.167	44.192	-0.445	-0.498	-2.893	-4.124
V	fcc	-0.476	1.167	44.192	-0.445	-0.498	-4.709	-6.736
V	hcp	-0.476	1.167	44.192	-0.445	-0.498	-4.635	-6.634
Cr	ontop	-0.586	1.346	53.429	-0.603	-0.575	-3.125	-4.134
Cr	fcc	-0.586	1.346	53.429	-0.603	-0.575	-4.719	-5.719
Cr	hcp	-0.586	1.346	53.429	-0.603	-0.575	-4.711	-5.903
Mn	ontop	-0.926	1.509	58.120	-1.078	-0.832	-3.096	-3.034
Mn	fcc	-0.926	1.509	58.120	-1.078	-0.832	-4.411	-4.437
Mn	hcp	-0.926	1.509	58.120	-1.078	-0.832	-4.570	-4.716
Fe	ontop	-1.077	1.648	69.888	-1.115	-1.052	-2.612	-3.397
Fe	fcc	-1.077	1.648	69.888	-1.115	-1.052	-4.137	-4.071
Fe	hcp	-1.077	1.648	69.888	-1.115	-1.052	-4.284	-4.508
Co	ontop	-1.090	1.544	75.076	-1.109	-1.077	-2.018	-1.851
Co	fcc	-1.090	1.544	75.076	-1.109	-1.077	-3.638	-4.153
Co	hcp	-1.090	1.544	75.076	-1.109	-1.077	-3.734	-3.682
Ni	ontop	-1.330	1.413	75.892	-1.335	-1.327	-1.891	-1.082
Ni	fcc	-1.330	1.413	75.892	-1.335	-1.327	-3.402	-3.007
Ni	hcp	-1.330	1.413	75.892	-1.335	-1.327	-3.294	-2.907
Cu	ontop	-2.280	1.190	76.247	-2.274	-2.284	-1.114	-0.429
Cu	fcc	-2.280	1.190	76.247	-2.274	-2.284	-2.879	-2.346
Cu	hcp	-2.280	1.190	76.247	-2.274	-2.284	-2.666	-2.196

```
        ind = (energies_clean < 5) & (energies_clean > -4) # The range of energies we're interested in
        energies_clean = energies_clean[ind] # The range of densities we're interested in
        eg = np.zeros(len(energies_clean)) # Create an array for the density
        t2g = np.zeros(len(energies_clean)) # Create an array for the density
        # Loop over each surface metal atom and each part of the d-band
        for ai in a_index:
            for t2gi in t2g_index:
                t2g += np.array((ados.site_dos(calc.resort[ai], t2gi)[ind]))
            for egi in eg_index:
                eg += np.array((ados.site_dos(calc.resort[ai], egi)[ind]))
        eg_states = np.trapz(eg, energies_clean) # Number of eg states
        t2g_states = np.trapz(t2g, energies_clean) # Number of t2g states
        eg_center = np.trapz(energies_clean * eg, energies_clean) / eg_states # eg center
        t2g_center = np.trapz(energies_clean * t2g, energies_clean) / t2g_states # t2g center
        d = t2g + eg
        d_states = np.trapz(d, energies_clean) # Number of d-states
        d_center = np.trapz(energies_clean * d, energies_clean) / d_states # d center
        d_centers = d_center * np.ones(len(energies_clean)) # We need this to calculate the width
        # Calculate the d-width
        d_width = np.sqrt(np.trapz((energies_clean - d_centers) ** 2 * d, energies_clean) / d_states)

    for site in sites:
        # We perform calculations on the adsorbate p-band center below
        with jasp('dos/surf-{atom}-expanded-{site}-low-kpts'.format(**locals())) as calc:
            ados = VaspDos(efermi=calc.get_fermi_level())
            energies = ados.energy
```

```
            ind = (energies < 5) & (energies > -10)
            energies = energies[ind]
            op = np.zeros(len(energies))
            for ai in o_index:
                for opi in op_index:
                    op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
        op_states = np.trapz(op, energies)
        op_center = np.trapz(energies * op, energies) / op_states
        energy = ads_energies[i]
        i += 1
        s = '|{0}|{1}|{2:1.3f}|{3:1.3f}|{4:1.3f}|{5:1.3f}|{6:1.3f}|{7:1.3f}|{8:1.3f}|'.format(**locals())
        print s.format(atom, site, d_center, d_width, d_states, eg_center,
                       t2g_center, op_cetner, energy)
```

