surfinpy Documentation

Release 2.0.0

Adam R. Symington

Dec 02, 2021

CONTENTS

This is the documentation for the open-source Python project, *surfinpy*. A library designed to facilitate the generation of publication ready phase diagrams from ab initio calculations. *surfinpy* is built on existing Python packages that those in the solid state physics/chemistry community should already be familiar with. It is hoped that this tool will bring some benfits to the solid state community and facilitate the generation of publication ready phase diagrams (powered by Matplotlib.)

The main features include:

1. **Method to generate surface phase diagrams as a function of chemical potential.**
   - Generate a diagram as a function of the chemical potential of two adsorbing species e.g. water and carbon dioxide.
   - Generate a diagram as a function of the chemical potential of one adsorbing species and a surface species e.g. water and oxygen vacancies.
   - Use experimental data combined with ab initio data to generate a temperature dependent phase diagram.
2. **Method to generate surface phase diagrams as a function of temperature and pressure.**
   - Use experimental data combined with ab initio data to generate a pressure vs temperature plot showing the state of a surface as a function of temperature and pressure of one species.
3. **Use calculated surface energies to built crystal morphologies.**
   - Use the surface energies produced by *surfinpy* alongside Pymatgen to built particle morphologies.
   - Evaulate how a particles shape changes with temperature and pressure.
4. **Method to generate bulk phase diagrams as a function of chemical potential.**
   - Generate a diagram as a function of the chemical potential of two species e.g. water and carbon dioxide.
   - Use experimental data combined with ab initio data to generate a temperature dependent phase diagram.
5. **Method to generate bulk phase diagrams as a function of temperature and pressure.**
   - Use experimental data combined with ab initio data to generate a pressure vs temperature plot showing the phase space as a function of temperature and pressure.
6. **Method to include vibrational properties in a phase diagram**
   - Module to calculate the zero point energy and vibrational entropy
   - Encorporate the zero point energy and/or the vibrational entropy into a phase diagram.

The code has been developed to analyse VASP calculations but is compatible with other ab initio codes. *surfinpy* was developed across several PhD projects and as such the functionality focuses on the research questions encountered during those projects, which we should clarify are wide ranging. Code contributions aimed at expanding the code to new problems are encouraged.

surfinpy is free to use.

CHAPTER

ONE

USAGE

A full list of examples can be found in the examples folder of the git repository, these include jupyter notebook tutorials which combine the full theory with code examples.

CHAPTER

TWO

INSTALLATION

surfinpy is a Python 3 package and requires a typical scientific Python stack. Use of the tutorials requires Anaconda/Jupyter to be installed.

To build from source:

```
pip install -r requirements.txt

python setup.py build

python setup.py install

python setup.py test
```

Or alternatively install with pip

```
pip install surfinpy
```

CHAPTER THREE

DOCUMENTATION

To build the documentation from scratch

```
cd docs

make html
```

CHAPTER

FOUR

LICENSE

surfinpy is made available under the MIT License.

CHAPTER

FIVE

DETAILED REQUIREMENTS

surfinpy is compatible with Python 3.5+ and relies on a number of open source Python packages, specifically:

- Numpy
- Scipy
- Matplotlib
- Pymatgen

CHAPTER

SIX

CONTRIBUTING

6.1 Contact

If you have questions regarding any aspect of the software then please get in touch with the developer Adam Symington via email - ars44@bath.ac.uk. Alternatively you can create an issue on the Issue Tracker or you can discuss your questions on our gitter channel.

6.2 Bugs

There may be bugs. If you think you've caught one, please report it on the Issue Tracker. This is also the place to propose new ideas for features or ask questions about the design of *surfinpy*. Poor documentation is considered a bug so feel free to request improvements.

6.3 Code contributions

We welcome help in improving and extending the package. This is managed through Github pull requests; for external contributions we prefer the "fork and pull" workflow while core developers use branches in the main repository:

1. First open an Issue to discuss the proposed contribution. This discussion might include how the changes fit surfinpy's scope and a general technical approach.
2. Make your own project fork and implement the changes there. Please keep your code style compliant with PEP8.
3. Open a pull request to merge the changes into the main project. A more detailed discussion can take place there before the changes are accepted.

For further information please contact Adam Symington, ars44@bath.ac.uk

CHAPTER

SEVEN

RESEARCH

- Strongly Bound Surface Water Affects the Shape Evolution of Cerium Oxide Nanoparticles
- The energetics of carbonated PuO2 surfaces affects nanoparticle morphology: a DFT+U study
- Exploiting cationic vacancies for increased energy densities in dual-ion batteries

7.1 Theory

There is a significant amount of theory behind the methods in *surfinpy*. The following three pages provide an explanation for the methods employed in the code.

surfinpy is a Python module to generate phase diagrams from energy minimisation data. Before using this code you will need to generate the relevant data.

7.1.1 Surface Theory

surfinpy has the capability to generate phase diagrams as a function of chemical potential of two varying species e.g. water and carbon dioxide. In such an example the user would require calculations with varying concentrations of water, carbon dioxide and water/carbon dioxide on a surface. Assuming that you have generated enough, reliable data then you are ready to use *surfinpy*.

Surface Energy

The physical quantity that is used to define the stability of a surface with a given composition is its surface energy γ (J m^{-2}). At its core, *surfinpy* is a code that calculates the surface energy of different slabs at varying chemical potential and uses these surface energies to construct a phase diagram. In this explantion of theory we will use the example of water adsorbing onto a surface of TiO_2 containing oxygen vacancies. In such an example there are two variables, water concentration and oxygen concentration. We are able to calculate the surface energy according to

$$\gamma_{Surf} = \frac{1}{2A}\left(E_{TiO_2}^{slab} - \frac{nTi_{slab}}{nTi_{Bulk}} E_{TiO_2}^{Bulk} \right) - \Gamma_O \mu_O - \Gamma_{H_2O} \mu_{H_2O},$$

where A is the surface area, $E_{TiO_2}^{slab}$ is the DFT energy of the slab, nTi_{Slab} is the number of cations in the slab, nTi_{Bulk} is the number of cations in the bulk unit cell, $E_{TiO_2}^{Bulk}$ is the DFT energy of the bulk unit cell,

$$\Gamma_O = \frac{1}{2A}\left(nO_{Slab} - \frac{nO_{Bulk}}{nTi_{Bulk}} nTi_{Slab} \right),$$

$$\Gamma_{H_2O} = \frac{nH_2O}{2A},$$

where nO_{Slab} is the number of anions in the slab, nO_{Bulk} is the number of anions in the bulk and nH_2O is the number of adsorbing water molecules. Γ_O / Γ_{H_2O} is the excess oxygen / water at the surface and μ_O / μ_{H_2O} is the oxygen / water chemcial potential. Clearly Γ and mu will only matter when the surface is non stoichiometric.

Temperature

The above phase diagram is at 0K. It is possible to use experimental data from the NIST_JANAF database to make the chemical potential a temperature dependent term and thus generate a phase diagram at a temperature (T). This is done according to

$$\gamma_{Surf} = \frac{1}{2A}\left(E_{TiO_2}^{slab} - \frac{nTi_{Slab}}{nTi_{Bulk}}E_{TiO_2}^{Bulk}\right) - \Gamma_O\mu_O - \Gamma_{H_2O}\mu_{H_2O} - n_O\mu_O(T) - n_{H_2O}\mu_{H_2O}(T)$$

where

$$\mu_O(T) = \frac{1}{2}\mu_O(T)(0K, DFT) + \frac{1}{2}\mu_O(T)(0K, EXP) + \frac{1}{2}\Delta G_{O_2}(\Delta T, Exp),$$

μ_O (T) (0 K , DFT) is the 0K free energy of an isolated oxygen molecule evaluated with DFT, μ_O (T) (0 K , EXP) is the 0 K experimental Gibbs energy for oxygen gas and $Delta$ G_{O_2} (Δ T, Exp) is the Gibbs energy defined at temperature T as

$$\Delta G_{O_2}(\Delta T, Exp) = \frac{1}{2}[H(T, O_2) - H(0K, O_2)] - \frac{1}{2}T[S(T, O_2)].$$

This will generate a phase diagram at temperature (T)

Pressure

Chemical potential can be converted to pressure values according to

$$P = \frac{\mu_X}{k_BT}$$

where P is the pressure, μ is the chemical potential of species X, k_B is the Boltzmnann constant and T is the temperature.

Pressure vs temperature

Surfinpy has the functionality to generate phase diagrams as a function of pressure vs temperature based upon the methodology used in Molinari et al according to

$$\gamma_{adsorbed,T,P} = \gamma_{bare} + (C(E_{ads,T} - RTln(\frac{p}{p^o})$$

where $\gamma_{adsorbed,T,p}$ is the surface energy of the surface with adsorbed species at temperature (T) and pressure (P), γ_{bare} is the suface energy of the bare surface, C is the coverage of adsorbed species, E_{ads} is the adsorption energy,

$$E_{ads,T} = E_{slab,adsorbant} - (E_{slab,bare} + n_{H_2O}E_{H_2O,T})/n_{H_2O}$$

where $E_{slab,adsorbant}$ is the energy of the surface and the adsorbed species, n_{H_2O} is he number of adsorbed species,

$$E_{H_2O,(T)} = E_{H_2O,(g)} - TS_{(T)}$$

where $S_{(T)}$ is the experimental entropy of gaseous water in the standard state.

7.1.2 Bulk Theory

Bulk phase diagrams enable the comparison of the thermodynamic stability of various different bulk phases under different chemical potentials giving valuable insight in to the syntheis of solid phases. This theory example will consider a series of bulk phases which can be defined through a reaction scheme across all phases, thus for this example including MgO, H_2O and CO_2 as reactions and A as a generic product.

$$x\mathrm{MgO} + y\mathrm{H_2O} + z\mathrm{CO_2} \rightarrow \mathrm{A}$$

The system is in equilibrium when the chemical potentials of the reactants and product are equal; i.e. the change in Gibbs free energy is $\delta G_{T,p} = 0$.

$$\delta G_{T,p} = \mu_A - x\mu_{\mathrm{MgO}} - y\mu_{\mathrm{H_2O}} - z\mu_{\mathrm{CO_2}} = 0$$

Assuming that H_2O and CO_2 are gaseous species, μ_{CO_2} and μ_{H_2O} can be written as

$$\mu_{\mathrm{H_2O}} = \mu^0_{\mathrm{H_2O}} + \Delta\mu_{\mathrm{H_2O}}$$

and

$$\mu_{\mathrm{CO_2}} = \mu^0_{\mathrm{CO_2}} + \Delta\mu_{\mathrm{CO_2}}$$

The chemical potential μ^0_x is the partial molar free energy of any reactants or products (x) in their standard states, in this example we assume all solid components can be expressed as

$$\mu_{\mathrm{component}} = \mu^0_{\mathrm{component}}$$

Hence, we can now rearrange the equations to produce;

$$\mu^0_A - x\mu^0_{\mathrm{MgO}} - y\mu^0_{\mathrm{H_2O}} - z\mu^0_{CO_2} = y\Delta\mu_{H_2O} + z\Delta\mu_{CO_2}$$

As μ^0_A corresponds to the partial molar free energy of product A, we can replace the left side with the Gibbs free energy (ΔG^0_{f}).

$$\delta G_{T,p} = \Delta G^0_{\mathrm{f}} - y\Delta\mu_{\mathrm{H_2O}} - z\Delta\mu_{\mathrm{CO_2}}$$

At equilibrium $\delta G_{T,p} = 0$, and hence

$$\Delta G^0_{\mathrm{f}} = y\Delta\mu_{\mathrm{H_2O}} + z\Delta\mu_{\mathrm{CO_2}}$$

Thus, we can find the values of $\Delta\mu_{H_2O}$ and $\Delta\mu_{CO_2}$ (or $(\mathrm{p}_{H_2O})^y$ and $\mathrm{p}^z_{CO_2}$ when Mg-rich phases are in thermodynamic equilibrium; i.e. they are more or less stable than MgO. This procedure can then be applied to all phases to identify which is the most stable, provided that the free energy ΔG^0_f is known for each Mg-rich phase.

The free energy can be calculated using

$$\Delta G^0_f = \sum \Delta G^{0,\mathrm{products}}_f - \sum \Delta G^{0,\mathrm{reactants}}_f$$

Where for this example the free energy (G) is equal to the calculated DFT energy (U_0).

Temperature

The previous method will generate a phase diagram at 0 K. This is not representative of normal conditions. Temperature is an important consideration for materials chemistry and we may wish to evaluate the phase thermodynamic stability at various synthesis conditions.

As before the free energy can be calculated using;

$$\Delta G_f^0 = \sum \Delta G_f^{0,\text{products}} - \sum \Delta G_f^{0,\text{reactants}}$$

Where for this exmaple the free energy (G) for solid phases is equal to is equal to the calculated DFT energy (U_0). For gaseous species, the standard free energy varies significantly with temperature, and as DFT simulations are designed for condensed phase systems, we use experimental data to determine the temperature dependent free energy term for gaseous species, where S$_{expt}(T)$ is specific entropy value for a given T and H-H$^0(T)$ is the, both can be obtained from the NIST database and can be calculated as;

$$G = U_0 + (H - H^0(T) - TS_{\text{expt}}(T))$$

Pressure

In the previous tutorials we went through the process of generating a simple phase diagram for bulk phases and introducing temperature dependence for gaseous species. This useful however, sometimes it can be more beneficial to convert the chemical potenials (eVs) to partial presure (bar).

Chemical potential can be converted to pressure values using

$$P = \frac{\mu_O}{k_B T},$$

where P is the pressure, μ is the chemical potential of oxygen, k_B is the Boltzmnann constant and T is the temperature.

7.1.3 Vibrational Theory

The vibrational entropy allows for a more accurate calculation of phase diagrams without the need to include experimental corrections for solid phases.

The standard free energy varies significantly with temperature, and as DFT simulations are designed for condensed phase systems, we use experimental data to determine the temperature dependent free energy term for gaseous species obtained from the NIST database. In addition we also calculate the vibrational properties for the solid phases modifying the free energy (G) for solid phases to be;

$$\Delta G_f = U_0 + U_{\text{ZPE}} + A_{\text{vib}}$$

U$_0$ is the calculated internal energy from a DFT calculation, U$_{ZPE}$ is the zero point energy and S$_{vib}$ is the vibrational entropy.

$$U_{\text{ZPE}} = \sum_i^{3n} \frac{R\theta_i}{2}$$

where A$_{vib}$ is the vibrational Helmholtz free energy and defined as;

$$A_{\text{vib}} = \sum_i^{3n} RT \ln\left(1 - e^{-\theta_i/T}\right)$$

3n is the total number of vibrational modes, n is the number of species and θ_i is the characteristic vibrational temperature (frequency of the vibrational mode in Kelvin).

$$\theta_i = \frac{h\nu_i}{k_B}$$

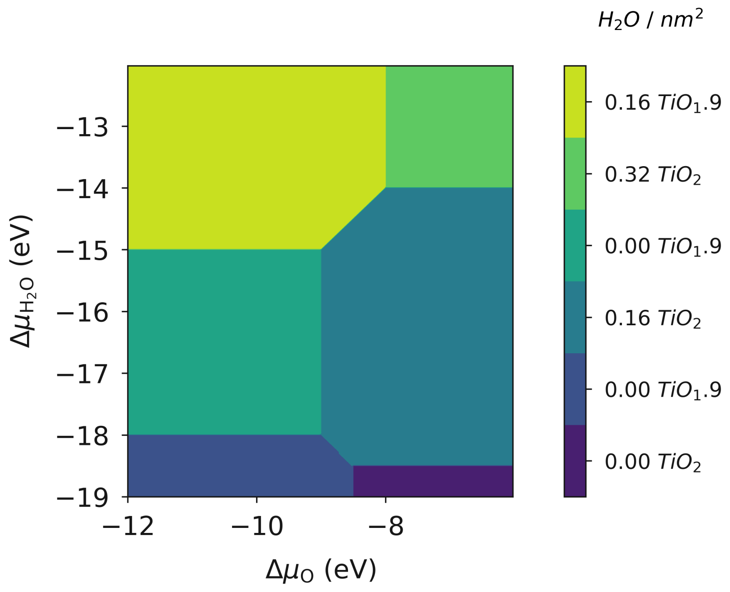
7.2 Gallery

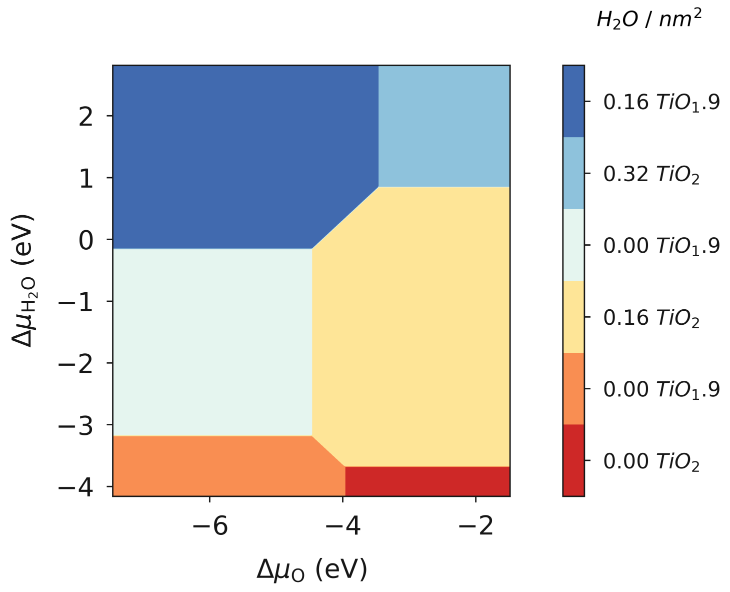
The gallery is a preview of some of the plots available in `surfinpy`. Clicking on a plot will provide a link to a tutorial for generating the plot.

