Spinney

Release '0.9.a2'

'Marco Arrigoni'

Oct 16, 2020

CONTENTS:

CHAPTER

ONE

THE `SPINNEY` PACKAGE

`Spinney` is a Python package dedicated to the study of point defects in solids. In particular, the code is designed to assist with the post-processing of first-principles calculations.

The aim of the package is to perform the several steps required in order to obtain meaningful quantities and properties concerning materials with point defects from the output files produced by first-principles codes.

These are some of the most relevant tasks that **`Spinney`** is able to perform:

- Compute, from external data (taken for example from online repositories), the ranges of validity for the elements chemical potentials according to thermodynamic constraints;
- Calculate the correction energy due to electrostatic finite-size-effects in charged supercells using state-of-the-art schemes;
- Calculate defect formation energies and transition levels;
- Calculate defects and carriers concentrations.

In developing **`Spinney`**, we aimed for a great code flexibility. For this reason, all the core routines take as argument either built-in Python objects or data structures ubiquitous in scientific computing, such as **NumPy** arrays.

This makes possible the integration of **`Spinney`** with any first-principle code, as the core routines can be used to write a proper interface for the relevant computer package.

For convenience, **`Spinney`** offers also a higher-level object for a direct calculation of the defect formation energy from the result of first-principles calculations. Such property is arguably the most relevant descriptor characterizing a point defect. For technologically-relevant materials, such as insulators and semiconductors, the presence of charged defects is ubiquitous. The *`PointDefect`* class offers a handy way for calculating the defect formation energy, including correction terms for dealing with electrostatic finite-size effects using state-of-the-art schemes. For convenience, such class can be initialized by a `ase.Atoms` instance of the **ASE** library.

Reading and processing the output of first principles calculations describing several different point defects in the same host material is straightforward through the *`DefectiveSystem`* class.

CHAPTER TWO

INSTALLATION

2.1 Setup

Spinney is available one the Python Package Index, the most simple way to install it is using *pip*:

```
pip install spinney
```

After the installation is completed, you can run some tests in order to see if the installation was successfull.

To do so, run the command:

```
spinney-run-tests
```

The **Spinney** package is also hosted on Gitlab.

In this case, to install the package, create a new directory and clone the repository. From the root directory, run the command:

```
python3 setup.py install
```

And to check if the installation was successfull, type:

```
python3 setup.py test
```

Note: We recommend to install **Spinney** within a Python virtual environment, such as those created by Virtualenv.

2.2 Requirements

Spinney requires Python3 and the following libraries:

- NumPy version 1.12 or newer;
- SciPy version 1.4 or newer;
- Pandas version 0.25 or newer;
- Matplotlib version 3.1.0 or newer;
- ASE version 3.18 or newer.

Warning: **`Spinney`** is **not** compatible with `ASE` versions older than the *3.18* one.

CHAPTER

THREE

CASE STUDY: MG-DOPED GAN

This tutorial will show how to calculate the main quantities of interest for a defective system using **Spinney**. It is meant to offer a quick overview of the capabilities of **Spinney** and how to use the code in practice. For a more detailed guide, see the full *Tutorial*.

We will take as an example Mg-doped GaN. GaN is a material which is extensevely employed in several devices. It shows an intrinsic n-type conductivity; on the other hand, the ability of synthesizing p-type GaN would be very beneficial for many technological applications such as optoelectronic devices. Obtaining p-type carriers is very challenging and Mg is arguably the only dopant that has been successfully employed in the synthesis of p-type GaN.

Contents

3.1 Step 1. Defining the values of chemical potentials

Important quantities that characterize a defective system, such as *defect formation energies* are affected by the chemical potential values of the elements forming the defective system.

The thermodynamic stability of the host material imposes some constraints on the values these chemical potentials can assume.(For more details see the *dedicated session in the main tutorial*).

For the case of Mg-doped GaN, let $\Delta\mu_i$ represents the value of the chemical potential of element from the element chemical potential in its standard state (crystalline orthorhombic phase for Ga, N_2 molecule for N and the crystalline

HCP phase for Mg). The thermodynamic stability of GaN requires that the following constraints are satisfied:

$$\begin{aligned}
\Delta\mu_{\mathrm{Ga}} + \Delta\mu_{\mathrm{N}} &= \Delta h_{\mathrm{GaN}} \\
x\Delta\mu_{\mathrm{Ga}} + y\Delta\mu_{\mathrm{N}} + z\Delta\mu_{\mathrm{Mg}} &\leq \Delta h_{\mathrm{Ga_xN_yMg_z}} \\
\Delta\mu_{\mathrm{Ga}} &\leq 0 \\
\Delta\mu_{\mathrm{N}} &\leq 0 \\
\Delta\mu_{\mathrm{Mg}} &\leq 0
\end{aligned}$$

where Δh is the formation enthalpy per formula unit.

For calculating physically acceptable values for the chemical potentials, we can use the *Range* class.

It is useful to prepare a text file, for example called *formation_energies.txt* which for each compound in the Ga-N-Mg system reports in the first column the compound formula unit and in the second column its formation energy per formula unit. In our example we can use:

```
# Formation energies per formula unit calculated with PBE for GaN
#Compound                 E_f (eV/fu)
MgGa2                     -0.3533064287
Mg3N2                     -3.8619941875
Mg2Ga5                    -0.7160803263
Ga                         0.0000000000
Mg                         0.0000000000
GaN                       -0.9283419463
N                          0.0000000000
```

Line starting with # are comments and will be skipped, more compounds can of course be consider, but we limit ourselves to few of them for simplicity.

The following snippet of code shows the minimum and maximum value that $\Delta\mu_i$ can assume in order to satisfy the constraints (3.1).

```
from spinney.thermodynamics.chempots import Range, get_chem_pots_conditions

data = 'formation_energies.txt'

equality_compounds = ['GaN'] # compound where the constraint is an equation
order = ['Ga', 'N', 'Mg'] # chemical potential 0 is for Ga, 1 for N etc
# obtain data to feed to the Range class
parsed_data= get_chem_pots_conditions(data, order, equality_compounds)
# prepare Range instances
crange = Range(*parsed_data[:-1])

# for each chemical potential, stores the minimum and maximum possible values
ranges = crange.variables_extrema
print(ranges)
```

Output:

```
[[-9.28341946e-01 -2.21944834e-14]
 [-9.28341946e-01 -3.94026337e-14]
 [           -inf -6.68436765e-01]]
```

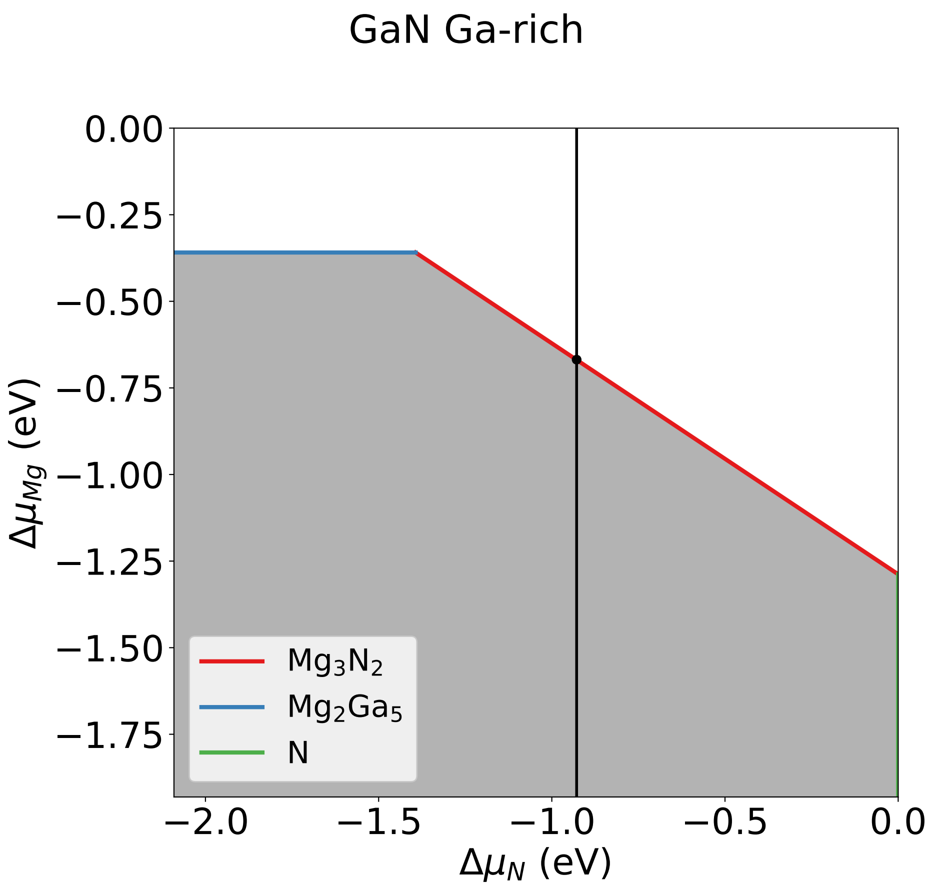
In principle both $\Delta\mu_{\mathrm{Ga}}$ and $\Delta\mu_{\mathrm{N}}$ can range from h_{GaN} to 0. Clearly the value of one chemical potential fixes the other as given by equation (3.1).

Suppose we are interested in GaN growth in Ga-rich conditions. We want then set $\Delta\mu_{\mathrm{Ga}} = 0$. We need to find acceptable values for $\Delta\mu_{\mathrm{N}}$ and $\Delta\mu_{\mathrm{Mg}}$. The former is fixed once $\Delta\mu_{\mathrm{Ga}}$ is fixed. For the latter, it is usually observed that high concentrations of Mg are needed to fabricate p-type GaN. We would then like to consider conditions in which the chemical potential of Mg is as close as possible to the one of pure metal Mg.

We can investigate the acceptable values of $\Delta\mu_{\mathrm{Mg}}$ by plotting the intersection of the feasible region with the plane $\Delta\mu_{\mathrm{Ga}} = 0$, and print the mimimum and maximum value the chemical potential can assume on this plane, as shows this code snippet:

```
# compound labels
crange.set_compound_dict(parsed_data[-1])
# let's use some pretty labels in the plot
# the order of the axes  must follow the order used for get_chem_pots_conditions
labels = [r'$\Delta \mu_{Ga}$ (eV)', r'$\Delta \mu_{N}$ (eV)',
          r'$\Delta \mu_{Mg}$ (eV)']
# intersection plane is defined by axes 1 and 2 (chem pot of N and Mg)
crange.plot_feasible_region_on_plane([1,2], x_label=labels[1],
                                     y_label=labels[2],
                                     title='GaN Ga-rich',
                                     save_plot=True)
# chemical potential boundaries on such plane
print(crange.variables_extrema_2d)
```

Output:



```
[[-0.92834195 -0.92834195]
 [       -inf -0.66843676]]
```

The shaded are in the plot show the feasible region corresponding to the inequality constraints of equation (3.1). From the plot it is clear that $\Delta\mu_{\mathrm{Mg}}$ can assume a maximum value of -0.668 eV. It is not possible to go to Mg-richer conditions as $\mathrm{Mg_3N_2}$ would start to precipitate.

In the study of defective Mg-doped GaN we will then consider the chemical potentials:

- $\Delta\mu_{\mathrm{Ga}} = 0$
- $\Delta\mu_{\mathrm{N}} = \Delta h_{\mathrm{GaN}}$
- $\Delta\mu_{\mathrm{Mg}} = -0.668$

Corresponding to a system in equilibrium with pure Ga and $\mathrm{Mg_3N_2}$.

3.2 Step 2. Set up the directory with the data about the defective system

Once we have an idea about the thermodynamic stability of the system, we can run the calculations of the defective system.

Most of the properties of interest of a defective system depend on the energy of the system with a point defect and pristine system. It is convenient to save the results of our calculations using the directory hierarchy understood by **Spinney**.

We will consider that all the calculations have been performed with the code VASP (for details when other codes are used see *Manage a defective system with the class DefectiveSystem*). Calculations have been performed at the PBE level using supercells with 96 atoms.

The directory tree might look like this:

```
data
├── data_defects
│   ├── Ga_int
│   │   ├── 0
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 1
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 2
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   └── 3
│   │       ├── OUTCAR
│   │       └── position.txt
│   ├── Ga_N
│   │   ├── 0
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 1
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── -1
│   │   │   ├── OUTCAR
```

(continues on next page)

(continued from previous page)

```
│    │    │    └── position.txt
│    │    ├── 2
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    └── 3
│    │         ├── OUTCAR
│    │         └── position.txt
...
├── Ga
│    └── OUTCAR
├── N2
│    └── OUTCAR
├── Mg
│    └── OUTCAR
└── pristine
     └── OUTCAR
```

For the study of Mg-doped GaN, we are considering the intrinsic defects as well, as these will always be present and will affect the properties of the doped system.

3.2.1 Initialize an `DefectiveSystem` instance

The easiest way to process the calculations of point defects is through the class *`DefectiveSystem`*.

```
from spinney.structures.defectivesystem import DefectiveSystem
# initialize the defective system, where calculations have been done with VASP
defective_system = DefectiveSystem('data', 'vasp')
```

Once the object has been initialized, we can use it to compute properties of interest.

3.3 Step 3. Calculate defect formation energies

The main quantity that characterizes a point defect it its formation energy. The periodic boundary conditions introduced by supercell calculations lead to some artifacts that should be corrected. In this example we will employ the correction scheme of Kumagai and Oba [KO14]. See the *relevant section* for more details.

In order to calculate the defect formation energy, we must feed to our *`DefectiveSystem`* instance some information about our system.

3.3.1 1. Calculate the value for the chemical potentials

To calculate the defect formation energy **`Spinney`** need the absolute values of the chemical potentials: $\mu_i = \mu_i^\circ + \Delta\mu_i$. In *Step 1. Defining the values of chemical potentials* we found $\Delta\mu_i$. We then need to calculate μ_i°. This can be easily done with the help of the `ase` library.

```
# values chemical potentials from standard state
dmu_ga = 0
dmu_mg = -0.668

# prepare chemical potentials with proper values
# paths with calculations results
path_defects = os.path.join('data', 'data_defects')
```

(continues on next page)

(continued from previous page)

```
path_pristine = os.path.join('data', 'pristine', 'OUTCAR')
path_ga = os.path.join('data', 'Ga', 'OUTCAR')
path_mg = os.path.join('data', 'Mg', 'OUTCAR')

# calculate chemical potentials
ase_pristine = ase.io.read(path_pristine, format='vasp-out')
mu_prist = 2*ase_pristine.get_total_energy()/ase_pristine.get_number_of_atoms()
ase_ga = ase.io.read(path_ga, format='vasp-out')
mu_ga = ase_ga.get_total_energy()/ase_ga.get_number_of_atoms() # Ga-rich
ase_mg = ase.io.read(path_mg, format='vasp-out')
mu_mg = ase_mg.get_total_energy()/ase_mg.get_number_of_atoms() # Mg-rich
mu_n = mu_prist - mu_ga # N-poor
mu_ga += dmu_ga
mu_mg += dmu_mg

# feed the data to the instance
defective_system.chemical_potentials = {'Ga':mu_ga, 'N':mu_n, 'Mg':mu_mg}
```

3.3.2 2. Add information about the pristine and defective system

We need to feed the data relative to the pristine and defective systems that need to be used to calculate the defect formation energies.

What is needed is:

- Valence band maximum eigenvalue
- Dielectric tensor
- Correction scheme method for electrostatic finite-size effects

```
# eigenvalue of the valence band maximum
vbm = 5.009256
# calculated dielectric tensor
e_rx = 5.888338 + 4.544304
e_rz = 6.074446 + 5.501630
e_r = [[e_rx, 0, 0], [0, e_rx, 0], [0, 0, e_rz]]

# feed the data
defective_system.vbm = vbm
defective_system.dielectric_tensor = e_r
defective_system.correction_scheme = 'ko' # Kumagai and Oba
```

3.3.3 3. Calculate the defect formation energy

All the required information has been fed to the `DefectiveSystem` instance. We can now calculate the defect formation energies.

```
defective_system.calculate_energies(False) # don't print to terminal
df = defective_system.data # data frame with calculated formation energies
print(df)
```

Output:

```
                Form Ene (eV)  Form Ene Corr. (eV)
Defect Charge
N_Ga    2          7.022139            7.272291
        3          7.435422            8.143598
       -1          9.977779           10.237273
        0          8.271712            8.271712
        1          7.461967            7.464786
Ga_int  2          3.190741            3.956642
        3          1.487580            3.063845
        0          8.339166            8.339166
        1          5.710694            5.964960
...
```

These are the defect formation energies calculated at the top of the valence band maximum, with or without electrostatic corrections for finite-size effects. We can also write them to a text file, in this case only the corrected values are written:

```
defective_system.write_formation_energies('formation_energies_Mg_GaN.txt')
```

Produces *formation_energies_Mg_GaN.txt*:

```
#System   Charge                Form Ene Corr. (eV)
N_Ga           2                     7.2722908069
N_Ga           3                     8.1435978547
N_Ga          -1                    10.2372728320
N_Ga           0                     8.2717122490
N_Ga           1                     7.4647858911
Ga_int         2                     3.9566415146
Ga_int         3                     3.0638453220
Ga_int         0                     8.3391662537
Ga_int         1                     5.9649599825
...
```

3.4 Step 4. Calculate charge transition levels

Once the defect formation energies have been calculated, we can calculate the *thermodynamic charge transition levels*. **Spinney** calculates them through the class *Diagram*. An instance of which is present in the DefectiveSystem class.

To use it we need to feed it the band edges. We will report them setting the 0 to the valence band maximum.

```
defective_system.gap_range = (0, 1.713)
# calculate charge transition levels
defective_system.diagram.transition_levels
```

Output:

```
#Defect type  q/q'
Ga_N          2/3      0.441121
              1/2      0.731836
              0/1      1.250109
Ga_int        2/3      0.892796
Mg_Ga         -1/0     0.179196
N_Ga          1/2      0.192495
              0/1      0.806926
```

(continues on next page)

(continued from previous page)

```
N_int          0/1      1.286762
Vac_Ga         -1/0     0.482381
               -2/-1    1.139556
```

We can also plot a diagram showing the transition levels within the band gap:

```
defective_system.diagram.plot(save_flag=True,
                              title='Mg-doped GaN Ga-rich limit',
                              legend=True,
                              x_label=r'$E_F$ (eV)',
                              save_title='diagram_defsys')
```

Saves the file *diagram_defsys.pdf*:

3.5 Step 4. Calculate defect concentrations

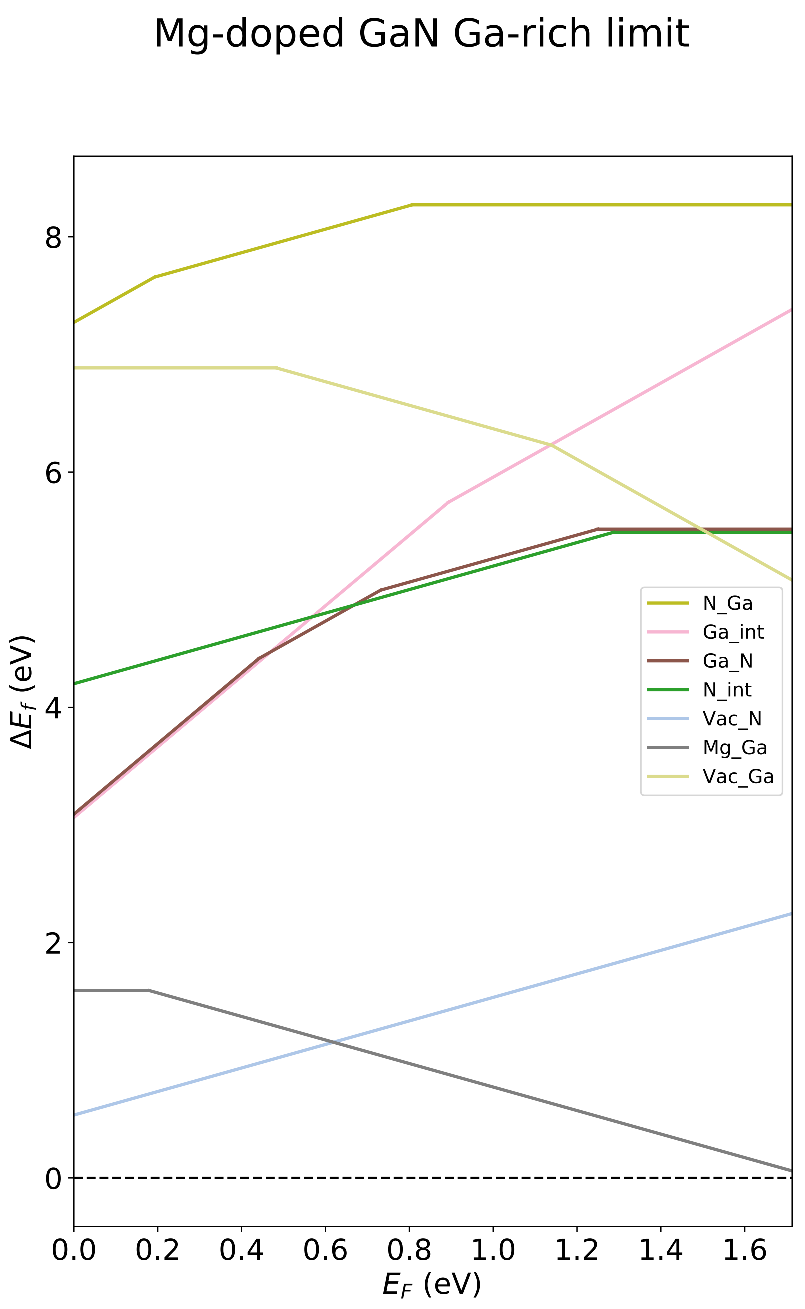
Once defect formation energies have been calculated, it is also possible to calculate equilibrium defect concentrations in *the dilute limit*. **Spinney** calculated defect concentrations through the `EquilibriumConcentrations`. An instance thereof is available in the `DefectiveSystem` instance. To calculate the concentrations, we need to feed some more data to the `DefectiveSystem` instance. For more details about these quantities see *Equilibrium defect concentrations in the dilute limit*.

```
from spinney.io.vasp import extract_dos
import numpy as np

# get the density of states of the pristine system
dos = extract_dos('vasprun.xml')[0]

# site concentrations for point defects
volume = ase_pristine.get_volume()/ase_pristine.get_number_of_atoms()
volume *= 4
factor = 1e-8**3 * volume
factor = 1/factor
site_conc = {'Ga_N':4, 'N_Ga':4, 'Vac_N':4, 'Vac_Ga':4,
             'Ga_int':6, 'N_int':6, 'Mg_Ga':4,
             'electron':36 , 'hole':36}
site_conc = {key:value*factor for key, value in site_conc.items()}

defective_system.dos = dos
defective_system.site_concentrations = site_conc
# calculate defect concentrations on a range of temperature
defective_system.temperature_range = np.linspace(250, 1000, 100)

concentrations = defective_system.concentrations
```

We can now access equilibrium properties of interest, such as equilibrium carrier concentrations:

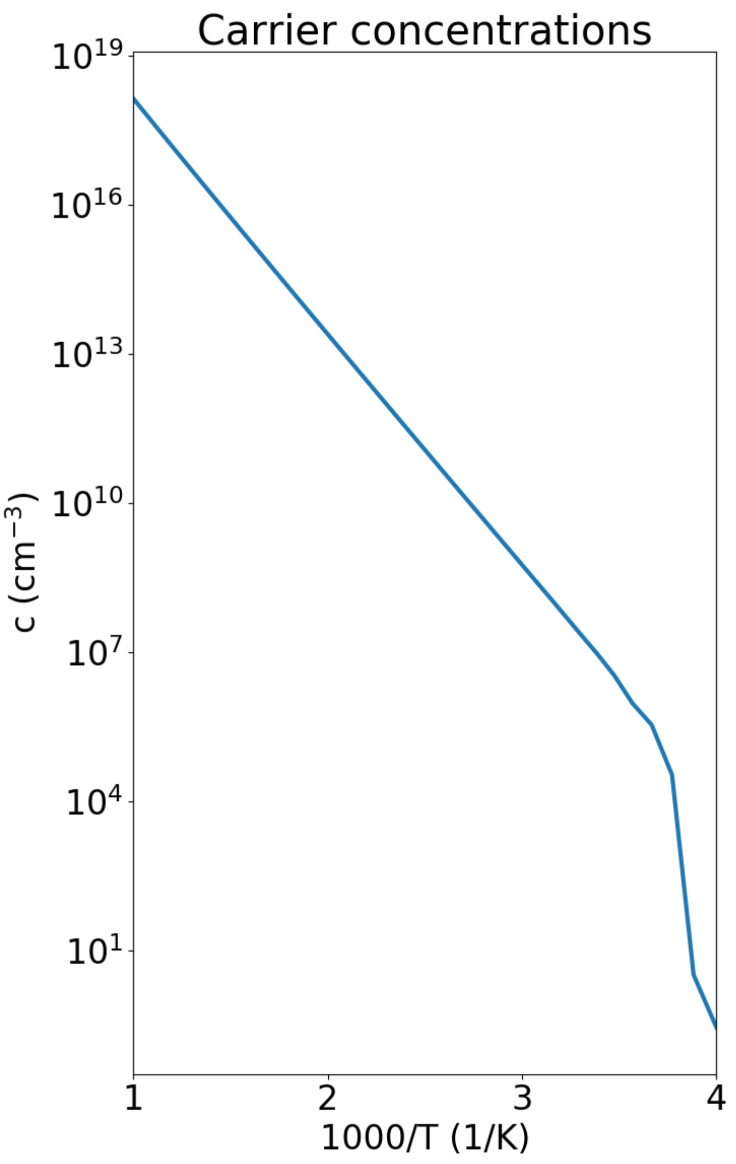
```
import matplotlib.pyplot as plt

carriers = concentrations.equilibrium_carrier_concentrations

# the sign is positive, except for very low temperatures:
# holes are the main carriers
T_plot = 1000/defective_system.temperature_range
```

(continues on next page)


Mg-doped GaN Ga-rich limit
ΔE_f (eV)
E_F (eV)
0
2
4
6
8
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
N_Ga
Ga_int
Ga_N
N_int
Vac_N
Mg_Ga
Vac_Ga

(continued from previous page)

```
plt.plot(T_plot, np.abs(carriers), linewidth=3)
plt.yticks([1e-10, 1, 1e10, 1e20])
plt.xlim(1, 4)
plt.yscale('log')
plt.title('Carrier concentrations')
plt.xlabel('1000/T (1/K)')
plt.ylabel(r'c (cm$^{-3}$)')
plt.tight_layout()
plt.show()
```



As the entry of `carriers` are always positive except for low temperatures, holes are the main charge carriers in the system, meaning that the doping with Mg is effective at high temperatures, while holes are fast compensated by intrinsic defects for lower temperatures.