Table 10: Data on the DOS

Atom	Ads site	d center	d width	d states	eg center	t2g center	op center	Ads Energy
Ti	ontop	0.021	0.944	41.079	0.050	0.003	-2.768	-3.898
Ti	fcc	0.021	0.944	41.079	0.050	0.003	-4.936	-6.354
Ti	hcp	0.021	0.944	41.079	0.050	0.003	-4.972	-6.275
V	ontop	-0.126	0.975	48.340	-0.063	-0.166	-3.469	-4.160
V	fcc	-0.126	0.975	48.340	-0.063	-0.166	-4.549	-4.945
V	hcp	-0.126	0.975	48.340	-0.063	-0.166	-5.991	-5.686
Mn	ontop	-1.006	1.789	63.899	-0.948	-1.047	-3.535	-1.594
Mn	fcc	-1.006	1.789	63.899	-0.948	-1.047	-5.227	0.803
Mn	hcp	-1.006	1.789	63.899	-0.948	-1.047	-5.114	-1.119
Co	ontop	-1.086	1.377	74.712	-1.088	-1.086	-1.994	-1.404
Co	fcc	-1.086	1.377	74.712	-1.088	-1.086	-4.346	-1.583
Co	hcp	-1.086	1.377	74.712	-1.088	-1.086	-3.775	-1.552
Ni	ontop	-0.816	0.973	75.711	-0.805	-0.824	-2.001	-1.505
Ni	fcc	-0.816	0.973	75.711	-0.805	-0.824	-3.687	-3.166
Ni	hcp	-0.816	0.973	75.711	-0.805	-0.824	-3.731	-3.509
Cu	ontop	-1.568	0.834	76.236	-1.575	-1.563	-1.060	-0.265
Cu	fcc	-1.568	0.834	76.236	-1.575	-1.563	-2.918	-2.206
Cu	hcp	-1.568	0.834	76.236	-1.575	-1.563	-2.959	-1.745

5.2.4. Analyzing the DOS of oxides

```
from jasp import *
from ase.calculators.vasp import Vasp, VaspDos
import matplotlib.pyplot as plt

atoms, sites, ads_energies = zip(*data)

# These are the indexes of the surface atoms needed to be analyzed.
# I found these by visualizing the atom using the ase.visualize.view function
atoms = ('Ti', 'V', 'Mn', 'Co', 'Ni', 'Cu')
a_index = (0, 11, 22, 33, 10, 21, 32, 43)
o_index = (44, 45)
t2g_index = (8, 9, 10, 11, 14, 15)
eg_index = (12, 13, 16, 17)
op_index = (2, 3, 4, 5, 6, 7)

sites = ('ontop', 'fcc', 'hcp')
```

```
17
18  print '|Atom|Ads site|d center|d width|dstates|eg center|t2g center|op center|Ads Energy|'
19  print '|--|'
20
21  i = 0
22
23  for atom in atoms:
24      fig = plt.figure(1, (5, 6))
25      # Calculate properties of the electronic structure of the clean surface
26      with jasp('dos/surf-{atom}0-clean'.format(**locals())) as calc:
27          ados = VaspDos(efermi=calc.get_fermi_level()) # Get the density of states
28          energies_clean = ados.energy # Get the list of energies and the density at each energy
29          ind = (energies_clean < 5) & (energies_clean > -4) # The range of energies we're interested in
30          energies_clean = energies_clean[ind] # The range of densities we're interested in
31          eg = np.zeros(len(energies_clean)) # Create an array for the density
32          t2g = np.zeros(len(energies_clean)) # Create an array for the density
33          # Loop over each surface metal atom and each part of the d-band
34          for ai in a_index:
35              for t2gi in t2g_index:
36                  t2g += np.array((ados.site_dos(calc.resort[ai], t2gi)[ind]))
37              for egi in eg_index:
38                  eg += np.array((ados.site_dos(calc.resort[ai], egi)[ind]))
39          eg_states = np.trapz(eg, energies_clean) # Number of eg states
40          t2g_states = np.trapz(t2g, energies_clean) # Number of t2g states
41          eg_center = np.trapz(energies_clean * eg, energies_clean) / eg_states # eg center
42          t2g_center = np.trapz(energies_clean * t2g, energies_clean) / t2g_states # t2g center
43          d = t2g + eg
44          d_states = np.trapz(d, energies_clean) # Number of d-states
45          d_center = np.trapz(energies_clean * d, energies_clean) / d_states # d center
46          d_centers = d_center * np.ones(len(energies_clean)) # We need this to calculate the width
47          # Calculate the d-width
48          d_width = np.sqrt(np.trapz((energies_clean-d_centers) ** 2 * d, energies_clean) / d_states)
49
50      for site in sites:
51          # We perform calculations on the adsorbate p-band center below
52          with jasp('dos/surf-{atom}0-{site}'.format(**locals())) as calc:
53              ados = VaspDos(efermi=calc.get_fermi_level())
54              energies = ados.energy
55              ind = (energies < 5) & (energies > -10)
56              energies = energies[ind]
57              op = np.zeros(len(energies))
58              for ai in o_index:
59                  for opi in op_index:
60                      op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
61          op_states = np.trapz(op, energies)
62          op_center = np.trapz(energies * op, energies) / op_states
63          energy = ads_energies[i]
64          i += 1
65          s = '|{0}|{1}|{2:1.3f}|{3:1.3f}|{4:1.3f}|{5:1.3f}|{6:1.3f}|{7:1.3f}|{8:1.3f}|'.format(**locals())
66          print s.format(atom, site, d_center, d_width, d_states,
67                         eg_center, t2g_center, op_center, energy)
```

5.3. Graphing relationships between d-band, p-bands, and adsorption energies

This is the code for making the graph found in the paper (Figure 3)

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  atoms, sites, dc, dw, ds, eg_centers, t2g_centers, op_centers, energies  = zip(*oxide_data)
5
6  fig = plt.figure(1, (6.5, 4.5))
7  ax2 = fig.add_subplot(122)
8
9  atoms, sites, d_centers, dw, ds, eg_centers, t2g_centers, op_centers, energies  = zip(*metal_data)
```

Table 11: Oxide adsorption DOS data

Atom	Ads site	d center	d width	dstates	eg center	t2g center	op center	Ads Energy
Ti	ontop	-0.165	0.837	52.802	-0.074	-0.205	-2.766	-3.874
Ti	fcc	-0.165	0.837	52.802	-0.074	-0.205	-4.416	-5.858
Ti	hcp	-0.165	0.837	52.802	-0.074	-0.205	-4.781	-6.187
V	ontop	-0.267	1.133	76.053	0.240	-0.590	-3.406	-3.958
V	fcc	-0.267	1.133	76.053	0.240	-0.590	-3.792	-4.182
V	hcp	-0.267	1.133	76.053	0.240	-0.590	-3.923	-5.339
Mn	ontop	-1.128	1.763	107.757	-1.191	-1.095	-2.450	-2.400
Mn	fcc	-1.128	1.763	107.757	-1.191	-1.095	-3.491	-4.253
Mn	hcp	-1.128	1.763	107.757	-1.191	-1.095	-4.101	-5.312
Co	ontop	-1.148	1.312	137.636	-0.832	-1.338	-2.159	-2.106
Co	fcc	-1.148	1.312	137.636	-0.832	-1.338	-3.515	-3.732
Co	hcp	-1.148	1.312	137.636	-0.832	-1.338	-3.642	-3.864
Ni	ontop	-0.702	0.981	142.994	-0.422	-0.878	-2.094	-1.565
Ni	fcc	-0.702	0.981	142.994	-0.422	-0.878	-3.193	-2.916
Ni	hcp	-0.702	0.981	142.994	-0.422	-0.878	-3.082	-3.843
Cu	ontop	-1.335	0.829	142.004	-1.082	-1.492	-0.996	-0.612
Cu	fcc	-1.335	0.829	142.004	-1.082	-1.492	-2.343	-2.556
Cu	hcp	-1.335	0.829	142.004	-1.082	-1.492	-2.750	-3.779

