Supporting information for: A linear response, DFT+U study of trends in the oxygen evolution activity of transition metal rutile dioxides

Zhongnan Xu,[†] Jan Rossmeisl,[‡] and John R. Kitchin[*,†]

Department of Chemical Engineering, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, and Center for Atomic-Scale Materials Design, Department of Physics, Technical University of Denmark, Building 307, 2800 Kgs. Lyngby, Denmark

E-mail: jkitchin@andrew.cmu.edu

[*]To whom correspondence should be addressed

[†]Department of Chemical Engineering, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213

[‡]Center for Atomic-Scale Materials Design, Department of Physics, Technical University of Denmark, Building 307, 2800 Kgs. Lyngby, Denmark

Contents

1 Introduction

This supporting information contains all of the code necessary to reproduce the calculation and analysis of our data. A majority of the code for the construction of figures reads data directly from finished calculations. This data can be found at `http://dx.doi.org/10.5281/zenodo.12635` (about 1.8 GB of organized computational data). A zip file can be downloaded from `http://zenodo.org/record/12635/files/rutile-OER-v1.0.zip`. You must unzip the file, and put the supporting-information.org file in the unzipped directory. The zip file is about 572 MB in size, and unpacks to about 1.8 GB. The scripts in this document read data from that repository. All necessary modules for this can be found in the Section 8.

The calculation and analysis of bulk properties can be found in Section 2. This includes the calculation of the equilibrium lattice constants, ground state magnetic configuration, and linear response U. Section 3 details the calculation of adsorption energies at $U = 0$, while Section 4 contains code for the analysis of adsorption energies $U = 0$ and construction of Figure 1 of the manuscript. Similarly, Section 5 and 6 contain the calculation and analysis scripts for adsorption energies at $U > 0$. Finally, 7 takes the adsorption energies and compares the activities of IrO_2, RuO_2, PtO_2, and RhO_2 for the oxygen evolution reaction.

The source for this document can be found here: . The source document is an 'org' file, which is a plain text file in org-mode syntax.[?]

2 Calculation of bulk properties

We need to calculate the bulk properties to accurately construct our surfaces. The major bulk properties we need are the atomic and magnetic structure along with the calculated linear response U value. The calculation of these properties is outlined below.

2.1 Lattice constants and magnetic ordering

Before constructing surface slabs needed for calculating adsorption energies, their equilibrium lattice coordinates and atomic positions must first be calculated. The code below calculates these for the systems we studied. We first perform an equation of state (EOS) around a large volume range to obtain a good guess of the equilibrium volume. We then take this guess and perform another fine EOS around this volume. Finally, we perform a full structure relaxation at the equilibrium volume predicted by the fine EOS, in which we extract the cell parameters a and c and the oxygen parameter u.

In addition to extracting the lattice constants, we also calculate the ground state magnetic ordering. We do this after we obtain a good guess of the equilibrium volume. The ground state magnetic ordering is obtained by performing the equation of state around the guess with nonmagnetic and ferromagnetic orderings.

2.1.1 Equilibrium volume guess from coarse EOS

The code below calculates an initial guess for the equilibrium volume of each of the structures we are interested in. We center the EOS around a volume of 62 Å per primitive cell, which is close to the known experimental volumes of a majority of rutile dioxides.

```
from espresso import *
from ase_addons import bulk
from ase.utils.eos import EquationOfState
from ase.visualize import view
import numpy as np

```

```
vol = 62
factors = (0.8, 0.9, 1.0, 1.1, 1.2)

elements = ['Mo', 'Ir', 'Ru', 'Pt', 'Ti',
            'Nb', 'Re', 'Rh',  'Mn', 'Cr']

print '#+CAPTION: Equilibrium volumes calculated from 3rd order polynomial fit to a coarse EOS'
print '#+ATTR_LATEX: :placement [H] c|c'
print '#+TBLNAME: coarse-EOS'
print '|Oxide|V ($\\AA$/primitive cell)|'
print '|----|'

for name in elements:
    ready = True
    volumes, energies = [], []
    for fac in factors:
        atoms = bulk.rutile((name, 'O'), mags=(0.6, 0))
        atoms.set_volume(fac * vol)
        calcdir = 'supporting-data/{name}O2/coarse-EOS/{name}O2-v-{fac:1.1f}'.format(**locals())
        with Espresso(calcdir, atoms=atoms,
                      disk_io='none', calculation='vc-relax',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      nspin=2, cell_dofree='shape',
                      kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
            try:
                energy = calc.get_potential_energy()
                energies.append(energy)
                volumes.append(atoms.get_volume())
            except (EspressoSubmitted, EspressoRunning):
                ready = False
                print calc.espressodir, 'running'
            except (EspressoNotConverged):
                ready = False
                calc.set(mixing_beta=0.3)

    eos = EquationOfState(volumes, energies)
    v0, e0, B = eos.fit()
    eos.plot('supporting-figures/{name}O2-coarse-EOS.png'.format(**locals()), show=False)
    print '|{name}O2|{v0:1.3f}|'.format(**locals())
```

Table S1: Equilibrium volumes calculated from 3rd order polynomial fit to a coarse EOS

Oxide	V (Å/primitive cell)
MoO2	66.048
IrO2	65.657
RuO2	64.122
PtO2	68.059
TiO2	64.265
RhO2	64.405
NbO2	72.301
ReO2	64.617
MnO2	56.996
CrO2	57.817

2.1.2 Equilibrium volume from fine EOS along with ground state magnetic ordering

The code below calculates an equation of state near the guessed equilibrium volume in both magnetic and non-magnetic states. We do this to obtain the ground state magnetic ordering along with the ground state volume.

```
from espresso import *
from ase.utils.eos import EquationOfState
from ase.visualize import view
import matplotlib.pyplot as plt
import numpy as np
from ase_addons import bulk

data = [['MoO2', 66.048],
        ['IrO2', 65.657],
        ['RuO2', 64.122],
        ['PtO2', 68.059],
        ['TiO2', 64.265],
        ['RhO2', 64.405],
        ['NbO2', 72.301],
        ['ReO2', 64.617],
        ['MnO2', 56.996],
        ['CrO2', 57.817]]

```

```
factors = (0.9, 0.95, 1.0, 1.05, 1.10)

print '#+CAPTION: Equilibrium volumes calculated from 3rd order polynomial fit to a Fine EOS'
print '#+ATTR_LATEX: :placement [H] c|c'
print '#+TBLNAME: fine-EOS'
print '|Oxide|V ($\\AA$/primitive cell)|'
print '|----|'

for name, vol in data:
    plt.figure(1, (4.5, 3))
    volumes, energies = [], []
    for fac in factors:
        atoms = bulk.rutile((name[:-2], 'O'), mags=(0.6, 0))
        atoms.set_volume(fac * vol)
        calcdir = 'supporting-data/{name}/fine-EOS/ferro/{name}-v-{fac:1.2f}'.format(**locals())
        with Espresso(calcdir, atoms=atoms,
                      disk_io='none', calculation='vc-relax',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      nspin=2, cell_dofree='shape', mixing_beta=0.3,
                      kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
            try:
                energy = calc.get_potential_energy()
                energies.append(energy)
                volumes.append(atoms.get_volume())
            except (EspressoSubmitted, EspressoRunning):
                pass
            except (EspressoNotConverged):
                pass

    min_E = min(energies)
    energies = np.array(energies) - min_E

    fit = np.poly1d(np.polyfit(volumes, energies, 3))
    fit_vols = np.linspace(min(volumes), max(volumes))
    plt.plot(volumes, energies, marker='o', ls='none', label='Ferromagnetic', c='r')
    plt.plot(fit_vols, fit(fit_vols), c='r')

    volumes, energies = [], []
    for fac in factors:
        atoms = bulk.rutile((name[:-2], 'O'), mags=(0, 0))
```

```
        atoms.set_volume(fac * vol)
        calcdir = 'supporting-data/{name}/fine-EOS/non-mag/{name}-v-{fac:1.2f}'.format(**locals())
        with Espresso(calcdir,  atoms=atoms,
                      disk_io='none', calculation='vc-relax',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      nspin=2, cell_dofree='shape', mixing_beta=0.3,
                      kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
            try:
                energy = calc.get_potential_energy()
                energies.append(energy)
                volumes.append(atoms.get_volume())
            except (EspressoSubmitted, EspressoRunning):
                pass
            except (EspressoNotConverged):
                pass

    energies = np.array(energies) - min_E

    fit = np.poly1d(np.polyfit(volumes, energies, 3))
    fit_vols = np.linspace(min(volumes), max(volumes))
    plt.plot(volumes, energies, marker='o', ls='none', label='Non-magnetic', c='k')
    plt.plot(fit_vols, fit(fit_vols), c='k')

    plt.xlabel('Volume')
    plt.ylabel('Relative Energy (eV)')
    plt.title('{name}'.format(**locals()))
    plt.legend(loc=9, prop={'size':'small'}, numpoints=1)
    plt.tight_layout()
    plt.savefig('supporting-figures/{name}-fine-EOS.png'.format(**locals()))
    plt.show()

    eos = EquationOfState(volumes, energies)
    v0, e0, B = eos.fit()
    print '|{name}|{v0:1.3f}|'.format(**locals())

for name, vol in data:
    print '\n#+CAPTION: 3rd order polynomial equation of state for bulk {name}'.format(**locals())
    print '#+ATTR_LATEX: :placement [H]'
    print '[[./supporting-figures/{name}-fine-EOS.png]]'.format(**locals())
```

Table S2: Equilibrium volumes calculated from 3rd order polynomial fit to a Fine EOS

Oxide	V (Å/primitive cell)
MoO_2	66.764
IrO_2	65.638
RuO_2	64.080
PtO_2	68.074
TiO_2	64.167
RhO_2	64.362
NbO_2	72.268
ReO_2	65.688
MnO_2	54.081
CrO_2	55.677

./supporting-figures/MoO2-fine-EOS.png

Figure S1: 3rd order polynomial equation of state for bulk MoO_2

For MoO_2, we had difficulty converging ferromagnetic magnetic states near the ground structure. However, the converged calculations clearly show that the non-magnetic configuration is as or more stable than the ferromagnetic. Hence, we chose to perform calculations non-magnetic.