CHAPTER

FOUR

TUTORIAL

The properties of many materials can be strongly affected by the introduction of impurity atoms. For example, the doping of semiconductor materials lies at the foundation electronics industry.

The amount of impurities sufficient to significantly alter the property of materials is in general in the ppm. Intrinsic defects, which are ubiquitous in every material, also generally appear in this concentration range.

For such small concentrations, it can be assumed that each defect will not feel the presence of other defects. Despite the truth of this claim should be carefully assessed on a case-by-case basis - defect complexes do occurr and concentrations of intrinsic defects might be considerable, as in the case of nonstoichiometric compounds - defect mutual independence, that is considering the *dilute limit*, is generelly assumed since it allows to easily obtains relevant properties of defective systems, such as defect and carrier concentrations.

Spinney implements the formulism devised for point defects in the *dilute limit*, it is also assumed that first-principles calculations of point defects are performed within the supercell formalism [PTA+92], which is currently the most widely employed.

This tutorial will review the most common steps involved in the study of point defects in solids using first-principles simulations and will show how **Spinney** can be used to assist with the investigation.

4.1 The defect formation energy in the supercell approach

Within the supercell approach, the formation energy of a point defect d in charge state q can be expressed as the difference in grand potential that the formation of the defect in an otherwise pristine crystal entails [ZN91]:

$$\Delta E_f(d;q) = G(d;q) - G(\text{bulk}) - \sum_i n_i \mu_i + q\left(\epsilon_{VBM} + E_F\right) + E_{corr} \tag{4.1}$$

where $\Delta E_f(d;q)$ represents the defect formation energy, $G(d;q)$ and $G(\text{bulk})$ are the free energies of the defective and pristine supercells, respectively, n_i are the amount of atoms added or removed from the host material in order to create the defect, μ_i are the chemical potentials of these atoms, ϵ_{VBM} is the valence band maximum eigenvalue, E_F is the Fermi level, which can range within the material band gap, and E_{corr} is an energy term that corrects for finite-size effects originating from the use of the supercell approach. As a common approximation, instead of the free energies we will use the electronic energies, the main quantity calculated by first-principles codes.

Note: In the following tutorials, to illustrate how $\Delta E_f(d;q)$ can be calculated with **Spinney**, we will use an example the formation energy of a B vacancy in the charge state -3 in cubic BN.

Read the *quickstart guide* for a succint tutorial explaining how to calculate the defect formation energy with **Spinney** using output files from VASP and WIEN2k.

Otherwise, read the *In-depth Tutorial*.

Contents

4.1.1 In-depth Tutorial

Initialize a `PointDefect` object

The easiest way to calculate $\Delta E_f(d;q)$ in **Spinney** is by using the *PointDefect* class. To initialize a `PointDefect` object, we need a `Atoms` object of the ASE library representing the defective system. Such object must have attached a `Calculator` implementing a `get_total_energy()` method that returns the energy of the defective system.

Using VASP

ASE supports several first-principles codes; for example, if VASP is used, and the result of the calculation of the defective system is stored in the `OUTCAR_def` file, then an `Atoms` object suitable for initializing a `PointDefect` instance can be simply obtained using the `ase.io.read()` function. The `PointDefect` object can in this case be initialized with few lines of Python code:

```python
from spinney.structures.pointdefect import PointDefect
import ase.io

# use ASE to read the OUTCAR file of the defective system
outcar = ase.io.read('OUTCAR_def', format='vasp-out')
# initialize a Spinney PointDefect object
pd = PointDefect(outcar)
```

Using WIEN2k or other codes not fully supported by ASE

While ASE can read the outputs of several first-principles codes, not all of them are supported. For example, up to the version `3.17`, the last ASE version compatible with **Spinney**, there is no function that can read the `.scf` files produced by WIEN2k. In this case, one can write an *ad hoc* function serving this purpose.

Warning: When writing the helper functions needed to create an ASE `Atoms` object to be used for initialize a **Spinney** `PointDefect` instance from the output files of a first-principles suite not yet supported by ASE, you should convert lengths in Angstrom and energies in eV. This is the defalut in ASE and what is assumed by the `PointDefect` class.

In the case of WIEN2k, ASE can read the `.struct` files of WIEN2k and return the proper `Atoms` object. In this case create the ASE `Atoms` object required by `PointDefect` is easy.

Suppose you have calculated the defective supercell using WIEN2k and have the files `defective.scf` and `defective.struct`. In this case, the general way for initializing a `PointDefect` object is:

1. read the total energy calculated by WIEN2k in the `defective.scf` file, for example using a `grep` command on Linux:

   ```
   grep :ENE defective.scf
   ```

2. use ASE to read the `defective.struct` file and initialize an `Atoms` object:

   ```
   struct=ase.io.read('defective.struct', format='struct')
   ```

3. use the *`DummyAseCalculator`* class to initialize a *dummy* calculator and use the method `set_total_energy()` to insert the calculated energy found in point 1 (**the energy has to be converted into units of eV, which is the energy unit used in ASE**),

   ```
   from spinney.structures.pointdefect import PointDefect, DummyAseCalculator
   from spinney.constants import conversion_table
   # convert the energy read from the .scf file to eV
   energy = -5048.05765754*conversion_table['Ry']['eV']
   calc = DummyAseCalculator(struct)
   calc.set_total_energy(energy)
   ```

4. attach the dummy calculator to the *Atoms* object created in step 2,

   ```
   struct.set_calculator(calc)
   ```

5. Use this *Atoms* object to initialize a new *PointDefect* object.

   ```
   pd = PointDefect(struct)
   ```

This procedure can be generally used for any first-principles code for which ASE can create an `Atoms` object with the structural information but without information about the system energy.

For WIEN2k, **`Spinney`** offers some helping functions. The proper `Atoms` object that can be used to initialize a `PointDefect` object can be obtained using the function *`prepare_ase_atoms_wien2k()`* which takes as arguments the `.struct` and `.scf` files and returns the `Atoms` object with a dummy calculator.

```
from spinney.io.wien2k import prepare_ase_atoms_wien2k
ase_scf = prepare_ase_atoms_wien2k('defective.struct', 'defective.scf')
pd = PointDefect(ase_scf)
```

Similar helping functions can be easily written for any other *ab initio* code.

Calculating the defect formation energy

Once a new `PointDefect` instance has been created, we can feed it the data necessary to calculate the defect formation energy. Then needed data come from the very definition of $\Delta E_f(d;q)$ of equation (4.1).

Feed the required data to the `PointDefect` instance

1. **Energy of the host material**.

   $G(\mathrm{bulk})$ is read from an ASE `Atoms` object representing the pristine system. Such object must thus have attached a calculator. The necessary steps are analogous to the one described in the previous section for the defective system. Once the instance has been created, we can feed it to the `PointDefect` object:

   ```
   # for example, if VASP was used
   ase_pristine = ase.io.read('OUTCAR_pristine', format='vasp-out')
   pd.set_pristine_system(ase_pristine)
   ```

2. **Chemical potential of the elements involved in the creation of the defect**.

   Suppose we are interested in the defect formation energy in the B-rich limit. In order to find μ_{B}, we need to calculate the reference state for the B atom. We can take it as the $\alpha-\mathrm{B}$ phase in the rhombohedral crystal family. μ_{B} is then equal to the system energy per atom. μ_{N} is then simply obtained as $\mu_{\mathrm{N}} = E_{\mathrm{BN}} - \mu_{\mathrm{B}}$. Suppose μ_{B} and μ_{N} have been stored in the variables `chem_pot_B` and `chem_pot_N`, respectively. Then we can inform the `PointDefect` object that we want to use these values of the chemical potentials by typing:

   ```
   pd.set_chemical_potential_values({'N':chem_pot_N, 'B':chem_pot_B})
   ```

3. **Information about the pristine system**.

   From equation (4.1), we see that we also need ϵ_{VBM} and E_F. If the values are stored in the variables `e_vbm` and `fermi_level`, respectively, then we can feed them to the `PointDefect` object using:

   ```
   pd.set_vbm(e_vbm)
   pd.set_fermi_level_value_from_vbm(fermi_level)
   ```

4. **Information about the defective system**.

   This information is needed in order to compute E_{corr}. This term mainly corrects for electrostatic finize-size effects due to the presence of charged defects. As other finize-size errors can generelly be made negligibly small by increasing the supercell size [FGH+14], the **`Spinney`** package implements two popular correction schemes for correcting electrostatic finite-size effects, as explained in detail in *Correction schemes for electrostatic finite-size effects*. Each of correction scheme requires some specific data in order to compute E_{corr}; additionally, some data are required by every scheme.

   General data:

   - The defect charge state q, `q=-3` in our example:

     ```
     pd.set_defect_charge(q)
     ```

   - The defect position with respect to the supercell basis vectors, `def_position=(0.5, 0.5, 0.5)`, if the vacancy was placed in the centre of the supercell:

     ```
     pd.set_defect_position(def_position)
     ```

   - The dielectric constant/tensor of the host material `dielectric_tensor`:

     ```
     pd.set_dielectric_tensor(dielectric_tensor)
     ```

   Correction-scheme-specific data:

   - First we need to inform the `PointDefect` instance about which correction scheme we intend to use. **`Spinney`** implements two state-of-the-art methods: the scheme proposed by Freysoldt, Neugebauer and

Van de Walle, [FNVdW09] and the improved version proposed by Kumagai and Oba [KO14]. More information about such schemes can be found in *Correction schemes for electrostatic finite-size effects*.

To specify the correction scheme of Freysoldt, Neugebauer and Van de Walle:

```
pd.set_finite_size_correction_scheme('fnv')
```

To specify the correction scheme of Kumagai and Oba:

```
pd.set_finite_size_correction_scheme('ko')
```

- Next, we need to add the required scheme-specific data. In this example we will consider the schem of Kumagai and Oba. We refer to *Correction schemes for electrostatic finite-size effects* for details about the scheme of Freysoldt, Neugebauer and Van de Walle.

  The method of Kumagai and Oba requires the electrostatic potential calculated at the ionic sites for the pristine and defective systems. Such data have to be stored in two arrays, `potential_pristine`, `potential_defective`. The way these data are obtained is specific of the employed first-principle package. We provide functions to extract such information from VASP and WIEN2k outputs.

  - **VASP**: the required output files are the OUTCAR files for the pristine and defective supercells, `OUTCAR_pristine` and `OUTCAR_def`, respectively.

    ```
    from spinney.io.vasp import extract_potential_at_core_vasp
    potential_pristine = extract_potential_at_core_vasp('OUTCAR_pristine')
    potential_defective = extract_potential_at_core_vasp('OUTCAR_def')
    ```

  - **WIEN2k**: the required output files are the `.struct` and `.vcoul` files for the pristine and defective systems: `pristine.struct/.vcoul` and `defective.struct/.vcoul`, respectively.

    ```
    from spinney.io.wien2k import extract_potential_at_core_wien2k
    potential_pristine = extract_potential_at_core_wien2k('pristine.struct',
    ↪'pristine.vcoul')
    potential_defective = extract_potential_at_core_wien2k('defective.struct',
    ↪ 'defective.vcoul')
    # convert to eV
    potential_pristine *= conversion_table['Ry']['eV']
    potential_defective *= conversion_table['Ry']['eV']
    ```

  - **Other first-principles codes**: for other codes, one needs to write an output-specific helper function in order to extract the ionic-site electrostatic potential and initialize the arrays: `potential_pristine` and `potential_defective`.

  Once the arrays containing the electrostatic potential at the ionic sites have been created, we can feed the to the `PointDefect` instance:

  ```
  pd.add_correction_scheme_data(potential_pristine=potential_pristine,
                                potential_defective=potential_defective)
  ```

  the *`add_correction_scheme_data()`* takes correction-scheme-specific keyword arguments.

Obtaining the defect formation energy

Once the `PointDefect` object has been initializated and the needed data has been fed to it, we can obtain the defect formation energy.

```
# energy without adding corrections for electrostatic finite-size effects
uncorrected_energy = pd.get_defect_formation_energy()
# corrected energy
corrected_energy = pd.get_defect_formation_energy(True)
```

4.1.2 Quickstart Guide

Calculate the formation energy of charged point defects using **`Spinney`** is easy and fast. In this getting-started guide we will explain how to obtain this quantity for a Boron vacancy in charge state -3 in the B-rich limit. For correcting for electrostatic finite-size effects we apply the method of Kumagai and Oba [KO14].

We give explicit examples using two popular first-principles code:

- *VASP*
- *WIEN2k*

Each *ab initio* code produces different outputs; however the information necessary to calculate the defect formation energy is generally present somewhere in these output files. To use **`Spinney`** with a different first-principle code, you will need to create some *ad hoc* helper functions. Read *the whole guide* for more information.

Using VASP

The snippet of code below shows how the formation energy of the defect can be calculated from VASP output files. It is assumed that the script working directory contains the following files:

- OUTCAR files of the pristine and defective supercells: `OUTCAR_pristine` and `OUTCAR_defective`, respectively.
- OUTCAR file for the reference state of bulk B: `OUTCAR_B`

Moreover, the following variables need to be created:

- `vbm`: the valence-band-maximum eigenvalue of the host material.
- `e_r`: the dielectric tensor/constant of the host material.
- `defect_position`: the defect position in the supercell in fractional coordinates.
- `q`: the defect charge state (-3 in this example).

```
from spinney.structures.pointdefect import PointDefect
from spinney.io.vasp import extract_potential_at_core_vasp
from spinney.tools.formulas import count_elements

import ase.io

### initialize a point defect object
ase_defective = ase.io.read('OUTCAR_defective', format='vasp-out')
pd = PointDefect(ase_defective)

### Feed it the data
ase_pristine = ase.io.read('OUTCAR_pristine', format='vasp-out')
```

(continues on next page)

(continued from previous page)

```
pd.set_pristine_system(ase_pristine)
# get the chemical potential of Boron
ase_boron = ase.io.read('OUTCAR_B', format='vasp-out')
chem_pot_B = ase_boron.get_total_energy()/ase_boron.get_number_of_atoms()
# get the chemical potential of Nitrogen in the B-rich conditions
elements = count_elements(ase_pristine.get_chemical_formula())
chem_pot_N = ase_pristine.get_total_energy() - elements['B']*chem_pot_B
chem_pot_N /= elements['N']
pd.set_chemical_potential_values({'N':chem_pot_N, 'B':chem_pot_B})
# set valence band maximum, the Fermi level is set by default to zero
pd.set_vbm(vbm)
# if one wants a Fermi level not equal to 0, uncomment:
# pd.set_fermi_level_value_from_vbm(E_F)

# data for the correction scheme
pd.pd.set_defect_charge(q)
pd.set_defect_position(defect_position)
pd.set_dielectric_tensor(e_r)
# scheme of Kumagai and Oba
pd.set_finite_size_correction_scheme('ko')
# get the required data and feed them to the instance
pot_pristine = extract_potential_at_core_vasp('OUTCAR_pristine')
pot_defective = extract_potential_at_core_vasp('OUTCAR_defective')
pd.add_correction_scheme_data(potential_pristine=pot_pristine,
                              potential_defective=pot_defective)

### calculate the defect formation energy
uncorrected_energy = pd.get_defect_formation_energy()
corrected_energy = pd.get_defect_formation_energy(True)
```

Using WIEN2k

The snippet of code below shows how the formation energy of the defect can be calculated from WIEN2k output files. It is assumed that the script working directory contains the following files:

- `.struct`, `.scf` and `.vcoul` files of the pristine and defective supercells: `pristine.struct/.scf/.vcoul` and `defective.struct/.scf/.vcoul`, respectively.
- `.struct` and `.scf` files for the reference state of bulk B: `boron.struct/.scf`

Moreover, the following variables need to be created:

- `vbm`: the valence-band-maximum eigenvalue of the host material.
- `e_r`: the dielectric tensor/constant of the host material.
- `defect_position`: the defect position in the supercell in fractional coordinates.
- `q`: the defect charge state (-3 in this example).

```
from spinney.structures.pointdefect import PointDefect
from spinney.io.wien2k import prepare_ase_atoms_wien2k
from spinney.io.wien2k import extract_potential_at_core_wien2k
from spinney.tools.formulas import count_elements
from spinney.constants import conversion_table

### initialize a point defect object
```

(continues on next page)

(continued from previous page)

```python
ase_defective = prepare_ase_atoms_wien2k('defective.struct', 'defective.scf')
pd = PointDefect(ase_defective)

### Feed it the data
ase_pristine = prepare_ase_atoms_wien2k('pristine.struct', 'pristine.scf')
pd.set_pristine_system(ase_pristine)
# get the chemical potential of Boron
ase_boron = prepare_ase_atoms_wien2k('boron.struct', 'boron.scf')
chem_pot_B = ase_boron.get_total_energy()/ase_boron.get_number_of_atoms()
# get the chemical potential of Nitrogen in the B-rich conditions
elements = count_elements(ase_pristine.get_chemical_formula())
chem_pot_N = ase_pristine.get_total_energy() - elements['B']*chem_pot_B
chem_pot_N /= elements['N']
pd.set_chemical_potential_values({'N':chem_pot_N, 'B':chem_pot_B})
# set valence band maximum, the Fermi level is set by default to zero
pd.set_vbm(vbm)
# if one wants a Fermi level not equal to 0, uncomment:
# pd.set_fermi_level_value_from_vbm(E_F)

# data for the correction scheme
pd.pd.set_defect_charge(q)
pd.set_defect_position(defect_position)
pd.set_dielectric_tensor(e_r)
# scheme of Kumagai and Oba
pd.set_finite_size_correction_scheme('ko')
# get the required data and feed them to the instance
pot_pristine = extract_potential_at_core_wien2k('pristine.struct',
                                                'pristine.vcoul')
pot_defective = extract_potential_at_core_wien2k('defective.struct',
                                                 'defective.vcoul')
# convert to eV
pot_pristine *= conversion_table['Ry']['eV']
pot_defective *= conversion_table['Ry']['eV']
pd.add_correction_scheme_data(potential_pristine=pot_pristine,
                              potential_defective=pot_defective)

### calculate the defect formation energy
uncorrected_energy = pd.get_defect_formation_energy()
corrected_energy = pd.get_defect_formation_energy(True)
```

4.1.3 Manage a defective system with the class `DefectiveSystem`

One is usually interested in the study of several point defects in a defective system. In this case, one could insert the code snippets shown above into a loop and create a *PointDefect* instance for each point defect.

This however might result in too much visual noise. **Spinney** offers the class *DefectiveSystem* to manage several point defects in the same host material. To initialize a new instance, one needs to specify the directory containing the results of the first-principles calculations (`data_path`) and the code used to produce them. The class expects this kind of directory tree structure:

```
"data_path"
 ├── data_defects
 │   ├── "defect_name"
 │   │   ├── "charge_state"
 │   │   │   └── "files"
```

(continues on next page)

(continued from previous page)

```
│    │    ├── "charge_state"
│    │    │    └── "files"
│    │    ├── "charge_state"
│    │    │    └── "files"
│    │    └── "charge_state"
│    │         └── "files"
│    ├── "defect_name"
...
└── pristine
     └── "files"
```

- the names `data_defect` and `pristine` are directories and are mandatory, as the class will look in these places in order to find the required information about the defective and pristine systems.
- `files` are the output of the first-principles calculations required to calculate the defect formation energy. The number and type of required files depends on the employed first-principles code and on whether one wants to apply finite-size-effects corrections.
  - For VASP: one needs at least the OUTCAR file. If the correction scheme `fnv` has to be used, then also the LOCPOT file must be present.
  - For WIEN2k: one needs at least the case.struct and case.scf files. If the correction scheme `ko` has to be used, then also the case.vcoul file must be present. The `fnv` method has not yet been implemented in WIEN2k.
  - In all cases, if any correction scheme has to be used, one needs to add a file named `position.txt` with the fractional coordinates of the defective site on a single line, each entry separated by a white space. For example:

    ```
    0.5 0.5 0.5
    ```

- `charge_state` are directories and must be integers representing the defect charge state. The value specify by `charge_state` will be considered to be the charge state of the defect.
- `defect_name` are directories, they can have any name that can describe the defective system.
- Any other directories can appears below `data_path`, these will be ignored by the class.

We can take as an example the study of the defect chemistry of intrinsic GaN using the code VASP. In this case the `data_path` directory tree would look like this:

```
data_path
├── data_defects
│    ├── Ga_int
│    │    ├── 0
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    ├── 1
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    ├── 2
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    └── 3
│    │         ├── OUTCAR
│    │         └── position.txt
│    ├── Ga_N
│    │    ├── 0
```

(continues on next page)

(continued from previous page)

```
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    ├── 1
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    ├── -1
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    ├── 2
│    │    │    ├── OUTCAR
│    │    │    └── position.txt
│    │    └── 3
│    │         ├── OUTCAR
│    │         └── position.txt
...
├── Ga
│    └── OUTCAR
├── N2
│    └── OUTCAR
└── pristine
     └── OUTCAR
```

We can now initialize a new instance of `DefectiveSystem`:

```
defective_system = DefectiveSystem('data', 'vasp')
```

Similarly to the initialization of a `PointDefect` object, in order to calculate the defect formation energies we need some data relative to the pristine system and the chemical potential values. These can be assigned as object attributes:

```
# calculate chemical potential values in the Ga-rich limit
path_pristine = os.path.join('data', 'pristine', 'OUTCAR')
path_ga = os.path.join('data', 'Ga', 'OUTCAR')
ase_ga = ase.io.read(path_ga, format='vasp-out')
ase_pristine = ase.io.read(path_pristine, format='vasp-out')
chem_pot_ga = ase_ga.get_total_energy()/ase_ga.get_number_of_atoms()
elements = count_elements(ase_pristine.get_chemical_formula())
chem_pot_n = ase_pristine.get_total_energy() - elements['Ga']*chem_pot_ga
chem_pot_n /= elements['N']

# valence band maximum and dielectric tensor
vbm = 5.009256
e_rx = 5.888338 + 4.544304
e_rz = 6.074446 + 5.501630
e_r = [[e_rx, 0, 0], [0, e_rx, 0], [0, 0, e_rz]]

# add the data
defective_system.vbm = vbm
defective_system.dielectric_tensor = e_r
defective_system.chemical_potentials = {'Ga':chem_pot_ga, 'N':chem_pot_n}

# use the correction scheme of Kumagai and Oba
defective_system.correction_scheme = 'ko'
# calculate defect formation energies for each point defect
# and print output to the screen
defective_system.calculate_energies(verbose=True)
```

The calculated defect formation energy are stored in a *Pandas* dataframe and can be accessed from the attribute *`data`*:

```
df = defective_system.data
printf(df)
```

will output:

```
                   Form Ene (eV)  Form Ene Corr. (eV)
Defect Charge
N_Ga     2          7.022139            7.272291
         3          7.435422            8.143598
        -1          9.977779           10.237273
         0          8.271712            8.271712
         1          7.461967            7.464786
Ga_int   2          3.190741            3.956642
         3          1.487580            3.063845
         0          8.339166            8.339166
         1          5.710694            5.964960
...
```

they can be written to a text file, in a format readable by other modules of **Spinney** by calling `write_formation_energies()`:

```
defective_system.write_formation_energies('formation_energies_GaN_Ga_rich.txt')
```

The `PointDefect` objects for each point defect processed by the `DefectiveSystem` instance are accessible in a list through the attribute `point_defects`:

```
for pdf in defective_system.point_defects:
    print(pdf.my_name)
```

will print:

```
N_Ga 2
N_Ga 3
N_Ga -1
N_Ga 0
N_Ga 1
Ga_int 2
Ga_int 3
Ga_int 0
Ga_int 1
...
```

4.2 Correction schemes for electrostatic finite-size effects

Contents

4.2.1 Theoretical introduction

The main artifact of the supercell approach for point-defect calculations consists in the introduction of periodic images of the defect located in the simulation cell. Such periodically-repeated array of defects corresponds to very high defect concentrations for commonly used supercell types.

In such case, defect-defect interactions are large and can considerably affect the predicted energy of the point defect.

Among the kinds of defect-defect interactions, electrostatic ones are never negligible for any practical supercell size.

The study of point defects in the dilute limit then requires some scheme to correct for such spurious electrostatic interactions.

The introduction of a point defect will induce a redistribution of the host material electronic charge density. In the ideal case of an isolated point defect, we call this defect-induced charge density ρ_{iso}. On the other hand, in the supercell method, due to the application of periodic boundary conditions, the defect-induced charge density will be different: ρ_{per}. Moreover, for charged point defects, periodic boundary conditions require the introduction of a neutralizing background, usually taken as an homogeneous jellium of density $-\frac{q}{V}$, where q is the defect charge state and V is the supercell volume.

The defect-induced electrostatic potential can be obtained from the induced charge density by solving Poisson equation with the proper boundary conditions. This will yield the potentials ϕ_{iso} and ϕ_{per} for the ρ_{iso} and ρ_{per}, respectively:

$$\nabla^2\phi(\mathbf{r}) = -4\pi\rho(\mathbf{r})$$

Correction schemes for electrostatic finite-size effects assume that the point defect induces a charge density localized in the supercell. In this case the charge densities induced by periodic and isolated defect can be considered to be the same within the supercell: $\rho_{iso} = \rho_{per} = \rho$ in V. In addition:

$$q = \int_V \rho_{iso}(\mathbf{r})\, d\mathbf{r} = \int_V \rho_{per}(\mathbf{r})\, d\mathbf{r}$$

Considering the defect-induced electrostatic energy per supercell, what it is obtained employing periodic boundary conditions is the quantity:

$$E_{per} = \frac{1}{2}\int_V \phi_{per}(\mathbf{r})\left(\rho(\mathbf{r}) - \frac{q}{V}\right)\, d\mathbf{r}$$

while ideally we would like to obtain:

$$E_{iso} = \frac{1}{2}\int_V \phi_{iso}(\mathbf{r})\rho(\mathbf{r})\, d\mathbf{r}$$

Correction schemes for electrostatic finite-size effects aim to calculate the corrective term:

$$E_{corr} = E_{iso} - E_{per} \tag{4.2}$$

`Spinney` implements two state-of-the-art correction schemes:

- The scheme proposed by Freysoldt, Neugebauer and Van de Walle [FNVdW09].
- The scheme proposed by Kumagai and Oba [KO14].

The latter is an improvement over the former. In both cases, the correction energy of equation (4.2) is expressed as:

$$E_{corr} = -E_{lat} + q\Delta\phi$$

E_{lat} takes into account the interaction of ρ with the host material and the jellium background in periodic boundary conditions, while $\Delta\phi$ is an alignment term.

ρ is modelled with a spherically-symmetric charge distribution. In the scheme of Freysoldt, Neugebauer and Van de Walle this is generally taken as a linear combination of a Gaussian function and an exponential one:

$$\rho(r) = q\left(xN_1e^{-r/\gamma} + (1-x)N_2e^{-r^2/\beta^2}\right) \tag{4.3}$$

where N_1 and N_2 are normalization constants. γ and β are the parameters describing the exponential and Gaussian functions, respectively. In the scheme of Kumagai and Oba a point-charge model is used instead. This allows to easily generalize the scheme to anisotropic materials.

Regarding $\Delta\phi$, this is obtained from comparing the electrostatic potential of the defective and pristine systems with the one generated by the model ρ in a region of the crystal far from the defect. The scheme of Freysoldt, Neugebauer and Van de Walle uses the plane-averaged potential in order to compute the alignment term; while the scheme of Kumagai and Oba uses atomic-site potentials. The latter have been show to converge better far from the defect, in particularly when lattice relaxations are important [KO14].

4.2.2 Implementation in `Spinney`

The *`PointDefect`* class can calculate the energy for correcting electrostatic finite-size effects using both the scheme of Freysoldt, Neugebauer and Van de Walle and the one of Kumagai and Oba.

Each scheme needs its own set of data in order to compute E_{corr}, this have to fed to a `PointDefect` instance using the method *`add_correction_scheme_data()`*. This method accepts correction-scheme-specific keywords arguments, which we will explain now.

Warning: each time *`add_correction_scheme_data()`* is called, all the keywords which have not been explicitly given by the user will take default values. This means that every time the method is called, the `PointDefect` object will overwrite the data passed with a former call of `add_correction_scheme_data`.

The scheme of Freysoldt, Neugebauer and Van de Walle

The keywords arguments for *`add_correction_scheme_data()`* are:

- **Mandatory arguments**:
  - `potential_pristine`: a 3D array containing the electrostatic potential calculated by first-principles on a 3D grid for the **pristine** supercell.
  - `potential_defective`: a 3D array containing the electrostatic potential calculated by first-principles on a 3D grid for the **defective** supercell.
  - `axis`: an integer with value `0`, `1` or `2` which specifies the cell vector along which the plane-averaged electrostatic potential will be calculated.
- **Optional arguments**:

- `defect_density`: a 3D array containing a model defect-induced charge density calculated on a 3D grid. For example, `defect_density` can be obtained by projecting the electronic charge density onto the defect-induced band.

  If this argument is present, **Spinney** will fit the model charge density of equation (4.3) to `defect_density`.
- `gamma`: a float. The parameter of the exponential function γ in equation (4.3). The default value is 1.
- `beta`: a float. The parameter of the Gaussian function β in equation (4.3). The default value is 1.
- `x_comb`: a float between 0 and 1. Weight of the exponential function with respect to the Gaussian function in modeling the defect-induced charge density. Default `x_comb = 0`. For `x_comb = 0` the charge density is modelled by a pure Gaussian; for `x_comb = 1` by a pure exponential function.
- `e_tol`: a float, break condition for the iterative calculation of the correction energy. Value in Hartree. Default: 1e-6 Ha.
- `shift_tol`: a float representing the tolerance to be used to locate the defect position along `axis`. The default value is: $1e - 5 \times a$, where a is the length of the cell parameter defining `axis`.

Note: Most of the times the default parameters will be enough.

You might need to increase `shift_tol` considerarbly if the 3D used to calculate the electrostatic potential is too coarse.

Example using VASP

To illustrate in detail how the correction scheme of Freysoldt, Neugebauer and Van de Walle is implemented in **Spinney**, we will take the example of a Ga vacancy in the charge state -3 in cubic GaAs, modeled using a $3 \times 3 \times 3$ supercell of the conventional cubic cell. This system is also studied in the authors' original paper [FNVdW09].

The electrostatic potential on a 3D grid can be obtained in VASP by adding to the `INCAR` file the following lines (example for VASP 5.2.12, compare with the documentation for you version):

```
PREC = High
LVHAR = .TRUE.
NGXF = 200
NGYF = 200
NGZF = 200
```

These options will write the `LOCPOT` file on a fine grid (the grid can be considerably less dense, depending on the system).

The data for the `potential_pristine` and `potential_defective` keyword arguments can be obtained easily using the class `VaspChargeDensity`.

```
from ase.calculators.vasp import VaspChargeDensity

locpot = VaspChargeDensity('LOCPOT')
potential_pristine = -1*locpot.chg[-1]*locpot.get_volume()
```

We can now calculate the defect formation energy. To do so we assume that the working directory contains the following files:

- `OUTCAR_prist` and `OUTCAR_def`: `OUTCAR` files for pristine and defective supercell, respectively.
- `LOCPOT_prist` and `LOCPOT_def`: `LOCPOT` files for pristine and defective supercell, respectively.

- OUTCAR_Ga: OUTCAR file for the reference state of Ga.

```
import numpy as np
from ase.calculators.vasp import VaspChargeDensity
import ase.io
from spinney.structures.pointdefect import PointDefect
from spinney.defects import fnv

# pristine system
pristine = ase.io.read('OUTCAR_prist')

# Specifications of the defective system: V_Ga -3 in GaAs
q = -3 # charge state
dielectric_constant = 12.4 # from Freysoldt et al. PRL (2009) paper
def_position = np.array([0.33333333333, 0.5, 0.5]) # fractional coordinates defect
axis_average = 2 # axis along where the average potential is calculated
vbm = 4.0513 # valence band maximum

# prepare a numpy 3D array with information about the electrostatic potential
locpot_prist = VaspChargeDensity('LOCPOT_prist')   # read using ase
supercell = locpot_prist.atoms[-1]
locpot_def = VaspChargeDensity('LOCPOT_def')
locpot_arr_p = locpot_prist.chg[-1]*supercell.get_volume()*(-1)
locpot_arr_def = locpot_def.chg[-1]*supercell.get_volume()*(-1)
del locpot_prist; del locpot_def

# Prepare the chemical potentials Ga-rich conditions
gallium = ase.io.read('OUTCAR_Ga')
mu_ga = gallium.get_total_energy()/gallium.get_number_of_atoms()

# initialite PointDefect instance
defective = ase.io.read('OUTCAR_def')
pd = PointDefect(ase.io.read(defect))
pd.set_dielectric_tensor(dielectric_constant)
pd.set_defect_position(def_position)
pd.set_defect_charge(q)
pd.set_pristine_system(pristine)
pd.set_vbm(vbm)
# correction scheme of Freysoldt, Neugebauer and Van de Walle
pd.set_finite_size_correction_scheme('fnv')
pd.add_correction_scheme_data(potential_pristine=locpot_arr_p,
                              potential_defective=locpot_arr_def,
                              axis=axis_average)
pd.set_chemical_potential_values({'Ga':mu_ga, 'As':None}, force=True)
print('Formation energy Ga-rich conditions, not corrected: ',
      pd.get_defect_formation_energy())
print('Formation energy Ga-rich conditions, corrected: ',
      pd.get_defect_formation_energy(True))
```

Example using WIEN2k

To illustrate in detail how the correction scheme of Freysoldt, Neugebauer and Van de Walle is implemented in **Spinney**, we will take the example of a Ga vacancy in the charge state -3 in cubic GaAs, modeled using a $3 \times 3 \times 3$ supercell of the conventional cubic cell. This system is also studied in the authors' original paper [FNVdW09].

Obtaining more information on the finite-size corrections

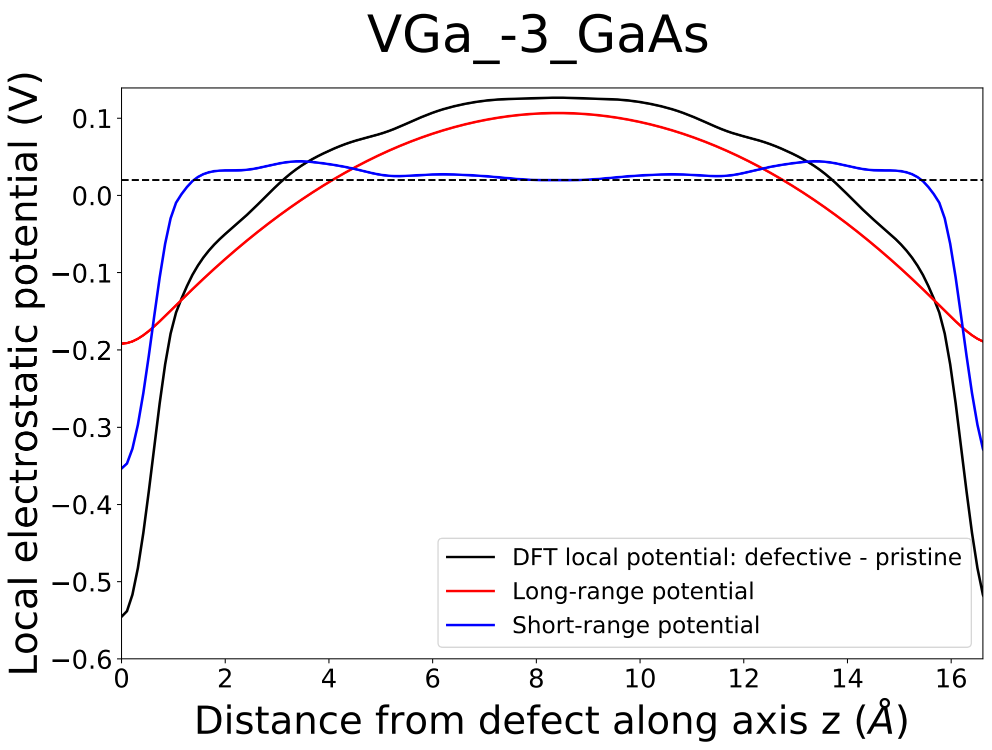
More information related to the correction scheme can be accessed through the method *calculate_finite_size_correction()* with the keyword argument `verbose=True`.

This returns a tuple, whose first element is the correction energy for finite-size effects and the second element is a dictionary.

The python interface is independent on the employed first-principles code. We only assume that a `PointDefect` instance has been initialized, as shown above. Such object is assumed to have been stored in the variable `pd`.

```python
ecorr, dd = pd.calculate_finite_size_correction(verbose=True)
# 1D grid along which the average potential has been calculated
axis_grid = dd['Potential grid']
# calculated long-range potential
lr_potential = dd['Model potential']
# calculated short-range potential
sr_potential = dd['Alignment potential']
# DFT potential defective system - pristine system
dft_potential = dd['DFT potential']
# Potential alignment part of the correction energy
alignment_term = dd['Alignment term']
# utility class to plot the electrostatic potential between the defect and
# its image
Plott = fnv.FPlotterPot(axis_grid, dft_potential, lr_potential, sr_potential,
                        -alignment_term, 'z')
# this will save a pdf file called plot_VGa_-3_GaAs.pdf
Plott.plot('VGa_-3_GaAs')
```

The result of this last block of code is the following picture:



The scheme of Kumagai and Oba

The keywords arguments for *add_correction_scheme_data()* are:

- **Mandatory arguments**:
  - `potential_pristine`: a 1D array of length equal to the number of atoms in the pristine system. The array elements store the electrostatic potential calculated by first-principles at the atomic sites in the **pristine** supercell.
  - `potential_defective`: a 1D array of length equal to the number of atoms in the defective system. The array elements store the electrostatic potential calculated by first-principles at the atomic sites in the **defective** supercell.
- **Optional arguments**:
  - `distance_tol`: a float or an array with 3 elements. The rounding tolerance for comparing distances. If a float is insert as input, it will be converted to an array of 3 elements. Each element is the tolerance value in percentage of the length of the corresponding cell parameter. The default value is 1% for each cell parameter.
  - `e_tol`: a float. The condition for breaking the loop in the iterative calculation of the correction energy. Value in eV. The default value is 1e-6 eV.

Example using VASP

To illustrate in detail how the correction scheme of Kumagai and Oba is implemented in **Spinney**, we will take the example of a B vacancy in the charge state -3 in cubic BN, modeled using a $3 \times 3 \times 3$ supercell of the conventional cubic cell. This system is also studied in the authors' original paper [KO14].

All the data needed by the Kumagai's and Oba's scheme are in the `OUTCAR` file.

The following block of code assumes that the working directory contains the files:

- `OUTCAR_prist`, `OUTCAR_def`: the `OUTCAR` files of the pristine and defective supercells, respectively.
- `OUTCAR_B`: the `OUTCAR` file of the reference state for Boron.

To calculate the defect formation energy with the correction scheme of Kumagai and Oba, you can use this snippet:

```
import numpy as np

import ase.io
from spinney.io.vasp import extract_potential_at_core_vasp
from spinney.structures.pointdefect import PointDefect

# pristine system
pristine = ase.io.read('OUTCAR_prist')

# Specifications of the defective system: V_B -3 in BN
q = -3 # charge state
e_r = (4.601064 + 2.314707) # electronic and ionic contribution to dielectric
                            # constant, calculated with VASP
a = 0.41667
def_position = np.ones(3)*a # fractional coordinates defect
vbm = 7.2981 # valence band maximum

# Prepare the chemical potentials B-rich conditions
boron = ase.io.read('OUTCAR_B')
mu_b = boron.get_total_energy()/boron.get_number_of_atoms()

# prepare a numpy array with information about the electrostatic potential
pot_prist = extract_potential_at_core_vasp('OUTCAR_prist')
pot_def = extract_potential_at_core_vasp('OUTCAR_def')

# initialite PointDefect instance
defective = ase.io.read('OUTCAR_def')
pd = PointDefect(defective)
pd.set_dielectric_tensor(e_r)
pd.set_defect_position(def_position)
pd.set_defect_charge(q)
pd.set_pristine_system(pristine)
pd.set_vbm(vbm)
pd.set_chemical_potential_values({'N':None, 'B':mu_b}, force=True)
# correction scheme of Kumagai and Oba
pd.set_finite_size_correction_scheme('ko')
pd.add_correction_scheme_data(potential_pristine=pot_prist, potential_defective=pot_
↪def)
print('Formation energy B-rich, uncorrected: {:.3f}'.format(
    pd.get_defect_formation_energy()))
print('Formation energy B-rich, corrected: {:.3f}'.format(
    pd.get_defect_formation_energy(True)))
```

The script will print:

```
Formation energy B-rich, uncorrected:  11.801
Formation energy B-rich, corrected:  14.275
```

Example using WIEN2k

To illustrate in detail how the correction scheme of Kumagai and Oba is implemented in **Spinney**, we will take the example of a B vacancy in the charge state -3 in cubic BN, modeled using a $3 \times 3 \times 3$ supercell of the conventional cubic cell. This system is also studied in the authors' original paper [KO14].

The data that are needed are the `.struct`, `.scf` and `.vcoul` files for the pristine and defective supercell.

Note: By default WIEN2k will not write the `.vcoul` file. To write it you need to modify the second line of the `.in0` file, by replacing `NR2V` with `R2V` before running your calculations.

The following block of code assumes that the working directory contains the files:

- `pristine.struct` and `defective.struct`: the `.struct` files for the pristine and defective supercell, respectively.
- `pristine.scf` and `defective.scf`: the `.scf` files for the pristine and defective supercell, respectively.
- `pristine.vcoul` and `defective.vcoul`: the `.vcoul` files for the pristine and defective supercell, respectively.
- `boron.struct` and `boron.scf`: the `.struct` and `.scf` files for the reference state of Boron.

To calculate the defect formation energy with the correction scheme of Kumagai and Oba, you can use this snippet:

```
import numpy as np

from spinney.structures.pointdefect import PointDefect
from spinney.constants import conversion_table
from spinney.io.wien2k import prepare_ase_atoms_wien2k
from spinney.io.wien2k import extract_potential_at_core_wien2k

# pristine system
pristine = pristine = prepare_ase_atoms_wien2k('pristine.struct',
                                               'pristine.scf')

# Specifications of the defective system: V_B -3 in BN
q = -3 # charge state
e_r = (4.601064 + 2.314707) # electronic and ionic contribution to dielectric
                            # constant
def_position = np.zeros(3) # fractional coordinates defect
vbm = 0.7037187925*conversion_table['Ry']['eV'] # valence band maximum

# Prepare the chemical potentials B-rich conditions
boron = prepare_ase_atoms_wien2k('boron.struct', 'boron.scf')
mu_b = boron.get_total_energy()/boron.get_number_of_atoms()

# prepare a numpy array with information about the electrostatic potential
pot_prist = extract_potential_at_core_wien2k('pristine.struct',
                                             'pristine.vcoul')
pot_prist *= conversion_table['Ry']['eV']
pot_def = extract_potential_at_core_wien2k('defective.struct',
                                           'defective.vcoul')
```

(continues on next page)

(continued from previous page)

```
pot_def *= conversion_table['Ry']['eV']

# initialite PointDefect instance
defective = prepare_ase_atoms_wien2k('defective.struct',
                                     'defective.scf')
pd = PointDefect(defective)
pd.set_dielectric_tensor(e_r)
pd.set_defect_position(def_position)
pd.set_defect_charge(q)
pd.set_pristine_system(pristine)
pd.set_vbm(vbm)
pd.set_chemical_potential_values({'N':None, 'B':mu_b}, force=True)
# correction scheme of Kumagai and Oba
pd.set_finite_size_correction_scheme('ko')
pd.add_correction_scheme_data(potential_pristine=pot_prist, potential_defective=pot_
↪def)
print('Formation energy B-rich, uncorrected: {:.3f}'.format(
    pd.get_defect_formation_energy()))
print('Formation energy B-rich, corrected: {:.3f}'.format(
    pd.get_defect_formation_energy(True)))
```

The script will print:

```
Formation energy B-rich, uncorrected: 11.995
Formation energy B-rich, corrected: 14.503
```

Obtaining more information on the finite-size corrections

More information related to the correction scheme can be accessed through the method *calculate_finite_size_correction()* with the keyword argument `verbose=True`.

This returns a tuple, whose first element is the correction energy for finite-size effects and the second element is a dictionary.

The dictionary contains the various contributions to E_{corr}, some information about the sampled atomic-site potential and the instance of the `KumagaiCorr`` class that has been used to calculate the correction energy term. Such instance can be accessed from the dictionary with the keyword `Corr object`. From it, we can extract useful information, which allow us, for example to plot the following picture:

This image can be obtained using this code snippet. The python interface is independent on the employed first-principles code. We only assume that a `PointDefect` instance has been initialized, as shown above. Such object is assumed to have been stored in the variable `pd`. For the meaning of the various potentials plotted in the figure, consult the authors’s paper in reference [KO14].

```
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams.update({'font.size': 26})

ecorr, dd = pd.calculate_finite_size_correction(verbose=True)
corr_obj = dd['Corr object']

plt.figure(figsize=(8,8))
dist, pot_align_vs_dist = corr_obj.alignment_potential_vs_distance_sampling_region
plt.scatter(dist, pot_align_vs_dist, color='green', marker='^',
```

(continues on next page)

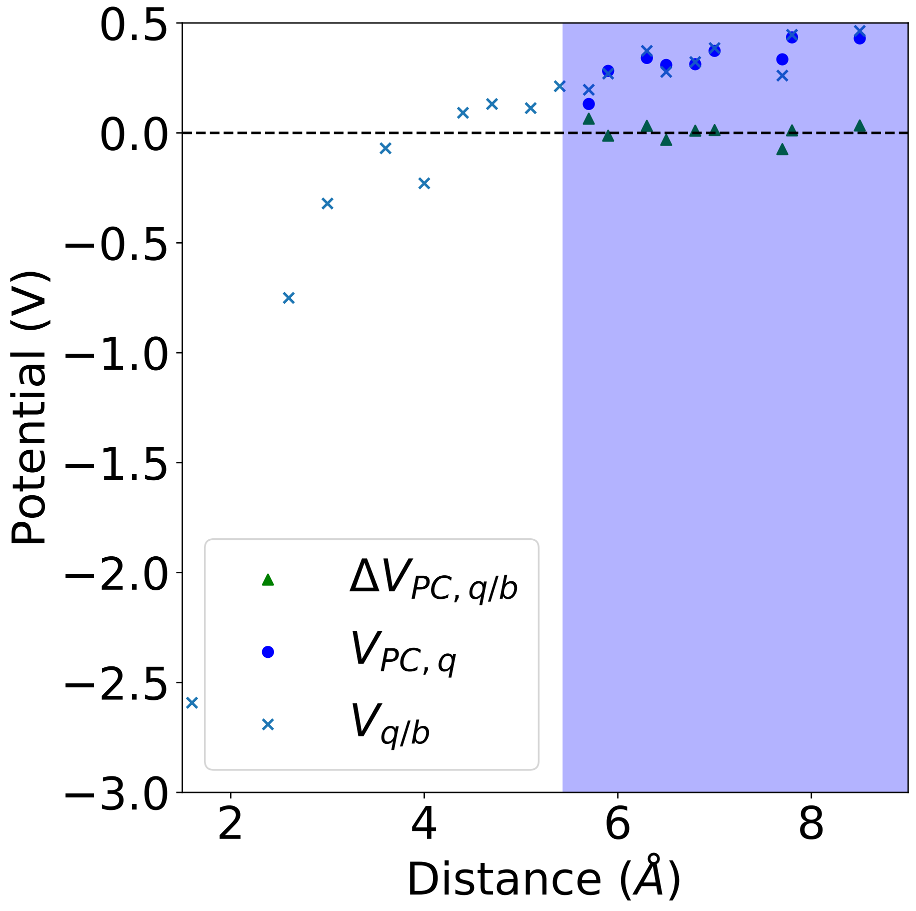


Fig. 4.1: The picture shows the relevant potentials as a function of the distance from the point defect. Far from the defect the potentials should be relatively flat and smooth. The shaded area represents the sampling region. Potentials that are strongly oscillating in the sampling region indicate that either the supercell is too small, or the defect-induced charge density is not very localized. In either case, convergence of the predicted defect formation energy as a function of the supercell size should be carefully assessed.

(continued from previous page)

```
            label=r'$\Delta V_{PC, q/b}$')
dist, pot = corr_obj.ewald_potential_vs_distance_sampling_region
plt.scatter(dist, pot, label=r'$V_{PC, q}$', color='blue')
dist, pot = corr_obj.difference_potential_vs_distance
plt.scatter(dist, pot, label=r'$V_{q/b}$', marker='x')
# beginning of the sampling region
radius = corr_obj.sphere_radius
plt.axvspan(radius, 9, color='blue', alpha=0.3)
plt.plot(np.linspace(1.5, 9, 100), np.zeros(100), color='black',
         linestyle='--')
plt.legend()
plt.xlabel(r'Distance ($\AA$)')
plt.ylabel(r'Potential (V)')
plt.xlim(1.5, 9)
plt.ylim(-3, 0.5)
plt.tight_layout()
plt.savefig('atomic_site_potential_convergence.pdf', format='pdf')
plt.show()
```

4.3 Thermodynamic limits for the chemical potentials

Contents

The defect formation energy of equation (4.1) explictly depends on the chemical potentials of the elements which are exchanged with the reservoir in order to introduct the point defect.

The thermodynamic stability of the host crystal imposes some constraints on the values these chemical potentials can assume.

Consider for example the rutile phase of $\mathrm{TiO_2}$. If μ_i represents the chemical potential of element i and $\Delta\mu_i$ represents its value with respect to the element chemical potential in its standard state (say the HCP structure for Ti and the triplet state of the $\mathrm{O_2}$ molecule for O), the following constraint have to be satisfied:

$$\begin{aligned}\mu_{\mathrm{Ti}} + 2\mu_{\mathrm{O}} &= \mu_{\mathrm{TiO_2(rutile)}}\\ x\mu_{\mathrm{Ti}} + y\mu_{\mathrm{O}} &\leq \mu_{\mathrm{Ti_xO_y}}\\ \mu_{\mathrm{Ti}} &\leq \mu_{\mathrm{Ti(HCP)}}\\ \mu_{\mathrm{O}} &\leq \frac{1}{2}\mu_{\mathrm{O_2}}\end{aligned}$$

Or equivalently:

$$\begin{aligned}\Delta\mu_{\mathrm{Ti}} + 2\Delta\mu_{\mathrm{O}} &= \Delta h_{\mathrm{TiO_2(rutile)}}\\ x\Delta\mu_{\mathrm{Ti}} + y\Delta\mu_{\mathrm{O}} &\leq \Delta h_{\mathrm{Ti_xO_y}}\\ \Delta\mu_{\mathrm{Ti}} &\leq 0\\ \Delta\mu_{\mathrm{O}} &\leq 0\end{aligned}$$

where Δh is the formation enthalpy per formula unit.

4.3.1 Limiting values of the chemical potentials

Defect formation energies are usually reported for the extreme conditions; in this case:

- O-rich conditions, where μ_{O} achieves its maximum value;
- Ti-rich conditions, where μ_{Ti} achieves its maximum values.

Such values can be obtained from the solution of the optimization problem of equation (4.3) or (4.3).

Spinney offers the *Range* class for this purpose.

Suppose that the formation energies per formula unit of the relevant Ti-O phases have been calculated and written on a text file, *formation_energies.txt* like the one below:

```
# Formation energies per formula unit calculated with PBEsol
#Compound                  E_f (eV/fu)
Ti                           0.0000000000
O2                           0.0000000000
TiO2_rutile                 -9.4742301250
Ti2O3                      -15.2633270550
TiO                         -5.3865639660
Ti3O5                      -24.8910858875
```

The problem in equation (4.3) can be solved creating an instance of the *Range* class.

The data needed by this class can be obtained from a text file analogous to the one above using the function *get_chem_pots_conditions()*.

This function takes four arguments:

- The text file with the calculated formation energies. *get_chem_pots_conditions()* assumes that each row represents either the formation energies **per formula unit** calculated for a compound with respect to its elements in the standard state, or the total energy of the compound, again **per formula unit**.

  Comments can be added by starting a line with #.

  Each line containing the energy data must have the form: `compound_name<delimiter>energy`. `compound_name` must be a string containing the formula unit of the compound. Underscores can then be added to separate other strings. `<delimiter>` is a string that divides `compound_name` and `energy`. By default blank spaces are assumed. `energy` is a string representing a floating-point number.
- A list specifying the order of the variables (that is, the chemical elements).
- A list with all the `compound_name` in the text file that represent compounds characterized by an equality constraint.
- (Optional) A string that specifies `<delimiter>` used in the text file.

The returned values can be used to initialize the *Range* class and some of its methods, like shown in the example below:

```
from spinney.thermodynamics.chempots import Range, get_chem_pots_conditions

data = 'formation_energies.txt'

equality_compounds = ['TiO2_rutile']
order = ['O', 'Ti']
parsed_data= get_chem_pots_conditions(data, order, equality_compounds)
```

(continues on next page)

(continued from previous page)

```
# prepare Range instances
crange = Range(*parsed_data[:-1])
```

The minimum and maximum value that each variable (chemical potential) can obtain within the feasible region can by accessed through the attribute `variables_extrema` of the class `Range`. This returns a 2D array, with n rows and 2 columns, with n being the number of variables in equation (4.3). Each row gives the minimum and maximum value of one variable, the order is the same we specified in `order`.

For example, typing:

```
print(crange.variables_extrema)
```

returns:

```
array([[-3.53160448e+00, -8.94320056e-14],
       [-9.47423008e+00, -2.41102114e+00]])
```

The first row is relative to $\Delta\mu_{\mathrm{O}}$ and indicates that this variable can range between -3.53 eV and 0 eV. The end point of this interval represent the value of $\Delta\mu_{\mathrm{O}}$ in the Ti-rich and O-rich limit, respectively.

One can see that $\Delta\mu_{\mathrm{Ti}}$ has a maximum value of -2.41 eV (Ti-rich limit), meaning that thermodynamic equilibrium between rutile and Ti is not possible.

4.3.2 Plotting the feasible region

Competing phases can be easily visualized by plotting the feasible region. This can be done in **Spinney** adding this code snippet to the previous script:

```
range.set_compound_dict(parsed_data[-1])
# let's use some pretty labels in the plot
# the order of the axes  must follow the order used for get_chem_pots_conditions
labels = [r'$\Delta \mu_{O}$ (eV)', r'$\Delta \mu_{Ti}$ (eV)']
crange.plot_feasible_region_on_plane([0,1], x_label=labels[0],
                                     y_label=labels[1],
                                     title='Rutile phase', save_plot=True)
```

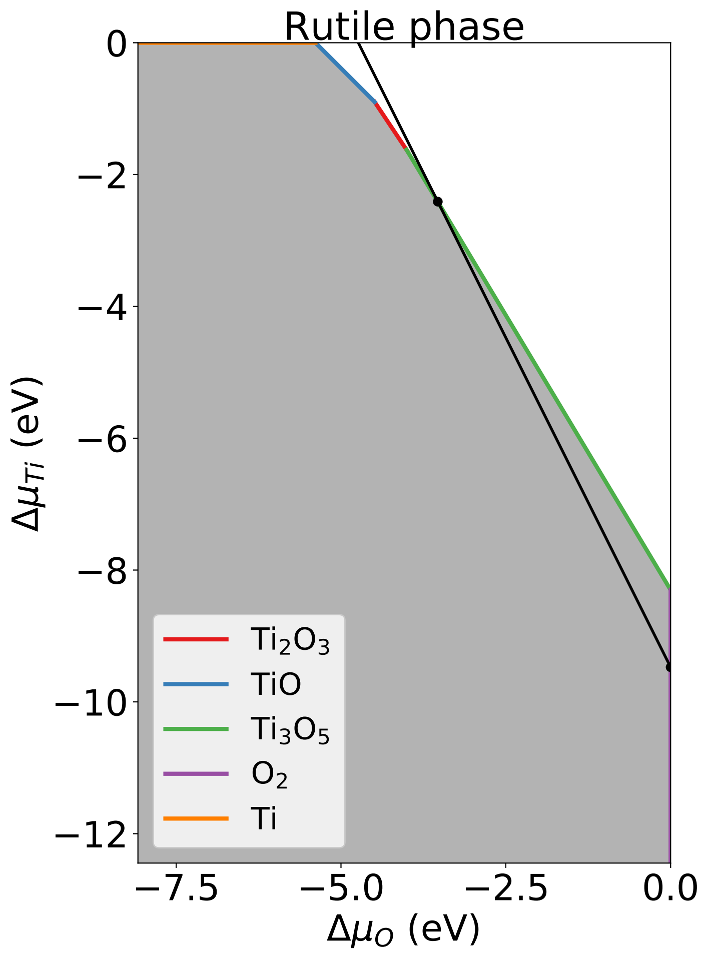
Which saves the plot in the pdf file `feasible_region_plane_0_1.pdf`:

The feasible region has been intersected with the `0-1` plane, which is the plane spanned by the independent chemical potentials. The ordering of these is the one that has been specified in `order`. In our example the variable 0 is $\Delta\mu_{\mathrm{O}}$ and variable 1 is $\Delta\mu_{\mathrm{Ti}}$. If instead we would have used `[1, 0]` as the first argument of `range.plot_feasible_region_on_plane`, we would have obtained a plot with switched axes, saved as `feasible_region_plane_1_0.pdf`

This plot represents the feasible region for the problem described by equation (4.3). In particular, the shaded area is the feasible region determined by the inequality constratints and the black line is determined by the equality constraint. Such plots offer a convenient way to visualize the relevant competing phases for given values of the element chemical potentials.

From the picture it is clear that $\mathrm{Ti_3O_5}$ would precipitate before the chemical potential of Ti reaches its standard state value.

Warning: A *PointDefect* object takes the **absolute** values of the chemical potentials in order to calculate the defect formation energy. These can be obtained from $\Delta\mu$ by adding to it the chemical potential of the standard


Rutile phase
0
−2
−4
−6
−8
−10
−12
−7.5
−5.0
−2.5
0.0
$\Delta\mu_O$ (eV)
$\Delta\mu_{Ti}$ (eV)
Ti_2O_3
TiO
Ti_3O_5
O_2
Ti

state. Equivalently, these values can be obtained by using equation (4.3) instead of equation (4.3) by modifying the file *formation_energies.txt* accordingly.

4.3.3 Beyond binary compounds

Suppose we now want to consider rutile doped with Nb. We consider as a possible competing phase $TiNb_2O_7$ and NbO_2 (other compounds could be considered of course).

The file *formation_energies.txt* is modified accordingly and renamed *formation_energies_with_nb.txt*:

```
# Formation energies per formula unit calculated with PBEsol
#Compound                  E_f (eV/fu)
Ti                           0.0000000000
O2                           0.0000000000
TiO2_rutile                 -9.4742301250
Ti2O3                      -15.2633270550
TiO                         -5.3865639660
Ti3O5                      -24.8910858875
TiNb2O7                    -28.9191827617
Nb                           0.0000000000
NbO2                       -8.06605626125
```

To instantiate the proper `Range` object, one can type:

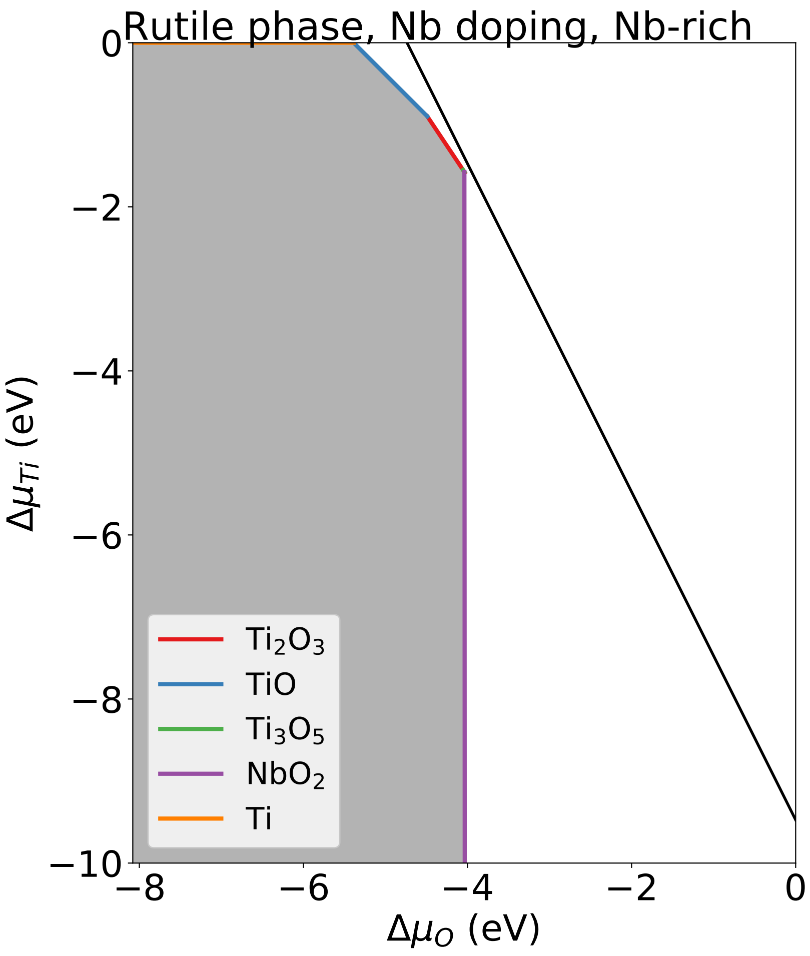
```
data_nb = 'formation_energies_with_nb.txt'

equality_compounds = ['TiO2_rutile']
order = ['O', 'Ti', 'Nb'] # must insert the new species
parsed_data= get_chem_pots_conditions(data_nb, order, equality_compounds)
# prepare Range instances
crange_nb = Range(*parsed_data[:-1])

crange_nb.set_compound_dict(parsed_data[-1])
labels = [r'$\Delta \mu_{O}$ (eV)', r'$\Delta \mu_{Ti}$ (eV)', r'$\Delta \mu_{Nb}$␣
↪(eV)']
```

We can now plot the feasible region on the O-Ti plane. Since we have an additional variable, we must specify the value of $\Delta\mu_{\mathrm{Nb}}$ that defines the plane intersecting the feasible region. We can do so by using the argument *plane_values* of `plot_feasible_region_on_plane()`. This is a list and specifies the values of the variables which are not used as independent variables in the plot.

We can for example consider Nb-rich conditions, where $\Delta\mu_{\mathrm{Nb}} = 0$. So that the independent variables are $\Delta\mu_{\mathrm{O}}$ and $\Delta\mu_{\mathrm{Ti}}$. To plot the intersection of the feasible region with the plane $\Delta\mu_{\mathrm{Nb}} = 0.$ of the 3D $\Delta\mu_{\mathrm{O}}$-$\Delta\mu_{\mathrm{Ti}}$-$\Delta\mu_{\mathrm{Nb}}$ space we set *plane_axes=[0,1]* and *plane_values=[0]*:

```
crange_nb.plot_feasible_region_on_plane([0, 1], x_label=labels[0],
                                        y_label=labels[1],
                                        plane_values = [0],
                                        y_val_min = -10, # minimum value second␣
↪variable
                                        title='Rutile phase, Nb doping, Nb-rich',
                                        save_plot=True,
                                        save_title='feasible_region_Nb_rich')
```

This snippet will save a pdf file called *feasible_region_Nb_rich.pdf*:

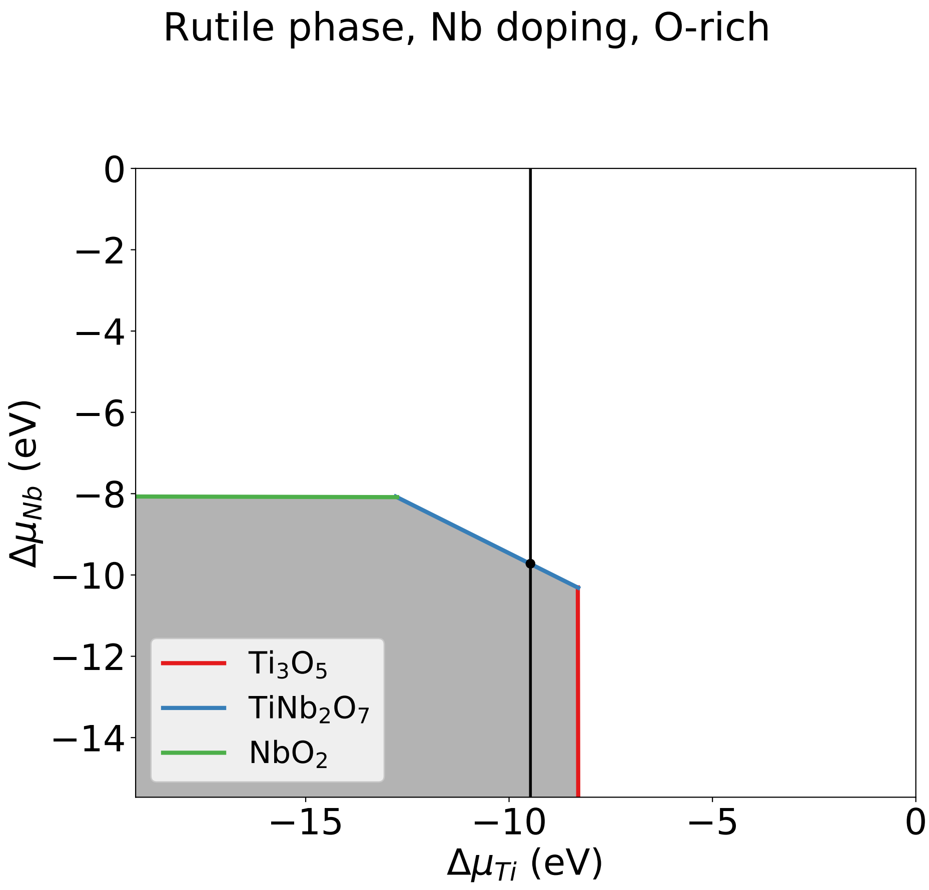

Rutile phase, Nb doping, Nb-rich
0
−2
−4
−6
−8
−10
−8
−6
−4
−2
0
$\Delta\mu_{Ti}$ (eV)
$\Delta\mu_{O}$ (eV)
Ti_2O_3
TiO
Ti_3O_5
NbO_2
Ti

From the feasible region it is clear that for Nb-rich conditions ($\Delta\mu_{\mathrm{Nb}} = 0$) rutile would not be thermodynamically stable, as $\mathrm{NbO_2}$ would precipitate instead.

We could instead ask for which value of $\Delta\mu_{\mathrm{Nb}}$ rutile would be stable in O-rich ($\Delta\mu_{\mathrm{O}} = 0$) conditions. To do so, we itersect the feasible region with the plane $\Delta\mu_{\mathrm{O}} = 0$ by setting *plane_axes=[1, 2]* and *plane_values=[0]*.

```
crange_nb.plot_feasible_region_on_plane([1, 2], x_label=labels[1],
                                        y_label=labels[2],
                                        plane_values = [0],
                                        title='Rutile phase, Nb doping, O-rich', save_
↪plot=True,
                                        save_title='feasible_region_O_rich')
```

The snippet will produce the picture:



We can see that the maximum value of $\Delta\mu_{\mathrm{Nb}}$ compatible with the stability of rutile in oxygen-rich conditions is determined by the formation of $\mathrm{TiNb_2O_7}$. Using the class attribure `variables_extrema_2d` one can have access to this value. The attribute is defined analogously as `variables_extrema`, but considers the optimization problem after the intersection has been considered and refers to the two independent variables defining the plane axes, $\Delta\mu_{\mathrm{Ti}}$ and $\Delta\mu_{\mathrm{Nb}}$ in this case.

```
print(crange_nb.variables_extrema_2d)
```

prints:

```
[[-9.47423013 -9.47423013]
 [       -inf -9.72247632]]
```

The second row of the array corresponds to $\Delta\mu_{\mathrm{Nb}}$ and shows that its maximum value compatible with the existance of rutile when $\Delta\mu_{\mathrm{O}} = 0$ is around -9.72 eV. At this point both $\Delta\mu_{\mathrm{Nb}}$ and $\Delta\mu_{\mathrm{O}}$ are fixed, there are then no more degrees of freedom and $\Delta\mu_{\mathrm{Ti}}$ can only have a single value which is fixed by the constraint of coexistance between rutile and $\mathrm{TiNb_2O_7}$.

4.3.4 Including temperature and pressure effects through the gas-phase chemical potentials

Spinney implements, for the common gas species $\mathrm{O_2}$, $\mathrm{H_2}$, $\mathrm{N_2}$, $\mathrm{F_2}$, and $\mathrm{Cl_2}$, conventient expressions for calculating the chemical potentials as a function of temperature and pressure. It uses the following formula, based on the Shomate equation [MWC98] and an ideal gas model:

$$\mu(T,p) = h(0,p^\circ) + [g(T,p^\circ) - g(0,p^\circ)] + k_B T \ln\left(\frac{p}{p^\circ}\right)$$

p° represents the standard pressure of 1 bar, g the molar free energy and $h(0,p^\circ)$ the molar enthalpy at zero temperature and standard pressure.

For example, consider the case of $\mathrm{O_2}$. In this case we would use the class *OxygenChemPot* class. For each of the above-mentioned species there is an analogous class whose construct takes two arguments: the unit of energy used to return μ and the unit of pressure. Valid units can be found in the variable *spinney.constants.available_units*:

```
available_units = ['J', 'eV', 'Ry', 'Hartree', 'kcal/mol', 'kJ/mol',
                   'm', 'Angstrom', 'Bohr', 'nm', 'cm',
                   'Pa', 'kPa', 'Atm', 'Torr']
```

For $\mathrm{O_2}$:

```
from spinney.thermodynamics.chempots import OxygenChemPot

o2 = OxygenChemPot(energy_units='eV', pressure_units='Atm')
```

We can now use the method `get_ideal_gas_chemical_potential_Shomate()` to obtain the chemical potential values as a function of temperature and pressure. The signature of this method is:

```
get_ideal_gas_chemical_potential_Shomate(mu_standard, partial_pressure, T)
```

mu_standard is a scalar, representing $h(0,p^\circ)$. It can be set equal to the electronic energy calculated for the molecule, alternatively, in this example we will set it to zero, in such a way we would obtain $\Delta\mu_{O_2}$, that is the difference from the reference state.

Both *partial_pressure* and *T* can be either scalars or 1-dimensional arrays.

For example, this snippet will plot $\Delta\mu_{O_2}$ as a function of temperature for $p = 1$ Atm.

Note: The Shomate equation is valid only for some values of T, if the input values are not in this range, a *ValueError* will be raised. The exception message will indicate the possible temperature range for that gas species.

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

```

(continues on next page)

(continued from previous page)

```
from spinney.thermodynamics.chempots import OxygenChemPot

o2 =  OxygenChemPot(energy_units='eV', pressure_units='Atm')

T_range = np.linspace(300, 1000, 200)
chem_pot_vs_T = o2.get_ideal_gas_chemical_potential_Shomate(0, 1, T_range)

plt.plot(T_range, chem_pot_vs_T, linewidth=2)
plt.xlabel('T (K)')
plt.ylabel(r'$\Delta \mu_{O_2}$ (eV)')
plt.show()
```

And this snippet will plot the same graph but at tree different pressures:

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

from spinney.thermodynamics.chempots import OxygenChemPot

o2 =  OxygenChemPot(energy_units='eV', pressure_units='Atm')

T_range = np.linspace(300, 1000, 200)
pressures = [1, 10, 1e-6]
chem_pot_vs_T = o2.get_ideal_gas_chemical_potential_Shomate(0, pressures, T_range)

plt.plot(T_range, chem_pot_vs_T, linewidth=2)
plt.legend([str(p) + ' Atm' for p in pressures])
plt.xlabel('T (K)')
plt.ylabel(r'$\Delta \mu_{O_2}$ (eV)')
plt.show()
```

4.4 Charge Transition Levels

In section *Thermodynamic limits for the chemical potentials* we have discussed how **Spinney** can be used for obtaining the values of the chemical potentials in given conditions.

The defect formation energy, equation (4.1), depends also on the chemical potential of the electron, usually expressed as $\epsilon_{VMB} + E_F$.

For this reason, the defect formation energy is often plotted as a function of the Fermi level E_F. **Spinney** offers the class `Diagram` for this purpose.

Such diagrams are also useful for visualizing the defect thermodynamic charge transition levels, which define the value of E_F for which two defect charge states have the same formation energy.

We can take again the defect chemistry of intrinsic GaN as an illustrative example. In particular, the electronic energy of the intrinsic defects of GaN in several charge states was calculated using a supercell containing 96 atoms and the PBE exchange-correlation functional using a 4x2x2 reciprocal-point mesh. The formation energies will be calculated in the Ga-rich limit.

Contents

4.4.1 Using the class `Diagram`

Step 1: Calculate the Defect Formation Energies

The easiest way to calculate defect formation energies, including electrostatic finite-size effect corrections, is through the `PointDefect` class. Since many defects in several charge states have to be considered, a straightforward way for processing the DFT data is collecting the results in a proper directory hierarchy. For example, using output from VASP, if we decide to use the correction scheme of Kumagai and Oba, which only requires the *OUTCAR* files, the directory tree might look like this (see also *Manage a defective system with the class DefectiveSystem*):

```
data_path
├── data_defects
│   ├── Ga_int
│   │   ├── 0
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 1
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 2
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   └── 3
│   │       ├── OUTCAR
│   │       └── position.txt
│   ├── Ga_N
│   │   ├── 0
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 1
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── -1
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   ├── 2
│   │   │   ├── OUTCAR
│   │   │   └── position.txt
│   │   └── 3
│   │       ├── OUTCAR
│   │       └── position.txt
...
├── Ga
│   └── OUTCAR
├── N2
│   └── OUTCAR
└── pristine
    └── OUTCAR
```

where the calculations result are collected for each charge state of any specific defect, as well for the pristine system and parent compounds.

Now the defect formation energy can be calculated for each defect by walking the directory tree. This code snippet also writes a file containing the formation energies for each defect in each charge state.

```python
import os

 import ase.io
 from spinney.structures.pointdefect import PointDefect
 from spinney.io.vasp import extract_potential_at_core_vasp
 from spinney.tools.formulas import count_elements

 path_defects = os.path.join('data', 'data_defects')
 path_pristine = os.path.join('data', 'pristine', 'OUTCAR')
 path_ga = os.path.join('data', 'Ga', 'OUTCAR')

 # prepare preliminary data
 ase_pristine = ase.io.read(path_pristine, format='vasp-out')
 pot_pristine = extract_potential_at_core_vasp(path_pristine)
 ase_ga = ase.io.read(path_ga, format='vasp-out')

 vbm = 5.009256 # valence band maximum pristine system
 e_rx = 5.888338 + 4.544304
 e_rz = 6.074446 + 5.501630
 # dielectric tensor
 e_r = [[e_rx, 0, 0], [0, e_rx, 0], [0, 0, e_rz]]

 # store defect positions in  dictionary for access convenience
 defect_positions = {'Ga_int' : (0.25, 0.55, 0.55),
                     'N_int'  : (0.5, 0.41667, 0.50156),
                     'Vac_Ga' : (0.5, 0.6666666666, 0.50156),
                     'Vac_N'  : (0.5, 0.3333333333, 0.46045),
                     'Ga_N'   : (0.5, 0.3333333333, 0.46045),
                     'N_Ga'   : (0.5, 0.6666666666, 0.50156)
                     }

 # get the chemical potential of Ga
 chem_pot_ga = ase_ga.get_total_energy()/ase_ga.get_number_of_atoms()
 # get the chemical potential of N in the Ga-rich conditions
 elements = count_elements(ase_pristine.get_chemical_formula())
 chem_pot_n = ase_pristine.get_total_energy() - elements['Ga']*chem_pot_ga
 chem_pot_n /= elements['N']

 # write the formation energy for Fermi level = 0 to a file
 energy_file = open('formation_energies_GaN_Ga_rich.txt', 'w')
 energy_file.write('#{:<8} {:8} {:>30}\n'.format('system',
                                                 'charge',
                                                 'formation energy (eV)'))
 for root, dirs, files in os.walk(path_defects):
     path = root.split(os.sep)
     if 'OUTCAR' in files:
         def_path = os.path.join(root, 'OUTCAR')
         ase_outcar = ase.io.read(def_path)
         pot_defective = extract_potential_at_core_vasp(def_path)
         defect_name = path[-2]
         charge_state = int(path[-1])
         print('Processing defect {} in charge state {}'.format(defect_name,
                                                               charge_state))
```

(continues on next page)

(continued from previous page)

```
        print('Data in: {}'.format(path))
        # prepare PointDefect object
        pdf = PointDefect(ase_outcar)
        pdf.set_pristine_system(ase_pristine)
        pdf.set_chemical_potential_values({'N':chem_pot_n, 'Ga':chem_pot_ga})
        pdf.set_vbm(vbm)
        pdf.set_defect_charge(charge_state)
        pdf.set_defect_position(defect_positions[defect_name])
        pdf.set_dielectric_tensor(e_r)
        pdf.set_finite_size_correction_scheme('ko')
        pdf.add_correction_scheme_data(potential_pristine=pot_pristine,
                                       potential_defective=pot_defective)
        corrected_energy = pdf.get_defect_formation_energy(True)
        energy_file.write('{:<8} {:>8} {:30.10f}\n'.format(defect_name,
                                                           charge_state,
                                                           corrected_energy))
        print('Done.\n------------------------------------------------------')

energy_file.close()
```

A completely similar approach can be used for WIEN2k or any other first-principle code.

Step 2: Get Thermodynamic Charge Transition Levels and plot the Diagram

The file *formation_energies_GaN_Ga_rich.txt*, containing the formation energies calculated for each point defect, has the general format of a text file, with optional headers starting with a *#*:

```
# description or other comments
defect_name    charge_state    formation_energy_at_vbm
```

formation_energy_at_vbm is the calculated value of the formation energy for an electron chemical potential equal to the valence band maximum, *i.e.* for $E_F = 0$.

Thermodynamic charge transition levels can be easily calculated by initializing a `Diagram` object.

The class constructor takes two mandatory parameters: a dictionary containing information about point defects formation energies (at $E_F = 0$) and their charge state and a 2-ple indicating the system valence band maximum and conduction band minimum. The latter argument will give the range within which possible transition levels are considered. Absolute values are intended for μ_e as independent variables; setting the valence band maximum to 0, will give E_F as independent variable.

The former argument, might be obtained from a file analogous to *formation_energies_GaN_Ga_rich.txt* using the function `extract_formation_energies_from_file()`.

The following code snipper will calculate the charge transition levels and print them in the text file *transition_levels.txt*.

```
from spinney.defects.diagrams import extract_formation_energies_from_file, Diagram

data_file = 'formation_energies_GaN_Ga_rich.txt'
defect_dictionary = extract_formation_energies_from_file(data_file)
dgm = Diagram(defect_dictionary, (0, 1.713)) # E_F from 0 to PBE band gap value
dgm.write_transition_levels('transition_levels.txt')
```

The content of *transition_levels.txt* will be:

```
#Defect type  q/q'
Ga_N          2/3      0.441121
              1/2      0.731836
              0/1      1.250109
Ga_int        2/3      0.892796
N_Ga          1/2      0.192495
              0/1      0.806926
...
```

Transition level values can also be accessed in a Python session using the attribute `transition_levels`:

```
dgm.transition_levels
```

returns a Pandas `Series` object with the transition levels.

Extending the band gap

It is well known that standard local and semilocal functionals severely underestimate the band gap of semiconductor materials. For this purpose the valence band maxima are often aligned to those obtained from more accurate functionals, like hybrids. Transition levels can be calculated in this case by specifying an *extended_gap* parameter in the constructor of `Diagram`.

For example, the paper if reference [LVdW17] finds that the valence band maximum of GaN calculated with HSE with the 31% of Hartree-Fock exchange lies 0.85 eV below the valence band maximum found by PBE and predicts a band gap of 3.51 eV.

We can calculate the transition levels in this extended gap by using the following snippet:

```
# Fermi level will range in the HSE range
dgm = Diagram(defect_dictionary, (0, 1.713), (-0.85, - 0.85 + 3.51))
dgm.write_transition_levels('transition_levels_extended.txt')
```

The file *transition_levels_extended.txt* will contain also transition levels located outside the PBE gap:

```
#Defect type  q/q'
Ga_N          2/3      0.441121
              1/2      0.731836
              0/1      1.250109
              -1/0     1.825432
Ga_int        2/3      0.892796
              1/2      2.008318
              0/1      2.374206
...
```

A plot is very useful to visualize these results. The following snippet shows how to make the plot with **Spinney**.

```
from spinney.defects.diagrams import extract_formation_energies_from_file, Diagram

data_file = 'formation_energies_GaN_Ga_rich.txt'
defect_dictionary = extract_formation_energies_from_file(data_file)

dgm = Diagram(defect_dictionary, (0, 1.713), (-0.85, - 0.85 + 3.51))
# use some prettier labels in the plot
dgm.labels = {'Ga_int' : r'$Ga_i$',
              'N_int'  : r'$N_i$',
              'Vac_Ga' : r'$Vac_{Ga}$',
```

(continues on next page)

(continued from previous page)

```
                'Vac_N'  : r'$Vac_N$',
                'Ga_N'   : r'$Ga_N$',
                'N_Ga'   : r'$N_{Ga}$'}
# personalize colors to use in the plot
colors = {'Ga_int' : 'red',
          'N_int'  : 'blue',
          'Vac_Ga' : 'orange',
          'Vac_N'  : 'gray',
          'Ga_N'   : 'magenta',
          'N_Ga'   : 'green'}
# save the plot
dgm.plot(save_flag=True, save_title='diagram',
         title='Intrinsic GaN, Ga-rich conditions', legend=True,
         colors_dict=colors, x_label=r'$E_F$ (eV)')
```

This code will save the file *diagram.pdf* with the plot:

4.4.2 Using the class `DefectiveSystem`

The directories structure used in the previous section is compatible with the one required by an instance of *`DefectiveSystem`*.

Such object has a *`Diagram`* instance accessible through the attribute *`diagram`*. In order to create this `Diagram` object, one needs to specify the parameters `gap_range` and eventually `extended_gap_range` needed by *`Diagram`*. These values can be added using the attributes `gap_range` and `extended_gap_range` of a *`DefectiveSystem`*.

For example, suppose that a *`DefectiveSystem`* instance as been initialized as `defective_system` (see *Manage a defective system with the class DefectiveSystem*). Then for our example, one can type:

```
defective_system.gap_range = (0, 1.713)
defective_system.extended_gap_range = (-0.85, - 0.85 + 3.51)
dgm = defective_system.diagram
```

`dgm` is an initialized *`Diagram`* object, which we can use as shown in the previous sections.

4.5 Equilibrium defect concentrations in the dilute limit

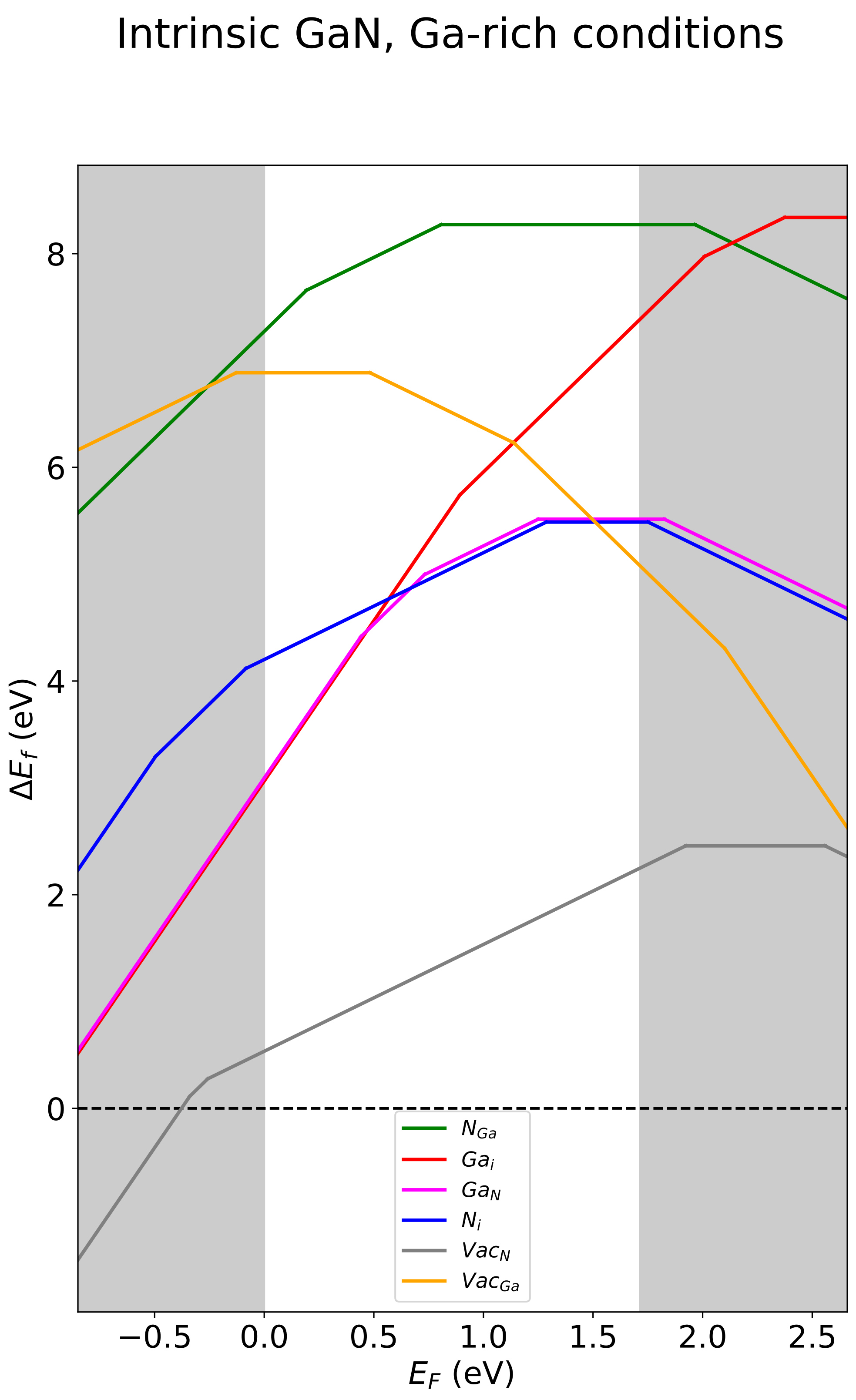
The formation energy $\Delta E_f(d;q)$ of a point defect *d* in charge state *q* is given by equation (4.1). In the dilute limit, one assumes non-interacting defects. In this case, the energy required for forming n defects of type *d* in charge state *q* is simply $n\Delta E_f(d;q) = \Delta E_f(nd;q)$.

At the thermodynamic equilibrium the system grand potential, Φ, is in a minimum:

$$\Phi(nd;q) = \Phi(\text{bulk}) + \Delta E_f(nd;q) + TS_{conf}(n) \tag{4.4}$$

where S_{conf} is the contribution of the configurational entropy to the grand potential of the defective system. Let $x \equiv (d,q)$ and assume that there are g_x possible configurations in which defect *x* has the same $\Delta E_f(d;q)$ (for example given by spin degeneracy). If the crystal is made of N unit cell and in each cell there are γ_x equivalent sites that defect *x* can occupy, the number of possible ways to place n non-interacting defects on $N\gamma_x$ sites is (for $n \ll N\gamma_x$):

$$\Omega_x = g_x^n \binom{N\gamma_x}{n} \tag{4.5}$$


Intrinsic GaN, Ga-rich conditions
ΔE_f (eV)
E_F (eV)
N_{Ga}
Ga_i
Ga_N
N_i
Vac_N
Vac_{Ga}
0
2
4
6
8
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

from which one gets: $S_{conf} = k_B \ln \Omega_x$. The equilibrium defect concentration follows from taking the derivative of (4.4) with respect to n using Stirling's approximation for expressing S_{conf}. One gets:

$$c_x = \frac{n}{N} = \frac{\gamma_x g_x}{\exp\left(\frac{\Delta E_f(d;q)}{k_B T}\right) + g_x} \tag{4.6}$$

Usually $\Delta E_f(d;q) \gg k_B T$ and equation (4.6) is approximated by:

$$c_x = g_x \gamma_x \exp\left(-\frac{\Delta E_f(d;q)}{k_B T}\right)$$

In case of more than one type of defect in the crystal, the equilibrium concentration of each defect is given by formula (4.6), assuming the dilute-limit holds.

Thermodynamic limits for the chemical potentials explained how **Spinney** can help in determining the equilibrium values of the atomic chemical potentils in different thermodynamic conditions. The only chemical potential that needs to be determined is the chemical potential of the electron, whose value is fixed by the charge-neutrality constraint:

$$\sum_x q c_x(\mu_e) + p_0(\mu_e) - n_0(\mu_e) = 0 \tag{4.7}$$

where n_0 is the concentration of free electrons:

$$n_0 = \int_{\epsilon_C}^{\infty} \frac{\omega(\epsilon)}{e^{(\epsilon - \mu_e)/k_B T} + 1} d\epsilon,$$

and p_0 is the concentration of free holes:

$$p_0 = \int_{-\infty}^{\epsilon_V} \frac{\omega(\epsilon)}{e^{(\mu_e - \epsilon)/k_B T} + 1} d\epsilon.$$

$\omega(\epsilon)$ is the density of Kohn-Sham states.

Spinney can find μ_e by finding the roots of equation (4.7) from data provided by the user and obtain the equilibrium defect concentrations.

4.5.1 Calculate equilibrium defect concentrations with `Spinney`

Spinney implements the *`EquilibriumConcentrations`* for calculating equilibrium defect concentrations.

The mandatory arguments for initializing a new instance are:

- `charge_states`: a dictionary mapping each studied defect with the considered charge states.
- `form_energy_vbm`: a dictionary mapping each studied defect with the formation energies calculated at the valence band maximum for each charge state in `charge_states`.
- `vbm`: the value of the valence band maximum used in calculating the defect formation energies recorded in `form_energy_vbm`.
- `e_gap`: the calculated band gap.
- `site_conc`: a dictionary mapping each studied defect with the concentration of available defect sites per unit cell (γ_x of equation (4.5)). In addition, it maps the concentrations of free holes and electrons. In this case the dictionary keys are *hole* and *electron*, respectively.
- `dos`: a 2D array. The first column reports the energies of the one-electron levels sorted in ascending order; the second column reports the corresponding density of states per simulation cell. The value in `dos` must be consistent with the values of `vbm` and `e_gap`.

- `T_range`: a 1D array with the temperatures to be used to calculate the defect concentration.

Optional arguments are:

- `g`: a dictionary mapping each studied defect with its degeneracy for each charge state in `charge_states`.
- `N_eff`: a number indicating an effective doping concentration. Its value will affect the calculated value of the equilibrium electron chemical potential using the equation:

$$\sum_x q c_x(\mu_e) + p_0(\mu_e) - n_0(\mu_e) = N_{eff}$$

- `units_energy`: the units of energy that are used, by default eV are used.
- `dos_down`: for spin-polarized systems, the DOS of spin-down electrons. Data structure completely analogous to `dos`.

We will use as an example the intrinsic defects in GaN, considered also in *Manage a defective system with the class DefectiveSystem* and in *Charge Transition Levels*.

```
charge_states   = {'N_Ga'   : [-1, 0, 1, 2, 3],
                   'Ga_int' : [0, 1, 2, 3],
                   'Ga_N'   : [-1, 0, 1, 2, 3],
                   ...
                  }

form_energy_vbm = {'N_Ga'   : [11.0872728320, 8.2717122490, ...],
                   'Ga_int' : [8.3391662537, 5.1149599825, ...],
                   'Ga_N'   : [8.1902691783, 5.5148367310, ...],
                   ...
                  }
```

Indicating, for example, that the formation energy, for an electron chemical potential equal to the valence band maximum, of a Ga interstitials in charge state +1 is 5.1149599825 eV. **The names used to indicate the various type of point defects must be consistent with each other.** Such data structures can be easily obtained from the text file produced by *`DefectiveSystem`* using the method *`write_formation_energies()`*, which have the format shown in *Step 2: Get Thermodynamic Charge Transition Levels and plot the Diagram*.

So as a first step, one calculates the defect formation energies for the various point defects of interest, for example using a *`DefectiveSystem`* object. As done in previous section we consider in this example the Ga-rich limit:

```
import os

import ase.io
from spinney.structures.defectivesystem import DefectiveSystem
from spinney.tools.formulas import count_elements

path_defects = os.path.join('data', 'data_defects')
path_pristine = os.path.join('data', 'pristine', 'OUTCAR')
path_ga = os.path.join('data', 'Ga', 'OUTCAR')

ase_ga = ase.io.read(path_ga, format='vasp-out')

# Band alignment
vbm_offset = 0.85
vbm = 5.009256 - vbm_offset # align the VBM with the HSE band
e_gap = 1.713
# dielectric tensor
```

(continues on next page)

(continued from previous page)

```
e_rx = 5.888338 + 4.544304
e_rz = 6.074446 + 5.501630
e_r = [[e_rx, 0, 0], [0, e_rx, 0], [0, 0, e_rz]]

# get the chemical potential of Ga
ase_pristine = ase.io.read(path_pristine, format='vasp-out')
chem_pot_ga = ase_ga.get_total_energy()/ase_ga.get_number_of_atoms()
# get the chemical potential of N in the Ga-rich conditions
elements = count_elements(ase_pristine.get_chemical_formula())
chem_pot_n = ase_pristine.get_total_energy() - elements['Ga']*chem_pot_ga
chem_pot_n /= elements['N']

# initialize a DefectiveSystem
defective_system = DefectiveSystem('data', 'vasp')
defective_system.vbm = vbm
defective_system.dielectric_tensor = e_r
defective_system.chemical_potentials = {'Ga':chem_pot_ga, 'N':chem_pot_n}
defective_system.correction_scheme = 'ko'
defective_system.calculate_energies(verbose=False)
# write the defect formation energies in a text file
defective_system.write_formation_energies('formation_energies_GaN_Ga_rich.txt')
```

Note that in:

```
# Band alignment
vbm_offset = 0.85
vbm = 5.009256 - vbm_offset # align the VBM with the HSE band
e_gap = 1.713
```

we have lowered the PBE valence band maximum by 0.85 eV in order to align it with the valence band value obtained using a hybrid functional (see *Charge Transition Levels*). This will lower the formation energies of positive charge states by $q \times 0.85$ and increase those of negative states by $|q| \times 0.85$. So aligning the valence band maximum will have considerable effects on the defect formation energies.

`charge_states` and `form_energy_vbm` can be obtained from the text file *formation_energies_GaN_Ga_rich.txt* calling the function `extract_formation_energies_from_file()` in the module `concentration`. The function takes as the only argument the text file with the defect formation energies and returns the dictionaries to be used as the parameters `charge_states` and `form_energy_vbm` of `EquilibriumConcentrations`:

```
from spinney.defects.concentration import extract_formation_energies_from_file

energy_file = 'formation_energies_GaN_Ga_rich.txt'
charge_states, form_energy_vbm = extract_formation_energies_from_file(energy_file)
```

We can now process the rest of the data:

- `vbm` will be equal to the shifted valence band maximum: `vbm = 5.009256 - vbm_offset`.

  Warning: `vbm` is the valence band maximum of the unit cell used to calculate `dos`. In GaN, the valence band maximum is located at the Γ point and the valence band eigenvalues of the primitive and pristine supercell are basically the same. For other materials, the values could differ. It is important that `vbm` and `dos` are always consistent with each other.

- `e_gap` is directly obtained from the output of first-principles calculations. Here we use the value predicted by PBE for the primitive cell.

- `site_conc` can be determined for example by looking at the multiplicities of the Wyckoff position of the site where the point defect sits. In the wurtzite structure, the Wyckoff position for both Ga and N is $(2b)$. Considering the primitive cell as reference cell, one would have:

  ```
  site_conc = {'Ga_N':4, 'N_Ga':4, 'Vac_N':4, 'Vac_Ga':4,
               'Ga_int':6, 'N_int':6, 'electron':36 , 'hole':36}
  ```

  the site symmetry of the interstitial atoms is compatible with the $(6c)$ Wyckoff position. For electron and holes we used the number of valence electrons per primitive cell. Using such `site_conc` in initializing *EquilibriumConcentrations* would make the code calculate defect concentrations per primitive cell. One usually reports defect concentrations in cm^{-3}. So, for GaN, whose equilibrium volume at the PBE level is 47.04 ^{3}, one needs to multiply the values in `site_conc` by *2.126e+22* in order to obtain concentrations in cm^{-3}.

- `dos` can be straightforwardly obtained from the first-principles calculations. However, in our example we need to modify it. For one, we have to shift the valence band maximum in the first column of `dos` by -0.85 eV, so that it agrees with `vbm`. If `dos` has been read and stored to a 2D `numpy` array, such operation is trivial:

  ```
  dos[:, 0] -= vbm_offset
  ```

- `T_range` can be chosen to be any array of interest. We can calculate defect concentrations from 250 K to 1000 K taking 100 sampling points:

  ```
  T_range = np.linspace(250, 1000, 100)
  ```

A instance of *EquilibriumConcentrations* can now be initialized:

```
concentrations = EquilibriumConcentrations(charge_states, form_energy_vbm,
                                            vbm, e_gap, site_conc, dos, T_range)
```

Equilibrium properties of the system can now be calculated an accessed through the instance attributes:

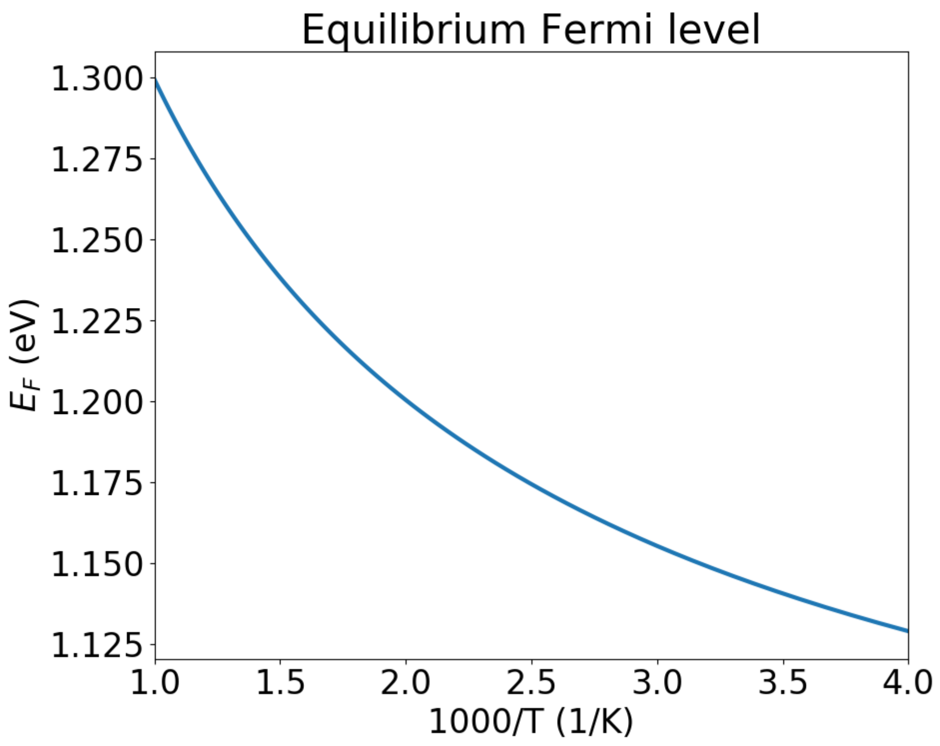
- `concentrations.equilibrium_fermi_level`: returns a Numpy array with `len(T_range)` elements, with the calculated equilibrium value of the electron chemical potential as a function of the input temperature. `concentrations.equilibrium_fermi_level - concentrations.vbm` can be used to obtain the Fermi level with respect to the valence band maximum.

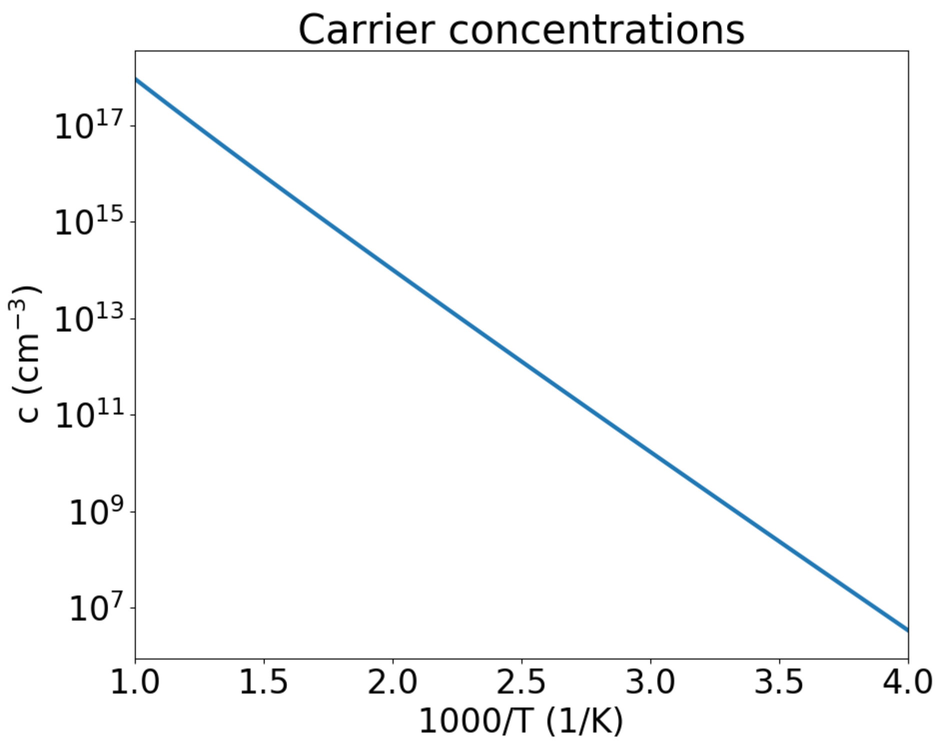
  The picture shows that intrinsic GaN is a *n*-type semiconductor and that the carrier concentration will increase as the temperature increases.

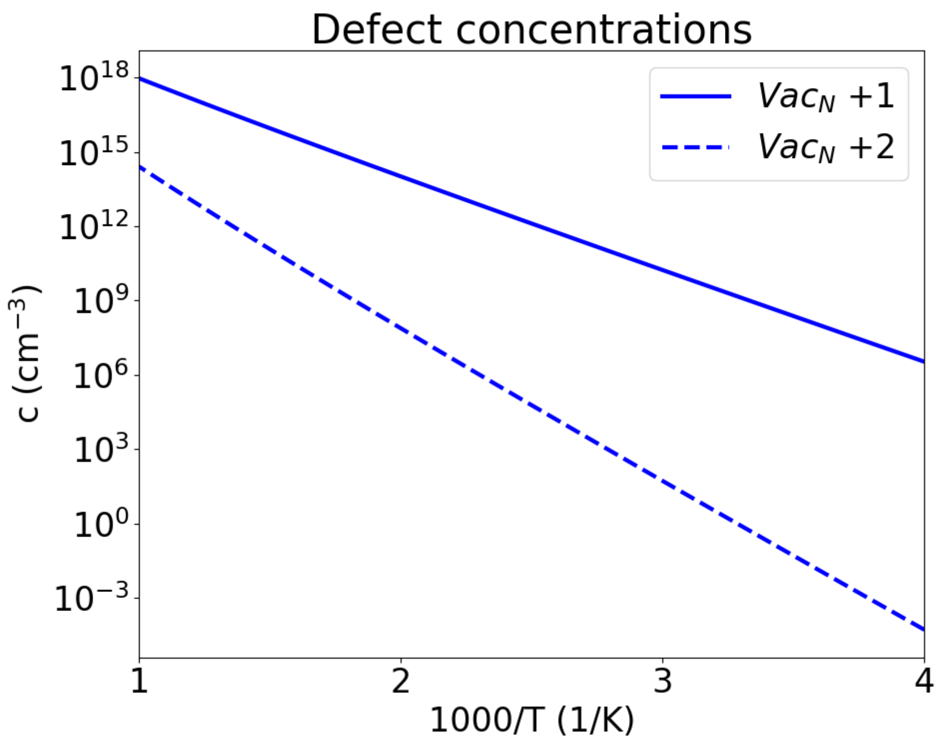
- `concentrations.equilibrium_carrier_concentrations` returns a Numpy array with the equilibrium carrier concentrations as a function of the temperature. The quantity returned is the difference between hole and electron concentrations. For intrinsic GaN the signs are negative, indicating that electrons are indeed the majority carriers. The plot below shows the absolute value of `concentrations.equilibrium_carrier_concentrations` as a function of $1000/T$. Note that the electron concentration will be largely overestimated as we have used the PBE band gap, which is much smaller than the experimental one. Opening the gap, *e. g.* by applying a scissor operator, might change the calculated concentrations by order of magnitudes.

- `concentrations.equilibrium_defect_concentrations` returns a dictionary, where each key is the name of the point defect, as used in `charge_states` and `form_energy_vbm`. The values are other dictionaries, where the keys are the defect charge state and the value a Numpy array with the defect concentrations as a function of the temperature.

  For example, the equilibrium defect concentrations of nitrogen vacancies, which we indicated using *Vac_N*, in the charge state +2 can be obtained from: `concentrations.equilibrium_defect_concentrations['Vac_N'][2]`.


Equilibrium Fermi level
E_F (eV)
1.300
1.275
1.250
1.225
1.200
1.175
1.150
1.125
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1000/T (1/K)


Carrier concentrations
c (cm^{-3})
10^{17}
10^{15}
10^{13}
10^{11}
10^{9}
10^{7}
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1000/T (1/K)



The picture shows that the majority carriers originate from the ionization of donor-type N vacancies, which have a low formation energy, as shown in Fig. **??** of section *Charge Transition Levels*.

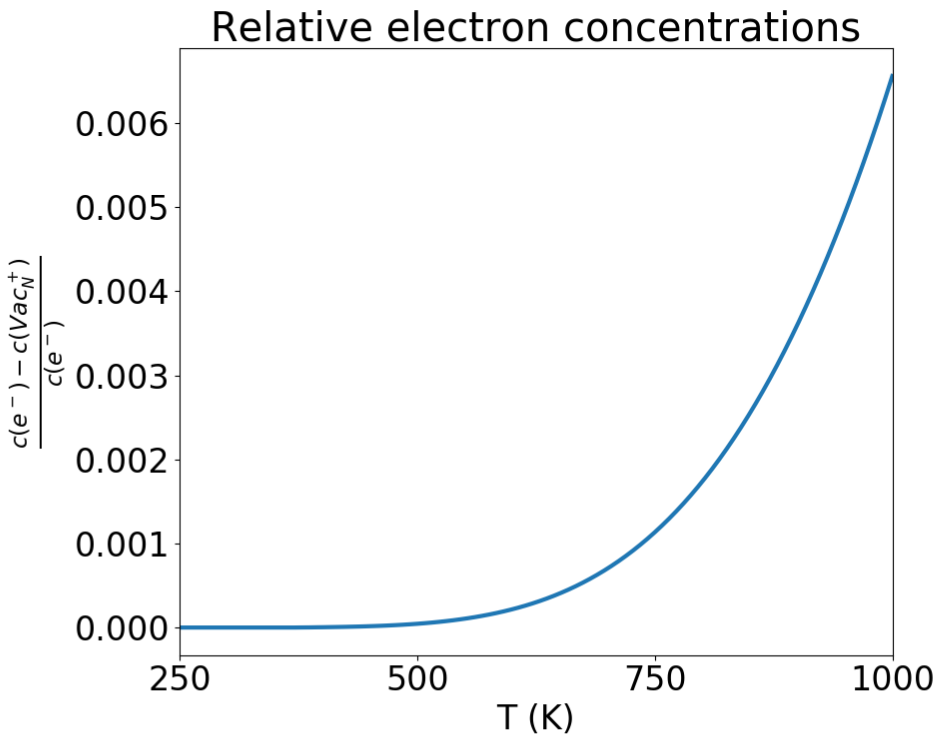
For convenience and for allowing further processing of defect concentrations, an `EquilibriumConcentrations` instance also collects the data in a *Pandas* `DataFrame` object, accesible through the attribute `defect_concentrations_dataframe`.

```
df = concentrations.defect_concentrations_dataframe
# the temperature is used for the row labels
df.loc[500] # Panda Series with formation energies at 500K
```

Equilibrium concentrations for free electrons and holes as a function of the temperature can be accessed through the attributes `equilibrium_electron_concentrations` and `equilibrium_holes_concentrations`, respectively. A *Pandas* `DataFrame` with the equilibrium carrier concentrations is given by the attribute `carrier_concentrations_dataframe`.

```
carriers_df = concentrations.carrier_concentrations_dataframe
# merge data frames for further processing
import pandas as pd
new_df = pd.concat([df, carriers_df], axis=1)
# show that free electron are almost entirely generated by single ionization␣
↪of Vac_N
data = new_df.loc[:, [('Vac_N', 1), 'electron']].values
plt.plot(T_range, (data[:, 1] - data[:, 0])/data[:, 1])
plt.show()
```

Through this last snippet, the curve reproduced in the below image is obtained, which shows that more than 99% of free electrons are due to the single ionization of nitrogen vacancies.



4.5.2 Using the class `DefectiveSystem`

Using the *usual directory structure* we can prepare an *`DefectiveSystem`* instance and access an *`EquilibriumConcentrations`* object from it.

The following code snippet will prepare an instance of the class *`DefectiveSystem`*, calculate the defect formation energies in the Ga-rich limit and produce an *`EquilibriumConcentrations`* instance, which can be accessed through the attribute `concentrations` of the *`DefectiveSystem`* instance. Such *`EquilibriumConcentrations`* instance can be used as shown in the previous section to obtain equilibrium defect and carriers concentrations as a function of the temperature.

```
import numpy as np
import os
import ase.io
from spinney.structures.defectivesystem import DefectiveSystem
from spinney.tools.formulas import count_elements
from spinney.io.vasp import extract_dos

path_defects = os.path.join('data', 'data_defects')
path_pristine = os.path.join('data', 'pristine', 'OUTCAR')
path_ga = os.path.join('data', 'Ga', 'OUTCAR')

ase_ga = ase.io.read(path_ga, format='vasp-out')

# Band alignment
vbm_offset = 0.85
vbm = 5.009256 - vbm_offset # align the VBM with the HSE band
e_gap = 1.713
```

(continues on next page)

(continued from previous page)

```
# dielectric tensor
e_rx = 5.888338 + 4.544304
e_rz = 6.074446 + 5.501630
e_r = [[e_rx, 0, 0], [0, e_rx, 0], [0, 0, e_rz]]

# get the chemical potential of Ga
ase_pristine = ase.io.read(path_pristine, format='vasp-out')
chem_pot_ga = ase_ga.get_total_energy()/ase_ga.get_number_of_atoms()
# get the chemical potential of N in the Ga-rich conditions
elements = count_elements(ase_pristine.get_chemical_formula())
chem_pot_n = ase_pristine.get_total_energy() - elements['Ga']*chem_pot_ga
chem_pot_n /= elements['N']

# get the density of states
dos = extract_dos('vasprun.xml')[0]
dos [:,0] -= vbm_offset

# site concentrations for point defects
volume = ase_pristine.get_volume()/ase_pristine.get_number_of_atoms()
volume *= 4
factor = 1e-8**3 * volume
factor = 1/factor
site_conc = {'Ga_N':4, 'N_Ga':4, 'Vac_N':4, 'Vac_Ga':4,
             'Ga_int':6, 'N_int':6, 'electron':36 , 'hole':36}
site_conc = {key:value*factor for key, value in site_conc.items()}

defective_system = DefectiveSystem(os.path.join('..', 'diagram', 'data'),
                                   'vasp')
defective_system.vbm = vbm
defective_system.dielectric_tensor = e_r
defective_system.chemical_potentials = {'Ga':chem_pot_ga, 'N':chem_pot_n}
defective_system.correction_scheme = 'ko'
# specific data for obtaining the EquilibriumDefectConcentration object
defective_system.gap_range = (vbm, vbm + e_gap)
defective_system.site_concentrations = site_conc
defective_system.temperature_range = np.linspace(250, 1000, 100)
defective_system.dos = dos
defective_system.calculate_energies(False)

# EquilibriumDefectConcentrations object
concentrations = defective_system.concentrations
```

CHAPTER FIVE

RELEASE NOTES

5.1 Version 0.9.a1

16 October 2020:

- Updated contacts emails.

5.2 Version 0.9.a1

21 July 2020:

- The package is now compatible also with Windows 10.

5.3 Version 0.9.a0

25 June 2020:

- Fixed some small bugs that gave problem in plotting the feasible region with newer versions of scipy.

5.4 Version 0.8.a3

27 February 2020:

- Added some internal helper functions.
- Fixed the sorting in the pandas series used for storing charge transition levels.

5.5 Version 0.8.a2

20 January 2020:

- Improved the method for plotting the intersection of the feasible region with the plane defined by constant chemical potentials.
- Added a complete case study in the tutorial.
- Small bug fixes.

5.6 Version 0.8.a1

7 January 2020:

- The class `DefectiveSystem` can be used for obtaining equilibrium defect concentrations.

5.7 Version 0.8.a0

18 December 2019:

- **Added compatibility with *ASE* version >= 3.18.0.**
- Added the class `DefectiveSystem` to manage defect formation energy calculations for a system with different point defects.

5.8 Version 0.7.a5

19 November 2019:

- First version released to the public.

CHAPTER

SIX

API REFERENCE

Spinney: A Python package for first-principles Point Defect calculations.

Spinney is a collection of Python modules aimed for the analysis and postprocessing of first-principles calculations of point defects in solids.

Spinney can assists with the major tasks necessary for the characterization of point defects in solids. The classes and functions that it implements can be divided into the following groups, which are related to the several steps necessary for processing the *ab-initio* calculations:

- **General high-level interface for point-defect calculations**

  `spinney.structures.pointdefect` Contains the *`PointDefect`* class which offers a convenient interface to calculate the properties of a point defect, such as its formation energy including finite-size corrections.

- **Determination of the possible values of equilibrium chemical potentials**

  `spinney.thermodynamics.chempots` Contains the *`Range`* class which is able to determine the possible equilibrium values of the chemical potentials given a set of competing phases.

  It also contains classes describing the chemical potentials of common gas phases, such as O_2, H_2, N_2, Cl_2 and F_2, as a function of temperature and pressure employing experimental data and empirical formulas.

- **Correction schemes for electrostatic finite-size effects in supercells**

  – ***`spinney.defects.kumagai`*** Implements the correction scheme proposed by Kumagai and Oba in Phys. Rev. B 89, 195205 (2014)

  – ***`spinney.defects.fnv`*** Implements the correction scheme proposed by Freysoldt, Neugebauer and Van de Walle in Phys. Rev. Lett. 102, 016402 (2009)

- **Calculation of equilibrium defect properties**

  – ***`spinney.defects.diagrams`*** Contains the *`spinney.defects.diagrams.Diagram`* class which allows to plot the defect formation energies as a function of the Fermi level and calculate charge transition levels.

  – ***`spinney.defects.concentration`*** Contains the `spinney.defects.concentration.EquilibriumConcentration` class that allows to calculate equilibrium properties, such as defects and carriers concentrations and the Fermi level position.

- **General-purpose tools**

  – ***`spinney.tools.formulas`*** Contains some helper functions useful for dealing with chemical formulas.

  – ***`spinney.tools.reactions`*** Contains helper functions useful for calculating reaction energies.

- **Support for first-principles codes** The **Spinney** package currently offers interfaces for these computer codes:
    - VASP: *spinney.io.vasp*
    - WIEN2k: *spinney.io.wien2k*

6.1 General high-level interface for point-defect calculations

- *spinney.structures.pointdefect*
- *spinney.structures.defectivesystem*

6.1.1 The `pointdefect` module

Module implementing a *PointDefect* class to store and process information concerning a system with point defects.

class spinney.structures.pointdefect.**DummyAseCalculator**(*atoms*)

Dummy calculator to be used when the calculations are done with a code not supported by ase. Use this dummy calculator only to pass custom Atoms objects to the *PointDefect* class. Any other use of ought to be avoided.

Parameters **atoms** (`ase.Atoms`) – the Atoms instance to be used with a *PointDefect* instance

class spinney.structures.pointdefect.**PointDefect**(*ase_atoms*)

PointDefect class.

It represents a 3D-periodic system containing one or more point defects.

Parameters **ase_atoms** (instance of `ase.Atoms`) – the Atoms object created from a completed point defect calculation.

Note: it is assumed that all the Atoms object have units of Angstrom for lengths and eV for energies. This is the default in `ase`.

my_name

an alphanumeric label for the instance

Type string, optional

defect_position

the scaled positions of the defect in the supercell

Type 1D numpy array

defect_charge

the charge state of the defect

Type float

pristine_system

the `ase.Atoms` describing the pristine system

Type `ase.Atoms` instance

parent_compounds

the keys are: `pristine`, `chemical_formula1`, . . . the value associated to `pristine` is the `ase.Atoms` instance representing the pristine system. `chemical_formula1` etc. are the chemical formulas of the other compounds involved in the creation of the point defect. These are added using *set_parent_elements()*. Their values are the `ase.Atoms` instances representing these compounds.

Excluding the pristine system, these data are not necessary, but can be used to test the chemical potential values to be used in the calculation of the defect formation energy.

For example, for an Oxygen vacancy in MgO, the keys of `parent_compounds` would be: 'pristine', 'O2', 'Mg2', which means the ase Atoms object for 'Mg2' contains 2 Mg atoms; i.e. it is the primitive HCP cell.

Type dict

`parent_elements`

each element is an instance of an `ase.Atoms` object representing the compound describing the standard state of the elements involved in the defective system.

For example, for an Oxygen vacancy in MgO, these would be the `ase.Atoms` instances for 'O2' and 'Mg2'

Type dict, optional

Examples

To initialize a `PointDefect` instance, it is only necessary to have an initialized `ase.Atoms` object with attached a calculator that supports the `Atoms.get_total_energy()` method.

Suppose that the output, obtained by a first-principle code, describing a defective system is saved in the file `output.fmt`. And that format `fmt` can be read by `ase.io.read()`. Then the following snippet can be used to initialize a `PointDefect` object representing the defective system.

```
>>> import ase.io
>>> defect = ase.io.read('output.fmt')
>>> pdf = PointDefect(defect)
```

`add_correction_scheme_data`(***kwargs*)

Add the extra data needed in order to calculate the defect formation energy with the choosen scheme

Parameters **`kwargs`** (*`dict`*) –

- For the correction schemes 'ko' and 'fnv':

  keys: `potential_pristine`, `potential_defective`

  the values are `numpy` arrays.

  – for 'ko':

    the arrays have to contain the value of the electrostatic potential at the ionic sites. The order has to be the same of the one used in the `ase.Atoms` objects employed in the initialization of the `PointDefect` instance and employed in `set_pristine_system()`

  – for 'fnv':

    the arrays have to contain the electrostatic potential on a 3D grid. The file has to match the supercell used to initialize the `PointDefect` instance.

- For 'fnv':

  `axis` the unit cell axis along which the electrostatic potential will be averaged.

  `defect_density` (optional) a 3D array with the charge density that can be used to model the defect-induced one. This will be used to fit the model charge to the defect-induced charge density.

x_comb (optional) a float between 0 and 1. Weight of the exponential function with respect to the Gaussian function in modeling the defect-induced charge density. Default x_comb = 0: the charge density is a pure Gaussian.

gamma (optional) a float. The parameter of the exponential function. Default value is 1.

beta (optional) a float. The parameter of the Gaussian function. Default value is 1.

shift_tol (optional) a float representing the tolerance to be used in order to locate the defect position along axis. Default value: $1e - 5 \times \mathrm{lengthcellparameterofaxis}$

e_tol (optional) a float, break condition for the iterative calculation of the correction energy. Value in Hartree. Default: 1e-6 Ha.

- For *ko*:

distance_tol (optional) rounding tolerance for comparing distances, in units of Angstrom. Default value is 5e-2 Angstroms.

e_tol (optional) a float, break condition for the iterative calculation of the correction energy. Value in eV. Default: 1e-6 eV.

calculate_finite_size_correction(*verbose=False*)
Calculate the energy correction for finite-size effects employing the method chosen with *set_finite_size_correction_scheme*.

Parameters **verbose** (*bool*) – If True, several details are returned as a dictionary. If False, only the correction energy is returned. This has to be added to the energy of the defective system.

get_defect_formation_energy(*include_corr=False*)
Returns the formation energy of the defective system.

Parameters **include_corr** (*bool*) – If True, the formation energy is already corrected for finite-size effects.

Returns **energy** – the defect formation energy

Return type float

set_Eg(*value*)

Parameters **value** (*float*) – The system band gap value.

set_chemical_potential_ranges(*ranges*)
For each element involved in the creation of the defective system, specify the minimum and maximum value that the chemical potential can have.

If *set_parent_elements()* was used, it will be checked that the input elements are the same.

Parameters **ranges** (*dictionary of 2-ples*) – For each element involved in the creation of the point defect, an element of ranges[element] is the 2-ple:

(minimum value atomic chemical potential, maximum value atomic chemical potential)

With 'atomic chemical potential' it is intended that the values are referred to a single atom, and not to the formula unit of the corresponding element used in *parent_elements*

set_chemical_potential_values(*chem_pots, force=False*)
Set the chemical potential values, for each species involved in forming the defective system, to be used in the calculation of the defect formation energy.

If these are not set, an exception will be raised. If chemical_potential_ranges is not None, it will be checked if the given chemical potentials are within these ranges; if not, an exception will be raised. In any case, the given chemical potential values will be tested againts the parent elements chemical potentials, if these are given. Use `force=True` to bypass these checks.

Parameters

- **`chem_pots`** (*`dict of floats`*) – each element is the chemical potential of one element (value give per atom) The keys must be the same used in *self.parent_elements*, if the parent elements were set.
- **`force`** (*`bool`*) – If True, the inserted values will be used for the calculations, even tough they are not physical valid values

`set_defect_charge`(*charge*)

Parameters **`charge`** (*`float`*) – The formal charge assigned to the point defect. It is assumed that only one localized charge is present in the supercell.

`set_defect_position`(*position*)

Set the fractional coordinates of the point defects in the system.

Parameters **`position`** (*`1D array`*) – the scaled position of the point defect with respect to `self.get_cell()`

`set_dielectric_tensor`(*value*)

Parameters **`value`** (*`2D array or float`*) – The value of the system dielectric tensor (or constant).

`set_fermi_level_value_from_vbm`(*value*)

Set the value of the Fermi level with respect to the system valence band maximum.

This value will be used to calculate the defect formation energy of charged defects.

Parameters **`value`** (*`float`*) –

`set_finite_size_correction_scheme`(*scheme*)

Set the correction scheme for finite-size effects in point defect calculations.

Parameters **`scheme`** (*`string`*) – The correction scheme to use. Possible values:

- 'ko' : Kumagai and Oba, PRB 89, 195205 (2014)
- 'fnv': Freysoldt, Neugebauer, and Van de Walle, Phys. Rev. Lett. 102, 016402 (2009)

`set_parent_elements`(*elements*)

Set the `ase.Atoms` objects representing the compounds, in their reference state, for the elements involved in the formation of the point defects.

Parameters **`elements`** (*`list or tuple`*) – each element of the sequence is an Atoms object. For example, if the defective system consists of the C vacancy- N_C complex in diamond, then: elements = (`ase.Atoms` for C (e.g. C diamond), `ase.Atoms` for N2)

`set_pristine_system`(*ase_pristine*)

Parameters **`ase_pristine`** (`ase.Atoms`) – The `ase.Atoms` object representing the pristine system.

`set_vbm`(*value*)

Parameters **`value`** (*`float`*) – The system valence band maximum, which determines the minimum value of the electron chemical potential.

6.1.2 The `defectivesystem` module

class spinney.structures.defectivesystem.**DefectiveSystem**(*data_path*, *calculator*)

Container class describing a system with point defects.

Parameters

- **data_path** (*string*) – path to the folder containing the results of the point defect calculations. It is expected a directory tree like this:

```
"data_path"
 ├── data_defects
 │    ├── "defect_name"
 │    │    ├── "charge_state"
 │    │    │    └── "files"
 │    │    ├── "charge_state"
 │    │    │    └── "files"
 │    │    ├── "charge_state"
 │    │    │    └── "files"
 │    │    └── "charge_state"
 │    │         └── "files"
 │    ├── "defect_name"
 ...
 └── pristine
      └── "files"
```

  – *defect_name* is a string describing the point defect.

  – *charge_state* must be the charge state of the considered defect.

  – *files* are the data needed for calculating defect formation energies. These depends on the calculator in use:

  – VASP: at least there should be the OUTCAR file.

  – WIEN2k: at least there should be the case.struct and case.scf files.

  For all calculators a file named *position.txt* containing the fractional coordinates of the defective site must be present.

- **calculator** (*string*) – the code used to calculate the data

vbm

the valence band maximum of the host material

Type float

dielectric_tensor

the dielectric tensor of the host material

Type float or 2D array

chemical_potentials

a dictionary whose keys are the parent elements forming the defective system and whose values are the chemical potentials to be used in the calculation of the defect formation energy

Type dict

correction_scheme

specifies the correction scheme for finite-size-effects to be used

Type str

data
collects the calculated formation energies for each processed point defect

Type pandas DataFrame object

point_defects
a list of `PointDefect` objects corresponding to the processed point defects

Type list

gap_range
a tuple containing the valence band maximum and conduction band minimum of the pristine system

Type tuple

extended_gap_range
a tuple containing the valence band maximum and conduction band minimum for an extetnded band gap of the pristine system. Used to initialize a Spinney Diagram object

Type tuple

diagram
an object representing the defect formation energies as a function of the electron chemical potential

Type Spinney Diagram object

calculate_energies(*verbose=True*)
Calculate defect formation energies :param verbose: if True, information about the process is printed :type verbose: bool

property defects_degeneracy_numbers
A dictionary o Represents the degeneracy for each type of defect in each of its charge states. The order has to match that of *charge_states[defect_type]*.

write_formation_energies(*out_file*)
Write the defect formation energies to a file in a format used by Spynney.

Parameters **out_file** (*str*) – name of the file used to write the energies

6.2 Determination of the possible values of equilibrium chemical potentials

- *spinney.thermodynamics.chempots*

6.2.1 The chempots module

Module with general tools and classes for handling chemical potential-related quantities.

The *Range* class allows to calculate valid chemical potential values given competing phases.

class spinney.thermodynamics.chempots.**ChlorineChemPot**(*energy_units='eV'*, *pressure_units='Pa'*)
Class for modelling the chemical potential of the Cl_2 gas molecule

Parameters

- **energy_units** (*string*) – the units to be used for the energy
- **pressure_units** (*string*) – the units to be used for the pressure

class spinney.thermodynamics.chempots.**FluorineChemPot**(*energy_units='eV'*, *pressure_units='Pa'*)

Class for modelling the chemical potential of the F_2 gas molecule

Parameters

- **energy_units** (*string*) – the units to be used for the energy
- **pressure_units** (*string*) – the units to be used for the pressure

class spinney.thermodynamics.chempots.**HydrogenChemPot**(*energy_units='eV'*, *pressure_units='Pa'*)

Class for modelling the chemical potential of the H_2 gas molecule

Parameters

- **energy_units** (*string*) – the units to be used for the energy
- **pressure_units** (*string*) – the units to be used for the pressure

class spinney.thermodynamics.chempots.**IdealGasChemPot**(*energy_units='eV'*, *pressure_units='Pa'*)

Class for modelling the chemical potential of an ideal gas molecule

Parameters

- **energy_units** (*string*) – the units to be used for the energy
- **pressure_units** (*string*) – the units to be used for the pressure

G_diff_Shomate_Eq(*T*)

Calculate the standard Gibbs free energy at temperature `T` with respect to the standard one at 0 K using Shomate equation.

Parameters T (*float*) – the temperature

Returns gibbs_energy – the standard Gibbs free energy at `T` in the units of `self.energy_units`

Return type float

get_ideal_gas_chemical_potential_Shomate(*mu_standard*, *partial_pressure*, *T*)

Given the standard chemical potential at 0K, e.g. calculated with DFT, returns the value at given temperature obtained using the Shomate equation and the ideal gas formulas.

$$\mu(T,p) = h(0,p^\circ) + [g(T,p^\circ) - g(0,p^\circ)] + k_B T \ln\left(\frac{p}{p^\circ}\right)$$

Parameters

- **mu_standard** (*float*) – standard-state chemical potential of the molecule at 0K
- **partial_pressure** (*array or float*) – the partial pressure
- **T** (*array or float*) – the gas temperature

Returns

chem_pot – The chemical potentials as a 2D numpy array of shape (len(T), len(partial_pressure)) if both T and partial_pressure are arrays; otherwise a 1D numpy array of length len(partial_pressure)

if partial_pressure is an array and T a float; otherwise a 2D array of shape (len(T), 1) if partial_pressure is a float and T an array; finally if both T and partial_pressure are float, the result is a float

Return type numpy array/float

class `spinney.thermodynamics.chempots.`**`NitrogenChemPot`**(*energy_units='eV'*, *pressure_units='Pa'*)

Class for modelling the chemical potential of the N_2 gas molecule

Parameters

- **`energy_units`** (*`string`*) – the units to be used for the energy
- **`pressure_units`** (*`string`*) – the units to be used for the pressure

class `spinney.thermodynamics.chempots.`**`OxygenChemPot`**(*energy_units='eV'*, *pressure_units='Pa'*)

Class for modelling the chemical potential of the O_2 gas molecule

Parameters

- **`energy_units`** (*`string`*) – the units to be used for the energy
- **`pressure_units`** (*`string`*) – the units to be used for the pressure

class `spinney.thermodynamics.chempots.`**`Range`**(*coeff_equalities*, *const_equalities*, *coeff_inequalities*, *const_inequalities*, *bounds*)

Class for finding the allowed ranges for the chemical potentials of given elements given the competing phases.

Parameters

- **`coeff_equalities`** (*`2D tuples`*) – the coefficients of the linear equalities. For each equality there should be an array with the corresponding coefficients. **Each element in the tuple must contain the same number of elements**. If a variable appears at least once in the constraint equations, it must be set to 0 in all the other equations.
- **`const_equalities`** (*`1D tuple`*) – the constant values of the linear equalities. const_equalities[i] has to be the constant value of the linear equation with coefficients coeff_equalities[i].

  Example:

  We have the linear constraints given by these equations:

  $$ax + by + cz = d$$
  $$a'x + b'y + c'z = d'$$
  $$a''x + b''y = d''$$

  then: coeff_equalities = ((a,b,c), (a', b', c'), (a", b", 0))

  const_equalities = (d, d', d")

  > **Warning:** If a variable is present **anywhere** in either the equality or inequality constraints, then it must be explicitly given even if its value is zero (the coefficient of z in the last equation of the previous example).

  coeff_inequalities and const_inequalities are analogously defined, but for the inequality conditions.

- **`bounds`** (*`tuple of tuples`*) – the bounds of the independent variables. If not present, they will be set to (None, None) for each independent variable, which will be treated as -inf and +inf

`number_of_variables`

number of chemical potentials

Type int

variables_extrema

shape = (number elements, 2) for each element, returns the minimum and maximum possible values for its chemical potential

Type 2D numpy array

variables_extrema_2d

shape = (2, 2) for each element, returns the minimum and maximum possible values for its chemical potential. Calculated after an intersection with a plane has been performed.

Type 2D numpy array

mu_labels

the symbols labeling the elements for which we caculate the chemical potentials

Type tuple

Notes

ALL INEQUALITIES SHOULD BE IN THE FORM:

coeff_inequalities * x <= const_inequalities

Examples

Suppose that we want to find the chemical potential extrema for Ti and O in rutile TiO2.

We have the following constraints:

$$
\begin{aligned}
\mu_{\mathrm{Ti}} + 2\mu_{\mathrm{O}} &= \mu_{\mathrm{TiO_2}} \\
\mu_{\mathrm{Ti}} + \mu_{\mathrm{O}} &\leq \mu_{\mathrm{TiO}} \\
2\mu_{\mathrm{Ti}} + 3\mu_{\mathrm{O}} &\leq \mu_{\mathrm{Ti_2O_3}} \\
\mu_{\mathrm{Ti}} &\leq \mu_{\mathrm{Ti}}(HCP) \\
\mu_{\mathrm{O}} &\leq \frac{1}{2}\mu_{\mathrm{O_2}}
\end{aligned}
$$

So, we will have to type:

```
>>> coeff_equalities = ((1,2))
>>> const_equalities = (mu_TiO2, )
>>> coeff_inequalities = ((1, 1), (2, 3))
>>> const_inequalities = (mu_TiO, mu_Ti2O3)
>>> bounds = ((None, mu_Ti(HCP)), (None, 1/2 mu_O2))
```

check_value_variables(*variables*, *eq_tol=1e-06*)

Checks whether a set of variables is within the feasible region.

Parameters

- **variables** (*array*) – the length is given by `no_variables`: the number of elements in the system
- **eq_tol** (*float*) – tolerance on equality conditions

Returns

result – a booleand and a numpy array of shape `self.no_variables` times the number of constraints

- First element:

True, if `variables` is within the feasible region, False, otherwise.

- Second element:

  Array of shape: `no_variables` times the number of constraints where the number of contraints is the number of equalities plus inequalities plus bounds conditions.

  This array is a 2D array whose rows represent the variables and whose columns represents the various constraints, the order is:

  equalities, inequalities and bound down, bound up

  in the same order given by the user at initialization. The values of such matrices are bool.

  Example:

  Suppose we have two variables, one equality and two inequality and 2 bounds constraints. If both variables satisfy the equality, first inequality and bounds. But variable 2 does not satisfy the second inequality; the funcion will return a np.ndarray `result` of shape (2,5):

```
result = [[True, True, True, True, True],
          [True, True, False, True, True]]
```

Return type tuple

plot_feasible_region_on_plane(*plane_axes*, *plane_values=None*, *save_plot=False*, *tol=1e-06*, ***kwargs*)

Plot the feasibile region intersection with the plane specified by `plane_axes`.

Parameters

- **plane_axes** (*1D array of length 2*) – the array specifies the indices of the variables which will form the axes of the intersection plane.
- **plane_values** (*1D array of length self.number_of_variables - 2*) – Constant value specifying the plane.
- **save_plot** (*bool*) – if True, the plot will be saved
- **tol** (*float*) – a numerical tolerance for determining the points giving the feasible region
- **kwargs** (*dict, optional*) – optional key:value pairs for plotting the diagram

Returns

Return type the figure instance of the plot.

Example

For example, if we are considering a ternary system A, B, C, and we want to intersect the feasible region with the plane B = b, using the axes given by variables A and B, then plane_axes = [0, 2], plane_values = [b]. This specifies the plane $B = b$.

By default *plane_values* is filled with zeros.

set_chemical_potential_labels(*labels*)

Set the name of the independent variables.

Parameters labels (*list*) – the names of the independent variables, one should use the same order used in `self.coeff_*`

set_color_map(*cmap*)

Set the color map to be used in the plot

Parameters **cmap** (*matplotlib.cm class*) – one of the color maps offered by matplotlib. By default Set1 is used, which is fine when less than 10 compounds are considered

set_compound_dict(*compounds_dict*)

Set a dictionary which contains the compound names relative to the equality, inequality and bound conditions.

Valid names are: formula + _ + description.

description is any string that described the compound

Note: Note that the validity of the name format is not checked internally and should be done by the user.

compounds_dict : dict

```
{'equality' : [compoundA1, ...],
 'inequality' : [compound1, ...],
 'bound' : [compoundA, ...]}
```

the order of the elements in the dictionary values has to be the same of the order used in the relative `const_equalities`, `const_inequalities` and `bounds` lists

spinney.thermodynamics.chempots.**find_partial_pressure_given_mu**(*mu, mu_standard, T, pa=1e-12, pb=1000000000.0, energy_units='eV', pressure_units='Pa'*)

Finds the partial pressure required to change the chemical potential of the ideal gas from `mu_standard` to `mu` at temperature `T`.

Parameters

- **mu** (*float*) – the desired value of the chemical potential
- **mu_standard** (*float*) – the standard state chemical potential of the ideal gas
- **T** (*float*) – the temperature of interest
- **pa** (*float*) – the lower bound for the partial pressure
- **pb** (*float*) – the upperbound for the partial pressure
- **energy_units** (*string*) – the unit of energy used for the value of `mu` and `mu_standard`
- **pressure_units** (*string*) – the unit of pressure

Returns **p** – the partial pressure giving `mu`

Return type float

spinney.thermodynamics.chempots.**get_chem_pots_conditions**(*file, order, equality_compounds, delimiter=None*)

Reads `file` and returns the data needed to initialize a *Range* class as well the dictionary describing the various compounds.

Parameters

- **file** (*str*) – the location of the file with the required information The format of this file must be:

  `Label_compound(delimiter)energy`

  Label_compound is a string, whose different entries must be separated by an underscore.

  delimiter is a string specifying the delimiter used to separate Label_compound from energy

  energy is a float representing the (formation) energy of the compound

- **order** (*array*) – ordered list of the chemical symbols to be used in specifying the equality and inequality conditions
- **equality_compounds** (*array of strings*) – the compounds in file corresponding to the equality conditions

Returns

result – The first 5 elements are needed to initialize a *Range* instance.

The last element is the argument for the method *Range.set_compound_dict()*

Return type tuple of 6 elements

`spinney.thermodynamics.chempots.`**`get_conditions_from_file`**(*file*, *order*, *delimiter=None*)

Reads and parses a file and returns a triplet with the elements, coefficients, and the constants. These values can be used for the inequality or equality conditions in *Range*.

Lines starting with '#' will be skipped.

Parameters

- **file** (*str*) – the file that has to be read. The format has to be: `Compound_formula_unit(delimiter)Energy_of_the_compound_per_fu`
- **Note** – **All inequalities are indented as** "<=": pay attention to prepare the input file in this way
- **order** (*array*) – the name of the elements in the order that has to be used for the coefficients. Eg, for the Mn-O system, `order` can be ['Mn', 'O'] or ['O', 'Mn']
- **delimiter** (*str*) – the symbol used to separate compound names from energy values.

Returns

compound_dict – the keys are the compound identifiers as read from the file. The elements of the tuple are:

- the formula unit of the compound
- the stoichiometric coefficients for the elements in the compound
- the (formation) energy per atom of the compound

Return type dictionary of 3-ple:

`spinney.thermodynamics.chempots.`**`ideal_gas_chemical_potential`**(*mu_standard*, *partial_pressure*, *T*, *energy_units='eV'*, *pressure_units='Pa'*)

Temperature and pressure-dependent chemical potential of an ideal gas

$$\mu = \mu^{\circ} + k_B T \ln\left(\frac{p}{p^{\circ}}\right)$$

Parameters

- **mu_standard** (*float*) – the gas molecule chemical potential at standard pressure
- **partial_pressure** (*array or float*) – the gas partial pressure with respect to the standard pressure
- **T** (*array or float*) – the temperature range of interest
- **energy_units** (*string*) – the units in which energy is expressed
- **pressure_unit** (*string*) – the units in which pressure is expressed

Returns chem_pots – The chemical potentials as a 2D numpy array of shape (len(T), len(partial_pressure)) if both T and partial_pressure are arrays; otherwise a 1D numpy array of length len(partial_pressure) if partial_pressure is an array and T a float; otherwise a 2D array of shape (len(T), 1) if partial_pressure is a float and T an array; finally if both T and partial_pressure are float, the result is a float

Return type array/float

`spinney.thermodynamics.chempots.`**`parse_mu_from_string`**(*string*, *delimiter=None*)

Reads and parses a string and returns a dictionary with the elements, coefficients, and the constants of the equality/inequality conditions.

Parameters

- **string** (*str*) –

  the string that has to be parsed. The format has to be:
  ```
  Formula_unit(delimiter)Energy_of_the_compound_per_fu
  # comments
  ```
  Example: SrTiO3 -12.3

  will return:
  ```
  result = {'Sr':1/5, 'Ti':1/5, 'O':3/5, 'energy':-12.3}
  ```
- **delimiter** (*str*) – the symbol used to separate compound names from energy values

Returns **result** – as in the example above

Return type dict

6.3 Correction schemes for electrostatic finite-size effects in supercells

- *spinney.defects.kumagai*
- *spinney.defects.fnv*

6.3.1 The `kumagai` module

Implementation of the correction scheme for charged point defects of Kumagai and Oba:

Y. Kumagai and F. Oba, PRB 89, 195205 (2014)

class `spinney.defects.kumagai.`**`KumagaiCorr`**(*cell*, *positions_defective*, *positions_pristine*, *defect_position*, *defect_formal_charge*, *dielectric_constant*, *dft_core_potential_def*, *dft_core_potential_prist*, *direct_cutoff=10*, *reciprocal_cutoff=1*, *alpha=None*, *length_units='Angstrom'*, *energy_units='eV'*, *tol_en=1e-06*, *min_steps=2*, *tol_dist=0.01*)

Implementation of the Kumagai correction scheme of: PRB 89, 195205 (2014)

Parameters

- **`cell`** (*2D numpy array*) – cell parameters defective supercell
- **`positions_defective/positions_pristine`** (*2D numpy array*) – fractional coordinates of the atoms in the **DEFECTIVE** and **PRISTINE** supercells of same size and shape.
- **`defect_position`** (*1D numpy array*) – fractional coordinates of the point defect
- **`defect_formal_charge`** (*float*) – charge of the point defect
- **`dielectric_constant`** (*2D numpy array*) – the dielectric tensor
- **`dft_core_potential_def/dft_core_potential_prist`** (*1D numpy array*) – potential at the ionic sites calculated by first-principles for **DEFECTIVE** and **PRISTINE** systems, respectively. Same atomic ordering as in the corresponding `positions_*` arrays
- **`direct_cutoff/reciprocal_cutoff/alpha`** – see the `spinney.defects.madelung.Ewald` class
- **`length_units/energy_units`** (*strings*) – the unit of length and energy used for `cell` and `dft_core_potential_*`
- **`tol_dist`** (*float*) – rounding tolerance for comparing distances, in units of `length_units`

`atomic_mapping`

element [0] has the indexes of the pristine system atoms with corresponding atoms in the defective system

element [1] has the indexes of the defective system mapped by `map[0]`. Where `map` is returned by *`map_atoms_pristine_defect()`*

Type 2-ple of 1D arrays

`atoms_in_sampling_region`

the first element contains the indices of the atoms of the pristine system in the sampling region, the second element contains the corresponding atoms of the defective system

Type 2-ple

`difference_potential_vs_distance`

element [0] is an array containing the distances from the point defects where the electrostatic core potential has been sampled

element [1] is the difference of the defective and pristine core potentials at the point whose distance from the defect is in element [0]

Type 2-ple of 1D arrays

`difference_potential_vs_distance_sampling_region`
as above, but only the sampling region is considered

Type 2-ple of 1D arrays

`alignment_potential_vs_distance_sampling_region`
element [0] is an array containing the distances from the defect of the sites within the sampling region.

element [1] is the corresponding alignment potential: site potential defect - site potential pristine - ewald potential

Type 2-ple of 1D arrays

`ewald_potential_vs_distance_sampling_region`
the ewald potential calculated at the sites in the sampling region. Structure analogous to *`alignment_potential_vs_distance_sampling_region`*

Type 2-ple of 1D arrays

`grouped_atom_by_distance`
the first element of the tuple reports the distances from the defect The second element is another 2-ple; each element of this tuple contains a list of lists. Each inner list contains the atomic index of the atoms that are at a given distance (within `tol_dist`) from the defective site for the pristine and defective supercell.

Type 2-ple

`property difference_potential_vs_distance`
Pot_def - Pot_pristine

`spinney.defects.kumagai.`**`kumagai_sampling_region`**(*cell*, *atom_coordinates*, *defect_position*)
Given the scaled coordinates of atoms in the supercell, returns the indices of the atoms belonging to the sampling region as defined in: Y. Kumagai and F. Oba, PRB 89, 195205 (2014)

Parameters

- **`cell`** (*`2D numpy array`*) – each row represents the Cartesian coordinates of the primitive vectors
- **`atom_coordinates`** (*`2D numpy array`*) – each row represents the fractional coordinates of the atoms in the supercell
- **`defect_position`** (*`1D numpa array`*) – the scaled coordinates of the defect atom in the supercell. It will be used to center the supercell

Returns

in_region, sphere_radius – `in_region` is a list with the indexes of the atoms in the sampling region.

`sphere_radius` is a float and it is the radius of the sphere inscribed in the Wigner-Seitz cell.

Return type tuple

`spinney.defects.kumagai.`**`map_atoms_pristine_defect`**(*cell*, *scaled_pos_prist*, *scaled_pos_defect*, *dist_tol=0.01*)
Return a mapping between the indexes of the atoms in the pristine supercell and those of the atoms in the defective supercell. Both supercells have to be the same.

Parameters

- **`cell`** (*`2D numpy array`*) – the supercell

- **scaled_pos_prist** (*2D numpy array*) – fractional atomic positions of the pristine system
- **scaled_pos_defect** (*2D numpy array*) – fractional positions of the defective system

> **Warning:** It is assumed that both pristine and defective system have the same cell: `cell`

- **dist_tol** (*float/array of 3 elements*) – tolerance value for which 2 distances are considered equal. If float, the value will be used to create an array with 3 elements. Each elements represent the tolerance along the corresponding cell parameter. Default value: 1% of the corresponding cell parameter.

Returns

map – `map[0]` has the indexes of the pristine system with correspective atoms in the defective system

`map[1]`has the indexes of the defective system atoms mapped by :data:`map[0]`

Return type 2-ple of 1D arrays

6.3.2 The `fnv` module

Standalone implementation of the correction scheme for charged defects proposed by Freysoldt, Neugebauer and Van de Walle.

Phys. Rev. Lett. 102, 016402 (2009)

Internally, the code use atomic units of Hartree and Bohr, as implemented in Freysoldt et al. papers.

The input values have to be inserted in units of eV and Angstrom and the final values will also be converted to these units.

class spinney.defects.fnv.**FChargeDistribution**(*x=0*, *gamma=1*, *beta=1*)

Gaussian + exponential tail radial charge distribution employed by Freysoldt et. al., Phys. Status Solidi B 248, 5 (2011)

$$q(r) = q\left(xN_1e^{-r/\gamma} + (1-x)N_2e^{-r^2/\beta^2}\right)$$
$$N_1 = \frac{1}{8\pi\gamma^3}$$
$$N_2 = \frac{1}{(\sqrt{\pi}\beta)^3}$$

`x` and `gamma` may be found from looking at the defect wave function. The value of `beta` is not so important as long as the defect remains localized. A Gaussian function usually performs well enough.

Parameters

- **x** (*float*) –
- **gamma** (*float*) – units of Angstrom
- **beta** (*float*) – units of Angstrom

Notes

For localized states, $x = 0$; for delocalized ones, usually x is around 0.54-0.6 in semiconductors.

Direct()
The normalized charge distribution in real space

Fourier_transform()
The Fourier trasform of `Direct()`

Fourier_transform_gto0()
The second derivative of the charge distribution with respect to g. It approximates the distribution for $g \to 0$:

```
q(g-->0) ~ q.Fourier_transform(0) + 1/2 q.Fourier_transform_gto0()
```

The term is used for getting the potential alignment V_0 of equation 17.

classmethod get_model_D()
Model function to be used for fitting

class spinney.defects.fnv.**FCorrection**(*supercell*, *q*, *charge_distribution*, *pristine_pot*, *defective_pot*, *dielectric_constant*, *def_position*, *axis_average*, *cutoff=500*, *tolerance=1e-06*, *shift_tol=1e-05*)
Implementation of Freysoldt et al. correction.

Note: the code assumes that the units are Angstrom for lengths and eV for energies!

Parameters

- **supercell** (*2D array*) – each rows represents the cartesian coordinates of the vectors describing the supercell
- **q** (*int*) – charge state of the defect
- **pristine_pot** (*3D numpy array*) – this array must contain the total electrostatic potential calculated on 3D grid for the pristine system. The shape of the array is: (Na, Nb, Nc)

  where Nx is the number of grid points along the cell parameter x
- **defective_pot** (*3D numpy array*) – as *pristine_pot* but for the defective system
- **charge_distribution** (*FChargeDistribution* instance) – the model charge distribution used for the defect-induced charge density
- **dielectric_constant** (*float*) – dielectric constant of the bulk system
- **def_position** (*array*) – defect position in the defective supercell in fractional coordinates with respect to *supercell*
- **axis_average** (*int*) – axis which will be used to perform the plane-average of the electrostatic potential:
  - 0 for a
  - 1 for b
  - 2 for c
- **cutoff** (*float*) – cutoff for reciprocal space vectors, **IN** eV

- **tolerance** (*float*) – tolerance threshold used in calculating the energy terms
- **shift_tol** (*float*) – tolerance in findind the defect position on the 3D grid

generate_reciprocal_space_grid(*cutoff*)
Generates a regular mesh of reciprocal space points given a cutoff energy.

Parameters **cutoff** (*float*) – maximum kinetic energy, in Hartree, of the k-point

Returns **grid** – the reciprocal lattice vectors in Cartesian coordinates in units of Bohr

Return type list

get_E_lat(*all_contributions=False*)
Gets the $E^{lat}[q]$ part of the correction scheme. From equation (8) in Freysoldt et al. PSS B 248, 5 (2011).

Parameters **all_contributions** (*bool*) – if True, the function returns the various contributions: E_lat, E_lat1, E_lat2; otherwise just the total E_lat

Returns **energy_values** – The values **IN** eV.

Return type float/tuple

get_correction_energy()
Returns the correction energy for finite-size effects to add to the calculated DFT energy.

Returns **E_corr** – The correction term

Return type float

get_potential_alignment()
Get the potential alignment term. I.e. the $q\Delta V$ term in equation (13) of PSS B 248, 5 (2011)

class spinney.defects.fnv.**FFitChargeDensity**(*cell*, *charge_model*, *density_proj_def*, *defect_position*, *axis_average*, *tol=1e-05*)
Fits the model charge density to the DFT defective charge density averaged along an axis.

Parameters

- **cell** (*2D numpy array*) – each row reoresents the Cartesian coordinates of one cell parameter of the crystal unit cell
- **charge_model** (*class*) – model of the charge density
- **density_proj_def** (*3D numpy array*) – the defect-induced charge density on a 3D grid. In case of spin-polarized calculations, it is the net charge density. The shape of the array is: (Na, Nb, Nc)

  where Nx is the number of grid points along the cell parameter x
- **defect_position** (*1D array*) – the position of the defect with respect to the cell parameters (i.e. the fractional coordinates)
- **axis_average** (*int*) – the axis along which perform the plane-average
- **tol** (*float*) – numerical spatial tolerance

fit_model()
Fit the model charge density

class spinney.defects.fnv.**FPlotterPot**(*coordinates*, *av_locpot_diff*, *av_lr_pot*, *av_sr_pot*, *pot_alignment*, *axis_name*)
Helper class for plotting the potentials.

Parameters

- **coordinates** (*array*) – coordinates to plot the potentials along an axis, can be fractional or Cartesian
- **av_locpot_diff** (*array*) – difference DFT potential between defective and pristine system plane-averaged along an axis
- **av_lr_pot** (*array*) – the analytical long-range potential, plane-averaged over an axis
- **av_sr_pot** (*array*) – the short-range potential
- **pot_alignment** (*float*) – the final aligment of the potentials far from the defect
- **axis_name** (*string*) – the name of the axis along which the potentials were calculated

plot (*title=''*)
Plot the potentials

Parameters **title** (*string, optional*) – the plot title

spinney.defects.fnv.**plane_average_potential** (*locpot, axis*)
Makes the plane-average of the electrostatic potential along the chosen axis.

Parameters

- **locpot** (*3D array*) – the local electrostatic potential on a 3D mesh
- **axis** (*int*) – 0 for a 1 for b 2 for c where a,b,c are the cell parameters along which the 3D grid is defined

Returns **1D numpy array**

Return type the plane-averaged potential along the chosen axis

6.4 Calculation of equilibrium defect properties

- *spinney.defects.diagrams*
- *spinney.defects.concentration*

6.4.1 The `diagrams` module

Implements the creation of the diagrams reporting the formation energy of point defects as a function of the electron chemical potential.

class spinney.defects.diagrams.**Diagram** (*defects_dictionary, gap_range, extended_gap_range=None, electron_mu=None*)
A diagram represent the formation energies of various point defects, in various charge states, as a function of the electron chemical potential, whose value ranges from the valence band maximum to the conduction band minimum of the host material.

This class is basically a composition of PointDefectLines instances with some tools for plotting the final diagram.

Parameters

- **defects_dictionary** (*dict of dicts*) – for each point defect named "defect", the value is a dictionary in the form: {charge_state1 : formation_energy_at_VBM, . . . }

  Example:

  For vacancy and N center in diamond:

```
>>> defects_dictionary = {
        '$V$' : {-1 : value1, 0 : value2, 1 : value3},
        r'$N_C$' : {0 : value4}
        }
```

The value of VBM must be consistent with gap_range[0]

- **gap_range** (*array of 2 elements*) – the band gap range. The first value must be consistent with the valence band maximum (VBM) used to calculate the defect formation energy in `defect_dictionary`
- **extended_gap_range** (*array of 2 elements*) – an extended range for the band gap. This value can be used for example when `gap_range` is the underestimated DFT band gap and we want to plot the defect formation energy lines on a wider gap. In this case, the difference between the extended region and the original region will be plotted in gray.
- **electrom_mu** (*float*) – pinned value of the electron chemical potential wrt the valence band maximum (`gap_range[0]`)

labels
{defect_name : defect_label, … } where defect_name is one of the top keys if `defects_dictionary`

Type dict

defects
{defect_name : Line instance representing that defect type, … }

Type dict

plot(***kwargs*)
Plot the diagram.

Parameters **kwargs** (*dict*) – optional key:values pair for plotting the diagram

write_transition_levels(*file_name*)
Writes the charge transition levels of the system on a txt file.

Parameters **file_name** (*string*) – the name of the file where the transition levels will be written

class spinney.defects.diagrams.**Line**(*m, q, x0*)
A simple class describing the equation of a line:

$$y = m(x - x_0) + q$$

where q is the intercept with the y = x0 axis

class spinney.defects.diagrams.**PointDefectLines**(*defect_name, e_form_min_dict, gap_range, extended_gap_range=None*)
This class holds the information to represent a line on the formation energy diagrams of point defects.

Parameters

- **defect_name** (*string*) – the name of the defect
- **e_form_min_dict** (*dict*) – {charge_state1 : formation_energy1_at_VBM, … }

  charge_state1 is an integer describing the defect charge state; formation_energy1_at_VBM is a floating-point number indicating the formation energy of that defect for an electron chemical potential equal to the valence band maximum (VBM) of the host material. This VBM value must be stored in `gap_range`

- **gap_range** (*array of 2 elements*) – (valence_band_maximum, conduction_band_minimum) representing the allowed range for the electron chemical potential. Note that the VBM to which *e_form_dict* refers to is gap_range[0]
- **extended_gap_range** (*array of 2 elements*) – an extended range for the band gap. This value can be used for example when *gap_range* is the underestimated DFT band gap and we want to evaluate the defect formation energy lines on a different gap.

lines

{charge_state1 : line1, . . . } for each defect charge state of the instance. line1 is a Line instance.

Type dict

intersections

For each charge state of the defect, one has the dictionary containing the coordinates of the intersection point between the line corresponding to that charge state and the lines of all other charge states and band edges as well.

Type dict of dicts

lines_limits

{defect_charge_state1 : [y_0, y_1], . . . } y_0 and y_1 are two numpy arrays with two elements each:

the values of the electron chemical potential and those of the defect formation energy corresponding to the two points between which the formation energy of defect_charge_state1 is the lowest one among all other defect charge states.

Type defaultdict

transition_levels

Returns the transition levels among the possible charge states as a dictionary:

```
{charge_state1 : {charge_state2 : transition_level,
                  charge_state3 : transition_level, ...}, ...}
```

transition_level is the electrom chemical potential value at the transition level between charge states. For a complete list of intersection between all defect lines, use `self.intersections`

Type dict

property intersections

Returns a dictionary of dictionaries. For each charge state of the defect, one has the dictionary containing the coordinates of the intersection point between the line corresponding to that charge state and all other charge states and band edges as well.

property lines_limits

The attribute *_lines_limits* is a defaultdict:

{defect_charge_state1 : [y_0, y_1], . . . } y_0 and y_1 are two numpy arrays with two elements each: the values of the electron chemical potential and those of the defect formation energy corresponding to the two points between which the formation energy of *defect_charge_state1* is the lowest one among all other defect charge states.

This is a subset of *self._intersections*

plot_lines(*ax='auto'*, ***kwargs*)

Given an existing axes, plots the defect formation energy lines

Parameters

- **`ax`** (`matplotlib.axes.Axes` instance) – the axes where the lines should be plotted
- **`kwargs`** (*`dict`*) – additional key:value pairs taken by matplotlib.pyplot.plot

`property transition_levels`

Returns the transition levels among the possible charge states as a dictionary:

{charge_state1 [{charge_state2][transition_level,] charge_state3 : transition_level, . . . }, . . . }

transition_level is the electrom chemical potential value at the transition level between charge states. For a complete list of intersection between all defect lines, use `self.intersections`

`spinney.defects.diagrams.`**`extract_formation_energies_from_file`**(*file_name*)

Read a file with the format:

```
# arbitrary number of comment lines
defect_name    defect_charge_state   defect_formation_energy
```

Parameters **`file_name`** (*`string`*) – the file with the data

Returns diagram_dict – dictionary containing the information necessary to initialize an *`Diagram`* instance.

Return type defaultdict(dict)

`spinney.defects.diagrams.`**`set_latex_globally`**()

Call this function to use latex synthax globally in matplotlip. Note that now every string used through matplotlib has to respect latex synthax.

6.4.2 The `concentration` module

Functions and classes for calculating defect and carrier concentrations and related quantities from their formation energies.

`class` `spinney.defects.concentration.`**`Carrier`**(*dos*, *vbm*, *cbm*, *mu*, *T*, *dos_down=None*, *energy_units='eV'*)

Object describing a free carrier in a semiconductor.

Parameters

- **`dos`** (*`2D array`*) – the first column stores the 1-electron energy levels, the second column stores the corresponding DOS. The energy must be sorted by increasing values.
- **`vbm`** (*`float`*) – the energy level corresponding to the valence band maximum
- **`cbm`** (*`float`*) – the energy level corresponding to the conduction band minimum
- **`mu`** (*`float`*) – the electron chemical potential
- **`T`** (*`float or array`*) – the temperature range where the concentration will be calculated
- **`dos_down`** (*`2D array`*) – If left to None, then the spin-down electrons are considered to have the same dos as `dos`.

  Note:

  in case the spin down dos is reported using negative numbers, you need to flip the sign before using them in this function.
- **`units`** (*`str`*) – the units of energy

mu
electron chemical potential

Type float

T
the temperature for which the Fermi Dirac distribution is calculated

Type float or 1D numpy array

class spinney.defects.concentration.**ConductionElectron**(*dos*, *vbm*, *cbm*, *mu*, *T*, *dos_down=None*, *energy_units='eV'*)
Class describing a free electron

get_conduction_electron_number()
Returns the number of electrons in the conduction band as a 1D numpy array of length of `T`

class spinney.defects.concentration.**DefectConcentration**(*E_form*, *N*, *T*, *g=1*, *energy_units='eV'*)
Class describing defect concentrations in the dilute limit at the thermodynamic equilibrium:

$$n_{eq}^d(T) = \frac{N_d g}{e^{E_f(d)/k_B T} + g}$$

Or the approximated expression often used:

$$n_{eq}^d(T) = N_d g e^{(-E_f(d)/k_B T}$$

N_d is the number of sites available for the specific defect in the crystal.

Parameters

- **E_form** (*float*) – The formation energy of the point defect
- **N** (*float*) – The number of sites available for the defect per volume unit. This is usually expressed in number of sites per cm^3, or number of sites per cell.
- **T** (*array or float*) – The temperatures (in K) to use for calculating the defect concentration
- **g** (*float*) – the intrinsic degeneracy of the defect
- **energy_units** (*string*) – the employed units of energy

dilute_limit_concentration
the defect concentration at each temperature. If T was a scalar, a scalar is returned.

Type 1D numpy array of length len(T)

dilute_limit_approx_concentration
the defect concentration at each temperature, using the more approximative exponential formula. If T was a scalar, a scalar is returned.

Type 1D numpy array of length len(T)

class spinney.defects.concentration.**EquilibriumConcentrations**(*charge_states*, *form_energy_vbm*, *vbm*, *e_gap*, *site_conc*, *dos*, *T_range*, *g=None*, *N_eff=0*, *units_energy='eV'*, *dos_down=None*)

Represents the defects and carriers concentrations for Fermi level values given by the charge neutrality condition in a homogeneous semiconductor:

$$\sum_i q_i n_d(q,T) + p - n = N_d$$

where q_i is the defect charge state, n_d its concentration, p and n are the concentration of free holes and electrons, respectively and N_d is the effective doping level.

Parameters

- **charge_states** (*dict of arrays or None*) – defect_type : list of charge states for all considered defects If equal to None, the pristine semiconductor is considered.

  Example:

  If we consider the Si vacancy and Si interstitials in the charge states -2, -1, 0, 1, 2; then:

  ```
  >>> charge_states = {'Si_int':[-2, -1, 0, 1, 2],
  ...                  'Vac_Si':[-2, -1, 0, 1, 2]}
  ```

- **form_energy_vbm** (*dict of arrays or None*) – defect_type : the defect formation energy calculated for an electron chemical potential equal to *vbm*. For any array, the order of the values must be the same as the one used in *charge_states*

  If equal to None, the pristine semiconductor is considered.

  Example:

  For the defects listed above, we would type:

  ```
  >>> form_energy_vbm = {'Si_int':[val_m2, val_m1, val_0,
  ...                              val_1, val_2],
  ...                    'Vac_Si':[valv_m2, valv_m1, valv_0,
  ...                              valv_1, valv_2]}
  ```

  where `val_m2` is the formation energy of the Si interstitial in charge state -2, `valv_m2` is the one of the Si vacancy in this charge state and so on.

- **vbm** (*float*) – the value of the valence band maximum
- **e_gap** (*float*) – the value of the band gap
- **site_conc** (*dict*) – defect_type : site_concentration for defect_type = 'electron' and 'hole', this value should be the concentration for the unit cell used to calculate `dos`.

  Example:

  For the defects listed above, we have:

  ```
  >>> site_conc = {'Si_int'  : conc_Si_int,
  ...            'Vac_Si'  : conc_Vac_Si,
  ...            'electron': conc_electrons,
  ...            'hole'    : conc_holes}
  ```

- **dos** (*2D array*) – The first column are the energies of 1-electron level, the second the DOS per cell. The values of *vbm* and `e_gap` must be consistent with the dos.
- **T_range** (*array or float*) – the temperature range

- **g** (*dict of lists or None*) – defect_type : [degeneracy charge state 1, degeneracy charge_state 2, . . .] each list represents the degeneracy for a given type of defect in each of its charge states. The order has to match that of *charge_states[defect_type]*.

  If None, all the degeneracy factors are taken equal to 1. The structure is analogous to `charge_states`.
- **N_eff** (*float*) – the effective-doping concentration
- **units_energy** (*str*) – employed energy units
- **dos_down** (*array*) – the eventual dos for the spin-down electrons

defect_order
the ordered sequence with the defect names.

Type tuple

charge_states
defect_name : sequence of charge states of the defect

Type dict

formation_energies_vbm
defect_name : sequence of the defect formation energies, for every charge state listed in `charge_states`, for an electron chemical potential equal to `vbm`

Type dict

formation_energies_equilibrium
defect_name : sequence of the defect formation energies, for every charge state listed in `charge_states`, for an electron chemical potential equal to `equilibrium_fermi_level`

Type dict

site_conc
defect_name : effective concentration for that kind of defect

Type dict

T
the temperatures considered for calculating the defect formation energies

Type numpy 1D array

vbm
the value of the pristine system valence band maximum

Type float

cbm
the value of the pristine system conduction band minimum

Type float

N_eff
effective dopant concentration

Type float

equilibrium_fermi_level
the value of the electron chemical potential at the equilibrium for a given temperature

Type numpy 1D array of length len(self.T)

equilibrium_defect_concentrations
defect_name : {defect_charge_state : array, ... } where array is a numpy 1D array of length len(self.T) holding the calculated defect concentration for that charge state at at given temperature

Type dict

equilibrium_electron_concentrations
len(self.T) the value of the electron concentration at the equilibrium for a given temperature

Type numpy 1D array of length

equilibrium_hole_concentrations
the value of the hole concentration at the equilibrium for a given temperature

Type numpy 1D array of length len(self.T)

equilibrium_carrier_concentrations
len(self.T) the value of the carrier concentration at the equilibrium for a given temperature. If the value is positive, holes are the main carriers; otherwise electrons.

Type numpy 1D array of length

find_root_algo
specifies which algorithm to use in order to find the roots of the charge neutrality condition. Possible values:

- *brentq*, *newton*, *bisect*

Type string

get_equilibrium_defect_concentrations()
Return the equilibrium defect concentration as a function of the temperature.

Returns **concentrations**

Return type 2D numpy array of shape (len(self._charge_states), len(self._T))

get_equilibrium_electron_concentrations()
Return the equilibrium electron concentration as a function of the temperature.

Returns **concentrations**

Return type 1D array of length len(self.T)

get_equilibrium_fermi_level()

Return the equilibrium value of the Fermi level as a function of the temperature.

Returns **equilibrium_fermi_level** – the calculated Fermi level as a function of T

Return type 1D numpy array of length len(self.T)

class spinney.defects.concentration.**FermiDiracDistribution**(*energy*, *mu*, *T*, *energy_units='eV'*)
Implementation of the Fermi-Dirac distribution:

$$\frac{1}{1 + e^{(E-\mu_e)/k_B T}}$$

Parameters

- **energy** (*float or 1D darray*) – the energies of the 1-electron levels
- **mu** (*float*) – the electron chemical potential

- **T** (*float or 1D array*) – the temperature
- **energy_units** (*string*) – the units in which `energy` and *mu* are expressed. `T` is always assumed to be in K.

mu
the electrom chemical potential

Type float

kb
Boltzman constant in terms of `energy_units`

Type float

values
The values of the Fermi distribution for those values of energy and temperature.

Type numpy 2D array of shape (len(energy), len(T))

class spinney.defects.concentration.**ValenceHole**(*dos, vbm, cbm, mu, T, dos_down=None, energy_units='eV'*)
Class describing a free hole

get_valence_holes_number()
Returns the number of holes in the valence band as a 1D numpy array of length of `T`.

spinney.defects.concentration.**extract_formation_energies_from_file**(*file_name*)
Read a file with the format:

```
# arbitrary number of comment lines
defect_name    defect_charge_state   defect_formation_energy
```

Parameters **file_name** (*string*) – the file with the data

Returns **init_args** – the tuple contains two elements: the two dictionaries to be used as the first two arguments for initializing a *EquilibriumConcentrations* instance

Return type tuple

6.5 General-purpose tools

- *spinney.tools.formulas*
- *spinney.tools.reactions*

6.5.1 The formulas module

Module containing functions useful for manipulating chemical formulas

spinney.tools.formulas.**count_elements**(*compound, total=False*)
Returns the number of atoms in a chemical system.

Parameters

- **compound** (*str*) – the compound of interest
- **total** (*bool*) – If True, the total number of atoms in the system is also returned

Returns **el_count** – Dictionary: `atom:number of atoms` for each atom in `compound`. If total, returns also the number of elements in `compound`

Return type dict/tuple

`spinney.tools.formulas.`**`get_formula_unit`**(*compound*)

Gets the formula unit of a particular compound

Parameters **`compound`** (*`string`*) – the formula of the compound.

Returns **formula_unit** – the formula unit of `compound`

Return type string

Notes

This function automatically reduces the coefficients to the smallest integers. It however preserves the number of symbols in the formula; e.g. C6H6 will return CH, but CH3COOH will return still CH3COOH.

`spinney.tools.formulas.`**`get_number_fu`**(*compound*, *fu=None*)

Returns how many formula units `fu` are present in `compound`

Parameters

- **`compound`** (*`string`*) – the formula of the compound
- **`fu`** (*`string`*) – the formula unit If None, the actual formula unit is used

Returns **no_units** – The number of formula units in `compound`

Return type float

`spinney.tools.formulas.`**`get_stoichiometry`**(*compound*, *fractional=True*)

Given `compound`, it returns its stoichiometry.

Parameters

- **`compound`** (*`string`*) – the compound's formula
- **`fractional`** (*`bool`*) – if True, for each element is returned its molar fraction; otherwise, it is returned the number of atoms per formula

Returns **elements_count** – A dictionary element:coefficient for each element in `compound`

Return type dict

6.5.2 The `reactions` module

Helper functions for calculating reaction energies.

`spinney.tools.reactions.`**`calculate_defect_formation_energy`**(*e_defect*, *e_pristine*, *chem_potentials*, *charge_state*, *E_corr=0*)

Calculate the formation energy of a point defect.

Parameters

- **`e_defect`** (*`dict`*) – {formula : energy of the defective supercell}
- **`e_pristine`** (*`dict`*) – {formula : energy of the pristine supercell}
- **`chem_potentials`** (*`dict`*) – {element_name : chemical_potential} element_name is the element symbol, in the case of the electron, element_name = electron
- **`charge_state`** (*`int`*) – the formal charge state of the defect

- **E_corr** (*float*) – eventual corrections to the defect formation energy

Returns **formation_energy**

Return type float

`spinney.tools.reactions.`**`calculate_formation_energy_fu`**(*compound_dict*, *components_dict*)

Calculated the formation energy of a given compound per formula unit

Parameters

- **compound_dict** (*dict*) – `formula : energy` for each compound needed to calculate the defect formation energy: `formula` is a string representing the compound formula, `energy` is a number, representing the corresponding compound energy.
- **components_dict** (*dict*) – `formula : energy` for each compound needed to calculate the defect formation energy: `formula` is a string representing the compound formula, `energy` is a number, representing the corresponding compound energy.

Examples

In order to calculate the formation energy of $SrTiO_3$, `compound_dict` and `components_dict` are:

```
>>> compound_dict = {'SrTiO3':Ec}
>>> components_dict = {'Sr':Esr, 'Ti':Eti, 'O2':Eo2}
```

`spinney.tools.reactions.`**`calculate_reaction_energy`**(*reaction*, *compound_energies*)

Calculates the reaction energy of a compound

Parameters **reaction** (*dict of lists of tuples*) – Describes the reaction that will be calculated. The dictionary keys are 'reactants' and 'products'. The value of 'reactants' is a list with tuples. Each tuple contains the FORMULA UNIT of the reactants and the number of moles for that reactant in the reaction. The value of 'products' is analogous, but for the products.

The compounds specified in `reaction` will be taken as the formula units in calculating the reaction.

compound_energies [dict of lists] Analogous structure as in `reaction`, but instead of the number of moles, the values are the calculated energies.

Note: the order of the compounds in each list has to match that in `reaction`!

E.g. we calculated the energies of 'Mn2' (bcc Mn cell), 'O2' and 'Mn32O48' (Pbca space group) Then we would have:

```
>>> compound_energies = {
                         'reactants' : [('Mn2',E1), ('O2', E2)],
                         'products'  : [('Mn32O48', E3)]
                        }
```

Returns **energy** – The energy of the reaction specified in `reaction`

Return type float

Examples

Suppose one is interested in the reaction:

$$2\mathrm{Mn} + 3/2\mathrm{O}_2 \rightarrow \mathrm{Mn}_2\mathrm{O}_3$$

And one has calculated the energy of Mn2 (bcc cell) and stored it in the variable `E1`, the energy of O2 is stored in `E2` and that of Mn2O3 in `E3`. Then to calculate the reaction energy one can use:

```
>>> reaction = {
                'reactants' : [('Mn', 2), ('O2', 3/2)],
                'products'  : [('Mn2O3', 1)]
               }
>>> compound_energies = {
                         'reactants' : [('Mn2', E1), ('O2', E2)],
                         'products'  : [('Mn2O3', E3)]}
>>> calculate_reaction_energy(reaction, compound_energies)
```

`spinney.tools.reactions.`**`get_compound_energy_per_atom`**(*energy*, *formula*)

Returns the energy of the compound per atom (unit_energy/atom)

Parameters

- **`energy`** (*`float`*) – the energy of the compound
- **`formula`** (*`string`*) – the formula of the compound

Returns **energy** – The energy per atom

Return type float

`spinney.tools.reactions.`**`get_compound_energy_per_formula_unit`**(*energy*, *formula*, *formula_unit=None*)

Returns the calculated energy of the compound per formula unit (unit_energy/f.u.)

Parameters

- **`energy`** (*`float`*) – the energy of the compound with formula `formula`
- **`formula`** (*`string`*) – the compound formula
- **`formula_unit`** (*`string`*) – The formula unit of the compound

Returns **energy** – The total energy per formula unit and the number of formula units in `formula`

Return type float

6.6 Support for first-principles codes

- *`spinney.io.vasp`*
- *`spinney.io.wien2k`*

6.6.1 The `io.vasp` module

Helper functions for VASP postprocessing

`spinney.io.vasp.`**`extract_dos`**`(`*vasprun_file*, *save_dos=False*`)`

From the vasprun.xml file extrat the DOS.

Parameters

- **`vasprun_file`** (*`string`*) – path to the vasprun.xml file
- **`save_dos`** (*`bool`*) – if True, the extracted DOS are saved as a text file. The first column is the energy (in eV) and the second the DOS (states/cell)

Returns **dos** – each element is a 2D numpy array. First column

Return type 2-ple, the DOS up and DOS down (if any)

`spinney.io.vasp.`**`extract_potential_at_core_vasp`**`(`*file*`)`

Read the VASP OUTCAR file and extract the values of the electrostatic potential evaluated at the ions positions.

Parameters **`file`** (*`string`*) – path to the OUTCAR file

Returns **result** – for each atom in the system, returns the electrostatic potential at the atomic sites

Return type 1D numpy array

6.6.2 The `io.wien2k` module

Helper functions for WIEN2k

`spinney.io.wien2k.`**`average_core_potential_wien2k`**`(`*potential*, *r0*, *rmax*`)`

Calculates the average core potential within a spherical shell with the smallest raius r0 and the largest rmax.

Parameters

- **`potential`** (*`1D numpy array`*) – the radial part of the potential corresponding to the l=0, m=0 angular component.
- **`r0`** (*`float`*) – the smallest radius of the shell
- **`rmax`** (*`float`*) – the largest radius of the shell

Returns **float**

Return type the averaged potential in the spherical shell

`spinney.io.wien2k.`**`extract_potential_at_core_wien2k`**`(`*struct*, *vcoul*`)`

Extracts the average electrostatic potential within the atomic spheres for each atom in the system.

Parameters

- **`struct`** (*`string`*) – the path to the .struct file
- **`vcoul`** (*`string`*) – the path to the .vcoul file

Returns **result** – for each atom in the system, returns the average electrostatic potential within the spherial region

Return type 1D numpy array

`spinney.io.wien2k.`**`prepare_ase_atoms_wien2k`**`(`*struct_file*, *scf_file*`)`

Prepare an `ase.Atoms` object compatible with the interface of the `PointDefect` class in Spinney.

Parameters

- **`struct_file`** (*`string`*) – the path to the WIEN2K .struct file
- **`total_energy`** (*`float`*) – the value of the total energy as found in the WIEN2K .scf file

Returns **ase_wien** – the `ase.Atoms` object representing the system

Return type `ase.Atoms`

`spinney.io.wien2k.`**`read_energy_wien2k`**(*scf_file*)

Returns the total electronic energy.

Parameters **`scf_file`** (*`string`*) – path to the WIEN2k .scf file

Returns **energy** – the energy of the system

Return type float

`spinney.io.wien2k.`**`read_wien2k_radial_data`**(*struct_file*)

Reads from a .struct file some information about the atom-centered spheres related to the radial potential within the sphere.

Parameters **`struct_file`** (*`str`*) – path to the struct file

Returns No_atoms is the number of irreducible atoms in the system, for each of them, we store R0, RMT, NPT, MULTI

Return type 2D tuple array of shape (No_atoms, 4)

`spinney.io.wien2k.`**`read_wien2k_vcoul`**(*vcoul_file*)

Reads the radial part of the electrostatic potential, corresponding to the angular term l=0, m=0, inside the atomic sphere.

Parameters **`vcoul_file`** (*`str`*) – the path of the .vcoul file

Returns **result** – each list contains the radial potential for one atom

Return type list of lists

CHAPTER

SEVEN

BIBLIOGRAPHY

CHAPTER EIGHT

CONTACT

8.1 E-mail

The developers of **`Spinney`** can be reached through these e-mail addresses:

- marco.arrigoni@tuwien.ac.at, marco.arrigoni@outlook.de
- georg.madsen@tuwien.ac.at

8.2 Gitlab

`Spinney` is hosted on Gitlab on the following web page:

- https://gitlab.com/Marrigoni/spinney

BIBLIOGRAPHY

[FGH+14] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and Chris G. Van de Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86:253–305, Mar 2014. doi:10.1103/RevModPhys.86.253.

[FNVdW09] Christoph Freysoldt, Jörg Neugebauer, and Chris G. Van de Walle. Fully ab initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, Jan 2009. doi:10.1103/PhysRevLett.102.016402.

[KO14] Yu Kumagai and Fumiyasu Oba. Electrostatics-based finite-size corrections for first-principles point defect calculations. *Phys. Rev. B*, 89:195205, May 2014. doi:10.1103/PhysRevB.89.195205.

[LVdW17] John L. Lyons and Chris G. Van de Walle. Computationally predicted energies and properties of defects in gan. *npj Computational Materials*, 3(1):12, 2017. doi:10.1038/s41524-017-0014-2.

[MWC98] Jr Malcolm W. Chase. *NIST-JANAF thermochemical tables*. Fourth edition. Washington, DC : American Chemical Society ; New York : American Institute of Physics for the National Institute of Standards and Technology, 1998., 1998. URL: https://search.library.wisc.edu/catalog/999842910902121.

[PTA+92] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. Iterative minimization techniques for ab initio total-energy calculations: molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64:1045–1097, Oct 1992. doi:10.1103/RevModPhys.64.1045.

[ZN91] S. B. Zhang and John E. Northrup. Chemical potential dependence of defect formation energies in gaas: application to ga self-diffusion. *Phys. Rev. Lett.*, 67:2339–2342, Oct 1991. doi:10.1103/PhysRevLett.67.2339.

PYTHON MODULE INDEX

INDEX