```
ax2.plot(op_centers, energies, label='Metals', marker='o', ls='none', c='y')
metal_fit = np.poly1d(np.polyfit(op_centers, energies, 1))
fit_energies = np.linspace(-6, 0)
ax2.plot(fit_energies, metal_fit(fit_energies), c='k')

atoms, sites, d_centers, dw, ds, eg_centers, t2g_centers, op_centers, energies  = zip(*expanded_data)
ax2.plot(op_centers, energies, label='Expanded metals', marker='o', ls='none', c='g')

atoms, sites, dc, dw, ds, eg_centers, t2g_centers, op_centers, energies  = zip(*oxide_data)
ax2.plot(op_centers, energies, label='Oxides', marker='o', ls='none', c='r')

ax2.set_xlabel(r'Oxygen $p$-band center (eV)')
ax2.set_ylabel('Adsorption energy (eV/O)')
ax2.text(0.85, 0.9, 'b)', fontsize=20, transform=ax2.transAxes)
ax2.annotate('MnO-top', xy=(-3.535, -1.594),
             xytext=(-3.535, -1.594+0.5),
             arrowprops=dict(arrowstyle='->'))
ax2.annotate('MnO-fcc', xy=(-5.227, 0.803),
             xytext=(-5.227+0.4, 0.803+0.4),
             arrowprops=dict(arrowstyle='->'))
ax2.annotate('MnO-hcp', xy=(-5.114, -1.119),
             xytext=(-5.114, -1.119+1),
             arrowprops=dict(arrowstyle='->'))
ax2.annotate('CoO-fcc', xy=(-4.346, -1.583),
             xytext=(-4.346, -1.583+1),
             arrowprops=dict(arrowstyle='->'))
ax2.annotate('CoO-hcp', xy=(-3.775, -1.552),
             xytext=(-3.775-1, -1.552-1),
             arrowprops=dict(arrowstyle='->'))
ax2.set_xlim((-5.5, -0.5))
ax2.set_xticks(range(-5, 0, 1))
ax2.legend(loc=4, prop={'size': 'small'}, numpoints=1)

atoms, sites, d_centers, dw, ds, eg_centers, t2g_centers, op_centers, energies  = zip(*metal_data)
op_ontop, op_fcc, op_hcp = [], [], []
```

```
d_mc = []

for atom, site, dc, dw, ds, eg, t2g, op, energy in metal_data:
    if atom in atoms:
        if site == 'ontop':
            d_mc.append(dc)
            op_ontop.append(op)
        elif site == 'fcc':
            op_fcc.append(op)
        elif site == 'hcp':
            op_hcp.append(op)
    else:
        pass

top_fit = np.poly1d(np.polyfit(d_mc, op_ontop, 1))
fcc_fit = np.poly1d(np.polyfit(d_mc, op_fcc, 1))
hcp_fit = np.poly1d(np.polyfit(d_mc, op_hcp, 1))
fit_energies = np.linspace(-2.5, 0)

ax1 = fig.add_subplot(121)
ax1.plot(d_mc, op_ontop, label='top sites', c='b', ls='none', marker='s')
ax1.plot(d_mc, op_fcc, label='fcc sites', c='c', ls='none', marker='s')
ax1.plot(d_mc, op_hcp, label='hcp sites', c='m', ls='none', marker='s')
ax1.plot(fit_energies, top_fit(fit_energies), c='b')
ax1.plot(fit_energies, fcc_fit(fit_energies), c='c')
ax1.plot(fit_energies, hcp_fit(fit_energies), c='m')
ax1.set_xlim(-2.5, 0)
ax1.set_ylim(-5.0, -1.0)
ax1.text(0.85, 0.9, 'a)', fontsize=20, transform=ax1.transAxes)
ax1.set_xlabel(r'Metal $d$-band center (eV)')
ax1.set_ylabel(r'Oxygen $p$-band center (eV)')
ax1.legend(loc=3, prop={'size':'small'}, numpoints=1)

fig.tight_layout()
fig.savefig('figures/FIG3.pdf')
plt.show()
```

5.4. Graphing density of states

The bottom two sections are code for making Figures 4 and 5 in the paper. These codes read data directly from the VASP calculations which are not included here. We cannot share the POTCAR files because of the VASP license, and without them these codes do not run.

5.4.1. Graphing the DOS associated with expansion

```
from jasp import *
from ase.calculators.vasp import Vasp, VaspDos
import matplotlib.pyplot as plt

a_index = (0, 1, 2, 3, 20, 21, 22, 23)
d_index = (8, 9, 10, 11, 12, 13, 14, 15, 16, 17)
op_index = (2, 3, 4, 5, 6, 7)
os_index = (0, 1)

fig = plt.figure(1, (12, 4))

ax11 = fig.add_axes((0.04, 0.52, 0.31, 0.41))
with jasp('dos/surf-Ti-fcc') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
```

```
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax11.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax11.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax11.axvline(0, ls='--', c='k', label='Fermi level')
    ax11.legend(loc=9, prop={'size':'small'})
    ax11.set_title('Ti')
    ax11.set_xlim(-7, 3)
    ax11.set_xticks(np.linspace(-6, 2, 9))
    ax11.set_xticklabels([], [])
    ax11.set_ylim(0, 5)
    ax11.text(0.9, 0.8, 'a)', transform=ax11.transAxes, fontsize=20)
    ax11.set_yticks((0, 1, 2, 3, 4))
    ax11.set_ylabel('States per atom')
    ax11.set_xticklabels([], [])

ax21 = fig.add_axes((0.04, 0.11, 0.31, 0.41))
with jasp('dos/surf-Ti-expanded-fcc-relax-low-kpts') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax21.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax21.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax21.axvline(0, ls='--', c='k')
    # ax21.legend(loc=0, prop={'size':'small'})
    ax21.set_xlim(-7, 3)
    ax21.set_ylim(0, 5)
    ax21.text(0.9, 0.8, 'b)', transform=ax21.transAxes, fontsize=20)
    ax21.set_xticks(np.linspace(-6, 2, 9))
    ax21.set_yticks((0, 1, 2, 3, 4))
    ax21.set_ylabel('States per atom')
    ax21.set_xlabel('Energy (eV)')

ax12 = fig.add_axes((0.35, 0.52, 0.31, 0.41))
with jasp('dos/surf-Mn-fcc') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
```

```
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax12.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax12.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax12.axvline(0, ls='--', c='k', label='Fermi level')
    # ax12.legend(loc=9, prop={'size':'small'})
    ax12.set_xlim(-7, 3)
    ax12.set_xticks(np.linspace(-6, 2, 9))
    ax12.set_ylim(0, 5)
    ax12.set_title('Mn')
    ax12.text(0.9, 0.8, 'c)', transform=ax12.transAxes, fontsize=20)
    ax12.set_xticklabels([], [])
    ax12.set_yticklabels([], [])

ax22 = fig.add_axes((0.35, 0.11, 0.31, 0.41))
with jasp('dos/surf-Mn-expanded-fcc-relax-low-kpts') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax22.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax22.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax22.axvline(0, ls='--', c='k')
    # ax22.legend(loc=0, prop={'size':'small'})
    ax22.set_xlim(-7, 3)
    ax22.set_xticks(np.linspace(-6, 2, 9))
    ax22.set_ylim(0, 5)
    ax22.text(0.9, 0.8, 'd)', transform=ax22.transAxes, fontsize=20)
    ax22.set_yticklabels([], [])
    ax22.set_xlabel('Energy (eV)')

ax13 = fig.add_axes((0.66, 0.52, 0.31, 0.41))
with jasp('dos/surf-Cu-fcc') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax13.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax13.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax13.axvline(0, ls='--', c='k')
    # ax13.legend(loc=0, prop={'size':'small'})
    ax13.set_title('Cu')
    ax13.set_xlim(-7, 3)
    ax13.set_xticks(np.linspace(-6, 2, 9))
    ax13.set_ylim(0, 5)
    ax13.text(0.9, 0.8, 'e)', transform=ax13.transAxes, fontsize=20)
```

```
    ax13.set_xticklabels([], [])
    ax13.set_yticklabels([], [])
    ax13x = ax13.twinx()
    ax13x.set_xlim(ax13.get_xlim())
    ax13x.set_ylim(ax13.get_ylim())
    ax13x.set_xticks(np.linspace(-6, 2, 9))
    ax13x.set_xticklabels([], [])
    ax13x.set_yticklabels([], [])
    ax13x.set_ylabel('Metal', size='large')

ax23 = fig.add_axes((0.66, 0.11, 0.31, 0.41))
with jasp('dos/surf-Cu-expanded-fcc-relax-low-kpts') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax23.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax23.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax23.axvline(0, ls='--', c='k')
    # ax23.legend(loc=0, prop={'size':'small'})
    ax23.set_xlim(-7, 3)
    ax23.set_xticks(np.linspace(-6, 2, 9))
    ax23.set_ylim(0, 5)
    ax23.text(0.9, 0.8, 'f)', transform=ax23.transAxes, fontsize=20)
    ax23.set_yticklabels([], [])
    ax23.set_xlabel('Energy (eV)')
    ax23x = ax23.twinx()
    ax23x.set_xticks(np.linspace(-6, 2, 9))
    ax23x.set_ylim(ax23.get_ylim())
    ax23x.set_yticklabels([], [])
    ax23x.set_ylabel('Expanded metal', size='large')

fig.savefig('figures/FIG4.pdf')
plt.show()
```

5.4.2. Graphing the DOS associated with oxidation

```
from jasp import *
from ase.calculators.vasp import Vasp, VaspDos
import matplotlib.pyplot as plt

a_index = (0, 1, 2, 3, 20, 21, 22, 23)
d_index = (8, 9, 10, 11, 12, 13, 14, 15, 16, 17)
op_index = (2, 3, 4, 5, 6, 7)
os_index = (0, 1)

fig = plt.figure(1, (12, 4))

ax11 = fig.add_axes((0.04, 0.52, 0.31, 0.41))
with jasp('dos/surf-Ti-expanded-fcc-relax-low-kpts') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
```

```
            for di in d_index:
                d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
        d /= 8
        op = np.zeros(len(energies))
        for ai in (range(len(calc.get_atoms()))[-1],
                   range(len(calc.get_atoms()))[-2]):
            for opi in op_index:
                op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
        op /= 2
        ax11.plot(energies, d, label=r'Metal $d$-bands', c='b')
        ax11.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
        ax11.axvline(0, ls='--', c='k', label='Fermi level')
        ax11.legend(loc=9, prop={'size':'small'})
        ax11.set_title('Ti')
        ax11.set_xlim(-10, 3)
        ax11.set_xticks(np.linspace(-9, 2, 12))
        ax11.set_ylim(0, 5)
        ax11.text(0.9, 0.8, 'a)', transform=ax11.transAxes, fontsize=20)
        ax11.set_yticks((0, 1, 2, 3, 4))
        ax11.set_ylabel('States per atom')
        ax11.set_xticklabels([], [])

    ax21 = fig.add_axes((0.04, 0.11, 0.31, 0.41))
    with jasp('dos/surf-TiO-fcc') as calc:
        ados = VaspDos(efermi=calc.get_fermi_level())
        energies = ados.energy
        ind = (energies < 5) & (energies > -10)
        energies = energies[ind]
        d = np.zeros(len(energies))
        for ai in a_index:
            for di in d_index:
                d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
        d /= 8
        op = np.zeros(len(energies))
        for ai in (range(len(calc.get_atoms()))[-1],
                   range(len(calc.get_atoms()))[-2]):
            for opi in op_index:
                op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
        op /= 2
        ax21.plot(energies, d, label=r'Metal $d$-bands', c='b')
        ax21.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
        ax21.axvline(0, ls='--', c='k', label='Fermi level')
        # ax21.legend(loc=0, prop={'size':'small'})
        ax21.set_xlim(-10, 3)
        ax21.set_ylim(0, 5)
        ax21.set_xticks(np.linspace(-9, 2, 12))
        ax21.text(0.9, 0.8, 'b)', transform=ax21.transAxes, fontsize=20)
        ax21.set_yticks((0, 1, 2, 3, 4))
        ax21.set_ylabel('States per atom')
        ax21.set_xlabel('Energy (eV)')

    ax12 = fig.add_axes((0.35, 0.52, 0.31, 0.41))
    with jasp('dos/surf-Mn-expanded-fcc-relax-low-kpts') as calc:
        ados = VaspDos(efermi=calc.get_fermi_level())
        energies = ados.energy
        ind = (energies < 5) & (energies > -10)
        energies = energies[ind]
        d = np.zeros(len(energies))
        for ai in a_index:
            for di in d_index:
                d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
        d /= 8
        op = np.zeros(len(energies))
        for ai in (range(len(calc.get_atoms()))[-1],
                   range(len(calc.get_atoms()))[-2]):
            for opi in op_index:
                op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
        op /= 2
```

```
    ax12.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax12.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax12.axvline(0, ls='--', c='k', label='Fermi level')
    # ax12.legend(loc=9, prop={'size':'small'})
    ax12.set_xlim(-10, 3)
    ax12.set_xticks(np.linspace(-9, 2, 12))
    ax12.set_ylim(0, 5)
    ax12.set_title('Mn')
    ax12.text(0.9, 0.8, 'c)', transform=ax12.transAxes, fontsize=20)
    ax12.set_xticklabels([], [])
    ax12.set_yticklabels([], [])

ax22 = fig.add_axes((0.35, 0.11, 0.31, 0.41))
with jasp('dos/surf-MnO-fcc') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax22.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax22.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax22.axvline(0, ls='--', c='k', label='Fermi level')
    #ax22.legend(loc=0, prop={'size':'small'})
    ax22.set_xlim(-10, 3)
    ax22.set_xticks(np.linspace(-9, 2, 12))
    ax22.set_ylim(0, 5)
    ax22.text(0.9, 0.8, 'd)', transform=ax22.transAxes, fontsize=20)
    ax22.set_yticklabels([], [])
    ax22.set_xlabel('Energy (eV)')

ax13 = fig.add_axes((0.66, 0.52, 0.31, 0.41))
with jasp('dos/surf-Cu-expanded-fcc-relax-low-kpts') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax13.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax13.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax13.axvline(0, ls='--', c='k', label='Fermi level')
    #ax13.legend(loc=0, prop={'size':'small'})
    ax13.set_title('Cu')
    ax13.set_xlim(-10, 3)
    ax13.set_xticks(np.linspace(-9, 2, 12))
    ax13.set_ylim(0, 5)
    ax13.text(0.9, 0.8, 'e)', transform=ax13.transAxes, fontsize=20)
    ax13.set_xticklabels([], [])
    ax13.set_yticklabels([], [])
```

```
    ax13x = ax13.twinx()
    ax13x.set_xlim(ax13.get_xlim())
    ax13x.set_xticks(np.linspace(-9, 2, 12))
    ax13x.set_ylim(ax13.get_ylim())
    ax13x.set_xticklabels([], [])
    ax13x.set_yticklabels([], [])
    ax13x.set_ylabel('Expanded Metal', size='large')

ax23 = fig.add_axes((0.66, 0.11, 0.31, 0.41))
with jasp('dos/surf-CuO-fcc') as calc:
    ados = VaspDos(efermi=calc.get_fermi_level())
    energies = ados.energy
    ind = (energies < 5) & (energies > -10)
    energies = energies[ind]
    d = np.zeros(len(energies))
    for ai in a_index:
        for di in d_index:
            d += np.array((ados.site_dos(calc.resort[ai], di)[ind]))
    d /= 8
    op = np.zeros(len(energies))
    for ai in (range(len(calc.get_atoms()))[-1],
               range(len(calc.get_atoms()))[-2]):
        for opi in op_index:
            op += np.array((ados.site_dos(calc.resort[ai], opi)[ind]))
    op /= 2
    ax23.plot(energies, d, label=r'Metal $d$-bands', c='b')
    ax23.plot(energies, op, label=r'Oxygen $p$-bands', c='r')
    ax23.axvline(0, ls='--', c='k', label='Fermi level')
    #ax23.legend(loc=0, prop={'size':'small'})
    ax23.set_xlim(-10, 3)
    ax23.set_xticks(np.linspace(-9, 2, 12))
    ax23.set_ylim(0, 5)
    ax23.text(0.9, 0.8, 'f)', transform=ax23.transAxes, fontsize=20)
    ax23.set_yticklabels([], [])
    ax23.set_xlabel('Energy (eV)')
    ax23x = ax23.twinx()
    ax23x.set_ylim(ax23.get_ylim())
    ax23x.set_xticks(np.linspace(-9, 2, 12))
    ax23x.set_yticklabels([], [])
    ax23x.set_ylabel('Oxide', size='large')

fig.savefig('figures/FIG5.pdf')
plt.show()
```

6. References

References

[1] C. Kittel, Introduction to Solid State Physics, 6th Edition, John Wiley & Sons, Inc., New York, 1986.

[2] A. Pearson, Studies on the lower oxides of titanium, Journal of Physics and Chemistry of Solids 5 (4) (1958) 316 – 327. doi:http://dx.doi.org/10.1016/0022-3697(58)90035-0.
URL http://www.sciencedirect.com/science/article/pii/0022369758900350

[3] R. E. Loehman, C. N. R. Rao, J. M. Honig, Crystallography and defect chemistry of solid solutions of vanadium and titanium oxides, The Journal of Physical Chemistry 73 (6) (1969) 1781–1784. arXiv:http://pubs.acs.org/doi/pdf/10.1021/j100726a025,

doi:10.1021/j100726a025.
URL http://pubs.acs.org/doi/abs/10.1021/j100726a025

[4] R. Wyckoff, Crystal Structures, Crystal Structures, Interscience Publ., 1964.
URL http://books.google.com/books?id=DD5RAAAAMAAJ

[5] M. Carey, F. Spada, A. Berkowitz, W. Cao, G. Thomas, Preparation and structural characterization of sputtered CoO, NiO, and $Ni_{0.5}Co_{0.5}O$ thin epitaxial films, Journal of Materials Research 6 (1991) 2680–2687. doi:10.1557/JMR.1991.2680.
URL http://journals.cambridge.org/article$_S$0884291400014618

[6] L. C. Bartel, B. Morosin, Exchange striction in NiO, Phys. Rev. B 3 (1971) 1039–1043. doi:10.1103/PhysRevB.3.1039.
URL http://link.aps.org/doi/10.1103/PhysRevB.3.1039

[7] A. Rohrbach, J. Hafner, G. Kresse, Ab initio study of the (0001) surfaces of hematite and chromia: Influence of strong electronic correlations, Phys. Rev. B 70 (2004) 125426. doi:10.1103/PhysRevB.70.125426.
URL http://link.aps.org/doi/10.1103/PhysRevB.70.125426

[8] W. Bergermayer, H. Schweiger, E. Wimmer, *Ab initio* thermodynamics of oxide surfaces: O_2 on $Fe_2O_3(0001)$, Phys. Rev. B 69 (2004) 195409. doi:10.1103/PhysRevB.69.195409.
URL http://link.aps.org/doi/10.1103/PhysRevB.69.195409