Supporting Information for "Tuning Oxide Activity through Modification of the Crystal and Electronic Structure: From Strain to Potential Polymorphs"

Zhongnan Xu and John R. Kitchin*

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

E-mail: jkitchin@andrew.cmu.edu

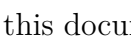
Contents

*To whom correspondence should be addressed

1 Introduction

This supporting information contains the code required to reproduce all of the work in this manuscript. In Section 2, we obtain relaxed bulk and surface unit cells of all structures. We then use these relaxed structures for calculation of adsorption energies in Section 3 and vacancy formation energies in Section 4. Section 5 contains a single script that re-calculates the electronic structure of certain surface structures to produce correct t_{2g} and e_g states. In Section 6, the total energies and density of states are then extracted from these final

calculations and stored into a single file **data.json**, which we will use in our analysis and construction of Figures 2, 3 and 4 in the manuscript. When running analysis scripts in Section 7, make sure scripts are run in the same folder that contains the **data.json** file.

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.[1]

Finally, the **data.json** file that contains all of the data required for construction of Figures 2-4 can be found here:  .

2 Obtaining relaxed bulk and surface cells

In this section, we present code for obtaining relaxed bulk primitive cells and surface slabs. From these structures, we will perform adsorption and vacancy formation energies. For construction of the surface slabs, we first make a new unit cell so that lattice vectors a_1 and a_2 form the desired plane of the surface. a_3 is then chosen to be as normal to the plane formed by a_1 and a_2 as possible but also minimizing the thickness of the slab required for a symmetric cell. This method was developed based on a recent paper for the efficient construction of surface slabs.[2]

To obtain a good starting guess of the atomic structure of the surface slab, we first calculate an equilibrium bulk structure of all polymorphs with the newly constructed unit cell. Once this bulk structure is calculated, we "cut" the cell in half along the plane formed by the a_1 and a_2 cell vectors at the desired termination that exposes a five coordinated metal site and is stoichiometric. We then extend the cell in the a_3 direction so there is 10 Å of vacuum.

2.1 Equilibrium bulk structures in a surface unit cell used for adsorption calculations

The three sections below each contain calculations required to obtain a relaxed, ground state structure of each metastable polymorph (anatase, brookite, columbite, pyrite) of RuO_2, RhO_2, IrO_2, and PtO_2. The unit cell used is not the primitive cell, but the cell used for surface calculations. The initial guess for each system is its equilibrium volume in the rutile phase, which shown in the table below. The first script calculates a "coarse" EOS, which are relaxed structures at 80%, 90%, 100% 110% and 120% volume relative to the equilibrium rutile volume, which is shown in the table below. Note for the anatase structure, we found it necessary to calculate volumes at 130%, 140% and 150% relative to the rutile equilibrium volume to capture the EOS minimum. The equilibrium volume is taken from a 3rd order polynomial fit and then used as a first guess for a "fine" EOS.

Rutile	Volume (Å/atom)
IrO_2	10.972
RhO_2	10.813
RuO_2	10.802
PtO_2	11.438

The second script computes a "fine" EOS calculation, which is made up of fixed-volume ground state structures at 90%, 95%, 100%, 105% and 110% with respect to the guess from the "coarse" EOS. The scaled coordinates are also taken from a fully relaxed structure from the "coarse" calculation.

The final script takes the equilibrium volume and scaled coordinates and computes a final relaxation that allows the ion positions, cell shape, and cell volume change to ensure a completely relaxed bulk cell.

2.1.1 Coarse EOS of metastable polymorphs bulk unit cells

The script below calculates an equation of state centered at the rutile equilibrium volume. The volumes calculated are at 80%, 90%, 100% 110% and 120% relative to the rutile equilibrium volume. Because anatase typically has a higher volume, we extended the equation of state to include volumes that are 130%, 140%, and 150% relative to the equilibrium rutile volume. After calculating the equation of state and finding a first guess for the equilibrium volume of the polymorph, we also print out the fraction $\frac{V_{eq}^{poly}}{V_{eq}^{rutile}}$. This value will be used for the calculation of the fine EOS in the upcoming section.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.utils.eos import EquationOfState

data = [['IrO2', 10.972 ],
        ['RhO2', 10.813 ],
        ['RuO2', 10.802 ],
        ['PtO2', 11.438 ]]

factors = [0.8, 0.9, 1.0, 1.1, 1.2]
anatase_factors = factors + [1.3, 1.4, 1.5]

print '|System|Polymorph|Vol|Vol frac|'
print '|---|'

structs = [['anatase', anatase001],
           ['brookite', brookite110],
           ['columbite', columbite101],
           ['pyrite', pyrite001]]

for name, vol in data:
    for struct, func in structs:
        m = name[:2]
        energies, vols = [], []
        ready = True
```

```
29
30          if struct == 'anatase':
31              factors = [0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
32          else:
33              factors = [0.8, 0.9, 1.0, 1.1, 1.2]
34
35          for f in factors:
36              atoms = func(B=m, X='O', vacuum=0, fixlayers=0, mags=[0, 0])
37              atoms.set_volume(f * vol * len(atoms))
38              with jasp('{name}/{struct}/EOS/v-{f:1.1f}'.format(**locals()), atoms=atoms,
39                        xc='PBE', lreal=False,
40                        encut=500, kpts=(7, 7, 7),
41                        ispin=1, lorbit=11,
42                        ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
43                        npar=4, nsim=4, lplane=True, lscalu=False,
44                        lwave=False) as calc:
45                  try:
46                      energies.append(atoms.get_potential_energy())
47                      vols.append(atoms.get_volume())
48                      print calc.vaspdir, 'Converged'
49                  except (VaspSubmitted, VaspRunning, VaspQueued):
50                      print calc.vaspdir, 'Running'
51
52          # We only want to compute the EOS if we have at least 3 converged volumes
53          if len(vols) > 2:
54              eos = EquationOfState(vols, energies)
55              v0, e0, B = eos.fit()
56              eos.plot('images/{name}-{struct}-EOS.png'.format(**locals()), show=True)
57              print '|{0}|{1}|{2:1.3f}|{3:1.2f}|'.format(name, struct,
58                                                         v0 / len(atoms),
59                                                         v0 / vol / len(atoms))
```

Table 1: Equilibrium volumes from coarse guess of structures in the rutile phase.

System	Polymorph	V_{eq} (Å^3/atom)	$\frac{V_{eq}^{struct}}{V_{eq}^{rutile}}$
IrO_2	anatase	12.624	1.2
IrO_2	columbite	11.000	1.0
IrO_2	pyrite	10.023	0.9
IrO_2	brookite	11.881	1.1
RhO_2	anatase	12.444	1.2
RhO_2	columbite	10.758	1.0
RhO_2	pyrite	9.833	0.9
RhO_2	brookite	11.639	1.1
RuO_2	anatase	12.441	1.2
RuO_2	columbite	10.738	1.0
RuO_2	pyrite	9.806	0.9
RuO_2	brookite	11.516	1.1
PtO_2	anatase	13.414	1.2
PtO_2	columbite	11.214	1.0
PtO_2	pyrite	10.437	0.9
PtO_2	brookite	12.010	1.1

2.1.2 Fine EOS of metastable polymorphs bulk unit cells

The script below calculates an equation of state centered at the equilibrium volume obtained from the coarse EOS. It also starts the scaled coordinates using the relaxed scaled coordinates a structure from the previous calculation that is closest to the calculated equilibrium volume. The volumes calculated are at 90%, 95%, 100% 105% and 110% relative to the equilibrium volume obtained from the coarse EOS.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.utils.eos import EquationOfState

data = [['IrO2', 'anatase', 12.624, 1.2],
        ['IrO2', 'columbite', 11.000, 1.0],
        ['IrO2', 'pyrite', 10.023, 0.9],
        ['IrO2', 'brookite', 11.881, 1.1],
```

```
        ['RhO2', 'anatase', 12.444, 1.2],
        ['RhO2', 'columbite', 10.758, 1.0],
        ['RhO2', 'pyrite',  9.833, 0.9],
        ['RhO2', 'brookite', 11.639, 1.1],
        ['RuO2', 'anatase', 12.441, 1.2],
        ['RuO2', 'columbite', 10.738, 1.0],
        ['RuO2', 'pyrite',  9.806, 0.9],
        ['RuO2', 'brookite', 11.516, 1.1],
        ['PtO2', 'anatase', 13.414, 1.2],
        ['PtO2', 'columbite', 11.214, 1.0],
        ['PtO2', 'pyrite', 10.437, 0.9],
        ['PtO2', 'brookite', 12.010, 1.1]]

factors = [0.9, 0.95, 1.0, 1.05, 1.1]

print '|System|Polymorph|Vol|Vol frac|'
print '|---|'

for name, struct, vol, frac in data:

    # First load up the structure of a previuosly converged bulk calculation from
    # the coarse EOS.
    with jasp('{name}/{struct}/EOS/v-{frac:1.1f}'.format(**locals())) as calc:
        atoms_original = calc.get_atoms()

    energies, vols = [], []
    ready = True
    for f in factors:
        atoms = atoms_original.copy()
        atoms.set_volume(f * vol * len(atoms))
        with jasp('{name}/{struct}/EOS-fine/v-{f:1.2f}'.format(**locals()),
                  atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 7),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=4, nsw=50, ediffg=-0.05, ediff=1e-6,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                energies.append(atoms.get_potential_energy())
                vols.append(atoms.get_volume())
```

```
53                 print calc.vaspdir, 'Converged'
54             except (VaspSubmitted, VaspRunning, VaspQueued):
55                 print calc.vaspdir, 'running'
56                 ready = False
57
58     if len(vols) > 2:
59         eos = EquationOfState(vols, energies)
60         v0, e0, B = eos.fit()
61         eos.plot('images/{name}-{struct}-EOS-fine.png'.format(**locals()),
62                  show=False)
63         print '|{0}|{3}|{1:1.3f}|{2:1.2f}|'.format(name,
64                                                   v0 / len(atoms),
65                                                   v0 / vol / len(atoms),
66                                                   struct)
```

Table 2: Equilibrium volumes from fine relaxation.

System	Polymorph	Vol V_{eq} (Å^3/atom)
IrO_2	anatase	12.598
IrO_2	columbite	10.891
IrO_2	pyrite	10.023
IrO_2	brookite	11.736
RhO_2	anatase	12.417
RhO_2	columbite	10.650
RhO_2	pyrite	9.833
RhO_2	brookite	11.500
RuO_2	anatase	12.451
RuO_2	columbite	10.691
RuO_2	pyrite	9.805
RuO_2	brookite	11.506
PtO_2	anatase	13.368
PtO_2	columbite	11.192
PtO_2	pyrite	10.380
PtO_2	brookite	11.969

2.1.3 Full volume, shape, and ionic relaxation of metastable polymorphs bulk unit cells

This final script takes the equilibrium volume and scaled position from the "fine" EOS calculation and performs a full relaxation. The volume, cell shape, and ionic positions are

now allowed to relax. The finished structural information is saved and used for future surface and adsorption energy calculations.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.utils.eos import EquationOfState

data = [['IrO2', 'anatase  ', 12.598],
        ['IrO2', 'columbite', 10.891],
        ['IrO2', 'pyrite   ', 10.023],
        ['IrO2', 'brookite ', 11.736],
        ['RhO2', 'anatase  ', 12.417],
        ['RhO2', 'columbite', 10.650],
        ['RhO2', 'pyrite   ',  9.833],
        ['RhO2', 'brookite ', 11.500],
        ['RuO2', 'anatase  ', 12.451],
        ['RuO2', 'columbite', 10.691],
        ['RuO2', 'pyrite   ',  9.805],
        ['RuO2', 'brookite ', 11.506],
        ['PtO2', 'anatase  ', 13.368],
        ['PtO2', 'columbite', 11.192],
        ['PtO2', 'pyrite   ', 10.380],
        ['PtO2', 'brookite ', 11.969]]

for name, struct, vol in data:
    with jasp('{name}/{struct}/EOS-fine/v-1.00'.format(**locals())) as calc:
        atoms = calc.get_atoms()

    atoms.set_volume(vol * len(atoms))
    with jasp('{name}/{struct}/ground'.format(**locals()), atoms=atoms,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 7),
              ispin=1, lorbit=11,
              ibrion=2, isif=3, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            calc.calculate()
```

```
            print calc.vaspdir, 'Converged'
        except (VaspSubmitted, VaspRunning, VaspQueued):
            print calc.vaspdir, 'running'
            ready = False
```

2.2 Equilibrium bulk structures in primitive cell for vacancy calculations

The two sections below contain scripts for obtaining relaxed crystal structures of all polymorphs and strained rutile in the primitive cell. The initial guesses for the volumes were taken from relaxed volumes of the surface unit cell, which we previously calculated. The relaxed primitive cells are then used for vacancy formation energy calculations.

2.2.1 Full volume, shape, and ionic relaxation of all bulk primitive cells

In addition to the modified unit cells for surface calculations, we also require the primitive cells for bulk vacancy formation calculations. The code below takes the normalized volumes from the surface unit cell calculates a fully relaxed structure. Note for rutile, previous work has shown that columbite and rutile have similar equilibrium volumes.[3] For ease, the initial guess on the rutile equilibrium volume is taken from the columbite equilibrium volume.

```
from jasp import *
JASPRC['queue.ppn'] = 1
JASPRC['queue.mem'] = '2GB'
from ase.visualize import view
from ase_addons.bulk import anatase, brookite, columbite, pyrite, rutile

data = [['IrO2', 'anatase  ', 12.598],
        ['IrO2', 'columbite', 10.891],
        ['IrO2', 'pyrite   ', 10.023],
        ['IrO2', 'brookite ', 11.736],
        ['RhO2', 'anatase  ', 12.417],
        ['RhO2', 'columbite', 10.650],
        ['RhO2', 'pyrite   ',  9.833],
```

```
14          ['RhO2', 'brookite ', 11.500],
15          ['RuO2', 'anatase  ', 12.451],
16          ['RuO2', 'columbite', 10.691],
17          ['RuO2', 'pyrite   ',  9.805],
18          ['RuO2', 'brookite ', 11.506],
19          ['PtO2', 'anatase  ', 13.368],
20          ['PtO2', 'columbite', 11.192],
21          ['PtO2', 'pyrite   ', 10.380],
22          ['PtO2', 'brookite ', 11.969]]
23
24  for name, struct, vol in data:
25      B = name[:2]
26      X = 'O'
27      atoms = eval(struct)(B, X)
28      atoms.set_volume(len(atoms) * vol)
29      with jasp('{name}/{struct}/ground'.format(**locals()), atoms=atoms,
30                xc='PBE', lreal=False,
31                encut=500, kpts=(7, 7, 7),
32                ispin=1, lorbit=11,
33                ibrion=2, isif=3, nsw=50, ediffg=-0.05, ediff=1e-6,
34                lplane=True, lscalu=False,
35                lwave=False) as calc:
36          try:
37              calc.calculate()
38          except (VaspSubmitted, VaspRunning, VaspQueued):
39              print calc.vaspdir, 'running'
40              ready = False
41
42      # We also want to do rutile, but we only want to do it once per atomic formula
43      # An easy way to do this is just perform the calculation after every columbite calculation
44      if struct == 'columbite':
45          struct = 'rutile'
46          atoms = eval(struct)([B, X], mags=[0, 0])
47          atoms.set_volume(len(atoms) * vol)
48          with jasp('{name}/{struct}/ground'.format(**locals()), atoms=atoms,
49                    xc='PBE', lreal=False,
50                    encut=500, kpts=(7, 7, 7),
51                    ispin=1, lorbit=11,
52                    ibrion=2, isif=3, nsw=50, ediffg=-0.05, ediff=1e-6,
53                    lplane=True, lscalu=False,
54                    lwave=False) as calc:
```

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

2.2.2 Full volume, shape, and ionic relaxation of ground state and strained rutile

The code below calculates strained rutile ground state structures. We also obtained rutile lattice parameters a, c, and u for strained rutile bulk cells. These parameters will be used in the construction of strained surfaces, and the relaxed strained bulk cells will be used in vacancy formation energies.

```
from jasp import *
JASPRC['queue.ppn'] = 1
JASPRC['queue.mem'] = '2GB'
JASPRC['queue.walltime'] = '24:00:00'
from ase.visualize import view
from ase_addons.bulk import anatase, brookite, columbite, pyrite, rutile
import ase_addons.bulk

print '|Name|Strain|a|c|u'
print '|-----|'

for name in ['RuO2', 'IrO2', 'PtO2', 'RhO2']:
    B = name[:2]
    X = 'O'
    with jasp('{name}/rutile/bulk-vacancy/ground'.format(**locals())) as calc:
        atoms = calc.get_atoms()
        v0 = atoms.get_volume()
    for f in [-0.15, -0.10, -0.05, 0.05, 0.10, 0.15]:
        atoms.set_volume(v0 * (1 + f))
        # We also want to do rutile
        wkdir = '{name}/rutile/bulk-vacancy/ground{f:+1.2f}'.format(**locals())
        with jasp(wkdir, atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 7),
```

```
                  ispin=1, lorbit=11,
                  ibrion=2, isif=4, nsw=50, ediffg=-0.05, ediff=1e-6,
                  lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
                atoms = calc.get_atoms()
                a = atoms.get_cell()[0][0]
                c = atoms.get_cell()[2][2]
                u = atoms.get_scaled_positions()[2][0]
                print '|{name}|{f:1.2f}|{a:1.3f}|{c:1.3f}|{u:1.3f}|'.format(**locals())
            except (VaspSubmitted, VaspRunning, VaspQueued):
                print '|{name}|{f:1.2f}|running|running|running|'.format(**locals())
                ready = False
```

Table 3: Relaxed primitive cell parameters (a, c) and oxygen parameter (u) of relaxed, strained rutile structure.

Name	Strain	a	c	u
RuO_2	-0.15	4.242	3.064	0.309
RuO_2	-0.10	4.350	3.085	0.307
RuO_2	-0.05	4.454	3.106	0.306
RuO_2	0.05	4.639	3.164	0.306
RuO_2	0.10	4.726	3.194	0.307
RuO_2	0.15	4.822	3.207	0.308
IrO_2	-0.15	4.241	3.113	0.311
IrO_2	-0.10	4.349	3.135	0.310
IrO_2	-0.05	4.449	3.161	0.309
IrO_2	0.05	4.635	3.219	0.308
IrO_2	0.10	4.724	3.247	0.309
IrO_2	0.15	4.811	3.273	0.309
PtO_2	-0.15	4.295	3.156	0.312
PtO_2	-0.10	4.397	3.188	0.311
PtO_2	-0.05	4.499	3.215	0.310
PtO_2	0.05	4.679	3.285	0.310
PtO_2	0.10	4.767	3.316	0.310
PtO_2	0.15	4.863	3.332	0.311
RhO_2	-0.15	4.256	3.042	0.309
RhO_2	-0.10	4.360	3.069	0.308
RhO_2	-0.05	4.461	3.094	0.307
RhO_2	0.05	4.641	3.160	0.307
RhO_2	0.10	4.731	3.186	0.308
RhO_2	0.15	4.822	3.206	0.309

2.3 Relaxed metastable polymorph surface unit cells

After we obtain the fully relaxed, ground state unit cell of each structure, we then create a surface slab from the bulk calculation. To do this, we merely extend the length of the a_3 cell vector to create 10 $\mathring{A}$ of vacuum between repeating slabs. We also fix the bottom half of the slab to simulate the bulk and relax the top half to simulate the surface.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[anatase001, 'anatase', 2],
         [columbite101, 'columbite', 4],
         [pyrite001, 'pyrite', 2],
         [brookite110, 'brookite', 2]]

for name in rutiles:
    for func, struct, layers in funcs:
        m = name[:2]
        # First load the atoms object with the correct constraints
        atoms = func(B=m, X='O', vacuum=0, fixlayers=layers, mags=[0, 0])
        with jasp('{name}/{struct}/ground'.format(**locals())) as calc:
            ground = calc.get_atoms()
            atoms.set_cell(ground.get_cell())
            atoms.set_positions(ground.get_positions())
        atoms.center(vacuum=10, axis=2)

        # Then perform the calculation
        with jasp('{name}/{struct}/bare'.format(**locals()), atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 1),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
```

```
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspSubmitted, VaspRunning, VaspQueued):
                pass
```

2.4 Relaxed rutile surface slab

In addition to relaxed surfaces of the metastable polymorphs, we also want the relaxed surface of rutile. These surfaces have already been calculated, so their relaxed atomic positions are loaded up from the json file of previous work.[4]

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase import Atoms
from ase.visualize import view
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

with open('rutiles-data.json', 'r') as f:
    data = json.load(f)

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

constr_index = [0, 1, 24, 3, 27, 4, 28, 2, 25, 26, 5, 29,
                6, 30, 11, 35, 7, 8, 31, 32, 9, 33, 10, 34]

for dopant in rutiles:
    symbols = [str(sym) for sym in data[dopant]['bare']['symbols']]
    pos = data[dopant]['bare']['pos']
    cell = data[dopant]['bare']['cell']

    atoms = Atoms(symbols=symbols, positions=pos, cell=cell)
    c = FixAtoms(indices=constr_index)
    atoms.set_constraint(c)

```

```
    with jasp('{dopant}/rutile/bare'.format(**locals()), atoms=atoms,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, setups={'Nb': '_sv'}) as calc:
        try:
            calc.calculate()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            print calc.vaspdir, 'running'
```

2.5 Relaxed strained rutile surface slabs

We also calculate adsorption energies on strained rutile surfaces. The a, c, and u lattice parameters are taken from strained rutile primitive cells.

```
from jasp import *
JASPRC['queue.ppn'] = 8
JASPRC['queue.mem'] = '16GB'
from ase import Atoms
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

data = [['RuO2', -0.15, 4.242, 3.064, 0.309],
        ['RuO2', -0.10, 4.350, 3.085, 0.307],
        ['RuO2', -0.05, 4.454, 3.106, 0.306],
        ['RuO2',  0.05, 4.639, 3.164, 0.306],
        ['RuO2',  0.10, 4.726, 3.194, 0.307],
        ['RuO2',  0.15, 4.822, 3.207, 0.308],
        ['IrO2', -0.15, 4.241, 3.113, 0.311],
        ['IrO2', -0.10, 4.349, 3.135, 0.310],
        ['IrO2', -0.05, 4.449, 3.161, 0.309],
        ['IrO2',  0.05, 4.635, 3.219, 0.308],
        ['IrO2',  0.10, 4.724, 3.247, 0.309],
        ['IrO2',  0.15, 4.811, 3.273, 0.309],
```

```
        ['PtO2', -0.15, 4.295, 3.156, 0.312],
        ['PtO2', -0.10, 4.397, 3.188, 0.311],
        ['PtO2', -0.05, 4.499, 3.215, 0.310],
        ['PtO2',  0.05, 4.679, 3.285, 0.310],
        ['PtO2',  0.10, 4.767, 3.316, 0.310],
        ['PtO2',  0.15, 4.863, 3.332, 0.311],
        ['RhO2', -0.15, 4.256, 3.042, 0.309],
        ['RhO2', -0.10, 4.360, 3.069, 0.308],
        ['RhO2', -0.05, 4.461, 3.094, 0.307],
        ['RhO2',  0.05, 4.641, 3.160, 0.307],
        ['RhO2',  0.10, 4.731, 3.186, 0.308],
        ['RhO2',  0.15, 4.822, 3.206, 0.309]]

for name, strain, a, c, u in data:
    B = name[:2]
    X = 'O'
    atoms = rutile110([B, X], a, c, u, vacuum=10, layers=12, fixlayers=6)

    with jasp('{name}/rutile/bare{strain:+1.2f}'.format(**locals()), atoms=atoms,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, setups={'Nb': '_sv'}) as calc:
        try:
            E_slab = atoms.get_potential_energy()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            E_slab = None
```

3 Calculation of ΔE^{O}_{ads}, ΔE^{OH}_{ads}, and ΔE^{OOH}_{ads}

The scripts below detail the adsorption energy calculations for O, OH, and OOH adsorbates. All adsorption calculations are done in two steps. The first script performs only allows the adsorbate to relax, while the second script starts from this relaxed structure and allows the top half of the surface slab relax. This is done to minimize major reconstructions that involve breaking of surface and/or adsorbate bonds. In previous work, we did not observe

major surface relaxations of rutile. Hence, for adsorption energies of rutile, we complete the process in one step, bypassing the calculation with a fixed surface.

3.1 ΔE_{ads}^{O} on rutile surfaces

The script below calculates the total energy of a rutile slab with an adsorbed O atom. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase import Atoms
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

ads_index = 44

for name in rutiles:
    ads = Atoms([Atom('O', (0, 0, 0), tag=-1)])
    # First load up the relaxed rutile110 surface
    with jasp('{name}/rutile/bare'.format(**locals())) as calc:
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2]
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
        if h_site > h_max:
            height = 2
        else:
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)
    old_atoms = slab.copy()

    # Then perform the calculation
```

```
    with jasp('{name}/rutile/O'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, lcharg=True) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms

            calc.calculate()
            converged = True
            required = False
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False
            required = True
```

3.2 ΔE^O_{ads} on metastable polymorph surfaces

3.2.1 Fixed surfaces

The script below calculates the total energy of all metastable polymorph surfaces with an adsorbed O atom. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']
```

```
11
12  funcs = [[anatase001, 'anatase', 4, 31],
13           [columbite101, 'columbite', 8, 38],
14           [pyrite001, 'pyrite', 4, 14],
15           [brookite110, 'brookite', 4, 31]]
16
17  O = Atoms([Atom('O', (0, 0, 0), tag=-1)])
18
19  for name in rutiles:
20      for func, struct, layers, ads_index in funcs:
21          ads = O.copy()
22          m = name[:2]
23
24          # First load the atoms object with the correct constraints
25          with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
26              if not calc.read_convergence():
27                  continue
28              slab = calc.get_atoms()
29              pos = slab.get_positions()[ads_index][:2]
30              h_site = slab.get_positions()[ads_index][2]
31              h_max = max(slab.get_positions()[:, 2])
32              if h_site > h_max:
33                  height = 2
34              else:
35                  height = 2 - (h_max - h_site)
36
37          add_adsorbate(slab, ads, height=height, position=pos)
38
39          # We want to fix all atoms except the adsorbate atom
40          c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
41          slab.set_constraint(c)
42
43          old_atoms = slab.copy()
44
45          # Then perform the calculation
46          with jasp('{name}/{struct}/O-fix-slab'.format(**locals()), atoms=slab,
47                    xc='PBE', lreal=False,
48                    encut=500, kpts=(7, 7, 1),
49                    ispin=1, lorbit=11,
50                    ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
51                    npar=4, nsim=4, lplane=True, lscalu=False,
```

```
                    lwave=False) as calc:
            try:
                atoms = calc.get_atoms()
                queued = hasattr(calc, 'vasp_queued')
                new_atoms = atoms != old_atoms

                calc.calculate()
                converged = True
                required = False
            except (VaspSubmitted, VaspRunning, VaspQueued):
                queued = True
                converged = False
                required = True
```

3.2.2 Relaxed surfaces

The script below calculates the total energy of all metastable polymorph surfaces with an adsorbed O atom. The initial structures are taken from the surface + adsorbate structure calculated in the previous section, where only the adsorbate was allowed to move. In this calculation, the top half of the slab in addition to the adsorbate are allowed to move.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['TiO2', 'IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[anatase001, 'anatase', 4, 31],
         [columbite101, 'columbite', 8, 38],
         [pyrite001, 'pyrite', 4, 14],
         [brookite110, 'brookite', 4, 31]]

O = Atoms([Atom('O', (0, 0, 0), tag=-1)])
```

```
18
19  print '|Compound|Struct|Queued|New Atoms?|Converged|Calc Required|'
20  print '|----|'
21
22  for name in rutiles:
23      for func, struct, layers, ads_index in funcs:
24          ads = O.copy()
25          m = name[:2]
26
27          # First load the atoms object with the correct constraints
28          with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
29              if not calc.read_convergence():
30                  continue
31              slab = calc.get_atoms()
32              pos = slab.get_positions()[ads_index][:2]
33              h_site = slab.get_positions()[ads_index][2]
34              h_max = max(slab.get_positions()[:, 2])
35              if h_site > h_max:
36                  height = 2
37              else:
38                  height = 2 - (h_max - h_site)
39
40          add_adsorbate(slab, ads, height=height, position=pos)
41
42          # Now load the relaxed coordinates of the first relaxation step
43          with jasp('{name}/{struct}/O-fix-slab'.format(**locals())) as calc:
44              try:
45                  if not calc.read_convergence():
46                      continue
47              except (IOError):
48                  print calc.vaspdir, 'No Outcar'
49                  continue
50              phase_1_atoms = calc.get_atoms()
51              phase_1_pos = phase_1_atoms.get_positions()
52              slab.set_positions(phase_1_pos)
53
54          old_atoms = slab.copy()
55
56          # Then perform the calculation
57          with jasp('{name}/{struct}/O'.format(**locals()), atoms=slab,
58                    xc='PBE', lreal=False,
```

```
                    encut=500, kpts=(7, 7, 1),
                    ispin=1, lorbit=11,
                    ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
                    npar=4, nsim=4, lplane=True, lscalu=False,
                    lwave=False) as calc:
            try:
                atoms = calc.get_atoms()
                queued = hasattr(calc, 'vasp_queued')
                new_atoms = atoms != old_atoms
                calc.calculate()
                converged = calc.read_convergence()
                required = calc.calculation_required(atoms, [])
                print '|{0}|{1}|{2}|{3}|{4}|{5}|'.format(name, struct, queued,
                                                        new_atoms, converged, required)
            except (VaspSubmitted, VaspRunning, VaspQueued):
                queued = True
                converged = False
                required = True
                print '|{0}|{1}|{2}|{3}|{4}|{5}|'.format(name, struct, queued,
                                                        new_atoms, converged, required)
```

3.3 ΔE_{ads}^{OH} on rutile surfaces

The script below calculates the total energy of a rutile slab with an adsorbed OH molecule. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase import Atoms
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

```

```
ads_index = 44

for name in rutiles:
    ads = Atoms([Atom('O', (0, 0, 0)),
                 Atom('H', (-0.85, 0, 0.35))])

    with jasp('{name}/rutile/bare'.format(**locals())) as calc:
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2]
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
        if h_site > h_max:
            height = 2
        else:
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)
    old_atoms = slab.copy()

    # Then perform the calculation
    with jasp('{name}/rutile/OH'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, lcharg=True) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms
            O_ind, H_ind = len(slab) - 1, len(slab) - 2
            OH_length = atoms.get_distance(O_ind, H_ind)

            calc.calculate()
            converged = calc.read_convergence()
            required = calc.calculation_required(atoms, [])
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False
            required = True
```

3.4 ΔE_{ads}^{OH} on metastable polymorph surfaces

3.4.1 Fixed surfaces

The script below calculates the total energy of all metastable polymorph surfaces with an adsorbed OH molecule. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[anatase001, 'anatase', 4, 31, 225 * np.pi / 180],
         [columbite101, 'columbite', 8, 38, 0],
         [pyrite001, 'pyrite', 4, 14, 0],
         [brookite110, 'brookite', 4, 31, 0]]

OH = Atoms([Atom('O', (0, 0, 0), tag=-1),
            Atom('H', (-0.85, 0, 0.35), tag=-1)])

for name in rutiles:
    for func, struct, layers, ads_index, angle in funcs:
        ads = OH.copy()
        ads.rotate('z', angle)
        m = name[:2]

        # First load the atoms object with the correct constraints
        with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
            if not calc.read_convergence():
                continue
            slab = calc.get_atoms()
            pos = slab.get_positions()[ads_index][:2]
```

```
            h_site = slab.get_positions()[ads_index][2]
            h_max = max(slab.get_positions()[:, 2])
            if h_site > h_max:
                height = 2
            else:
                height = 2 - (h_max - h_site)

        add_adsorbate(slab, ads, height=height, position=pos)

        c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
        slab.set_constraint(c)

        old_atoms = slab.copy()

        # Then perform the calculation
        with jasp('{name}/{struct}/OH-fix-slab'.format(**locals()), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 1),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                atoms = calc.get_atoms()
                queued = hasattr(calc, 'vasp_queued')
                new_atoms = atoms != old_atoms
                O_ind, H_ind = len(slab) - 1, len(slab) - 2
                OH_length = atoms.get_distance(O_ind, H_ind)

                calc.calculate()
                converged = calc.read_convergence()
                required = calc.calculation_required(atoms, [])
            except (VaspSubmitted, VaspRunning, VaspQueued):
                queued = True
                converged = False
                required = True
```

3.4.2 Relaxed surfaces

The script below calculates the total energy of all metastable polymorph surfaces with an adsorbed OH molecule. The initial structures are taken from the surface + adsorbate structure calculated in the previous section, where only the adsorbate was allowed to move. In this calculation, the top half of the slab in addition to the adsorbate are allowed to move.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['TiO2', 'IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[anatase001, 'anatase', 4, 31, 225 * np.pi / 180],
         [columbite101, 'columbite', 8, 38, 0],
         [pyrite001, 'pyrite', 4, 14, 0],
         [brookite110, 'brookite', 4, 31, 0]]

OH = Atoms([Atom('O', (0, 0, 0), tag=-1),
            Atom('H', (-0.85, 0, 0.35), tag=-1)])

for name in rutiles:
    for func, struct, layers, ads_index, angle in funcs:
        ads = OH.copy()
        ads.rotate('z', angle)
        m = name[:2]

        # First load the atoms object with the correct constraints
        with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
            if not calc.read_convergence():
                continue
            slab = calc.get_atoms()
            pos = slab.get_positions()[ads_index][:2]
```

```
33              h_site = slab.get_positions()[ads_index][2]
34              h_max = max(slab.get_positions()[:, 2])
35              if h_site > h_max:
36                  height = 2
37              else:
38                  height = 2 - (h_max - h_site)
39
40          add_adsorbate(slab, ads, height=height, position=pos)
41
42          # Now load the relaxed coordinates of the first relaxation step
43          with jasp('{name}/{struct}/OH-fix-slab'.format(**locals())) as calc:
44              try:
45                  if not calc.read_convergence():
46                      continue
47              except (IOError):
48                  print calc.vaspdir, 'No Outcar'
49                  continue
50              phase_1_atoms = calc.get_atoms()
51              phase_1_pos = phase_1_atoms.get_positions()
52              slab.set_positions(phase_1_pos)
53
54          old_atoms = slab.copy()
55
56          # Then perform the calculation
57          with jasp('{name}/{struct}/OH'.format(**locals()), atoms=slab,
58                    xc='PBE', lreal=False,
59                    encut=500, kpts=(7, 7, 1),
60                    ispin=1, lorbit=11,
61                    ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
62                    npar=4, nsim=4, lplane=True, lscalu=False,
63                    lwave=False) as calc:
64              try:
65                  atoms = calc.get_atoms()
66                  queued = hasattr(calc, 'vasp_queued')
67                  new_atoms = atoms != old_atoms
68                  O_ind, H_ind = len(slab) - 1, len(slab) - 2
69                  OH_length = atoms.get_distance(O_ind, H_ind)
70
71                  calc.calculate()
72                  converged = calc.read_convergence()
73                  required = calc.calculation_required(atoms, [])
```

```
            except (VaspSubmitted, VaspRunning, VaspQueued):
                queued = True
                converged = False
                required = True
```

3.5 ΔE_{ads}^{OOH} on rutile surfaces

The script below calculates the total energy of a rutile slab with an adsorbed OOH molecule. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2.

```
from jasp import *
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase import Atoms
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

rutiles = ['TiO2', 'IrO2', 'RhO2', 'RuO2', 'PtO2']

ads_index = 44

for name in rutiles:
    ads = Atoms([Atom('O', (0, 0, 0)),
                 Atom('O', (0, -1.165, 0.686)),
                 Atom('H', (-1, -0.8689, 1))])
    ads.rotate('z', 90 * np.pi / 180)

    with jasp('{name}/rutile/bare'.format(**locals())) as calc:
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2]
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
        if h_site > h_max:
            height = 2
        else:
```

```
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)
    old_atoms = slab.copy()

    # Then perform the calculation
    with jasp('{name}/rutile/OOH'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, lcharg=True) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms

            calc.calculate()
            converged = True
            required = False
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False
            required = True
```

3.6 ΔE_{ads}^{OOH} on metastable polymorph surfaces

Due to the numerous possible orientations the OOH adsorbate can take, care must be taken to assure that we are only capturing the strength of the surface-O bond. Hence, several iterations of starting geometries had to be tested, namely for the brookite and pyrite surfaces. Furthermore, initial starting geometries for none of the surfaces were similar. The scripts below show only the final iteration.

3.6.1 Fixed surfaces: anatase (001)

For the anatase (001) surface, we had to arrange the adsorbate to limit interactions of the surface and the adsorbate outside of the surface-O bond. The code below shows the final iteration that gave a relaxed OOH adsorbate. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']


OOH = Atoms([Atom('O', (0, 0, 0), tag=-1),
             Atom('O', (0, -1.165, 0.686), tag=-1),
             Atom('H', (0, -0.8, 1.7), tag=-1)])

for name in rutiles:
    func = anatase001
    struct = 'anatase'
    layers = 4
    ads_index = 31
    angle = 90 * np.pi / 180

    ads = OOH.copy()
    ads.rotate('z', angle)
    m = name[:2]

    # First load the atoms object with the correct constraints
    with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2]
```

```
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
        if h_site > h_max:
            height = 2
        else:
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)

    c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
    slab.set_constraint(c)

    O1_ind, O2_ind, H_ind = len(slab) - 3, len(slab) - 2, len(slab) - 1

    old_atoms = slab.copy()

    # Then perform the calculation
    with jasp('{name}/{struct}/OOH-fix-slab'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms
            OO_length = atoms.get_distance(O1_ind, O2_ind)
            OH_length = atoms.get_distance(O2_ind, H_ind)
            calc.calculate()
            converged = calc.read_convergence()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False

        status = '|{0}|{1}|{2}|{3}|{4}|{5:1.2f}|{6:1.2f}|'
```

3.6.2 Fixed surfaces: pyrite (001)

For the pyrite (001) surface, we had to arrange the adsorbate to limit interactions of the surface and the adsorbate outside of the surface-O bond. The code below shows the final iteration that gave a relaxed OOH adsorbate. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

OOH = Atoms([Atom('O', (0, 0, 0), tag=-1),
             Atom('O', (0.8, 0,  1.2), tag=-1),
             Atom('H', (0, 0, 2.0), tag=-1)])

for name in rutiles:
    func = pyrite001
    struct = 'pyrite'
    layers = 4
    ads_index = 14
    angle = 90 * np.pi / 180

    ads = OOH.copy()
    m = name[:2]

    # First load the atoms object with the correct constraints
    with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2]
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
```

```
        if h_site > h_max:
            height = 2
        else:
            height = 2 - (h_max - h_site)

    pos += [-1.2, 0]

    add_adsorbate(slab, ads, height=height, position=pos)

    c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
    slab.set_constraint(c)

    O1_ind, O2_ind, H_ind = len(slab) - 3, len(slab) - 2, len(slab) - 1

    old_atoms = slab.copy()

    # Then perform the calculation
    with jasp('{name}/{struct}/OOH-fix-slab'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms
            OO_length = atoms.get_distance(O1_ind, O2_ind)
            OH_length = atoms.get_distance(O2_ind, H_ind)
            calc.calculate()
            converged = calc.read_convergence()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False

        status = '|{0}|{1}|{2}|{3}|{4}|{5:1.2f}|{6:1.2f}|'
    atoms.set_calculator()
    view(atoms)
```

3.6.3 Fixed surfaces: brookite (110)

For the brookite (110) surface, we had to arrange the adsorbate to limit interactions of the surface and the adsorbate outside of the surface-O bond. The code below shows the final iteration that gave a relaxed OOH adsorbate. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

OOH = Atoms([Atom('O', (0, 0, 0), tag=-1),
             Atom('O', (0, -1.165, 0.9), tag=-1),
             Atom('H', (0, -1, 1.8), tag=-1)])

for name in rutiles:
    func = brookite110
    struct = 'brookite'
    layers = 4
    ads_index = 31
    angle = 90 * np.pi / 180
    ads = OOH.copy()
    ads.rotate('z', angle)
    m = name[:2]

    # First load the atoms object with the correct constraints
    with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
        if not calc.read_convergence():
            continue
        slab = calc.get_atoms()
        pos = slab.get_positions()[ads_index][:2] + np.array([-0.5, 0.7])
```

```
        h_site = slab.get_positions()[ads_index][2]
        h_max = max(slab.get_positions()[:, 2])
        if h_site > h_max:
            height = 2
        else:
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)

    c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
    slab.set_constraint(c)

    O1_ind, O2_ind, H_ind = len(slab) - 3, len(slab) - 2, len(slab) - 1

    old_atoms = slab.copy()

    # Then perform the calculation
    with jasp('{name}/{struct}/OOH-fix-slab'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms
            OO_length = atoms.get_distance(O1_ind, O2_ind)
            OH_length = atoms.get_distance(O2_ind, H_ind)
            calc.calculate()
            converged = calc.read_convergence()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False

        status = '|{0}|{1}|{2}|{3}|{4}|{5:1.2f}|{6:1.2f}|'
```

3.6.4 Fixed surfaces: columbite (101)

For the columbite (101) surface, we had to arrange the adsorbate to limit interactions of the surface and the adsorbate outside of the surface-O bond. The code below shows the final iteration that gave a relaxed OOH adsorbate. The relaxed surfaces are taken from the converged surface calculation calculated in Section 2. Only the adsorbate is allowed to relax.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[columbite101, 'columbite', 8, 38, 0]]

OOH = Atoms([Atom('O', (0, 0, 0), tag=-1),
             Atom('O', (0, -1.165, 0.686), tag=-1),
             Atom('H', (-1, -0.8689, 1), tag=-1)])

for name in rutiles:
    for func, struct, layers, ads_index, angle in funcs:
        ads = OOH.copy()
        ads.rotate('z', angle)
        m = name[:2]

        # First load the atoms object with the correct constraints
        with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
            if not calc.read_convergence():
                continue
            slab = calc.get_atoms()
            pos = slab.get_positions()[ads_index][:2]
            h_site = slab.get_positions()[ads_index][2]
            h_max = max(slab.get_positions()[:, 2])
```

```
            if h_site > h_max:
                height = 2
            else:
                height = 2 - (h_max - h_site)

        add_adsorbate(slab, ads, height=height, position=pos)

        c = FixAtoms(indices=[atom.index for atom in slab if atom.tag != -1])
        slab.set_constraint(c)

        O1_ind, O2_ind, H_ind = len(slab) - 3, len(slab) - 2, len(slab) - 1

        old_atoms = slab.copy()

          # Then perform the calculation
        with jasp('{name}/{struct}/OOH-fix-slab'.format(**locals()), atoms=slab,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 1),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
                  npar=4, nsim=4, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                atoms = calc.get_atoms()
                queued = hasattr(calc, 'vasp_queued')
                new_atoms = atoms != old_atoms
                OO_length = atoms.get_distance(O1_ind, O2_ind)
                OH_length = atoms.get_distance(O2_ind, H_ind)
                calc.calculate()
                converged = calc.read_convergence()
            except (VaspSubmitted, VaspRunning, VaspQueued):
                queued = True
                converged = False

            status = '|{0}|{1}|{2}|{3}|{4}|{5:1.2f}|{6:1.2f}|'
```

3.6.5 Relaxed surfaces

The script below calculates the total energy of all metastable polymorph surfaces with an adsorbed OOH molecule. The initial structures are taken from the surface + adsorbate

structure calculated in the previous sections, where only the adsorbate was allowed to move. In this calculation, the top half of the slab in addition to the adsorbate are allowed to move.

```
from jasp import *
import numpy as np
JASPRC['queue.ppn'] = 16
JASPRC['queue.mem'] = '32GB'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.lattice.surface import add_adsorbate
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']

funcs = [[anatase001, 'anatase', 4, 31, 225 * np.pi / 180],
         [columbite101, 'columbite', 8, 38, 0],
         [pyrite001, 'pyrite', 4, 14, 270 * np.pi / 180],
         [brookite110, 'brookite', 4, 31, 270 * np.pi / 180]]

OOH = Atoms([Atom('O', (0, 0, 0), tag=-1),
             Atom('O', (0, -1.165, 0.686), tag=-1),
             Atom('H', (-1, -0.8689, 1), tag=-1)])

for name in rutiles:
    for func, struct, layers, ads_index, angle in funcs:
        ads = OOH.copy()
        ads.rotate('z', angle)
        m = name[:2]

        # First load the atoms object with the correct constraints
        with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
            if not calc.read_convergence():
                continue
            slab = calc.get_atoms()
            pos = slab.get_positions()[ads_index][:2]
            h_site = slab.get_positions()[ads_index][2]
            h_max = max(slab.get_positions()[:, 2])
            if h_site > h_max:
                height = 2
            else:
```

```
            height = 2 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)

    # Now load the relaxed coordinates of the first relaxation step
    with jasp('{name}/{struct}/OOH-fix-slab'.format(**locals())) as calc:
        try:
            if not calc.read_convergence():
                continue
        except (IOError):
            print calc.vaspdir, 'No Outcar'
            continue
        phase_1_atoms = calc.get_atoms()
        phase_1_pos = phase_1_atoms.get_positions()
        slab.set_positions(phase_1_pos)

    old_atoms = slab.copy()
    O1_ind, O2_ind, H_ind = len(slab) - 3, len(slab) - 2, len(slab) - 1

    # Then perform the calculation
    with jasp('{name}/{struct}/OOH'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False) as calc:
        try:
            atoms = calc.get_atoms()
            queued = hasattr(calc, 'vasp_queued')
            new_atoms = atoms != old_atoms
            OO_length = atoms.get_distance(O1_ind, O2_ind)
            OH_length = atoms.get_distance(O2_ind, H_ind)
            calc.calculate()
            converged = calc.read_convergence()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            queued = True
            converged = False
```

3.7 ΔE^{O}_{ads} on strained rutile surfaces

In addition to adsorption energies on ground state structures, we also calculated adsorption energies on strained rutile. Note, due to the well behaved nature of the rutile (110) surface, we initialized the starting geometry straight from the a, c, and u calculated from the relaxed, strained rutile primitive cell in 2. These values were taken from Table 3.

```
from jasp import *
JASPRC['queue.ppn'] = 8
JASPRC['queue.mem'] = '16GB'
from ase import Atoms
from ase.visualize import view
from ase.lattice.surface import add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase_addons.surfaces import rutile110
import json

data = [['RuO2', -0.15, 4.242, 3.064, 0.309],
        ['RuO2', -0.10, 4.350, 3.085, 0.307],
        ['RuO2', -0.05, 4.454, 3.106, 0.306],
        ['RuO2',  0.05, 4.639, 3.164, 0.306],
        ['RuO2',  0.10, 4.726, 3.194, 0.307],
        ['RuO2',  0.15, 4.822, 3.207, 0.308],
        ['IrO2', -0.15, 4.241, 3.113, 0.311],
        ['IrO2', -0.10, 4.349, 3.135, 0.310],
        ['IrO2', -0.05, 4.449, 3.161, 0.309],
        ['IrO2',  0.05, 4.635, 3.219, 0.308],
        ['IrO2',  0.10, 4.724, 3.247, 0.309],
        ['IrO2',  0.15, 4.811, 3.273, 0.309],
        ['PtO2', -0.15, 4.295, 3.156, 0.312],
        ['PtO2', -0.10, 4.397, 3.188, 0.311],
        ['PtO2', -0.05, 4.499, 3.215, 0.310],
        ['PtO2',  0.05, 4.679, 3.285, 0.310],
        ['PtO2',  0.10, 4.767, 3.316, 0.310],
        ['PtO2',  0.15, 4.863, 3.332, 0.311],
        ['RhO2', -0.15, 4.256, 3.042, 0.309],
        ['RhO2', -0.10, 4.360, 3.069, 0.308],
        ['RhO2', -0.05, 4.461, 3.094, 0.307],
        ['RhO2',  0.05, 4.641, 3.160, 0.307],
```

```
        ['RhO2',  0.10, 4.731, 3.186, 0.308],
        ['RhO2',  0.15, 4.822, 3.206, 0.309]]

ads_index = 44

O2 = -9.849707 + 1.198

print '|Name|Strain|E_ads|'
print '|----|'

for name, strain, a, c, u in data:
    ads = Atoms([Atom('O', (0, 0, 0), tag=-1)])
    B = name[:2]
    X = 'O'
    atoms = rutile110([B, X], a, c, u, vacuum=10, layers=12, fixlayers=6)
    slab = atoms.copy()

    pos = slab.get_positions()[ads_index][:2]
    h_site = slab.get_positions()[ads_index][2]
    h_max = max(slab.get_positions()[:, 2])
    if h_site > h_max:
        height = 1.8
    else:
        height = 1.8 - (h_max - h_site)

    add_adsorbate(slab, ads, height=height, position=pos)

    with jasp('{name}/rutile/O{strain:+1.2f}'.format(**locals()), atoms=slab,
              xc='PBE', lreal=False,
              encut=500, kpts=(7, 7, 1),
              ispin=1, lorbit=11,
              ibrion=2, isif=2, nsw=50, ediffg=-0.05, ediff=1e-6,
              npar=4, nsim=4, lplane=True, lscalu=False,
              lwave=False, lcharg=True) as calc:
        try:
            E_ads = slab.get_potential_energy()
        except (VaspSubmitted, VaspRunning, VaspQueued):
            E_ads = None
```

4 Calculation of ΔE^O_{vac}

4.1 ΔE^O_{vac} in ground state structures of all polymorphs

The script below performs total energy calculations of bulk unit cells with a missing oxygen anion. The chosen vacancy concentration is 6.25%, or one out of every 16 oxygen anions missing. Given that primitive cells of different polymorphs have different number of formula units, we ensured all vacancy concentrations are the same by creating a $1 \times 2 \times 2$ supercell of primitive rutile, a $2 \times 2 \times 1$ supercell of primitive anatase, a $1 \times 1 \times 2$ of primtive columbite, and a $2 \times 1 \times 1$ primitive pyrite. Brookite's primitive cell was large enough so a single oxygen vacancy resulted in a vacancy concentration of 6.25%.

```
import os
from jasp import *
JASPRC['queue.ppn'] = 4
JASPRC['queue.mem'] = '8GB'
from ase.visualize import view
from ase_addons.bulk import anatase, brookite, columbite, pyrite, rutile

for name in ['TiO2', 'IrO2', 'RhO2', 'RuO2', 'PtO2']:
    for struct in ['rutile', 'anatase', 'columbite', 'pyrite', 'brookite']:
        with jasp('{name}/{struct}/bulk-vacancy/ground'.format(**locals())) as calc:
            if not os.path.exists('OUTCAR') or calc.read_convergence() == False:
                print '|{name}|{struct}|ground not done|ground not done|'.format(**locals())
                continue
            atoms = calc.get_atoms()
            atoms.set_constraint()
        if struct == 'rutile':
            atoms *= (1, 2, 2)
        elif struct == 'anatase':
            atoms *= (2, 2, 1)
        elif struct == 'columbite':
            atoms *= (1, 1, 2)
        elif struct == 'pyrite':
            atoms *= (2, 1, 1)
        n0 = atoms.get_chemical_symbols().count('O')
        atoms.pop(atoms.get_chemical_symbols().index('O'))
```

```
        n = atoms.get_chemical_symbols().count('O')
        frac = float(n) / float(n0)

        with jasp('{name}/{struct}/bulk-vacancy/{frac:1.4f}'.format(**locals()), atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 7),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=3, nsw=50, ediffg=-0.05, ediff=1e-6,
                  lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
                time = calc.get_elapsed_time()
            except (VaspSubmitted, VaspRunning, VaspQueued):
                pass
```

4.2 ΔE^{O}_{vac} in strained rutile

The script below performs total energy calculations of strained rutile unit cells with a missing oxygen anion. The chosen vacancy concentration is 6.25%, or one out of every 16 oxygen anions missing. The starting structure is read a previously relaxed strained rutile structure, which was calculated in Section 2.

```
from jasp import *
JASPRC['queue.ppn'] = 4
JASPRC['queue.mem'] = '8GB'
JASPRC['queue.walltime'] = '168:00:00'
from ase.visualize import view
from ase_addons.bulk import anatase, brookite, columbite, pyrite, rutile
import ase_addons.bulk

O2 = -9.849707 + 1.198

for name in ['RuO2', 'IrO2', 'PtO2', 'RhO2']:
    B = name[:2]
    X = 'O'
    for f in [-0.15, -0.10, -0.05, 0.05, 0.10, 0.15]:
        with jasp('{name}/rutile/bulk-vacancy/ground{f:+1.2f}'.format(**locals())) as calc:
```

```
            atoms = calc.get_atoms()
            E0 = atoms.get_potential_energy()
            atoms.set_constraint()
            atoms *= (1, 2, 2)

        atoms.pop(atoms.get_chemical_symbols().index('O'))
        # We also want to do rutile
        wkdir = '{name}/rutile/bulk-vacancy/0.9375{f:+1.2f}'.format(**locals())
        with jasp(wkdir, atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 7),
                  ispin=1, lorbit=11,
                  ibrion=2, isif=4, nsw=30, ediffg=-0.05, ediff=1e-6,
                  lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspSubmitted, VaspRunning, VaspQueued):
                pass
```

5 Re-calculation of density of states of correctly oriented surface slabs of brookite, columbite, and pyrite

In VASP, the orbital specific, atom projected density of states is given directly in the DOSCAR. For the d-states, the d_{xy}, d_{xz}, d_{yz}, d_{z^2}, and $d_{x^2-y^2}$ are given separately. If the six oxygen atoms lie directly on the x, y, and z axis, than d_{z^2}, and $d_{x^2-y^2}$ make up the e_g orbitals while d_{xy}, d_{xz} and d_{yz} orbitals make up the t_{2g} orbitals. Hence, in order to extract out the e_g and t_{2g} density of states, we must make sure our structure is oriented correctly.

In anatase (001), this is already the case. For rutile (110), there is a 45° rotation, so the d_{xy} and $d_{x^2-y^2}$ can be switched for the correct interpretation. For columbite (101), brookite (110), and pyrite (001), there are slight rotations one need to perform so the oxygen atoms

are correct oriented. The script below does this.

```
from jasp import *
JASPRC['queue.ppn'] = 6
JASPRC['queue.mem'] = '12GB'
JASPRC['queue.walltime'] = '168:00:00'
from ase_addons.surfaces import *
import ase_addons.bulk
from ase.visualize import view
from ase.utils.eos import EquationOfState

rutiles = ['IrO2', 'RhO2', 'RuO2', 'PtO2']
funcs = [[columbite101, 'columbite', [38, 35, 46, 43]],
         [pyrite001, 'pyrite', [14, 20, 18, 16]],
         [brookite110, 'brookite', [31, 35, 20, 39]]]

for name in rutiles:
    for func, struct, ind in funcs:
        if (name == 'PtO2' and struct == 'columbite'):
            continue
        m = name[:2]

        with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
            atoms = calc.get_atoms()

        if struct == 'pyrite':
            pos = atoms.get_scaled_positions()
            pos[20] = pos[20] - np.array([0, 1, 0])
            atoms.set_scaled_positions(pos)

        atoms.set_calculator()
        m, subO, surfOx, surfOy = ind
        # First rotate the cell
        vz = np.array([0, 0, -1])
        pos = atoms.get_positions()
        vsubO = pos[subO] - pos[m]
        vsubO /= np.linalg.norm(vsubO)
        angle = np.vdot(vz, vsubO)
        angle = np.arccos(angle)
        cross = np.cross(vz, vsubO)
        cross *= angle / np.linalg.norm(cross)
```

```
        atoms.rotate(-cross, rotate_cell=True)

        pos = atoms.get_positions()
        vx = np.array([1, 0, 0])
        vy = np.array([0, 1, 0])
        vsurf0x = pos[surf0x] - pos[m]
        vsurf0x[2] = 0
        vsurf0x /= np.linalg.norm(vsurf0x)
        vsurf0y = pos[surf0y] - pos[m]
        vsurf0y[2] = 0
        vsurf0y /= np.linalg.norm(vsurf0y)
        x_angle = np.vdot(vx, vsurf0x)
        x_angle = np.arccos(x_angle)
        y_angle = np.vdot(vy, vsurf0y)
        y_angle = np.arccos(y_angle)

        angle = x_angle/2 + y_angle/2

        atoms.rotate('z', -angle, rotate_cell=True)

        # Then perform the calculation
        with jasp('{name}/{struct}/bare-dos'.format(**locals()), atoms=atoms,
                  xc='PBE', lreal=False,
                  encut=500, kpts=(7, 7, 1),
                  ispin=1, lorbit=11,
                  ibrion=-1, ediff=1e-6,
                  npar=2, nsim=2, lplane=True, lscalu=False,
                  lwave=False) as calc:
            try:
                calc.calculate()
            except (VaspSubmitted, VaspRunning, VaspQueued):
                pass
```

6 Storage of all data for analysis into data.json

The script below stores all of the data necessary for performing the full analysis in a single JSON file, which we name **data.json** and have attached to the supporting information. The organization of **data.json** is as follows.

```
{
    compound:
    {
        struct:
        {
            "strain": frac,
            "vac": E_vac_O,
            "ads": E_ads_O,
            "OER": {"O": E_ads_O, "OH": E_ads_OH, "OOH": E_ads_OOH},
            "DOS": {"E": energies, "eg": eg_dos, "t2g": t2g_dos},
            "bulk":
            {
                "ground":{"pos":pos, "cell":cell, "syms":syms},
                "vac"   :{"pos":pos, "cell":cell, "syms":syms},
            },
            "surf":
            {
                "bare":{"pos":pos, "cell":cell, "syms":syms},
                "O"   :{"pos":pos, "cell":cell, "syms":syms},
                "OH"  :{"pos":pos, "cell":cell, "syms":syms},
                "OOH" :{"pos":pos, "cell":cell, "syms":syms},
            },

            // The following data only exist if the structure
            // we are looking at is rutile
            strain:
            {
                "vac": E_vac_O,
                "ads": E_ads_O,
                "DOS": {"E": energies, "eg": eg_dos, "t2g": t2g_dos},
                "bulk":
                {
                    "ground":{"pos":pos, "cell":cell, "syms":syms},
                    "vac"   :{"pos":pos, "cell":cell, "syms":syms},
                },
                "surf":
                {
                    "bare":{"pos":pos, "cell":cell, "syms":syms},
                    "ads" :{"pos":pos, "cell":cell, "syms":syms},
                },
```

```
41
42              },
43          },
44      },
45  }
```

The above JSON data structure displays the data contained for a single MO_2 (M=Ru, Rh, Ir, Pt) compound and structure (anatase, brookite, columbite, pyrite, rutile). The **data.json** file is loaded as a single dictionary object which will have four immediate child dictionary objects given by the variable `compound` key and will have values `"RuO2"`, `"RhO2"`, `"IrO2"`, and `"PtO2"`. Each `compound` dictionary will in turn have five child dictionary objects named by the key variable `struct` that will each correspond to the given structure. The five string names will be `"anatase"`, `"brookite"`, `"columbite"`, `"pyrite"`, and `"rutile"`.

Each `struct` dictionary in a given `compound` will contain data that fully describes its chemical properties. The immediate child objects within `struct` dictionary that are single variables are given by the keys `"strain"`, `"vac"`, and `"ads"`. `"strain"` will hold the variable `frac`, which is the relative volume of that specific `compound` and `struct` pair with respect to the the volume of `compound` in the equilibrium rutile phase. `"vac"` and `"ads"` contain the vacancy formation (`E_vac_O`) and adsorption energies (`E_ads_O`) of oxygen in the bulk and on the surface, respectively. Both of these values are calculated with respect to gaseous oxygen with the O_2 correction found by the Wolverton group.[5]

The `struct` dictionary contains a dictionary labeled `"OER"`. The `"OER"` dictionary contains the adsorption energies of O (`E_ads_O`), OH (`E_ads_OH`), and OOH (`E_ads_OOH`) with respect to standard hydrogen electrode (SHE) with included zero-point energies calculated in previous work.[6] The keys that refer to these adsorption energies is `"O"`, `"OH"`, and `"OOH"`.

The `struct` dictionary also contains a dictionary labeled `"DOS"`. This dictionary contains the necessary information to construct the density of states of the e_g and t_{2g} orbitals of the metal ion at the adsorption site of the bare surface. The three variables in each `"DOS"` dictionary are given by the `"E"`, `"eg"`, and `"t2g"` keys. The `"E"` key returns a list of energy

values `energies` at which the e_g and t_{2g} DOS will be given. These energies will be between -10 eV and 5 eV with respect to the fermi level of that specific system. The `"eg"` and `"t2g"` keys contain a list of densities (`eg_dos` and `t2g_dos`) that corresponds to the `energies` variable. Plotting a figure with `energies` on the x-axis and either `eg_dos` and `t2g_dos` on the y-axis will return a plot of the density of states.

The structural data for each `compound` and `struct` pair will be given in the `"bulk"` and `"surf"` dictionaries. The `"bulk"` dictionary will contain two dictionary objects with keys `"ground"` and `"vac"`, each containing three variables that fully describe the crystal structure of the ground state bulk structure and the bulk structure at an oxygen vacancy concentration of 6.25%, respectively. The three variables are three list objects of the positions (`pos`), unit cell (`cell`) and chemical symbols (`symbols`).

The `"surf"` dictionary inside the specific `compound` and `struct` pair will contain the complete crystal structure of the bare slab and slab with adsorbates O, OH, and OOH. Each specific `"surf"` will have four dictionaries titled `"bare"`, `"O"`, `"OH"`, and `"OOH"`, and each of these dictionaries will have a list of positions, the cell, and atomic symbols.

In addition, any `compound` where the `struct` dictionary is `"rutile"` will also contain data on strained rutile. The keys for the different amounts of strain for each rutile `compound` are `"-0.15"`, `"-0.10"`, `"-0.05"`, `"0.05"`, `"0.10"`, and `"0.15"`, and these will be the keys for dictionaries that contain the data. In each `strain` dictionary, a similar data structure exists, with information about the vacancy formation energy (`"vac"`), the adsorption energy of oxygen (`"ads"`), the density of states of metal ion at the adsorption site (`"DOS"`), and structural information of both the bulk and surface slabs (`"bulk"` and `"surf"`). The format of each of these keys is the same as above. One difference is that the `"OER"` data is missing, since this was not used in our analysis.

The script below creates the **data.json** file after all calculations are done. In order to run this script, an additional python module is used and is stored in a file named **dos_data.py**. This file can be found in the Required modules section in Section 8.

```python
import numpy as np
from dos_data import DOS_data
from jasp import *
import json

data = {}

names = ['RuO2', 'RhO2', 'IrO2', 'PtO2']
structs = ['rutile', 'anatase', 'brookite', 'columbite', 'pyrite']

O2 = -9.849707 + 1.198

H2 = -6.762966
H2O = -14.216243

site_ind = {'anatase': 31,
            'brookite': 31,
            'columbite': 38,
            'pyrite': 14,
            'rutile': 44}

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
    '''

    if bare == None:
        return 'Unstable', 'Unstable', 'Unstable'
    H2 = -6.762966 # From H2 in a box
    H2O = -14.216243 # From H2O in a box

    if OH == None:
        OH_ads = 'Unstable'
    else:
        OH_ads = OH - bare - (H2O - 0.5 * H2) + 0.35
    if O == None:
        O_ads = 'Unstable'
    else:
```

```
        O_ads = O - bare - (H2O - H2) + 0.05
    if OOH == None:
        OOH_ads = 'Unstable'
    else:
        OOH_ads = OOH - bare - (2*H2O - 3./2. * H2) + 0.4

    return OH_ads, O_ads, OOH_ads

for name in names:
    if name not in data:
        data[name] = {}
    for struct in structs:
        if struct not in data[name]:
            data[name][struct] = {}
        # First calculate and store the vacancy formation energy
        # We also need the relative strain with respect to rutile
        with jasp('{name}/{struct}/bulk-vacancy/ground'.format(**locals())) as calc:
            atoms = calc.get_atoms()
            data[name][struct]['bulk'] = {}
            data[name][struct]['bulk']['ground'] = {}
            data[name][struct]['bulk']['ground']['pos'] = atoms.get_positions().tolist()
            data[name][struct]['bulk']['ground']['cell'] = atoms.get_cell().tolist()
            data[name][struct]['bulk']['ground']['syms'] = atoms.get_chemical_symbols()
            n0 = len(atoms)
            if struct == 'rutile':
                v0 = atoms.get_volume() / len(atoms)
                v = atoms.get_volume() / len(atoms)
            else:
                v = atoms.get_volume() / len(atoms)
            data[name][struct]['strain'] = v/v0 - 1.0
            E0 = atoms.get_potential_energy()

        with jasp('{name}/{struct}/bulk-vacancy/0.9375'.format(**locals())) as calc:
            atoms = calc.get_atoms()
            data[name][struct]['bulk'] = {}
            data[name][struct]['bulk']['vac'] = {}
            data[name][struct]['bulk']['vac']['pos'] = atoms.get_positions().tolist()
            data[name][struct]['bulk']['vac']['cell'] = atoms.get_cell().tolist()
            data[name][struct]['bulk']['vac']['syms'] = atoms.get_chemical_symbols()
            n = len(atoms)
            E = atoms.get_potential_energy()
```

```
83
84          E_vac = E + O2/2.0 - (n + 1) / n0 * E0
85          data[name][struct]['vac'] = E_vac
86
87          # Now store adsorption energies of O, OH, and OOH
88          with jasp('{name}/{struct}/bare'.format(**locals())) as calc:
89              if (not os.path.isfile('OUTCAR') or not calc.read_convergence()):
90                  bare, OH, O, OOH = None, None, None, None
91              else:
92                  atoms = calc.get_atoms()
93                  data[name][struct]['surf'] = {}
94                  data[name][struct]['surf']['bare'] = {}
95                  data[name][struct]['surf']['bare']['pos'] = atoms.get_positions().tolist()
96                  data[name][struct]['surf']['bare']['cell'] = atoms.get_cell().tolist()
97                  data[name][struct]['surf']['bare']['syms'] = atoms.get_chemical_symbols()
98                  bare = atoms.get_potential_energy()
99
100         if bare != None:
101             with jasp('{name}/{struct}/O'.format(**locals())) as calc:
102                 if (not os.path.isfile('OUTCAR') or not calc.read_convergence()):
103                     O = None
104                 else:
105                     atoms = calc.get_atoms()
106                     data[name][struct]['surf'] = {}
107                     data[name][struct]['surf']['O'] = {}
108                     data[name][struct]['surf']['O']['pos'] = atoms.get_positions().tolist()
109                     data[name][struct]['surf']['O']['cell'] = atoms.get_cell().tolist()
110                     data[name][struct]['surf']['O']['syms'] = atoms.get_chemical_symbols()
111                     O = atoms.get_potential_energy()
112
113             with jasp('{name}/{struct}/OH'.format(**locals())) as calc:
114                 if (not os.path.isfile('OUTCAR') or not calc.read_convergence()):
115                     OH = None
116                 else:
117                     atoms = calc.get_atoms()
118                     data[name][struct]['surf'] = {}
119                     data[name][struct]['surf']['OH'] = {}
120                     data[name][struct]['surf']['OH']['pos'] = atoms.get_positions().tolist()
121                     data[name][struct]['surf']['OH']['cell'] = atoms.get_cell().tolist()
122                     data[name][struct]['surf']['OH']['syms'] = atoms.get_chemical_symbols()
123                     OH = atoms.get_potential_energy()
```

```
124
125             with jasp('{name}/{struct}/OOH'.format(**locals())) as calc:
126                 if (not os.path.isfile('OUTCAR') or not calc.read_convergence()):
127                     OOH = None
128                 else:
129                     atoms = calc.get_atoms()
130                     data[name][struct]['surf'] = {}
131                     data[name][struct]['surf']['OOH'] = {}
132                     data[name][struct]['surf']['OOH']['pos'] = atoms.get_positions().tolist()
133                     data[name][struct]['surf']['OOH']['cell'] = atoms.get_cell().tolist()
134                     data[name][struct]['surf']['OOH']['syms'] = atoms.get_chemical_symbols()
135                     OOH = atoms.get_potential_energy()
136
137         # We also want to calculate the adsorption energy of O referenced to O2 as well
138         if type(O) == float:
139             data[name][struct]['ads'] = O - bare - O2/2.0
140         else:
141             data[name][struct]['ads'] = np.nan
142
143         # Now calculate OH, O, and OOH adsorption energies with zero-point contributions
144         OH, O, OOH = ads_energy(bare, OH, O, OOH)
145         data[name][struct]['OER'] = {}
146         for ads_name, E_ads in [['O', O], ['OH', OH], ['OOH', OOH]]:
147             if type(E_ads) == float:
148                 data[name][struct]['OER'][ads_name] = E_ads
149             else:
150                 data[name][struct]['OER'][ads_name] = np.nan
151
152         # Now store atom projected density of states at the adsorption site
153         data[name][struct]['DOS'] = {}
154         if struct in ['brookite', 'columbite', 'pyrite']:
155             wkdir = '{name}/{struct}/bare-dos'.format(**locals())
156         else:
157             wkdir = '{name}/{struct}/bare'.format(**locals())
158
159         d_index = range(4, 9)
160         if struct == 'rutile':
161             t2g_index = [5, 7, 8]
162             eg_index = [6, 4]
163         else:
164             t2g_index = [5, 7, 4]
```

```
            eg_index = [6, 8]
        with jasp(wkdir) as calc:
            dos_data = DOS_data(calc)
            energies = dos_data.energies
            occupied = dos_data.occupied
            t2g, t2g_width, t2g_center = dos_data.get_property(site_ind[struct], t2g_index)
            d, d_width, d_center = dos_data.get_property(site_ind[struct], d_index)
            eg, eg_width, eg_center = dos_data.get_property(site_ind[struct], eg_index)

        data[name][struct]['DOS'] = {}
        data[name][struct]['DOS']['E'] = list(energies)
        data[name][struct]['DOS']['eg'] = list(eg)
        data[name][struct]['DOS']['t2g'] = list(t2g)

        # If we are looking at rutile, then we also need to store strained calculations
        if struct != 'rutile':
            continue

        for f in [-0.15, -0.10, -0.05, 0.05, 0.10, 0.15]:
            strain = '{f:1.2f}'.format(**locals())
            data[name][struct][strain] = {}
            # First store the vacancy formation energy
            with jasp('{name}/{struct}/bulk-vacancy/ground{f:+1.2f}'.format(**locals())) as calc:
                atoms = calc.get_atoms()
                E0 = atoms.get_potential_energy() * 4
            data[name][struct][strain]['bulk'] = {}
            data[name][struct][strain]['bulk']['ground'] = {}
            data[name][struct][strain]['bulk']['ground']['pos'] = atoms.get_positions().tolist()
            data[name][struct][strain]['bulk']['ground']['cell'] = atoms.get_cell().tolist()
            data[name][struct][strain]['bulk']['ground']['syms'] = atoms.get_chemical_symbols()

            with jasp('{name}/{struct}/bulk-vacancy/0.9375{f:+1.2f}'.format(**locals())) as calc:
                atoms = calc.get_atoms()
                E = atoms.get_potential_energy()
            data[name][struct][strain]['bulk'] = {}
            data[name][struct][strain]['bulk']['vac'] = {}
            data[name][struct][strain]['bulk']['vac']['pos'] = atoms.get_positions().tolist()
            data[name][struct][strain]['bulk']['vac']['cell'] = atoms.get_cell().tolist()
            data[name][struct][strain]['bulk']['vac']['syms'] = atoms.get_chemical_symbols()
```

```
206             data[name][struct][strain]['vac'] = E + O2/2.0 - E0
207
208             # Now store the adsorption energy of O
209             with jasp('{name}/{struct}/bare{f:+1.2f}'.format(**locals())) as calc:
210                 atoms = calc.get_atoms()
211                 data[name][struct][strain]['surf'] = {}
212                 data[name][struct][strain]['surf']['bare'] = {}
213                 data[name][struct][strain]['surf']['bare']['pos'] = atoms.get_positions().tolist()
214                 data[name][struct][strain]['surf']['bare']['cell'] = atoms.get_cell().tolist()
215                 data[name][struct][strain]['surf']['bare']['syms'] = atoms.get_chemical_symbols()
216                 E_slab = atoms.get_potential_energy()
217
218             with jasp('{name}/{struct}/O{f:+1.2f}'.format(**locals())) as calc:
219                 atoms = calc.get_atoms()
220                 data[name][struct][strain]['surf'] = {}
221                 data[name][struct][strain]['surf']['ads'] = {}
222                 data[name][struct][strain]['surf']['ads']['pos'] = atoms.get_positions().tolist()
223                 data[name][struct][strain]['surf']['ads']['cell'] = atoms.get_cell().tolist()
224                 data[name][struct][strain]['surf']['ads']['syms'] = atoms.get_chemical_symbols()
225                 E_ads = atoms.get_potential_energy()
226
227             data[name][struct][strain]['ads'] = E_ads - (E_slab + O2/2)
228
229             # Finally store the DOS of the strained surfaces
230             data[name][struct][strain]['DOS'] = {}
231             wkdir = '{name}/rutile/bare{f:+1.2f}'.format(**locals())
232             with jasp(wkdir) as calc:
233                 t2g_index = [5, 7, 8]
234                 eg_index = [6, 4]
235                 dos_data = DOS_data(calc)
236                 energies = dos_data.energies
237                 occupied = dos_data.occupied
238                 t2g, t2g_width, t2g_center = dos_data.get_property(site_ind[struct], t2g_index)
239                 d, d_width, d_center = dos_data.get_property(site_ind[struct], d_index)
240                 eg, eg_width, eg_center = dos_data.get_property(site_ind[struct], eg_index)
241
242             data[name][struct][strain]['DOS']['E'] = list(energies)
243             data[name][struct][strain]['DOS']['eg'] = list(eg)
244             data[name][struct][strain]['DOS']['t2g'] = list(t2g)
245
246 with open('data.json', 'w') as f:
```

```
247     json.dump(data, f)
```

7 Analysis of data and construction of figures

The section below contains the code used to construct Figure 1-4 in the manuscript. Figure 1 is just visualization of the crystal structures of all polymorphs, whereas Figure 2-4 are plots of the data found in the **data.json** file. All scripts are written in python and do not require installation of VASP or jasp to construct.

7.1 FIG1: Visualization of polymorphs

In order to show the octahedral coordination of all metal ions in polymorphs, we used VESTA.[7] To import the crystal structures into VESTA, we first produce Crystallographic Information Framework (CIF) files from the crystal structures saved in the `ase_addons` module. After adding the octahedral shapes, we then saved them as png images, which we then put together in a single figure using python. While the process of using VESTA was difficult to illustrate, the scripts for generating the CIF files and putting VESTA images together were written using python and shown in two sectons below.

7.1.1 Creating CIF files for visualizing of polymorphs in VESTA

1. Anatase

```
1  from ase import Atom, Atoms
2  from ase.io import write
3  import numpy as np
4  from ase.visualize import view
5
6  B='Ti'
7  X='O'
8  a=3.7842
9  c=2*4.7573
```

```
z=0.0831

a1 = a*np.array([1.0, 0.0, 0.0])
a2 = a*np.array([0.0, 1.0, 0.0])
a3 = np.array([0.5*a, 0.5*a, 0.5*c])

b3 = 2*a3 - a1 - a2

atoms = Atoms([Atom(B, -0.125*a1 + 0.625*a2 + 0.25*a3),
               Atom(B,  0.125*a1 + 0.375*a2 + 0.75*a3),
               Atom(X, -z*a1 + (0.25-z)*a2 + 2*z*a3),
               Atom(X, -(0.25+z)*a1 + (0.5-z)*a2 + (0.5+2*z)*a3),
               Atom(X, z*a1 - (0.25 - z)*a2 + (1-2*z)*a3),
               Atom(X, (0.25 + z)*a1 + (0.5 + z)*a2 + (0.5-2*z)*a3),
               Atom(B, -0.125*a1 + 0.625*a2 + 0.25*a3 + a3 - a2),
               Atom(B,  0.125*a1 + 0.375*a2 + 0.75*a3 + a3 - a2 - a1),
               Atom(X, -z*a1 + (0.25-z)*a2 + 2.*z*a3 + a3),
               Atom(X, -(0.25+z)*a1 + (0.5-z)*a2 + (0.5+2*z)*a3 + a3 - a2),
               Atom(X, z*a1 - (0.25 - z)*a2 + (1-2*z)*a3 + a3 - a1),
               Atom(X, (0.25 + z)*a1 + (0.5 + z)*a2 + (0.5-2*z)*a3 + a3 - a2 - a1)],
               cell=[a1,a2,b3])
view(atoms)
write('data/anatase.cif', atoms)
```

2. Columbite

```
from ase import Atom, Atoms
from ase.io import write
import numpy as np
from ase.visualize import view

B='Ti'
X='O'
a=4.947
b=5.951
c=5.497
mags=[0.5, 0],
u=0.178
x=0.276
y=0.41
z=0.425
```

```
vacuum=10

b1 = np.array([(a**2 + c**2)**0.5, 0, 0])
b2 = b*np.array([0.0, 1.0, 0.0])
b3 = np.array([(a**2 - c**2) / (a**2 + c**2)**0.5, 0, 2*a*c/(a**2 + c**2)**0.5])

atoms = Atoms([Atom(B, 7*b1/8 + b2*u + b3/8),
               Atom(B, 5*b1/8 + b2*(-u + 1) + 3*b3/8),
               Atom(B, b1/8 + b2*(u + 0.5) + 3*b3/8),
               Atom(B, 7*b1/8 + b2*(-u + 0.5) + 5*b3/8),
               Atom(X, b1*(x/2 - z/2 + 1) + b2*y + b3*(x/2 + z/2)),
               Atom(X, b1*(-x/2 + z/2) + b2*(-y + 1) + b3*(-x/2 - z/2 + 1)),
               Atom(X, b1*(-x/2 - z/2 + 1) + b2*(-y + 0.5) + b3*(-x/2 + z/2 + 0.5)),
               Atom(X, b1*(x/2 + z/2) + b2*(y + 0.5) + b3*(x/2 - z/2 + 0.5)),
               Atom(X, b1*(x/2 + z/2 - 0.25) + b2*(-y + 0.5) + b3*(x/2 - z/2 + 0.75)),
               Atom(X, b1*(-x/2 - z/2 + 1.25) + b2*(y + 0.5) + b3*(-x/2 + z/2 + 0.25)),
               Atom(X, b1*(-x/2 + z/2 + 0.25) + b2*y + b3*(-x/2 - z/2 + 0.75)),
               Atom(X, b1*(x/2 - z/2 + 0.75) + b2*(-y + 1) + b3*(x/2 + z/2 + 0.25)),
               Atom(B, 3*b1/8 + b2*u + 5*b3/8),
               Atom(B, b1/8 + b2*(-u + 1) + 7*b3/8),
               Atom(B, 5*b1/8 + b2*(u + 0.5) + 7*b3/8),
               Atom(B, 3*b1/8 + b2*(-u + 0.5) + b3/8),
               Atom(X, b1*(x/2 - z/2 + 0.5) + b2*y + b3*(x/2 + z/2 + 0.5)),
               Atom(X, b1*(-x/2 + z/2 + 0.5) + b2*(-y + 1) + b3*(-x/2 - z/2 + 0.5)),
               Atom(X, b1*(-x/2 - z/2 + 0.5) + b2*(-y + 0.5) + b3*(-x/2 + z/2)),
               Atom(X, b1*(x/2 + z/2 + 0.5) + b2*(y + 0.5) + b3*(x/2 - z/2 + 1)),
               Atom(X, b1*(x/2 + z/2 + 0.25) + b2*(-y + 0.5) + b3*(x/2 - z/2 + 0.25)),
               Atom(X, b1*(-x/2 - z/2 + 0.75) + b2*(y + 0.5) + b3*(-x/2 + z/2 + 0.75)),
               Atom(X, b1*(-x/2 + z/2 + 0.75) + b2*y + b3*(-x/2 - z/2 + 1.25)),
               Atom(X, b1*(x/2 - z/2 + 0.25) + b2*(-y + 1) + b3*(x/2 + z/2 - 0.25))],
              cell = [b1, b2, b3])

view(atoms)
write('data/columbite.cif', atoms)
```

3. Brookite

```
from ase import Atom, Atoms
from ase.io import write
import numpy as np
from ase.visualize import view
```

```
5
6  B='Ti'
7  X='O'
8  a=9.16
9  b=5.43
10 c=5.13
11 x1=0.12
12 x2=0.01
13 x3=0.23
14 y1=0.11
15 y2=0.15
16 y3=0.10
17 z1=-0.12
18 z2=0.18
19 z3=-0.46
20
21 h1 = np.array([0.0, (a**2 + b**2)**0.5, 0.0])
22 h2 = np.array([0, (a**2 - b**2) / (a**2 + b**2)**0.5, 2*a*b/(a**2 + b**2)**0.5])
23 h3 = c * np.array([1.0, 0.0, 0.0])
24
25 atoms = Atoms([Atom(B, h1*(x1/2 - y1/2) + h2*(x1/2 + y1/2) + h3*(z1 + 1)),
26                Atom(B, h1*(x1/2 + y1/2) + h2*(x1/2 - y1/2 + 0.5) - h3*z1),
27                Atom(B, h1*(-x1/2 - y1/2 + 0.75) + h2*(-x1/2 + y1/2 + 0.25) + h3*(-z1 + 0.5)),
28                Atom(B, h1*(-x1/2 + y1/2 + 0.25) + h2*(-x1/2 - y1/2 + 0.25) + h3*(z1 + 0.5)),
29                Atom(B, h1*(-x1/2 + y1/2 + 1) + h2*(-x1/2 - y1/2 + 1) - h3*z1),
30                Atom(B, h1*(-x1/2 - y1/2 + 1) + h2*(-x1/2 + y1/2 + 0.5) + h3*(z1 + 1)),
31                Atom(B, h1*(x1/2 + y1/2 + 0.75) + h2*(x1/2 - y1/2 + 0.25) + h3*(z1 + 0.5)),
32                Atom(B, h1*(x1/2 - y1/2 + 0.25) + h2*(x1/2 + y1/2 + 0.25) + h3*(-z1 + 0.5)),
33                Atom(X, h1*(x2/2 - y2/2 + 1) + h2*(x2/2 + y2/2) + h3*z2),
34                Atom(X, h1*(x2/2 + y2/2) + h2*(x2/2 - y2/2 + 0.5) + h3*(-z2 + 1)),
35                Atom(X, h1*(-x2/2 - y2/2 + 0.75) + h2*(-x2/2 + y2/2 + 0.25) + h3*(-z2 + 0.5)),
36                Atom(X, h1*(-x2/2 + y2/2 + 0.25) + h2*(-x2/2 - y2/2 + 0.25) + h3*(z2 + 0.5)),
37                Atom(X, h1*(-x2/2 + y2/2) + h2*(-x2/2 - y2/2 + 1) + h3*(-z2 + 1)),
38                Atom(X, h1*(-x2/2 - y2/2 + 1) + h2*(-x2/2 + y2/2 + 0.5) + h3*z2),
39                Atom(X, h1*(x2/2 + y2/2 + 0.75) + h2*(x2/2 - y2/2 + 0.25) + h3*(z2 + 0.5)),
40                Atom(X, h1*(x2/2 - y2/2 + 0.25) + h2*(x2/2 + y2/2 + 0.25) + h3*(-z2 + 0.5)),
41                Atom(X, h1*(x3/2 - y3/2) + h2*(x3/2 + y3/2) + h3*(z3 + 1)),
42                Atom(X, h1*(x3/2 + y3/2) + h2*(x3/2 - y3/2 + 0.5) - h3*z3),
43                Atom(X, h1*(-x3/2 - y3/2 + 0.75) + h2*(-x3/2 + y3/2 + 0.25) + h3*(-z3 + 0.5)),
44                Atom(X, h1*(-x3/2 + y3/2 + 0.25) + h2*(-x3/2 - y3/2 + 0.25) + h3*(z3 + 0.5)),
45                Atom(X, h1*(-x3/2 + y3/2 + 1) + h2*(-x3/2 - y3/2 + 1) - h3*z3),
```

```
                Atom(X, h1*(-x3/2 - y3/2 + 1) + h2*(-x3/2 + y3/2 + 0.5) + h3*(z3 + 1)),
                Atom(X, h1*(x3/2 + y3/2 + 0.75) + h2*(x3/2 - y3/2 + 0.25) + h3*(z3 + 0.5)),
                Atom(X, h1*(x3/2 - y3/2 + 0.25) + h2*(x3/2 + y3/2 + 0.25) + h3*(-z3 + 0.5)),
                Atom(B, h1*(x1/2 - y1/2 + 0.5) + h2*(x1/2 + y1/2 + 0.5) + h3*(z1 + 1)),
                Atom(B, h1*(x1/2 + y1/2 + 0.5) + h2*(x1/2 - y1/2) - h3*z1),
                Atom(B, h1*(-x1/2 - y1/2 + 0.25) + h2*(-x1/2 + y1/2 + 0.75) + h3*(-z1 + 0.5)),
                Atom(B, h1*(-x1/2 + y1/2 + 0.75) + h2*(-x1/2 - y1/2 + 0.75) + h3*(z1 + 0.5)),
                Atom(B, h1*(-x1/2 + y1/2 + 0.5) + h2*(-x1/2 - y1/2 + 0.5) - h3*z1),
                Atom(B, h1*(-x1/2 - y1/2 + 0.5) + h2*(-x1/2 + y1/2 + 1) + h3*(z1 + 1)),
                Atom(B, h1*(x1/2 + y1/2 + 0.25) + h2*(x1/2 - y1/2 + 0.75) + h3*(z1 + 0.5)),
                Atom(B, h1*(x1/2 - y1/2 + 0.75) + h2*(x1/2 + y1/2 + 0.75) + h3*(-z1 + 0.5)),
                Atom(X, h1*(x2/2 - y2/2 + 0.5) + h2*(x2/2 + y2/2 + 0.5) + h3*z2),
                Atom(X, h1*(x2/2 + y2/2 + 0.5) + h2*(x2/2 - y2/2 + 1) + h3*(-z2 + 1)),
                Atom(X, h1*(-x2/2 - y2/2 + 0.25) + h2*(-x2/2 + y2/2 + 0.75) + h3*(-z2 + 0.5)),
                Atom(X, h1*(-x2/2 + y2/2 + 0.75) + h2*(-x2/2 - y2/2 + 0.75) + h3*(z2 + 0.5)),
                Atom(X, h1*(-x2/2 + y2/2 + 0.5) + h2*(-x2/2 - y2/2 + 0.5) + h3*(-z2 + 1)),
                Atom(X, h1*(-x2/2 - y2/2 + 0.5) + h2*(-x2/2 + y2/2) + h3*z2),
                Atom(X, h1*(x2/2 + y2/2 + 0.25) + h2*(x2/2 - y2/2 + 0.75) + h3*(z2 + 0.5)),
                Atom(X, h1*(x2/2 - y2/2 + 0.75) + h2*(x2/2 + y2/2 + 0.75) + h3*(-z2 + 0.5)),
                Atom(X, h1*(x3/2 - y3/2 + 0.5) + h2*(x3/2 + y3/2 + 0.5) + h3*(z3 + 1)),
                Atom(X, h1*(x3/2 + y3/2 + 0.5) + h2*(x3/2 - y3/2) - h3*z3),
                Atom(X, h1*(-x3/2 - y3/2 + 0.25) + h2*(-x3/2 + y3/2 + 0.75) + h3*(-z3 + 0.5)),
                Atom(X, h1*(-x3/2 + y3/2 + 0.75) + h2*(-x3/2 - y3/2 + 0.75) + h3*(z3 + 0.5)),
                Atom(X, h1*(-x3/2 + y3/2 + 0.5) + h2*(-x3/2 - y3/2 + 0.5) - h3*z3),
                Atom(X, h1*(-x3/2 - y3/2 + 0.5) + h2*(-x3/2 + y3/2 + 1) + h3*(z3 + 1)),
                Atom(X, h1*(x3/2 + y3/2 + 0.25) + h2*(x3/2 - y3/2 + 0.75) + h3*(z3 + 0.5)),
                Atom(X, h1*(x3/2 - y3/2 + 0.75) + h2*(x3/2 + y3/2 + 0.75) + h3*(-z3 + 0.5))],
               cell=[h3,h1,h2])

view(atoms)
write('data/brookite.cif', atoms)
```

4. Pyrite

```
from ase import Atom, Atoms
from ase.io import write
import numpy as np
from ase.visualize import view

B='Ti'
X='O'
```

```
a=5.407
u=2.0871/5.407

a1 = a*np.array([1.0, 0.0, 0.0])
a2 = a*np.array([0.0, 1.0, 0.0])
a3 = a*np.array([0.0, 0.0, 1.0])

atoms = Atoms([Atom(B, 0.25*a3),
               Atom(B, 0.5*a2 + 0.75*a3),
               Atom(B, 0.5*a1 + 0.75*a3),
               Atom(B, 0.5*a1 + 0.5*a2 + 0.25*a3),
               Atom(X, u*a1 + u*a2 + (u + 0.25)*a3),
               Atom(X, (1 - u)*a1 + (1 - u)*a2 + (1.25 - u)*a3),
               Atom(X, (0.5 + u)*a1 + (0.5 - u)*a2 + (1.25 - u)*a3),
               Atom(X, (0.5 - u)*a1 + (u - 0.5)*a2 + (u + 0.25)*a3),
               Atom(X, (1 - u)*a1 + (0.5 + u)*a2 + (0.75 - u)*a3),
               Atom(X, u*a1 + (0.5 - u)*a2 + (u - 0.25)*a3),
               Atom(X, (0.5 - u)*a1 + (1 - u)*a2 + (u - 0.25)*a3),
               Atom(X, (u + 0.5)*a1 + u*a2 + (0.75 - u)*a3)],
               cell=[a1,a2,a3])

view(atoms)
write('data/pyrite.cif', atoms)
```

5. Rutile

```
from ase import Atom, Atoms
from ase.io import write
import numpy as np
from ase.visualize import view
from ase_addons.surfaces import rutile110

atoms = rutile110(['Ti', 'O'], a=4.242, c=3.064, u=0.309)
atoms.center()

view(atoms)
write('data/rutile.cif', atoms)
```

7.1.2 Code for creating Figure 1 of the manuscript

The code below takes png files generated from VESTA and puts them together.

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

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

# Add the anatase crystal structure
ax_ana = fig.add_axes([0.05, 0.5, 0.3, 0.5], frameon=False)
ax_ana.set_xticks([])
ax_ana.set_yticks([])
ana = mpimg.imread('data/anatase.png')
ax_ana.imshow(ana)

# Add the columbite crystal structure
ax_col = fig.add_axes([0.01, -0.05, 0.4, 0.4], frameon=False)
ax_col.set_xticks([])
ax_col.set_yticks([])
col = mpimg.imread('data/columbite.png')
ax_col.imshow(col)

# Add the pyrite crystal structure
ax_pyr = fig.add_axes([0.55, -0.1, 0.45, 0.5], frameon=False)
ax_pyr.set_xticks([])
ax_pyr.set_yticks([])
pyr = mpimg.imread('data/pyrite.png')
ax_pyr.imshow(pyr)

# Add the brookite crystal structure
ax_brk = fig.add_axes([0.5, 0.55, 0.45, 0.45], frameon=False)
ax_brk.set_xticks([])
ax_brk.set_yticks([])
brk = mpimg.imread('data/brookite.png')
ax_brk.imshow(brk)

# Add the rutile crystal structure
ax_rut = fig.add_axes([0.3, 0.25, 0.4, 0.4], frameon=False)
ax_rut.set_xticks([])
ax_rut.set_yticks([])
```

```
rut = mpimg.imread('data/rutile.png')
ax_rut.imshow(rut)

# Now add the titles
fig.text(0.05, 0.93, 'Anatase (001)')
fig.text(0.6, 0.93, 'Brookite (110)')
fig.text(0.5, 0.6, 'Rutile (110)', ha='center')
fig.text(0.05, 0.28, 'Columbite (101)')
fig.text(0.65, 0.28, 'Pyrite (001)')

fig.savefig('images/FIG1.png', dpi=600)
fig.savefig('images/FIG1.eps', dpi=600)
plt.show()
```

7.2 FIG2: Analysis of chemical properties on polymorphs and strained rutile with respect to equilibrium rutile

The script below is used to construct FIG2 of the manuscript. All data is read completely from the **data.json** file attached to the supporting information PDF.

```
import json
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from copy import copy

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

with open('data/data.json', 'r') as f:
    data = json.load(f)

names = ['IrO2', 'RhO2', 'RuO2', 'PtO2']
structs = ['anatase', 'columbite', 'pyrite', 'brookite']

OH, O, OOH = [], [], []

m_dict = {'anatase':'s',
```

```
            'columbite':'^',
            'pyrite':'v',
            'brookite':'D'}

c_dict = {'IrO2':'b',
            'RhO2':'r',
            'RuO2':'gray',
            'PtO2':'k'}

all_Os, vacs = [], []
OH, O, OOH = [], [], []

axes_loc = [[0.25, 0.345, 0.33, 0.3],
             [0.62, 0.345, 0.33, 0.3],
             [0.25, 0.02, 0.33, 0.3],
             [0.62, 0.02, 0.33, 0.3]]

title_loc = {'IrO2':[2, 0.3],
              'RhO2':[2, 0.3],
              'RuO2':[2, 0.3],
              'PtO2':[2, 0.3]}

fig = plt.figure(1, [3.2, 4.5])
ax1 = fig.add_axes([0.25, 0.57, 0.7, 0.38])
ax1.text(-0.16, 0.45, '(a)')

for name in names:
    # First collect the rutile data
    E_rut, S_rut = [], []
    for strain in [-0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15]:
        if strain == 0:
            E = data[name]['rutile']['ads']
            E0 = copy(E)
            E_rut.append(E)
        else:
            E_rut.append(data[name]['rutile']['{0:1.2f}'.format(strain)]['ads'])
        S_rut.append(strain)
    E_rut = np.array(E_rut) - E*np.ones(len(E_rut))
    ax1.plot(S_rut, E_rut, marker='o', c=c_dict[name], ls='--', fillstyle='none')

    # Now plot the polymorph data
```

```
60      for struct in ['anatase', 'brookite', 'columbite', 'pyrite']:
61          S = data[name][struct]['strain']
62          E = data[name][struct]['ads']
63          E -= E0
64          ax1.plot(S, E, marker=m_dict[struct], c=c_dict[name])
65
66  ax1.set_xlim(-0.175, 0.175)
67  ax1.set_ylim(-0.6, 0.6)
68  ax1.set_ylabel(r'$\Delta E_{ads}^{O} - \Delta E_{ads,rutile}^{O}$ (eV)')
69  ax1.set_xticks([-0.10, 0, 0.10])
70  ax1.set_xticklabels([])
71
72  ax2 = fig.add_axes([0.25, 0.15, 0.7, 0.38])
73  ax2.text(-0.16, 0.3, '(b)')
74
75  for name in names:
76      # First collect the rutile data
77      E_rut, S_rut = [], []
78      for strain in [-0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15]:
79          if strain == 0:
80              E = data[name]['rutile']['vac']
81              E0 = copy(E)
82              E_rut.append(E)
83          else:
84              E_rut.append(data[name]['rutile']['{0:1.2f}'.format(strain)]['vac'])
85          S_rut.append(strain)
86      E_rut = np.array(E_rut) - E*np.ones(len(E_rut))
87      ax2.plot(S_rut, E_rut, marker='o', c=c_dict[name], ls='--', fillstyle='none')
88
89      # Now plot the polymorph data
90      for struct in ['anatase', 'brookite', 'columbite', 'pyrite']:
91          S = data[name][struct]['strain']
92          E = data[name][struct]['vac']
93          E -= E0
94          ax2.plot(S, E, marker=m_dict[struct], c=c_dict[name])
95
96  ax2.set_xlim(-0.175, 0.175)
97  ax2.set_xlabel(r'Strain $(V - V^{eq}_{rutile})/V^{eq}_{rutile}$')
98  ax2.set_ylim(-4, 1)
99  ax2.set_ylabel(r'$\Delta E_{vac}^{O} - \Delta E_{vac,rutile}^{O}$ (eV)')
100 ax2.set_xticks([-0.10, 0, 0.10])
```

```
101
102 plt.savefig('images/FIG2.png', dpi=600)
103 plt.savefig('images/FIG2.eps', dpi=600)
104 plt.show()
```

7.3 FIG3: Interesting chemical properties of oxide polymorphs

The script below is used to construct FIG3 of the manuscript. All data is read completely from the **data.json** file attached to the supporting information PDF.

```
1  import json
2  import matplotlib
3  import matplotlib.pyplot as plt
4  import numpy as np
5  from math import isnan
6
7  matplotlib.rc('xtick', labelsize=10)
8  matplotlib.rc('ytick', labelsize=10)
9
10 with open('data/data.json', 'r') as f:
11     data = json.load(f)
12
13 fig = plt.figure(1, (3.2, 4.5))
14
15 m_dict = {'rutile':'o',
16           'anatase':'s',
17           'columbite':'^',
18           'pyrite':'v',
19           'brookite':'D'}
20
21 c_dict = {'IrO2':'b',
22           'RhO2':'r',
23           'RuO2':'gray',
24           'PtO2':'k'}
25
26 # First plot the correlation between vacancy formation and adsorption energy
27 ax1 = fig.add_axes([0.25, 0.6, 0.6, 0.35])
28 ax1.text(-2.5, 0.1, '(a)')
29 ax1.set_ylabel(r'$\Delta E_{ads}^{O}$ (eV)', size='small')
```

```
ax1.set_xlabel(r'$\Delta E_{vac}^{O}$ (eV)', size='small')
ax1.set_xlim(-3, 5)
ax1.set_xticks([-2, 0, 2, 4])
ax1.annotate('', xy=(2, -1), xytext=(-2, -1), arrowprops=dict(arrowstyle='->', color='k'))
ax1.text(-2.5, -0.87, 'Stability towards \npartial oxidation/reduction', size=9)

for name in ['IrO2', 'RhO2', 'RuO2', 'PtO2']:
    # First collect the rutile data
    E_ads, E_vac = [], []
    for strain in [-0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15]:
        if strain == 0:
            E = data[name]['rutile']['OER']['O']
            E = E - 0.05 + (-14.216243 - -6.762966) - (-9.849707 + 1.198)/2
            E_ads.append(E)
            E = data[name]['rutile']['vac']
            E_vac.append(E)
        else:
            E_ads.append(data[name]['rutile']['{0:1.2f}'.format(strain)]['ads'])
            E_vac.append(data[name]['rutile']['{0:1.2f}'.format(strain)]['vac'])

    # We also want to plot a linear fit of this relationship
    f = np.poly1d(np.polyfit(E_vac, E_ads, 1))
    E_vac_fit = np.linspace(min(E_vac) - 1, max(E_vac) + 1)
    ax1.plot(E_vac_fit, f(E_vac_fit), ls='--', c=c_dict[name])

    ax1.plot(E_vac, E_ads, marker='o', c=c_dict[name], ls='none', fillstyle='none')

    # Now plot the polymorph data
    for struct in ['anatase', 'brookite', 'columbite', 'pyrite']:
        E_vac = data[name][struct]['vac']
        E_ads = data[name][struct]['OER']['O']
        E_ads = E_ads - 0.05 + (-14.216243 - -6.762966) - (-9.849707 + 1.198)/2
        ax1.plot(E_vac, E_ads, marker=m_dict[struct], c=c_dict[name])

# Now plot the scaling relationships
OHs, Os, OOHs = [], [], []

ax2 = fig.add_axes([0.15, 0.12, 0.3, 0.35])
ax2.text(0.15, 5.2, '(b)')
ax2.set_xlabel(r'$\Delta E_{ads}^{OH}$ (eV)', size='small')
ax2.set_ylabel(r'$\Delta E_{ads}$ (eV)', size='small')
```

```
ax2.set_ylim(0, 6)
ax2.set_xlim(0, 1.5)
ax2.set_xticks([0, 0.5, 1.0, 1.5])

for name in ['IrO2', 'RhO2', 'RuO2', 'PtO2']:
    for struct in ['anatase', 'brookite', 'columbite', 'pyrite', 'rutile']:
        if name == 'PtO2' and struct == 'anatase':
            continue
        OH = data[name][struct]['OER']['OH']
        O = data[name][struct]['OER']['O']
        OOH = data[name][struct]['OER']['OOH']
        OHs.append(OH)
        Os.append(O)
        OOHs.append(OOH)

        if struct == 'rutile':
            f = 'none'
        else:
            f = 'full'

        ax2.plot(OH, O, c=c_dict[name], marker=m_dict[struct], fillstyle=f, ls='none')
        ax2.plot(OH, OOH, c=c_dict[name], marker=m_dict[struct], fillstyle=f, ls='none')

ax2.text(0.8, 1.4, r'$\Delta E_{ads}^{O}$', size=10)
ax2.text(0.7, 5, r'$\Delta E_{ads}^{OOH}$', size=10)

# Finally plot the OER activity relationships
ax3 = fig.add_axes([0.53, 0.12, 0.3, 0.35])
ax3.text(0.95, 0.34, '(c)')
ax3.plot((0 - 0.5, 1.6), (1.97 + 0.5, 0.37), color='k')
ax3.plot((1.6, 3.2 + 0.5), (0.37, 1.97 + 0.5), color='k')
ax3.set_yticklabels([])
ax3.set_ylim(1.2, 0.2)
ax3.set_xlim(0.8, 2.4)
ax3.set_xlabel('$\Delta G_{O*} - \Delta G_{OH*}$ (eV)', size='small')
ax4 = ax3.twinx()
ax4.set_ylabel(r'$\eta^{OER}$ (V)', size='small')
ax4.set_ylim(1.2, 0.2)
ax4.set_xlim(0.8, 2.4)
ax4.set_xticks([1, 1.5, 2])

```

```
112
113 for struct in ['anatase', 'brookite', 'columbite', 'pyrite', 'rutile']:
114     for name in ['IrO2', 'RhO2', 'RuO2', 'PtO2']:
115         complete = True
116         for ads in 'O', 'OH', 'OOH':
117             if isnan(data[name][struct]['OER'][ads]):
118                 complete = False
119             else:
120                 OH = data[name][struct]['OER']['OH']
121                 O = data[name][struct]['OER']['O']
122                 OOH = data[name][struct]['OER']['OOH']
123         if not complete:
124             continue
125
126         r1 = OH
127         r2 = O - OH
128         r3 = OOH - O
129         r4 = 4.92 - OOH
130         eta = max([r1, r2, r3, r4]) - 1.23
131         if struct == 'rutile':
132             f = 'none'
133             m = 1
134         else:
135             f = 'full'
136             m = None
137         plt.plot(r2, eta, marker=m_dict[struct], c=c_dict[name], ms=9, fillstyle=f,
138                  mew=m)
139
140 plt.savefig('images/FIG3.png', dpi=600)
141 plt.savefig('images/FIG3.eps', dpi=600)
142 plt.show()
```

7.4 FIG4: Analysis of DOS of IrO_2 rutile, columbite, and strained rutile

The script below is used to construct FIG4 of the manuscript. All data is read completely from the **data.json** file attached to the supporting information PDF.

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

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

with open('data/data.json', 'r') as f:
    data = json.load(f)

fig = plt.figure(1, (3.2, 6))

# First plot the rutile DOS
energies = np.array(data['IrO2']['rutile']['DOS']['E'])
occupied = energies <= 0.0
t2g = np.array(data['IrO2']['rutile']['DOS']['t2g'])
eg = np.array(data['IrO2']['rutile']['DOS']['eg'])

ax1 = fig.add_axes([0.2, 0.77, 0.75, 0.21])
ax1.text(-9.1, 2.8, '(a)')
E_ads = r'$\Delta E_{ads}^{O}=$'
E_ads += '\n{0:1.2f}'.format(data['IrO2']['rutile']['ads'])
E_ads += ' eV'
ax1.text(2.5, 2.4, E_ads, size=10, ha='center')
ax1.text(-7, 2.4, '$\mathdefault{IrO_{2}}$' + '\nrutile', size=10)
ax1.plot(energies, t2g, c='c', label=r'$t_{2g}$')
ax1.fill_between(x=energies[occupied], y1=t2g[occupied],
                 y2=np.zeros(t2g[occupied].shape), color='lightcyan')
ax1.plot(energies, eg, c='g', label=r'$e_g$')
ax1.fill_between(x=energies[occupied], y1=eg[occupied],
                 y2=np.zeros(eg[occupied].shape), color='lightgreen')

ax1.axvline(0, ls='--', c='k')
ax1.set_xlim(-10, 5)
ax1.set_xticklabels([])
ax1.set_ylim(0, 3.5)
ax1.set_yticks([0, 1, 2, 3])
ax1.set_ylabel('DOS (states/M)', size='small')

# Plot the rutile DOS with both max tensile and compressive strain
```

```
energies = np.array(data['IrO2']['rutile']['0.15']['DOS']['E'])
occupied = energies <= 0.0
t2g = np.array(data['IrO2']['rutile']['0.15']['DOS']['t2g'])
eg = np.array(data['IrO2']['rutile']['0.15']['DOS']['eg'])

ax2 = fig.add_axes((0.2, 0.54, 0.75, 0.21))
ax2.text(-9.1, 2.8, '(b)')
E_ads = r'$\Delta E_{ads}^{O}=$'
E_ads += '\n{0:1.2f}'.format(data['IrO2']['rutile']['0.15']['ads'])
E_ads += ' eV'
ax2.text(2.5, 2.4, E_ads, size=10, ha='center')
ax2.text(-7, 2.4, '$\mathdefault{IrO_{2}}$' + '\nrutile +15%', size=10)
ax2.plot(energies, t2g, c='c', label=r'$t_{2g}$')
ax2.fill_between(x=energies[occupied], y1=t2g[occupied],
                 y2=np.zeros(t2g[occupied].shape), color='lightcyan')
ax2.plot(energies, eg, c='g', label=r'$e_g$')
ax2.fill_between(x=energies[occupied], y1=eg[occupied],
                 y2=np.zeros(eg[occupied].shape), color='lightgreen')

ax2.axvline(0, ls='--', c='k')
ax2.set_xlim(-10, 5)
ax2.set_xticklabels([])
ax2.set_ylim(0, 3.5)
ax2.set_yticks([0, 1, 2, 3])
ax2.set_ylabel('DOS (states/M)', size='small')

energies = np.array(data['IrO2']['rutile']['-0.15']['DOS']['E'])
occupied = energies <= 0.0
t2g = np.array(data['IrO2']['rutile']['-0.15']['DOS']['t2g'])
eg = np.array(data['IrO2']['rutile']['-0.15']['DOS']['eg'])

ax3 = fig.add_axes((0.2, 0.31, 0.75, 0.21))
ax3.text(-9.1, 2.8, '(c)')
E_ads = r'$\Delta E_{ads}^{O}=$'
E_ads += '\n{0:1.2f}'.format(data['IrO2']['rutile']['-0.15']['ads'])
E_ads += ' eV'
ax3.text(2.5, 2.4, E_ads, size=10, ha='center')
ax3.text(-7, 2.4, '$\mathdefault{IrO_{2}}$' + '\nrutile -15%', size=10)
ax3.plot(energies, t2g, c='c', label=r'$t_{2g}$')
ax3.fill_between(x=energies[occupied], y1=t2g[occupied],
                 y2=np.zeros(t2g[occupied].shape), color='lightcyan')
```

```
ax3.plot(energies, eg, c='g', label=r'$e_g$')
ax3.fill_between(x=energies[occupied], y1=eg[occupied],
                 y2=np.zeros(eg[occupied].shape), color='lightgreen')

ax3.axvline(0, ls='--', c='k')
ax3.set_xlim(-10, 5)
ax3.set_xticklabels([])
ax3.set_ylim(0, 3.5)
ax3.set_yticks([0, 1, 2, 3])
ax3.set_ylabel('DOS (states/M)', size='small')

energies = np.array(data['IrO2']['columbite']['DOS']['E'])
occupied = energies <= 0.0
t2g = np.array(data['IrO2']['columbite']['DOS']['t2g'])
eg = np.array(data['IrO2']['columbite']['DOS']['eg'])

ax4 = fig.add_axes((0.2, 0.08, 0.75, 0.21))
ax4.text(-9.1, 2.8, '(d)')
E_ads = r'$\Delta E_{ads}^{O}=$'
E_ads += '\n{0:1.2f}'.format(data['IrO2']['columbite']['ads'])
E_ads += ' eV'
ax4.text(2.5, 2.4, E_ads, size=10, ha='center')
ax4.text(-7, 2.4, '$\mathdefault{IrO_{2}}$' + '\ncolumbite', size=10)
ax4.plot(energies, t2g, c='c', label=r'$t_{2g}$')
ax4.fill_between(x=energies[occupied], y1=t2g[occupied],
                 y2=np.zeros(t2g[occupied].shape), color='lightcyan')
ax4.plot(energies, eg, c='g', label=r'$e_g$')
ax4.fill_between(x=energies[occupied], y1=eg[occupied],
                 y2=np.zeros(eg[occupied].shape), color='lightgreen')

ax4.axvline(0, ls='--', c='k')
ax4.set_xlabel('Energy (eV)', size='small')
ax4.set_xlim(-10, 5)
ax4.set_ylim(0, 3.5)
ax4.set_yticks([0, 1, 2, 3])
ax4.set_ylabel('DOS (states/M)', size='small')

plt.savefig('images/FIG4.png', dpi=600)
plt.savefig('images/FIG4.eps', dpi=600)
plt.show()
```

8 Required modules

In addition to having a working license to the Vienna Ab-initio Simulation Package (VASP), one also requires at least Python 2.7 and several Python modules to reproduce all of our data. These modules are summarized below.

8.1 `numpy`, `scipy`, and `matplotlib`

`numpy`, `scipy`, and `matplotlib` are standard suites for performing scientific analysis using Python. Most linux distributions provide these packages along with Python, but they can also be easily downloaded via the Enthought package (www.enthought.com).

8.2 `ase`

`ase` is the Atomic Simulation Environment, which is wrapper for performing and analyzing quantum-chemical calculations using Python. It can be downloaded at `https://wiki.fysik.dtu.dk/ase/`.

8.3 `jasp`

`jasp` is a Python wrapper of the `ase.calculators.vasp` module written by Professor John Kitchin. This module facilitates performing calculations on a super computer with the submission of jobs, organization, and I/O control schemes. It allows for easy job organization and I/O of calculations. This can be found at `https://github.com/jkitchin/jasp`.

8.4 `ase_addons`

The python module `ase_addons` is a convenience module written by Zhongnan Xu and can be found at `http://github.com/zhongnanxu/ase_addons`. It contains bulk and surface structures along with a number of convenience functions. In this work, we primarily use it to load the rutile bulk unit cell and surface slab.

8.5 `dos_data`

The python module `dos_data` is a short python script for quickly analyzing the density of states of finished VASP calculations. It is attached to this supporting information and can be found below:

References

(1) Dominik, C. *The Org-Mode 7 Reference Manual: Organize Your Life with GNU Emacs*; Network Theory: UK, 2010.

(2) Sun, W.; Ceder, G. Efficient Creation and Convergence of Surface Slabs. *Surf. Sci.* **2013**, *617*, 53–59.

(3) Mehta, P.; Salvador, P. A.; Kitchin, J. R. Identifying Potential BO_2 Oxide Polymorphs for Epitaxial Growth Candidates. *ACS Appl. Mater. Interfaces* **2014**, *6*, 3630–3639.

(4) Xu, Z.; Kitchin, J. R. Relationships Between the Surface Electronic and Chemical Properties of Doped 4d and 5d Late Transition Metal Dioxides. *J. Chem. Phys.* **2015**, *142*, 104703.

(5) Grindy, S.; Meredig, B.; Kirklin, S.; Saal, J. E.; Wolverton, C. Approaching Chemical Accuracy With Density Functional Calculations: Diatomic Energy Corrections. *Phys. Rev. B* **2013**, *87*, 075150.

(6) Man, I. C.; Su, H.-Y.; Calle-Vallejo, F.; Hansen, H. A.; Martínez, J. I.; Inoglu, N. G.; Kitchin, J.; Jaramillo, T. F.; Nørskov, J. K.; Rossmeisl, J. Universality in Oxygen Evolution Electrocatalysis on Oxide Surfaces. *ChemCatChem* **2011**, *3*, 1159–1165.

(7) Momma, K.; Izumi, F. VESTA 3 for Three-Dimensional Visualization of Crystal, Volumetric and Morphology Data. *J. Appl. Crystallogr.* **2011**, *44*, 1272–1276.