qmpy Documentation

Release 0.4.9

Scott Kirklin

July 02, 2014

CONTENTS

qmpy is the backend responsible for creating and running the Open Quantum Materials Database (http://oqmd.org). The OQMD is a project created in Chris Wolverton's group at Northwestern University (http://wolverton.northwestern.edu).

CHAPTER

ONE

INSTALLATION

1.1 From repo

Install qmpy with pip or easy install:

```
>>> pip install qmpy
```

or:

```
>>> easy_install -U qmpy
```

Note: Using pip or easy_install to install scipy or numpy can be unreliable. It is better to install from a proper repository for your linux distribution. However, if that version of SciPy is earlier than 0.12.0 you will need to obtain another installation. If necessary, you can obtain the needed libraries with:

```
$ sudo apt-get install libatlas-dev libatlas-base-dev
$ sudo apt-get install liblapack-dev gfortran
```

1.2 From GitHub repo

Obtain the source with:

```
$ git clone https://github.com/wolverton-research-group/qmpy.git
$ cd qmpy
$ python setup.py install
```

Be aware that f you want to install qmpy from source, you will be responsible for ensuring that you have all of the following required packages installed.

1.3 Required Packages

- Django (https://www.djangoproject.com/)
- Numpy (http://www.numpy.org/)
- Scipy (http://www.scipy.org/)
- PyYAML (http://pyyaml.org/)
- python-MySQL (https://pypi.python.org/pypi/MySQL-python)

- python-memcached
- django-extensions
- PuLP (https://pythonhosted.org/PuLP/) (required for grand canonical linear programming and high-dimensional phase diagram slices)

1.4 Recommended Packages

- matplotlib (http://matplotlib.org/) (required for creating figures)
- networx (http://networkx.github.io/) (required for creating spin lattices, and some high-dimensional phase diagram analysis)

Warning: In order for pulp to work, you must have a working linear programming package installed. PuLP provides a simple library for this, but it is up to you to make sure it is working.

CHAPTER

TWO

SETTING UP THE DATABASE

The database can be downloaded from http://oqmd.org/static/downloads/qmdb.sql.gz

Once you have the database file, you need to unzip it and load it into a database MySQL. On a typical linux installation this process will look like:

```
$ wget http://oqmd.org/static/downloads/qmdb.sql.gz
$ gunzip qmdb.sql.gz
$ mysql < qmdb.sql
```

Note: Assuming your install is on linux, and assuming you haven't used MySQL at all, you will need to enter a mysql session as root ("mysql -u root -p"), create a user within MySQL ("CREATE USER 'newuser'@'localhost';"), grant that user permissions ("GRANT ALL PRIVILEGES ON * . * TO 'newuser'@'localhost'; FLUSH PRIVILEGES;").

Note: The name of the deployed database has changed since previous releases (qmdb_prod). If your install isn't working, make sure that the database name agrees with what is found in qmpy/db/settings.py.

Once this is done, you need to edit qmpy/db/settings.py. Set the DATABASES variable such that 'USER' is the user with permissions to access the newly installed database.

Note: For windows/cygwin users: To use MySQL in Cygwin, you need to install MySQL via the Oracle website for windows. Only after MySQL is install in windows can use mysql in Cygwin. You can find the download for MySQL here: http://dev.mysql.com/downloads/windows/installer.

It is free, but you have to register with Oracle to access. Next, you need to move the database file over to the Windows MySQL data drive. It may vary by version, but you might find it at C:ProgramDataMySQLMySQL Server 5.6data. Copy the downloaded database directory into this folder. Finally, in the db/settings.py file, the HOST has to be set to '127.0.0.1', and set the PORT and PASSWORD variables as well according to your MySQL installation.

To verify that the database is properly installed and has appropriate permissions, run:

```
mysql> select count(*) from entries;
+----------+
| count(*) |
+----------+
|   173653 |
+----------+
```

The number may not match what is shown above, but as long as you don't recieve any errors, your database should be working properly.

CHAPTER

THREE

TUTORIALS

qmpy is a package containing many tools for computational materials science.

The qmpy package comes bundled with two executable scripts, *qmpy* and *oqmd*. *qmpy* is a simple bash script that starts an interactive python environment and imports qmpy:

```
$ qmpy
>>>
```

To write your own python script that utilizes qmpy functionality, simply start with an import like:

```
from qmpy import *
```

and all of the commands shown below should work.

3.1 Database entries

Once the database is installed, you can query it very flexibly and easily. In this section we will explore the data structure of entries in the OQMD and provide several examples of how to make queries. For deeper understanding of how django models work, you should check out the (excellent) django documentation.

First, lets look at how to access an entry from the database. As an example, lets pull up an entry for an `Element`:

```
>>> fe = Element.objects.get(symbol='Fe')
>>> fe
<Element: Fe>
```

Django models have a number of fields that can be accessed directly once the database entry has been loaded. For example, with an element you can:

```
>>> fe.symbol
u'Fe'
>>> fe.z
26L
```

For a complete list of the model attributes that are stored in the database, refer to the documentation for the model you are interested in, in this case `Element`.

Note: When strings are returned, they are returned as unicode strings, (indicated by the "u" preceding the string) integers are returned as long integers (indicated by the trailing "L"). For most purposes this makes no difference, as these data types will generally behave exactly as expected, i.e.:

```
>>> fe.z == 26
True
```

```
>>> fe.symbol == "Fe"
True
```

In addition to data attributes like these, django models have relationships to other models. In qmpy there are two flavors of relationships: one-to-many and many-to-many. An example of a one-to-many relationship would be the relationship between an `Element` and an `Atom`. There are many atoms which are a given element, but each atom is only one element. In the case of Fe:

```
>>> fe.atom_set
<django.db.models.fields.related.RelatedManager object at 0x7f0997fa2690>
>>> fe.atom_set.count()
127585
>>> atom = fe.atom_set.first()
>>> print atom
<Atom: Fe @ 0.000000 0.000000 0.000000>
>>> atom.element
<Element: Fe>
```

A `RelatedManager` is an object that deals with obtaining other django models that are related to the main object. We can use the objects.count() method to fidn the number of `Atom` objects that are Fe, and find ~125,000. To obtain one of these atoms, we use the objects.first() method, which simply returns the first `Atom` which is Fe. Much more functionality of Managers and RelatedManagers will be shown throughout this tutorial and in the examples, but for a proper understanding you should refer to the django docs.

An example of a many-to-many relationship would be the relationship between an `Element` and a `Composition`. A composition (e.g. Fe2O3) can contain many elements (Fe and O), and an element can be a part of many compositions (Fe3O4 and FeO as well). This is the nature of a many-to-many relationship. In the case of Fe:

```
>>> fe.composition_set.count()
10882
>>> comp = fe.composition_set.filter(ntypes=2)[0]
>>> comp
<Composition: AcFe>
>>> comp.element_set.all()
[<Element: Ac>, <Element: Fe>]
```

In this example we have taken our base object (the `Element`) and filtered its composition_set for `Composition` objects which meet the condition ntypes=2 (i.e. there are two elements in the composition), and taken the first such `Composition` (index 0 in the `QuerySet` that is returned).

3.2 Creating a structure

There are several ways to create a structure, but we will start with reading in a POSCAR:

```
>>> s = io.read(INSTALL_PATH+'/io/files/POSCAR_BCC')
```

Once you have the `Structure` object, the important features of a crystal structure can be accessed readily.:

```
>>> s.cell
array([[ 3.,  0.,  0.],
       [ 0.,  3.,  0.],
       [ 0.,  0.,  3.]])
>>> s.lat_params
[3.0, 3.0, 3.0, 90.0, 90.0, 90.0]
>>> s.atoms
[<Atom: Cu @ 0.000000 0.000000 0.000000>, <Atom: Cu @ 0.500000 0.500000
0.500000>]
>>> s.composition
<Composition: Cu>
>>> s.volume
27.0
```

You can also readily construct a `Structure` from scratch, from the lattice vectors and the atom positions.:

```
>>> s2 = Structure.create([3,3,3], [('Cu', [0,0,0]),
                                    ('Cu', [0.5,0.5,0.5])])
>>> s2 == s
True
```

3.3 First Principles Calculations

At this time qmpy only supports automation of calculations using the Vienna Ab Initio Simulation Package (VASP). The reading and creation of these calculations are handled by the `Calculation` model. To read in an existing calculation:

```
>>> path = '/analysis/vasp/files/normal/fine_relax/'
>>> calc = Calculation.read(INSTALL_PATH+path)
```

qmpy will search the directory for an OUTCAR or OUTCAR.gz file. If it is able to find an OUTCAR, it will attempt to read the file. Next, we will demonstrate several of the key attributes you may wish to access:

```
>>> calc.energy # the final total energy
-12.416926999999999
>>> calc.energies # the total energies of each step
array([-12.415236 -12.416596, -12.416927])
>>> calc.volume # the output volume
77.787375068172508
>>> calc.input
<Structure: SrGe2>
>>> calc.output
<Structure: SrGe2>
>>> from pprint import pprint
>>> pprint(calc.settings)
{'algo': 'fast',
 'ediff': 0.0001,
 'encut': 373.0,
 'epsilon': 1.0,
 'ibrion': 1,
 'idipol': 0,
 'isif': 3,
 'ismear': 1,
 'ispin': 1,
 'istart': 0,
 'lcharg': True,
 'ldipol': False,
 'lorbit': 0,
 'lreal': False,
 'lvtot': False,
 'lwave': False,
 'nbands': 24,
 'nelm': 60,
 'nelmin': 5,
 'nsw': 40,
 'potentials': [{'name': 'Ge_d', 'paw': True, 'us': False, 'xc': 'PBE'},
                {'name': 'Sr_sv', 'paw': True, 'us': False, 'xc': 'PBE'}],
 'potim': 0.5,
 'prec': 'med',
 'pstress': 0.0,
 'sigma': 0.2}
```

3.4 Searching for models

The documentation for Django for searching for models is ver complete, and should be taken as the ultimate reference for searching for models in qmpy, but a basic overview is provided here.

3.4.1 Searching for entries based on stability

Formation energies are stored as FormationEnergy instances, which are associated with an *:mod:~qmpy.Entry* and a *:mod:~qmpy.Calculation*. Knowing this, we can search for stable Entries using:

```
>>> stable = Entry.objects.filter(formationenergy__stability__lt=0)
>>> stable.count()
18150
```

The same concept can be applied to searching for other quantities, as long as you can relate them to a FormationEnergy by "__" constructions:

```
>>> stable_comps = Composition.objects.filter(formationenergy__stability__lt=0)
>>> stable_comps.count()
18150
>>> s = Structure.objects.filter(calculated__formationenergy__stability__lt=0)
>>> s.count()
18150
```

Adding other search criteria lets you explore a little more:

```
>>> stable = FormationEnergy.objects.filter(stability__lt=0)
>>> # Find the number of stable compounds containing O
>>> stable.filter(composition__element_set='O').count()
4017
>>> # or Fe. Is it surprising that this is smaller?
>>> stable.filter(composition__element_set='Fe').count()
653
>>> # Meta data is also a possiblity. How many stable compounds were found
>>> # in the course of calculations for a particular project?
>>> stable.filter(entry__project_set='prototypes').count()
3119
```

3.4.2 Searching for entries based on composition

You can find compositions in a few ways using filters and excludes. If you want a specific region of phase space (including related subspaces):

```
>>> elts = [ 'Fe', 'Li', 'O' ]
>>> others = Element.objects.exclude(symbol__in=elts)
>>> comps = Composition.objects.exclude(element_set=others)
```

This searchs finds every composition that doesn't have any elements that aren't in the region of phase space requested. For binary or ternary phase spaces it can be more efficient to search permutations of sub-spaces:

```
>>> comps = Composition.objects.filter(ntypes=3)
>>> for e in elts:
>>>     comps = comps.filter(element_set=e)
>>> for e in elts:
>>>     e_comps = Composition.objects.filter(element_set=e, ntypes=1)
>>>     comps |= e_comps
>>> for e1, e2 in itertools.combinations(elts, r=2):
>>>     bin_comps = Composition.objects.filter(element_set=e1)
>>>     bin_comps = bin_comps.filter(element_set=e2, ntypes=2)
>>>     comps |= bin_comps
>>> comps.distinct().count()
```

However, for larger regions of phase space (4 or 5 or more) the number of subqueries of the second approach rapidly becomes more expensive than the single, more complicated query of the first.

CHAPTER

FOUR

DATA MODELS

qmpy is a package containing many tools for computational materials science.

exception `qmpy.`**`qmpyBaseError`**

Baseclass for qmpy Exceptions

4.1 Materials models

4.1.1 Structure

class `qmpy.`**`Structure`**(**args*, ***kwargs*)

Structure model. Principal attributes are a lattice and basis set.

Relationships:

`Entry` via entry

`Atom` via atom_set: Atoms in the structure. More commonly handled by the managed attributed *atoms*.

`Calculation` via calculated. Calculation objects that the structure is an *output* from.

`Calculation` via calculation_set. Returns calculation objects that the structure is an *input* to.

`Composition` via composition.

`Element` via element_set

`Spacegroup` via spacegroup

`Species` via species_set

`Prototype` via prototype. If the structure belongs to a prototypical structure, it is referred to here.

`Reference` Original literature reference.

`MetaData` via meta_data

Attributes:

Identification

id

label: key in the Entry.structures dictionary.

natoms: Number of atoms.

nsites: Number of sites.

ntypes: Number of elements.

measured: Experimentally measured structure?

source: Name for source.

Lattice
x1, x2, x3
y1, y2, y3
z1, z2, z3: Lattice vectors of the cell. Accessed via *cell*.
volume
volume_pa

Calculated properties
delta_e: Formation energy (eV/atom)
meta_stability: Distance from the convex hull (eV/atom)
energy: Total DFT energy (eV/FU)
energy_pa: Total DFT energy (eV/atom)
magmom: Total magnetic moment (&Mu;<sub>b</sub>)
magmom_pa: Magnetic moment per atom.
sxx, sxy, syy
syx, szx, szz: Stresses on the cell. Accessed via *stresses*.

Examples:

```
>>> s = io.read(INSTALL_PATH+'/io/files/POSCAR_FCC')
>>> s.atoms
>>> s.cell
>>> s.magmoms
>>> s.forces
>>> s.stresses
```

add_atom(*atom*, *tol=0.01*)
Adds *atom* to the structure if it isn't already contained.

atom_types
List of atomic symbols, length equal to number of atoms.

atomic_numbers
List of atomic numbers, length equal to number of atoms.

atoms
List of `Atoms` in the structure.

cartesian_coords
Return atomic positions in cartesian coordinates.

cell
Lattice vectors, 3x3 numpy.ndarray.

comment_objects
Return list of comments (MetaData objects of type comment)

comp
Composition dictionary.

compare(*other*, *tol=0.01*, *atom_tol=10*, *volume=False*, *allow_distortions=False*, *check_spacegroup=False*, *wildcard=None*)
Credit to K. Michel for the algorithm.

1. Check that elements are the same in both structures

2.Convert both structures to primitive form

3.Check that the total number of atoms in primitive cells are the same

4. Check that the number of atoms of each element are the same in primitive cells

4b. Check that the spacegroup is the same.

5.If needed check that the primitive cell volumes are the same

6. Convert both primitive cells to reduced form There is one issue here - the reduce cell could be type I (all angles acute) or type II (all angles obtuse) and a slight difference in the initial cells could cause two structures to reduce to different types. So at this step, if the angles are not correct, the second cell is transformed as [[-1, 0, 0], [0, -1, 0], [0, 0, 1]].

7. Check that the cell internal angles are the same in both reduced cells.

8. Check that the ratios of reduced cell basis lengths are the same. ie a1/b1 = a2/b2, a1/c1 = a2/c2, and b1/c1 = b2/c2 where a1, b1, c1, are the lengths of basis vectors in cell 1 and a2, b2, c2 are the lengths of cell 2.

9. Get the lattice symmetry of the reduced cell 2 (this is a list of all rotations that leave the lattice itself unchanged). In turn, apply all rotations to reduced cell 2 and for each search for a vector that overlaps rotated cell positions with positions in reduced cell 1. If a rotation + translation overlaps reduced cells, then they are the same.

MODIFICATIONS: Only need one "base" atom from the first structure Get the distance from the origin for every atom first

Arguments: other: Another `Structure`.

Keyword Arguments: tol: Percent deviation in lattice parameters and angles.

Not Implemented Yet: wildcard: Elements of the specified type can match with any atom.

coords
numpy.ndarray of atom coordinates.

copy()
Create a complete copy of the structure, with any primary keys removed, so it is not associated with the original.

static **create**(*cell*, *atoms=[ ]*, ***kwargs*)
Creates a new Structure.

Arguments: cell: 3x3 lattice vector array

Keyword Arguments: atoms: List of `Atom` creation arguments. Can be a list of [element, coord], or a list of [element, coord, kwargs].

Examples:

```
>>> a = 3.54
>>> cell = [[a,0,0],[0,a,0],[0,0,a]]
>>> atoms = [('Cu', [0,0,0]),
             ('Cu', [0.5, 0.5, 0.5])]
>>> s = Structure.create(cell, atoms)
>>> atoms = [('Cu', [0,0,0], {'magmom':3}),
        ('Cu', [0.5, 0.5, 0.5], {'magmom':-3})]
>>> s2 = Structure.create(cell, atoms)
```

create_vacuum(*direction*, *amount*, *in_place=True*)
Add vacuum along a lattice direction.

Arguments: direction: direction to add the vacuum along. (0=x, 1=y, 2=z) amount: amount of vacuum in Angstroms.

Keyword Arguments: in_place: apply change to current structure, or return a new one.

Examples:

```
>>> s = io.read(INSTALL_PATH+'/io/files/POSCAR_FCC')
>>> s.create_vacuum(2, 5)
```

`elements`
List of Elements

`find_lattice_points_by_transform`(*transform*, *tol=1e-06*)
Find the lattice points contained within the transformation.

`find_lattice_points_within_distance`(*distance*, *tol=1e-06*)
Find the lattice points contained within radius *distance* from the origin.

`find_nearest_neighbors`(*method='closest'*, *tol=0.05*, *limit=5.0*)
Determine the nearest neighbors for all Atoms in Structure.

Calls `get_nearest_neighbors()` and assigns the nearest neighbor dictionary to *Structure._neighbor_dict*. Each atom is also given a list, *nearest_neighbors* that contains the nearest neighbor atoms. For atoms which have the "same" atom as a nearest neighbor across different periodic boundaries, a single atom may appear multiple times on the list.

Keyword Arguments: limit: How far to look from each atom for nearest neighbors. Default=5.0.

tol: A tolerance which determines how much further than the closest atom a second atom can be and still be a part of the nearest neighbor shell.

Returns: None

`forces`
numpy.ndarray of forces on atoms.

`get_distance`(*atom1*, *atom2*, *limit=None*, *wrap_self=True*)
Calculate the shortest distance between two atoms.

Note: This is not as trivial a problem as it sounds. It is easy to demonstrate that for any non-cubic cell, the normal method of calculating the distance by wrapping the vector in fractional coordinates to the range (-0.5, 0.5) fails for cases near (0.5,0.5) in Type I cells and near (0.5, -0.5) for Type II.

To get the correct distance, the vector must be wrapped into the Wigner-Seitz cell.

Arguments: atom1, atom2: (`Atom`, `Site`, int).

Keyword Arguments:

limit: If a limit is provided, returns None if the distance is greater than the limit.

wrap_self: If True, the distance from an atom to itself is 0, otherwise it is the distance to the shortest periodic image of itself.

`get_sites`(*tol=0.1*)
From self.atoms, creates a list of Sites. Atoms which are closer than tol from one another are considered on the same site.

`get_spin_lattice`(*elements=None*, *supercell=None*)
Constructs a lattice of sites.

Keyword Arguments:

elements: If *elements* is supplied, get_spin_lattice will return the lattice of those elements only.

supercell: Accepts any valid input to Structure.transform to construct a supercell, and return its lattice. Useful for finding AFM orderings for structures which have a smaller periodicity than their magnetic structure.

Returns: A SpinLattice, which is a container for a lattice graph, which contains nodes which are atoms and edges which indicate nearest neighbors.

Examples:

```
>>> s = io.read(INSTALL_PATH+'/io/files/fe3o4.cif')
>>> sl = s.get_spin_lattice(elements=['Fe'])
>>> sl.set_fraction(0.33333)
>>> sl.fraction
0.3333333333333333
>>> sl.run_MC()
```

get_volume()
Calculates the volume from the triple product of self.cell

group_atoms_by_symmetry()
Sort self.atoms according to the site they occupy.

inv
Precalculates the inverse of the lattice, for faster conversion between cartesian and direct coordinates.

is_buerger_cell(*tol=1e-05*)
Tests whether or not the structure is a Buerger cell.

is_niggli_cell(*tol=1e-05*)
Tests whether or not the structure is a Niggli cell.

joggle_atoms(*distance=0.001*, *in_place=True*)
Randomly displace all atoms in each direction by a distance up to +/- the distance keyword argument (in Angstroms).

Optional keyword arguments:

distance [Range within all internal coordinates are] displaced. Default=1e-3

in_place [If True, returns an ndarray of the applied] translations. If False, returns (Structure, translations).

Examples:

```
>>> s = io.read('POSCAR')
>>> s2, trans = s.joggle_atoms(in_place=False)
>>> trans = s.joggle_atoms(1e-1)
>>> trans = s.joggle_atoms(distance=1e-1)
```

keyword_objects
Return list of keywords (MetaData objects of type keyword)

lat_param_dict
Dictionary of lattice parameters.

lat_param_string(*format='screen'*)
Generates a human friendly representation of the lattice parameters of a structure.

Keyword Arguments: format: ('screen'|'html'|'mathtype')

lat_params
Tuple of lattice parameters (a, b, c, alpha, beta, gamma).

lp
Tuple of lattice parameters (a, b, c, alpha, beta, gamma).

magmoms
numpy.ndarray of magnetic moments of shape (natoms,).

make_conventional(*in_place=True*, *tol=1e-05*)
Uses spglib to convert to the conventional cell.

Keyword Arguments:

in_place: If True, changes the current structure. If false returns a new one

tol: Symmetry precision for symmetry analysis

Examples:

```
>>> s = io.read(INSTALL_PATH+'io/files/POSCAR_FCC_prim')
>>> print len(s)
1
>>> s.make_conventional()
>>> print len(s)
4
```

make_perfect(*in_place=True*, *tol=0.1*)
Constructs options for a 'perfect' lattice from the structure.

If a site is not fully occupied, but has only one atom type on it, it will be filled the rest of the way. If a site has two or more atom types on it, the higher fraction element will fill the site.

Keyword Arguments: in_place: If False returns a new Structure, otherwise returns None

tol: maximum defect concentration.

Examples:

```
>>> s = io.read(INSTALL_PATH+'/io/files/partial_vac.cif')
>>> s
<Structure: Mn3.356Si4O16>
>>> s.make_perfect()
>>> s
<Structure: MnSiO4>
>>> s = io.read(INSTALL_PATH+'/io/files/partial_mix.cif')
>>> s
<Structure: Mn4.264Co3.736Si4O16>
>>> s2 = s.make_perfect(in_place=False)
>>> s2
<Structure: MnCoSiO4>
```

make_primitive(*in_place=True*, *tol=1e-05*)
Uses spglib to convert to the primitive cell.

Keyword Arguments:

in_place: If True, changes the current structure. If false returns a new one

tol: Symmetry precision for symmetry analysis

Examples:

```
>>> s = io.read(INSTALL_PATH+'io/files/POSCAR_FCC')
>>> print len(s)
4
>>> s.make_primitive()
>>> print len(s)
1
```

metical_matrix

np.dot(self.cell.T, self.cell)

name

Unformatted name.

nearest_neighbor_dict

Dict of Atom:[list of Atom] pairs.

pdf_compare(*other*, *tol=0.01*)

Compute the PDF for each structure and evaluate the overlap integral for all pairs of species.

recenter(*atom*, *in_place=True*, *middle=False*)

Translate the internal coordinates to center the specified atom. Atom can be an actual Atom from the Structure.atoms list, or can be identified by index.

Keyword Arguments:

in_place: If False, return a new Structure with the transformation applied. defaults to True.

middle: If False, "centers" the cell by putting the atom at the origin. If True, "centers" the cell by putting the atom at (0.5,0.5,0.5). defaults to False.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.recenter(s[2])
>>> s2 = s.recenter(s[0], in_place=False)
>>> s2.recenter(2)
>>> s == s2
True
```

reciprocal_lattice

Reciprocal lattice of the structure.

reduce(*tol=0.001*, *limit=1000*, *in_place=True*)

Get the transformation matrix from unit to reduced cell Acta. Cryst. (1976) A32, 297 Acta. Cryst. (2003) A60, 1

Optional keyword arguments:

tol [] eps_rel in Acta. Cryst. 2003 above. Similar to tolerance for floating point comparisons. Defaults to 1e-5.

limit [] maximum number of loops through the algorithm. Defaults to 1000.

in_place [] Change the Structure or return a new one. If True, the transformation matrix is returned. If False, a tuple of (Structure, transformation_matrix) is returned.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.reduce()
>>> s.reduce(in_place=False, get_transform=False)
```

refine (*tol=0.001*, *recurse=True*)

Identify atoms that are close to high symmetry sites (within *tol* and shift them onto them.

Note: "symprec" doesn't appear to do anything with spglib, so I am unable to get "loose" symmetry operations. Without which, this doesn't work.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.symmetrize()
>>> print s.spacegroup
225L
>>> s.refine()
>>> print s.spacegroup
1L
```

set_magnetism (*type*, *scheme='primitive'*)

Assigns magnetic moments to all atoms in accordance with the specified magnetism scheme.

Schemes:

Keyword	Description
None	all magnetic moments = None
"ferro"	atoms with partially filled d and f shells are assigned a magnetic moment of 5 mu_b
"anti"	finds a highly ordererd arrangement arrangement of up and down spins. If only 1 magnetic atom is found a ferromagnetic arrangment is used. raises NotImplementedError

set_natoms (*n*)

Set self.atoms to n blank Atoms.

set_nsites (*n*)

Sets self.sites to n blank Sites.

set_volume (*value*)

Rescales the unit cell to the specified volume, keeping the direction and relative magnitudes of all lattice vectors the same.

site_coords

numpy.ndarray of site coordinates.

sites

List of `Sites` in the structure.

spec_comp

Species composition dictionary.

species

List of species

species_types

List of species, length equal to number of atoms.

stresses

Calculated stresses, a numpy.ndarray of shape (6,)

sub (*replace*, *rescale=True*, *in_place=False*, ***kwargs*)

Replace atoms, as specified in a dict of pairs.

Keyword Arguments:

rescale: rescale the volume of the final structure based on the per atom volume of the new composition.

in_place: change the species of the current Structure or return a new one.

Examples:

```
>>> s = io.read('POSCAR-Fe2O3')
>>> s2 = s.substitute({'Fe':'Ni', 'O':'F'} rescale=True)
>>> s2.substitute({'Ni':'Co'}, in_place=True, rescale=False)
```

substitute (*replace*, *rescale=True*, *in_place=False*, ***kwargs*)
Replace atoms, as specified in a dict of pairs.

Keyword Arguments:

rescale: rescale the volume of the final structure based on the per atom volume of the new composition.

in_place: change the species of the current Structure or return a new one.

Examples:

```
>>> s = io.read('POSCAR-Fe2O3')
>>> s2 = s.substitute({'Fe':'Ni', 'O':'F'} rescale=True)
>>> s2.substitute({'Ni':'Co'}, in_place=True, rescale=False)
```

symmetrize (*tol=0.001*, *angle_tol=-1*)
Analyze the symmetry of the structure. Uses spglib to find the symmetry.

symmetrize sets:

- spacegroup -> Spacegroup
- uniq_sites -> list of unique Sites
- orbits -> lists of equivalent Atoms
- rotations -> List of rotation operations
- translations -> List of translation operations
- operatiosn -> List of (rotation,translation) pairs
- for each atom: atom.wyckoff -> WyckoffSite
- for each site: site.multiplicity -> int

t (*transform*, *in_place=True*, *tol=1e-05*)
Apply lattice transform to the structure. Accepts transformations of shape (3,) and (3,3).

Optional keyword arguments:

in_place [If False, return a new Structure with the] transformation applied.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.transform([2,2,2]) # 2x2x2 supercell
>>> s.transform([[0,1,0],[1,0,0],[0,0,1]]) # swap axis 1 for 2
>>> s2 = s.transform([2,2,2], in_place=False)
```

transform (*transform*, *in_place=True*, *tol=1e-05*)
Apply lattice transform to the structure. Accepts transformations of shape (3,) and (3,3).

Optional keyword arguments:

in_place [If False, return a new Structure with the] transformation applied.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.transform([2,2,2]) # 2x2x2 supercell
>>> s.transform([[0,1,0],[1,0,0],[0,0,1]]) # swap axis 1 for 2
>>> s2 = s.transform([2,2,2], in_place=False)
```

`translate`(*cv*, *cartesian=True*, *in_place=True*)
Shifts the contents of the structure by a vector.

Optional keyword arguments:

cartesian [If True, translation vector is taken to be] cartesian coordinates. If False, translation vector is taken to be in fractional coordinates. Default=True

in_place [If False, return a new Structure with the] transformation applied.

Examples:

```
>>> s = io.read('POSCAR')
>>> s.translate([1,2,3])
>>> s.translate([0.5,0.5, 0.5], cartesian=False)
>>> s2 = s.translate([-1,2,1], in_place=False)
```

`unit_comp`
Composition dict, where sum(self.unit_comp.values()) == 1

class `qmpy.`**`Prototype`**(**args*, ***kwargs*)
Base class for a prototype structure.

Relationships:

`Composition` via composition_set
`Structure` via structure_set
`Entry` via entry_set

Attributes:

name: Prototype name.

classmethod `get`(*name*)
Retrieves a `Prototype` named *name* if it exists. If not, creates a new one.

Examples:

```
>>> proto = Prototype.get('Corundum')
```

4.1.2 Atom

class `qmpy.`**`Atom`**(**args*, ***kwargs*)
Model for an Atom.

Relationships:

`Structure` via structure
`Element` via element
`Site` via site
`WyckoffSite` via wyckoff

Attributes:

id

x, y, z: Coordinate of the atom

fx, fy, fz: Forces on the atom

magmom: Magnetic moment on the atom (in &Mu;<sub>b</sub>)

occupancy: Occupation fraction (0-1).

ox: Oxidation state of the atom (can be different from charge)

charge: Charge on the atom

volume: Volume occupied by the atom

`cart_coord`
Cartesian coordinates of the Atom.

`coord`
[x,y,z] coordinates.

`copy`()
Creates an exact copy of the atom, only without the matching primary key.

Examples:

```
>>> a = Atom.get('Fe', [0,0,0])
>>> a.save()
>>> a.id
1
>>> a.copy()
>>> a
<Atom: Fe - 0.000, 0.000, 0.000>
>>> a.id
None
```

classmethod `create`(*element*, *coord*, ***kwargs*)
Creates a new Atom object.

Arguments: element (str or Element): Specifies the element of the Atom. coord (iterable of floats): Specifies the coordinate of the Atom.

Keyword Arguments:

forces: Specifies the forces on the atom.

magmom: The magnitude of the magnetic moment on the atom.

charge: The charge on the Atom.

volume: The atomic volume of the atom (Angstroms^3).

Examples:

```
>>> Atom.create('Fe', [0,0,0])
>>> Atom.create('Ni', [0.5, 0.5, 0.5], ox=2, magmom=5,
>>>                                 forces=[0.2, 0.2, 0.2],
>>>                                 volume=101, charge=1.8,
>>>                                 occupancy=1)
```

`forces`
Forces on the Atom in [x, y, z] directions.

`index`
None if not in a `Structure`, otherwise the index of the atom in the structure.

`is_on`(*site*, *tol=0.001*)
Tests whether or not the `Atom` is on the specified `Site`.

Examples:

```
>>> a = Atom.create('Fe', [0,0,0])
>>> s = a.get_site()
>>> a2 = Atom.create('Ni', [0,0,0])
>>> a2.is_on(s)
True
```

species
Formatted Species string. e.g. Fe3+, O2-

4.1.3 Site

class qmpy.**Site**(**args*, ***kwargs*)
A lattice site.

A site can be occupied by one Atom, many Atoms or no Atoms.

Relationships:

`Structure` via structure
`Atom` via atom_set
`WyckoffSite` via wyckoff

Attributes:

id
x, y, z: Coordinate of the Site

add_atom(*atom*, *tol=0.01*)
Adds Atom to *Site.atoms*.

Notes: If the Site being assigned to doens't have a coordinate, it is assigned the coordinate of *atom*.

Arguments: atom (Atom): Atom to add to the structure.

Keyword Arguments:

tol (float): Distance between *atom* and the Site for the Atom to be assigned to the Site. Raises a SiteError if the distance is greater than *tol*.

Raises: SiteError: If *atom* is more than *tol* from the Site.

Examples:

```
>>> s = Site.create([0,0,0])
>>> a = Atom.create('Fe', [0,0,0])
>>> s.add_atom(a)
>>> s2 = Site()
>>> s2.add_atom(a)
```

atoms
List of Atoms on the Site.

cart_coord
Cartesian coordinates of the Atom.

comp
Composition dictionary of the Site.

Returns: dict: of (element, occupancy) pairs.

Examples:

```
>>> a1 = Atom('Fe', [0,0,0], occupancy=0.2)
>>> a2 = Atom('Ni', [0,0,0], occupancy=0.8)
>>> s = Site.from_atoms([a1,a2])
>>> s.comp
{'Fe':0.2, 'Ni':0.8}
```

`coord`
[Site.x, Site.y, Site.z]

static `create` (*coord*, *comp=None*)
Constructs a Site from a coordinate.

Note: The Site is created without any Atoms occupying it.

Arguments:

coord (length 3 iterable): Assigns the x, y, and z coordinates of the Site.

Keyword Arguments:

comp (dict, string, or qmpy.Element): Composition dictionary. Flexible about input forms. Options include: <Element: Fe>, 'Fe', {"Fe":0.5, "Co":0.5}, and {<Element: Ni>:0.5, <Element: Co>:0.5}.

Raises: TypeError: if *comp* isn't a string, `Atom`, `Element`.

Examples:

```
>>> s = Site.create([0.5,0.5,0.5])
```

classmethod `from_atoms` (*atoms*, *tol=0.0001*)
Constructs a Site from an iterable of Atoms.

Notes: Site.coord is set as the average coord of all assigned Atoms.

Checks that the Atoms are close together. If the Atoms are further apart than *tol*, raises a SiteError

Arguments: atoms (iterable of *Atom*): List of Atoms to occupy the Site.

Keyword Arguments: tol (float): Atoms must be within *tol* of each other to be assigned to the same Site. Defaults to 1e-4.

Examples:

```
>>> a1 = Atom.create('Fe', [0,0,0])
>>> a2 = Atom.create('Ni', [1e-5, -1e-5, 0])
>>> s = Site.from_atoms([a1,a1])
```

`label`
Assigns a human friendly label for the Site, based on its atomic composition. If singly occupied, returns the symbol of the atom on the site. If multiply occupied, returns a comma seperated string

Examples:

```
>>> a1 = Atom.create('Fe', [0,0,0], occupancy=0.2)
>>> a2 = Atom.create('Ni', [0,0,0], occupancy=0.8)
>>> s = Site.from_atoms([a1,a2])
```

`magmom`
Calculates the composition weighted average magnetic moment of the atoms on the Site.

Returns: float or None

occupancy
Calculates the total occupancy of the site.

Returns: float or None

ox
Calculates the composition weighted average oxidation state of the atoms on the Site.

Returns: float or None

spec_comp
Composition dictionary of the Site.

Returns: dict: of (species, occupancy) pairs.

Examples:

```
>>> a1 = Atom('Fe', [0,0,0], occupancy=0.2)
>>> a2 = Atom('Ni', [0,0,0], occupancy=0.8)
>>> s = Site.from_atoms([a1,a2])
>>> s.comp
{'Fe':0.2, 'Ni':0.8}
```

4.1.4 Element

class qmpy.**Element**(**args*, ***kwargs*)
Core model for an element.

Relationships:

`Atom` via atom_set
`Species` via species_set
`Structure` via structure_set
`Entry` via entry_set
`Composition` via composition_set
`Calculation` via calculation_set
`Potential` via potential_set
`Hubbard` via hubbards
`HubbardCorrection` via hubbardcorrection_set
`ReferenceEnergy` via referenceenergy_set

Attributes:

Identification
z: atomic number
name: full atomic name
symbol: atomic symbol
group: group in the periodic table
period: period in the periodic table

Physical properties
mass: Atomic mass, in AMU (float)
density: Density at STP, in g/cm^3 (float)
volume: Atomic volume at STP, in A^3/atom (float)
atomic_radii: in A (float)

van_der_waals radii: in A (float)
covalent_radii: in A (float)
scattering_factors: A dictionary of scattering factor coeffs.

Thermodynamic properties
melt: melting point in K
boil: boiling point in K
specific_heat: C_p in J/K

Electronic properties
electronegativity: Pauling electronegativity
ion_energy: First ionization energy. (eV)
s_elec: # of s electrons
p_elec: # of p electrons
d_elec: # of d electrons
f_elec: # of f electrons

Additional information
production: Annual tons of element produced.
abundance: Amount in earths crust (ppm)
radioactive: Are all isotopes unstable?
HHI_P: Herfindahl-Hirschman Index for production.
HHI_R: Herfindahl-Hirschman Index for reserve

Note: HHI values from Gaultois, M. et al. Chem. Mater. 25, 2911-2920 (2013).

classmethod `get` (*value*)
Return an element object. Accepts symbols and atomic numbers, or a list of symbols/atomic numbers.

Examples:

```
>>> Element.get('Fe')
>>> Element.get(26)
>>> Element.get(['Fe', 'O'])
```

class `qmpy.`**`Species`**(**args*, ***kwargs*)
Base model for an atomic species. (Element + charge state).

Relationships:

`Element` via element
`Entry` via entry_set
`Structure` via structure_set

Attributes:

name: Species name. e.g. Fe3+, O2-
ox: Oxidation state (float)

classmethod `get` (*value*)
Gets or creates the specified species.

Arguments:

value: Accepts multiple input types. Can be a string, e.g. Fe3+ or a tuple of (symbol, oxidation state) pairs, e.g. (Fe, 3).

Return: A `Species` or list of `Species`.

Examples:

```
>>> Species.get('Fe3+')
>>> Species.get('Fe3')
>>> Species.get(('Fe', 3))
>>> Species.get([ 'Fe3+', 'O2-', 'Li1+'])
```

4.1.5 Composition

class qmpy.**Composition**(**args*, ***kwargs*)

Base class for a composition.

Relationships:

`Calculation` via calculation_set

`Element` via element_set

`Entry` via entry_set

`ExptFormationEnergy` via exptformationenergy_set

`FormationEnergy` via formationenergy_set

`MetaData` via meta_data

`Structure` via structure_set

`Prototype` via prototype_set

Attributes:

formula: Electronegativity sorted and normalized composition string.
e.g. Fe2O3, LiFeO2

generic: Genericized composition string. e.g. A2B3, ABC2.

mass: Mass per atom in AMUs

meidema: Meidema model energy for the composition

ntypes: Number of elements.

comp

Return an element:amount composition dictionary.

delta_e

Return the lowest formation energy.

experiment

Return the lowest experimantally measured formation energy at the compositoin.

classmethod get(*composition*)

Classmethod for getting Composition objects - if the Composition existsin the database, it is returned. If not, a new Composition is created.

Examples:

```
>>> Composition.get('Fe2O3')
<Composition: Fe2O3>
```

classmethod get_list(*bounds*, *calculated=False*, *uncalculated=False*)

Classmethod for finding all compositions within the space bounded by a sequence of compositions.

Examples:

```
>>> from pprint import pprint
>>> comps = Composition.get_list(['Fe','O'], calculated=True)
>>> pprint(list(comps))
[<Composition: Fe>,
 <Composition: FeO>,
 <Composition: FeO3>,
 <Composition: Fe2O3>,
 <Composition: Fe3O4>,
 <Composition: Fe4O5>,
 <Composition: O>]
```

ground_state
Return the most stable entry at the composition.

icsd_delta_e
Return the lowest formation energy calculated from experimentally measured structures - i.e. excluding prototypes.

ndistinct
Return the number of distinct entries.

space
Return the set of element symbols

unit_comp
Return an element:amoutn composition dictionary normalized to a unit composition.

4.2 Calculation models

4.2.1 Calculation

class qmpy.**Calculation**(**args*, ***kwargs*)
Base class for storing a VASP calculation.

Relationships:

`Composition` via composition
`DOS` via dos
`Structure` via input. Input structure.
`Structure` via output. Resulting structure.
`Element` via element_set.
`Potential` via potential_set.
`Hubbard` via hubbard_set.
`Entry` via entry.
`Fit` via fit. Reference energy sets that have been fit using
this calculation.
`FormationEnergy` via formationenergy_set. Formation
energies computed from this calculation, for different choices of
fit sets.
`MetaData` via meta_data

Attributes:

id

label: key for entry.calculations dict.

attempt: # of this attempt at a calculation.

band_gap: Energy gap occupied by the fermi energy.

configuration: Type of calculation (module).

converged: Did the calculation converge electronically and ionically.

energy: Total energy (eV/UC)

energy_pa: Energy per atom (eV/atom)

irreducible_kpoints: # of irreducible k-points.

magmom: Total magnetic moment (mu_b)

magmom_pa: Magnetic moment per atom. (mu_b/atom)

natoms: # of atoms in the input.

nsteps: # of ionic steps.

path: Calculation path.

runtime: Runtime in seconds.

settings: dictionary of VASP settings.

address_errors()

Attempts to fix any encountered errors.

backup(*path=None*)

Create a copy of the calculation folder in a subdirectory of the current Calculation.

Keyword arguments: path: If None, the backup folder is generated based on the Calculation.attempt and Calculation.errors.

Return: None

compress(*files=['OUTCAR', 'CHGCAR', 'CHG', 'PROCAR', 'LOCPOT', 'ELFCAR']*)

gzip every file in *files*

Keyword arguments: files: List of files to zip up.

Return: None

copy()

Create a deep copy of the Calculation.

Return: None

error_objects

Return list of errors (MetaData objects of type error)

get_outcar()

Sets the calculations outcar attribute to a list of lines from the outcar.

Examples:

```
>>> calc = Calculation.read('calculation_path')
>>> print calc.outcar
None
>>> calc.get_outcar()
>>> len(calc.outcar)
12345L
```

static **read**(*path*)

Reads the outcar specified by the objects path. Populates input field values, as well as outputs, in addition to finding errors and confirming convergence.

Examples:

```
>>> path = '/analysis/vasp/files/normal/standard/'
>>> calc = Calculation.read(INSTALL_PATH+path)
```

read_charges()

Reads and returns VASP's calculated charges for each atom. Returns the RAW charge, not NET charge.

Examples:

```
>>> calc = Calculation.read('path_to_calculation')
>>> calc.read_charges()
```

read_chgcar(*filename='CHGCAR.gz'*, *filetype='CHGCAR'*)

Reads a VASP CHGCAR or ELFCAR and returns a GridData instance.

read_elements()

Reads the elements of the atoms in the structure. Returned as a list of atoms of shape (natoms,).

Examples:

```
>>> calc = Calculation.read('path_to_calculation')
>>> calc.read_elements()
['Fe', 'Fe', 'O', 'O', 'O']
```

read_energies()

Returns a numpy.ndarray of energies over all ionic steps.

Examples:

```
>>> calc = Calculation.read('calculation_path')
>>> calc.read_energies()
array([-12.415236, -12.416596, -12.416927])
```

read_lattice_vectors()

Reads and returns a numpy ndarray of lattice vectors for every ionic step of the calculation.

Examples:

```
>>> path = 'analysis/vasp/files/magnetic/standard'
>>> calc = Calculation.read(INSTALL_PATH+'/'+path)
>>> calc.read_lattice_vectors()
array([[[ 5.707918,  0.      ,  0.      ],
        [ 0.      ,  5.707918,  0.      ],
        [ 0.      ,  0.      ,  7.408951]],
       [[ 5.707918,  0.      ,  0.      ],
        [ 0.      ,  5.707918,  0.      ],
        [ 0.      ,  0.      ,  7.408951]]])
```

read_n_ionic()

Reads the number of ionic steps, and assigns the value to nsteps.

read_natoms()

Reads the number of atoms, and assigns the value to natoms.

set_chgcar(*source*)

Copy the CHGCAR specified by *source* to this calculation.

Arguments: source: can be another `Calculation` instance or a string containing a path to a CHGCAR. If it is a path, it should be a absolute, i.e. begin with "/", and can either end with the CHGCAR or simply point to the path that contains it. For example, if you want to take the CHGCAR from a previous calculation you can do any of:

```
>>> c1 # old calculation
>>> c2 # new calculation
>>> c2.set_chgcar(c1)
>>> c2.set_chgcar(c1.path)
>>> c2.set_chgcar(c1.path+'/CHGCAR')
```

set_wavecar(*source*)

Copy the WAVECAR specified by *source* to this calculation.

Arguments: source: can be another `Calculation` instance or a string containing a path to a WAVECAR. If it is a path, it should be a absolute, i.e. begin with "/", and can either end with the WAVECAR or simply point to the path that contains it. For example, if you want to take the WAVECAR from a previous calculation you can do any of:

```
>>> c1 # old calculation
>>> c2 # new calculation
>>> c2.set_wavecar(c1)
>>> c2.set_wavecar(c1.path)
>>> c2.set_wavecar(c1.path+'/WAVECAR')
```

static **setup**(*structure*, *configuration='static'*, *path=None*, *entry=None*, *hubbard='wang'*, *potentials='vasp_rec'*, *settings={}*, *chgcar=None*, *wavecar=None*, ***kwargs*)

Method for creating a new VASP calculation.

Arguments: structure: `Structure` instance, or string indicating an input structure file.

Keyword Arguments:

configuration: String indicating the type of calculation to perform. Options can be found with qmpy.VASP_SETTINGS.keys(). Create your own configuration options by adding a new file to configuration/vasp_settings/inputs/ using the files already in that directory as a guide. Default="static"

settings: Dictionary of VASP settings to be applied to the calculation. Is applied after the settings which are provided by the *configuration* choice.

path: Location at which to perform the calculation. If the calculation takes repeated iterations to finish successfully, all steps will be nested in the *path* directory.

entry: If the full qmpy data structure is being used, you can specify an entry to associate with the calculation.

hubbard: String indicating the hubbard correctionconvention. Options found with qmpy.HUBBARDS.keys(), and can be added to or altered by editing configuration/vasp_settings/hubbards.yml. Default="wang".

potentials: String indicating the vasp potentials to use. Options can be found with qmpy.POTENTIALS.keys(), and can be added to or altered by editing configuration/vasp_settings/potentials/yml. Default="vasp_rec".

chgcar/wavecar: Calculation, or path, indicating where to obtain an initial CHGCAR/WAVECAR file for the calculation.

warning_objects

Return list of warnings (MetaData objects of type warning)

4.2.2 Density of States

class `qmpy.DOS`(**args*, ***kwargs*)

Electronic density of states..

Relationships:

`Entry` via entry
`MetaData` via meta_data
`Calculation` via calculation

Attributes:

id
data: Numpy array of DOS occupations.
file: Source file.
gap: Band gap in eV.

`energy`
Return the array with the energies.

`read_doscar`(*fname='DOSCAR'*)
Read a VASP DOSCAR file

`site_dos`(*atom*, *orbital*)
Return an NDOSx1 array with dos for the chosen atom and orbital.

atom: int Atom index

orbital: int or str Which orbital to plot

If the orbital is given as an integer: If spin-unpolarized calculation, no phase factors: s = 0, p = 1, d = 2 Spin-polarized, no phase factors: s-up = 0, s-down = 1, p-up = 2, p-down = 3, d-up = 4, d-down = 5 If phase factors have been calculated, orbitals are s, py, pz, px, dxy, dyz, dz2, dxz, dx2 double in the above fashion if spin polarized.

4.2.3 Potential

class `qmpy.Potential`(**args*, ***kwargs*)

Class for storing a VASP potential.

Relationships:

calculation
element

Attributes:

name
date
electrons: Electrons in potential.
enmax
enmin
gw
id
paw
potcar
us

xc

class qmpy.**Hubbard**(**args*, ***kwargs*)
Base class for a hubbard correction parameterization.

Attributes:

calculation
convention
correction
element
id
l
ligand
ox
u

4.3 Thermodynamics models

4.3.1 Formation Energies

class qmpy.**FormationEnergy**(**args*, ***kwargs*)
Base class for a formation energy.

Relationships:

`Calculation` via calculation
`Composition` via composition
`Entry` via entry
`FormationEnergy` via equilibrium
`Fit` via fit

Attributes:

id
delta_e: Formation energy (eV/atom)
description: A label of the source of the formation energy.
stability: Distance from the convex hull (eV/atom)

class qmpy.**ExptFormationEnergy**(**args*, ***kwargs*)
Experimentally measured formation energy.

Any external formation energy should be entered as an ExptFormationEnergy object, rather than a FormationEnergy. If the external source is also computational, set the "dft" attribute to be True.

Relationships:

`Composition` via composition
`Fit` via fit

Attributes:

id: integer primary key.
delta_e: measured formation energy.
delta_g: measured free energy of formation.
dft: (bool) True if the formation energy is from a non-OQMD DFT

calculation.

source: (str) Identifier for the source.

4.3.2 Reference energies

class qmpy.**Fit**(**args*, ***kwargs*)

The core model for a reference energy fitting scheme.

The Fit model links to the experimental data (ExptFormationEnergy objects) that informed the fit, as well as the DFT calculations (Calculation objects) that were matched to each experimental formation energy. Once the fit is completed, it also stores a list of chemical potentials both as a relationship to ReferenceEnergy and HubbardCorrection objects. These correction energies can also be accessed by dictionaries at Fit.mus and Fit.hubbard_mus.

Relationships:

`Calculation` via dft

`ExptFormationEnergy` via experiments

`FormationEnergy` via formationenergy_set

`HubbardCorrection` via hubbard_correction_set

`ReferenceEnergy` via reference_energy_set

Attributes:

name: Name for the fitting

Examples:

```
>>> f = Fit.get('standard')
>>> f.experiments.count()
>>> f.dft.count()
>>> f.mus
>>> f.hubbard_mus
```

class qmpy.**ReferenceEnergy**(**args*, ***kwargs*)

Elemental reference energy for evaluating heats of formation.

Relationships:

`Fit` via fit

`Element` via element

Attributes:

id

value: Reference energy (eV/atom)

class qmpy.**HubbardCorrection**(**args*, ***kwargs*)

Energy correction for DFT+U energies.

Relationships:

`Fit` via fit

`Element` via element

`Hubbard` via hubbard

Attributes:

id

value: Correction energy (eV/atom)

4.3.3 Phase Space

class `qmpy.`**`PhaseSpace`**(*bounds*, *mus=None*, *data=None*, ***kwargs*)

A PhaseSpace object represents, naturally, a region of phase space.

The most fundamental property of a PhaseSpace is its bounds, which are given as a hyphen-delimited list of compositions. These represent the extent of the phase space, and determine which phases are within the space.

Next, a PhaseSpace has an attribute, data, which is a PhaseData object, and is a container for Phase objects, which are used when performing thermodynamic analysis on this space.

The majority of attributes are lazy, that is, they are only computed when they are requested, and how to get them (of which there are often several ways) is decided based on the size and shape of the phase space.

`bound_elements`

Alphabetically ordered list of elements with constrained composition.

`bound_space`

Set of elements _of fixed composition in the PhaseSpace.

Examples:

```
>>> s = PhaseSpace('Fe-Li', 'O=-1.4')
>>> s.bound_space
set(['Fe', 'Li'])
```

`chempot_dimension`

Chemical potential dimension.

Examples:

```
>>> s = PhaseSpace('Fe-Li', 'O=-2.5')
>>> s.chempot_dimension
0
>>> s = PhaseSpace('Fe-Li', 'N=0:-5')
>>> s.chempot_dimension
1
>>> s = PhaseSpace('Fe-Li', 'N=0:-5 F=0:-5')
>>> s.chempot_dimension
2
```

`chempot_scan`()

Scan through chemical potentials of *element* from *umin* to *umax* identifing values at which phase transformations occur.

`clear_all`()

Clears input data and analyzed results. Same as: >>> PhaseData.clear_data() >>> PhaseData.clear_analysis()

`clear_analysis`()

Clears all calculated results.

`clear_data`()

Clears all phase data.

`cliques`

Iterator over maximal cliques in the phase space. To get a list of cliques, use list(PhaseSpace.cliques).

`comp`(*coord*)

Returns the composition of a coordinate in phase space.

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> space.comp([0.2, 0.2, 0.6])
{'Fe': 0.2, 'O': 0.6, 'Li': 0.2}
```

comp_dimension

Compositional dimension of the region of phase space.

Examples:

```
>>> s = PhaseSpace('Fe-Li-O')
>>> s.comp_dimension
2
>>> s = PhaseSpace('FeO-Ni2O-CoO-Ti3O4')
>>> s.comp_dimension
3
```

compute_formation_energies()

Evaluates the formation energy of every phase with respect to the chemical potentials in the PhaseSpace.

compute_stabilities(**args*, ***kwargs*)

Calculate the stability for every Phase.

Keyword Arguments:

phases: List of Phases. If None, uses every Phase in PhaseSpace.phases

save: If True, save the value for stability to the database.

new_only: If True, only compute the stability for Phases which did not import a stability from the OQMD. False by default.

compute_stability(*p*)

Compute the energy difference between the formation energy of a Phase, and the energy of the convex hull in the absence of that phase.

coord(*composition*, *tol=0.0001*)

Returns the barycentric coordinate of a composition, relative to the bounds of the PhaseSpace. If the object isn't within the bounds, raises a PhaseSpaceError.

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> space.coord({'Fe':1, 'Li':1, 'O':2})
array([ 0.25,  0.25,  0.5 ])
>>> space = PhaseSpace('Fe2O3-Li2O')
>>> space.coord('Li5FeO4')
array([ 0.25,  0.75])
```

dual_spaces

List of sets of elements, such that any possible tie-line between two phases in phases is contained in at least one set, and no set is a subset of any other.

elements

Alphabetically ordered list of elements present in the PhaseSpace.

find_reaction_mus(*element=None*)

Find the chemical potentials of a specified element at which reactions occur.

Examples:

```
>>> s = PhaseSpace('Fe-Li-O')
>>> s.find_reaction_mus('O')
```

gclp(*composition={}*, *mus={}*, *phases=[]*)
Returns energy, phase composition which is stable at given composition

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> phases, energy = space.gclp('FeLiO2')
>>> print phases
>>> print energy
```

get_hull_points()
Gets out-of PhaseSpace points. i.e. for FeSi2-Li, there are no other phases in the space, but there are combinations of Li-Si phases and Fe-Si phases. This method returns a list of phases including composite phases from out of the space.

Examples:

```
>>> space = PhaseSpace('FeSi2-Li')
>>> space.get_hull_points()
[<Phase FeSi2 (23408): -0.45110217625>,
<Phase Li (104737): 0>,
<Phase 0.680 Li13Si4 + 0.320 FeSi : -0.3370691816>,
<Phase 0.647 Li8Si3 + 0.353 FeSi : -0.355992801765>,
<Phase 0.133 Fe3Si + 0.867 Li21Si5 : -0.239436904167>,
<Phase 0.278 FeSi + 0.722 Li21Si5 : -0.306877209723>]
```

get_minima(*phases*, *bounds*)
Given a set of Phases, get_minima will determine the minimum free energy elemental composition as a weighted sum of these compounds

get_phase_diagram()
Creates a Renderer attribute with appropriate phase diagram components.

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> space.get_renderer()
>>> plt.show()
```

get_qhull(*phases=None*, *mus={}*)
Get the convex hull for a given space.

get_reaction(*var*, *facet=None*)
For a given composition, what is the maximum delta_composition reaction on the given facet. If None, returns the whole reaction for the given PhaseSpace.

Examples:

```
>>> space = PhaseSpace('Fe2O3-Li2O')
>>> equilibria = space.hull[0]
>>> space.get_reaction('Li2O', facet=equilibria)
```

get_reactions(*var*, *electrons=2.0*)
Returns a list of Reactions.

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> space.get_reactions('Li', electrons=1)
```

get_tie_lines_by_gclp(*iterable=False*)
Runs over pairs of Phases and tests for equilibrium by GCLP. Not recommended, it is very slow.

graph
`networkx.Graph` representation of the phase space.

hull
List of facets of the convex hull.

in_bounds(*composition*)
Returns True, if the composition is within the bounds of the phase space

Examples:

```
>>> space = PhaseSpace('Fe2O3-NiO2-Li2O')
>>> space.in_bounds('Fe3O4')
False
>>> space.in_bounds('Li5FeO8')
True
```

in_space(*composition*)
Returns True, if the composition is in the right elemental-space for this PhaseSpace.

Examples:

```
>>> space = PhaseSpace('Fe-Li-O')
>>> space.in_space('LiNiO2')
False
>>> space.in_space('Fe2O3')
True
```

load(***kwargs*)
Loads oqmd data into the associated PhaseData object.

make_1d_vs_chempot(***kwargs*)
Plot of phase stability vs chemical potential for a single composition.

Examples:

```
>>> s = PhaseSpace('Fe', mus={'O':[0,-4]})
>>> r = s.make_vs_chempot()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

make_as_binary(***kwargs*)
Construct a binary phase diagram (convex hull) and write it to a `Renderer`.

Examples:

```
>>> s = PhaseSpace('Fe-P')
>>> r = s.make_as_binary()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

make_as_graph(***kwargs*)
Construct a graph-style visualization of the phase diagram.

make_as_quaternary(***kwargs*)
Construct a quaternary phase diagram and write it to a `Renderer`.

Examples:

```
>>> s = PhaseSpace('Fe-Li-O-P')
>>> r = s.make_as_quaternary()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

make_as_ternary(***kwargs*)

Construct a ternary phase diagram and write it to a `Renderer`.

Examples:

```
>>> s = PhaseSpace('Fe-Li-O-P')
>>> r = s.make_as_quaternary()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

make_as_unary(***kwargs*)

Plot of phase volume vs formation energy.

Examples:

```
>>> s = PhaseSpace('Fe2O3')
>>> r = s.make_as_unary()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

make_vs_chempot(***kwargs*)

Plot of phase stability vs chemical potential for a range of compositions.

Examples:

```
>>> s = PhaseSpace('Fe-Li', mus={'O':[0,-4]})
>>> r = s.make_vs_chempot()
>>> r.plot_in_matplotlib()
>>> plt.show()
```

phase_diagram

Renderer of a phase diagram of the PhaseSpace

plot_reactions(*var*, *electrons=2.0*, *save=False*)

Plot the convex hull along the reaction path, as well as the voltage profile.

save_tie_lines()

Save all tie lines in this PhaseSpace to the OQMD. Stored in Formation.equilibrium

shape

(# of compositional dimensions, # of chemical potential dimensions) The shape attribute of the PhaseSpace determines what type of phase diagram will be drawn.

Examples:

```
>>> s = PhaseSpace('Fe-Li', 'O=-1.2')
>>> s.shape
(1, 0)
>>> s = PhaseSpace('Fe-Li', 'O=0:-5')
>>> s.shape
(1, 1)
>>> s = PhaseSpace('Fe-Li-P', 'O=0:-5')
>>> s.shape
(2,1)
>>> s = PhaseSpace('Fe', 'O=0:-5')
>>> s.shape
(0, 1)
```

space

Set of elements present in the PhaseSpace.

Examples:

```
>>> s = PhaseSpace('Pb-Te-Se')
>>> s.space
set(['Pb', 'Te', 'Se'])
>>> s = PhaseSpace('PbTe-Na-PbSe')
>>> s.space
set(['Pb', 'Te', 'Na', 'Se'])
```

spaces

List of lists of elements, such that every phase in self.phases is contained in at least one set, and no set is a subset of any other. This corresponds to the smallest subset of spaces that must be analyzed to determine the stability of every phase in your dataset.

Examples:

```
>>> pa, pb, pc = Phase('A', 0), Phase('B', 0), Phase('C', 0)
>>> p1 = Phase('AB2', -1)
>>> p2 = Phase('B3C', -4)
>>> s = PhaseSpace('A-B-C', load=None)
>>> s.phases = [ pa, pb, pc, p1, p2 ]
>>> s.spaces
[['C', 'B'], ['A', 'B']]
```

stability_range(*p*, *element=None*)

Calculate the range of phase *p* with respect to *element*.

stable

List of stable phases

tie_lines

List of length 2 tuples of phases with tie lines between them

unstable

List of unstable phases.

class qmpy.**PhaseData**

A PhaseData object is a container for storing and organizing phase data. Most importantly used when doing a large number of thermodynamic analyses and it is undesirable to access the database for every space you want to consider.

add_phase(*phase*)

Add a phase to the PhaseData collection. Updates the PhaseData.phase_dict and PhaseData.phases_by_elt dictionaries appropriately to enable quick access.

Examples:

```
>>> pd = PhaseData()
>>> pd.add_phase(Phase(composition='Fe2O3', energy=-3))
>>> pd.add_phase(Phase(composition='Fe2O3', energy=-4))
>>> pd.add_phase(Phase(composition='Fe2O3', energy=-5))
>>> pd.phase_dict
{'Fe2O3': <Phase Fe2O3 : -5}
>>> pd.phases_by_elt['Fe']
[<Phase Fe2O3 : -3>, <Phase Fe2O3 : -4>, <Phase Fe2O3 : -5>]
```

add_phases(*phases*)

Loops over a sequence of phases, and applies *add_phase* to each.

Equivalent to:

```
>>> pd = PhaseData()
>>> for p in phases:
>>>     pd.add_phase(p)
```

dump (*filename=None*, *minimal=True*)
Writes the contents of the phase data to a file or to stdout.

Keyword Arguments:

filename: If None, prints the file to stdout, otherwise writes the file to the specified filename. Default=None.

minimal: Dump _every_ phase in the PhaseData object, or only those that can contribute to a phase diagram. If True, only the lowest energy phase at a given composition will be written. Default=True.

get_phase_data (*space*)
Using an existing PhaseData object return a PhaseData object which is populated by returning a subset which is inside a given region of phase space.

Arguments: space: formatted as in `qmpy.PhaseSpace.__init__()`

Examples:

```
>>> pd = PhaseData()
>>> pd.read_file('legacy.dat')
>>> new_pd = pd.get_phase_data(['Fe', 'O'])
>>> new_pd.phase_dict
```

load_library (*library*)
Load a library file, containing self-consistent thermochemical data.

Equivalent to:

```
>>> pd = PhaseData()
>>> pd.read_file(INSTALL_PATH+'/data/thermodata/%s' % library)
```

load_oqmd (*space=None*, *search={}*, *exclude={}*, *stable=False*, *fit='standard'*, *total=False*)
Load data from the OQMD.

Keyword Arguments:

space: sequence of elements. If supplied, will return only phases within that region of phase space. i.e. ['Fe', 'O'] will return Fe, O and all iron oxides.

search: dictionary of database search keyword:value pairs.

stable: Restrict search to only stable phases (faster, but relies on having current phase stability analyses).

Examples:

```
>>> pd = PhaseData()
>>> search = {'calculation__path__contains':'icsd'}
>>> pd.load_oqmd(space=['Fe','O'], search=search, stable=True)
```

phases
List of all phases.

read_file (*filename*, *per_atom=True*)
Read in a thermodata file (named filename).

File format:

```
composition energy
Fe 0.0
O 0.0
Li 0.0
Fe3O4 -0.21331204979
FeO -0.589343204057
Fe3O4 -0.21331204979
FeLiO2 -0.446739168889
FeLi5O4 -0.198830531099
```

Keyword Arguments: per_atom: If True, the supplied energies are per atom, not per formula unit. Defaults to True.

4.4 Database models

4.4.1 Entries

class `qmpy.`**`Entry`**`(`**args*, ***kwargs*`)`

Base class for a database entry.

The core model for typical database entries. An Entry model represents an input structure to the database, and can be created from any input file. The Entry also ties together all of the associated `qmpy.Structure`, `qmpy.Calculation`, `qmpy.Reference`, `qmpy.FormationEnergies`, and other associated databas entries.

Relationships:

`Calculation` via calculation_set
`DOS` via dos_set
`Entry` via duplicate_of
`Entry` via duplicates
`Element` via element_set
`FormationEnergy` via formationenergy_set
`Job` via job_set
`MetaData` via meta_data
`Project` via project_set
`Prototype` via prototype
`Species` via species_set
`Structure` via structure_set
`Task` via task_set
`Reference` via reference
`Composition` via composition

Attributes:

id: Primary key (auto-incrementing int)
natoms: Number of atoms in the primitive input cell
ntypes: Number of elements in the input structure
path: Path to input file, and location of subsequent calculations.
label: An identifying name for the structure. e.g. icsd-1001 or A3

calculations
Dictionary of label:Calculation pairs.

chg
Attempts to load the charge density of the final calculation, if it is done. If not, returns False.

static create (*source*, *keywords*=[ ], *projects*=[ ], ***kwargs*)
Attempts to create an Entry object from a provided input file.

Processed in the following way:

1. If an Entry exists at the specified path, returns that Entry.
2. Create an Entry, and assign all fundamental attributes. (natoms, ntypes, input, path, elements, keywords, projects).
3. If the input file is a CIF, and because CIF files have additional composition and reference information, if that file format is found, an additional test is performed to check that the reported composition matches the composition of the resulting structure. The reference for the work is also created and assigned to the entry.
4. Attempt to identify another entry that this is either exactly equivalent to, or a defect cell of.

Keywords: keywords: list of keywords to associate with the entry. projects: list of project names to associate with the entry.

do (*module*, **args*, ***kwargs*)
Looks for a computing script matching the first argument, and attempts to run it with itself as the first argument. Sends args and kwargs to the script. Should return a Calculation object, or list of Calculation objects.

Examples:

```
>>> e = Entry.objects.get(id=123)
>>> e.do('relaxation')
<Calculation: 523 @ relaxation settings>
```

elements
List of Elements

energy
If the structure has been relaxed, returns the formation energy of the final relaxed structure. Otherwise, returns None.

errors
List of errors encountered in all calculations.

hold_objects
Return list of holds (MetaData objects of type hold)

holds
A note indicating a reason the entry should not be calculated

html
HTML formatted name

keyword_objects
Return list of keywords (MetaData objects of type keyword)

keywords
Descriptive keyword for looking up entries

latex
LaTeX formatted name

mass
Return the mass of the entry, normalized to per atom.

move (**args*, ***kwargs*)
Moves all calculation files to the specified path.

name
Unformatted name

projects
List of Projects

red_comp
Composition dictionary, in reduced form.

reset ()
Deletes all calculations, removes all associated structures - returns the entry to a pristine state.

save (**args*, ***kwargs*)
Saves the Entry, as well as all associated objects.

space
Return the set of elements in the input structure.

Examples:

```
>>> e = Entry.create("fe2o3/POSCAR") # an input containing Fe2O3
>>> e.space
set(["Fe", "O"])
```

spec_comp
Composition dictionary, using species (element + oxidation state) instead of just the elements.

species
List of Species

total_energy
If the structure has been relaxed, returns the formation energy of the final relaxed structure. Otherwise, returns None.

unit_comp
Composition dictionary, normalized to 1 atom.

visualize (*structure='source'*)
Attempts to open the input structure for visualization using VESTA

volume
If the entry has gone through relaxation, returns the relaxed volume. Otherwise, returns the input volume.

class qmpy.**MetaData** (**args*, ***kwargs*)
Base class for variable typed model tagging.

Model for arbitrary meta-data descriptors for various qmpy objects. Generally accessed by properties and methods added by the "add_label" descriptor. See "add_label" for a more detailed description of its use

Relationships

`Calculation` via calculation_set
`Composition` via composition_set
`DOS` via dos_set

`Entry` via entry_set
`Structure` via structure_set

Attributes:

id: Autoincrementing primary key
type: Label for the kind of meta data, e.g. "hold", "keyword"
value: Content of the meta data. e.g. "repeated failure", "known
anti-ferromagnetic"

Examples:

```
>>> MetaData.get('Keyword', 'ICSD')
<Keyword: ICSD>
```

4.4.2 References

class qmpy.**Reference**(**args*, ***kwargs*)
Base class for a reference to a publication.

Relatioships:

`Author` via author_set
`Journal` via journal
`Entry` via entry_set
`Structure` via structure_set

Database fields:

id
title
year
volume
page_first
page_last

4.4.3 Authors

class qmpy.**Author**(**args*, ***kwargs*)
Base class for an author.

Relationships:

`Reference` via references

Database Fields:

id
first
last

4.4.4 Journals

class qmpy.**Journal**(**args*, ***kwargs*)
Base class for a journal

Relationships:

`Reference` via references

Database fields:

id
name
code

4.5 Symmetry models

4.5.1 Spacegroup

class `qmpy.`**`Spacegroup`**(**args*, ***kwargs*)

Base class for a space group.

Relationships:

`Structure` via structure_set
`Translation` via centering_vectors
`Operation` via operations
`WyckoffSite` via site_set

Attributes:

number: Spacegroup #. (primary key)
centrosymmetric: (bool) Is the spacegroup centrosymmetric.
hall: Hall symbol.
hm: Hermann-Mauguin symobl.
lattice_system: Cubic, Hexagonal, Tetragonal, Orthorhombic,
    Monoclinic or Triclinic.
pearson: Pearson symbol
schoenflies: Schoenflies symbol.

`get_site`(*symbol*)

Gets WyckoffSite by symbol.

`rotations`

List of rotation operations for the spacegroup.

`sym_ops`

List of (rotation, translation) pairs for the spacegroup

`symbol`

Returns the Hermann-Mauguin symbol for the spacegroup

`translations`

List of translation operations for the spacegroup.

`wyckoff_sites`

List of WyckoffSites.

4.5.2 Wyckoff Site

class `qmpy.`**`WyckoffSite`**(**args*, ***kwargs*)

Base class for a Wyckoff site. (e.g. a "b" site).

Relationships:

`Spacegroup` via spacegroup
`Atom` via atom_set
`Site` via site_set

Attributes:

id
symbol: Site symbol
multiplicity: Site multiplicity
x, y, z: Coordinate symbols.

4.5.3 Symmetry Operations

class `qmpy.`**`Operation`**(**args*, ***kwargs*)

A symmetry operation (rotation + translation).

Relationships:

`Spacegroup` via spacegroup
`Rotation` via rotation_set
`Translation` via translation_set

Attributes:

id

Examples:

```
>>> op = Operation.get('x+y-1/2,-z-y+1/2,x-z+1/2')
>>> print op
<Operation: +x+y+1/2,-y-z+1/2,+x-z+1/2>
```

classmethod **`get`**(*value*)

Accepts symmetry operation strings, i.e. "+x, x+1/2, x+y-z" or a tuple of rotation matrix and translation vector.

Example:

```
>>> Operation.get("x,y,-y")
>>> Operation.get(( rot, trans ))
```

class `qmpy.`**`Translation`**(**args*, ***kwargs*)

A translation operation.

Relationships:

`Spacegroup` via spacegroup
`Operation` via operation

Attributes:

id
x, y, z: Translation vector. Accessed via *vector*.

Examples:

```
>>> op = Operation.get('x', 'x+y', 'z-x+1/2')
>>> print op.translsation
<Translation: 0,0,+1/2>
>>> print op.translation.vector
array([ 0. ,  0. ,  0.5])
```

class qmpy.**Rotation**(**args*, ***kwargs*)
A rotation operation.

Relationships:

Spacegroup via spacegroup
Operation via operation

Attributes:

id
a11, a12, a13
a21, a22, a23
a31, a32, a33: Rotation matrix. Accessed via *matrix*.

Examples:

```
>>> op = Operation.get('x', 'x+y', 'z-x+1/2')
>>> print op.rotation
<Rotation: x,x+y,-x+z>
>>> print op.rotation.matrix
array([[ 1.,  0.,  0.],
       [ 1.,  1.,  0.],
       [-1.,  0.,  1.]])
```

4.6 Resource models

4.6.1 Host

class qmpy.**Host**(**args*, ***kwargs*)
Host model - stores all host information for a cluster.

Relationships:

account
allocation

Attributes:

name: Primary key.
binaries: dict of label:path pairs for vasp binaries.
check_queue: Path to showq command
checked_time: datetime object for the last time the queue was
checked.
hostname: Full host name.
ip_address: Full ip address.
nodes: Total number of nodes.
ppn: Number of processors per node.

running: dict of PBS_ID:state pairs.

sub_script: Path to qsub command

sub_text: Path to queue file template.

utilization: Number of active cores (based on showq).

walltime: Maximum walltime on the machine.

state: State code. 1=Up, 0=Full (auto-resets to 1 when jobs are collected), -1=Down.

`check_host``()`
Pings the host to see if it is online. Returns False if it is offline.

`check_running``()`
Uses the hosts data and one of the associated accounts to check the PBS queue on the Host. If it has been checked in the last 2 minutes, it will return the previously returned result.

static `create``()`
Classmethod to create a Host model. Script will ask you questions about the host to add, and will return the created Host.

4.6.2 Account

class `qmpy.`**`Account`**`(`**args*, ***kwargs*`)`
Base class for a *User* account on a *Host*.

Attributes:

host

id

job

run_path

state

user

username

4.6.3 User

class `qmpy.`**`User`**`(`**args*, ***kwargs*`)`
User model - stores an oqmd users information.

Relationships:

`Account` via account_set

`Allocation` via allocation_set

`Project` via project_set

Attributes:

id

username

first_name

last_name

date_joined

is_active

is_staff

is_superuser
last_login
email

4.6.4 Allocation

class `qmpy.`**`Allocation`**`(`**args*, ***kwargs*`)`
Base class for an Allocation on a computing resources.

Attributes:

host
job
key
name
project
state
users

4.6.5 Project

class `qmpy.`**`Project`**`(`**args*, ***kwargs*`)`
Base class for a project within qmpy.

Attributes:

allocations
entry
name
priority
state
task
users

4.7 Queue models

4.7.1 Task

class `qmpy.`**`Task`**`(`**args*, ***kwargs*`)`
Model for a :Task: to be done.

A :Task: consists of a module, which is the name of a computing script, and a set of keyword arguments, specified as a dictionary as the *kwargs* attribute of the task. In order for a Task for be completed, it must also be assigned one or more :Project:s.

Relationships:

`Entry` via entry
`Job` via job_set
`Project` via project_set

Attributes:

id
created: datetime object for when the task was created.
finished: datetime object for when the task was completed.
module: The name of a function in `scripts`
kwargs: dict of keyword:value pairs to pass to the calculation
module.
priority: Priority of the task. Lower values are more urgent.
state: State code, given by the table below.

Task codes:

Code	Description
-2	being held
-1	encountered error
0	ready to run
1	jobs running
2	completed

`complete`()
Sets the Task state to 2 and populates the finished field.

`errors`
List of errors encountered by related calculations.

`get_jobs`(*project=None*, *allocation=None*, *account=None*, *host=None*)
Check the calculation module specified by the *Task*, and returns a list of `Job` objects accordingly.

Calls the task's entry's "do" method with the *Task.module* as the first argument, and passing *Task.kwargs* as keyword arguments.

Returns: List of Job objects. When nothing is left to do for the task, returns empty.

Raises:

ResourceUnavailableError: Raise if for the specified project, allocation, account and/or host there are no available cores.

`jobs`
List of jobs related to the task.

`projects`
List of related projects.

4.7.2 Job

class `qmpy.`**`Job`**(**args*, ***kwargs*)
Base class for job submitted to a compute cluster.

Relationships:

`Task` via task
`Account` via account. The account the calculation is
performed on.
`Allocation` via allocation. The allocation on which the
calculation is being performed.
`Entry` via entry

Attributes:

id
created: datetime object for when the task was created.
finished: datetime object for when the task was completed.
ncpus: # of processors assigned.
path: Origination path of the calculation.
run_path: Path of the calculation on the compute resource.
qid: PBS queue ID number.
walltime: Max walltime (in seconds).
state: State code, defined as in the table below.

Job codes

Code	Description
-1	encountered error
0	ready to submit
1	currently running
2	completed

4.8 Analysis Tools

class `qmpy.`**`PDF`**`(`*structure*, *limit=6*`)`
Container class for a Pair-distribution function.

Attributes: structure: `Structure` pairs: dict of (atom1, atom2):[distances] limit: maximum distance

`get_pair_distances``()`
Loops over pairs of atoms that are within radius max_dist of one another. Returns a dict of (atom1, atom2):[list of distances].

class `qmpy.`**`XRD`**`(`*structure=None*, *measured=False*, *wavelength=1.5418*, *min_2th=10*, *max_2th=60*, *resolution=0.01*`)`
Container for an X-ray diffraction pattern.

Attributes:

peaks (List): List of `Peak` instances.

measured (bool): True if the XRD is a measured pattern, otherwise False.

min_2th (float): Minimum 2theta angle allowed. Defaults to 60 degrees.

max_2th (float): Maximum 2theta angle allowed. Defaults to 10 degrees.

wavelength (float): X-ray wavelength. Defaults to 1.5418 Ang.

resolution (float): Minimum 2theta angle the XRD will distinguish between.

`get_intensities``(`*bfactors=None*, *scale=None*`)`
Loops over all peaks calculating intensity.

Keyword Arguments:

bfactors (list) [list of B factors for each atomic site. Care must] taken to ensure that the order of B factors agrees with the order of atomic orbits.

scale (float) [Scaling factor to multiply the intensities by. If] scale evaluates to False, the intensities will be re-normalized at the end such that the highest peak is 1.

`get_peaks``()`

class `qmpy.Peak`(*angle*, *multiplicity=None*, *intensity=None*, *hkl=None*, *xrd=None*, *width=None*, *measured=False*)

Attributes:

angle (float): Peak 2*theta angle in radians.

hkl (list): HKL indices of the peak.

multiplicity (int): Number of HKL indices which generate the peak.

`lp_factor()`

Calculates the Lorentz-polarization factor.

http://reference.iucr.org/dictionary/Lorentz%E2%80%93polarization_correction

`thermal_factor`(*bfactor=1.0*)

Calculates the Debye-Waller factor for a peak.

http://en.wikipedia.org/wiki/Debye-Waller_factor

class `qmpy.Miedema`(*composition*)

`H_form_ord()`

Calculate the enthalpy of formation for an ordered compound of elements A and B with a composition xB of element B.

`P`

Chooses a value of P based on the transition metal status of the elements A and B.

There are 3 values of P for the cases where: both A and B are TM only one of A and B is a TM neither are TMs.

`RtoP`

Calculate and return the value of RtoP based on the transition metal status of elements A and B, and the elemental values of RtoP for elements A and B.

`gamma`

Calculate and return the value of Gamma_AB (= Gamma_BA) for the solvation of element A in element B.

`pick_a`(*elt*)

Choose a value of a based on the valence of element A.

class `qmpy.GridData`(*data*, *lattice=None*)

Container for 3d data, e.g. charge density or electron localization function.

`find_min_coord`(*N=1*)

Find the *N* lowest valued indices.

`ind_to_cart`(*ind*)

Converts an [i,j,k] index to [X,Y,Z] cartesian coordinate.

`ind_to_coord`(*ind*)

Converts an [i,j,k] index to [x,y,z] frational coordinate.

`interpolate`(*point*, *cart=False*)

Calculates the value at *point* using trilinear interpolation.

Arguments: point: point to evaluate the value at.

Keyword Arguments: cart: If True, the point is taken as a cartesian coordinate. If not, it is assumed to be in fractional coordinates. default=False.

local_min(*index*)
Starting from *index* find the local value minimum.

Returns: index: shape (3,) index of local minimum. value: Value of grid at the local minimum.

path(*origin*, *end*)
Gets a 1D array of values for a line connecting *origin* and *end*.

slice(*point*, *orientation*)
Return a 2D array of values for a slice through the GridData passing through *point* with normal vector *orientation*.

class qmpy.**SpinLattice**(*pairs*)

attempt_flip()
Randomly selects a lattice point, and attempts to flip it.

dE = 2*J*sum(neighboring spins)

compute_total_lattice_energy()
Compute the total energy of the lattice using the Ising model hamiltonian:

H(s) = -J * sum_{i, j}(s_i * s_j)

So, for a positive interaction, J, the energy is minimized when all pairs are alike. Likewise, when J is negative, the enegy is minimized when all pairs are unlike.

run_GCMC(*mu=0*)
Run Monte Carlo in the Grand Canonical Ensemble.

Examples: >>> sl = SpinLattice.create_2d(10) >>> sl.run_GCMC() >>> sl.run_GCMC(-1) >>> sl.run_GCMC(1)

run_MC(*x=None*)
Run Monte Carlo in the Canonical Ensemble

Examples:

```
>>> sl = SpinLattice.create_2d(10)
>>> sl.run_MC()
>>> sl.run_MC(0.1)
>>> sl.run_MC(0.25)
```

qmpy.analysis.nearest_neighbors.**find_nearest_neighbors**(*structure*, *method='closest'*, *limit=5*, *tol=0.2*)
For each atom in the *structure* assign the nearest neighbors.

Keyword Arguments:

method ('closest' or 'voronoi'):

closest: Atoms A and B are neighbors, if and only if there is no atom C such that AC < AB and BC < AB. Once all pairs of this kind have been assigned, the nearest neighbors are those within *tol* of the shortest distance.

voronoi: Assign nearest neighbors based on voronoi construction. For each atom which generates a voronoi facet, *tol* specifies the minimum area of the facet before the atom is considered a nearest neighbor.

defaults to 'closest'

limit: Range outside of the unit cell that will be searched for nearest neighbor atoms.

tol: Varied depending on the method being used.

Returns: dict of Atom:list of Atom pairs. For each atom in the structure, its "neighbors" attribute will be set to the list of its nearest neighbors.

Note: Recommended to use the "closest" method unless you are sure you what the "voronoi" method will do. The voronoi neighbors are useful for some purposes, but are often not what are normally considered nearest neighbors. For example in BCC the second nearsest neighbors contribute to facets in the an atoms voronoi cell. The tolerance specification for this method sets the minimum area of such a facet required to add the atom as a nearest neighbor. In the BCC case, this cutoff must be set to at least 2.3 A^2 to exclude the facets due to second nearest neighbors.

4.9 Renderer

class `qmpy.utils.rendering.`**`Renderer`**(*format='matplotlib'*, *lines=[ ]*, *points=[ ]*, *point_collections=[ ]*, *text=[ ]*, ***kwargs*)

class `qmpy.utils.rendering.`**`Text`**(*pt*, *text*, ***kwargs*)

class `qmpy.utils.rendering.`**`Line`**(*pts*, *label=None*, *fill=False*, ***kwargs*)

class `qmpy.utils.rendering.`**`PointCollection`**(*points*, *label=None*, *fill=False*, ***kwargs*)

class `qmpy.utils.rendering.`**`Point`**(*coord*, *label=None*, ***kwargs*)

class `qmpy.utils.rendering.`**`Axis`**(*name*, *label=''*, *units=''*, *template='{label} [{units}]'*, ***kwargs*)

CHAPTER

FIVE

USING QMPY TO MANAGE A HIGH-THROUGHPUT CALCULATION PROJECT

qmpy can manage almost every aspect of a high-throughput materials screening project. In this section we will walk through all of the necessary steps to get a new project off the ground in a new installation. This tutorial assumes that you have a functional installation of qmpy, as well as a working database (does not need to have a copy of the OQMD included, a blank slate database will work fine).

In this tutorial we will undertake to explore a wide range of (CH3NH3)PbI3 perovskites, which have recently recieved much attention as high-efficiency photovoltaic cells. We will run through the process of setting up compute resources, constructing simple lattice decorations of this structure, as well as some not-so-simple decorations, running the calculations, and finally evaluating the relative stability of the resulting energies.

5.1 Setting up computational resources

We will begin with configuring qmpy to find the right computational resources to be able to perform calculations using qmpy. First, you need to establish the computational resources that are available to it: a `Account`, which is a `Host` paired with a `User`. A `Account` is then granted access to at least one `Allocation`.

Warning: In the current implementation, qmpy is only able to run calculations on clusters that utilize PBS/torque and Maui.

Lets start by configuring a host. Lets assume that we are running qmpy from one cluster, but we want to do our calculations on another machine. Lets edit the resources/hosts.yml file (inside the qmpy installation):

```
# hosts.yml
bigcluster:
  binaries: {vasp_53: /usr/local/bin/vasp_53}
  check_queue: /usr/local/maui/bin/showq
  hostname: big.cluster.edu
  ip_address: XXX.XX.XX.XXX
  nodes: 100
  ppn: 12
  sub_script: /usr/local/bin/qsub
  sub_text: bigcluster.q
  walltime: 691200
```

In this configuration file, we are creating a list of dictionaries. Each outer loop entry creates a `Host`.

Important attributes to be aware of:

- binaries is a dictionary of binary name -> binary path pairs. By default, qmpy calculations try to run vasp_53, and expects a path to a vasp 5.3.3 binary, but should be reliable for most vasp 5.X versions.

- sub_script is the path on the cluster to pbs qsub command
- check_queue is the path on the cluster to maui's showq command
- sub_text is the name of a file in qmpy/configuration/qfiles. An example qfile template is shown below.
- ppn = # of processors / node
- nodes = # of nodes you want qmpy to be able to use (does not need to match the number of nodes on the cluster)

Note: Note that these files must be parseable YAML format files. See *this guide <http://www.yaml.org/start.html>* _ for an introduction to the YAML format.

The queue files that qmpy submits must be tailored both to the job being submitted and the cluster being submitted to. To that end, qmpy uses a simple template system, with the most basic template (that should work in many cases) is:

```
#!/bin/bash
#MSUB -l nodes={nodes}:ppn={ppn}
#MSUB -l walltime={walltime}
#MSUB -N {name}
#MSUB -o jobout.txt
#MSUB -e joberr.txt
#MSUB -A {key}

cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
#running on {host}

{header}

{mpi} {binary} {pipes}

{footer}
```

The "{variable}" construction is used to automatically replace strings based on the calculation requirements. Some variables (nodes, ppn, name, walltime, host) are fairly constant for the host. Others, like "key", specify which allocation to charge the hours to, which is defined by the `Allocation` associated with the calculation. Finally, the rest of the variables are set based on the requirements of the calculation. For general calculations, the header variable is used to unzip any zipped files in the folder (e.g., CHGCAR), the for parallel calculations the mpi variable contains the mpirun + argumnts command and for serial calculations it is left blank. The binary variable will be replaced with the path to the binary, as defined in the hosts.yml file. The pipes variable will pipe stdout and stderr, which by default is always to stdout.txt and stderr.txt. Finally, footer zips the CHGCAR, OUTCAR, PROCAR and ELFCAR, if they exist.

and resources/hosts.yml:

```
# users.yml
oqmdrunner:
  bigcluster: {run_path: /home/oqmdrunner, username: oqmdrunner}

oqmduser:
  bigcluster: {run_path: /home/oqmduser/rundir, username: oqmduser}
  smallcluster: {run_path: /home/oqmduser/rundir, username: oqmduser}
```

loop entry creates a `User` with that name, and each cluster listed then creates an `Account` for that `User`, with username (given by username) and configured to run calculations in run_path. Here we are assuming that a second non-compute cluster, smallcluster, was also defined in hosts.yml.

Warning: Passwordless ssh must be configured to each account (either as a user:host pair, or host-based authentication) from the account you are running qmpy on. The `Account` class has a create_passwordless_ssh method that can set this up for you, however, this process can be unreliable, so if it fails you will need to sort those problems out for yourself.

Next, we configure our allocations, using the allocations.yml file:

```
# allocations.yml
bigcluster:
  host: bigcluster
  users: [oqmdrunner, oqmduser]
  key: alloc1234
```

An allocation takes a host, a list of users and an optional key. The host and list of users are used to determine who is allowed to run calculations on the allocation, while the key is used to identify the allocation to moab, if that feature is implemented.

Finally, we can create a `Project`, defined in projects.yml:

```
# projects.yml
example:
  allocations: [bigcluster]
  priority: 0
  users: [oqmdrunner]
```

We title the project "example", (since it is just an example) and then define the lists of allocations that this project is authorized to use, and the users that are associated with the project. In order to apply these changes, run:

```
>>> from qmpy import *
>>> sync_resources()
```

5.2 Working with Structures

If, for example, we say that we want to calculate i) a range of defects in a host matrix or ii) a wide range of compositions in a particular structure. It is possible to do this in the framework of qmpy, with minimal effort. Sadly, our starting point, the CH3NH3PbI3 (since CH3NH3 is methylammonium, let us call the compound MaPbI3 from now on) is not fully resolved in XRD. The best structure I can find only has the Pb, C, N and I sites determined. So, lets take this structure (reproduced here from Constantinos C. Stoumpos; et. al., Inorganic Chemistry 52 (2013) 9019-9038):

```
I C Pb N
 1.0
8.849000 0.000000 0.000000
0.000000 8.849000 0.000000
0.000000 0.000000 12.642000
C I N Pb
4 12 4 4
direct
 0.5000000000 0.0000000000 0.3520000000
 0.0000000000 0.5000000000 0.3520000000
 0.5000000000 0.0000000000 0.8520000000
 0.0000000000 0.5000000000 0.8520000000
 0.2858300000 0.2141700000 0.0046000000
 0.7858300000 0.2858300000 0.0046000000
 0.2141700000 0.7141700000 0.0046000000
 0.7141700000 0.7858300000 0.0046000000
 0.0000000000 0.0000000000 0.2472000000
 0.5000000000 0.5000000000 0.2472000000
 0.7141700000 0.2141700000 0.5046000000
 0.2141700000 0.2858300000 0.5046000000
 0.7858300000 0.7141700000 0.5046000000
 0.2858300000 0.7858300000 0.5046000000
 0.0000000000 0.0000000000 0.7472000000
 0.5000000000 0.5000000000 0.7472000000
 0.5000000000 0.0000000000 0.2420000000
 0.0000000000 0.5000000000 0.2420000000
 0.5000000000 0.0000000000 0.7420000000
 0.0000000000 0.5000000000 0.7420000000
 0.0000000000 0.0000000000 0.0000000000
 0.5000000000 0.5000000000 0.0000000000
 0.0000000000 0.0000000000 0.5000000000
 0.5000000000 0.5000000000 0.5000000000
```

Now, to populate this structure with hydrogen atoms, lets write a small script to add atoms to the C and N sites. We can see from the POSCAR that the C-N pairs are always oriented along the z-direction, so we will add 3 hydrogen atoms “above” each C and “below” each N.:

```
>>> n_h_bond = 1.01 #A
>>> c_h_bond = 1.09 #A
>>>
>>> s = io.read('POSCAR') # just reading in original structure
>>> for atom in s:
        if atom.element.symbol == 'C':
            x = np.sin(np.pi/4)*c_h_bond
            x2 = np.sin(np.pi/4)*x
            ref = atom.cart_coord
            s.add_atom(Atom.create('H', s.get_coord(ref+[x,   0.0, x2])))
            s.add_atom(Atom.create('H', s.get_coord(ref+[-x2, -x2, x2])))
            s.add_atom(Atom.create('H', s.get_coord(ref+[-x2,  x2, x2])))
        elif atom.element.symbol == 'N':
            x = np.sin(np.pi/4)*c_h_bond
            x2 = np.sin(np.pi/4)*x
            # Note the angles of the N's attached hydrogens are rotated
            # relative to the C's attached hydrogens.
            s.add_atom(Atom.create('H', s.get_coord(ref+[-x, 0.0, -x2])))
            s.add_atom(Atom.create('H', s.get_coord(ref+[x2,  x2, -x2])))
            s.add_atom(Atom.create('H', s.get_coord(ref+[x2, -x2, -x2])))

>>> io.poscar.write(s, 'POSCAR_mod')
```

You can verify that the created structure has the right bond lengths:

```
>>> s = io.read('POSCAR_mod')
>>> for a1, a2 in itertools.combinations(s.atoms, r=2):
        d = s.get_distance(a1, a2)
        elts = set([ a1.element.symbol, a2.element.symbol ])
        if elts in [ set(['H', 'N']), set(['H', 'C']) ]:
            if d < 1.5:
                print elts, d
set([u'H', u'C']) 1.08999999999
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.08999999999
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.08999999999
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.09000000017
set([u'H', u'C']) 1.08999999999
set([u'H', u'C']) 1.09000000017
set([u'H', u'N']) 1.00999999987
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999987
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999987
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999987
set([u'H', u'N']) 1.00999999961
set([u'H', u'N']) 1.00999999961
```

As you can see, all N-H and C-H bond lengths are correct.

5.3 Combinatorial site replacements

Now that we have a good structure, lets start replacing atoms! First, we need to specify the substitutions we will make. Lets start with simply isovalent substitutions for Pb and I. In MePbI3, Methylammonium is a +1 ion, Pb is +2 and I is -1. We will specify replacements in the form of lists of substitutions.:

```
>>> pb_sub = [ 'Pb', 'Sn',
            'Be', 'Mg', 'Ca', 'Sr', 'Ba',
            'V', 'Mn', 'Ni', 'Co', 'Fe', 'Zn',
            'Cd', 'Eu' ,'Ru', 'Pd', 'Pt', 'As']
>>> i_sub = [ 'I', 'Br', 'Cl', 'F', 'H',
            'N', 'S' ]
```

Next, we need to create a directory structure that is reasonable for understanding where the structures are. This is primarily to make it easier for people to find the calculations by hand; qmpy would be find with randomly generated strings for folder names. We organize the structures into nested directories of the form {anion}/{cation}. The substitutions are implemented by the `substitute()` method.:

```
>>> # first we write a little helper function for making folders
>>> def mkdir(path):
        if not os.path.exists(path):
            os.mkdir(path)
>>> # now we loop through antion/cation pairs and for each we:
>>> #   create the POSCAR
>>> #   create an Entry
>>> project = Project.get('example')
>>> for anion in i_sub:
        mkdir(anion)
        for cation in pb_sub:
            new_dir = '%s/%s' % (anion, cation)
            mkdir(new_dir)
            new_struct = s.replace({'Pb':cation,
                                    'I':anion})
            io.poscar.write(s, new_dir+'/POSCAR')
            entry = Entry.create(new_dir+'/POSCAR', projects=[project])
            entry.save()
            task = Task.create(entry, 'static')
```

5.4 Running calculations with qmpy

In order to run calculations with qmpy, we utilize a JobManager and TaskManager. The role of the TaskManager is to look at the calculations that have been requested (in the form of Tasks), and will attempt to fill the available resources with those calculations. The calculations are stored as Jobs, which tracks where the calculation is being run, and where it came from. This is where the JobManager takes over, checking all running Jobs, and if it is found to be done, it is collected. These managers can be accessed through the oqmd script (qmpy/bin/oqmd) either as a daemon process or, more safely, in a screen.:

```
$ oqmd jobserver -T run
```

and:

```
$ oqmd taskserver -T run
```

As Tasks and Jobs are processed, both of these methods will continuously report job submissions, task completions, as well as errors encountered.

CHAPTER

SIX

ANALYSIS TOOLS

Basically all thermodynamic analysis in qmpy is done starting from a PhaseSpace instance. If you have the database install and working, these are very easy to construct:

```
>>> ps = PhaseSpace('Li-Si')
>>> ps
<PhaseSpace bound by Li-Si>
```

Since the PhaseSpace was created without any extra arguments, it was assumed that you wanted to pull thermodynamic data from the OQMD, but you can fine tune the data that is included very easily. More on that later.

6.1 Convex Hull Construction

To obtain the convex hull for any phase space, simply access the *hull* attribute:

```
>>> ps.hull
set([<Equilibrium: Li13Si4-Li21Si5>, <Equilibrium: Li12Si7-Li7Si3>,
<Equilibrium: Li13Si4-Li7Si3>, <Equilibrium: LiSi-Li12Si7>, <Equilibrium:
Li21Si5-Li>, <Equilibrium: LiSi-Si>])
```

The hull is a set of Equilibrium objects, which have very natural attributes:

```
>>> eq = list(ps.hull)[0]
>>> eq.phases
[<Phase Li13Si4 : -0.240>, <Phase Li21Si5 : -0.212>]
>>> eq.chem_pots
{u'Si': -0.74005684434211016, u'Li': -0.086299552894736051}
```

6.2 Phase Stability

Positive for unstable phases, negative for stable phases.

Examples:

```
>>> p = ps.phase_dict['Li13Si4'] # for just one phase
>>> ps.compute_stability(p)
>>> p.stability
-0.007333029175317474
>>> ps.compute_stabilities()
>>> ps.phase_dict['Li2Si'].stability
0.03116726059829092
```

6.3 Grand Canonical Linear Programming

Examples:

```
>>> energy, phases = ps.gclp('LiSi2')
>>> energy
-0.404968066250002
>>> phases
{<Phase LiSi : -0.202>: 2.0, <Phase Si : 0>: 1.0}
>>> energy, phases = ps.gclp('Si', mus={'Li':-0.4})
>>> phases
{<Phase LiSi : -0.202>: 2.0}
```

6.4 Convex Hull Slices

Works by recursively using linear programming to find the lowest point contained within the a specified compositional region.

Examples:

```
>>> ps = PhaseSpace('Fe2O3-Li2O')
>>> ps.hull
set([<Equilibrium: LiFe5O8-LiFeO2>, <Equilibrium:
LiFeO2-Li5FeO4>, <Equilibrium: LiFe5O8-Fe2O3>, <Equilibrium:
Li5FeO4-Li2O>])
```

6.5 Reaction Enumeration

6.6 Stability Conditions

CHAPTER

SEVEN

EXAMPLES

To be filled out in more detail

7.1 Identification of FCC decortations

First, we will find all binary entries:

```
>>> binaries = Entry.objects.filter(ntypes=2)
>>> fcc = Composition.get('Cu').ground_state.structure
```

Then we run through every structure, and see if replacing all atoms with Cu results in a structure that is equivalent (on volume scaling) with FCC Cu.:

```
>>> fccs = []
>>> for entry in binaries[:100]:
>>>     struct = entry.structure
>>>     ## Construct a dictionary of elt:replacement_elt pairs
>>>     ## where every replacement is Cu
>>>     rdict = dict((k, 'Cu') for k in entry.comp)
>>>     test = struct.substitute(rdict, rescale=False,
>>>                                     in_place=False)
>>>     if fcc == test: # simple equality testing will work
>>>         fccs.append(entry)
```

Warning: If you actually try to run this on the entire database, understand that it will take a pretty long time! Each entry tested takes between 0.1 and 1 second, so it would take most of 24 hours to run through all 80,000+ binary database entries.

7.2 Deviation from Vagard's Law

Use the element_groups dictionary to look get a list of all simple metals:

```
>>> elts = element_groups['simple-metals']
```

Then, for each pair of metals get all of the entries, and their volumes.:

```
>>> vols = {}
>>> for e1, e2 in itertools.combinations(elts, r=2):
>>>     entries = Composition.get_list([e1, e2])
>>>     for entry in entries:
```

```
>>>         vol = entry.structure.volume_pa
>>>         vols[entry.name] = vols.get(entry.name, []) + [vol]
```

Then, for every composition get the Vagard's law volume.:

```
>>> vagards = {}
>>> for comp in vols:
>>>     comp = parse_comp(comp) # returns a elt:amt dictionary
>>>     uc = unit_comp(comp) # reduces to a total of 1 atom
>>>     vvol = 0
>>>     for elt, amt in uc.items():
>>>         vvol += elements[elt]['volume']*amt
```

More things you can do: * Calculate an average error for each system * Make a scatter plot for a few binaries show in volume vs x * Look for cases where some are above and some are below * Get relaxed volume of all stable compounds * What about including the "nearly stable"

7.3 Compute all A-B bond lengths

This script loops over pairs of elements, gets the binary PhaseSpace, and then loops over structures on the convex hull.:

```
>>> for e1, e2 in itertools.combinations(elts, r=2):
>>>     # do logic
>>>     if e1 == e2:
>>>         break
>>>     ps = PhaseSpace([e1,e2])
>>>     k = frozenset([e1,e2])
>>>     bonds = []
>>>     for p in ps.stable:
>>>         s = p.calculation.input
>>>         if s.ntypes < 2:
>>>             continue
>>>         dists = get_pair_distances(s, 10)
>>>         bonds.append(min(dists[k]))
>>>     print e1, e2, np.average(bonds), np.std(bonds)
```

7.4 Integrating with Sci-kit Learn

First, the necessary imports:

```
>>> from sklearn.svm import SVR
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn import cross_validation
>>> from sklearn.decomposition import PCA
>>> from sklearn import linear_model
>>> from sklearn import grid_search
>>> from qmpy import *
```

As an example problem, we will build a very simple model that predicts the volume of a compound at a given composition based only on the composition:

```
>>> elts = Element.objects.filter(symbol__in=element_groups['simple-metals'])
>>> out_elts = Element.objects.exclude(symbol__in=element_groups['simple-metals'])
>>> models = Calculation.objects.filter(path__contains='icsd')
>>> models = models.filter(converged=True, label__in=['static', 'standard'])
>>> models = models.exclude(composition__element_set=out_elts)
>>> data = models.values_list('composition_id', 'output__volume_pa')
```

Now we will build a fit set and test set:

```
>>> y = []
>>> X = []
>>> for c, v in data:
>>>     y.append(v)
>>>     X.append(get_basic_composition_descriptors(c).values())
>>> X = np.array(X)
>>> y = np.array(y)
>>> x1, x2, y1, y2 = cross_validation.train_test_split(X, y, train_size=0.5)
```

Now to actually implement the model:

```
>>> clf = linear_model.LinearRegression()
>>> clf.fit(x1, y1)
>>> clf.score(x2, y2)
```

CHAPTER

EIGHT

CONTRIBUTE

- Issue Tracker: http://github.com/wolverton-research-group/qmpy/issues
- Source: http://github.com/wolverton-research-group/qmpy

CHAPTER NINE

SUPPORT

If you are having issues, please let us know. We can be reached at oqmd.contact@gmail.com.

CHAPTER TEN

LICENSE

The project is licensed under the MIT license.

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX