The Role of van der Waals Interactions in Coverage Dependent Adsorption Energies of Various Adsorbates on Pt(111) and Pd(111)

Hari Thirumalai[a], John R. Kitchin[a,*]

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

Contents

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

1. Introduction

This document contains all the data, codes and parameters that were used for the work 'The Role of van der Waals Interactions in Coverage Dependent Adsorption Energies of Various Adsorbates on Pt(111) and Pd(111)'. As part of a new data sharing initiative, we wish to make our work completely transparent and open to reproduction. All the data from the DFT calculations were stored in JSON databases, while anything outside of this has been duly cited. JSON databases are a simple, plain text format for storing data. It can be read in many programming languages used in the open domain. The calculations and the storing of data has been done completely in the PYTHON programming language. JSON files can also be viewed in a human-readable format using a variety of native applications or on the web (https://www.jsoneditoronline.org).

The next section in this document contains the codes that were used to generate the relaxed adsorbed structures analyzed in this work. These codes

are preceeded by a small note, which provide more details. This section is followed by Calculation and Storage of Adsorption Energies, which provides the codes used to calculate the adsorption energies, standard errors and van der Waals contributions for each system. This is followed by a section, which contains codes for the generation of plots used to analyze the data. The final section consists of additional information, plots and figures regarding configurational correlations that are not included in the manuscript.

All the calculations performed in this work have been done using the Vienna ab-initio Simulation Package [1]. JASP, a wrapper, written by John Kitchin [2] was used as an interface between the VASP calculator and the user, and greatly simplifies the process of initializing calculations. All structures are created using the Atomic Simulation Environment (ASE).

2. Calculations

2.1. Convergence studies for Pt and Pd

To ensure a good criteria for convergence, and for a good guess for the lattice constant, convergence studies were performed. An initial guess of a lattice constant, close to the experimental value was taken, and first, a k-point convergence study was done. From figure 1 and figure 2, we chose a kpoint grid of $(12 \times 12 \times 12)$ for both Pt and Pd, as it represented an excellent compromise between accuracy and computational expense. Similarly, from figures 4 and 7, a plane wave cutoff of 520 eV was chosen for the same reasons mentioned earlier. Once these two parameters were obtained, the lattice constant of the metal was calculated. Finally, using a range of lattice constants around this calculated value, a range of ground state energies was

obtained. These lattice constants and energies were fitted to an equation of state, to finally obtain a minimum in lattice constant. These is shown in figures 3, and 6. The optimum lattice constant obtained for Pt(111) was 3.9934 Å, and for Pd(111), 3.9791 Å. These values of lattice constants were used in all future calculations.

2.1.1. Pt

K-points convergence.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
from jasp import *
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime'] = '24:00:00'

# Lattice constant
a = 4;

# Defining a Pt bulk unit cell
Pt = Atoms([Atom('Pt', (0, 0, 0))], cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                                              [0.0, 1.0, 1.0],
                                                              [1.0, 0.0, 1.0]]))

# Empty arrays
kpoints = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
energies = []

# Looping to show k point convergence
for i, k in enumerate(kpoints):
    with jasp('calculations/Pt/kpoints/{0}'.format(k),
              xc='PBE',
              gga='BF',
```

```
                encut=520,
                sigma=0.01,
                nsw=10,
                kpts=(k, k, k),
                ibrion=2,
                isif=3,
                atoms=Pt) as calc:

        try:
            atoms = calc.get_atoms()
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')

# Plotting k point convergence
import matplotlib.pyplot as plt

plt.plot(kpoints, energies)
plt.xlabel('K-points $(k \times k \times k)$')
plt.ylabel('Potential energy $(eV/\AA)$')
plt.title('K-point convergence')
plt.savefig('si-images/Pt-kpoint-convergence.png')
```

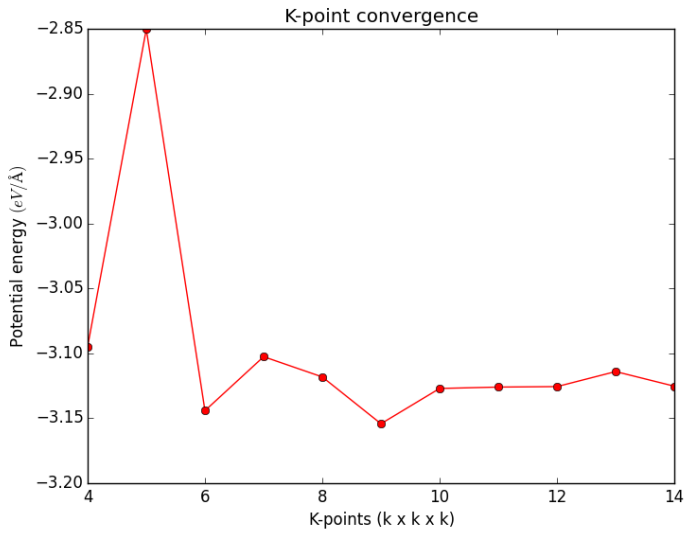

Figure 1: k-point convergence for Pt(111)

ENCUT convergence.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
from jasp import *
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime']='24:00:00'

# Lattice constant
a = 4;

# Defining a Pt bulk unit cell
Pt = Atoms([Atom('Pt', (0, 0, 0))], cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                                              [0.0, 1.0, 1.0],
                                                              [1.0, 0.0, 1.0]]))

```

```python
# Empty arrays
encuts = [250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800]
energies = []

for i, e in enumerate(encuts):
    with jasp('calculations/Pt/Encuts/{0}'.format(e),
              xc='PBE',
              gga='BF',
              encut=e,
              sigma=0.01,
              nsw=10,
              kpts=(12, 12, 12),
              ibrion=2,
              isif=3,
              ediff=1e-7,
              atoms=Pt) as calc:
        try:
            atoms = calc.get_atoms()
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')


# Plotting ENCUT convergence
import matplotlib.pyplot as plt
plt.plot(encuts, energies, 'ro-')
plt.xlabel('plane wave cutoff (eV)')
plt.ylabel('Potential energy $(eV/\AA)$')
plt.title('ENCUT convergence')
plt.savefig('si-images/Pt-encut-convergence.png')
plt.show()
```

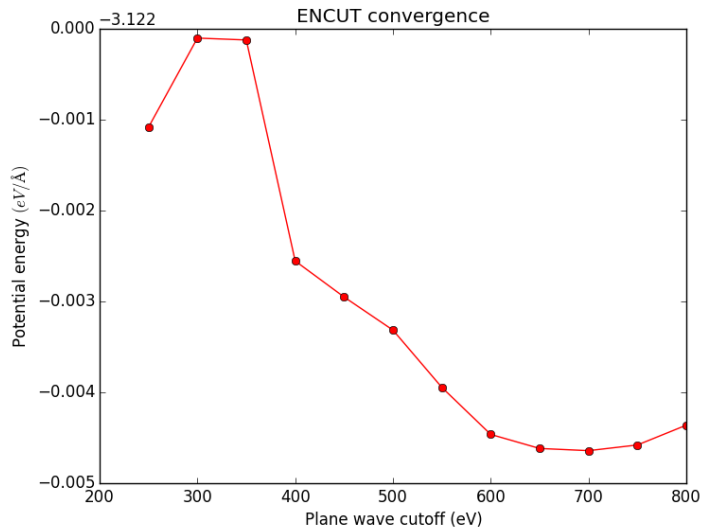

Figure 2: Plane wave cutoff convergence for Pd(111)

Calculation of volume.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
from jasp import *
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime']='24:00:00'

# Lattice constant
a = 4;

# Defining a Pt bulk unit cell
Pt = Atoms([Atom('Pt', (0, 0, 0))], cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                                              [0.0, 1.0, 1.0],
                                                              [1.0, 0.0, 1.0]]))

```

```python
# Calculating the relaxed geometry
with jasp('calculations/Pt/Pt-k12-e520',
          xc='PBE',
          gga='BF',
          encut=520,
          sigma=0.01,
          nsw=10,
          kpts=(12, 12, 12),
          ediff=1e-7,
          ibrion=2,
          isif=3,
          atoms=Pt) as calc:
    try:
        atoms = calc.get_atoms()
        calc.calculate()
        volume = atoms.get_volume()
        energy = atoms.get_potential_energy()

    except(VaspQueued, VaspSubmitted):
        print('In the queue!')


# Printing the required output
a = (4*volume)**(1.0/3)
print('The volume of the Unit cell is {0:0.4f} $\AA^3$'.format(volume))
print('Lattice constant = {0:0.4f} $\AA$'.format(a))
print('The potential energy is {0} eV/\AA'.format(energy))
```

```
The volume of the Unit cell is 15.9099 $\AA^3$
Lattice constant = 3.9925 $\AA$
The potential energy is -3.12548955 eV/\AA
```

Fitting to an equation of state.

```python
from jasp import *
from ase import atom, atoms
```

```python
import numpy as np
from ase.utils.eos import EquationOfState

JASPRC['queue.walltime']='24:00:00'
a = 3.9934

x = np.linspace(0.9, 1.1, 20)

# Empty arrays
volumes, energies = [], []

# looping over LC values to get volume and energies
for x1 in (x):
    a1 = x1 * a
    Pt = Atoms([Atom('Pt', (0, 0, 0))], cell=(0.5 * a1) * np.array([[1.0, 1.0, 0.0],
                                                                    [0.0, 1.0, 1.0],
                                                                    [1.0, 0.0, 1.0]]))

    with jasp('calculations/Pt/LC/{0:0.6f}'.format(a1),
              xc='PBE',
              gga='BF',
              encut=520,
              sigma=0.01,
              nsw=10,
              kpts=(12, 12, 12),
              ibrion=2,
              ediff=1e-7,
              isif=2,
              atoms=Pt) as calc:
        try:
            atoms = calc.get_atoms()
            calc.calculate()
            volumes.append(atoms.get_volume())
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')

```

```
# Fitting to the equation of state module
eos = EquationOfState(volumes, energies)
v0, e0, B = eos.fit()
a = (4 * v0)**(1.0/3)

print('Volume of lowest energy unit cell = {0:0.4f} $\AA^3$'.format(v0))
print('Corresponding lattice constant = {0:0.4f} \AA'.format(a))

eos.plot('si-images/Pt-eos.png')
```

```
Volume of lowest energy unit cell = 15.9177 $\AA^3$
Corresponding lattice constant = 3.9931 \AA
```

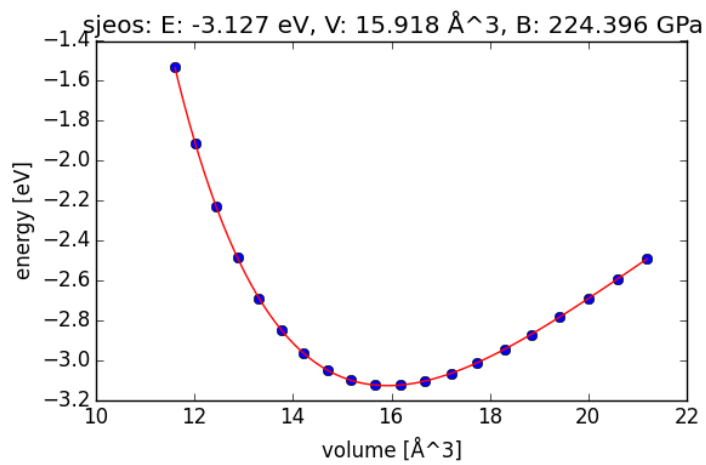

Figure 3: Fitting energy and volume data to an equation of state for Pt(111)

2.1.2. Pd

K-points convergence.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
```

```python
from jasp import *
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime'] = '24:00:00'

# Lattice constant
a = 3.9;

# Defining a Pd bulk unit cell
Pd = Atoms([Atom('Pd', (0, 0, 0))],
           cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                    [0.0, 1.0, 1.0],
                                    [1.0, 0.0, 1.0]]))

# Empty arrays
kpoints = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
energies = []

# Looping to show k point convergence
for i, k in enumerate(kpoints):
    with jasp('calculations/Pd/kpoints/{0}'.format(k),
              xc='PBE',
              gga='BF',
              encut=520,
              sigma=0.01,
              nsw=10,
              ediff=1e-7,
              kpts=(k, k, k),
              ibrion=2,
              isif=3,
              atoms=Pd) as calc:

        try:
            atoms = calc.get_atoms()
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
```

```
            print('In the queue!')


# Plotting k point convergence
import matplotlib.pyplot as plt

plt.plot(kpoints, energies, 'ro-')
plt.xlabel('K-points $(k \times k \times k)$')
plt.ylabel('Potential energy $(eV/\AA)$')
plt.title('K-point convergence')
plt.savefig('si-images/Pd-kpoint-convergence.png')
```

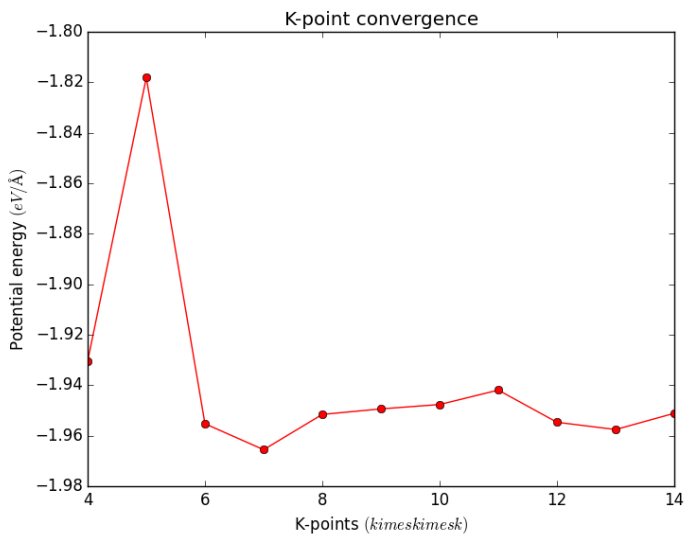

Figure 4: k-point convergence for Pd(111)

ENCUT convergence.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
from jasp import *
```

```python
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime']='24:00:00'

# Lattice constant
a = 3.9;

# Defining a Pt bulk unit cell
Pd = Atoms([Atom('Pd', (0, 0, 0))],
           cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                    [0.0, 1.0, 1.0],
                                    [1.0, 0.0, 1.0]]))

# Empty arrays
encuts = [250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800]
energies = []

for i, e in enumerate(encuts):
    with jasp('calculations/Pd/Encuts/{0}'.format(e),
              xc='PBE',
              gga='BF',
              encut=e,
              sigma=0.01,
              ediff=1e-7,
              nsw=10,
              kpts=(12, 12, 12),
              ibrion=2,
              isif=3,
              atoms=Pd) as calc:
        try:
            atoms = calc.get_atoms()
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')


```

```
# Plotting ENCUT convergence
import matplotlib.pyplot as plt
plt.plot(encuts, energies, 'ro-')
plt.xlabel('Plane wave cutoff (eV)')
plt.ylabel('Potential energy $(eV/\AA)$')
plt.title('ENCUT convergence')
plt.savefig('/home-research/hthiruma/beef-coverage/si-images/Pd-encut-convergence.png')
```

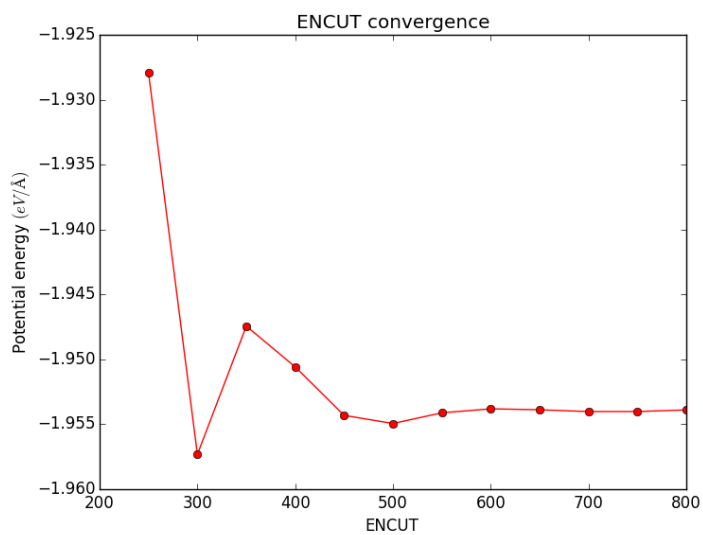

Figure 5: Plane wave cutoff for convergence for Pd(111)

Calculation of volume.

```
from ase.io import write
from ase.lattice.cubic import FaceCenteredCubic
from ase.visualize import view
from jasp import *
from ase import atom, atoms
from ase.lattice import bulk

JASPRC['queue.walltime']='24:00:00'
```

```python

# Lattice constant
a = 3.9;

# Defining a Pt bulk unit cell
Pd = Atoms([Atom('Pd', (0, 0, 0))],
           cell=0.5 * a * np.array([[1.0, 1.0, 0.0],
                                    [0.0, 1.0, 1.0],
                                    [1.0, 0.0, 1.0]]))

# Calculating the relaxed geometry
with jasp('calculations/Pd/Pd-k12-e520',
          xc='PBE',
          gga='BF',
          encut=520,
          sigma=0.01,
          nsw=10,
          kpts=(12, 12, 12),
          ibrion=2,
          isif=3,
          atoms=Pd) as calc:
    try:
        atoms = calc.get_atoms()
        calc.calculate()
        volume = atoms.get_volume()
        energy = atoms.get_potential_energy()

    except(VaspQueued, VaspSubmitted):
        print('In the queue!')


# Printing the required output
a = (4*volume)**(1.0/3)
print('The volume of the Unit cell is {0:0.4f} $\AA^3$'.format(volume))
print('Lattice constant = {0:0.4f} $\AA$'.format(a))
print('The potential energy is {0} eV/\AA'.format(energy))
```

```
The volume of the Unit cell is 15.7621 $\AA^3$
```

Lattice constant = 3.9801 $\AA$

The potential energy is -1.95464612 eV/\AA

Fitting to an equation of state.

```
from jasp import *
from ase import atom, atoms
import numpy as np
from ase.utils.eos import EquationOfState

JASPRC['queue.walltime']='24:00:00'

# Lattice constant array
LC = np.linspace(0.9, 1.1, 20)

# Empty arrays
volumes, energies = [], []

# looping over LC values to get volume and energies
for i, a in enumerate(LC):
    Pd = Atoms([Atom('Pd', (0, 0, 0))],
               cell=0.5 * 3.9801 * a * np.array([[1.0, 1.0, 0.0],
                                                 [0.0, 1.0, 1.0],
                                                 [1.0, 0.0, 1.0]]))

    with jasp('calculations/Pd/LC/{0:0.4f}'.format(a),
              prec = 'Accurate',
              xc='PBE',
              gga='BF',
              encut=520,
              sigma=0.01,
              nsw=10,
              kpts=(12, 12, 12),
              ediff=1e-7,
              ibrion=2,
              isif=2,
              atoms=Pd) as calc:
```

```python
        try:
            atoms = calc.get_atoms()
            calc.calculate()
            volumes.append(atoms.get_volume())
            energies.append(atoms.get_potential_energy())

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')


# Fitting to the equation of state module
eos = EquationOfState(volumes, energies)
v0, e0, B = eos.fit()
a = (4 * v0)**(1.0/3)

eos.plot('si-images/Pd-eos.png')

print('Volume of lowest energy unit cell = {0:0.4f} $\AA^3$'.format(v0))
print('Corresponding lattice constant = {0:0.4f} \AA'.format(a))
```

```
Volume of lowest energy unit cell = 15.7413 $\AA^3$
Corresponding lattice constant = 3.9783 \AA
```

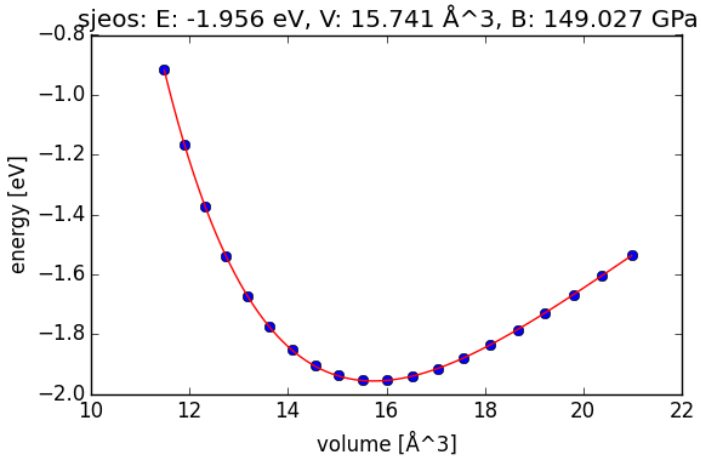

Figure 6: Fitting energy and volume data to an equation of state for Pd(111)

2.2. Calculation of Potential Energies

2.2.1. Gas Phase Energies of Adsorbates

The adsorbates considered in this work were Oxygen, Sulfur, Carbon, Nitrogen, Fluorine, Chlorine and Bromine. We have calculated the gas phase energies for this work, as they have previously shown applicability in scaling and configurational relations. The adsorbate atoms were placed at coordinates of (2 Å, 3 Å, 4 Å) in a box of dimensions 5 Å × 6 Å × 7 Å. They were relaxed using the same parameters calculated and assumed earlier, i.e a plane wave cutoff of 520 eV and a Gaussian smearing factor (sigma) of 0.01. The ground state energies were stored in a file atoms-data.json.

```
from jasp import *
from ase import Atom, Atoms
from ase.visualize import view
from ase.dft.bee import BEEFEnsemble
from modules import getvdw
```

```python
import json

# Initializing an empty dictionary
data = {}

# Gas phase adsorbates
molecules = ['C', 'N', 'O', 'F', 'S', 'Cl', 'Br']

for i, molecule in enumerate(molecules):

    # Atom is in a unit cell of dimensions 5 x 6 x 7 \AA
    data['{0}'.format(molecule)] = {}
    adsorbate = Atoms([Atom(molecule, [2, 3, 4], magmom = 2)],
                      cell = (5, 6, 7))

    # Initializing the VASP Calculator and running the calculation
    with jasp('calculations/molecules/{0}'.format(molecule),
              xc='PBE',
              gga='BF',
              sigma=0.01,
              ibrion=2,
              encut=520,
              ispin=2,
              nsw=20,
              atoms=adsorbate) as calc:

        try:
            # Get the ground state energy and ensemble energies
            atoms = calc.get_atoms()
            energy = atoms.get_potential_energy()
            ensemble = BEEFEnsemble(atoms).get_ensemble_energies()
            data['{0}'.format(molecule)]['energy'] = [energy, list(ensemble)]

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')

    vdw = getvdw('calculations/molecules/{0}'.format(molecule))
    data['{0}'.format(molecule)]['vdw'] = vdw
```

```python

# Store data in file atoms-data.json
with open('data/atoms-data.json', 'w') as f:
    json.dump(data, f)
```

2.2.2. Relaxation of Clean Slabs

A 2 × 2 surface unit cell, with 4 layers was chosen for this work. These slabs were surrounded by 10 Å of vacuum along the z-direction. The bottom two layers were constrained to remain fixed during the calculation process. The convergence parameters used here were the results of the convergence studies performed earlier, i.e a plane wave cutoff of 520 eV and k point grid of (12 × 12 × 12) for a single bulk metal atom were used. After the relaxation process, the ground state energies, and the error ensemble generated by the BEEF-vdW functional are stored in the file clean-energies.json.

```python
from jasp import *
from ase.lattice.surface import fcc111
from ase.constraints import FixAtoms
from ase.visualize import view
from ase.dft.bee import BEEFEnsemble
from modules import getvdw

JASPRC['queue.mem']='4GB'
JASPRC['queue.walltime']='24:00:00'

# Metals, lattice constants and k-point grids
metals = ['Pt', 'Pd']
lc = [3.9931, 3.9783]

data = {}

for i, metal in enumerate(metals):

```

```
    data[metal] = {}

    # Initialize metal atoms object and constrain the 2 lower layers
    slab = fcc111(metal, a = lc[i], size=(2, 2, 4), vacuum=6.0)
    constraint = FixAtoms(mask=[atom.tag > 2 for atom in slab])
    slab.set_constraint(constraint)

    # Initialize and run the calculator
    with jasp('calculations/clean/{0}'.format(metal),
              xc='PBE',
              gga='BF',
              encut=520,
              kpts=(6, 6, 1),
              sigma=0.1,
              ediffg=-5e-2,
              isif=2,
              ibrion=2,
              nsw=50,
              atoms=slab) as calc:

        try:
            # Parse out the ground state energy and error ensembles
            atoms = calc.get_atoms()
            energy = atoms.get_potential_energy()
            ensemble = BEEFEnsemble(atoms).get_ensemble_energies()
            data[metal]['energy'] = [energy, list(ensemble)]

        except(VaspQueued, VaspSubmitted):
            print('In the queue!')

    # Get van der Waals contribution
    vdw = getvdw('calculations/clean/{0}'.format(metal))
    data[metal] = [energy, list(ensemble), vdw]

# Store data
with open('data/clean-energies.json', 'w') as f:
    json.dump(data, f)
```

2.2.3. Relaxation of Adsorbed slabs

A 2 × 2 surface unit cell, with 4 layers was chosen for this work. These slabs were surrounded by 10 Å of vacuum along the z-direction. The bottom two layers were constrained to remain fixed during the calculation process. Adsorbates were added on the fcc and the atop sites. An additional constraint was added for the adsorbate on the ontop site, such that they can move only in the z-direction. Upon relaxation, the energies and the error ensembles were stored in the file pot-energies.json.

```
from jasp import *
from ase.lattice.surface import fcc111, add_adsorbate
from ase.constraints import FixAtoms, FixScaled
from ase.visualize import view
import numpy as np
from ase.dft.bee import BEEFEnsemble
from modules import getvdw

# Calculation specifics
metals = ['Pt', 'Pd']
lc = [3.9931, 3.9783]
site = ['fcc', 'ontop']
adsorbate = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverage = [0.25, 0.5, 0.75, 1.0]
h = [1.2, 2.0]

# Initializing the calculation
for i, metal in enumerate(metals):
    for j, s in enumerate(site):
        for ads in adsorbate:
            for c in coverage:
                # Initialize the metal atoms object
                atoms = fcc111(metal, a=lc[i], size=(2, 2, 4), vacuum=6.0)

                # Add one adsorbate
```

```
                if c == 0.25:
                    add_adsorbate(atoms, ads, height=h[j], position=s)
                    nums = np.arange(16, len(atoms))

                # Add two adsorbates
                elif c == 0.5:
                    add_adsorbate(atoms, ads, height=h[j], position=s)
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(1, 0))
                    nums = np.arange(16, len(atoms))

                # Add three adsorbates
                elif c == 0.75:
                    add_adsorbate(atoms, ads, height=h[j], position=s)
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(1, 0))
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(0, 1))
                    nums = np.arange(16, len(atoms))

                # Add four adsorbates
                elif c == 1.0:
                    add_adsorbate(atoms, ads, height=h[j], position=s)
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(1, 0))
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(0, 1))
                    add_adsorbate(atoms, ads, height=h[j], position=s, offset=(1, 1))
                    nums = np.arange(16, len(atoms))

                if s == 'fcc':
                    # Fix the lower 2 layers of the metal surface
                    constraint1 = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
                    atoms.set_constraint(constraint1)

                else:
                    const = []
                    # Fix the lower 2 layers of the metal surface
                    # and the movement of the adsorbates in the X and Y directions
                    constraint1 = FixAtoms(mask=[atom.tag > 2 for atom in atoms])
                    const.append(constraint1)
                    for x in nums:
                        constraint2 = FixScaled(atoms.get_cell(), x, [True, True, False])
```

```
                        const.append(constraint2)

                    atoms.set_constraint(const)

                wd = 'calculations/adsorbed/m-{0}/s-{1}/a-{2}/c-{3}'.format(metal, s, ads, c)

                # Initialize the calculation
                with jasp(wd,
                          xc='PBE',
                          gga='BF',
                          kpts=(6, 6, 1),
                          encut=520,
                          sigma=0.1,
                          ediffg=-5e-2,
                          isif=2,
                          ibrion=2,
                          nsw=50,
                          atoms=atoms) as calc:

                    try:
                        atoms = calc.get_atoms()
                        e = atoms.get_potential_energy()
                        ens = BEEFEnsemble(atoms).get_ensemble_energies()

                    except(VaspQueued, VaspSubmitted, IOError):
                        print wd+' In the queue'

                # Get van der Waals contribution
                vdw = getvdw('calculations/adsorbed/m-{0}/s-{1}/a-{2}/c-{3}'.format(metal, s, ads, c))
                data1[metal][s][ads]['{0}'.format(c)] = [e, list(ens), vdw]

# Storing the data is file energies.json
with open('data/pot-energies.json', 'w') as f:
    json.dump(data1, f)
```

3. Calculation and Storage of Adsorption Energies, Standard Errors and van der Waals Contributions

The adsorption energies for each configuration was calculated using equation 1,

$$\Delta E_{ads}(eV/\mathring{A}) = \frac{1}{n}.(E_{adsorbed\ slab} - E_{clean\ slab} - n.E^{gas}_{adsorbate}) \tag{1}$$

and the van der Waals interactions were calculated using an analogous equation 2

$$\Delta E^{nl-c}_{ads}(eV/\mathring{A}) = \frac{1}{n}.(E^{nl-c}_{adsorbed\ slab} - E^{nl-c}_{clean\ slab} - n.E^{gas,\ nl-c}_{adsorbate}) \tag{2}$$

The adsorption energies, standard errors and the van der Waals interactions are calculated per atom of the adsorbing species and stored in the file adsorption-energies.json. A special function was required for obtained the van der Waals contribution from each calculation, and this can be found in the python module modules. The databases required for the generation of these energies are clean-energies.json, atoms-data.json and pot-energies.json. All of these files may be found attached to the supporting information document.

```
import json
import numpy as np

# Calculation specifics
metals = ['Pt', 'Pd']
site = ['fcc', 'ontop']
adsorbate = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverage = ['0.25', '0.5', '0.75', '1.0']

```

```python
# Opening potential energy databases
with open('data/clean-energies.json') as f:
    clean = json.load(f)

with open('data/pot-energies.json') as f:
    pot = json.load(f)

with open('data/atoms-data.json') as f:
    atoms = json.load(f)

data1 = {}

# Initializing the calculation
for i, metal in enumerate(metals):
    clean_e, clean_ens, clean_vdw = clean[metal]
    clean_ens = np.array(clean_ens)
    data1[metal] = {}

    for j, s in enumerate(site):
        data1[metal][s]= {}
        for ads in adsorbate:
            atom_e = atoms[ads]['energy'][0]
            atom_ens = np.array(atoms[ads]['energy'][1])
            atom_vdw = atoms[ads]['vdw']

            data1[metal][s][ads]= {}
            for k, c in enumerate(coverage):
                if metal == 'Pd' and s == 'ontop' and ads == 'N' and c == '0.5':
                    x, y, z = 0, 0, 0
                elif metal == 'Pt' and s == 'fcc' and ads == 'Br' and c == '0.5':
                    x, y, z = 0, 0, 0

                else:
                    ads_e, ads_ens, ads_vdw = pot[metal][s][ads][c]
                    ads_ens = np.array(ads_ens)

                    x = (ads_e - clean_e - atom_e*(k+1))/(k+1)
                    y = ((ads_ens - clean_ens - atom_ens*(k+1))/(k+1)).std()
```

```
                    z = (ads_vdw - clean_vdw - atom_vdw*(k+1))/(k+1)

                    # Store adsorption energy, standard error and van der waals contribution
                    data1[metal][s][ads][c] = [x, y, z]

# Write json file
with open('data/adsorption-energies.json', 'w') as f:
    json.dump(data1, f)
```

4. Generation of Plots

4.1. Plotting Adsorption Energies and Errors

```
import json
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
from matplotlib.ticker import MaxNLocator
import seaborn as sns

sns.set_style('ticks')
sns.set_context("paper")

# Opening adsorption energy database
with open('data/adsorption-energies.json') as f:
    ads = json.load(f)

# Calculation specifics
metals = ['Pt', 'Pd']
sites = ['fcc', 'ontop']
adsorbates = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverages = ['0.25', '0.5', '0.75', '1.0']
markers = ['o', 'v', '^', 's', 'D', 'h', 'p']
colors = ['r', 'g', 'b', 'k', 'y', 'm', 'c']

# Generate empty plots
f1, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(4, 6))
```

```python
f2, (ax3, ax4) = plt.subplots(2, sharex=True, figsize=(4, 6))
Pt_plot = [ax1, ax2]
Pd_plot = [ax3, ax4]
plts = [Pt_plot, Pd_plot]

# Parsing out data from the database
for i, m in enumerate(metals):
    for j, s in enumerate(sites):
        plts[i][j].set_xticks([0.25, 0.5, 0.75, 1.0])
        for k, a in enumerate(adsorbates):
            energy = []
            errors = []
            cov = []
            for l, c in enumerate(coverages):

                if m == 'Pd' and s == 'ontop' and a == 'N' and c == '0.5':
                    pass
                elif m == 'Pt' and s == 'fcc' and a == 'Br' and c == '0.5':
                    pass
                else:
                    e, err = ads[m][s][a][c][0], ads[m][s][a][c][1]

                c = float(c)
                energy.append(e)
                errors.append(err)
                cov.append(c)

                # Plotting shaded areas
                plts[i][j].fill_between(np.linspace(c-0.15, c+0.15), e-err, e+err, facecolor=colors[k], alpha=0.3)
                plts[i][j].errorbar(c, e, yerr=err, marker=markers[k], color=colors[k])

            # Plotting data points and fits to the data points
            fit = np.polyfit(cov, energy, 1)
            fit_func = np.poly1d(fit)
            plts[i][j].plot(cov, fit_func(cov), linestyle='-', color=colors[k], linewidth=1.5)
            plts[i][j].plot(cov, energy, color=colors[k], marker=markers[k], linestyle='none')

        plts[i][j].set_xlim([0, 1.2])
```

```
        plts[i][j].set_xlabel('Coverage')
        plts[i][j].set_ylabel('$E_{ads}$ (eV)')
        plts[i][j].locator_params(axis='y', nbins=7)
        plts[i][j].set_xticklabels([r'$\mathdefault{\frac{1}{4}}$',
                                    r'$\mathdefault{\frac{1}{2}}$',
                                    r'$\mathdefault{\frac{3}{4}}$',
                                    '1'])

# Generate plot text
ax1.text(0.025, -0.6, r'(a) Pt(111) fcc'.format(m), fontsize=10)
ax3.text(0.025, -0.6, r'(a) Pd(111) fcc'.format(m), fontsize=10)
ax2.text(0.025, -0.4, r'(b) Pt(111) ontop'.format(m), fontsize=10)
ax4.text(0.025, -0.4, r'(b) Pd(111) ontop'.format(m), fontsize=10)

ax1.axes.get_xaxis().set_visible(False)
ax3.axes.get_xaxis().set_visible(False)
f1.tight_layout()
f2.tight_layout()

# Save plots
for ext in ['.eps', '.png', '.pdf']:
    f1.savefig('images/Pt-errors' + ext, dpi=300)
    f2.savefig('images/Pd-errors' + ext, dpi=300)

plt.show()
```

4.2. van der Waals Contribution Plots

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

sns.set_style('ticks')
sns.set_context("paper")

# Open required databases
with open('data/adsorption-energies.json') as f:
```

```
    ads = json.load(f)

# Calculation specifics
metals = ['Pt', 'Pd']
sites = ['fcc', 'ontop']
adsorbates = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverages = ['0.25', '0.5', '0.75', '1.0']
colors = ['r', 'g', 'b', 'm', 'y', 'k', 'c']
markers = ['o', 'v', '^', 's', 'D', 'h', 'p']
xfit = np.linspace(0.25, 1.0, 100)

# Generate empty plots
f1, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(4, 6))
f2, (ax3, ax4) = plt.subplots(2, sharex=True, figsize=(4, 6))
Pt_plot = [ax1, ax2]
Pd_plot = [ax3, ax4]
plts = [Pt_plot, Pd_plot]

# Parsing out van der Waals contributions
for i, m in enumerate(metals):
    for j, s in enumerate(sites):
        plts[i][j].set_xticks([0.25, 0.5, 0.75, 1.0])
        for k, a in enumerate(adsorbates):
            vdw = []
            cov = []
            for l, c in enumerate(coverages):

                if m == 'Pd' and s == 'ontop' and a == 'N' and c == '0.5':
                    pass
                elif m == 'Pt' and s == 'fcc' and a == 'Br' and c == '0.5':
                    pass
                else:
                    e_vdw = ads[m][s][a][c][2]
                    vdw.append(e_vdw)
                    cov.append(float(c))

            # Plotting van der Waals energies
            plts[i][j].plot(cov, vdw, marker=markers[k], color=colors[k], linestyle='dotted')
```

```
            plts[i][j].set_xlim([0.2, 1.05])

        plts[i][j].set_xlabel('Coverage')
        plts[i][j].set_ylabel('$E_{vdW}$ (eV)')
        plts[i][j].locator_params(axis='y', nbins=7)
        plts[i][j].set_xticklabels([r'$\mathdefault{\frac{1}{4}}$',
                                    r'$\mathdefault{\frac{1}{2}}$',
                                    r'$\mathdefault{\frac{3}{4}}$',
                                    '1'])
ax1.set_ylim([-1.3, 0.2])
ax3.set_ylim([-1.3, 0.2])
ax2.set_ylim([-1.3, 0.4])
ax4.set_ylim([-1.3, 0.4])

# Generate plot text
ax1.text(0.225, 0.1, r'(a) Pt(111) fcc'.format(m), fontsize=10)
ax3.text(0.225, 0.1, r'(a) Pd(111) fcc'.format(m), fontsize=10)
ax2.text(0.225, 0.27, r'(b) Pt(111) ontop'.format(m), fontsize=10)
ax4.text(0.225, 0.27, r'(b) Pd(111) ontop'.format(m), fontsize=10)

ax1.axes.get_xaxis().set_visible(False)
ax3.axes.get_xaxis().set_visible(False)
f1.tight_layout()
f2.tight_layout()

# Saving plots
for ext in ['.eps', '.png', '.pdf']:
    f1.savefig('images/Pt-vdws' + ext, dpi=300)
    f2.savefig('images/Pd-vdws' + ext, dpi=300)

plt.show()
```

4.3. Parity Plot

```
import json
import matplotlib.pyplot as plt
import numpy as np
#from modules import check
```

```
import matplotlib.cm as cm
#import seaborn as sns

#sns.set_style('ticks')
#sns.set_context("paper")

# Opening database of adsorption energies provided by Xu et al
with open('data/zz-energies.json') as f:
    data1 = json.load(f)

# Opening the database of adsorption energies generated in this work
with open('data/adsorption-energies.json') as f:
    ads = json.load(f)

# Calculation specifics
metals = ['Pt', 'Pd']
adsorbates = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverages = ['0.25', '0.5', '0.75', '1.0']
markers = ['s',  'o', '^', 'D']
sites = ['fcc', 'ontop']
colors = ['red', 'green', 'blue', 'yellow']

fig, ax = plt.subplots(figsize=(3, 4))

for i, m in enumerate(metals):
    for l, s in enumerate(sites):
        for j, a in enumerate(adsorbates):
            for k, c in enumerate(coverages):

                if m == 'Pd' and s == 'ontop' and a == 'N' and c == '0.5':
                    pass
                elif m == 'Pt' and s == 'fcc' and a == 'Br' and c == '0.5':
                    pass
                else:

                    e1 = data1[m][a][s][c]
                    e2 = ads[m][s][a][c][0]
                    ax.scatter(e2, e1, s=10, zorder=10,
```

```
                                linewidth=0.1, color=colors[l], marker=markers[l])

lims = [np.min([ax.get_xlim(), ax.get_ylim()]),
        np.max([ax.get_xlim(), ax.get_ylim()])]

ax.plot(lims, lims, 'k-', alpha=0.75, zorder=0)
ax.set_xlim(lims)
ax.set_ylim(lims)

ax.set_xlabel('$E_{ads}^{BEEF}$ (eV)')
ax.set_ylabel('$E_{ads}^{PBE}$ (eV)')
ax.locator_params(axis='x', nbins=6)
ax.locator_params(axis='y', nbins=6)
plt.tight_layout()

# Save plots
for ext in ['.eps', '.png', '.pdf']:
    plt.savefig('images/parity-plot' + ext, dpi=300)

plt.show()
```

4.4. fcc vs ontop Configurational Correlation Plots

```
import json
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

sns.set_style('ticks')
sns.set_context("paper")

# Calculation specifics
metals = ['Pt', 'Pd']
sites = ['fcc', 'ontop']
adsorbates = ['O', 'S', 'C', 'N', 'F', 'Cl', 'Br']
coverages = ['0.25', '0.5', '0.75', '1.0']
colors = ['r', 'g', 'b', 'k', 'y', 'm', 'c']
```

```python

fig = plt.figure(figsize=(4, 6))
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

# Open required databases
with open('data/adsorption-energies.json') as f:
    ads = json.load(f)

# Initializing empty arrays for r squared, standard deviation
# slope and coverage
rsq = []
std = []
cov = []
slp = []

# Parsing out adsorption energies
for j, c in enumerate(coverages):
    e_fcc = []
    err_fcc = []
    e_ontop = []
    err_ontop = []

    for i, m in enumerate(metals):
        for k, a in enumerate(adsorbates):
            for l, s in enumerate(sites):
                if m == 'Pd' and s == 'ontop' and a == 'N' and c == '0.5':
                    pass
                elif m == 'Pt' and s == 'fcc' and a == 'Br' and c == '0.5':
                    pass

                else:
                    e, err = ads[m][s][a][c][0], ads[m][s][a][c][1]

                if s == 'fcc':
                    e_fcc.append(e)
                    err_fcc.append(err)

```

```
                else:
                    e_ontop.append(e)
                    err_ontop.append(err)

    # Plotting the adsorption energy
    ax1.errorbar(e_fcc, e_ontop, xerr=err_fcc, yerr=err_ontop, marker='o', color=colors[j], linestyle='None')
    fit = np.polyfit(e_fcc, e_ontop, 1)
    fit_func = np.poly1d(fit)
    ax1.plot(e_fcc, fit_func(e_fcc), linestyle='-', color=colors[j], linewidth=2.0)
    slope, intercept, r_value, p_value, std_err = stats.linregress(e_fcc, e_ontop)

    # Store fitting statistics
    rsq.append(r_value**2)
    std.append(std_err)
    cov.append(float(c))
    slp.append(slope)

    ax1.set_xlabel('$E_{ads}^{fcc}$ (eV)')
    ax1.set_ylabel('$E_{ads}^{ontop}$ (eV)')
    ax1.locator_params(axis='x', nbins=5)
    ax1.locator_params(axis='y', nbins=6)

ax2.set_xticks([0.25, 0.5, 0.75, 1.0])
ax2.errorbar(cov, slp, yerr=std, marker='o', linestyle='--')
ax2.set_xlim([0.2, 1.05])
ax2.set_xlabel('Coverage')
ax2.set_ylabel('Slope')
ax2.set_xticklabels([r'$\mathdefault{\frac{1}{4}}$',
                     r'$\mathdefault{\frac{1}{2}}$',
                     r'$\mathdefault{\frac{3}{4}}$',
                     '1'])
# Save plot
plt.tight_layout()
for ext in ['.eps', '.png', '.pdf']:
    plt.savefig('si-images/fcc-ontop' + ext, dpi=300)

plt.show()
```

4.5. Graphical Abstract

```python
import json
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
from matplotlib.ticker import MaxNLocator
import seaborn as sns

sns.set_style('ticks')
sns.set_context("paper")

# Opening adsorption energy database
with open('data/adsorption-energies.json') as f:
    ads = json.load(f)

# Calculation specifics
coverages = ['0.25', '0.5', '0.75', '1.0']

fig = plt.figure(figsize=(13/2.54, 5/2.54))
ax = fig.add_subplot(111)    # The big subplot
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)

# Turn off axis lines and ticks of the big subplot
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')
ax.tick_params(labelcolor='w', top='off', bottom='off',
               left='off', right='off')

# Set common labels
ax1.set_xlabel('Coverage', fontsize=12)
ax2.set_xlabel('Coverage', fontsize=12)

ax.tick_params(axis='x', which='major')
```

```
ax.tick_params(axis='x', which='minor')
ax.tick_params(axis='x', which='minor')

# Plot adsorption energy as a function of coverage for O on Pt(111)
# at a coverage of 0.5 ML at the hollow site
ax1.set_xticks([0.25, 0.5, 0.75, 1.0])

cov, energies, errors, vdws = [], [], [], []
for c in coverages:
    e, err, vdw = ads['Pt']['fcc']['O'][c]
    energies.append(e)
    errors.append(err)
    vdws.append(vdw)
    c = float(c)
    cov.append(c)

    ax1.fill_between(np.linspace(c - 0.15, c + 0.15),
                     e-err, e+err, facecolor='red', alpha=0.2)
    ax1.errorbar(c, e, yerr=err, marker='o', color='red')

# Plotting data points and fits to the data points
fit = np.polyfit(cov, energies, 1)
fit_func = np.poly1d(fit)
ax1.plot(cov, fit_func(cov), linestyle='-', color='red', linewidth=1.5)
ax1.plot(cov, energies, color='red', marker='o', linestyle='none')
ax1.set_ylabel('$E_{ads}$ (eV)', fontsize=12)

# Plot vdW contributions as a function of coverage for O on Pt(111)
# at a coverage of 0.5 ML at the hollow site
ax2.set_xticks([0.25, 0.5, 0.75, 1.0])
ax2.plot(cov, vdws, marker='s', color='green', linestyle=':')
ax2.set_ylabel('$E_{vdW}$ (eV)', fontsize=12)

for ax in [ax1, ax2]:
    ax.locator_params(axis='y', nbins=5)
    ax.set_xlim([0, 1.25])

ax2.set_ylim([-0.65, -0.5])
```

```python

for ax in [ax1, ax2]:
    ax.set_xticklabels([r'$\mathdefault{\frac{1}{4}}$',
                        r'$\mathdefault{\frac{1}{2}}$',
                        r'$\mathdefault{\frac{3}{4}}$',
                        '1'])

ax1.set_position([0.16, 0.26, 0.3, 0.65])
ax2.set_position([0.62, 0.26, 0.3, 0.65])

for ext in ['.eps', '.png', '.pdf']:
    plt.savefig('images/graphical-abstract' + ext, dpi=300)

# [[./images/graphical-abstract.png]]
```

5. Configurational correlations with coverage and sites

The existence of configuration correlations between adsorption of species at the fcc hollow sites and at the atop sites was investigated. We find that inclusion of vdW interactions does show the existence of these correlations. The slope of the fits to the data points at different coverages of 0.25 ML, 0.5 ML, 0.75 ML and 1.0 ML show a gradual but similar change in slope. This is seen in figure 7.

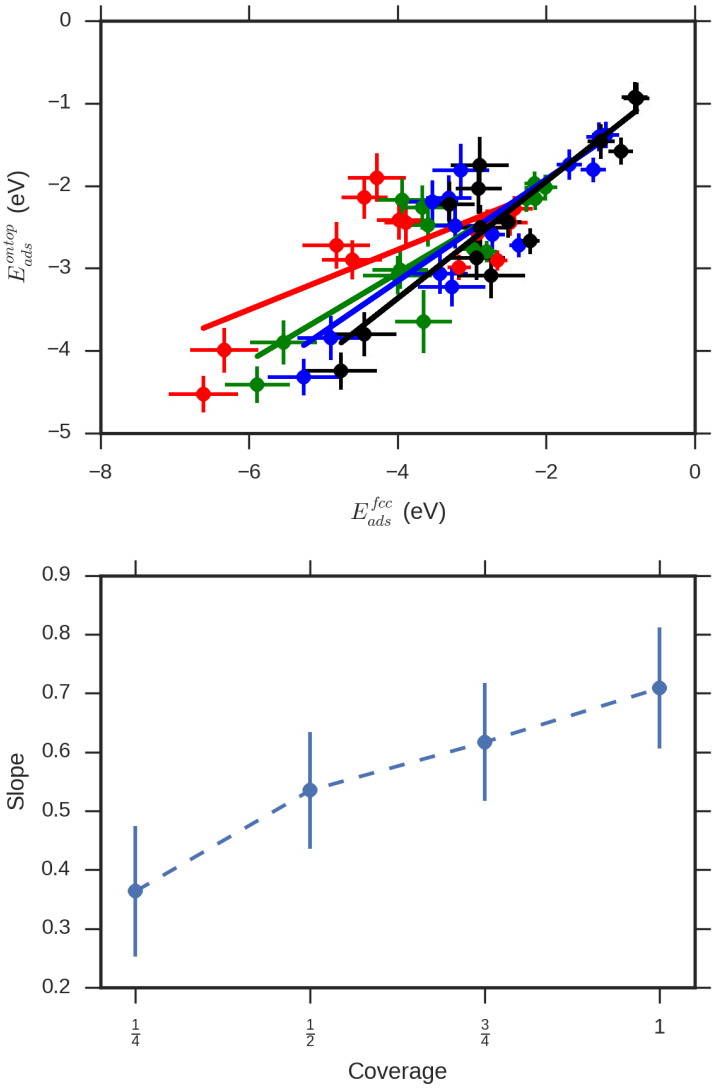

Figure 7: Configurational correlations between hollow sites and ontop sites. Existence of correlations were studied based on specific coverage. Here 0.25 ML is in red, 0.5 ML is in green, 0.75 ML is in blue and 1.0 ML is black in color.

6. References

References

[1] G. Kresse, J. Hafner, Ab Initio Molecular Dynamics for Liquid Metals, Physical Review B 47 (1) (1993) 558–561. doi:10.1103/physrevb.47.558.
URL http://dx.doi.org/10.1103/physrevb.47.558

[2] J. R. Kitchin, JASP-A wrapper for VASP, https://github.com/jkitchin/jasp.


```
from pylab import *

# create some data to use for the plot
dt = 0.001
t = arange(0.0, 10.0, dt)
r = exp(-t[:1000]/0.05)               # impulse response
x = randn(len(t))
s = convolve(x,r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plot(t, s)
axis([0, 1, 1.1*amin(s), 2*amax(s) ])
xlabel('time (s)')
ylabel('current (nA)')
title('Gaussian colored noise')

# this is an inset axes over the main axes
a = axes([.65, .6, .2, .2], axisbg='None')
n, bins, patches = hist(s, 400, normed=1)
title('Probability')
setp(a, xticks=[], yticks=[])

```

```
# this is another inset axes over the main axes
a = axes([0.2, 0.6, .2, .2], axisbg='y')
plot(t[:len(r)], r)
title('Impulse response')
setp(a, xlim=(0,.2), xticks=[], yticks=[])


show()
```