./supporting-figures/IrO2-fine-EOS.png

Figure S2: 3rd order polynomial equation of state for bulk IrO_2

./supporting-figures/RuO2-fine-EOS.png

Figure S3: 3rd order polynomial equation of state for bulk RuO_2

./supporting-figures/PtO2-fine-EOS.png

Figure S4: 3rd order polynomial equation of state for bulk PtO_2

./supporting-figures/TiO2-fine-EOS.png

Figure S5: 3rd order polynomial equation of state for bulk TiO_2

./supporting-figures/RhO2-fine-EOS.png

Figure S6: 3rd order polynomial equation of state for bulk RhO_2

./supporting-figures/NbO2-fine-EOS.png

Figure S7: 3rd order polynomial equation of state for bulk NbO_2

./supporting-figures/ReO2-fine-EOS.png

Figure S8: 3rd order polynomial equation of state for bulk ReO_2

./supporting-figures/MnO2-fine-EOS.png

Figure S9: 3rd order polynomial equation of state for bulk MnO_2

./supporting-figures/CrO2-fine-EOS.png

Figure S10: 3rd order polynomial equation of state for bulk CrO_2

From these results, we see that only the $3d$ oxides of CrO_2, and MnO_2 require magnetism. The other materials can be non-magnetic.

2.1.3 Final relaxation at equilibrium volume and magnetic ordering

The final piece of code takes information on the equilibrium volume and magnetic ordering and fully relaxes the structure with those settings. It also prints out a table of the cell and atomic parameters needed for the construction of surfaces.

```
from espresso import *
from ase.utils.eos import EquationOfState
from ase.visualize import view
import matplotlib.pyplot as plt
import numpy as np
from ase_addons import bulk

data = [['MoO2', 66.764],
        ['IrO2', 65.638],
        ['RuO2', 64.080],
        ['PtO2', 68.074],
        ['TiO2', 64.167],
        ['RhO2', 64.362],
```

```
        ['NbO2', 72.268],
        ['ReO2', 65.688],
        ['MnO2', 54.081],
        ['CrO2', 55.677]]

mag_elements = ('MnO2', 'CrO2')

print '#+CAPTION: Relaxed lattice coordinates of all rutile structures'
print '#+TBLNAME: rutile-struct'
print '|System|a|c|u|'
print '|----|'

for name, vol in data:
    if name in mag_elements:
        atoms = bulk.rutile((name[:-2], 'O'), mags=(0.6, 0))
    else:
        atoms = bulk.rutile((name[:-2], 'O'), mags=(0.0, 0))

    atoms.set_volume(vol)
    with Espresso('supporting-data/{name}/ground'.format(**locals()),
                  atoms=atoms,
                  disk_io='none', calculation='vc-relax',
                  ecutwfc=40.0, ecutrho=500.0,
                  occupations='smearing', smearing='mp', degauss=0.01,
                  nspin=2, cell_dofree='shape', mixing_beta=0.3,
                  kpts=(5, 5, 5), walltime='18:00:00', ppn=1) as calc:
        try:
            calc.calculate()
            pos = calc.atoms.get_positions()
            cell = calc.atoms.get_cell()
            a = cell[0][0]
            c = cell[2][2]
            u = pos[2][0] / a
            print '|{name}|{a:1.2f}|{c:1.2f}|{u:1.2f}|'.format(**locals())
        except (EspressoSubmitted, EspressoRunning):
            print calc.espressodir, 'running'
            pass
        except (EspressoNotConverged):
            # calc.write_input()
            # calc.run(series=True)
            print calc.espressodir, 'Not Converged'
```

Table S3: Relaxed lattice coordinates of all rutile structures.

System	a (Å)	c (Å)	u
MoO2	4.95	2.73	0.28
IrO2	4.54	3.18	0.31
RuO2	4.53	3.12	0.31
PtO2	4.59	3.23	0.31
TiO2	4.65	2.97	0.31
RhO2	4.55	3.11	0.31
NbO2	4.94	2.96	0.29
ReO2	4.95	2.68	0.28
MnO2	4.36	2.84	0.30
CrO2	4.38	2.90	0.30

2.2 Linear response U values

The linear response U is calculated from a $2 \times 2 \times 2$ super cell of the rutile crystal structure. The theory behind calculating the linear response U can be found in the seminal paper by Cococcioni and Gironcoli (2005).[?] Details of the method and specific executable used can be found at `http://media.quantum-espresso.org/santa_barbara_2009_07/index.php`. The code for calculating the linear response U values of the oxides is shown below, which uses the `espresso` module. The table of linear response U values is shown below after completion of the calculation.

```
import numpy as np
from espresso import *
from ase_addons import bulk

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]
```

```python

mag_elements = ('MnO2', 'CrO2')

indexes = {0:(0, 1, 6, 7, 12, 13, 18, 19, 24, 25, 30, 31, 36, 37, 42, 43),
           2:(2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17, 20, 21, 22, 23,
              26, 27, 28, 29, 32, 33, 34, 35, 38, 39, 40, 41, 44, 45, 46, 47)}

for name, a, c, u in data:
    if name in mag_elements:
        atoms = bulk.rutile((name[:-2], 'O'), a, c, u, mags=(0.6, 0))
    else:
        atoms = bulk.rutile((name[:-2], 'O'), a, c, u, mags=(0, 0))
    atoms.set_constraint()
    atoms *= (2, 2, 2)
    hubbard_Us = 1e-20 * np.ones(len(atoms))

    with Espresso('supporting-data/linear-response/{name}'.format(**locals()), atoms=atoms,
                  ecutwfc=40.0, ecutrho=500.0,
                  occupations='smearing', smearing='mp', degauss=0.01,
                  kpts=(4, 4, 4), nspin=2,
                  lda_plus_u=True, U_projection_type='atomic', Hubbard_U=hubbard_Us,
                  nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
        calc.get_linear_response_Us(indexes)
```

Table S4: Calculated linear response U values of all rutile dioxides

System	U
MoO2	4.83
IrO2	5.91
RuO2	6.73
PtO2	6.25
TiO2	4.95
RhO2	5.97
NbO2	3.32
ReO2	5.27
MnO2	6.63
CrO2	7.15

3 Calculation of adsorption energies at $U = 0$

We first calculate adsorption energies at $U = 0$ for several reasons. One, we would like good initial guesses of both the bare surface and surface with adsorbates for calculations with U. Two, we also test the two layer slab, which we wish to use for calculations with U. We want to see whether the two layer slabs gives similar adsorption energies and falls on the same scaling relationship as the four layer slab. Note, the original scaling relationships and activity comparisons were done on the four layer slab. Third, the data given by these calculations also gives us the scaling relationships we need for comparison to $U > 0$ data.

3.1 Two layer slabs

3.1.1 Relaxation of bare slabs

The code below first relaxes the surface from the bulk crystal coordinates. It takes information from the bulk structure and constructs a two layer surface slab.

```
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = ('MnO2', 'CrO2')

for name, a, c, u in data:
    if name in mag_elements:
```

```
        atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6, base=2, layers=7, vacuum=10)
        nspin=2
    else:
        atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.0, base=2, layers=7, vacuum=10)
        nspin=1

    constraints = []
    for i, atom in enumerate(atoms):
        if atom.symbol != 'H':
            constraints.append(FixScaled(atoms.get_cell(), i,
                                         [True, True, False]))
        else:
            constraints.append(FixScaled(atoms.get_cell(), i,
                                         [False, False, False]))

    atoms.set_constraint(constraints)

    with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals()),
                  atoms=atoms,
                  calculation='relax', disk_io='none',
                  ecutwfc=40.0, ecutrho=500.0,
                  occupations='smearing', smearing='mp', degauss=0.01,
                  kpts=(4, 4, 1), nspin=nspin,
                  nodes=2, ppn=8, walltime='48:00:00') as calc:
        try:
            calc.calculate()
            print calc.espressodir, 'Complete'
        except (EspressoSubmitted, EspressoRunning):
            print calc.espressodir, 'running'
        except (EspressoNotConverged):
            print calc.espressodir, 'Not Converged'
            calc.set(mixing_beta=0.3)
            calc.write_input()
            calc.run(series=True)
```

3.1.2 Calculation of adsorption energies at $U = 0$

The code below takes the relaxed surfaces calculated in the previous code, attaches the adsorbate onto the *5cus* site, and relaxes the surface. For magnetic systems, we also constrain

the total magnetic moment to speed up convergence. The total magnetic moment is chosen after first performing a static calculation without a constrained magnetic moment, reading the magnetic moment from the converged calculation, and then applying the magnetic moment.

```
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23},
                'CrO2': {'O':14, 'OH':15, 'OOH':15}}

O = Atom('O', (0, 0, 0))
OH = Atoms([Atom('O', (0, 0, 0)),
            Atom('H', (-0.85, 0, 0.35))])
OOH = Atoms([Atom('O', (0, 0, 0)),
             Atom('O', (0, -1.165, 0.686)),
             Atom('H', (0, -0.8689, 1.633))])

for name, a, c, u in data:
    for ads in ('O', 'OH', 'OOH'):
        if name in mag_elements:
            nspin = 2
            tot_mag = mag_elements[name][ads]
        else:
            nspin = 1
            tot_mag = None
        if ads is not 'O':
```

```
            h = 0.04
        else:
            h = -0.2
        with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals())) as calc:
            atoms = calc.atoms.copy()
        slab_x = atoms.get_cell()[0][0]
        slab_y = atoms.get_cell()[1][1]

        z_metal = atoms.get_positions()[22][2]
        z_oxy = atoms.get_positions()[25][2]

        if z_metal < z_oxy:
            h = 2 - (z_oxy - z_metal)
        else:
            h = 2

        add_adsorbate(atoms, eval(ads), height=h,
                      position=(slab_x * 0.5, slab_y * 0.5))

        constraints = []
        indexes = range(len(atoms))
        ads_indexes = indexes[-3:]

        for i, atom in enumerate(atoms):
            if i in ads_indexes:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [False, True, False]))
            else:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [True, True, True]))
        atoms.set_constraint(constraints)


        with Espresso('supporting-data/{name}/Eads-2-layers/{ads}'.format(**locals()),
                      atoms=atoms,
                      calculation='relax', disk_io='none',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
                      tot_magnetization=tot_mag,
                      nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
```

```
        try:
            print calc.espressodir, calc.get_potential_energy()
        except (EspressoSubmitted, EspressoRunning):
            print calc.espressodir, 'running'
        except (EspressoNotConverged):
            print calc.espressodir, 'Not Converged'
            calc.write_input()
            calc.run(series=True)
        except:
            print calc.espressodir, 'error'
```

3.2 Four layer slabs

3.2.1 Relaxation of bare slabs

The code below first relaxes the surface from the bulk crystal coordinates. It takes information from the bulk structure and constructs a four layer surface slab.

```
from espresso import *
from ase_addons.surfaces import rutile110

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = ('MnO2', 'CrO2')

for name, a, c, u in data:
    if name in mag_elements:
        atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6,
                          base=3, layers=12, vacuum=12, fixlayers=6)
        nspin=2
```

```
    else:
        atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.0,
                          base=3, layers=12, vacuum=12, fixlayers=6)
        nspin=1

    with Espresso('supporting-data/{name}/Eads-4-layers/bare'.format(**locals()),
                  atoms=atoms,
                  calculation='relax', disk_io='none',
                  ecutwfc=40.0, ecutrho=500.0,
                  occupations='smearing', smearing='mp', degauss=0.01,
                  kpts=(4, 4, 1), nspin=nspin,
                  nodes=3, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
        try:
            calc.calculate()
            print calc.espressodir, 'Converged'
        except (EspressoSubmitted, EspressoRunning):
            print calc.espressodir, 'running'
        except (EspressoNotConverged):
            print calc.espressodir, 'Not Converged'
            calc.set(mixing_beta=0.3)
            calc.write_input()
            calc.run(series=True)
```

3.2.2 Calculation of adsorption energies at $U = 0$

The code below takes the relaxed surfaces calculated in the previous code, attaches the adsorbate onto the *5cus* site, and relaxes the surface. For magnetic systems, we also constrain the total magnetic moment to speed up convergence. The total magnetic moment is chosen after first performing a static calculation without a constrained magnetic moment, reading the magnetic moment from the converged calculation, and then applying the magnetic moment.

For four layer slabs, the relaxations are done in two steps. First, only the adsorbate is allowed to relax. After this calculation has converged, we then allow the top two slabs to relax. Both scripts are below.

```python
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = {'MnO2': {'O':46, 'OH':47, 'OOH':47},
                'CrO2': {'O':30, 'OH':31, 'OOH':31}}

O = Atom('O', (0, 0, 0))
OH = Atoms([Atom('O', (0, 0, 0)),
            Atom('H', (-0.85, 0, 0.35))])
OOH = Atoms([Atom('O', (0, 0, 0)),
             Atom('O', (0, -1.165, 0.686)),
             Atom('H', (0, -0.8689, 1.633))])

for name, a, c, u in data:
    for ads in ('O', 'OH', 'OOH'):
        if name in mag_elements:
            nspin = 2
            tot_mag = mag_elements[name][ads]
        else:
            nspin = 1
            tot_mag = None

        with Espresso('supporting-data/{name}/Eads-4-layers/bare'.format(**locals())) as calc:
            atoms = calc.atoms.copy()
        slab_x = atoms.get_cell()[0][0]
        slab_y = atoms.get_cell()[1][1]

        z_metal = atoms.get_positions()[44][2]
```

```
        z_oxy = atoms.get_positions()[47][2]

        if z_metal < z_oxy:
            h = 2 - (z_oxy - z_metal)
        else:
            h = 2

        add_adsorbate(atoms, eval(ads), height=h,
                      position=(slab_x * 0.5, slab_y * 0.5))

        constraints = []
        indexes = range(len(atoms))
        ads_indexes = indexes[-3:]

        for i, atom in enumerate(atoms):
            if i in ads_indexes:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [False, True, False]))
            else:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [True, True, True]))
        atoms.set_constraint(constraints)

        with Espresso('supporting-data/{name}/Eads-4-layers/{ads}'.format(**locals()),
                      atoms=atoms,
                      calculation='relax', disk_io='none',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
                      nodes=2, ppn=16, processor='xeon16', walltime='48:00:00') as calc:
            try:
                print calc.get_potential_energy()
            except (EspressoSubmitted, EspressoRunning):
                print calc.espressodir, 'running'
            except (EspressoNotConverged):
                print calc.espressodir, 'Not Converged'
                calc.set(tot_magnetization=tot_mag, mixing_beta=0.1)
                calc.write_input()
                calc.run(series=True)
            except:
                print calc.espressodir, 'error'
```

```python
import glob
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = {'MnO2': {'O':46, 'OH':47, 'OOH':47},
                'CrO2': {'O':30, 'OH':31, 'OOH':31}}

O = Atom('O', (0, 0, 0))
OH = Atoms([Atom('O', (0, 0, 0)),
            Atom('H', (-0.85, 0, 0.35))])
OOH = Atoms([Atom('O', (0, 0, 0)),
             Atom('O', (-0.5, 0.5, 1.2)),
             Atom('H', (0, 0, 1.9))])

for name, a, c, u in data:
    for ads in ('O', 'OH', 'OOH'):
        if name in mag_elements:
            atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6,
                              base=3, layers=12, vacuum=12, fixlayers=6)
            nspin = 2
            tot_mag = mag_elements[name][ads]
        else:
            atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.0,
                              base=3, layers=12, vacuum=12, fixlayers=6)
            nspin = 1
            tot_mag = None
        if ads is not 'O':
```

```
            h = 0.04
        else:
            h = -0.2

        slab_x = atoms.get_cell()[0][0]
        slab_y = atoms.get_cell()[1][1]

        add_adsorbate(atoms, eval(ads), height=h,
                      position=(slab_x * 0.5, slab_y * 0.5))
        with Espresso('supporting-data/{name}/Eads-4-layers/{ads}'.format(**locals())) as calc:
            old_atoms = calc.get_atoms()

        atoms.set_positions(old_atoms.get_positions())

        with Espresso('supporting-data/{name}/Eads-4-layers/{ads}-relax'.format(**locals()),
                      atoms=atoms,
                      calculation='relax', disk_io='none',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
                      tot_magnetization=tot_mag,
                      nodes=2, ppn=16, processor='xeon16', walltime='48:00:00') as calc:
            try:
                calc.calculate()
                print calc.espressodir, 'Converged', calc.get_walltime()
            except (EspressoSubmitted, EspressoRunning):
                print calc.espressodir, 'running'
            except (EspressoNotConverged):
                # Get the number of times this job has been run...
                n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '.o*'))
                if calc.electronic_converged == False:
                    print '|{0}|{2}|Electronic|{1:d}|'.format(name, n, ads)
                    calc.set(mixing_beta=0.3, Hubbard_U=hubbard_Us)
                else:
                    print '|{0}|{2}|Structural|{1:d}|'.format(name, n, ads)
                    calc.write_input()
                    calc.run(series=True)
```

4 Analysis of adsorption energies at $U = 0$

The code below takes the adsorption energies calculated at $U = 0$ and constructs Figure 1 in the manuscript. The purpose of this analysis is to validate the two layer slab and determine the scaling relationships from this set of data. The raw adsorption data is summarized in the table below.

Table S5: Adsorption energies of OH, O, and OOH on two and four layer slabs. All units are in eV.

Surface	Layers	OH	O	OOH
MoO2	2	0.208	1.177	3.720
IrO2	2	0.234	1.903	3.969
RuO2	2	0.819	2.535	4.456
PtO2	2	1.246	3.624	4.861
TiO2	2	2.657	5.034	5.463
RhO2	2	1.186	3.363	4.775
NbO2	2	0.062	1.278	3.558
ReO2	2	-0.524	-0.123	3.028
MnO2	2	1.987	4.014	5.291
CrO2	2	1.546	3.166	5.016
MoO2	4	1.043	2.635	4.377
IrO2	4	0.233	1.939	3.932
RuO2	4	0.756	2.372	4.375
PtO2	4	1.227	3.592	4.793
TiO2	4	3.039	5.932	5.650
RhO2	4	1.211	3.421	4.756
NbO2	4	0.374	1.774	3.799
ReO2	4	-0.413	0.151	3.145
MnO2	4	2.180	4.225	5.312
CrO2	4	1.777	3.549	5.129

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

data = [['MoO2', 2,  0.208,  1.177, 3.720],
        ['IrO2', 2,  0.234,  1.903, 3.969],
        ['RuO2', 2,  0.819,  2.535, 4.456],
        ['PtO2', 2,  1.246,  3.624, 4.861],
        ['TiO2', 2,  2.657,  5.034, 5.463],
```

```
        ['RhO2', 2,  1.186,  3.363, 4.775],
        ['NbO2', 2,  0.062,  1.278, 3.558],
        ['ReO2', 2, -0.524, -0.123, 3.028],
        ['MnO2', 2,  1.987,  4.014, 5.291],
        ['CrO2', 2,  1.546,  3.166, 5.016],
        ['MoO2', 4,  1.043,  2.635, 4.377],
        ['IrO2', 4,  0.233,  1.939, 3.932],
        ['RuO2', 4,  0.756,  2.372, 4.375],
        ['PtO2', 4,  1.227,  3.592, 4.793],
        ['TiO2', 4,  3.039,  5.932, 5.650],
        ['RhO2', 4,  1.211,  3.421, 4.756],
        ['NbO2', 4,  0.374,  1.774, 3.799],
        ['ReO2', 4, -0.413,  0.151, 3.145],
        ['MnO2', 4,  2.180,  4.225, 5.312],
        ['CrO2', 4,  1.777,  3.549, 5.129]]

OH_2layer, O_2layer, OOH_2layer = [], [], []
OH_4layer, O_4layer, OOH_4layer = [], [], []

for atoms, layers, OH, O, OOH in data:
    if layers == 2:
        OH_2layer.append(OH)
        O_2layer.append(O)
        OOH_2layer.append(OOH)
    else:
        OH_4layer.append(OH)
        O_4layer.append(O)
        OOH_4layer.append(OOH)

fig = plt.figure(1, (3.5, 4))

ax1 = fig.add_axes([0.15, 0.12, 0.33, 0.38])

ax1.plot(OH_2layer, OH_4layer, marker='o', ls='none', label='OH')
ax1.plot(O_2layer, O_4layer, marker='s', ls='none', label='O')
ax1.plot(OOH_2layer, OOH_4layer, marker='^', ls='none', label='OOH')
ax1.plot((-1, 7), (-1, 7), ls='--', c='k')

ax1.set_xlabel(r'$\Delta E_{ads}^{\mathrm{2\/layers}}$ (eV)', size='small')
ax1.set_ylabel(r'$\Delta E_{ads}^{\mathrm{4\/layers}}$ (eV)', size='small')
ax1.set_xticks([0, 2, 4, 6])
```

```
ax1.set_yticks([0, 2, 4, 6])
ax1.set_xlim(-1, 6.5)
ax1.set_ylim(-1, 6.5)

ax1.text(-0.3, 5.3, 'b)')

# Plot scaling relationships

# First figure out fits on scaling relationships

OH = OH_2layer + OH_4layer
O = O_2layer + O_4layer
OOH = OOH_2layer + OOH_4layer

OH_O_params = np.polyfit(OH, O, 1)
OH_OOH_params = np.polyfit(OH, OOH, 1)

OH_O_slope = OH_O_params[1]
OH_OOH_slope = OH_OOH_params[1]

OH_O_fit = np.poly1d(OH_O_params)
OH_OOH_fit = np.poly1d(OH_OOH_params)

ax2 = fig.add_axes([0.52, 0.12, 0.33, 0.38])

ax2.plot(OH_2layer, O_2layer, marker='o',
         ls='none', c='r', label='O (2 layers)')
ax2.plot(OH_2layer, OOH_2layer, marker='o',
         ls='none', c='g', label='OOH (2 layers)')
ax2.plot(OH_4layer, O_4layer, marker='s',
         ls='none', c='orange', label='O (4 layers)')
ax2.plot(OH_4layer, OOH_4layer, marker='s',
         ls='none', c='greenyellow', label='OOH (4 layers)')
ax2.set_xticks([0, 1, 2, 3])
ax2.set_yticks([0, 2, 4, 6])
ax2.set_yticklabels([])
ax2.set_ylim(-1, 6.5)

ax2.plot((-1, 3.5), OH_O_fit([-1, 3.5]), ls='--', c='r')
ax2.plot((-1, 3.5), OH_OOH_fit([-1, 3.5]), ls='--', c='g')

```

```
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$ (eV)', size='small')

ax3 = ax2.twinx()
ax3.set_xticks([0, 1, 2, 3])
ax3.set_yticks([0, 2, 4, 6])
ax3.set_ylim(-1, 6.5)

ax3.set_ylabel(r'$\Delta E_{ads}$ (eV)', size='small')
ax3.text(-0.6, 5.3, 'c)')

# Finally load the images of the structures
ax4 = fig.add_axes([0.05, 0.52, 0.9, 0.4], frameon=False)
ax4.set_xticks([])
ax4.set_yticks([])
img = mpimg.imread('supporting-figures/atoms.png')
ax4.imshow(img)
ax4.text(-50, -15, 'a)')
plt.savefig('figures/FIG1.png', dpi=300)
plt.savefig('figures/FIG1.eps', dpi=300)
plt.show()
```

5 Calculation of adsorption energies at $U > 0$

5.1 Calculation of bare slab at $U > 0$

We first calculate the relaxed surface of the bare, two layer slab at varying U values. The initial guess of the two layer slab is the relaxed slab calculated at U=0.

```
import os
import glob
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
```

```
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23, 'bare':24},
                'CrO2': {'O':14, 'OH':15, 'OOH':15, 'bare':16}}

Us = np.linspace(0.0, 8.0, 17)
Us[0] = 1e-20

print '|System|Ads|U|Status|Times ran|'
print '|---|'

for name, a, c, u in data:
    with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals())) as calc:
        atoms = calc.get_atoms()
    atoms.set_constraint()

    # We now apply the constraints. This is tricky because the H atoms on
    # the bottom of the slab need to be allowed to relax,
    # the atoms in the bulk must
    # fixed x and y, and the adsorbates should be allowed to relax fully

    constraints = []
    for i, atom in enumerate(atoms):
        # Bulk atoms relax in z direction
        if (atom.symbol != 'H'):
            constraints.append(FixScaled(atoms.get_cell(), i,
                                         [True, True, False]))
        else:
            constraints.append(FixScaled(atoms.get_cell(), i,
                                         [False, False, False]))

    atoms.set_constraint(constraints)

    # Fix magnetic moments to speed convergence
    if name in mag_elements:
```

```
        nspin = 2
        tot_mag = mag_elements[name]['bare']
    else:
        nspin = 1
        tot_mag = None

    for U in Us:
        # Assign Hubbard U values
        hubbard_Us = []
        for atom in atoms:
            if atom.symbol != 'O' and atom.symbol != 'H':
                hubbard_Us.append(U)
            else:
                hubbard_Us.append(0)

        with Espresso('supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals()),
                      atoms=atoms,
                      calculation='relax', disk_io='none',
                      ecutwfc=40.0, ecutrho=500.0,
                      occupations='smearing', smearing='mp', degauss=0.01,
                      kpts=(4, 4, 1), nspin=nspin, tot_magnetization=tot_mag,
                      lda_plus_u=True, U_projection_type='atomic',
                      Hubbard_U=hubbard_Us,
                      nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
            try:
                calc.calculate()
            except (EspressoSubmitted, EspressoRunning):
                n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '.o*'))
                print '|{0}|bare|{1:1.1f}|Running|{2:d}|'.format(name, U, n)
            except (EspressoNotConverged):
                # Get the number of times this job has been run...
                n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '.o*'))
                if calc.electronic_converged == False:
                    print '|{0}|bare|{1:1.1f}|Electronic|{2:d}|'.format(name, U, n)
                    calc.set(mixing_beta=0.2, Hubbard_U=hubbard_Us)
                    continue
                else:
                    print '|{0}|bare|{1:1.1f}|Structural|{2:d}|'.format(name, U, n)
                calc.write_input()
                calc.run(series=True)
```

5.2 Calculation of slab with OH, O, and OOH adsorbates at $U > 0$

After the calculation of the relaxed bare slab, we now attach adsorbates to those surfaces and relax the surface.

```
import glob
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23},
                'CrO2': {'O':14, 'OH':15, 'OOH':15}}

Us = np.linspace(0.0, 8.0, 17)
Us[0] = 1e-20

print '|System|Ads|U|Status|Times ran|'
print '|---|'

for name, a, c, u in data:
    for ads in ('O', 'OH', 'OOH'):
        with Espresso('supporting-data/{name}/Eads-2-layers/{ads}-relax-surf'.format(**locals())) as calc:
            atoms = calc.get_atoms()
        atoms.set_constraint()

        # We now apply the constraints. This is tricky because the H atoms on
        # the bottom of the slab need to be allowed to relax,
        # the atoms in the bulk must
        # fixed x and y, and the adsorbates should be allowed to relax fully
```

```
        ads_indexes = range(len(atoms))[30:]

        constraints = []
        for i, atom in enumerate(atoms):
            # Bulk atoms relax in z direction
            if i in ads_indexes:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [False, False, False]))
            elif (atom.symbol != 'H'):
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [True, True, False]))
            else:
                constraints.append(FixScaled(atoms.get_cell(), i,
                                             [False, False, False]))

        atoms.set_constraint(constraints)

        # Fix magnetic moments to speed convergence
        if name in mag_elements:
            nspin = 2
            tot_mag = mag_elements[name][ads]
        else:
            nspin = 1
            tot_mag = None

        for U in Us:
            # Assign Hubbard U values
            hubbard_Us = []
            for atom in atoms:
                if atom.symbol != 'O' and atom.symbol != 'H':
                    hubbard_Us.append(U)
                else:
                    hubbard_Us.append(0)

            with Espresso('supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals()),
                          atoms=atoms,
                          calculation='relax', disk_io='none',
                          ecutwfc=40.0, ecutrho=500.0,
                          occupations='smearing', smearing='mp', degauss=0.01,
                          kpts=(4, 4, 1), nspin=nspin, tot_magnetization=tot_mag,
                          lda_plus_u=True, U_projection_type='atomic',
```

```
                        Hubbard_U=hubbard_Us,
                        nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
                try:
                    calc.calculate()
                except (EspressoSubmitted, EspressoRunning):
                    n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '.o*'))
                    print '|{0}|{3}|{1:1.1f}|Running|{2:d}|'.format(name, U, n, ads)
                except (EspressoNotConverged):
                    # Get the number of times this job has been run...
                    n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '.o*'))
                    if calc.electronic_converged == False:
                        print '|{0}|{3}|{1:1.1f}|Electronic|{2:d}|'.format(name, U, n, ads)
                        calc.set(mixing_beta=0.3, Hubbard_U=hubbard_Us)
                    else:
                        print '|{0}|{3}|{1:1.1f}|Structural|{2:d}|'.format(name, U, n, ads)
                    calc.write_input()
                    calc.run(series=True)
```

6 Analysis of adsorption energies at $U > 0$

The sections below reads the results directly from the calculations and constructs figures that illustrate the dependence of adsorption energies and scaling relationships on U. Section 6.1 analyzes all of the data, while Section 6.2 and 6.3 analyzes select $4d/5d$ and $3d$ systems for presentation in Figures 2 and 3 of the manuscript, respectively.

6.1 Graph all scaling relationships and adsorption energies at different U values

The code below constructs two figures for each system we studied. The first figure displays how the addition of U changes the adsorption energies of OH, O, and OOH, while the second figure displays dependence of the scaling relationship on applying U. All figures are reproduced following the code.

```python
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
import numpy as np

data = [['MoO2', 4.95, 2.73, 0.28],
        ['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['TiO2', 4.65, 2.97, 0.31],
        ['RhO2', 4.55, 3.11, 0.31],
        ['NbO2', 4.94, 2.96, 0.29],
        ['ReO2', 4.95, 2.68, 0.28],
        ['MnO2', 4.36, 2.84, 0.30],
        ['CrO2', 4.38, 2.90, 0.30]]

linUs = [['MoO2', 4.83],
         ['IrO2', 5.91],
         ['RuO2', 6.73],
         ['PtO2', 6.25],
         ['TiO2', 4.95],
         ['RhO2', 5.97],
         ['NbO2', 3.32],
         ['ReO2', 5.27],
         ['MnO2', 6.63],
         ['CrO2', 7.15]]

U_dict = {}
for oxide, U in linUs:
    U_dict[oxide] = U

def ads_energy(bare, OH, O, OOH):
    '''The reaction is shown below
    H2O + *  <=> HO* + H + e
    HO*      <=> O* + H + e
    O* + H2O <=> HOO* + H + e
    HOO*     <=> O2 + H + e
    '''
```

```
    H2 = -31.6933245045 # From H2 in a box
    H2O = -470.68191439 # From H2O in a box

    try:
        OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
    except:
        OH_ads = None

    try:
        O_ads = O - bare - (H2O - H2) + 0.05
    except:
        O_ads = None

    try:
        OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
    except:
        OOH_ads = None

    return OH_ads, O_ads, OOH_ads

Us = np.linspace(0.0, 8.0, 17)

# First get data for two layer slabs

two_layer_energies = []

for name, a, c, u in data:
    O_energies, OH_energies, OOH_energies = [], [], []
    O_Us, OH_Us, OOH_Us = [], [], []
    O_dict, OH_dict, OOH_dict = {}, {}, {}
    for U in Us:
        # First get the slab energy
        calcdir = 'supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                bare = calc.get_potential_energy()
            else:
                continue

        Eads = {}
```

```
    for ads in ('OH', 'O', 'OOH'):
        calcdir = 'supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                Eads[ads] = calc.get_potential_energy()
            else:
                Eads[ads] = None

    energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
    for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
        if E_ads is not None:
            eval(ads + '_energies').append(E_ads)
            eval(ads + '_Us').append(U)
            eval(ads + '_dict')[U] = E_ads

OH_energies_norm = np.array(OH_energies) - OH_energies[0]
O_energies_norm = np.array(O_energies) - O_energies[0]
OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]

# First plot the variation of adsorption energies with U

plt.figure(1, (4.5, 3.5))
plt.plot(OH_Us, OH_energies_norm, marker='o', c='b', label='OH')
plt.plot(O_Us, O_energies_norm, marker='o', c='r', label='O')
plt.plot(OOH_Us, OOH_energies_norm, marker='o', c='g', label='OOH')
plt.axvline(U_dict[name], ls='--', c='k')

plt.title(name + r' $\Delta E_{ads}(U)$')
plt.xlabel('U (eV)')
plt.ylabel(r'$\Delta\Delta E_{ads} (eV)$')
plt.legend(numpoints=1, loc=0, prop={'size':'small'})
plt.tight_layout()
plt.savefig('supporting-figures/{name}-EvsU-ads.png'.format(**locals()))
plt.show()
plt.close()

print '#+CAPTION: EvsU of adsorption energies on {name}'.format(**locals())
print '#+ATTR_LATEX: :placement [H] :width 3.5in'
print '[[./supporting-figures/{name}-EvsU-ads.png]]\n'.format(**locals())

# Now plot the scaling relationships with respect to U
```

```
    E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
    U_OH_OH, U_OH_O, U_OH_OOH = [], [], []

    for U in Us:
        if type(OH_dict.get(U)) == float:
            E_OH_OH.append([OH_dict[U], OH_dict[U]])
            U_OH_OH.append(U)
        if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
            E_OH_O.append([OH_dict[U], O_dict[U]])
            U_OH_O.append(U)
        if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
            E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
            U_OH_OOH.append(U)

    norm = Normalize(vmin=0, vmax=8)

    plt.figure(1, (4.5, 3.5))
    x, y = zip(*E_OH_O)
    plt.scatter(x, y, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
                marker='s', label='O')
    x, y = zip(*E_OH_OOH)
    plt.scatter(x, y, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
                marker='^', label='OOH')

    plt.legend(loc=6, numpoints=1, prop={'size':'small'})

    # Also plot the original scaling relationships from with U calculations
    xs = np.array([min(x), max(x)])
    plt.plot(xs, 1.54 * xs + 1.17, ls='--', c='r')
    plt.plot(xs, 0.77 * xs + 3.67, ls='--', c='g')

    plt.xlabel(r'$\Delta E_{ads}^{OH}$')
    plt.ylabel(r'$\Delta E_{ads}$')

    plt.title(name + r' $\Delta E_{ads}^{OOH}(E_{ads}^{OH}),\Delta E_{ads}^{O}(E_{ads}^{OH})$')

    plt.tight_layout()
    plt.savefig('supporting-figures/{name}-EvsU-scaling.png'.format(**locals()))
    plt.show()
    plt.close()

```

```
    print '#+CAPTION: $\\Delta E_{ads}(U)$ of scaling relationships energies on' + '{name}'.format(**locals())
    print '[[./supporting-figures/{name}-EvsU-scaling.png]]'.format(**locals())
```

./supporting-figures/MoO2-EvsU-ads.png

Figure S11: $\Delta E_{ads}(U)$ of adsorption energies on MoO_2

./supporting-figures/MoO2-EvsU-scaling.png

Figure S12: $\Delta E_{ads}(U)$ of scaling relationships energies on MoO_2

./supporting-figures/IrO2-EvsU-ads.png

Figure S13: $\Delta E_{ads}(U)$ of adsorption energies on IrO_2

./supporting-figures/IrO2-EvsU-scaling.png

Figure S14: $\Delta E_{ads}(U)$ of scaling relationships energies on IrO_2

./supporting-figures/RuO2-EvsU-ads.png

Figure S15: $\Delta E_{ads}(U)$ of adsorption energies on RuO_2

./supporting-figures/RuO2-EvsU-scaling.png

Figure S16: $\Delta E_{ads}(U)$ of scaling relationships energies on RuO_2

./supporting-figures/PtO2-EvsU-ads.png

Figure S17: $\Delta E_{ads}(U)$ of adsorption energies on PtO_2

./supporting-figures/PtO2-EvsU-scaling.png

Figure S18: $\Delta E_{ads}(U)$ of scaling relationships energies on PtO_2

./supporting-figures/TiO2-EvsU-ads.png

Figure S19: $\Delta E_{ads}(U)$ of adsorption energies on TiO_2

./supporting-figures/TiO2-EvsU-scaling.png

Figure S20: $\Delta E_{ads}(U)$ of scaling relationships energies on TiO_2

./supporting-figures/RhO2-EvsU-ads.png

Figure S21: $\Delta E_{ads}(U)$ of adsorption energies on RhO_2

./supporting-figures/RhO2-EvsU-scaling.png

Figure S22: $\Delta E_{ads}(U)$ of scaling relationships energies on RhO_2

./supporting-figures/NbO2-EvsU-ads.png

Figure S23: $\Delta E_{ads}(U)$ of adsorption energies on NbO_2

./supporting-figures/NbO2-EvsU-scaling.png

Figure S24: $\Delta E_{ads}(U)$ of scaling relationships energies on NbO_2

./supporting-figures/ReO2-EvsU-ads.png

Figure S25: $\Delta E_{ads}(U)$ of adsorption energies on ReO_2

./supporting-figures/ReO2-EvsU-scaling.png

Figure S26: $\Delta E_{ads}(U)$ of scaling relationships energies on ReO_2

./supporting-figures/MnO2-EvsU-ads.png

Figure S27: $\Delta E_{ads}(U)$ of adsorption energies on MnO_2

./supporting-figures/MnO2-EvsU-scaling.png

Figure S28: $\Delta E_{ads}(U)$ of scaling relationships energies on MnO_2

./supporting-figures/CrO2-EvsU-ads.png

Figure S29: $\Delta E_{ads}(U)$ of adsorption energies on CrO_2

./supporting-figures/CrO2-EvsU-scaling.png

Figure S30: $\Delta E_{ads}(U)$ of scaling relationships energies on CrO_2

6.2 Sample $4d$ and $5d$ adsorption energies at $U > 0$ graph for manuscript

The code below graphs the dependence of the adsorption energies and scaling relationships on U for two sample $4d$ and $5d$ systems. This analysis will show the general behavior of early and late $4d$ and $5d$ transition systems. The produced figure is Figure 2 in the manuscript.

```
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
import numpy as np

from matplotlib import rc, rcParams

rc('xtick', labelsize=10)
rc('ytick', labelsize=10)

U_dict = {'MoO2': 4.83,
          'IrO2': 5.91,
          'RuO2': 6.73,
          'PtO2': 6.25,
          'TiO2': 4.95,
          'RhO2': 5.97,
          'NbO2': 3.32,
          'ReO2': 5.27,
          'MnO2': 6.63,
          'CrO2': 7.15}

def ads_energy(bare, OH, O, OOH):
    '''The reaction is shown below
    H2O + *  <=> HO* + H + e
    HO*      <=> O* + H + e
    O* + H2O <=> HOO* + H + e
    HOO*     <=> O2 + H + e
    '''

```

```
    H2 = -31.6933245045 # From H2 in a box
    H2O = -470.68191439 # From H2O in a box

    try:
        OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
    except:
        OH_ads = None

    try:
        O_ads = O - bare - (H2O - H2) + 0.05
    except:
        O_ads = None

    try:
        OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
    except:
        OOH_ads = None

    return OH_ads, O_ads, OOH_ads

Us = np.linspace(0.0, 8.0, 17)

# First plot the two figures for IrO2

fig = plt.figure(1, (7, 6))

O_energies, OH_energies, OOH_energies = [], [], []
O_Us, OH_Us, OOH_Us = [], [], []
O_dict, OH_dict, OOH_dict = {}, {}, {}
for U in Us:
    # First get the slab energy
    calcdir = 'supporting-data/IrO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
    with Espresso(calcdir) as calc:
        if calc.converged == True:
            bare = calc.get_potential_energy()
        else:
            continue

    Eads = {}
    for ads in ('OH', 'O', 'OOH'):
        calcdir = 'supporting-data/IrO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
```

```
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                Eads[ads] = calc.get_potential_energy()
            else:
                Eads[ads] = None

    energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
    for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
        if E_ads is not None:
            eval(ads + '_energies').append(E_ads)
            eval(ads + '_Us').append(U)
            eval(ads + '_dict')[U] = E_ads

OH_energies_norm = np.array(OH_energies) - OH_energies[0]
O_energies_norm = np.array(O_energies) - O_energies[0]
OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]

# First plot the variation of adsorption energies with U

ax1 = fig.add_axes([0.1, 0.1, 0.375, 0.375])

ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
ax1.axvline(5.91, c='k', ls='--')
ax1.text(0.3, 0.63, 'c)')
ax1.set_xlabel('U (eV')
ax1.set_ylabel(r'$\Delta\Delta E_{ads} (eV)$')

plt.legend(numpoints=1, loc=6, prop={'size':'small'})

# Now plot the scaling relationships with respect to U
E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
U_OH_OH, U_OH_O, U_OH_OOH = [], [], []

for U in Us:
    if type(OH_dict.get(U)) == float:
        E_OH_OH.append([OH_dict[U], OH_dict[U]])
        U_OH_OH.append(U)
    if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
        E_OH_O.append([OH_dict[U], O_dict[U]])
```

```
        U_OH_O.append(U)
    if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
        E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
        U_OH_OOH.append(U)

norm = Normalize(vmin=0, vmax=8)

ax2 = fig.add_axes([0.575, 0.1, 0.375, 0.375])

x, y1 = zip(*E_OH_O)
ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='d', label='O')
x, y2 = zip(*E_OH_OOH)
ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='v', label='OOH')

ax2.legend(loc=6, numpoints=1, prop={'size':'small'})

# Also plot the original scaling relationships from with U calculations. We
# want to offset these so it starts
xs = np.array([min(x) - 0.1, max(x) + 0.1])
ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
ax2.text(0.07, 4.15, 'd)')
ax2.set_xlim(0.05, 0.55)
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$')
ax2.set_ylabel(r'$\Delta E_{ads}$')

# Plot the color legend
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))
ax3 = fig.add_axes((0.8, 0.31, 0.1, 0.02))
ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
ax3.set_yticks([],[])
ax3.set_xticks((0, 256))
ax3.set_xticklabels((0, 10))
fig.text(0.845, 0.34, 'U', size=10, style='italic')

# Now  plot the two figures for MoO2

fig = plt.figure(1, (7, 6))
```

```
O_energies, OH_energies, OOH_energies = [], [], []
O_Us, OH_Us, OOH_Us = [], [], []
O_dict, OH_dict, OOH_dict = {}, {}, {}
for U in Us:
    # First get the slab energy
    calcdir = 'supporting-data/NbO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
    with Espresso(calcdir) as calc:
        if calc.converged == True:
            bare = calc.get_potential_energy()
        else:
            continue

    Eads = {}
    for ads in ('OH', 'O', 'OOH'):
        calcdir = 'supporting-data/NbO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                Eads[ads] = calc.get_potential_energy()
            else:
                Eads[ads] = None

    energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
    for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
        if E_ads is not None:
            eval(ads + '_energies').append(E_ads)
            eval(ads + '_Us').append(U)
            eval(ads + '_dict')[U] = E_ads

OH_energies_norm = np.array(OH_energies) - OH_energies[0]
O_energies_norm = np.array(O_energies) - O_energies[0]
OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]

# First plot the variation of adsorption energies with U

ax1 = fig.add_axes([0.1, 0.575, 0.375, 0.375])

ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
ax1.axvline(3.32, c='k', ls='--')
```

```
ax1.text(0.3, 0.72, 'a)')
ax1.set_xlabel('U (eV)')
ax1.set_ylabel(r'$\Delta\Delta E_{ads} (eV)$')
plt.legend(numpoints=1, loc=6, prop={'size':'small'})

# Now plot the scaling relationships with respect to U
E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
U_OH_OH, U_OH_O, U_OH_OOH = [], [], []

for U in Us:
    if type(OH_dict.get(U)) == float:
        E_OH_OH.append([OH_dict[U], OH_dict[U]])
        U_OH_OH.append(U)
    if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
        E_OH_O.append([OH_dict[U], O_dict[U]])
        U_OH_O.append(U)
    if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
        E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
        U_OH_OOH.append(U)

norm = Normalize(vmin=0, vmax=8)

ax2 = fig.add_axes([0.575, 0.575, 0.375, 0.375])

x, y1 = zip(*E_OH_O)
ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='d', label='O')
x, y2 = zip(*E_OH_OOH)
ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='v', label='OOH')

ax2.legend(loc=6, numpoints=1, prop={'size':'small'})

# Also plot the original scaling relationships from with U calculations. We want to
# offset these so it starts
xs = np.array([min(x) - 0.05, max(x) + 0.05])
ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
ax2.text(-0.71, 3.4, 'b)')
ax2.set_xlim(-0.725, -0.4)
ax2.set_xticks([-0.7, -0.6, -0.5, -0.4])
```

```
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$')
ax2.set_ylabel(r'$\Delta E_{ads}$')

# Plot the color legend
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))
ax3 = fig.add_axes((0.8, 0.8, 0.1, 0.02))
ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
ax3.set_yticks([],[])
ax3.set_xticks((0, 256))
ax3.set_xticklabels((0, 10))
fig.text(0.845, 0.83, 'U', size=10, style='italic')
fig.savefig('figures/FIG2.png', dpi=300)
fig.savefig('figures/FIG2.eps', dpi=300)
plt.show()
```

6.3 Sample 3d adsorption energies at $U > 0$ graph for manuscript

The code below graphs the dependence of the adsorption energies and scaling relationships on U for two sample 3d systems. This analysis will show the effect of applying a Hubbard U to adsorption on TiO_2 and MnO_2/CrO_2. This is Figure 3 in the manuscript.

```
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
import numpy as np

from matplotlib import rc, rcParams

rc('xtick', labelsize=10)
rc('ytick', labelsize=10)

U_dict = {'MoO2': 4.83,
          'IrO2': 5.91,
          'RuO2': 6.73,
          'PtO2': 6.25,
```

```
            'TiO2': 4.95,
            'RhO2': 5.97,
            'NbO2': 3.32,
            'ReO2': 5.27,
            'MnO2': 6.63,
            'CrO2': 7.15}

def ads_energy(bare, OH, O, OOH):
    '''The reaction is shown below
    H2O + *  <=> HO* + H + e
    HO*      <=> O* + H + e
    O* + H2O <=> HOO* + H + e
    HOO*     <=> O2 + H + e
    '''

    H2 = -31.6933245045 # From H2 in a box
    H2O = -470.68191439 # From H2O in a box

    try:
        OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
    except:
        OH_ads = None

    try:
        O_ads = O - bare - (H2O - H2) + 0.05
    except:
        O_ads = None

    try:
        OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
    except:
        OOH_ads = None

    return OH_ads, O_ads, OOH_ads

Us = np.linspace(0.0, 8.0, 17)

# First plot the two figures for TiO2

fig = plt.figure(1, (7, 6))

```

```
O_energies, OH_energies, OOH_energies = [], [], []
O_Us, OH_Us, OOH_Us = [], [], []
O_dict, OH_dict, OOH_dict = {}, {}, {}
for U in Us:
    # First get the slab energy
    calcdir = 'supporting-data/TiO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
    with Espresso(calcdir) as calc:
        if calc.converged == True:
            bare = calc.get_potential_energy()
        else:
            continue

    Eads = {}
    for ads in ('OH', 'O', 'OOH'):
        calcdir = 'supporting-data/TiO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                Eads[ads] = calc.get_potential_energy()
            else:
                Eads[ads] = None

    energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
    for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
        if E_ads is not None:
            eval(ads + '_energies').append(E_ads)
            eval(ads + '_Us').append(U)
            eval(ads + '_dict')[U] = E_ads

OH_energies_norm = np.array(OH_energies) - OH_energies[0]
O_energies_norm = np.array(O_energies) - O_energies[0]
OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]

# First plot the variation of adsorption energies with U

ax1 = fig.add_axes([0.1, 0.1, 0.375, 0.375])

ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
ax1.axvline(U_dict['TiO2'], c='k', ls='--')
ax1.set_ylim(-0.15, 0.35)
```

```
ax1.text(7.2, 0.3, 'c)')
ax1.set_xlabel('U (eV)')
ax1.set_ylabel(r'$\Delta\Delta E_{ads} (eV)$')

plt.legend(numpoints=1, loc=2, prop={'size':'small'})

# Now plot the scaling relationships with respect to U
E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
U_OH_OH, U_OH_O, U_OH_OOH = [], [], []

for U in Us:
    if type(OH_dict.get(U)) == float:
        E_OH_OH.append([OH_dict[U], OH_dict[U]])
        U_OH_OH.append(U)
    if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
        E_OH_O.append([OH_dict[U], O_dict[U]])
        U_OH_O.append(U)
    if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
        E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
        U_OH_OOH.append(U)

norm = Normalize(vmin=0, vmax=8)

ax2 = fig.add_axes([0.575, 0.1, 0.375, 0.375])

x, y1 = zip(*E_OH_O)
ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='d', label='O')
x, y2 = zip(*E_OH_OOH)
ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='v', label='OOH')

ax2.legend(loc=6, numpoints=1, prop={'size':'small'})

# Also plot the original scaling relationships from with U calculations. We want to
# offset these so it starts
xs = np.array([1.49, 1.61])
ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
ax2.text(1.495, 5.24, 'd)')
ax2.set_xlim(1.49, 1.61)
```

```
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$')
ax2.set_ylabel(r'$\Delta E_{ads}$')

# Plot the color legend
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))
ax3 = fig.add_axes((0.8, 0.28, 0.1, 0.02))
ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
ax3.set_yticks([],[])
ax3.set_xticks((0, 256))
ax3.set_xticklabels((0, 10))
fig.text(0.845, 0.31, 'U', size=10, style='italic')

# Now plot the two figures for MnO2

fig = plt.figure(1, (7, 6))

O_energies, OH_energies, OOH_energies = [], [], []
O_Us, OH_Us, OOH_Us = [], [], []
O_dict, OH_dict, OOH_dict = {}, {}, {}
for U in Us:
    # First get the slab energy
    calcdir = 'supporting-data/MnO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
    with Espresso(calcdir) as calc:
        if calc.converged == True:
            bare = calc.get_potential_energy()
        else:
            continue

    Eads = {}
    for ads in ('OH', 'O', 'OOH'):
        calcdir = 'supporting-data/MnO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                Eads[ads] = calc.get_potential_energy()
            else:
                Eads[ads] = None

    energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
    for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
        if E_ads is not None:
```

```
            eval(ads + '_energies').append(E_ads)
            eval(ads + '_Us').append(U)
            eval(ads + '_dict')[U] = E_ads

OH_energies_norm = np.array(OH_energies) - OH_energies[0]
O_energies_norm = np.array(O_energies) - O_energies[0]
OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]

# First plot the variation of adsorption energies with U

ax1 = fig.add_axes([0.1, 0.575, 0.375, 0.375])

ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
ax1.axvline(U_dict['MnO2'], c='k', ls='--')
ax1.set_ylim(0, 3.0)
ax1.text(7.2, 2.7, 'a)')
ax1.set_xlabel('U (eV)')
ax1.set_ylabel(r'$\Delta\Delta E_{ads} (eV)$')
plt.legend(numpoints=1, loc=2, prop={'size':'small'})

# Now plot the scaling relationships with respect to U
E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
U_OH_OH, U_OH_O, U_OH_OOH = [], [], []

for U in Us:
    if type(OH_dict.get(U)) == float:
        E_OH_OH.append([OH_dict[U], OH_dict[U]])
        U_OH_OH.append(U)
    if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
        E_OH_O.append([OH_dict[U], O_dict[U]])
        U_OH_O.append(U)
    if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
        E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
        U_OH_OOH.append(U)

norm = Normalize(vmin=0, vmax=8)

ax2 = fig.add_axes([0.575, 0.575, 0.375, 0.375])

```

```
x, y1 = zip(*E_OH_O)
ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='d', label='O')
x, y2 = zip(*E_OH_OOH)
ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
            marker='v', label='OOH')

ax2.legend(loc=6, numpoints=1, prop={'size':'small'})

# Also plot the original scaling relationships from with U calculations. We want to
# offset these so it starts
xs = np.array([1.25, 3.25])
ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
ax2.text(1.35, 6.1, 'b)')
ax2.set_xlim(1.25, 3.25)
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$')
ax2.set_ylabel(r'$\Delta E_{ads}$')

# Plot the color legend
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))
ax3 = fig.add_axes((0.8, 0.7, 0.1, 0.02))
ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
ax3.set_yticks([],[])
ax3.set_xticks((0, 256))
ax3.set_xticklabels((0, 10))
fig.text(0.845, 0.73, 'U', size=10, style='italic')
fig.savefig('figures/FIG3.png', dpi=300)
fig.savefig('figures/FIG3.eps', dpi=300)
plt.show()
```

7 Activity trends with DFT+U

This section looks at the oxygen evolution activity of IrO_2, RuO_2, RhO_2, and PtO_2 using DFT and DFT+U with the linear response U. Section 7.1 reads the relevant adsorption energies at varying U values, fits a polynomial to the data, and backs out the approximate

reaction energy at the linear response calculated U value. Since the adsorption energies of the systems we are looking at have smooth, monotonic behavior and we have calculated them in 0.5 eV intervals, we expect our fitted adsorption energy is very close to the adsorption energy calculated with the linear response U value. Section 7.2 then takes the adsorption energies at the linear response U data and uses the atomistic thermodynamic framework summarized in the manuscript to calculate theoretical minimum overpotentials of each system. Figure 4 in the manuscript is also produced by code in Section 7.2.

7.1 Store Gibbs free reaction energies into tables

The adsorption energies of OH, O, and OOH can be used to calculate the thermodynamic activity barriers for oxygen evolution. This is done below for both the adsorption energies at $U = 0$ and $U = U_{calc}$. The code below reads the adsorption energies at all U values and extracts out the reaction energy of each reaction step at the linear response U value. The zero-point energy contributions, taken from a previous study, are included in the calculation of adsorption energies.[?] This amounts to a value of 0.35 eV, 0.05 eV and 0.4 eV added to the adsorption energies of OH, O, and OOH, respectively.

```
from espresso import *
from ase_addons.surfaces import rutile110
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
from scipy.interpolate import interp1d
import numpy as np

data = [['IrO2', 4.54, 3.18, 0.31],
        ['RuO2', 4.53, 3.12, 0.31],
        ['PtO2', 4.59, 3.23, 0.31],
        ['RhO2', 4.55, 3.11, 0.31]]

linUs = [['IrO2', 5.91],
         ['RuO2', 6.73],
```

```
          ['PtO2', 6.25],
          ['RhO2', 5.97]]

U_dict = {}
for oxide, U in linUs:
    U_dict[oxide] = U

def ads_energy(bare, OH, O, OOH):
    '''The reaction is shown below
    H2O + *  <=> HO* + H + e
    HO*      <=> O* + H + e
    O* + H2O <=> HOO* + H + e
    HOO*     <=> O2 + H + e
    '''

    H2 = -31.6933245045 # From H2 in a box
    H2O = -470.68191439 # From H2O in a box

    try:
        OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
    except:
        OH_ads = None

    try:
        O_ads = O - bare - (H2O - H2) + 0.05
    except:
        O_ads = None

    try:
        OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
    except:
        OOH_ads = None

    return OH_ads, O_ads, OOH_ads

Us = np.linspace(0.0, 8.0, 17)

table_header =  '|Name|$\\Delta G_{1}^{U0}$|$\\Delta G_{2}^{U0}$|$\\Delta G_{3}^{U0}$|$\\Delta G_{4}^{U0}$|'
table_header += '$\\Delta G_{1}^{Ucalc}$|$\\Delta G_{2}^{Ucalc}$|$\\Delta G_{3}^{Ucalc}$|$\\Delta G_{4}^{Ucalc}$|'

print '#+CAPTION: Reaction energies at both U=0 the calculated linear response U value'
```

```
print '#+ATTR_LATEX: :placement [H] :align |c|c|c|c|c|c|c|c|c|'
print '#+TBLNAME: rxn-energies'
print '|---|'

for name, a, c, u in data:
    O_energies, OH_energies, OOH_energies = [], [], []
    O_Us, OH_Us, OOH_Us = [], [], []
    for U in Us:
        # First get the slab energy
        calcdir = 'supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
        with Espresso(calcdir) as calc:
            if calc.converged == True:
                bare = calc.get_potential_energy()
            else:
                continue

        Eads = {}
        for ads in ('OH', 'O', 'OOH'):
            calcdir = 'supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
            with Espresso(calcdir) as calc:
                if calc.converged == True:
                    Eads[ads] = calc.get_potential_energy()
                else:
                    Eads[ads] = None

        energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
        for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
            if E_ads is not None:
                eval(ads + '_energies').append(E_ads)
                eval(ads + '_Us').append(U)

    # After the data is collected, read the adsorption energy data at U0 and Ucalc
    O_func = interp1d(O_Us, O_energies, kind='linear')
    OH_func = interp1d(OH_Us, OH_energies, kind='linear')
    OOH_func = interp1d(OOH_Us, OOH_energies, kind='linear')

    E_O_U0 = O_func(0)
    E_OH_U0 = OH_func(0)
    E_OOH_U0 = OOH_func(0)

    E_O_Ucalc = O_func(U_dict[name])
```

```
    E_OH_Ucalc = OH_func(U_dict[name])
    E_OOH_Ucalc = OOH_func(U_dict[name])

    # Now print out the reaction barriers
    r1_U0 = E_OH_U0
    r2_U0 = E_O_U0 - E_OH_U0
    r3_U0 = E_OOH_U0 - E_O_U0
    r4_U0 = 4.92 - E_OOH_U0

    r1_Ucalc = E_OH_Ucalc
    r2_Ucalc = E_O_Ucalc - E_OH_Ucalc
    r3_Ucalc = E_OOH_Ucalc - E_O_Ucalc
    r4_Ucalc = 4.92 - E_OOH_Ucalc

    s = '|{0}|{1:1.3f}|{2:1.3f}|{3:1.3f}|{4:1.3f}|{5:1.3f}|{6:1.3f}|{7:1.3f}|{8:1.3f}|'

    print s.format(name, float(r1_U0), float(r2_U0), float(r3_U0), float(r4_U0),
                   float(r1_Ucalc), float(r2_Ucalc), float(r3_Ucalc), float(r4_Ucalc))
```

Table S6: Reaction energies at both $U = 0$ the calculated linear response U value

Name	ΔG_1^{U0}	ΔG_2^{U0}	ΔG_3^{U0}	ΔG_4^{U0}	ΔG_1^{Ucalc}	ΔG_2^{Ucalc}	ΔG_3^{Ucalc}	ΔG_4^{Ucalc}
IrO2	0.089	1.392	1.983	1.456	0.410	1.539	1.751	1.219
RuO2	0.569	1.102	2.336	0.912	0.824	1.207	2.181	0.708
PtO2	0.964	2.197	1.196	0.563	1.115	2.393	0.922	0.490
RhO2	0.989	1.821	1.489	0.621	1.289	2.015	1.191	0.425

7.2 Graph Gibbs free energy values in volcano plot

The code below takes the data stored in Table S6 and constructs Figure 4 in the manuscript. We also draw the theoretical volcano, which was originally produced in.[?] This is given as

$$\eta^{OER} = \text{Max}[(\Delta G_O - \Delta G_{OH}), 3.2eV - (\Delta G_O - \Delta G_{OH})]/e - 1.23V. \quad (1)$$

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

# The reaction energy at different reaction steps using both DFT and DFT+U. The organization is as follows
```

```
#        Name  , G1_U0, G2_U0, G3_U0, G4_U0, G1_Ucalc, G2_Ucalc, G1_Ucalc, G1_Ucalc
data = [['IrO2', 0.089, 1.392, 1.983, 1.456, 0.410,    1.539,    1.751,    1.219],
        ['RuO2', 0.569, 1.102, 2.336, 0.912, 0.824,    1.207,    2.181,    0.708],
        ['PtO2', 0.964, 2.197, 1.196, 0.563, 1.115,    2.393,    0.922,    0.490],
        ['RhO2', 0.989, 1.821, 1.489, 0.621, 1.289,    2.015,    1.191,    0.425]]

fig = plt.figure(1,(3.25,3.5))
ax = fig.add_subplot(111)

ax.plot((0 - 0.5, 1.6), (1.97 + 0.5, 0.37), color='k')
ax.plot((1.6, 3.2 + 0.5), (0.37, 1.97 + 0.5), color='k')

for name, G1_U0, G2_U0, G3_U0, G4_U0, G1_Ucalc, G2_Ucalc, G3_Ucalc, G4_Ucalc in data:
    eta_U0 = max([G1_U0, G2_U0, G3_U0, G4_U0]) - 1.23
    eta_Ucalc = max([G1_Ucalc, G2_Ucalc, G3_Ucalc, G4_Ucalc]) - 1.23
    p1, = plt.plot(G2_U0, eta_U0, marker='o', c='c', ms=8, ls='none')
    p2, = plt.plot(G2_Ucalc, eta_Ucalc, marker='s', c='r', ms=8, ls='none')
    plt.annotate('', xytext=(G2_U0, eta_U0), xy=(G2_Ucalc, eta_Ucalc),
                 arrowprops=dict(arrowstyle='simple', color='k'))

plt.text(0.55, 1.0, r'$\mathdefault{RuO_{2}}$')
plt.text(1.0, 0.6, r'$\mathdefault{IrO_{2}}$')
plt.text(2.0, 0.66, r'$\mathdefault{RhO_{2}}$')
plt.text(2.3, 1.05, r'$\mathdefault{PtO_{2}}$')

ax.set_ylim(1.3, 0)
ax.set_xlim(0.5, 2.7)
ax.set_ylabel(r'$\eta^{OER}$ (V)')
ax.set_xlabel('$\Delta G_{O*} - \Delta G_{OH*}$ (eV)')
plt.legend([p1, p2], ['DFT', r'DFT+$\mathdefault{U_{calc}}$'], numpoints=1, prop={'size':'medium'})
plt.tight_layout()
plt.savefig('figures/FIG4.png', dpi=300)
plt.savefig('figures/FIG4.eps', dpi=300)
plt.show()
```

8 Required modules

We used the Atomic Simulation Environment (ASE), which can be downloaded at `https://wiki.fysik.dtu.dk/ase/`. In addition, the standard python scientific computing modules

of `numpy`, `scipy`, and `matplotlib` were used. The modules below were custom made by John R. Kitchin and Zhongnan Xu and are required to generate the code and results in this paper.

8.1 `espresso`

`espresso` is the quantum-espresso wrapper Zhongnan Xu wrote to integrate ASE with the QUANTUM-ESPRESSO package. A most recent version of the code can be found at `https://github.com/zhongnanxu/espresso`.

8.2 `ase_addons`

The ase_addons module is a ASE addons module containing code for constructing surfaces, bulk crystal structures, etc. It can be found at `https://github.com/zhongnanxu/ase_addons`.

References

.