7.2.1 Surfaces

chemical potential

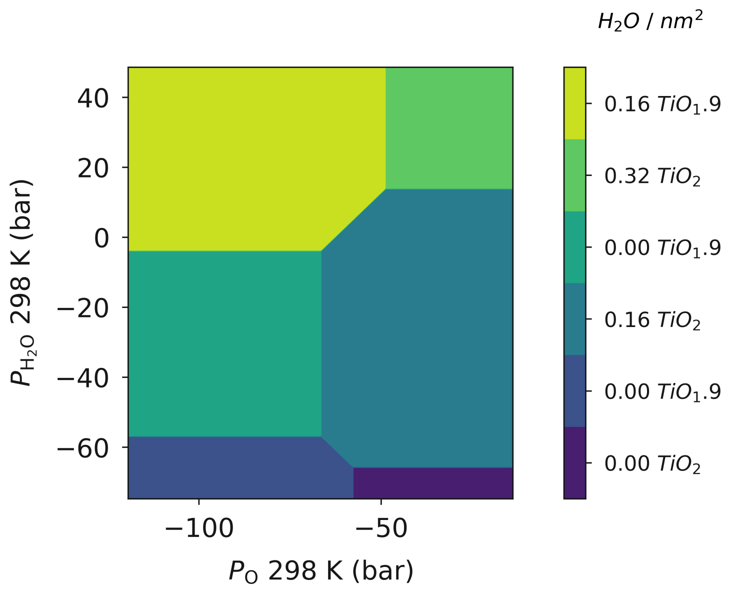
The following are examples of a phase diagram as a function of chemical potential. The first is the default output and the rest are generated by playing with the style and colourmap.





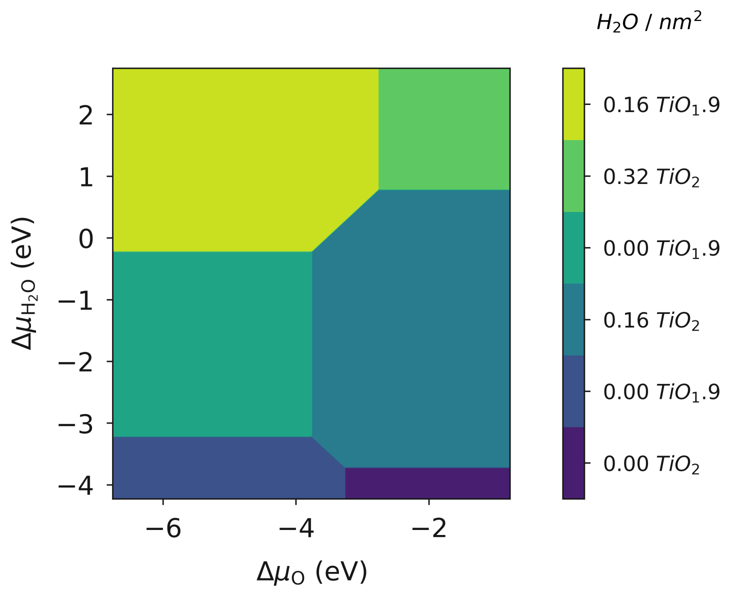
Pressure

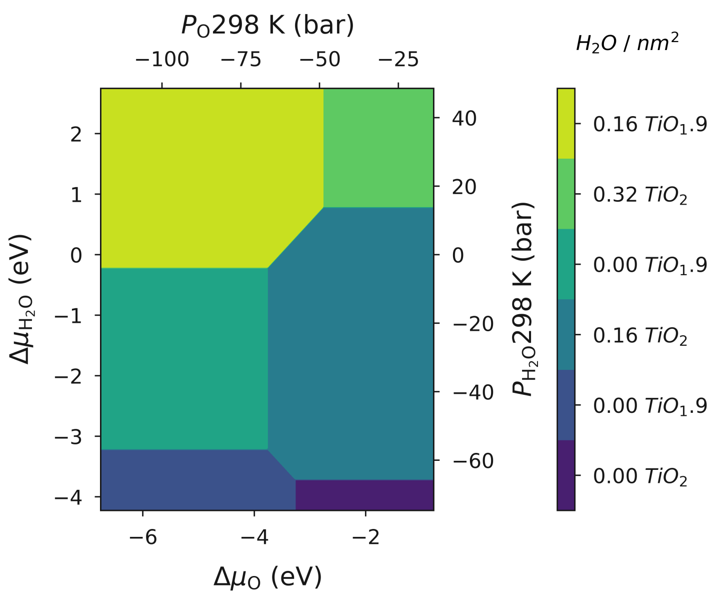
Chemical potential can be converted to pressure and a diagram with pressure of species A/B displayed.



Chemical potential and pressure

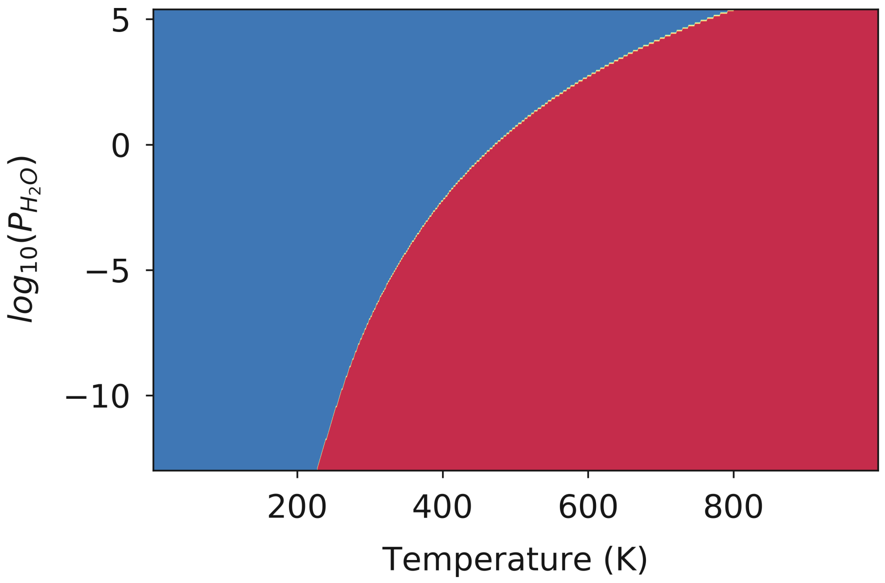
surfinpy can produce a plot with the chemical potential of A/B on axes X/Y and the pressure of A/B on axes X2/Y2.





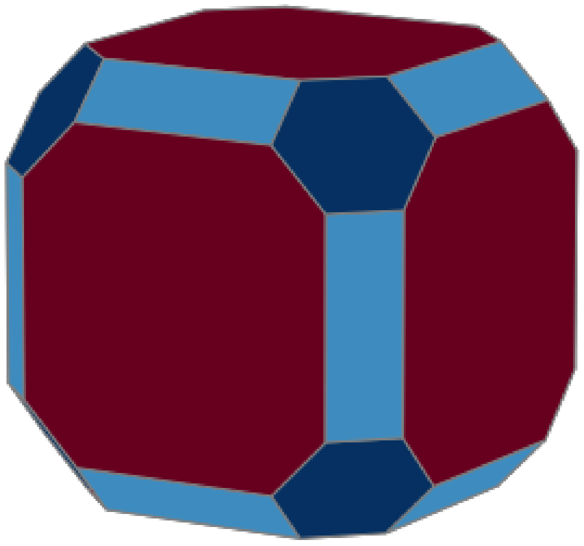
Temperature vs Pressure

surfinpy can produce simple pvt plots showing the relationship between a single species "A" at the surface e.g. water.



Particle Morphology

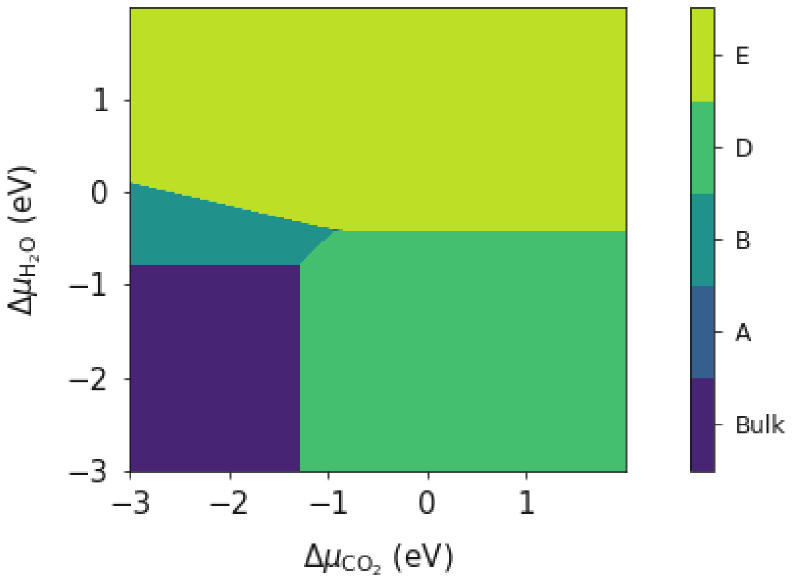
surfinpy provides examples of how to use the surface energy calculation alongside pymatgen to generate particle morphologies at different temperature and pressure values.



7.2.2 Bulk

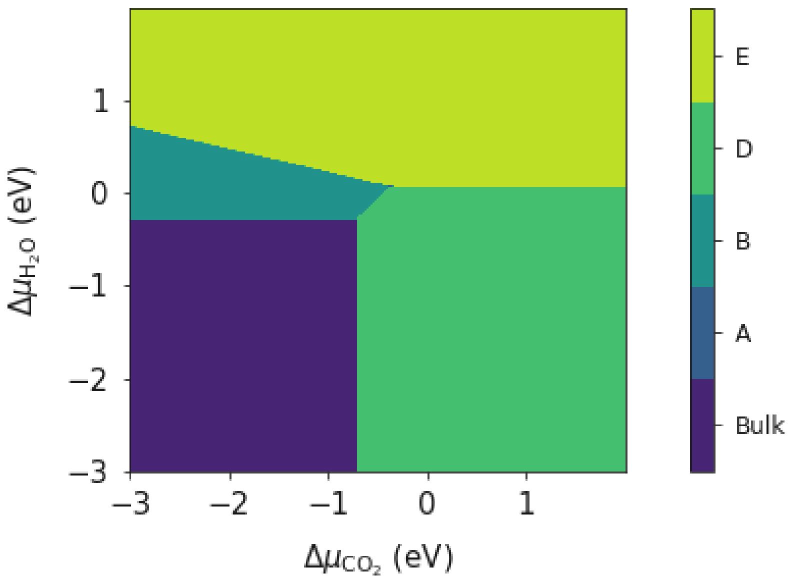
Chemical Potential

The following are examples of a phase diagram as a function of chemical potential.



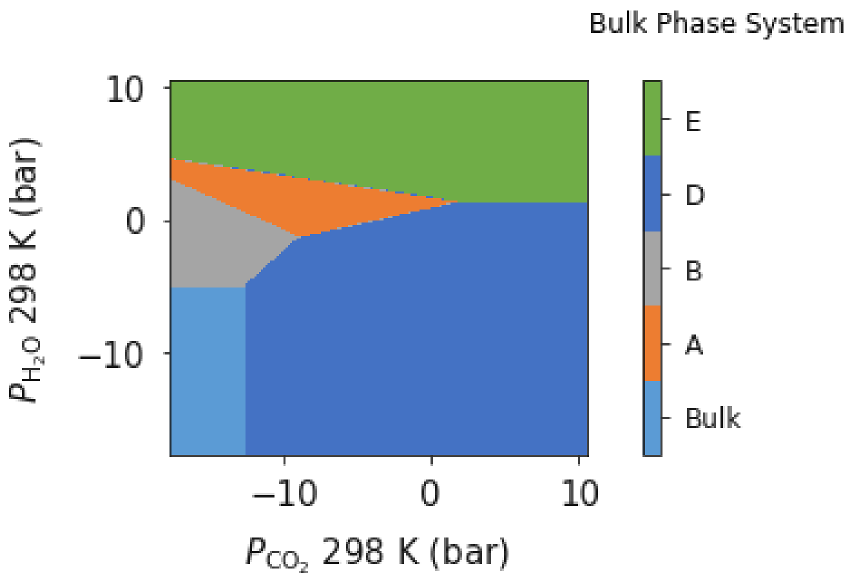
Temperature

The following are examples of a phase diagram as a function of chemical potential with a temperature contribution introduced.



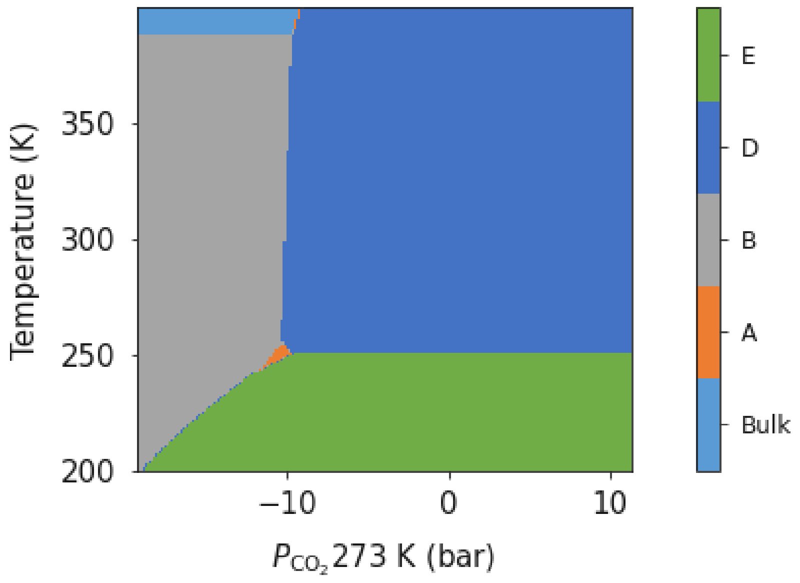
Pressure

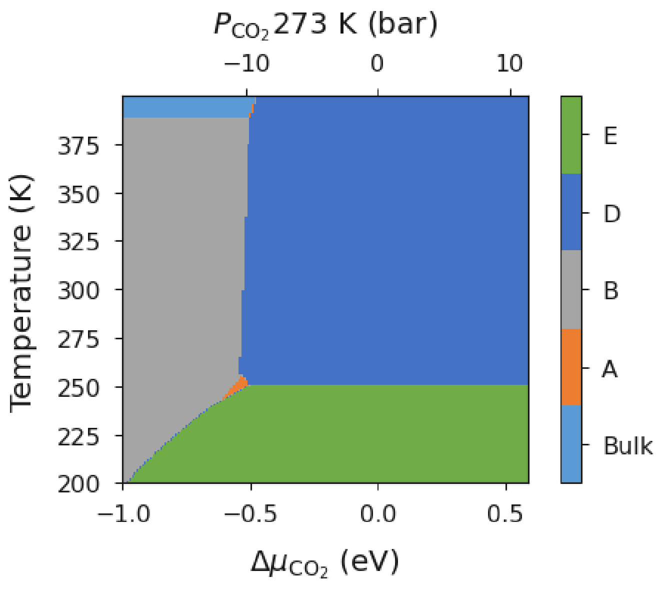
The following are examples of a phase diagram as a function of pressure.



Pressure vs Temperature

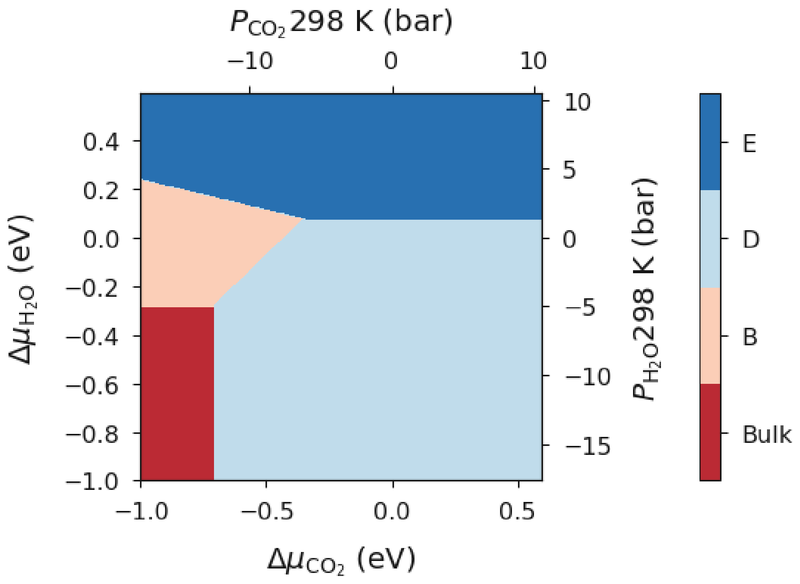
The following are examples of a phase diagram as a function of chemical potential, chemical potential and pressure, and temperature.

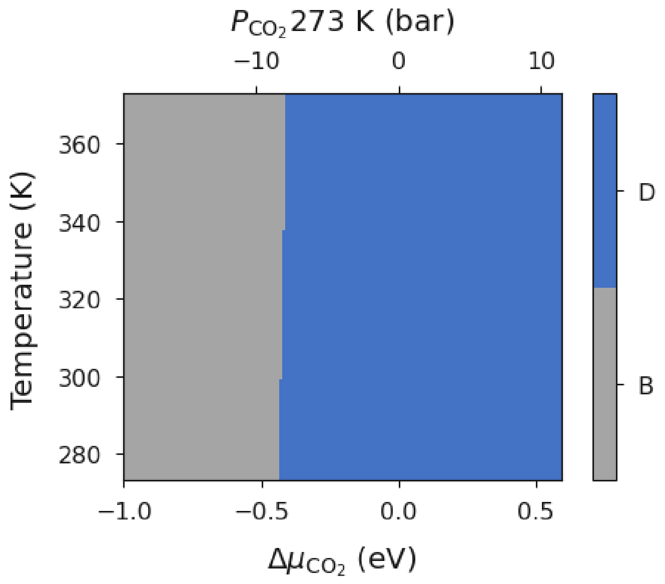


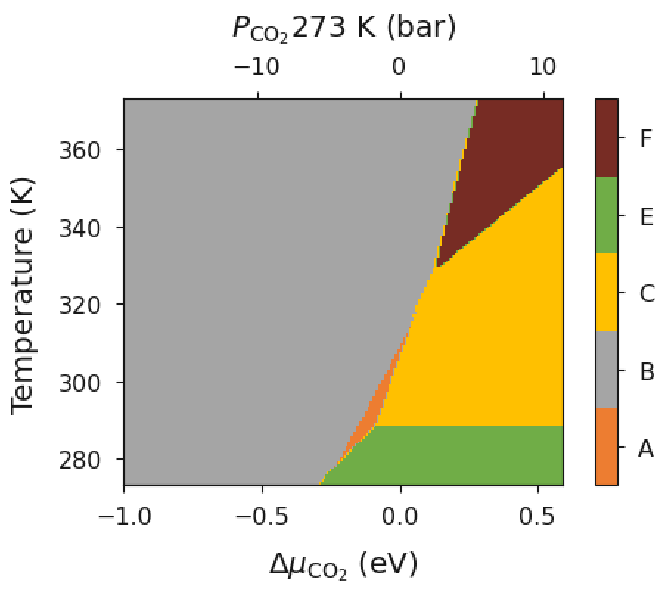


Vibrational Entropy

The following are examples of how to include the effects of vibrational entropy to the phase diagrams.







7.3 Tutorials

These tutorials are replicated in jupyter notebook form and contained within examples. The accompanying python scripts are also included here. All of code examples within these tutorials can be found in examples/Scripts.

7.3.1 Surfaces

Chemical Potential

The physical quantity that is used to define the stability of a surface with a given composition is its surface energy γ (J m^{-2}). At its core, *surfinpy* is a code that calculates the surface energy of different slabs and uses these surface energies to build a phase diagram. In this explantion of theory we will use the example of water adsorbing onto a surface of TiO_2 containing oxygen vacancies. In such an example there are two variables, water concentration and oxygen vacancy concentration. We are able to calculate the surface energy according to

$$\gamma_{Surf} = \frac{1}{2A}\left(E_{TiO_2}^{slab} - \frac{nTi_{slab}}{nTi_{Bulk}} E_{TiO_2}^{Bulk}\right) - \Gamma_O \mu_O - \Gamma_{H_2O}\mu_{H_2O},$$

where A is the surface area, $E_{TiO_2}^{slab}$ is the DFT energy of the slab, nTi_{Slab} is the number of cations in the slab, nTi_{Bulk} is the number of cations in the bulk unit cell, $E_{TiO_2}^{Bulk}$ is the DFT energy of the bulk unit cell and

$$\Gamma_O = \frac{1}{2A}\left(nO_{Slab} - \frac{nO_{Bulk}}{nTi_{Bulk}} nTi_{Slab}\right),$$

$$\Gamma_{H_2O} = \frac{nH_2O}{2A},$$

where nO_{Slab} is the number of anions in the slab, nO_{Bulk} is the number of anions in the bulk and nH_2O is the number of adsorbing water molecules. Γ_O / Γ_{H_2O} is the excess oxygen / water at the surface and μ_O / μ_{H_2O} is the oxygen / water chemcial potential. Clearly Γ and mu will only matter when the surface is non stoichiometric.

Usage

The first thing to do is input the data that we have generated from our DFT calculations. The input data needs to be contained within a dictionary. First we have created the dictionary for the bulk data, where `Cation` is the number of cations, `Anion` is the number of anions, `Energy` is the DFT energy and `F-Units` is the number of formula units.

```
bulk = {'Cation' : Cations in Bulk Unit Cell,
        'Anion' : Anions in Bulk Unit Cell,
        'Energy' :  Energy of Bulk Calculation,
        'F-Units' : Formula units in Bulk Calculation}
```

Next we create the slab dictionaries - one for each slab calculation or "phase". `Cation` is the number of cations, `X` is in this case the number of oxygen species (corresponding to the X axis of the phase diagram), `Y` is the number of in this case water molecules (corresponding to the Y axis of our phase diagram), `Area` is the surface area, `Energy` is the DFT energy, `Label` is the label for the surface (appears on the phase diagram) and finally `nSpecies` is the number of adsorbing species (In this case we have a surface with oxygen vacancies and adsorbing water molecules - so nSpecies is 1 as oxygen vacancies are not an adsorbing species, they are a constituent part of the surface).

```
surface = {'Cation': Cations in Slab,
           'X': Number of Species X in Slab,
           'Y': Number of Species Y in Slab,
           'Area': Surface area in the slab,
           'Energy': Energy of Slab,
           'Label': Label for phase,
           'nSpecies': How many species are non stoichiometric}
```

This data needs to be contained within a list. Don't worry about the order, *surfinpy* will sort that out for you.

We also need to declare the range in chemical potential that we want to consider. Again these exist in a dictionary. `Range` corresponds to the range of chemcial potential values to be considered and `Label` is the axis label.

```
deltaX = {'Range': Range of Chemical Potential,
          'Label': Species Label}
```

```
from surfinpy import mu_vs_mu

bulk = {'Cation' : 1, 'Anion' : 2, 'Energy' : -780.0, 'F-Units' : 4}

pure =      {'Cation': 24, 'X': 48, 'Y': 0, 'Area': 60.0,
             'Energy': -575.0,    'Label': 'Stoich',   'nSpecies': 1}
H2O =       {'Cation': 24, 'X': 48, 'Y': 2, 'Area': 60.0,
             'Energy': -612.0,    'Label': '1 Water', 'nSpecies': 1}
H2O_2 =     {'Cation': 24, 'X': 48, 'Y': 4, 'Area': 60.0,
             'Energy': -640.0,    'Label': '2 Water', 'nSpecies': 1}
H2O_3 =     {'Cation': 24, 'X': 48, 'Y': 8, 'Area': 60.0,
             'Energy': -676.0,    'Label': '3 Water', 'nSpecies': 1}
Vo =        {'Cation': 24, 'X': 46, 'Y': 0, 'Area': 60.0,
             'Energy': -558.0,    'Label': 'Vo', 'nSpecies': 1}
H2O_Vo =    {'Cation': 24, 'X': 46, 'Y': 2, 'Area': 60.0,
             'Energy': -594.0,   'Label': 'Vo + 1 Water', 'nSpecies': 1}
H2O_Vo_2 = {'Cation': 24, 'X': 46, 'Y': 4, 'Area': 60.0,
             'Energy': -624.0,   'Label': 'Vo + 2 Water', 'nSpecies': 1}
H2O_Vo_3 = {'Cation': 24, 'X': 46, 'Y': 6, 'Area': 60.0,
             'Energy': -640.0, 'Label': 'Vo + 3 Water', 'nSpecies': 1}
```

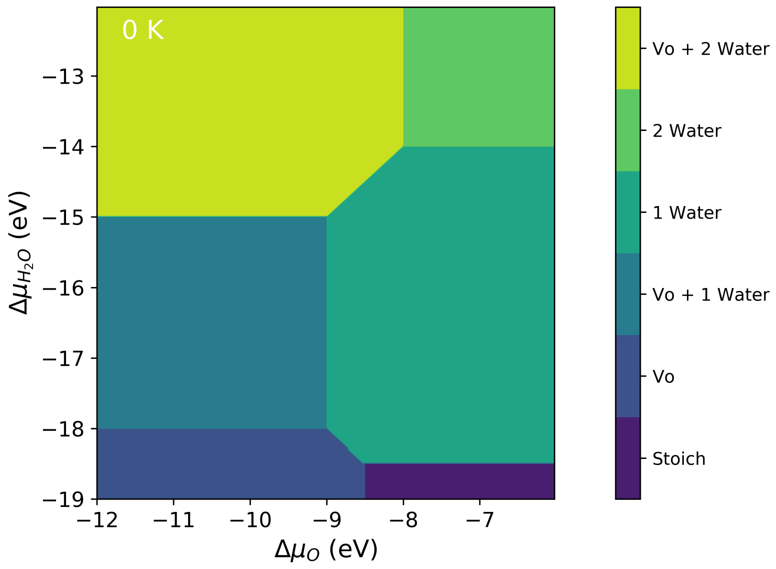
(continues on next page)

(continued from previous page)

```
H2O_Vo_4 = {'Cation': 24, 'X': 46, 'Y': 8, 'Area': 60.0,
            'Energy': -670.0, 'Label': 'Vo + 4 Water', 'nSpecies': 1}

data = [pure, H2O_2, H2O_Vo, H2O,  H2O_Vo_2, H2O_3, H2O_Vo_3,  H2O_Vo_4, Vo]

deltaX = {'Range': [ -12, -6],  'Label': 'O'}
deltaY = {'Range': [ -19, -12], 'Label': 'H_2O'}
```

This data will be used in all subsequent examples and will not be declared again. Once the data has been declared it is a simple two line process to generate the diagram.

```
system = mu_vs_mu.calculate(data, bulk, deltaX, deltaY)
system.plot_phase()
```



Temperature

The previous phase diagram is at 0K. It is possible to use experimental data from the NIST_JANAF database to make the chemical potential a temperature dependent term and thus generate a phase diagram at a temperature (T). Using oxygen as an example, this is done according to

$$\gamma_{Surf} = \frac{1}{2A}\left(E_{TiO_2}^{slab} - \frac{nTi_{Slab}}{nTi_{Bulk}}E_{TiO_2}^{Bulk}\right) - \Gamma_O\mu_O - \Gamma_{H_2O}\mu_{H_2O} - n_O\mu_O(T) - n_{H_2O}\mu_{H_2O}(T)$$

where

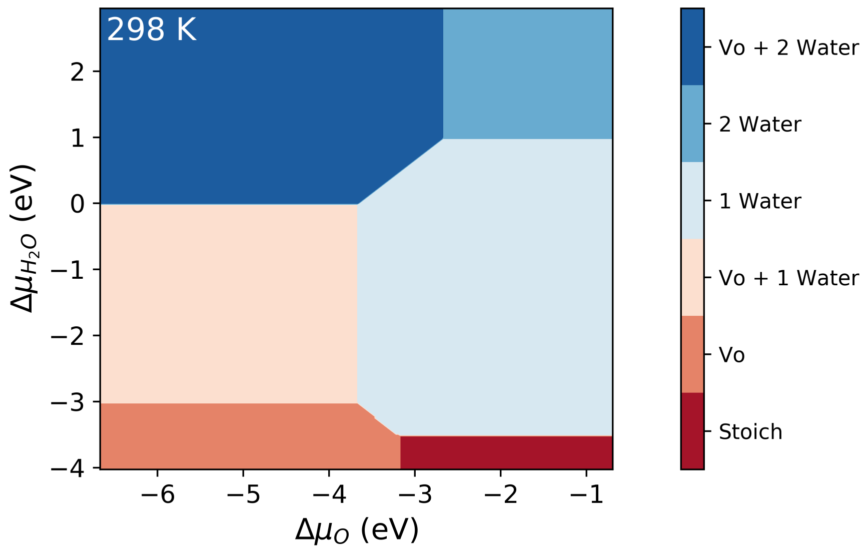
$$\mu_O(T) = \frac{1}{2}\mu_O(T)(0K, DFT) + \frac{1}{2}\mu_O(T)(0K, EXP) + \frac{1}{2}\Delta G_{O_2}(\Delta T, Exp),$$

μ_O (T) (0 K , DFT) is the 0K free energy of an isolated oxygen molecule evaluated with DFT, μ_O (T) (0 K , EXP) is the 0 K experimental Gibbs energy for oxygen gas and $Delta$ G_{O_2} (Δ T, Exp) is the Gibbs energy defined at temperature T as

$$\Delta G_{O_2}(\Delta T, Exp) = \frac{1}{2}[H(T, O_2) - H(0K, O_2)] - \frac{1}{2}T[S(T, O_2])$$

surfinpy has a built in function to read a NIST_JANAF table and calculate this temperature_correction for you. In the following example you will also see an example of how you can tweak the style and colourmap of the plot.

```
from surfinpy import mu_vs_mu

Oxygen_exp = mu_vs_mu.temperature_correction("O2.txt", 298)
Water_exp = mu_vs_mu.temperature_correction("H2O.txt", 298)

Oxygen_corrected = (-9.08 + -0.86 + Oxygen_exp)
Water_corrected = -14.84 + 0.55 + Water_exp

system =  mu_vs_mu.calculate(data, bulk, deltaX, deltaY,
                             x_energy=Oxygen_corrected,
                             y_energy=Water_corrected)
system.plot_phase(temperature=298, set_style="fast",
                  colourmap="RdBu")
```



Pressure

The chemical potential can be converted to pressure values according to

$$P = \frac{\mu_O}{k_B T}$$

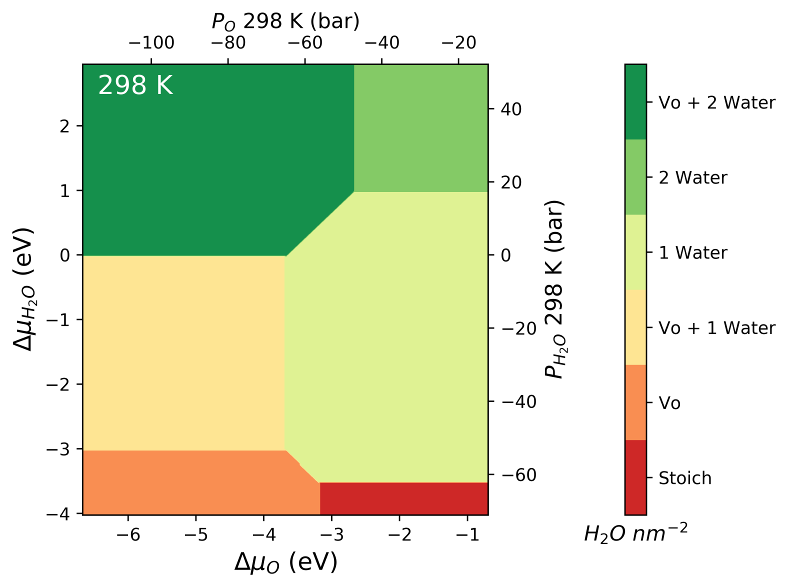
where P is the pressure, μ is the chemical potential of oxygen, k_B is the Boltzmnann constant and T is the temperature.

```
from surfinpy import mu_vs_mu

Oxygen_exp = mu_vs_mu.temperature_correction("O2.txt", 298)
Water_exp = mu_vs_mu.temperature_correction("H2O.txt", 298)

Oxygen_corrected = (-9.08 + -0.86 + Oxygen_exp)
Water_corrected = -14.84 + 0.55 + Water_exp
```

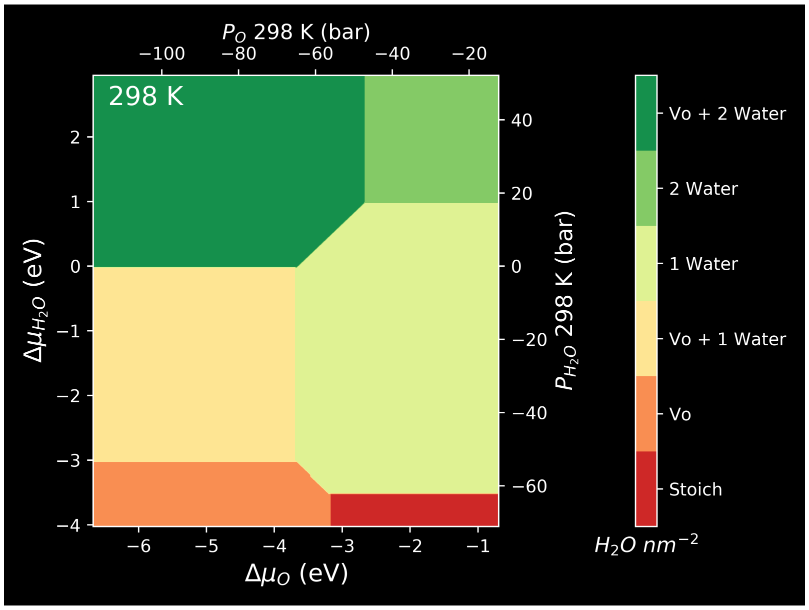
(continues on next page)

(continued from previous page)

```
system =  mu_vs_mu.calculate(data, bulk, deltaX, deltaY,
                             x_energy=Oxygen_corrected,
                             y_energy=Water_corrected)
system.plot_mu_p(output="Example_ggrd", colourmap="RdYlGn",
                 temperature=298)
```



```
system.plot_mu_p(output="Example_ggrd",
                 set_style="dark_background",
                 colourmap="RdYlGn",
                 temperature=298)
```

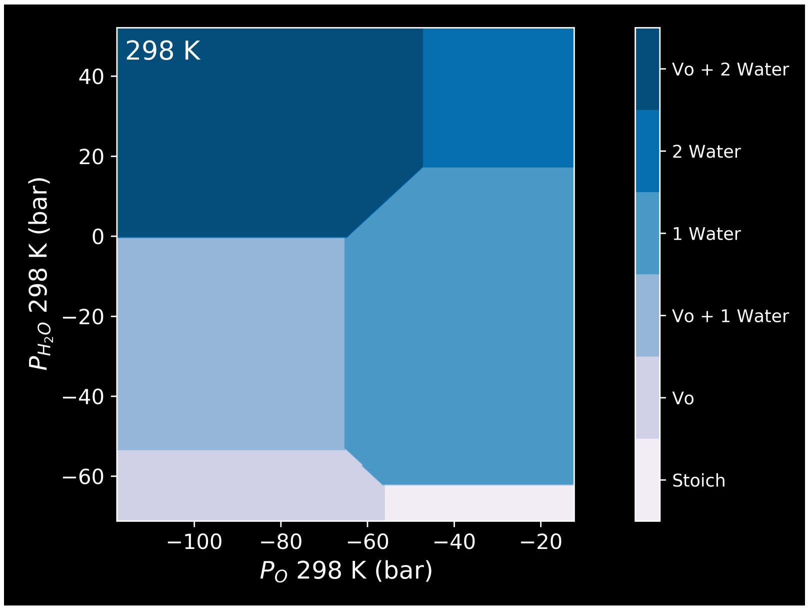


```
system.plot_pressure(output="Example_dark_rdgn",
                     set_style="dark_background",
```

(continues on next page)

(continued from previous page)

```
                          colourmap="PuBu",
                          temperature=298)
```



Pressure vs Temperature

Surfinpy has the functionality to generate phase diagrams as a function of pressure vs temperature based upon the methodology used in Molinari et al according to

$$\gamma_{adsorbed,T,P} = \gamma_{bare} + (C(E_{ads,T} - RTln(\frac{p}{p^o})$$

where $\gamma_{adsorbed,T,p}$ is the surface energy of the surface with adsorbed species at temperature (T) and pressure (P), γ_{bare} is the suface energy of the bare surface, C is the coverage of adsorbed species, E_{ads} is the adsorption energy,

$$E_{ads,T} = E_{slab,adsorbant} - (E_{slab,bare} + n_{H_2O}E_{H_2O,T})/n_{H_2O}$$

where $E_{slab,adsorbant}$ is the energy of the surface and the adsorbed species, n_{H_2O} is he number of adsorbed species,

$$E_{H_2O,(T)} = E_{H_2O,(g)} - TS_{(T)}$$

where $S_{(T)}$ is the experimental entropy of gaseous water in the standard state.

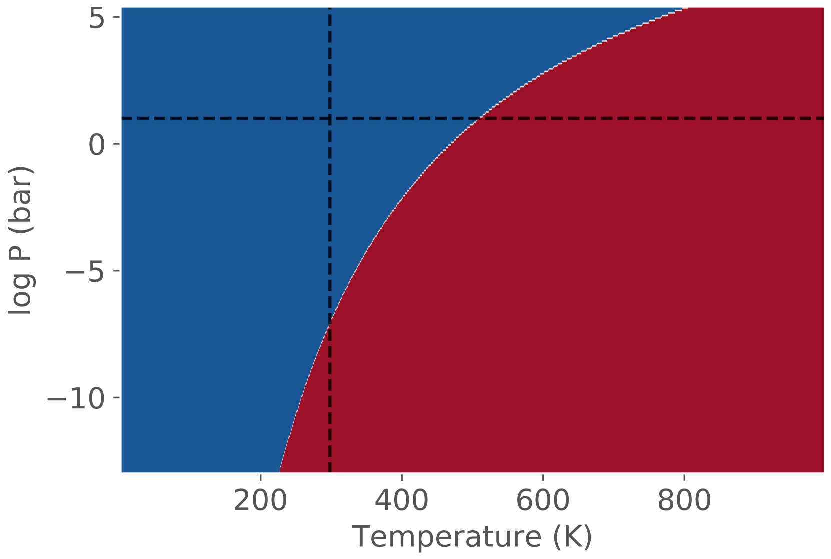
Usage

```
from surfinpy import utils as ut
from surfinpy import p_vs_t

adsorbant = -14.00
SE = 1.40

stoich = {'Cation': 24, 'X': 48, 'Y': 0, 'Area': 60.22,
```

(continues on next page)

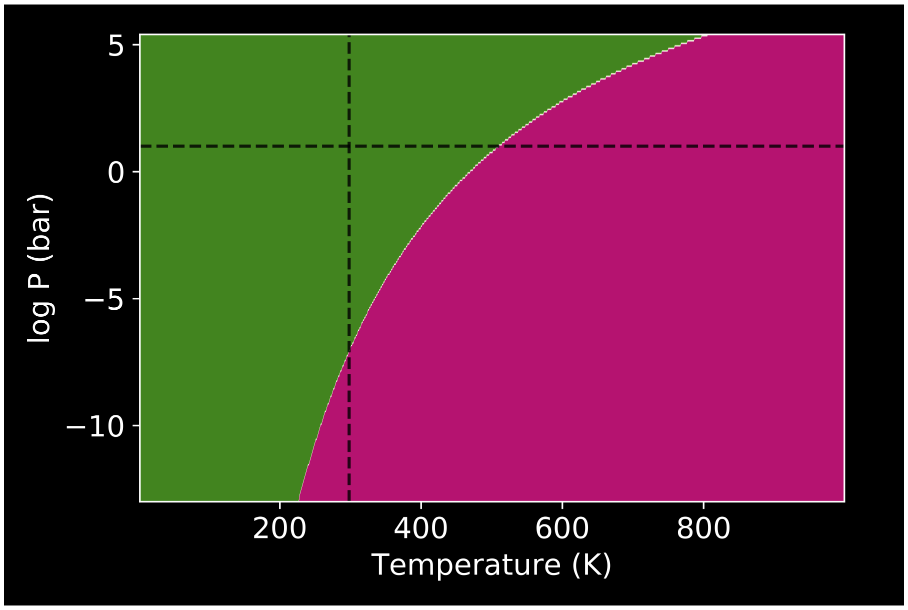
(continued from previous page)

```
           'Energy': -575.00, 'Label': 'Bare'}
H2O =    {'Cation': 24, 'X': 48, 'Y': 2, 'Area': 60.22,
           'Energy': -605.00, 'Label': '1 Water'}
H2O_2 =  {'Cation': 24, 'X': 48, 'Y': 8, 'Area': 60.22,
           'Energy': -695.00, 'Label': '2 Water'}
data = [H2O, H2O_2]

coverage = ut.calculate_coverage(data)

thermochem = ut.read_nist("H2O.txt")

system = p_vs_t.calculate(stoich, data, SE,
                          adsorbant,
                          thermochem,
                          coverage)
system.plot()
```



Alternatively you can also tweak the style

```
system.plot(output="dark_pvt.png",
            set_style="dark_background",
            colourmap="PiYG")
```



Particle Morphology

It is sometimes useful to use surface energies in order to generate particle morphologies. This tutorial demonstrates how to obtain surface energies for surfaces containing adsorbed species using *surfinpy*. With these you can then generate a wulff construction using pymatgen. A Wulff construction is a method to determine the equilibrium shape of a crystal. So by calculating the surface energies of multiple different surfaces, at different temperature and pressure values we can generate a particle morphology for the material, in the prescence of an adsorbing species, at a specific temperature and pressure.

surfinpy has a module called wulff that will return a surface energy at a given temperature and pressure value. These can then be used in conjunction with Pymatgen for a wulff construction. So first we need to declare the data for each surface and calculate the surface energies. As an aside, it is possible to provide multiple coverages, the return will be an array of surface energies, corresponding to each surface coverage, you would then select the minimum value with *np.amin()*

```
import numpy as np
from surfinpy import p_vs_t as pt
from surfinpy import wulff
from surfinpy import utils as ut
from pymatgen.core.surface import SlabGenerator,
                                  generate_all_slabs,
                                  Structure, Lattice
from pymatgen.analysis.wulff import WulffShape

adsorbant = -14.22
thermochem = ut.read_nist('H2O.txt')
```

The first thing to do is calculate the surface energy at a temperature and pressure value for each surface.

```
SE = 1.44
stoich =      {'M': 24, 'X': 48, 'Y': 0, 'Area': 60.22,
              'Energy': -575.66, 'Label': 'Stoich'}
Adsorbant_1 = {'M': 24, 'X': 48, 'Y': 2, 'Area': 60.22,
```

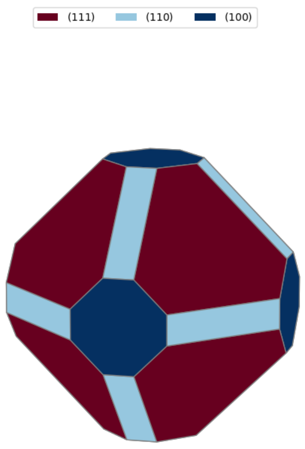
(continues on next page)

(continued from previous page)

```
               'Energy': -609.23, 'Label': '1 Species'}
data = [Adsorbant_1]
Surface_100_1 = wulff.calculate_surface_energy(stoich,
                                               data,
                                               SE,
                                               adsorbant,
                                               thermochem,
                                               298,
                                               0)

SE = 1.06
stoich =      {'M': 24, 'X': 48, 'Y': 0, 'Area': 85.12,
               'Energy': -672.95, 'Label': 'Stoich'}
Adsorbant_1 = {'M': 24, 'X': 48, 'Y': 2, 'Area': 85.12,
               'Energy': -705.0, 'Label': '1 Species'}
data = [Adsorbant_1]
Surface_110_1 = wulff.calculate_surface_energy(stoich,
                                               data,
                                               SE,
                                               adsorbant,
                                               thermochem,
                                               298,
                                               0)

SE = 0.76
stoich =      {'M': 24, 'X': 48, 'Y': 0, 'Area': 77.14,
               'Energy': -579.61, 'Label': 'Stoich'}
Adsorbant_1 = {'M': 24, 'X': 48, 'Y': 2, 'Area': 77.14,
               'Energy': -609.24, 'Label': '1 Species'}
data = [Adsorbant_1]
Surface_111_1 = wulff.calculate_surface_energy(stoich,
                                               data,
                                               SE,
                                               adsorbant,
                                               thermochem,
                                               298,
                                               0)
```

The with these surface energies we can build a particle morphology using pymatgen

```
lattice = Lattice.cubic(5.411)
ceo = Structure(lattice,["Ce", "O"],
               [[0,0,0], [0.25,0.25,0.25]])
surface_energies_ceo = {(1,1,1): np.amin(Surface_111_1),
                        (1,1,0): np.amin(Surface_110_1),
                        (1,0,0): np.amin(Surface_100_1)}

miller_list = surface_energies_ceo.keys()
e_surf_list = surface_energies_ceo.values()

wulffshape = WulffShape(ceo.lattice, miller_list, e_surf_list)
wulffshape.show(color_set="RdBu", direction=(1.00, 0.25, 0.25))
```



7.3.2 Bulk

Chemical Potential

In this tutorial we learn how to generate a basic bulk phase diagram from DFT energies. This enables the comparison of the thermodynamic stability of various different bulk phases under different chemical potentials giving valuable insight in to the syntheis of solid phases. This example will consider a series of bulk phases which can be defined through a reaction scheme across all phases, thus for this example including Bulk, H_2O and CO_2 as reactions and A as a generic product.

$$x\text{Bulk} + y\text{H}_2\text{O} + z\text{CO}_2 \rightarrow \text{A}$$

The system is in equilibrium when the chemical potentials of the reactants and product are equal; i.e. the change in Gibbs free energy is $\delta G_{T,p} = 0$.

$$\delta G_{T,p} = \mu_A - x\mu_{\text{Bulk}} - y\mu_{\text{H}_2\text{O}} - z\mu_{\text{CO}_2} = 0$$

Assuming that H_2O and CO_2 are gaseous species, μ_{CO_2} and μ_{H_2O} can be written as

$$\mu_{\text{H}_2\text{O}} = \mu^0_{\text{H}_2\text{O}} + \Delta\mu_{\text{H}_2\text{O}}$$

and

$$\mu_{\text{CO}_2} = \mu^0_{\text{CO}_2} + \Delta\mu_{\text{CO}_2}$$

The chemical potential μ^0_x is the partial molar free energy of any reactants or products (x) in their standard states, in this example we assume all solid components can be expressed as

$$\mu_{\text{component}} = \mu^0_{\text{component}}$$

Hence, we can now rearrange the equations to produce;

$$\mu^0_A - x\mu^0_{\text{Bulk}} - y\mu^0_{\text{H}_2\text{O}} - z\mu^0_{CO_2} = y\Delta\mu_{H_2O} + z\Delta\mu_{CO_2}$$

As μ_A^0 corresponds to the partial molar free energy of product A, we can replace the left side with the Gibbs free energy (ΔG_f^0).

$$\delta G_{T,p} = \Delta G_\mathrm{f}^0 - y\Delta\mu_{\mathrm{H_2O}} - z\Delta\mu_{\mathrm{CO_2}}$$

At equilibrium $\delta G_{T,p} = 0$, and hence

$$\Delta G_\mathrm{f}^0 = y\Delta\mu_{\mathrm{H_2O}} + z\Delta\mu_{\mathrm{CO_2}}$$

Thus, we can find the values of $\Delta\mu_{H_2O}$ and $\Delta\mu_{CO_2}$ (or $(\mathrm{p}_{H_2O})^y$ and $\mathrm{p}_{CO_2}^z$ when solid phases are in thermodynamic equilibrium; i.e. they are more or less stable than Bulk. This procedure can then be applied to all phases to identify which is the most stable, provided that the free energy ΔG_f^0 is known for each solid phase.

The free energy can be calculated using

$$\Delta G_f^0 = \sum \Delta G_f^{0,\mathrm{products}} - \sum \Delta G_f^{0,\mathrm{reactants}}$$

Where for this tutorial the free energy (G) is equal to the calculated DFT energy (U_0).

```
import matplotlib.pyplot as plt
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import data
```

The first thing to do is input the data that we have generated from our DFT simulations. The input data needs to be contained within a class. First we have created the class for the bulk data, where 'Cation' is the number of cations, 'Anion' is the number of anions, 'Energy' is the DFT energy and 'F-Units' is the number of formula units.

```
bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10)
```

Next we create the bulk phases classes - one for each phase. 'Cation' is the number of cations, 'x' is in this case the number of water species (corresponding to the X axis of the phase diagram), 'y' is the number of in this case CO_2 molecules (corresponding to the Y axis of our phase diagram), 'Energy' is the DFT energy and finally 'Label' is the label for the phase (appears on the phase diagram).

```
Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92.0, label = "Bulk")
A = data.DataSet(cation = 10, x = 5, y = 20, energy = -468.0, label = "A")
B = data.DataSet(cation = 10, x = 0, y = 10, energy = -228.0, label = "B")
C = data.DataSet(cation = 10, x = 10, y = 30, energy = -706.0, label = "C")
D = data.DataSet(cation = 10, x = 10, y = 0, energy = -310.0, label = "D")
E = data.DataSet(cation = 10, x = 10, y = 50, energy = -972.0, label = "E")
F = data.DataSet(cation = 10, x = 8, y = 10, energy = -398.0, label = "F")
```

Next we need to create a list of our data. Don't worry about the order, *surfinpy* will sort that out for you.

```
data = [Bulk, A, B, C,  D, E, F]
```

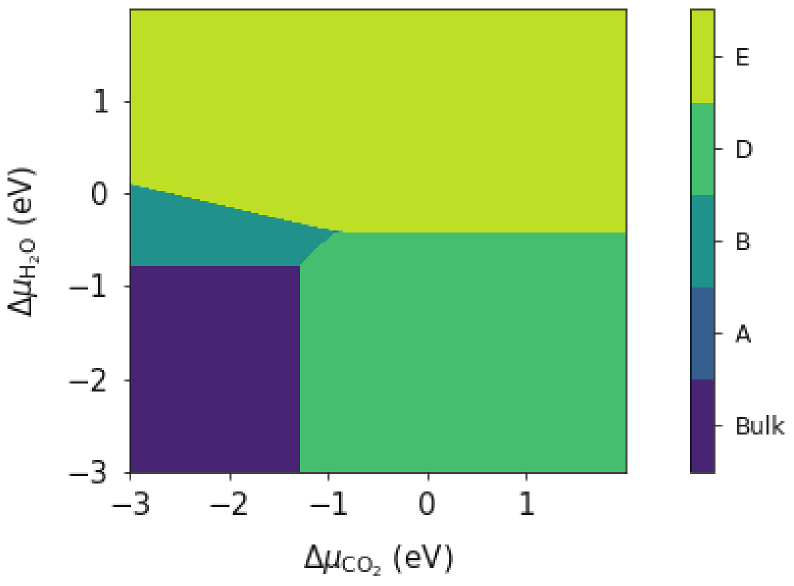
We now need to generate our X and Y axis, or more appropriately, our chemical potential values. These exist in a dictionary. 'Range' corresponds to the range of chemcial potential values to be considered and 'Label' is the axis label. Additionally, the x and y energy need to be specified.

```
deltaX = {'Range': Range of Chemical Potential,
          'Label': Species Label}
```

```
deltaX = {'Range': [ -3, 2],  'Label': 'CO_2'}
deltaY = {'Range': [ -3, 2], 'Label': 'H_2O'}
x_energy=-20.53412969
y_energy=-12.83725889
```

And finally we can generate our plot using these 6 variables of data.

```
system = bmvm.calculate(data, bulk, deltaX, deltaY, x_energy, y_energy)

ax = system.plot_phase()
plt.show()
```



Temperature

In the previous example we generated a phase diagram at 0 K. However, this is not representative of normal conditions. Temperature is an important consideration for materials chemistry and we may wish to evaluate the phase thermodynamic stability at various synthesis conditions. This example will again be using the $Bulk - CO_2 - H_2O$ system.

As before the free energy can be calculated using;

$$\Delta G_f^0 = \sum \Delta G_f^{0,\text{products}} - \sum \Delta G_f^{0,\text{reactants}}$$

Where for this tutorial the free energy (G) for solid phases is equal to is equal to the calculated DFT energy (U_0). For gaseous species, the standard free energy varies significantly with temperature, and as DFT simulations are designed for condensed phase systems, we use experimental data to determine the temperature dependent free energy term for gaseous species, where S$_{expt}(T)$ is specific entropy value for a given T and H-H$^0(T)$ is the , both can be obtained from the NIST database and can be calculated as;

$$G = U_0 + (H - H^0(T) - TS_{\text{expt}}(T))$$

```
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import data
```

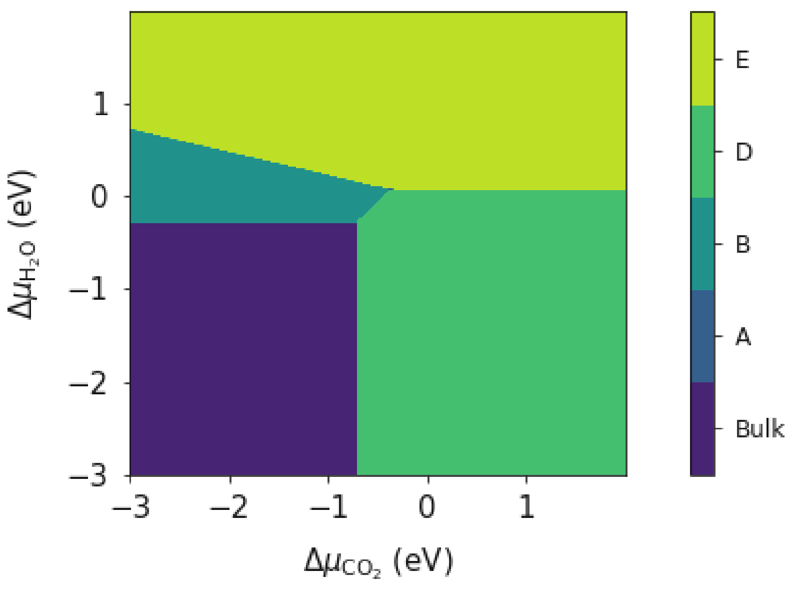
```
bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10)

Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92.0, label = "Bulk")
A = data.DataSet(cation = 10, x = 5, y = 20, energy = -468.0, label = "A")
B = data.DataSet(cation = 10, x = 0, y = 10, energy = -228.0, label = "B")
C = data.DataSet(cation = 10, x = 10, y = 30, energy = -706.0, label = "C")
D = data.DataSet(cation = 10, x = 10, y = 0, energy = -310.0, label = "D")
E = data.DataSet(cation = 10, x = 10, y = 50, energy = -972.0, label = "E")
F = data.DataSet(cation = 10, x = 8, y = 10, energy = -398.0, label = "F")
data = [Bulk, A, B, C,  D, E, F]

x_energy=-20.53412969
y_energy=-12.83725889
```

In order to calculate S$_{expt}(T)$ for H_2O and CO_2 we need to use experimental data from the NSIT JANAF database. As a user you will need to download the tables for the species you are interested in (in this example water and carbon dioxide). *surfinpy* has a function that can read this data, assuming it is in the correct format and calculate the temperature correction for you. Provide the path to the file and the temperature you want.

```
CO2_exp = ut.fit_nist("CO2.txt")[298]
Water_exp = ut.fit_nist("H2O.txt")[298]

CO2_corrected = x_energy + CO2_exp
Water_corrected = y_energy + Water_exp

deltaX = {'Range': [ -3, 2],  'Label': 'CO_2'}
deltaY = {'Range': [ -3, 2], 'Label': 'H_2O'}
```

CO2_corrected and H2O_corrected are now temperature depenent terms correcsponding to a temperature of 298 K. The resulting phase diagram will now be at a temperature of 298 K.

```
system = bmvm.calculate(data, bulk, deltaX, deltaY, x_energy=CO2_corrected, y_
↪energy=Water_corrected)

system.plot_phase(temperature=298)
```



Pressure

In the previous example we went through the process of generating a simple phase diagram for bulk phases and introducing temperature dependence for gaseous species. This useful however, sometimes it can be more beneficial to convert the chemical potenials (eVs) to partial presure (bar).

Chemical potential can be converted to pressure values using

$$P = \frac{\mu_O}{k_B T},$$

where P is the pressure, μ is the chemical potential of oxygen, k_B is the Boltzmnann constant and T is the temperature.

```python
import matplotlib.pyplot as plt
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import data

colors = ['#5B9BD5', '#4472C4', '#A5A5A5', '#772C24', '#ED7D31', '#FFC000', '#70AD47']
```

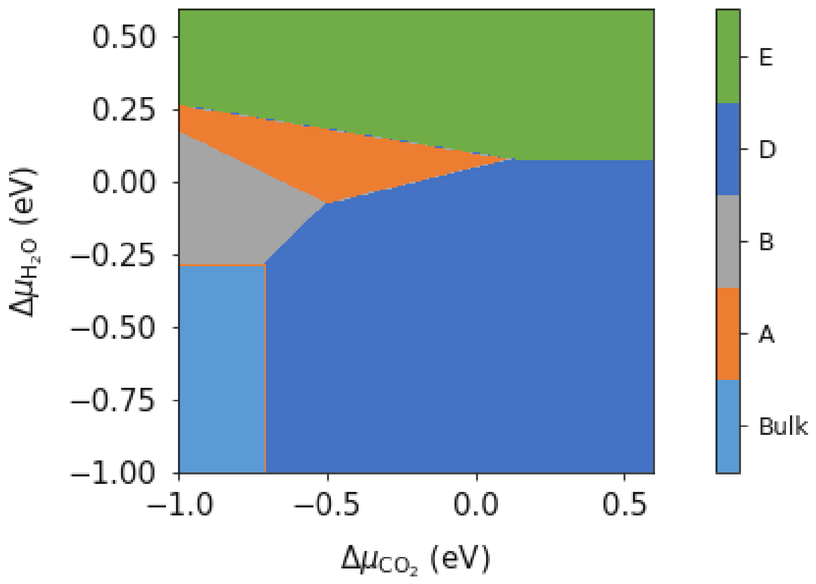
Additionally, *surfinpy* has the functionality to allow you to choose which colours are used for each phase. Specify within the DataSet class color.

```python
bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10)

Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92.0, color=colors[0], label =
↪"Bulk")
A = data.DataSet(cation = 10, x = 10, y = 0, energy = -310.0, color=colors[1], label = "A
↪")
B = data.DataSet(cation = 10, x = 0, y = 10, energy = -228.0, color=colors[2], label = "B
↪")
C = data.DataSet(cation = 10, x = 8, y = 10, energy = -398.0, color=colors[3], label = "C
↪")
```

(continues on next page)

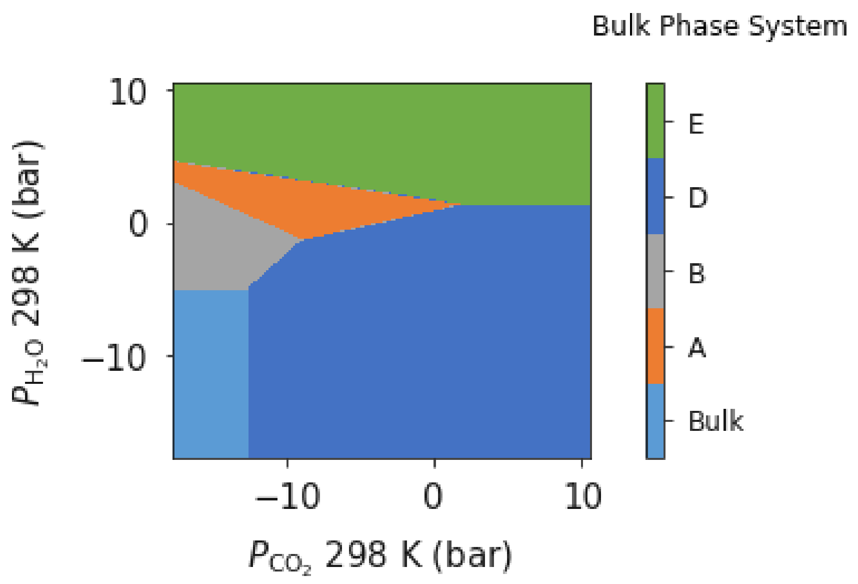
(continued from previous page)

```
D = data.DataSet(cation = 10, x = 5, y = 20, energy = -468.0, color=colors[4], label = "D
↪")
E = data.DataSet(cation = 10, x = 10, y = 30, energy = -706.0, color=colors[5], label =
↪"E")
F = data.DataSet(cation = 10, x = 10, y = 50, energy = -972.0, color=colors[6], label =
↪"F")

data = [Bulk, A, B, C,  D, E, F]

x_energy=-20.53412969
y_energy=-12.83725889

CO2_exp = ut.fit_nist("CO2.txt")[298]
Water_exp = ut.fit_nist("H2O.txt")[298]

CO2_corrected = x_energy + CO2_exp
Water_corrected = y_energy + Water_exp

deltaX = {'Range': [ -1, 0.6],  'Label': 'CO_2'}
deltaY = {'Range': [ -1, 0.6], 'Label': 'H_2O'}

system = bmvm.calculate(data, bulk, deltaX, deltaY, x_energy=CO2_corrected, y_
↪energy=Water_corrected)

system.plot_phase()
```



To convert chemical potential to pressure use the plot_pressure command and the temperature at which the pressure is calculated. For this example we have used 298 K.

```
system.plot_pressure(temperature=298)
```



Pressure vs Temperature

In the previous example, we showed how experimental data could be used to determine the temperature dependent free energy term for gaseous species and then plot a phase diagram that represents 298 K. This same method can be used in conjuction with a temperature range to produce a phase diagram of temperature as a function of pressure (or chemical potential). This is an important step to producing relatable phase diagrams that can be compared to experimental findings.

To reiterate, the free energy can be calculated using;

$$\Delta G_f^0 = \sum \Delta G_f^{0,\text{products}} - \sum \Delta G_f^{0,\text{reactants}}$$

Where for this tutorial the free energy (G) for solid phases is equal to is equal to the calculated DFT energy (U_0). For gaseous species, the standard free energy varies significantly with temperature, and as DFT simulations are designed for condensed phase systems, we use experimental data to determine the temperature dependent free energy term for gaseous species, where $S_{text{expt}}(T)$ is specific entropy value for a given T and $H-H^0(T)$ is the , both can be obtained from the NIST database and can be calculated as;

$$G = U_0 + (H - H^0(T) - TS_{\text{expt}}(T))$$

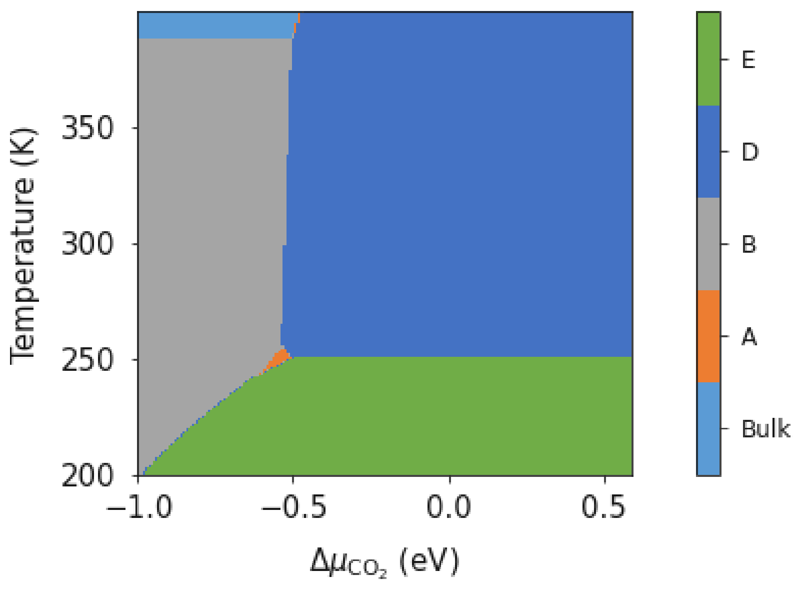
```
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import bulk_mu_vs_t as bmvt
from surfinpy import data
import numpy as np
from seaborn import palettes

import matplotlib.pyplot as plt
colors = ['#5B9BD5', '#4472C4', '#A5A5A5', '#772C24', '#ED7D31', '#FFC000', '#70AD47']
```

temperature_range sets the temperature range which is calculated for the phase diagram and needs to be specified within the data ReferenceDataSet and DataSet.

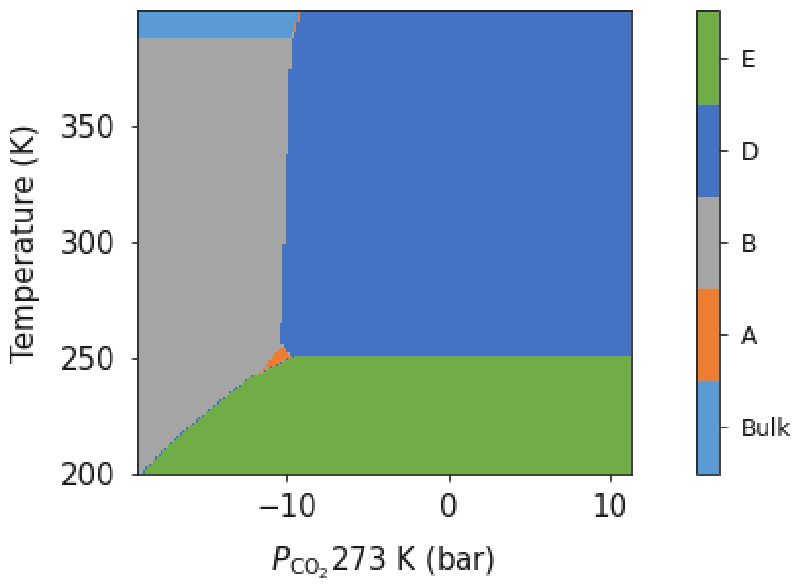
```
temperature_range = [200, 400]

bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10, file =
↪'bulk_vib.yaml', temp_range=temperature_range)


Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92., color=colors[0],
                label = "Bulk", file = 'ref_files/bulk_vib.yaml'',
                funits = 10, temp_range=temperature_range)

D = data.DataSet(cation = 10, x = 10, y = 0, energy = -310.,  color=colors[1],
                label = "D", file = 'ref_files/D_vib.yaml',
                funits =  10, temp_range=temperature_range)

B = data.DataSet(cation = 10, x = 0, y = 10, energy = -227.,  color=colors[2],
                label = "B", file = 'ref_files/B_vib.yaml',
                funits =  10, temp_range=temperature_range)

F = data.DataSet(cation = 10, x = 8, y = 10, energy = -398.,  color=colors[3],
                label = "F", file = 'ref_files/F_vib.yaml',
                funits =  2, temp_range=temperature_range)

A = data.DataSet(cation = 10, x = 5, y = 20, energy = -467.,  color=colors[4],
                label = "A", file = 'ref_files/A_vib.yaml',
                funits = 5, temp_range=temperature_range)

C = data.DataSet(cation = 10, x = 10, y = 30, energy = -705.,  color=colors[5],
                label = "C", file = 'ref_files/C_vib.yaml',
                funits = 10, temp_range=temperature_range)

E = data.DataSet(cation = 10, x = 10, y = 50, energy = -971.,  color=colors[6],
                label = "E", file = 'ref_files/E_vib.yaml',
                funits =  10, temp_range=temperature_range)

data = [Bulk, A, B, C,  D, E, F]
```

deltaZ specifies the temperature range which is plotted (Note that this must be the same as temperature_range). mu_y is the chemical potential (eV) of third component, in this example we use a chemical potential of water = 0 eV which is equivalent to 1 bar pressure.

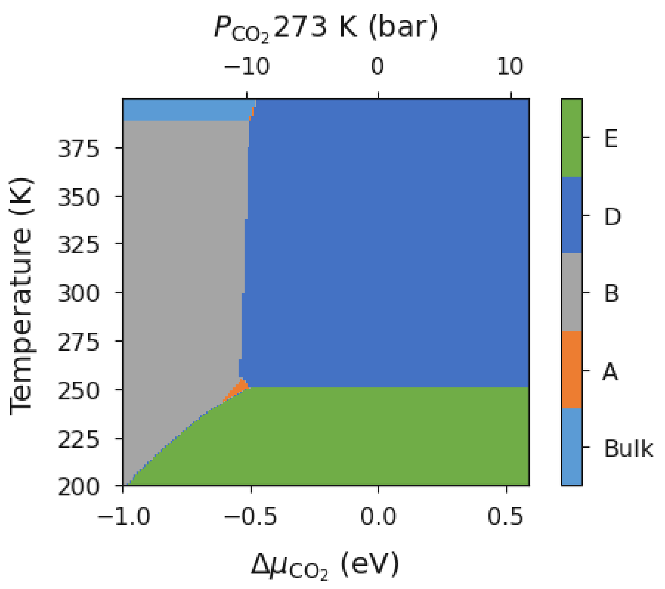
```
deltaX = {'Range': [ -1, 0.6],  'Label': 'CO_2'}
deltaZ = {'Range': [ 200, 400], 'Label': 'Temperature'}
x_energy=-20.53412969
y_energy=-12.83725889
mu_y = 0

exp_x = ut.temperature_correction_range("CO2.txt", deltaZ)
exp_y = ut.temperature_correction_range("H2O.txt", deltaZ)

system = bmvt.calculate(data, bulk, deltaX, deltaZ, x_energy, y_energy, mu_y, exp_x, exp_
↪y)
ax = system.plot_mu_vs_t()
```



```
system.plot_p_vs_t(temperature=273, set_style="seaborn-dark-palette", colourmap="RdYlBu")
```



```
system.plot_mu_vs_t_vs_p(temperature=273, set_style="seaborn-dark-palette", colourmap=
↪"RdYlBu")
```



Vibrational Entropy

In this example we will expand this methodology to calculate the vibrational properties for solid phases (i.e. zero point energy, vibrational entropy) and include these values in the generation of the phase diagrams. This allows for a more accurate calculation of phase diagrams without the need to include experimental corrections for solid phases.

As with previous examples, the standard free energy varies significantly with temperature, and as DFT simulations are designed for condensed phase systems, we use experimental data to determine the temperature dependent free energy term for gaseous species obtained from the NIST database. In addition we also calculate the vibrational properties for the solid phases modifying the free energy (G) for solid phases to be;

$$\Delta G_f = U_0 + U_{\text{ZPE}} + A_{\text{vib}}$$

U$_0$ is the calculated internal energy from a DFT calculation, U$_{ZPE}$ is the zero point energy and S$_{vib}$ is the vibrational entropy.

$$U_{\text{ZPE}} = \sum_{i}^{3n} \frac{R\theta_i}{2}$$

where A$_{vib}$ is the vibrational Helmholtz free energy and defined as;

$$A_{\text{vib}} = \sum_{i}^{3n} RT \ln\left(1 - e^{-\theta_i/T}\right)$$

3n is the total number of vibrational modes, n is the number of species and θ_i is the characteristic vibrational temperature (frequency of the vibrational mode in Kelvin).

$$\theta_i = \frac{h\nu_i}{k_B}$$

```
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import data
```

(continues on next page)

(continued from previous page)

```
temperature_range = [298, 299]

bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10, file =
↪'bulk_vib.yaml', entropy=True, zpe=True, temp_range=temperature_range)
```

In addition to entropy and zpe keyword you must provide the a file containing the vibrational modes and number of formula units used in taht calculations. You must create the yaml file using the following format

```
F-Units : number
Frequencies :
- mode1
- mode2
```

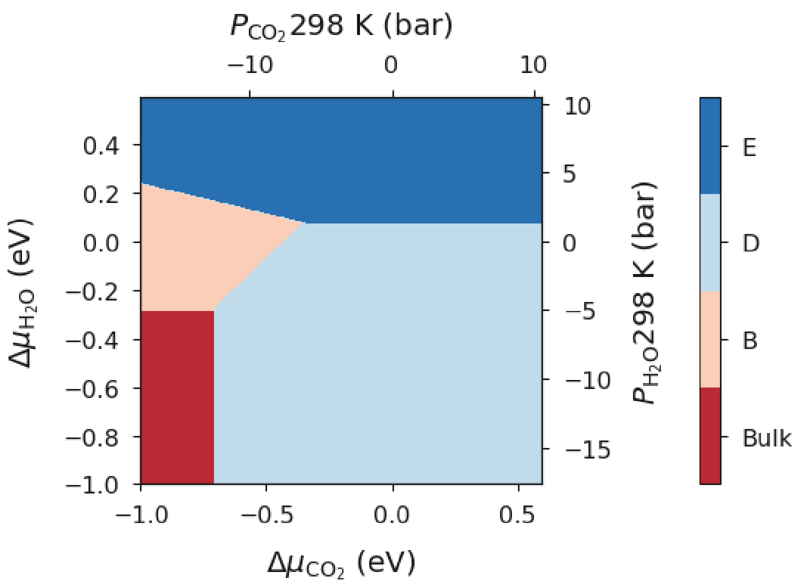
Vibrational modes can be calculated via a density functional pertibation calculation or via the phonopy code.

```
Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92,
                  label = "Bulk", entropy = True, zpe=True, file = 'ref_files/bulk_vib.
↪yaml',
                  funits = 10, temp_range=temperature_range)

A = data.DataSet(cation = 10, x = 5, y = 20, energy = -468,
                  label = "A", entropy = True, zpe=True, file = 'ref_files/A_vib.yaml',
                  funits = 5, temp_range=temperature_range)

B = data.DataSet(cation = 10, x = 0, y = 10, energy = -228,
                  label = "B", entropy = True, zpe=True, file = 'ref_files/B_vib.yaml',
                  funits =  10, temp_range=temperature_range)

C = data.DataSet(cation = 10, x = 10, y = 30, energy = -706,
                  label = "C", entropy = True, zpe=True, file = 'ref_files/C_vib.yaml',
                  funits = 10, temp_range=temperature_range)

D = data.DataSet(cation = 10, x = 10, y = 0, energy = -310,
                  label = "D", entropy = True, zpe=True,  file = 'ref_files/D_vib.yaml',
                  funits =  10, temp_range=temperature_range)

E = data.DataSet(cation = 10, x = 10, y = 50, energy = -972,
                  label = "E", entropy = True, zpe=True, file = 'ref_files/E_vib.yaml',
                  funits =  10, temp_range=temperature_range)

F = data.DataSet(cation = 10, x = 8, y = 10, energy = -398,
                  label = "F", entropy = True, zpe=True, file = 'ref_files/F_vib.yaml',
                  funits =  2, temp_range=temperature_range)


data = [Bulk, A, B, C, D, E, F]

x_energy=-20.53412969
y_energy=-12.83725889
```

(continues on next page)

(continued from previous page)

```
CO2_exp = ut.fit_nist("CO2.txt")[298]
Water_exp = ut.fit_nist("H2O.txt")[298]

CO2_corrected = x_energy + CO2_exp
Water_corrected = y_energy + Water_exp

deltaX = {'Range': [ -1, 0.6],  'Label': 'CO_2'}
deltaY = {'Range': [ -1, 0.6], 'Label': 'H_2O'}

temp_298 = bmvm.calculate(data, bulk, deltaX, deltaY, CO2_corrected, Water_corrected)
ax = temp_298.plot_mu_p(temperature=298, set_style="fast", colourmap="RdBu")
```



Temperature

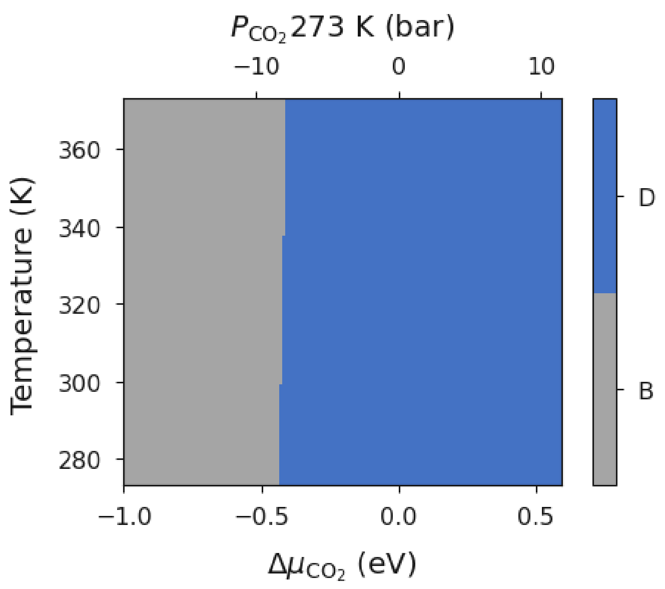
In tutorial 5 we showed how SurfinPy can be used to calculate the vibrational entropy and zero point energy for solid phases and in tutorial 4 we showed how a temperature range can be used to calculate the phase diagram of temperature as a function of presure. In this example we will use both lesson from these tutorials to produce a phase diagram of temperature as a function of pressure including the vibrational properties for solid phases. Again this produces results which are easily compared to experimental values in addition to increasing the level of theory used.

```
import matplotlib.pyplot as plt
from surfinpy import bulk_mu_vs_mu as bmvm
from surfinpy import utils as ut
from surfinpy import bulk_mu_vs_t as bmvt
from surfinpy import data
import numpy as np

colors = ['#5B9BD5', '#4472C4', '#A5A5A5', '#772C24', '#ED7D31', '#FFC000', '#70AD47']

temperature_range = [273, 373]

```

(continues on next page)

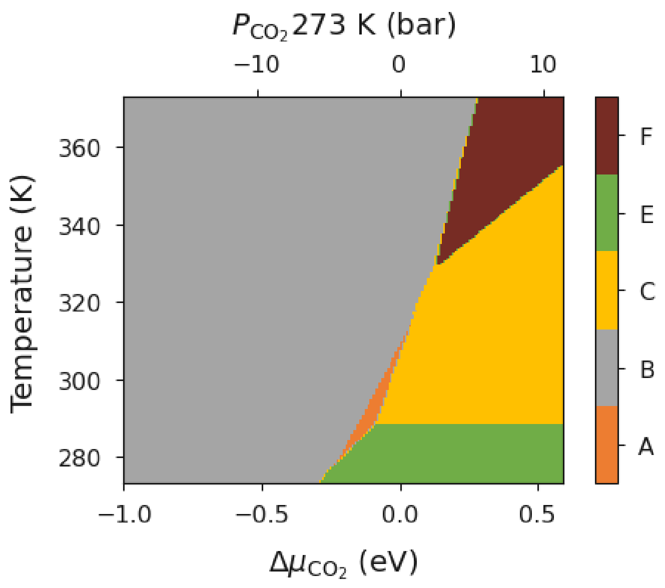
(continued from previous page)

```
bulk = data.ReferenceDataSet(cation = 1, anion = 1, energy = -92.0, funits = 10, file =
↪'bulk_vib.yaml', entropy=True, zpe=True, temp_range=temperature_range)


Bulk = data.DataSet(cation = 10, x = 0, y = 0, energy = -92., color=colors[0],
                label = "Bulk", entropy = True, zpe=True, file = 'ref_files/bulk_vib.yaml
↪',
                funits = 10, temp_range=temperature_range)

D = data.DataSet(cation = 10, x = 10, y = 0, energy = -310.,  color=colors[1],
                label = "D", entropy = True, zpe=True,  file = 'ref_files/D_vib.yaml',
                funits =  10, temp_range=temperature_range)

B = data.DataSet(cation = 10, x = 0, y = 10, energy = -227.,  color=colors[2],
                label = "B", entropy = True, zpe=True, file = 'ref_files/B_vib.yaml',
                funits =  10, temp_range=temperature_range)

F = data.DataSet(cation = 10, x = 8, y = 10, energy = -398.,  color=colors[3],
                label = "F", entropy = True, zpe=True, file = 'ref_files/F_vib.yaml',
                funits =  2, temp_range=temperature_range)

A = data.DataSet(cation = 10, x = 5, y = 20, energy = -467.,  color=colors[4],
                label = "A", entropy = True, zpe=True, file = 'ref_files/A_vib.yaml',
                funits = 5, temp_range=temperature_range)


C = data.DataSet(cation = 10, x = 10, y = 30, energy = -705.,  color=colors[5],
                label = "C", entropy = True, zpe=True, file = 'ref_files/C_vib.yaml',
                funits = 10, temp_range=temperature_range)

E = data.DataSet(cation = 10, x = 10, y = 50, energy = -971.,  color=colors[6],
                label = "E", entropy = True, zpe=True, file = 'ref_files/E_vib.yaml',
                funits =  10, temp_range=temperature_range)

data = [Bulk, A, B, C,  D, E, F]

deltaX = {'Range': [ -1, 0.6],  'Label': 'CO_2'}
deltaZ = {'Range': [ 273, 373], 'Label': 'Temperature'}
x_energy=-20.53412969
y_energy=-12.83725889
mu_y = 0


exp_x = ut.temperature_correction_range("CO2.txt", deltaZ)
exp_y = ut.temperature_correction_range("H2O.txt", deltaZ)

system = bmvt.calculate(data, bulk, deltaX, deltaZ, x_energy, y_energy, mu_y, exp_x, exp_
↪y)
ax = system.plot_mu_vs_t_vs_p(temperature=273)
```



When investigating the phase diagram for certain systems it could be beneficial to remove a kinetically inhibited but thermodynamically stable phase to investigate the metastable phase diagram. Within SurfinPy this can be acheived via recreating the data list without the phase in question then recalculating the phse diagram, as below.

```
data = [Bulk, A, B, C, E, F]

system = bmvt.calculate(data, bulk, deltaX, deltaZ, x_energy, y_energy, mu_y, exp_x, exp_
↪y)
ax = system.plot_mu_vs_t_vs_p(temperature=273)
```



7.4 Using surfinpy

The are number of ways to get and use surfinpy

- Fork the code: please feel free to fork the code on [GitHub](https://github.com/symmy596/SurfinPy) and add functionality that interests you. There are already plans for version 2, these will all be added to the issues section in due course.
- Run it locally: surfinpy is available through the pip package manager.
- Get in touch: Adam R.Symington (ars44@bath.ac.uk) is always keen to chat to potential users.

7.5 API

7.5.1 surfinpy.data

`class surfinpy.data.`**`DataSet`**(*cation*, *x*, *y*, *energy*, *label*, *color=None*, *funits=0*, *file=None*, *area=None*, *nspecies=None*, *entropy=False*, *temp_range=False*, *zpe=False*)

Bases: `object`

Object that contains information about a DFT calculation to be added to the phase diagram calculation. This object is used in both the surface and bulk phase diagram methods.

Parameters

- **`cation`** (`int`) – Number of cations in dataset
- **`x`** (`int`) – Number of species x in dataset
- **`y`** (`int`) – Number of species y in dataset
- **`energy`** (`float`) – DFT evaluated energy of reference dataset
- **`label`** (`str`) – Label of dataset to be used in phase diagram
- **`color`** (`string`) – Desired color of this phase in the phase diagram
- **`funits`** (`int`) – Number of formula units in dataset
- **`file`** (`str`) – yaml file containing vibrational frequencies
- **`area`** (`float`) – Surface area - required for surface calculations
- **`nspecies`** (`int`) – Number of species that are constituent parts of the surface.
- **`entropy`** (`bool`) – Is entropy to be considered?
- **`temp_range`** (`list`) – Temperature range to calculate vibrational entropy across
- **`zpe`** (`bool`) – Is the zero point energy to be considered?

`class surfinpy.data.`**`ReferenceDataSet`**(*cation*, *anion*, *energy*, *funits*, *color=None*, *file=None*, *entropy=False*, *temp_range=None*, *zpe=False*)

Bases: `object`

Object that contains information about the reference DFT calculation to be used in the phase diagram calculation. This object is used in both the surface and bulk phase diagram methods.

Parameters

- **`cation`** (`int`) – Number of cations in reference dataset
- **`anion`** (`int`) – Number of anions in reference dataset

- **energy** (float) – DFT evaluated energy of reference dataset
- **funits** (int) – Number of formula units in reference dataset
- **color** (string) – Desired color of this phase in the phase diagram
- **file** (str):) – yaml file containing vibrational frequencies
- **entropy** (bool) – Is entropy to be considered?
- **temp_range** (list) – Temperature range to calculate vibrational entropy across
- **zpe** (bool) – Is the zero point energy to be considered?

7.5.2 surfinpy.mu_vs_mu

Functions related to the generation of surface phase diagrams as a function of chemical potential. An explanation of theory can be found here

surfinpy.mu_vs_mu.**calculate**(*data*, *bulk*, *deltaX*, *deltaY*, *x_energy=0*, *y_energy=0*, *increments=0.025*)

Initialise the surface energy calculation.

Parameters

- **data** (list) – List of *surfinpy.data.DataSet* for each phase
- **bulk** (*surfinpy.data.ReferenceDataSet*) – Data for bulk
- **deltaX** (dict) – Range of chemical potential/label for species X
- **DeltaY** (dict) – Range of chemical potential/label for species Y
- **x_energy** (float) – DFT energy of adsorbing species
- **y_energy** (float) – DFT energy of adsorbing species

Returns **system** – Plotting object

Return type *surfinpy.plotting.ChemicalPotentialPlot*

surfinpy.mu_vs_mu.**calculate_excess**(*adsorbant*, *slab_cations*, *area*, *bulk*, *nspecies=1*, *check=False*)

Calculates the excess of a given species at the surface. Depending on the nature of the species, there are two ways to do this. If the species is a constituent part of the surface, e.g. Oxygen in TiO_2 then the calculation must account for the stoichiometry of that material. Using the TiO_2 example

$$\Gamma_O = \frac{1}{2A}\left(nO_{Slab} - \frac{nO_{Bulk}}{nTi_{Bulk}}nTi_{Slab}\right)$$

where nO_{Slab} is the number of oxygen in the slab, nO_{Bulk} is the number of oxygen in the bulk, A is the surface area, nTi_{Bulk} is the number of Ti in the bulk and nTi_{Slab} is the number of Ti in the slab. If the species is just an external adsorbant, e.g. water or carbon dioxide then one does not need to consider the state of the surface, as there was none there to begin with.

$$\Gamma_{H_2O} = \frac{nH_2O}{2A}$$

where nH_2O is the number of water molecules and A is the surface area.

Parameters

- **adsorbant** (int) – Number of species
- **slab_cations** (int) – Number of cations
- **area** (float) – Area of surface

- **bulk** (`dict`) – Dictonary of bulk properties
- **nspecies** (`int`) – number of external species
- **check** (`bool`) – Check if this is an external or constituent species.

Returns Surface excess of given species.

Return type `float`

`surfinpy.mu_vs_mu.`**`calculate_normalisation`**(*slab_energy*, *slab_cations*, *bulk*, *area*)

Normalises the slab energy relative to the bulk material. Thus allowing the different slab calculations to be compared.

$$Energy = \frac{1}{2A}\left(E_{MO}^{slab} - \frac{nCat_{slab}}{nCat_{Bulk}} E_{MO}^{Bulk}\right)$$

where Energy is the slab energy normalised to the bulk, E_{MO}^{slab} is the DFT slab energy, $nCat_{slab}$

is the number of slab cations, $nCat_{Bulk}$ is the number of bulk

cations, E_{MO}^{Bulk} is the DFT bulk energy A is the surface area.

Parameters

- **slab_energy** (`float`) – Energy of the slab from DFT
- **slab_cations** (`int`) – Total number of cations in the slab
- **bulk** (*`surfinpy.data.DataSet`*) – Bulk properties
- **area** (`float`) – Surface area

Returns Constant normalising the slab energy to the bulk energy.

Return type `float`

`surfinpy.mu_vs_mu.`**`calculate_surface_energy`**(*deltamux*, *deltamuy*, *x_energy*, *y_energy*, *xexcess*, *yexcess*, *normalised_bulk*)

Calculates the surface for a given chemical potential of species x and species y for a single phase.

$$\gamma_{Surf} = \frac{1}{2S}\left(E_{MO}^{slab} - \frac{nCat_{Slab}}{nCat_{Bulk}} E_{MO}^{Bulk}\right) - \Gamma_O \mu_O - \Gamma_{H_2O} \mu_{H_2O} - \Gamma_O \mu_O(T) - \Gamma_{H_2O} \mu_{H_2O}(T)$$

where S is the surface area, E_{MO}^{slab} is the DFT energy of the stoichiometric slab, $nCat_{Slab}$ is the number of cations in the slab, $nCat_{Slab}$ is the number of cations in the bulk unit cell, E_{MO}^{Bulk} is the DFT energy of the bulk unit cell, Γ_O Γ_{H_2O} is the excess oxygen / water at the surface and μ_O μ_{H_2O} is the oxygen / water chemcial potential.

Parameters

- **deltamux** (`array_like`) – Chemical potential of species x
- **deltamuy** (`array_like`) – Chemical potential of species y
- **x_energy** (`float`) – DFT energy or temperature corrected DFT energy
- **y_energy** (`float`) – DFT energy or temperature corrected DFT energy
- **xexcess** (`float`) – Surface excess of species x
- **yexcess** (`float`) – Surface excess of species y
- **normalised_bulk** (`float`) – Slab energy normalised to the bulk value.

Returns 2D array of surface energies as a function of chemical potential of x and y

Return type `array_like`

`surfinpy.mu_vs_mu.`**`evaluate_phases`**`(`*data*, *bulk*, *x*, *y*, *nsurfaces*, *x_energy*, *y_energy*`)`

Calculates the surface energies of each phase as a function of chemical potential of x and y. Then uses this data to evaluate which phase is most stable at that x/y chemical potential cross section.

Parameters

- **`data`** (`list`) – List containing the *`surfinpy.data.DataSet`* for each phase
- **`bulk`** (*`surfinpy.data.DataSet`*) – Data for bulk
- **`x`** (`dict`) – X axis chemical potential values
- **`y`** (`dict`) – Y axis chemical potential values
- **`nsurfaces`** (`int`) – Number of phases
- **`x_energy`** (`float`) – DFT 0K energy for species x
- **`y_energy`** (`float`) – DFT 0K energy for species y

Returns **phase_data** – array of ints, with each int corresponding to a phase.

Return type `array_like`

7.5.3 surfinpy.bulk_mu_vs_mu

`surfinpy.bulk_mu_vs_mu.`**`calculate`**`(`*data*, *bulk*, *deltaX*, *deltaY*, *x_energy*, *y_energy*`)`

Initialise the free energy calculation.

Parameters

- **`data`** (`list`) – List of *`surfinpy.data.DataSet`* object for each phase
- **`bulk`** (*`surfinpy.data.ReferenceDataSet`*) – Reference dataset
- **`deltaX`** (`dict`) – Range of chemical potential/label for species X
- **`DeltaY`** (`dict`) – Range of chemical potential/label for species Y
- **`x_energy`** (`float`) – DFT energy of adsorbing species
- **`y_energy`** (`float`) – DFT energy of adsorbing species

Returns **system** – Plotting object

Return type *`surfinpy.plotting.ChemicalPotentialPlot`*

`surfinpy.bulk_mu_vs_mu.`**`calculate_bulk_energy`**`(`*deltamux*, *deltamuy*, *x_energy*, *y_energy*, *phase*, *normalised_bulk*`)`

Calculates the free energy of a given phase (DFT calculation) as a function of chemical potential of x and y.

Parameters

- **`deltamux`** (`array_like`) – Chemical potential of species x
- **`deltamuy`** (`array_like`) – Chemical potential of species y
- **`x_energy`** (`float`) – DFT energy or temperature corrected DFT energy
- **`y_energy`** (`float`) – DFT energy or temperature corrected DFT energy
- **`phase`** (*`surfinpy.data.DataSet`*) – DFT calculation
- **`normalised_bulk`** (`float`) – Bulk energy normalised to the bulk value.

Returns 2D array of free energies as a function of chemical potential of x and y

Return type `array_like`

`surfinpy.bulk_mu_vs_mu.`**`evaluate_phases`**(*data*, *bulk*, *x*, *y*, *nphases*, *x_energy*, *y_energy*)

Calculates the free energies of each phase as a function of chemical potential of x and y. Then uses this data to evaluate which phase is most stable at that x/y chemical potential cross section.

Parameters

- **`data`** (`list`) – List of *`surfinpy.data.DataSet`* objects
- **`bulk`** (*`surfinpy.data.ReferenceDataSet`* object) – Reference dataset
- **`x`** (`dict`) – X axis chemical potential values
- **`y`** (`dict`) – Y axis chemical potential values
- **`nphases`** (`int`) – Number of phases
- **`x_energy`** (`float`) – DFT 0 K energy for species x
- **`y_energy`** (`float`) – DFT 0 K energy for species y

Returns **phase_data** – array of ints, with each int corresponding to a phase.

Return type `array_like`

`surfinpy.bulk_mu_vs_mu.`**`normalise_phase_energy`**(*phase*, *bulk*)

Converts normalises each phase to be consistent with the bulk. DFT calculations may have differing numbers of formula units compared to the bulk and this must be accounted for. Furthermore, the vibrational entropy and zero point energy are accounted for (if required).

Parameters

- **`phase`** (*`surfinpy.data.DataSet`*) – surfinpy dataset object.
- **`bulk`** (*`surfinpy.data.DataSet`*) – surfinpy ReferenceDataSet object.

Returns Normalised phase energy

Return type `float`

7.5.4 surfinpy.bulk_mu_vs_t

`surfinpy.bulk_mu_vs_t.`**`calculate`**(*data*, *bulk*, *deltaX*, *deltaY*, *x_energy*, *y_energy*, *mu_z*, *exp_x*, *exp_y*)

Initialise the free energy calculation.

Parameters

- **`data`** (`list`) – List containing the *`surfinpy.data.DataSet`* objects for each phase
- **`bulk`** (*`surfinpy.data.ReferenceDataSet`*) – Reference dataset
- **`x`** (`dict`) – X axis chemical potential values
- **`y`** (`dict`) – Y axis chemical potential values
- **`nphases`** (`int`) – Number of phases
- **`x_energy`** (`float`) – DFT 0K energy for species x
- **`y_energy`** (`float`) – DFT 0K energy for species y
- **`mu_z`** (`float`) – Set chemical potential for species y
- **`exp_x`** (`float`) – Experimental correction for species x

- **exp_y** (`float`) – Experimental correction for species y

Returns **system** – Plotting object

Return type *surfinpy.plotting.MuTPlot*

`surfinpy.bulk_mu_vs_t.`**`calculate_bulk_energy`**(*deltamux*, *ynew*, *x_energy*, *z_energy*, *deltamuy*, *phase*, *bulk*, *normalised_bulk*, *exp_xnew*, *exp_znew*, *new_bulk_svib*, *new_data_svib*)

Calculates the free energy of a given phase (DFT calculation) as a function of chemical potential of x and y.

Parameters

- **deltamux** (`array_like`) – Chemical potential of species x
- **ynew** (`array_like`) – description needed
- **x_energy** (`float`) – DFT energy or temperature corrected DFT energy
- **y_energy** (`float`) – DFT energy or temperature corrected DFT energy
- **deltamuy** (`array_like`) – Chemical potential of species y
- **phase** (*surfinpy.data.DataSet*) – DFT calculation
- **bulk** (*surfinpy.data.ReferenceDataSet*) – DFT calculation
- **normalised_bulk** (`float`) – Bulk energy normalised to the bulk value.
- **exp_xnew** (`array_like`) – Experimental correction for species x
- **exp_znew** (`array_like`) – Experimental correction for species y
- **new_bulk_svib** (`float`) – Vibrational entopy for the bulk reference cell calculated at the temperature range provided
- **new_data_svib** (`float`) – Vibrational entopy for the phase calculated at the temperature range provided

Returns Free energy

Return type `array_like`

`surfinpy.bulk_mu_vs_t.`**`evaluate_phases`**(*data*, *bulk*, *x*, *y*, *nphases*, *x_energy*, *y_energy*, *mu_z*, *exp_x*, *exp_z*)

Calculates the surface energies of each phase as a function of chemical potential of x and y. Then uses this data to evaluate which phase is most stable at that x/y chemical potential cross section.

Parameters

- **data** (`list`) – List containing the *surfinpy.data.DataSet* objects for each phase
- **bulk** (*surfinpy.data.ReferenceDataSet*) – Reference dataset
- **x** (`dict`) – X axis chemical potential values
- **y** (`dict`) – Y axis chemical potential values
- **nphases** (`int`) – Number of phases
- **x_energy** (`float`) – DFT 0K energy for species x
- **y_energy** (`float`) – DFT 0K energy for species y
- **mu_z** (`float`) – Set chemical potential for species y
- **exp_x** (`float`) – Experimental correction for species x
- **exp_z** (`float`) – Experimental correction for species y

Returns **phase_data** – array of ints, with each int corresponding to a phase.

Return type `array_like`

`surfinpy.bulk_mu_vs_t.`**`normalise_phase_energy`**`(`*phase*, *bulk*`)`

Converts normalises each phase to be consistent with the bulk. DFT calculations may have differing numbers of formula units compared to the bulk and this must be accounted for. Furthermore, the vibrational entropy and zero point energy are accounted for (if required).

Parameters

- **`phase`** (*`surfinpy.data.DataSet`*) – surfinpy dataset object.
- **`bulk`** (*`surfinpy.data.ReferenceDataSet`*) – surfinpy ReferenceDataSet object.

Returns Constant normalising the slab energy to the bulk energy.

Return type `float`

7.5.5 surfinpy.vibrational_data

`surfinpy.vibrational_data.`**`entropy_calc`**`(`*freq*, *temp*, *vib_prop*`)`

Calculates and returns the vibrational entropy for the system.

Parameters

- **`freq`** (`array_like`) – Vibrational frequencies for system.
- **`temp`** (`array_like`) – Temperature range at which the vibrational entropy is calculated
- **`vib_prop`** (`array_like`) – Vibrational Properties read from input yaml file

Returns **svib** – Vibrational entropy for the system calculated using the temperature range provided.

Return type `array_like`

`surfinpy.vibrational_data.`**`recalculate_vib`**`(`*dataset*, *bulk*`)`

`surfinpy.vibrational_data.`**`vib_calc`**`(`*vib_file*, *temp_r*`)`

Calculates and returns the Zero Point Energy (ZPE) and vibrational entropy for the temperature range provided.

Parameters

- **`vib_file`** (`str`):) – yaml file containing vibrational frequencies
- **`temp_r`** (`array_like`) – Temperature range at which the vibrational entropy is calculated

Returns

- **zpe** (`float`) – Zero Point energy for the system
- **svib** (`array_like`) – Vibrational entropy for the system calculated using the temperature range provided.

`surfinpy.vibrational_data.`**`zpe_calc`**`(`*vib_prop*`)`

Calculates and returns the zero point energy for the system.

Parameters **`vib_prop`** (`array_like`) – Vibrational Properties read from input yaml file

Returns **zpe** – Zero Point energy for the system

Return type `float`

7.5.6 surfinpy.p_vs_t

Functions related to the generation of surface phase diagrams as a function of pressure and temperature. An explanation of theory can be found here

`surfinpy.p_vs_t.`**`adsorption_energy`**(*data*, *stoich*, *adsorbant_t*)

From the dft data provided - calculate the adsorbation energy of a species at the surface.

Parameters

- **`data`** (`list`) – list of *`surfinpy.data.DataSet`* objects containing info about each calculation
- **`stoich`** (*`surfinpy.data.DataSet`*) – info about the stoichiometric surface calculation
- **`adsorbant_t`** (`array_like`) – dft energy of adsorbing species as a function of temperature

Returns **AE** – Adsorbtion energy of adsorbing species in each calculation as a function of temperature

Return type `array_like`

`surfinpy.p_vs_t.`**`calculate`**(*stoich*, *data*, *SE*, *adsorbant*, *thermochem*, *max_t=1000*, *min_p=- 13*, *max_p=5.5*, *coverage=None*, *transform=True*)

Collects input variables and intitialises the calculation.

Parameters

- **`stoich`** (*`surfinpy.data.DataSet`*) – information about the stoichiometric surface
- **`data`** (`list`) – list of *`surfinpy.data.DataSet`* objects on the "adsorbed" surfaces
- **`SE`** (`float`) – surface energy of the stoichiomteric surface
- **`adsorbant`** (`float`) – dft energy of adsorbing species
- **`coverage`** (`array_like` (default None)) – Numpy array containing the different coverages of adsorbant.
- **`thermochem`** (`array_like`) – Numpy array containing thermochemcial data downloaded from NIST_JANAF for the adsorbing species.
- **`max_t`** (`int`) – Maximum temperature in the phase diagram
- **`min_p`** (`int`) – Minimum pressure of phase diagram
- **`max_p`** (`int`) – Maximum pressure of phase diagram

Returns **system** – plotting object

Return type *`surfinpy.plotting.PTPlot`*

`surfinpy.p_vs_t.`**`calculate_adsorption_energy`**(*adsorbed_energy*, *slab_energy*, *n_species*, *adsorbant_t*)

Calculates the adsorption energy in units of eV

Parameters

- **`adsorbed_energy`** ((`float`):) – slab energy of slab and adsorbed species from DFT
- **`slab_energy`** ((`float`):) – bare slab energy from DFT
- **`n_species`** ((`int`):) – number of adsorbed species at the surface
- **`adsorbant_t`** ((`array_like`):) – dft energy of adsorbing species as a function of temperature

Returns adsorption energy as a function of temperature

Return type `array_like`

`surfinpy.p_vs_t.`**`calculate_surface_energy`**(*AE*, *lnP*, *T*, *coverage*, *SE*, *nsurfaces*)

Calculates the surface energy as a function of pressure and temperature for each surface system according to

$$\gamma_{adsorbed,T,p} = \gamma_{bare} + (C(E_{ads,T} - RTln(\frac{p}{p^o})$$

where $\gamma_{adsorbed,T,p}$ is the surface energy of the surface with adsorbed species at a given temperature and pressure, γ_{bare} is the suface energy of the bare surface, C is the coverage of adsorbed species, $E_{ads,T}$ is the adsorption energy, R is the gas constant, T is the temperature, and $\frac{p}{p^o}$ is the partial pressure.

Parameters

- **AE** (`list`) – list of adsorption energies
- **lnP** ((`array_like`) – full pressure range
- **T** (`array_like`) – full temperature range
- **coverage** (`array_like`) – surface coverage of adsorbing species in each calculation
- **SE** (`float`) – surface energy of stoichiomteric surface
- **data** (`list`) – list of dictionaries containing info on each surface
- **nsurfaces** (`int`) – total number of surface

Returns **SE_array** – array of integers corresponding to lowest surface energies

Return type `array_like`

`surfinpy.p_vs_t.`**`convert_adsorption_energy_units`**(*AE*)

Converts the adsorption energy into units of KJ/mol

Parameters **AE** (`array_like`) – array of adsorption energies

Returns array of adsorption energies in units of KJ/mol

Return type `array_like`

`surfinpy.p_vs_t.`**`inititalise`**(*thermochem*, *adsorbant*, *max_t*, *min_p*, *max_p*)

Builds the numpy arrays for each calculation.

Parameters

- **thermochem** (`array_like`) – array containing NIST_JANAF thermochemical data
- **adsorbant** (`float`) – dft energy of adsorbing species
- **max_t** (`int`) – Maximum temperature of phase diagram
- **min_p** (`int`) – Minimum pressure of phase diagram
- **max_p** (`int`) – Maximum pressure of phase diagram

Returns

- **lnP** (`array_like`) – numpy array of pressure values
- **logP** (`array_like`) – log of lnP (hard coded range -13 - 5.0)
- **T** (`array_like`) – array of temperature values (hard coded range 2 - 1000 K)
- **adsrobant_t** (`array_like`) – dft values of adsorbant scaled to temperature

7.5.7 surfinpy.plotting

class `surfinpy.plotting.`**`ChemicalPotentialPlot`**(*x, y, z, labels, ticks, colors, xlabel, ylabel*)

Bases: `object`

Class that plots a phase diagram as a function of chemical potential.

Parameters

- **x** (`array_like`) – x axis, chemical potential of species x
- **y** (`array_like`) – y axis, chemical potential of species y
- **z** (`array_like`) – two dimensional grid, phase info
- **labels** (`list`) – `list`): of phase labels
- **ticks** (`list`) – `list`): of phases
- **colors** (`list`) – `list`): of phases
- **xlabel** (`str`) – species name for x axis label
- **ylabel** (`str`) – species name for y axis label

`plot_mu_p`(*temperature, colourmap=None, set_style=None, cbar_title=None, figsize=(6, 6)*)

Plots a phase diagram with two sets of axis, one as a function of chemical potential and the second is as a function of pressure.

Parameters

- **temperature** (`int`) – temperature
- **colourmap** (`str`) – colourmap for the plot
- **set_style** (`str`) – Plot style
- **cbar_label** (`str`) – Label for colorbar

`plot_phase`(*temperature=None, colourmap=None, set_style=None, figsize=None, cbar_title=None*)

Plots a simple phase diagram as a function of chemical potential.

Parameters

- **temperature** (`int`) – Temperature.
- **colourmap** (`str`) – Colourmap for the plot.
- **set_style** (`str`) – Plot style
- **figsize** (`tuple`) – Set a custom figure size.

`plot_pressure`(*temperature, colourmap=None, set_style=None, figsize=(6, 6), cbar_title=None*)

Plots a phase diagram as a function of pressure.

Parameters

- **temperature** (`int`) – temperature
- **colourmap** (`str`) – colourmap for the plot
- **set_style** (`str`) – Plot style

class `surfinpy.plotting.`**`MuTPlot`**(*x, y, z, labels, ticks, colors, xlabel, ylabel*)

Bases: `object`

Class that plots a phase diagram as a function of chemical potential and temperature.

Parameters

- **x** (`array_like`) – x axis, chemical potential of species x
- **y** (`array_like`) – y axis, chemical potential of species y
- **z** (`array_like`) – two dimensional grid, phase info
- **labels** (`list`) – `list`): of phase labels
- **ticks** (`list`) – `list`): of phases
- **colors** (`list`) – `list`): of phases
- **xlabel** (`str`) – species name for x axis label
- **ylabel** (`str`) – species name for y axis label

plot_mu_vs_t(*colourmap=None*, *set_style=None*, *figsize=(6, 6)*, *cbar_title=None*)
Plots a simple phase diagram as a function of chemical potential.

Parameters

- **colourmap** (`str`) – Colourmap for the plot. Default='viridis'
- **set_style** (`str`) – Plot style
- **figsize** (`tuple`) – Set a custom figure size.

plot_mu_vs_t_vs_p(*temperature*, *colourmap=None*, *set_style=None*, *figsize=(6, 6)*, *cbar_title=None*)
Plots a simple phase diagram as a function of chemical potential.

Parameters

- **temperature** (`int`) – Temperature.
- **colourmap** (`str`) – Colourmap for the plot. Default='viridis'
- **set_style** (`str`) – Plot style
- **figsize** (`tuple`) – Set a custom figure size.

plot_p_vs_t(*temperature*, *colourmap=None*, *set_style=None*, *figsize=(6, 6)*, *cbar_title=None*)
Plots a simple phase diagram as a function of chemical potential.

Parameters

- **temperature** (`int`) – Temperature.
- **colourmap** (`str`) – Colourmap for the plot. Default='viridis'
- **set_style** (`str`) – Plot style
- **figsize** (`tuple`) – Set a custom figure size.

class surfinpy.plotting.**PTPlot**(*x*, *y*, *z*)
Bases: `object`

Class for plotting of temperature vs pressure phase diagrams.

Parameters

- **x** (`array_like`) – x axis
- **y** (`array_like`) – y axis
- **z** (`array_like`) – two dimensional array of phases

plot(*colourmap='viridis'*, *set_style=None*, *figsize=(6, 6)*, *ylabel='log P (bar)'*, *xlabel='Temperature (K)'*)
plots phase diagram

Parameters

- **colourmap** (str) – colourmap for phase diagram
- **set_style** (str) – Plot style

7.5.8 surfinpy.wulff

The module required for the generation of wulff plots. An explanation of theory can be found *here <theory.html>*

surfinpy.wulff.**calculate_surface_energy**(*stoich*, *data*, *SE*, *adsorbant*, *thermochem*, *T*, *P*, *coverage=None*)
Calculate the surface energy at a specific temperature and pressure.

Parameters

- **stoich** (*surfinpy.data.ReferenceDataSet*) – information about the stoichiometric surface
- **data** (list) – list of dictionaries containing information on the "adsorbed" surfaces
- **SE** (float) – surface energy of the stoichiomteric surface
- **adsorbant** (float) – dft energy of adsorbing species
- **coverage** (array_like) – Numpy array containing the different coverages of adsorbant.
- **thermochem** (array_like) – Numpy array containing thermochemcial data downloaded from NIST_JANAF for the adsorbing species.
- **T** (float) – Temperature to calculate surface energy
- **P** (float) – Pressure to calculate the surface energy
- **coverage** – Coverage of adsorbed specied on the surface.

Returns **SEs** – surface energies for each surface at T/P

Return type array_like

surfinpy.wulff.**temperature_correction**(*T*, *thermochem*, *adsorbant*)
Make the energy of the adsorbing species a temperature dependent term by scaling it with experimental data.

Parameters

- **T** (int) – Temperature to scale the energy to
- **thermochem** (array_like) – nist_janaf table
- **adsorbant** (float) – DFT energy of adsorbant

Returns **adsorbant** – Scaled energy of adsorbant

Return type float

7.5.9 surfinpy.utils

The utils module contains functions that are common and find various uses throughout the code.

surfinpy.utils.**build_entgrid**(*z*, *y*, *ynew*)
Builds a 2D grip of values for the x axis.

Parameters

- **x** (array_like) – One dimensional numpy array representing one dimension of phase diagram

- **y** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type `array_like`

`surfinpy.utils.`**`build_freqgrid`**(*z*, *y*)
Builds a 2D grip of values for the x axis.

Parameters

- **x** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram
- **y** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type `array_like`

`surfinpy.utils.`**`build_tempgrid`**(*z*, *y*)
Builds a 2D grip of values for the x axis.

Parameters

- **x** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram
- **y** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type `array_like`

`surfinpy.utils.`**`build_xgrid`**(*x*, *y*)
Builds a 2D grip of values for the x axis.

Parameters

- **x** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram
- **y** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type `array_like`

`surfinpy.utils.`**`build_ygrid`**(*x*, *y*)
Builds a 2D grip of values for the y axis.

Parameters

- **x** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram
- **y** (`array_like`) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type `array_like`

surfinpy.utils.**build_zgrid**(*z*, *y*)

Builds a 2D grip of values for the x axis.

Parameters

- **x** (array_like) – One dimensional numpy array representing one dimension of phase diagram
- **y** (array_like) – One dimensional numpy array representing one dimension of phase diagram

Returns **xnew** – Two dimensional numpy array required for energy calculations

Return type array_like

surfinpy.utils.**calculate_coverage**(*data*)

Calcualte the coverage of the adsorbing species on each surface.

Parameters **data** (list) – list of dictionaries containing info on each surface calculation

Returns **coverage** – Coverage values in units of n/nm^2

Return type array_like

surfinpy.utils.**calculate_gibbs**(*t*, *s*, *h*)

Calculate the gibbs free energy from thermochemcial data obtained from the NIST_JANAF database

Parameters

- **t** (array_like) – Temperature range
- **s** (array_like) – delta s values from nist
- **h** (array_like) – selta h values from nist

Returns **g** – gibbs energy as a function of temperature

Return type array_like

surfinpy.utils.**cs_fit**(*x*, *y*, *t*)

Fit a polynominal function to thermochemical data from NIST_JANAF

Parameters

- **x** (array_like) – x axis for fit
- **y** (array_like) – y axis for fit
- **t** (array_like) – x axis to be fitted

Returns **shift** – data fitted from x and y to t

Return type array_like

surfinpy.utils.**fit_nist**(*nist_file*, *increments=1*, *method='cs'*)

Use experimental data to correct the DFT free energy of an adsorbing species to a specific temperature.

Parameters **nist_file** (array_like) – numpy array containing experiemntal data from NIST_JANAF

Returns **gibbs** – correct free energy

Return type float

surfinpy.utils.**get_labels**(*ticks*, *data*)

Reads the phase diagram data and returns the labels that correspond to the phases displayed on the phase diagram.

Parameters

- **ticks** (list) – Phases that are displayed.
- **data** (list) – list of (*surfinpy.data.DataSet*): objects.

Returns **labels** – list of labels.

Return type `list`

`surfinpy.utils.`**`get_levels`**`(`*X*`)`

Builds the levels used in the contourf plot. This is neccesary to ensure that each color correpsonds to a single phase.

Parameters **X** (array_like) – 2D array of ints corresponding to each phase.

Returns **levels** – numpy array of ints

Return type `array_like`

`surfinpy.utils.`**`get_phase_data`**`(`*S*, *nsurfaces*`)`

Determines which surface composition is most stable at a given x and y value.

Parameters

- **S** (array_like) – 2D array of surface energies
- **nsurfaces** (int) – Total number of surfaces

Returns **x** – array of ints corresponding to the position of the lowest phase

Return type `array_like`

`surfinpy.utils.`**`get_ticks`**`(`*X*`)`

Sets the tick marks to show all phases plotted on the cbar plot.

`surfinpy.utils.`**`list_colors`**`(`*phases*, *ticks*`)`

Reads the phase diagram data and returns the colors that correspond to the phases displayed on the phase diagram.

Parameters

- **phases** (list) – list of (*surfinpy.data.DataSet*): objects.
- **ticks** (list) – Phases that are displayed.

Returns **colors** – list of colors.

Return type `list`

`surfinpy.utils.`**`poly_fit`**`(`*x*, *y*, *t*`)`

Fit a polynominal function to thermochemical data from NIST_JANAF :param x: x axis for fit :type x: array like :param y: y axis for fit :type y: array like :param t: x axis to be fitted :type t: array like

Returns **shift** – data fitted from x and y to t

Return type array like

`surfinpy.utils.`**`pressure`**`(`*chemical_potential*, *t*`)`

Converts chemical potential at a specific temperature (T) to a pressure value.

$$P = \frac{\mu}{k * T}$$

where P is the pressure, μ is the chemcial potential, k is the Boltzmann constant and T is the temperature.

Parameters

- **chemical_potential** (array_like) – delta mu values
- **t** (int) – temperature

Returns **pressure** – pressure values as a function of chemcial potential

Return type `array_like`

`surfinpy.utils.`**`read_nist`**`(`*File*`)`

Read a downloaded NIST_JANAF thermochemcial table

Parameters **`File`** (`str`) – Filename of NIST_JANAF thermochemcial table

Returns **data** – NIST_JANAF thermochemcial as an array

Return type `array_like`

`surfinpy.utils.`**`read_vibdata`**`(`*vib_file*`)`

Reads a yaml file containing the vribational frequencies from a DFT calculation.

Parameters **`vib_file`** ((`str`):) – File name

Returns **vib_prop** – Dictionary of vibrational freqencies.

Return type `dict`

`surfinpy.utils.`**`temperature_correction_range`**`(`*nist_file*, *deltaY*`)`

Use experimental data to correct the DFT free energy of an adsorbing species to a specific temperature.

Parameters

- **`nist_file`** (`array_like`) – numpy array containing experiemntal data from NIST_JANAF
- **`temperature`** (`int`) – Temperature to correct to

Returns **gibbs** – correct free energy

Return type `float`

`surfinpy.utils.`**`transform_numbers`**`(`*Z*, *ticks*`)`

transform numbers - Takes the phase diagram array and converts the numbers to numbers scaled 0, 1, 2, etc in order to make plotting easier

Parameters

- **`Z`** (`array_like`) – array of integers
- **`ticks`** (`list`) – unique phases

Returns **Z** – Normalised to a continuous set of numbers.

Return type `array_like`

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX