

**Supporting information for:**

**A linear response, DFT+U study of trends in the  
oxygen evolution activity of transition metal  
rutile dioxides**

Zhongnan Xu,<sup>†</sup> Jan Rossmeisl,<sup>‡</sup> and John R. Kitchin<sup>\*,†</sup>

*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

---

<sup>\*</sup>To whom correspondence should be addressed

<sup>†</sup>Department of Chemical Engineering, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213

<sup>‡</sup>Center for Atomic-Scale Materials Design, Department of Physics, Technical University of Denmark, Building 307, 2800 Kgs. Lyngby, Denmark

# Contents


<b>1</b>	<b>Introduction</b>	<b>S3</b>
<b>2</b>	<b>Calculation of bulk properties</b>	<b>S4</b>
2.1	Lattice constants and magnetic ordering . . . . .	S4
2.1.1	Equilibrium volume guess from coarse EOS . . . . .	S4
2.1.2	Equilibrium volume from fine EOS along with ground state magnetic ordering . . . . .	S6
2.1.3	Final relaxation at equilibrium volume and magnetic ordering . . . . .	S14
2.2	Linear response $U$ values . . . . .	S16
<b>3</b>	<b>Calculation of adsorption energies at <math>U = 0</math></b>	<b>S18</b>
3.1	Two layer slabs . . . . .	S18
3.1.1	Relaxation of bare slabs . . . . .	S18
3.1.2	Calculation of adsorption energies at $U = 0$ . . . . .	S19
3.2	Four layer slabs . . . . .	S22
3.2.1	Relaxation of bare slabs . . . . .	S22
3.2.2	Calculation of adsorption energies at $U = 0$ . . . . .	S23
<b>4</b>	<b>Analysis of adsorption energies at <math>U = 0</math></b>	<b>S28</b>
<b>5</b>	<b>Calculation of adsorption energies at <math>U &gt; 0</math></b>	<b>S31</b>
5.1	Calculation of bare slab at $U > 0$ . . . . .	S31
5.2	Calculation of slab with OH, O, and OOH adsorbates at $U > 0$ . . . . .	S34
<b>6</b>	<b>Analysis of adsorption energies at <math>U &gt; 0</math></b>	<b>S36</b>
6.1	Graph all scaling relationships and adsorption energies at different $U$ values	S36
6.2	Sample 4d and 5d adsorption energies at $U > 0$ graph for manuscript . . . . .	S51
6.3	Sample 3d adsorption energies at $U > 0$ graph for manuscript . . . . .	S57

<b>7</b>	<b>Activity trends with DFT+<math>U</math></b>	<b>S63</b>
7.1	Store Gibbs free reaction energies into tables . . . . .	S64
7.2	Graph Gibbs free energy values in volcano plot . . . . .	S67
<b>8</b>	<b>Required modules</b>	<b>S68</b>
8.1	<code>espresso</code> . . . . .	S69
8.2	<code>ase_addons</code> . . . . .	S69

# 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  $\text{IrO}_2$ ,  $\text{RuO}_2$ ,  $\text{PtO}_2$ , and  $\text{RhO}_2$  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.

---

```
1 from espresso import *
2 from ase_addons import bulk
3 from ase.utils.eos import EquationOfState
4 from ase.visualize import view
5 import numpy as np
6
```

```

7  vol = 62
8  factors = (0.8, 0.9, 1.0, 1.1, 1.2)
9
10 elements = ['Mo', 'Ir', 'Ru', 'Pt', 'Ti',
11             'Nb', 'Re', 'Rh', 'Mn', 'Cr']
12
13 print '#+CAPTION: Equilibrium volumes calculated from 3rd order polynomial fit to a coarse EOS'
14 print '#+ATTR_LATEX: :placement [H] c|c'
15 print '#+TBLNAME: coarse-EOS'
16 print '|Oxide|V ($\\AA$/primitive cell)|'
17 print '|----|'
18
19 for name in elements:
20     ready = True
21     volumes, energies = [], []
22     for fac in factors:
23         atoms = bulk.rutile((name, 'O'), mags=(0.6, 0))
24         atoms.set_volume(fac * vol)
25         calcdir = 'supporting-data/{name}O2/coarse-EOS/{name}O2-v-{fac:1.1f}'.format(**locals())
26         with Espresso(calcdir, atoms=atoms,
27                       disk_io='none', calculation='vc-relax',
28                       ecutwfc=40.0, ecutrho=500.0,
29                       occupations='smearing', smearing='mp', degauss=0.01,
30                       nspin=2, cell_dofree='shape',
31                       kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
32             try:
33                 energy = calc.get_potential_energy()
34                 energies.append(energy)
35                 volumes.append(atoms.get_volume())
36             except (EspressoSubmitted, EspressoRunning):
37                 ready = False
38                 print calc.espressodir, 'running'
39             except (EspressoNotConverged):
40                 ready = False
41                 calc.set(mixing_beta=0.3)
42
43 eos = EquationOfState(volumes, energies)
44 v0, e0, B = eos.fit()
45 eos.plot('supporting-figures/{name}O2-coarse-EOS.png'.format(**locals()), show=False)
46 print '|{name}O2|{v0:1.3f}|'.format(**locals())

```

---

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

Oxide	V ( $\text{\AA}$ /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.

---

```

1  from espresso import *
2  from ase.utils.eos import EquationOfState
3  from ase.visualize import view
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from ase_addons import bulk
7
8  data = [['MoO2', 66.048],
9          ['IrO2', 65.657],
10         ['RuO2', 64.122],
11         ['PtO2', 68.059],
12         ['TiO2', 64.265],
13         ['RhO2', 64.405],
14         ['NbO2', 72.301],
15         ['ReO2', 64.617],
16         ['MnO2', 56.996],
17         ['CrO2', 57.817]]
18

```

```

19  factors = (0.9, 0.95, 1.0, 1.05, 1.10)
20
21  print '#+CAPTION: Equilibrium volumes calculated from 3rd order polynomial fit to a Fine EOS'
22  print '#+ATTR_LATEX: :placement [H] c|c'
23  print '#+TBLNAME: fine-EOS'
24  print '|Oxide|V ($\\AA$/primitive cell)|'
25  print '|----|'
26
27  for name, vol in data:
28      plt.figure(1, (4.5, 3))
29      volumes, energies = [], []
30      for fac in factors:
31          atoms = bulk.rutile((name[:-2], 'O'), mags=(0.6, 0))
32          atoms.set_volume(fac * vol)
33          calcdir = 'supporting-data/{name}/fine-EOS/ferro/{name}-v-{fac:1.2f}'.format(**locals())
34          with Espresso(calcdir, atoms=atoms,
35                        disk_io='none', calculation='vc-relax',
36                        ecutwfc=40.0, ecutrho=500.0,
37                        occupations='smearing', smearing='mp', degauss=0.01,
38                        nspin=2, cell_dofree='shape', mixing_beta=0.3,
39                        kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
40              try:
41                  energy = calc.get_potential_energy()
42                  energies.append(energy)
43                  volumes.append(atoms.get_volume())
44              except (EspressoSubmitted, EspressoRunning):
45                  pass
46              except (EspressoNotConverged):
47                  pass
48
49      min_E = min(energies)
50      energies = np.array(energies) - min_E
51
52      fit = np.poly1d(np.polyfit(volumes, energies, 3))
53      fit_vols = np.linspace(min(volumes), max(volumes))
54      plt.plot(volumes, energies, marker='o', ls='none', label='Ferromagnetic', c='r')
55      plt.plot(fit_vols, fit(fit_vols), c='r')
56
57      volumes, energies = [], []
58      for fac in factors:
59          atoms = bulk.rutile((name[:-2], 'O'), mags=(0, 0))

```

```

60     atoms.set_volume(fac * vol)
61     calcdir = 'supporting-data/{name}/fine-EOS/non-mag/{name}-v-{fac:1.2f}'.format(**locals())
62     with Espresso(calcdir, atoms=atoms,
63                   disk_io='none', calculation='vc-relax',
64                   ecutwfc=40.0, ecutrho=500.0,
65                   occupations='smearing', smearing='mp', degauss=0.01,
66                   nspin=2, cell_dofree='shape', mixing_beta=0.3,
67                   kpts=(5, 5, 5), walltime='18:00:00', ppn=4) as calc:
68         try:
69             energy = calc.get_potential_energy()
70             energies.append(energy)
71             volumes.append(atoms.get_volume())
72         except (EspressoSubmitted, EspressoRunning):
73             pass
74         except (EspressoNotConverged):
75             pass
76
77     energies = np.array(energies) - min_E
78
79     fit = np.poly1d(np.polyfit(volumes, energies, 3))
80     fit_vols = np.linspace(min(volumes), max(volumes))
81     plt.plot(volumes, energies, marker='o', ls='none', label='Non-magnetic', c='k')
82     plt.plot(fit_vols, fit(fit_vols), c='k')
83
84     plt.xlabel('Volume')
85     plt.ylabel('Relative Energy (eV)')
86     plt.title('{name}'.format(**locals()))
87     plt.legend(loc=9, prop={'size': 'small'}, numpoints=1)
88     plt.tight_layout()
89     plt.savefig('supporting-figures/{name}-fine-EOS.png'.format(**locals()))
90     plt.show()
91
92     eos = EquationOfState(volumes, energies)
93     v0, e0, B = eos.fit()
94     print '|{name}|{v0:1.3f}|'.format(**locals())
95
96 for name, vol in data:
97     print '\n#+CAPTION: 3rd order polynomial equation of state for bulk {name}'.format(**locals())
98     print '##+ATTR_LATEX: :placement [H]'
99     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 ( $\text{\AA}$ /primitive cell)
MoO <sub>2</sub>	66.764
IrO <sub>2</sub>	65.638
RuO <sub>2</sub>	64.080
PtO <sub>2</sub>	68.074
TiO <sub>2</sub>	64.167
RhO <sub>2</sub>	64.362
NbO <sub>2</sub>	72.268
ReO <sub>2</sub>	65.688
MnO <sub>2</sub>	54.081
CrO <sub>2</sub>	55.677



Figure S1: 3rd order polynomial equation of state for bulk MoO<sub>2</sub>

For MoO<sub>2</sub>, 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.

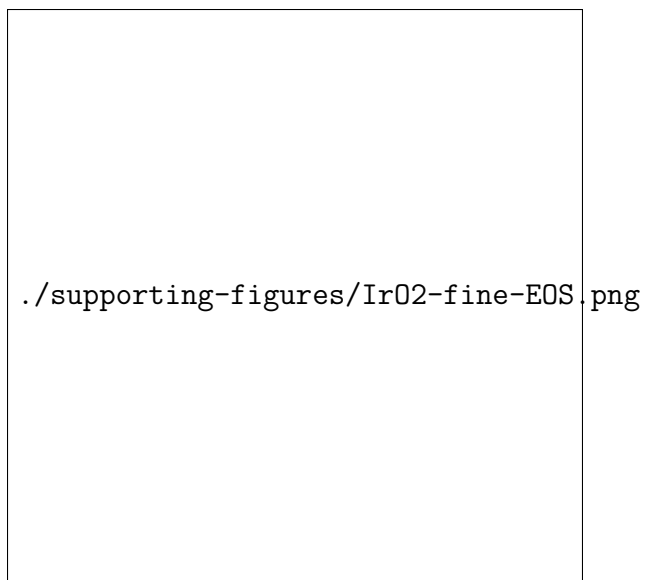


Figure S2: 3rd order polynomial equation of state for bulk IrO<sub>2</sub>

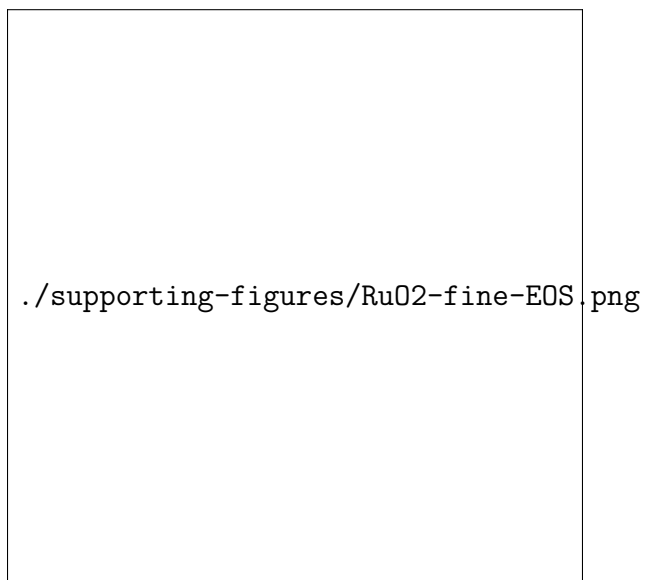


Figure S3: 3rd order polynomial equation of state for bulk RuO<sub>2</sub>

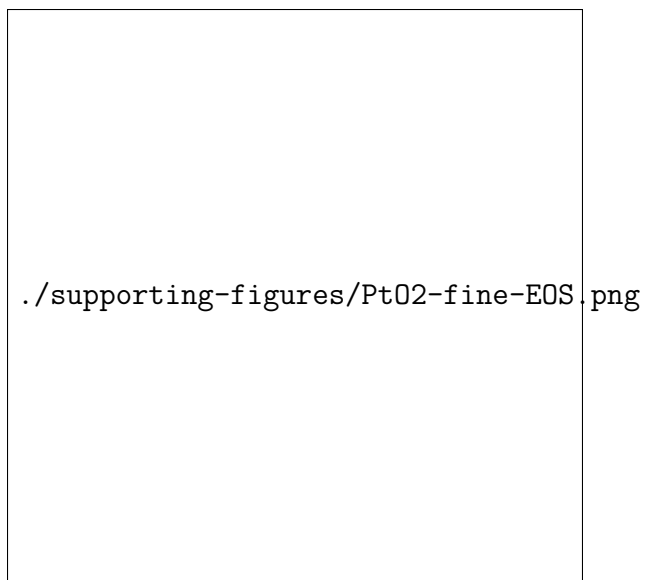


Figure S4: 3rd order polynomial equation of state for bulk PtO<sub>2</sub>



Figure S5: 3rd order polynomial equation of state for bulk TiO<sub>2</sub>



Figure S6: 3rd order polynomial equation of state for bulk  $\text{RhO}_2$



Figure S7: 3rd order polynomial equation of state for bulk  $\text{NbO}_2$



Figure S8: 3rd order polynomial equation of state for bulk  $\text{ReO}_2$



Figure S9: 3rd order polynomial equation of state for bulk  $\text{MnO}_2$

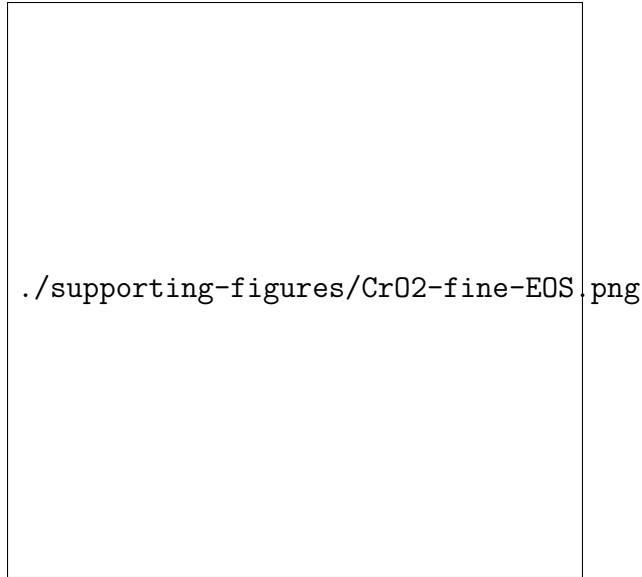


Figure S10: 3rd order polynomial equation of state for bulk  $\text{CrO}_2$

From these results, we see that only the  $3d$  oxides of  $\text{CrO}_2$ , and  $\text{MnO}_2$  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.

---

```
1 from espresso import *
2 from ase.utils.eos import EquationOfState
3 from ase.visualize import view
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from ase_addons import bulk
7
8 data = [['MoO2', 66.764],
9         ['IrO2', 65.638],
10        ['RuO2', 64.080],
11        ['PtO2', 68.074],
12        ['TiO2', 64.167],
13        ['RhO2', 64.362],
```

```

14         ['NbO2', 72.268],
15         ['ReO2', 65.688],
16         ['MnO2', 54.081],
17         ['CrO2', 55.677]]
18
19     mag_elements = ('MnO2', 'CrO2')
20
21     print '#+CAPTION: Relaxed lattice coordinates of all rutile structures'
22     print '#+TBLNAME: rutile-struct'
23     print '|System|a|c|u|'
24     print '|----|'
25
26     for name, vol in data:
27         if name in mag_elements:
28             atoms = bulk.rutile((name[:-2], 'O'), mags=(0.6, 0))
29         else:
30             atoms = bulk.rutile((name[:-2], 'O'), mags=(0.0, 0))
31
32     atoms.set_volume(vol)
33     with Espresso('supporting-data/{name}/ground'.format(**locals()),
34                  atoms=atoms,
35                  disk_io='none', calculation='vc-relax',
36                  ecutwfc=40.0, ecutrho=500.0,
37                  occupations='smearing', smearing='mp', degauss=0.01,
38                  nspin=2, cell_dofree='shape', mixing_beta=0.3,
39                  kpts=(5, 5, 5), walltime='18:00:00', ppn=1) as calc:
40         try:
41             calc.calculate()
42             pos = calc.atoms.get_positions()
43             cell = calc.atoms.get_cell()
44             a = cell[0][0]
45             c = cell[2][2]
46             u = pos[2][0] / a
47             print '|{name}|{a:1.2f}|{c:1.2f}|{u:1.2f}|'.format(**locals())
48         except (EspressoSubmitted, EspressoRunning):
49             print calc.espressodir, 'running'
50             pass
51         except (EspressoNotConverged):
52             # calc.write_input()
53             # calc.run(series=True)
54             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).<sup>?</sup> Details of the method and specific executable used can be found at [http://media.quantum-espresso.org/santa\\_barbara\\_2009\\_07/index.php](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.

---

```
1  import numpy as np
2  from espresso import *
3  from ase_addons import bulk
4
5  data = [['MoO2', 4.95, 2.73, 0.28],
6          ['IrO2', 4.54, 3.18, 0.31],
7          ['RuO2', 4.53, 3.12, 0.31],
8          ['PtO2', 4.59, 3.23, 0.31],
9          ['TiO2', 4.65, 2.97, 0.31],
10         ['RhO2', 4.55, 3.11, 0.31],
11         ['NbO2', 4.94, 2.96, 0.29],
12         ['ReO2', 4.95, 2.68, 0.28],
13         ['MnO2', 4.36, 2.84, 0.30],
14         ['CrO2', 4.38, 2.90, 0.30]]
```



```

15
16 mag_elements = ('MnO2', 'CrO2')
17
18 indexes = {0:(0, 1, 6, 7, 12, 13, 18, 19, 24, 25, 30, 31, 36, 37, 42, 43),
19            2:(2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17, 20, 21, 22, 23,
20            26, 27, 28, 29, 32, 33, 34, 35, 38, 39, 40, 41, 44, 45, 46, 47)}
21
22 for name, a, c, u in data:
23     if name in mag_elements:
24         atoms = bulk.rutile((name[:-2], 'O'), a, c, u, mags=(0.6, 0))
25     else:
26         atoms = bulk.rutile((name[:-2], 'O'), a, c, u, mags=(0, 0))
27     atoms.set_constraint()
28     atoms *= (2, 2, 2)
29     hubbard_Us = 1e-20 * np.ones(len(atoms))
30
31     with Espresso('supporting-data/linear-response/{name}'.format(**locals()), atoms=atoms,
32                   ecutwfc=40.0, ecutrho=500.0,
33                   occupations='smearing', smearing='mp', degauss=0.01,
34                   kpts=(4, 4, 4), nspin=2,
35                   lda_plus_u=True, U_projection_type='atomic', Hubbard_U=hubbard_Us,
36                   nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
37         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.

---

```
1 from espresso import *
2 from ase_addons-surfaces import rutile110
3 from ase.visualize import view
4
5 data = [['MoO2', 4.95, 2.73, 0.28],
6         ['IrO2', 4.54, 3.18, 0.31],
7         ['RuO2', 4.53, 3.12, 0.31],
8         ['PtO2', 4.59, 3.23, 0.31],
9         ['TiO2', 4.65, 2.97, 0.31],
10        ['RhO2', 4.55, 3.11, 0.31],
11        ['NbO2', 4.94, 2.96, 0.29],
12        ['ReO2', 4.95, 2.68, 0.28],
13        ['MnO2', 4.36, 2.84, 0.30],
14        ['CrO2', 4.38, 2.90, 0.30]]
15
16 mag_elements = ('MnO2', 'CrO2')
17
18 for name, a, c, u in data:
19     if name in mag_elements:
```

```

20     atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6, base=2, layers=7, vacuum=10)
21     nspin=2
22 else:
23     atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.0, base=2, layers=7, vacuum=10)
24     nspin=1
25
26 constraints = []
27 for i, atom in enumerate(atoms):
28     if atom.symbol != 'H':
29         constraints.append(FixScaled(atoms.get_cell(), i,
30                                     [True, True, False]))
31     else:
32         constraints.append(FixScaled(atoms.get_cell(), i,
33                                     [False, False, False]))
34
35 atoms.set_constraint(constraints)
36
37 with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals()),
38               atoms=atoms,
39               calculation='relax', disk_io='none',
40               ecutwfc=40.0, ecutrho=500.0,
41               occupations='smearing', smearing='mp', degauss=0.01,
42               kpts=(4, 4, 1), nspin=nspin,
43               nodes=2, ppn=8, walltime='48:00:00') as calc:
44     try:
45         calc.calculate()
46         print calc.espressodir, 'Complete'
47     except (EspressoSubmitted, EspressoRunning):
48         print calc.espressodir, 'running'
49     except (EspressoNotConverged):
50         print calc.espressodir, 'Not Converged'
51         calc.set(mixing_beta=0.3)
52         calc.write_input()
53         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.

---

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3  from ase.visualize import view
4  from ase.lattice.surface import add_adsorbate
5
6  data = [['MoO2', 4.95, 2.73, 0.28],
7          ['IrO2', 4.54, 3.18, 0.31],
8          ['RuO2', 4.53, 3.12, 0.31],
9          ['PtO2', 4.59, 3.23, 0.31],
10         ['TiO2', 4.65, 2.97, 0.31],
11         ['RhO2', 4.55, 3.11, 0.31],
12         ['NbO2', 4.94, 2.96, 0.29],
13         ['ReO2', 4.95, 2.68, 0.28],
14         ['MnO2', 4.36, 2.84, 0.30],
15         ['CrO2', 4.38, 2.90, 0.30]]
16
17  mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23},
18                 'CrO2': {'O':14, 'OH':15, 'OOH':15}}
19
20  O = Atom('O', (0, 0, 0))
21  OH = Atoms([Atom('O', (0, 0, 0)),
22             Atom('H', (-0.85, 0, 0.35))])
23  OOH = Atoms([Atom('O', (0, 0, 0)),
24              Atom('O', (0, -1.165, 0.686)),
25              Atom('H', (0, -0.8689, 1.633))])
26
27  for name, a, c, u in data:
28      for ads in ('O', 'OH', 'OOH'):
29          if name in mag_elements:
30              nspin = 2
31              tot_mag = mag_elements[name][ads]
32          else:
33              nspin = 1
34              tot_mag = None
35          if ads is not 'O':

```

```

36         h = 0.04
37     else:
38         h = -0.2
39     with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals())) as calc:
40         atoms = calc.atoms.copy()
41         slab_x = atoms.get_cell()[0][0]
42         slab_y = atoms.get_cell()[1][1]
43
44         z_metal = atoms.get_positions()[22][2]
45         z_oxy = atoms.get_positions()[25][2]
46
47         if z_metal < z_oxy:
48             h = 2 - (z_oxy - z_metal)
49         else:
50             h = 2
51
52     add_adsorbate(atoms, eval(ads), height=h,
53                  position=(slab_x * 0.5, slab_y * 0.5))
54
55     constraints = []
56     indexes = range(len(atoms))
57     ads_indexes = indexes[-3:]
58
59     for i, atom in enumerate(atoms):
60         if i in ads_indexes:
61             constraints.append(FixScaled(atoms.get_cell(), i,
62                                          [False, True, False]))
63         else:
64             constraints.append(FixScaled(atoms.get_cell(), i,
65                                          [True, True, True]))
66     atoms.set_constraint(constraints)
67
68
69     with Espresso('supporting-data/{name}/Eads-2-layers/{ads}'.format(**locals())),
70         atoms=atoms,
71         calculation='relax', disk_io='none',
72         ecutwfc=40.0, ecutrho=500.0,
73         occupations='smearing', smearing='mp', degauss=0.01,
74         kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
75         tot_magnetization=tot_mag,
76         nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:

```

```

77         try:
78             print calc.espressodir, calc.get_potential_energy()
79         except (EspressoSubmitted, EspressoRunning):
80             print calc.espressodir, 'running'
81         except (EspressoNotConverged):
82             print calc.espressodir, 'Not Converged'
83             calc.write_input()
84             calc.run(series=True)
85         except:
86             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.

---

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3
4  data = [['MoO2', 4.95, 2.73, 0.28],
5          ['IrO2', 4.54, 3.18, 0.31],
6          ['RuO2', 4.53, 3.12, 0.31],
7          ['PtO2', 4.59, 3.23, 0.31],
8          ['TiO2', 4.65, 2.97, 0.31],
9          ['RhO2', 4.55, 3.11, 0.31],
10         ['NbO2', 4.94, 2.96, 0.29],
11         ['ReO2', 4.95, 2.68, 0.28],
12         ['MnO2', 4.36, 2.84, 0.30],
13         ['CrO2', 4.38, 2.90, 0.30]]
14
15  mag_elements = ('MnO2', 'CrO2')
16
17  for name, a, c, u in data:
18      if name in mag_elements:
19          atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6,
20                           base=3, layers=12, vacuum=12, fixlayers=6)
21          nspin=2

```

```

22     else:
23         atoms = rutile110((name[:-2], '0'), a, c, u, mag=0.0,
24                             base=3, layers=12, vacuum=12, fixlayers=6)
25         nspin=1
26
27     with Espresso('supporting-data/{name}/Eads-4-layers/bare'.format(**locals()),
28                   atoms=atoms,
29                   calculation='relax', disk_io='none',
30                   ecutwfc=40.0, ecutrho=500.0,
31                   occupations='smearing', smearing='mp', degauss=0.01,
32                   kpts=(4, 4, 1), nspin=nspin,
33                   nodes=3, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
34     try:
35         calc.calculate()
36         print calc.espressodir, 'Converged'
37     except (EspressoSubmitted, EspressoRunning):
38         print calc.espressodir, 'running'
39     except (EspressoNotConverged):
40         print calc.espressodir, 'Not Converged'
41         calc.set(mixing_beta=0.3)
42         calc.write_input()
43         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.

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3  from ase.visualize import view
4  from ase.lattice.surface import add_adsorbate
5
6  data = [['MoO2', 4.95, 2.73, 0.28],
7          ['IrO2', 4.54, 3.18, 0.31],
8          ['RuO2', 4.53, 3.12, 0.31],
9          ['PtO2', 4.59, 3.23, 0.31],
10         ['TiO2', 4.65, 2.97, 0.31],
11         ['RhO2', 4.55, 3.11, 0.31],
12         ['NbO2', 4.94, 2.96, 0.29],
13         ['ReO2', 4.95, 2.68, 0.28],
14         ['MnO2', 4.36, 2.84, 0.30],
15         ['CrO2', 4.38, 2.90, 0.30]]
16
17  mag_elements = {'MnO2': {'O':46, 'OH':47, 'OOH':47},
18                 'CrO2': {'O':30, 'OH':31, 'OOH':31}}
19
20  O = Atom('O', (0, 0, 0))
21  OH = Atoms([Atom('O', (0, 0, 0)),
22             Atom('H', (-0.85, 0, 0.35))])
23  OOH = Atoms([Atom('O', (0, 0, 0)),
24              Atom('O', (0, -1.165, 0.686)),
25              Atom('H', (0, -0.8689, 1.633))])
26
27  for name, a, c, u in data:
28      for ads in ('O', 'OH', 'OOH'):
29          if name in mag_elements:
30              nspin = 2
31              tot_mag = mag_elements[name][ads]
32          else:
33              nspin = 1
34              tot_mag = None
35
36      with Espresso('supporting-data/{name}/Eads-4-layers/bare'.format(**locals())) as calc:
37          atoms = calc.atoms.copy()
38          slab_x = atoms.get_cell()[0][0]
39          slab_y = atoms.get_cell()[1][1]
40
41          z_metal = atoms.get_positions()[44][2]

```



```

42     z_oxy = atoms.get_positions()[47][2]
43
44     if z_metal < z_oxy:
45         h = 2 - (z_oxy - z_metal)
46     else:
47         h = 2
48
49     add_adsorbate(atoms, eval(ads), height=h,
50                   position=(slab_x * 0.5, slab_y * 0.5))
51
52     constraints = []
53     indexes = range(len(atoms))
54     ads_indexes = indexes[-3:]
55
56     for i, atom in enumerate(atoms):
57         if i in ads_indexes:
58             constraints.append(FixScaled(atoms.get_cell(), i,
59                                         [False, True, False]))
60         else:
61             constraints.append(FixScaled(atoms.get_cell(), i,
62                                         [True, True, True]))
63     atoms.set_constraint(constraints)
64
65     with Espresso('supporting-data/{name}/Eads-4-layers/{ads}'.format(**locals()),
66                  atoms=atoms,
67                  calculation='relax', disk_io='none',
68                  ecutwfc=40.0, ecutrho=500.0,
69                  occupations='smearing', smearing='mp', degauss=0.01,
70                  kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
71                  nodes=2, ppn=16, processor='xeon16', walltime='48:00:00') as calc:
72         try:
73             print calc.get_potential_energy()
74         except (EspressoSubmitted, EspressoRunning):
75             print calc.espressodir, 'running'
76         except (EspressoNotConverged):
77             print calc.espressodir, 'Not Converged'
78             calc.set(tot_magnetization=tot_mag, mixing_beta=0.1)
79             calc.write_input()
80             calc.run(series=True)
81         except:
82             print calc.espressodir, 'error'

```

---

---

```

1  import glob
2  from espresso import *
3  from ase_addons-surfaces import rutile110
4  from ase.visualize import view
5  from ase.lattice.surface import add_adsorbate
6
7  data = [['MoO2', 4.95, 2.73, 0.28],
8          ['IrO2', 4.54, 3.18, 0.31],
9          ['RuO2', 4.53, 3.12, 0.31],
10         ['PtO2', 4.59, 3.23, 0.31],
11         ['TiO2', 4.65, 2.97, 0.31],
12         ['RhO2', 4.55, 3.11, 0.31],
13         ['NbO2', 4.94, 2.96, 0.29],
14         ['ReO2', 4.95, 2.68, 0.28],
15         ['MnO2', 4.36, 2.84, 0.30],
16         ['CrO2', 4.38, 2.90, 0.30]]
17
18  mag_elements = {'MnO2': {'O':46, 'OH':47, 'OOH':47},
19                 'CrO2': {'O':30, 'OH':31, 'OOH':31}}
20
21  O = Atom('O', (0, 0, 0))
22  OH = Atoms([Atom('O', (0, 0, 0)),
23             Atom('H', (-0.85, 0, 0.35))])
24  OOH = Atoms([Atom('O', (0, 0, 0)),
25              Atom('O', (-0.5, 0.5, 1.2)),
26              Atom('H', (0, 0, 1.9))])
27
28  for name, a, c, u in data:
29      for ads in ('O', 'OH', 'OOH'):
30          if name in mag_elements:
31              atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.6,
32                               base=3, layers=12, vacuum=12, fixlayers=6)
33              nspin = 2
34              tot_mag = mag_elements[name][ads]
35          else:
36              atoms = rutile110((name[:-2], 'O'), a, c, u, mag=0.0,
37                               base=3, layers=12, vacuum=12, fixlayers=6)
38              nspin = 1
39              tot_mag = None
40          if ads is not 'O':

```

```

41         h = 0.04
42     else:
43         h = -0.2
44
45     slab_x = atoms.get_cell()[0][0]
46     slab_y = atoms.get_cell()[1][1]
47
48     add_adsorbate(atoms, eval(ads), height=h,
49                   position=(slab_x * 0.5, slab_y * 0.5))
50     with Espresso('supporting-data/{name}/Eads-4-layers/{ads}'.format(**locals())) as calc:
51         old_atoms = calc.get_atoms()
52
53     atoms.set_positions(old_atoms.get_positions())
54
55     with Espresso('supporting-data/{name}/Eads-4-layers/{ads}-relax'.format(**locals()),
56                  atoms=atoms,
57                  calculation='relax', disk_io='none',
58                  ecutwfc=40.0, ecutrho=500.0,
59                  occupations='smearing', smearing='mp', degauss=0.01,
60                  kpts=(4, 4, 1), nspin=nspin, mixing_beta=0.3,
61                  tot_magnetization=tot_mag,
62                  nodes=2, ppn=16, processor='xeon16', walltime='48:00:00') as calc:
63         try:
64             calc.calculate()
65             print calc.espressodir, 'Converged', calc.get_walltime()
66         except (EspressoSubmitted, EspressoRunning):
67             print calc.espressodir, 'running'
68         except (EspressoNotConverged):
69             # Get the number of times this job has been run...
70             n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '*.o*'))
71             if calc.electronic_converged == False:
72                 print '{0}|{2}|Electronic|{1:d}|'.format(name, n, ads)
73                 calc.set(mixing_beta=0.3, Hubbard_U=hubbard_Us)
74             else:
75                 print '{0}|{2}|Structural|{1:d}|'.format(name, n, ads)
76                 calc.write_input()
77                 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

---

```
1 import matplotlib.pyplot as plt
2 import matplotlib.image as mpimg
3 import numpy as np
4
5 data = [['MoO2', 2, 0.208, 1.177, 3.720],
6         ['IrO2', 2, 0.234, 1.903, 3.969],
7         ['RuO2', 2, 0.819, 2.535, 4.456],
8         ['PtO2', 2, 1.246, 3.624, 4.861],
9         ['TiO2', 2, 2.657, 5.034, 5.463],
```

```

10         ['RhO2', 2, 1.186, 3.363, 4.775],
11         ['NbO2', 2, 0.062, 1.278, 3.558],
12         ['ReO2', 2, -0.524, -0.123, 3.028],
13         ['MnO2', 2, 1.987, 4.014, 5.291],
14         ['CrO2', 2, 1.546, 3.166, 5.016],
15         ['MoO2', 4, 1.043, 2.635, 4.377],
16         ['IrO2', 4, 0.233, 1.939, 3.932],
17         ['RuO2', 4, 0.756, 2.372, 4.375],
18         ['PtO2', 4, 1.227, 3.592, 4.793],
19         ['TiO2', 4, 3.039, 5.932, 5.650],
20         ['RhO2', 4, 1.211, 3.421, 4.756],
21         ['NbO2', 4, 0.374, 1.774, 3.799],
22         ['ReO2', 4, -0.413, 0.151, 3.145],
23         ['MnO2', 4, 2.180, 4.225, 5.312],
24         ['CrO2', 4, 1.777, 3.549, 5.129]]
25
26 OH_2layer, O_2layer, OOH_2layer = [], [], []
27 OH_4layer, O_4layer, OOH_4layer = [], [], []
28
29 for atoms, layers, OH, O, OOH in data:
30     if layers == 2:
31         OH_2layer.append(OH)
32         O_2layer.append(O)
33         OOH_2layer.append(OOH)
34     else:
35         OH_4layer.append(OH)
36         O_4layer.append(O)
37         OOH_4layer.append(OOH)
38
39 fig = plt.figure(1, (3.5, 4))
40
41 ax1 = fig.add_axes([0.15, 0.12, 0.33, 0.38])
42
43 ax1.plot(OH_2layer, OH_4layer, marker='o', ls='none', label='OH')
44 ax1.plot(O_2layer, O_4layer, marker='s', ls='none', label='O')
45 ax1.plot(OOH_2layer, OOH_4layer, marker='^', ls='none', label='OOH')
46 ax1.plot((-1, 7), (-1, 7), ls='--', c='k')
47
48 ax1.set_xlabel(r'$\Delta E_{\text{ads}}^{\text{2\layers}}$ (eV)', size='small')
49 ax1.set_ylabel(r'$\Delta E_{\text{ads}}^{\text{4\layers}}$ (eV)', size='small')
50 ax1.set_xticks([0, 2, 4, 6])

```

```

51  ax1.set_yticks([0, 2, 4, 6])
52  ax1.set_xlim(-1, 6.5)
53  ax1.set_ylim(-1, 6.5)
54
55  ax1.text(-0.3, 5.3, 'b')
56
57  # Plot scaling relationships
58
59  # First figure out fits on scaling relationships
60
61  OH = OH_2layer + OH_4layer
62  O = O_2layer + O_4layer
63  OOH = OOH_2layer + OOH_4layer
64
65  OH_O_params = np.polyfit(OH, O, 1)
66  OH_OOH_params = np.polyfit(OH, OOH, 1)
67
68  OH_O_slope = OH_O_params[1]
69  OH_OOH_slope = OH_OOH_params[1]
70
71  OH_O_fit = np.poly1d(OH_O_params)
72  OH_OOH_fit = np.poly1d(OH_OOH_params)
73
74  ax2 = fig.add_axes([0.52, 0.12, 0.33, 0.38])
75
76  ax2.plot(OH_2layer, O_2layer, marker='o',
77          ls='none', c='r', label='O (2 layers)')
78  ax2.plot(OH_2layer, OOH_2layer, marker='o',
79          ls='none', c='g', label='OOH (2 layers)')
80  ax2.plot(OH_4layer, O_4layer, marker='s',
81          ls='none', c='orange', label='O (4 layers)')
82  ax2.plot(OH_4layer, OOH_4layer, marker='s',
83          ls='none', c='greenyellow', label='OOH (4 layers)')
84  ax2.set_xticks([0, 1, 2, 3])
85  ax2.set_yticks([0, 2, 4, 6])
86  ax2.set_yticklabels([])
87  ax2.set_ylim(-1, 6.5)
88
89  ax2.plot((-1, 3.5), OH_O_fit([-1, 3.5]), ls='--', c='r')
90  ax2.plot((-1, 3.5), OH_OOH_fit([-1, 3.5]), ls='--', c='g')
91

```

```

92 ax2.set_xlabel(r'$\Delta E_{ads}^{\sim OH}$ (eV)', size='small')
93
94 ax3 = ax2.twinx()
95 ax3.set_xticks([0, 1, 2, 3])
96 ax3.set_yticks([0, 2, 4, 6])
97 ax3.set_ylim(-1, 6.5)
98
99 ax3.set_ylabel(r'$\Delta E_{ads}$ (eV)', size='small')
100 ax3.text(-0.6, 5.3, 'c')
101
102 # Finally load the images of the structures
103 ax4 = fig.add_axes([0.05, 0.52, 0.9, 0.4], frameon=False)
104 ax4.set_xticks([])
105 ax4.set_yticks([])
106 img = mpimg.imread('supporting-figures/atoms.png')
107 ax4.imshow(img)
108 ax4.text(-50, -15, 'a')
109 plt.savefig('figures/FIG1.png', dpi=300)
110 plt.savefig('figures/FIG1.eps', dpi=300)
111 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  $U=0$ .

---

```

1 import os
2 import glob
3 from espresso import *
4 from ase_addons-surfaces import rutile110
5 from ase.visualize import view
6 from ase.lattice.surface import add_adsorbate
7
8 data = [['MoO2', 4.95, 2.73, 0.28],
9         ['IrO2', 4.54, 3.18, 0.31],
10        ['RuO2', 4.53, 3.12, 0.31],

```

```

11         ['PtO2', 4.59, 3.23, 0.31],
12         ['TiO2', 4.65, 2.97, 0.31],
13         ['RhO2', 4.55, 3.11, 0.31],
14         ['NbO2', 4.94, 2.96, 0.29],
15         ['ReO2', 4.95, 2.68, 0.28],
16         ['MnO2', 4.36, 2.84, 0.30],
17         ['CrO2', 4.38, 2.90, 0.30]]
18
19 mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23, 'bare':24},
20                 'CrO2': {'O':14, 'OH':15, 'OOH':15, 'bare':16}}
21
22 Us = np.linspace(0.0, 8.0, 17)
23 Us[0] = 1e-20
24
25 print '|System|Ads|U|Status|Times ran|'
26 print '|---|'
27
28 for name, a, c, u in data:
29     with Espresso('supporting-data/{name}/Eads-2-layers/bare'.format(**locals())) as calc:
30         atoms = calc.get_atoms()
31         atoms.set_constraint()
32
33         # We now apply the constraints. This is tricky because the H atoms on
34         # the bottom of the slab need to be allowed to relax,
35         # the atoms in the bulk must
36         # fixed x and y, and the adsorbates should be allowed to relax fully
37
38         constraints = []
39         for i, atom in enumerate(atoms):
40             # Bulk atoms relax in z direction
41             if (atom.symbol != 'H'):
42                 constraints.append(FixScaled(atoms.get_cell(), i,
43                                             [True, True, False]))
44             else:
45                 constraints.append(FixScaled(atoms.get_cell(), i,
46                                             [False, False, False]))
47
48         atoms.set_constraint(constraints)
49
50         # Fix magnetic moments to speed convergence
51         if name in mag_elements:

```



```

52     nspin = 2
53     tot_mag = mag_elements[name]['bare']
54 else:
55     nspin = 1
56     tot_mag = None
57
58 for U in Us:
59     # Assign Hubbard U values
60     hubbard_Us = []
61     for atom in atoms:
62         if atom.symbol != 'O' and atom.symbol != 'H':
63             hubbard_Us.append(U)
64         else:
65             hubbard_Us.append(0)
66
67     with Espresso('supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals()),
68                  atoms=atoms,
69                  calculation='relax', disk_io='none',
70                  ecutwfc=40.0, ecutrho=500.0,
71                  occupations='smearing', smearing='mp', degauss=0.01,
72                  kpts=(4, 4, 1), nspin=nspin, tot_magnetization=tot_mag,
73                  lda_plus_u=True, U_projection_type='atomic',
74                  Hubbard_U=hubbard_Us,
75                  nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
76         try:
77             calc.calculate()
78         except (EspressoSubmitted, EspressoRunning):
79             n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '*.o*'))
80             print '|{0}|bare|{1:1.1f}|Running|{2:d}|'.format(name, U, n)
81         except (EspressoNotConverged):
82             # Get the number of times this job has been run...
83             n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '*.o*'))
84             if calc.electronic_converged == False:
85                 print '|{0}|bare|{1:1.1f}|Electronic|{2:d}|'.format(name, U, n)
86                 calc.set(mixing_beta=0.2, Hubbard_U=hubbard_Us)
87                 continue
88             else:
89                 print '|{0}|bare|{1:1.1f}|Structural|{2:d}|'.format(name, U, n)
90             calc.write_input()
91             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.

---

```
1  import glob
2  from espresso import *
3  from ase_addons-surfaces import rutile110
4  from ase.visualize import view
5  from ase.lattice.surface import add_adsorbate
6
7  data = [['MoO2', 4.95, 2.73, 0.28],
8          ['IrO2', 4.54, 3.18, 0.31],
9          ['RuO2', 4.53, 3.12, 0.31],
10         ['PtO2', 4.59, 3.23, 0.31],
11         ['TiO2', 4.65, 2.97, 0.31],
12         ['RhO2', 4.55, 3.11, 0.31],
13         ['NbO2', 4.94, 2.96, 0.29],
14         ['ReO2', 4.95, 2.68, 0.28],
15         ['MnO2', 4.36, 2.84, 0.30],
16         ['CrO2', 4.38, 2.90, 0.30]]
17
18  mag_elements = {'MnO2': {'O':22, 'OH':23, 'OOH':23},
19                 'CrO2': {'O':14, 'OH':15, 'OOH':15}}
20
21  Us = np.linspace(0.0, 8.0, 17)
22  Us[0] = 1e-20
23
24  print '|System|Ads|U|Status|Times ran|'
25  print '|---|'
26
27  for name, a, c, u in data:
28      for ads in ('O', 'OH', 'OOH'):
29          with Espresso('supporting-data/{name}/Eads-2-layers/{ads}-relax-surf'.format(**locals())) as calc:
30              atoms = calc.get_atoms()
31              atoms.set_constraint()
32
33              # We now apply the constraints. This is tricky because the H atoms on
34              # the bottom of the slab need to be allowed to relax,
35              # the atoms in the bulk must
36              # fixed x and y, and the adsorbates should be allowed to relax fully
```

```

37     ads_indexes = range(len(atoms))[30:]
38
39     constraints = []
40     for i, atom in enumerate(atoms):
41         # Bulk atoms relax in z direction
42         if i in ads_indexes:
43             constraints.append(FixScaled(atoms.get_cell(), i,
44                                         [False, False, False]))
45         elif (atom.symbol != 'H'):
46             constraints.append(FixScaled(atoms.get_cell(), i,
47                                         [True, True, False]))
48         else:
49             constraints.append(FixScaled(atoms.get_cell(), i,
50                                         [False, False, False]))
51
52     atoms.set_constraint(constraints)
53
54     # Fix magnetic moments to speed convergence
55     if name in mag_elements:
56         nspin = 2
57         tot_mag = mag_elements[name][ads]
58     else:
59         nspin = 1
60         tot_mag = None
61
62     for U in Us:
63         # Assign Hubbard U values
64         hubbard_Us = []
65         for atom in atoms:
66             if atom.symbol != 'O' and atom.symbol != 'H':
67                 hubbard_Us.append(U)
68             else:
69                 hubbard_Us.append(0)
70
71     with Espresso('supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals()),
72                  atoms=atoms,
73                  calculation='relax', disk_io='none',
74                  ecutwfc=40.0, ecutrho=500.0,
75                  occupations='smearing', smearing='mp', degauss=0.01,
76                  kpts=(4, 4, 1), nspin=nspin, tot_magnetization=tot_mag,
77                  lda_plus_u=True, U_projection_type='atomic',

```

```

78         Hubbard_U=hubbard_Us,
79         nodes=2, ppn=8, processor='xeon8', walltime='48:00:00') as calc:
80     try:
81         calc.calculate()
82     except (EspressoSubmitted, EspressoRunning):
83         n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '*.o*'))
84         print '{0}|{3}|{1:1.1f}|Running|{2:d}|'.format(name, U, n, ads)
85     except (EspressoNotConverged):
86         # Get the number of times this job has been run...
87         n = len(glob.glob1('.', os.path.basename(calc.espressodir) + '*.o*'))
88         if calc.electronic_converged == False:
89             print '{0}|{3}|{1:1.1f}|Electronic|{2:d}|'.format(name, U, n, ads)
90             calc.set(mixing_beta=0.3, Hubbard_U=hubbard_Us)
91         else:
92             print '{0}|{3}|{1:1.1f}|Structural|{2:d}|'.format(name, U, n, ads)
93         calc.write_input()
94         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.

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3  from ase.visualize import view
4  from ase.lattice.surface import add_adsorbate
5  import matplotlib.pyplot as plt
6  from matplotlib.colors import Normalize
7  import numpy as np
8
9  data = [['MoO2', 4.95, 2.73, 0.28],
10         ['IrO2', 4.54, 3.18, 0.31],
11         ['RuO2', 4.53, 3.12, 0.31],
12         ['PtO2', 4.59, 3.23, 0.31],
13         ['TiO2', 4.65, 2.97, 0.31],
14         ['RhO2', 4.55, 3.11, 0.31],
15         ['NbO2', 4.94, 2.96, 0.29],
16         ['ReO2', 4.95, 2.68, 0.28],
17         ['MnO2', 4.36, 2.84, 0.30],
18         ['CrO2', 4.38, 2.90, 0.30]]
19
20  linUs = [['MoO2', 4.83],
21          ['IrO2', 5.91],
22          ['RuO2', 6.73],
23          ['PtO2', 6.25],
24          ['TiO2', 4.95],
25          ['RhO2', 5.97],
26          ['NbO2', 3.32],
27          ['ReO2', 5.27],
28          ['MnO2', 6.63],
29          ['CrO2', 7.15]]
30
31  U_dict = {}
32  for oxide, U in linUs:
33      U_dict[oxide] = U
34
35  def ads_energy(bare, OH, O, OOH):
36      '''The reaction is shown below
37      H2O + * <=> HO* + H + e
38      HO*      <=> O* + H + e
39      O* + H2O <=> HOO* + H + e
40      HOO*     <=> O2 + H + e
41      '''

```

```

42
43     H2 = -31.6933245045 # From H2 in a box
44     H2O = -470.68191439 # From H2O in a box
45
46     try:
47         OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
48     except:
49         OH_ads = None
50
51     try:
52         O_ads = O - bare - (H2O - H2) + 0.05
53     except:
54         O_ads = None
55
56     try:
57         OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
58     except:
59         OOH_ads = None
60
61     return OH_ads, O_ads, OOH_ads
62
63     Us = np.linspace(0.0, 8.0, 17)
64
65     # First get data for two layer slabs
66
67     two_layer_energies = []
68
69     for name, a, c, u in data:
70         O_energies, OH_energies, OOH_energies = [], [], []
71         O_Us, OH_Us, OOH_Us = [], [], []
72         O_dict, OH_dict, OOH_dict = {}, {}, {}
73         for U in Us:
74             # First get the slab energy
75             calcdir = 'supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
76             with Espresso(calcdir) as calc:
77                 if calc.converged == True:
78                     bare = calc.get_potential_energy()
79                 else:
80                     continue
81
82         Eads = {}

```

```

83     for ads in ('OH', 'O', 'OOH'):
84         calcdir = 'supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
85         with Espresso(calcdir) as calc:
86             if calc.converged == True:
87                 Eads[ads] = calc.get_potential_energy()
88             else:
89                 Eads[ads] = None
90
91     energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
92     for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
93         if E_ads is not None:
94             eval(ads + '_energies').append(E_ads)
95             eval(ads + '_Us').append(U)
96             eval(ads + '_dict')[U] = E_ads
97
98     OH_energies_norm = np.array(OH_energies) - OH_energies[0]
99     O_energies_norm = np.array(O_energies) - O_energies[0]
100    OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]
101
102    # First plot the variation of adsorption energies with U
103
104    plt.figure(1, (4.5, 3.5))
105    plt.plot(OH_Us, OH_energies_norm, marker='o', c='b', label='OH')
106    plt.plot(O_Us, O_energies_norm, marker='o', c='r', label='O')
107    plt.plot(OOH_Us, OOH_energies_norm, marker='o', c='g', label='OOH')
108    plt.axvline(U_dict[name], ls='--', c='k')
109
110    plt.title(name + r' $\Delta E_{\text{ads}}(U)$')
111    plt.xlabel('U (eV)')
112    plt.ylabel(r'$\Delta\Delta E_{\text{ads}}$ (eV)')
113    plt.legend(numpoints=1, loc=0, prop={'size': 'small'})
114    plt.tight_layout()
115    plt.savefig('supporting-figures/{name}-EvsU-ads.png'.format(**locals()))
116    plt.show()
117    plt.close()
118
119    print '##CAPTION: EvsU of adsorption energies on {name}'.format(**locals())
120    print '##ATTR_LATEX: :placement [H] :width 3.5in'
121    print '[./supporting-figures/{name}-EvsU-ads.png]\n'.format(**locals())
122
123    # Now plot the scaling relationships with respect to U

```

```

124     E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
125     U_OH_OH, U_OH_O, U_OH_OOH = [], [], []
126
127     for U in Us:
128         if type(OH_dict.get(U)) == float:
129             E_OH_OH.append([OH_dict[U], OH_dict[U]])
130             U_OH_OH.append(U)
131         if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
132             E_OH_O.append([OH_dict[U], O_dict[U]])
133             U_OH_O.append(U)
134         if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
135             E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
136             U_OH_OOH.append(U)
137
138     norm = Normalize(vmin=0, vmax=8)
139
140     plt.figure(1, (4.5, 3.5))
141     x, y = zip(*E_OH_O)
142     plt.scatter(x, y, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
143               marker='s', label='O')
144     x, y = zip(*E_OH_OOH)
145     plt.scatter(x, y, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
146               marker='^', label='OOH')
147
148     plt.legend(loc=6, numpoints=1, prop={'size': 'small'})
149
150     # Also plot the original scaling relationships from with U calculations
151     xs = np.array([min(x), max(x)])
152     plt.plot(xs, 1.54 * xs + 1.17, ls='--', c='r')
153     plt.plot(xs, 0.77 * xs + 3.67, ls='--', c='g')
154
155     plt.xlabel(r'$\Delta E_{\text{ads}}^{\text{OH}}$')
156     plt.ylabel(r'$\Delta E_{\text{ads}}$')
157
158     plt.title(name + r' $\Delta E_{\text{ads}}^{\text{OOH}}(E_{\text{ads}}^{\text{OH}})$, $\Delta E_{\text{ads}}^{\text{O}}(E_{\text{ads}}^{\text{OH}})$')
159
160     plt.tight_layout()
161     plt.savefig('supporting-figures/{name}-EvsU-scaling.png'.format(**locals()))
162     plt.show()
163     plt.close()
164

```



```

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

```

---



Figure S11:  $\Delta E_{ads}(U)$  of adsorption energies on MoO<sub>2</sub>



Figure S12:  $\Delta E_{ads}(U)$  of scaling relationships energies on MoO<sub>2</sub>



Figure S13:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{IrO}_2$



Figure S14:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{IrO}_2$



Figure S15:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{RuO}_2$



Figure S16:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{RuO}_2$



Figure S17:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{PtO}_2$



Figure S18:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{PtO}_2$



Figure S19:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{TiO}_2$



Figure S20:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{TiO}_2$



Figure S21:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{RhO}_2$



Figure S22:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{RhO}_2$



Figure S23:  $\Delta E_{ads}(U)$  of adsorption energies on NbO<sub>2</sub>



Figure S24:  $\Delta E_{ads}(U)$  of scaling relationships energies on NbO<sub>2</sub>



Figure S25:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{ReO}_2$



Figure S26:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{ReO}_2$





Figure S27:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{MnO}_2$



Figure S28:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{MnO}_2$



Figure S29:  $\Delta E_{ads}(U)$  of adsorption energies on  $\text{CrO}_2$



Figure S30:  $\Delta E_{ads}(U)$  of scaling relationships energies on  $\text{CrO}_2$

## 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.

---

```
1 from espresso import *
2 from ase_addons-surfaces import rutile110
3 from ase.visualize import view
4 from ase.lattice.surface import add_adsorbate
5 import matplotlib.pyplot as plt
6 from matplotlib.colors import Normalize
7 import numpy as np
8
9 from matplotlib import rc, rcParams
10
11 rc('xtick', labelsz=10)
12 rc('ytick', labelsz=10)
13
14 U_dict = {'MoO2': 4.83,
15           'IrO2': 5.91,
16           'RuO2': 6.73,
17           'PtO2': 6.25,
18           'TiO2': 4.95,
19           'RhO2': 5.97,
20           'NbO2': 3.32,
21           'ReO2': 5.27,
22           'MnO2': 6.63,
23           'CrO2': 7.15}
24
25 def ads_energy(bare, OH, O, OOH):
26     '''The reaction is shown below
27     H2O + * <=> HO* + H + e
28     HO*      <=> O* + H + e
29     O* + H2O <=> HOO* + H + e
30     HOO*     <=> O2 + H + e
31     '''
32
```

```

33     H2 = -31.6933245045 # From H2 in a box
34     H2O = -470.68191439 # From H2O in a box
35
36     try:
37         OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
38     except:
39         OH_ads = None
40
41     try:
42         O_ads = O - bare - (H2O - H2) + 0.05
43     except:
44         O_ads = None
45
46     try:
47         OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
48     except:
49         OOH_ads = None
50
51     return OH_ads, O_ads, OOH_ads
52
53 Us = np.linspace(0.0, 8.0, 17)
54
55 # First plot the two figures for IrO2
56
57 fig = plt.figure(1, (7, 6))
58
59 O_energies, OH_energies, OOH_energies = [], [], []
60 O_Us, OH_Us, OOH_Us = [], [], []
61 O_dict, OH_dict, OOH_dict = {}, {}, {}
62 for U in Us:
63     # First get the slab energy
64     calcdir = 'supporting-data/IrO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
65     with Espresso(calcdir) as calc:
66         if calc.converged == True:
67             bare = calc.get_potential_energy()
68         else:
69             continue
70
71     Eads = {}
72     for ads in ('OH', 'O', 'OOH'):
73         calcdir = 'supporting-data/IrO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())

```

```

74         with Espresso(calcdir) as calc:
75             if calc.converged == True:
76                 Eads[ads] = calc.get_potential_energy()
77             else:
78                 Eads[ads] = None
79
80     energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
81     for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
82         if E_ads is not None:
83             eval(ads + '_energies').append(E_ads)
84             eval(ads + '_Us').append(U)
85             eval(ads + '_dict')[U] = E_ads
86
87     OH_energies_norm = np.array(OH_energies) - OH_energies[0]
88     O_energies_norm = np.array(O_energies) - O_energies[0]
89     OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]
90
91     # First plot the variation of adsorption energies with U
92
93     ax1 = fig.add_axes([0.1, 0.1, 0.375, 0.375])
94
95     ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
96     ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
97     ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
98     ax1.axvline(5.91, c='k', ls='--')
99     ax1.text(0.3, 0.63, 'c')
100    ax1.set_xlabel('U (eV)')
101    ax1.set_ylabel(r'$\Delta\Delta E_{ads}$ (eV)')
102
103    plt.legend(numpoints=1, loc=6, prop={'size': 'small'})
104
105    # Now plot the scaling relationships with respect to U
106    E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
107    U_OH_OH, U_OH_O, U_OH_OOH = [], [], []
108
109    for U in Us:
110        if type(OH_dict.get(U)) == float:
111            E_OH_OH.append([OH_dict[U], OH_dict[U]])
112            U_OH_OH.append(U)
113        if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
114            E_OH_O.append([OH_dict[U], O_dict[U]])

```

```

115         U_OH_0.append(U)
116         if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
117             E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
118             U_OH_OOH.append(U)
119
120 norm = Normalize(vmin=0, vmax=8)
121
122 ax2 = fig.add_axes([0.575, 0.1, 0.375, 0.375])
123
124 x, y1 = zip(*E_OH_0)
125 ax2.scatter(x, y1, c=U_OH_0, s=64, cmap=plt.get_cmap('jet'), norm=norm,
126             marker='d', label='O')
127 x, y2 = zip(*E_OH_OOH)
128 ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
129             marker='v', label='OOH')
130
131 ax2.legend(loc=6, numpoints=1, prop={'size': 'small'})
132
133 # Also plot the original scaling relationships from with U calculations. We
134 # want to offset these so it starts
135 xs = np.array([min(x) - 0.1, max(x) + 0.1])
136 ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
137 ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
138 ax2.text(0.07, 4.15, 'd')
139 ax2.set_xlim(0.05, 0.55)
140 ax2.set_xlabel(r'$\Delta E_{\text{ads}}^{\text{OH}}$')
141 ax2.set_ylabel(r'$\Delta E_{\text{ads}}$')
142
143 # Plot the color legend
144 gradient = np.linspace(0, 1, 256)
145 gradient = np.vstack((gradient, gradient))
146 ax3 = fig.add_axes((0.8, 0.31, 0.1, 0.02))
147 ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
148 ax3.set_yticks([], [])
149 ax3.set_xticks((0, 256))
150 ax3.set_xticklabels((0, 10))
151 fig.text(0.845, 0.34, 'U', size=10, style='italic')
152
153 # Now plot the two figures for MoO2
154
155 fig = plt.figure(1, (7, 6))

```

```

156
157 O_energies, OH_energies, OOH_energies = [], [], []
158 O_Us, OH_Us, OOH_Us = [], [], []
159 O_dict, OH_dict, OOH_dict = {}, {}, {}
160 for U in Us:
161     # First get the slab energy
162     calcdir = 'supporting-data/NbO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
163     with Espresso(calcdir) as calc:
164         if calc.converged == True:
165             bare = calc.get_potential_energy()
166         else:
167             continue
168
169     Eads = {}
170     for ads in ('OH', 'O', 'OOH'):
171         calcdir = 'supporting-data/NbO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
172         with Espresso(calcdir) as calc:
173             if calc.converged == True:
174                 Eads[ads] = calc.get_potential_energy()
175             else:
176                 Eads[ads] = None
177
178     energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
179     for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
180         if E_ads is not None:
181             eval(ads + '_energies').append(E_ads)
182             eval(ads + '_Us').append(U)
183             eval(ads + '_dict')[U] = E_ads
184
185     OH_energies_norm = np.array(OH_energies) - OH_energies[0]
186     O_energies_norm = np.array(O_energies) - O_energies[0]
187     OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]
188
189     # First plot the variation of adsorption energies with U
190
191     ax1 = fig.add_axes([0.1, 0.575, 0.375, 0.375])
192
193     ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
194     ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
195     ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
196     ax1.axvline(3.32, c='k', ls='--')

```

```

197 ax1.text(0.3, 0.72, 'a')
198 ax1.set_xlabel('U (eV)')
199 ax1.set_ylabel(r'$\Delta\Delta E_{\text{ads}}$ (eV)')
200 plt.legend(numpoints=1, loc=6, prop={'size': 'small'})
201
202 # Now plot the scaling relationships with respect to U
203 E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
204 U_OH_OH, U_OH_O, U_OH_OOH = [], [], []
205
206 for U in Us:
207     if type(OH_dict.get(U)) == float:
208         E_OH_OH.append([OH_dict[U], OH_dict[U]])
209         U_OH_OH.append(U)
210     if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
211         E_OH_O.append([OH_dict[U], O_dict[U]])
212         U_OH_O.append(U)
213     if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
214         E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
215         U_OH_OOH.append(U)
216
217 norm = Normalize(vmin=0, vmax=8)
218
219 ax2 = fig.add_axes([0.575, 0.575, 0.375, 0.375])
220
221 x, y1 = zip(*E_OH_O)
222 ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
223            marker='d', label='O')
224 x, y2 = zip(*E_OH_OOH)
225 ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
226            marker='v', label='OOH')
227
228 ax2.legend(loc=6, numpoints=1, prop={'size': 'small'})
229
230 # Also plot the original scaling relationships from with U calculations. We want to
231 # offset these so it starts
232 xs = np.array([min(x) - 0.05, max(x) + 0.05])
233 ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
234 ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
235 ax2.text(-0.71, 3.4, 'b')
236 ax2.set_xlim(-0.725, -0.4)
237 ax2.set_xticks([-0.7, -0.6, -0.5, -0.4])

```



```

238 ax2.set_xlabel(r'$\Delta E_{\text{ads}}^{\text{OH}}$')
239 ax2.set_ylabel(r'$\Delta E_{\text{ads}}$')
240
241 # Plot the color legend
242 gradient = np.linspace(0, 1, 256)
243 gradient = np.vstack((gradient, gradient))
244 ax3 = fig.add_axes((0.8, 0.8, 0.1, 0.02))
245 ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
246 ax3.set_yticks([], [])
247 ax3.set_xticks((0, 256))
248 ax3.set_xticklabels((0, 10))
249 fig.text(0.845, 0.83, 'U', size=10, style='italic')
250 fig.savefig('figures/FIG2.png', dpi=300)
251 fig.savefig('figures/FIG2.eps', dpi=300)
252 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  $\text{TiO}_2$  and  $\text{MnO}_2/\text{CrO}_2$ . This is Figure 3 in the manuscript.

---

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3  from ase.visualize import view
4  from ase.lattice.surface import add_adsorbate
5  import matplotlib.pyplot as plt
6  from matplotlib.colors import Normalize
7  import numpy as np
8
9  from matplotlib import rc, rcParams
10
11  rc('xtick', labels=10)
12  rc('ytick', labels=10)
13
14  U_dict = {'MoO2': 4.83,
15           'IrO2': 5.91,
16           'RuO2': 6.73,
17           'PtO2': 6.25,

```

```

18         'TiO2': 4.95,
19         'RhO2': 5.97,
20         'NbO2': 3.32,
21         'ReO2': 5.27,
22         'MnO2': 6.63,
23         'CrO2': 7.15}
24
25 def ads_energy(bare, OH, O, OOH):
26     '''The reaction is shown below
27     H2O + * <=> HO* + H + e
28     HO*      <=> O* + H + e
29     O* + H2O <=> HOO* + H + e
30     HOO*     <=> O2 + H + e
31     '''
32
33     H2 = -31.6933245045 # From H2 in a box
34     H2O = -470.68191439 # From H2O in a box
35
36     try:
37         OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
38     except:
39         OH_ads = None
40
41     try:
42         O_ads = O - bare - (H2O - H2) + 0.05
43     except:
44         O_ads = None
45
46     try:
47         OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
48     except:
49         OOH_ads = None
50
51     return OH_ads, O_ads, OOH_ads
52
53 Us = np.linspace(0.0, 8.0, 17)
54
55 # First plot the two figures for TiO2
56
57 fig = plt.figure(1, (7, 6))
58

```

```

59  O_energies, OH_energies, OOH_energies = [], [], []
60  O_Us, OH_Us, OOH_Us = [], [], []
61  O_dict, OH_dict, OOH_dict = {}, {}, {}
62  for U in Us:
63      # First get the slab energy
64      calcdir = 'supporting-data/TiO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
65      with Espresso(calcdir) as calc:
66          if calc.converged == True:
67              bare = calc.get_potential_energy()
68          else:
69              continue
70
71  Eads = {}
72  for ads in ('OH', 'O', 'OOH'):
73      calcdir = 'supporting-data/TiO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
74      with Espresso(calcdir) as calc:
75          if calc.converged == True:
76              Eads[ads] = calc.get_potential_energy()
77          else:
78              Eads[ads] = None
79
80  energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
81  for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
82      if E_ads is not None:
83          eval(ads + '_energies').append(E_ads)
84          eval(ads + '_Us').append(U)
85          eval(ads + '_dict')[U] = E_ads
86
87  OH_energies_norm = np.array(OH_energies) - OH_energies[0]
88  O_energies_norm = np.array(O_energies) - O_energies[0]
89  OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]
90
91  # First plot the variation of adsorption energies with U
92
93  ax1 = fig.add_axes([0.1, 0.1, 0.375, 0.375])
94
95  ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
96  ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
97  ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
98  ax1.axvline(U_dict['TiO2'], c='k', ls='--')
99  ax1.set_ylim(-0.15, 0.35)

```

```

100 ax1.text(7.2, 0.3, 'c')
101 ax1.set_xlabel('U (eV)')
102 ax1.set_ylabel(r'$\Delta\Delta E_{ads}$ (eV)$')
103
104 plt.legend(numpoints=1, loc=2, prop={'size': 'small'})
105
106 # Now plot the scaling relationships with respect to U
107 E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
108 U_OH_OH, U_OH_O, U_OH_OOH = [], [], []
109
110 for U in Us:
111     if type(OH_dict.get(U)) == float:
112         E_OH_OH.append([OH_dict[U], OH_dict[U]])
113         U_OH_OH.append(U)
114     if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
115         E_OH_O.append([OH_dict[U], O_dict[U]])
116         U_OH_O.append(U)
117     if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
118         E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
119         U_OH_OOH.append(U)
120
121 norm = Normalize(vmin=0, vmax=8)
122
123 ax2 = fig.add_axes([0.575, 0.1, 0.375, 0.375])
124
125 x, y1 = zip(*E_OH_O)
126 ax2.scatter(x, y1, c=U_OH_O, s=64, cmap=plt.get_cmap('jet'), norm=norm,
127             marker='d', label='O')
128 x, y2 = zip(*E_OH_OOH)
129 ax2.scatter(x, y2, c=U_OH_OOH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
130             marker='v', label='OOH')
131
132 ax2.legend(loc=6, numpoints=1, prop={'size': 'small'})
133
134 # Also plot the original scaling relationships from with U calculations. We want to
135 # offset these so it starts
136 xs = np.array([1.49, 1.61])
137 ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
138 ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
139 ax2.text(1.495, 5.24, 'd')
140 ax2.set_xlim(1.49, 1.61)

```

```

141 ax2.set_xlabel(r'$\Delta E_{\text{ads}}^{\text{OH}}$')
142 ax2.set_ylabel(r'$\Delta E_{\text{ads}}$')
143
144 # Plot the color legend
145 gradient = np.linspace(0, 1, 256)
146 gradient = np.vstack((gradient, gradient))
147 ax3 = fig.add_axes((0.8, 0.28, 0.1, 0.02))
148 ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
149 ax3.set_yticks([], [])
150 ax3.set_xticks((0, 256))
151 ax3.set_xticklabels((0, 10))
152 fig.text(0.845, 0.31, 'U', size=10, style='italic')
153
154 # Now plot the two figures for MnO2
155
156 fig = plt.figure(1, (7, 6))
157
158 O_energies, OH_energies, OOH_energies = [], [], []
159 O_Us, OH_Us, OOH_Us = [], [], []
160 O_dict, OH_dict, OOH_dict = {}, {}, {}
161 for U in Us:
162     # First get the slab energy
163     calcdir = 'supporting-data/MnO2/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
164     with Espresso(calcdir) as calc:
165         if calc.converged == True:
166             bare = calc.get_potential_energy()
167         else:
168             continue
169
170     Eads = {}
171     for ads in ('OH', 'O', 'OOH'):
172         calcdir = 'supporting-data/MnO2/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
173         with Espresso(calcdir) as calc:
174             if calc.converged == True:
175                 Eads[ads] = calc.get_potential_energy()
176             else:
177                 Eads[ads] = None
178
179     energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
180     for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
181         if E_ads is not None:

```

```

182         eval(ads + '_energies').append(E_ads)
183         eval(ads + '_Us').append(U)
184         eval(ads + '_dict')[U] = E_ads
185
186     OH_energies_norm = np.array(OH_energies) - OH_energies[0]
187     O_energies_norm = np.array(O_energies) - O_energies[0]
188     OOH_energies_norm = np.array(OOH_energies) - OOH_energies[0]
189
190     # First plot the variation of adsorption energies with U
191
192     ax1 = fig.add_axes([0.1, 0.575, 0.375, 0.375])
193
194     ax1.plot(OH_Us, OH_energies_norm, marker='o', c='c', label='OH')
195     ax1.plot(O_Us, O_energies_norm, marker='s', c='m', label='O')
196     ax1.plot(OOH_Us, OOH_energies_norm, marker='^', c='g', label='OOH')
197     ax1.axvline(U_dict['MnO2'], c='k', ls='--')
198     ax1.set_ylim(0, 3.0)
199     ax1.text(7.2, 2.7, 'a')
200     ax1.set_xlabel('U (eV)')
201     ax1.set_ylabel(r'$\Delta\Delta E_{ads}$ (eV)')
202     plt.legend(numpoints=1, loc=2, prop={'size': 'small'})
203
204     # Now plot the scaling relationships with respect to U
205     E_OH_OH, E_OH_O, E_OH_OOH = [], [], []
206     U_OH_OH, U_OH_O, U_OH_OOH = [], [], []
207
208     for U in Us:
209         if type(OH_dict.get(U)) == float:
210             E_OH_OH.append([OH_dict[U], OH_dict[U]])
211             U_OH_OH.append(U)
212         if type(OH_dict.get(U)) == float and type(O_dict.get(U)) == float:
213             E_OH_O.append([OH_dict[U], O_dict[U]])
214             U_OH_O.append(U)
215         if type(OH_dict.get(U)) == float and type(OOH_dict.get(U)) == float:
216             E_OH_OOH.append([OH_dict[U], OOH_dict[U]])
217             U_OH_OOH.append(U)
218
219     norm = Normalize(vmin=0, vmax=8)
220
221     ax2 = fig.add_axes([0.575, 0.575, 0.375, 0.375])
222

```

```

223 x, y1 = zip(*E_OH_0)
224 ax2.scatter(x, y1, c=U_OH_0, s=64, cmap=plt.get_cmap('jet'), norm=norm,
225             marker='d', label='0')
226 x, y2 = zip(*E_OH_0OH)
227 ax2.scatter(x, y2, c=U_OH_0OH, s=64, cmap=plt.get_cmap('jet'), norm=norm,
228             marker='v', label='0OH')
229
230 ax2.legend(loc=6, numpoints=1, prop={'size': 'small'})
231
232 # Also plot the original scaling relationships from with U calculations. We want to
233 # offset these so it starts
234 xs = np.array([1.25, 3.25])
235 ax2.plot(xs, 1.54 * (xs - x[0]) + y1[0], c='r')
236 ax2.plot(xs, 0.77 * (xs - x[0]) + y2[0], c='g')
237 ax2.text(1.35, 6.1, 'b')
238 ax2.set_xlim(1.25, 3.25)
239 ax2.set_xlabel(r'$\Delta E_{\text{ads}}^{\text{OH}}$')
240 ax2.set_ylabel(r'$\Delta E_{\text{ads}}$')
241
242 # Plot the color legend
243 gradient = np.linspace(0, 1, 256)
244 gradient = np.vstack((gradient, gradient))
245 ax3 = fig.add_axes((0.8, 0.7, 0.1, 0.02))
246 ax3.imshow(gradient, aspect='auto', cmap=plt.get_cmap('jet'))
247 ax3.set_yticks([], [])
248 ax3.set_xticks((0, 256))
249 ax3.set_xticklabels((0, 10))
250 fig.text(0.845, 0.73, 'U', size=10, style='italic')
251 fig.savefig('figures/FIG3.png', dpi=300)
252 fig.savefig('figures/FIG3.eps', dpi=300)
253 plt.show()

```

---

## 7 Activity trends with DFT+ $U$

This section looks at the oxygen evolution activity of  $\text{IrO}_2$ ,  $\text{RuO}_2$ ,  $\text{RhO}_2$ , and  $\text{PtO}_2$  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.<sup>?</sup> 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.

---

```

1  from espresso import *
2  from ase_addons-surfaces import rutile110
3  from ase.visualize import view
4  from ase.lattice.surface import add_adsorbate
5  import matplotlib.pyplot as plt
6  from matplotlib.colors import Normalize
7  from scipy.interpolate import interp1d
8  import numpy as np
9
10 data = [['IrO2', 4.54, 3.18, 0.31],
11         ['RuO2', 4.53, 3.12, 0.31],
12         ['PtO2', 4.59, 3.23, 0.31],
13         ['RhO2', 4.55, 3.11, 0.31]]
14
15 linUs = [['IrO2', 5.91],
16          ['RuO2', 6.73],
```



```

17         ['PtO2', 6.25],
18         ['RhO2', 5.97]]
19
20     U_dict = {}
21     for oxide, U in linUs:
22         U_dict[oxide] = U
23
24     def ads_energy(bare, OH, O, OOH):
25         '''The reaction is shown below
26         H2O + * <=> HO* + H + e
27         HO*      <=> O* + H + e
28         O* + H2O <=> HOO* + H + e
29         HOO*     <=> O2 + H + e
30         '''
31
32         H2 = -31.6933245045 # From H2 in a box
33         H2O = -470.68191439 # From H2O in a box
34
35         try:
36             OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
37         except:
38             OH_ads = None
39
40         try:
41             O_ads = O - bare - (H2O - H2) + 0.05
42         except:
43             O_ads = None
44
45         try:
46             OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4
47         except:
48             OOH_ads = None
49
50         return OH_ads, O_ads, OOH_ads
51
52     Us = np.linspace(0.0, 8.0, 17)
53
54     table_header = ' |Name|$\Delta G_{1}^{U0}$|$\Delta G_{2}^{U0}$|$\Delta G_{3}^{U0}$|$\Delta G_{4}^{U0}$| '
55     table_header += '$\Delta G_{1}^{Ucalc}$|$\Delta G_{2}^{Ucalc}$|$\Delta G_{3}^{Ucalc}$|$\Delta G_{4}^{Ucalc}$| '
56
57     print '#+CAPTION: Reaction energies at both U=0 the calculated linear response U value'

```

```

58 print '#+ATTR_LATEX: :placement [H] :align |c|c|c|c|c|c|c|c|c|'
59 print '#+TBLNAME: rxn-energies'
60 print '|---|'
61
62 for name, a, c, u in data:
63     O_energies, OH_energies, OOH_energies = [], [], []
64     O_Us, OH_Us, OOH_Us = [], [], []
65     for U in Us:
66         # First get the slab energy
67         calcdir = 'supporting-data/{name}/Eads-2-layers/bare-U-{U:1.1f}'.format(**locals())
68         with Espresso(calcdir) as calc:
69             if calc.converged == True:
70                 bare = calc.get_potential_energy()
71             else:
72                 continue
73
74     Eads = {}
75     for ads in ('OH', 'O', 'OOH'):
76         calcdir = 'supporting-data/{name}/Eads-2-layers/{ads}-U-{U:1.1f}'.format(**locals())
77         with Espresso(calcdir) as calc:
78             if calc.converged == True:
79                 Eads[ads] = calc.get_potential_energy()
80             else:
81                 Eads[ads] = None
82
83     energies = ads_energy(bare, Eads['OH'], Eads['O'], Eads['OOH'])
84     for ads, E_ads in zip(('OH', 'O', 'OOH'), energies):
85         if E_ads is not None:
86             eval(ads + '_energies').append(E_ads)
87             eval(ads + '_Us').append(U)
88
89     # After the data is collected, read the adsorption energy data at U0 and Ucalc
90     O_func = interp1d(O_Us, O_energies, kind='linear')
91     OH_func = interp1d(OH_Us, OH_energies, kind='linear')
92     OOH_func = interp1d(OOH_Us, OOH_energies, kind='linear')
93
94     E_O_U0 = O_func(0)
95     E_OH_U0 = OH_func(0)
96     E_OOH_U0 = OOH_func(0)
97
98     E_O_Ucalc = O_func(U_dict[name])

```

```

99     E_OH_Ucalc = OH_func(U_dict[name])
100     E_OOH_Ucalc = OOH_func(U_dict[name])
101
102     # Now print out the reaction barriers
103     r1_U0 = E_OH_U0
104     r2_U0 = E_O_U0 - E_OH_U0
105     r3_U0 = E_OOH_U0 - E_O_U0
106     r4_U0 = 4.92 - E_OOH_U0
107
108     r1_Ucalc = E_OH_Ucalc
109     r2_Ucalc = E_O_Ucalc - E_OH_Ucalc
110     r3_Ucalc = E_OOH_Ucalc - E_O_Ucalc
111     r4_Ucalc = 4.92 - E_OOH_Ucalc
112
113     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}|'
114
115     print s.format(name, float(r1_U0), float(r2_U0), float(r3_U0), float(r4_U0),
116                     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	$\Delta G_1^{U0}$	$\Delta G_2^{U0}$	$\Delta G_3^{U0}$	$\Delta G_4^{U0}$	$\Delta G_1^{Ucalc}$	$\Delta G_2^{Ucalc}$	$\Delta G_3^{Ucalc}$	$\Delta G_4^{Ucalc}$
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.<sup>?</sup> This is given as

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

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # The reaction energy at different reaction steps using both DFT and DFT+U. The organization is as follows

```

```

5  #      Name , G1_U0, G2_U0, G3_U0, G4_U0, G1_Ucalc, G2_Ucalc, G3_Ucalc, G4_Ucalc
6  data = [['IrO2', 0.089, 1.392, 1.983, 1.456, 0.410, 1.539, 1.751, 1.219],
7          ['RuO2', 0.569, 1.102, 2.336, 0.912, 0.824, 1.207, 2.181, 0.708],
8          ['PtO2', 0.964, 2.197, 1.196, 0.563, 1.115, 2.393, 0.922, 0.490],
9          ['RhO2', 0.989, 1.821, 1.489, 0.621, 1.289, 2.015, 1.191, 0.425]]
10
11  fig = plt.figure(1,(3.25,3.5))
12  ax = fig.add_subplot(111)
13
14  ax.plot((0 - 0.5, 1.6), (1.97 + 0.5, 0.37), color='k')
15  ax.plot((1.6, 3.2 + 0.5), (0.37, 1.97 + 0.5), color='k')
16
17  for name, G1_U0, G2_U0, G3_U0, G4_U0, G1_Ucalc, G2_Ucalc, G3_Ucalc, G4_Ucalc in data:
18      eta_U0 = max([G1_U0, G2_U0, G3_U0, G4_U0]) - 1.23
19      eta_Ucalc = max([G1_Ucalc, G2_Ucalc, G3_Ucalc, G4_Ucalc]) - 1.23
20      p1, = plt.plot(G2_U0, eta_U0, marker='o', c='c', ms=8, ls='none')
21      p2, = plt.plot(G2_Ucalc, eta_Ucalc, marker='s', c='r', ms=8, ls='none')
22      plt.annotate('', xytext=(G2_U0, eta_U0), xy=(G2_Ucalc, eta_Ucalc),
23                  arrowprops=dict(arrowstyle='simple', color='k'))
24
25  plt.text(0.55, 1.0, r'$\mathdefault{RuO_{2}}$')
26  plt.text(1.0, 0.6, r'$\mathdefault{IrO_{2}}$')
27  plt.text(2.0, 0.66, r'$\mathdefault{RhO_{2}}$')
28  plt.text(2.3, 1.05, r'$\mathdefault{PtO_{2}}$')
29
30  ax.set_ylim(1.3, 0)
31  ax.set_xlim(0.5, 2.7)
32  ax.set_ylabel(r'$\eta^{\text{OER}}$ (V)')
33  ax.set_xlabel(r'$\Delta G_{0*} - \Delta G_{\text{OH}*}$ (eV)')
34  plt.legend([p1, p2], ['DFT', r'DFT+$\mathdefault{U_{\text{calc}}}$'], numpoints=1, prop={'size':'medium'})
35  plt.tight_layout()
36  plt.savefig('figures/FIG4.png', dpi=300)
37  plt.savefig('figures/FIG4.eps', dpi=300)
38  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](https://github.com/zhongnanxu/ase_addons).

## References

.