Onsager Documentation

Release 0.9.1

Dallas R. Trinkle

Jun 10, 2016

CONTENTS

Contents:

CHAPTER

ONE

ONSAGER

Documentation now available at the [Onsager github page](http://dallastrinkle.github.io/Onsager/). Please cite as [![DOI](https://zenodo.org/badge/14172/DallasTrinkle/Onsager.svg){}](https://zenodo.org/badge/latestdoi/14172/DallasTrinkle/Onsager)

The Onsager package provides routines for the general calculation of transport coefficients in vacancy-mediated diffusion and interstitial diffusion. It does this using a Green function approach, combined with point group symmetry reduction for maximum efficiency.

Typical usage looks like:

```
#!/usr/bin/env python

    from onsager import crystal
    from onsager import OnsagerCalc

...
```

Many of the subpackages within Onsager are support for the main attraction, which is in OnsagerCalc. Interstitial calculation examples are avaliable in *bin*, including three YAML input files, as well as a interstitial diffuser. An example of vacancy-mediated diffusion is shown in *bin/fivefreq.py*, which computes the well-known five-frequency model for substitutional solute transport in an FCC lattice.

The tests for the package are include in *test*; *tests.py* will run all of the tests in the directory with verbosity level 2. This can be time-consuming (on the order of several of minutes) to run all tests; coverage is currently >90%.

The code uses YAML files for input/output of diffusion data for the interstitial calculator. The vacancy-mediated calculator requires much more data, and uses HDF5 format to save/reload as needed. The vacancy-mediated calculator uses tags (unique human-readable-ish strings) to identify all (symmetry-unique) vacancy, solute, and complex states, and transitions between them.

Releases:

0.9. Full release of Interstitial calculator, along with theory paper (see References below). 0.9.1. Added spin degrees of freedom to *crystal* for symmetry purposes; added *supercell* class to aid in automated setup of calculation.

CHAPTER

TWO

REFERENCES

- Dallas R. Trinkle, “Diffusivity and derivatives for interstitial solutes: Activation energy, volume, and elastodiffusion tensors.” [arXiv:1605.03623](http://arxiv.org/abs/1605.03623)

CHAPTER

THREE

CONTRIBUTORS

- Dallas R. Trinkle, initial design, derivation, and implementation.
- Ravi Agarwal, testing of HCP interstitial calculations; testing of HCP vacancy-mediated diffusion calculations
- Abhinav Jain, testing of HCP vacancy-mediated diffusion calculations.

Thanks to discussions with Maylise Nastar (CEA, Saclay), Thomas Garnier (CEA, Saclay and UIUC), Thomas Schuler (CEA, Saclay), and Pascal Bellon (UIUC).

CHAPTER

FOUR

SUPPORT

This work has been supported in part by

- DOE/BES grant DE-FG02-05ER46217,
- ONR grant N000141210752,
- NSF/CDSE grant 1411106.
- Dallas R. Trinkle began the theoretical work for this code during the long program on Material Defects at the [Institute for Pure and Applied Mathematics](https://www.ipam.ucla.edu/) at UCLA, Fall 2012, which is supported by the National Science Foundation.

CHAPTER

FIVE

CRYSTAL

Crystal:

The crystal module defines the `crystal` class, and `GroupOp` for group operations. Crystal class

Class to store definition of a crystal, along with some analysis 1. geometric analysis (nearest neighbor displacements) 2. space group operations 3. point group operations for each basis position 4. Wyckoff position generation (for interstitials)

`crystal.`**`CombineTensorBasis`**(*b1*, *b2*, *symmetric=True*)

Combines (intersects) two tensor spaces into one; uses SVD to compute null space.

Parameters

- **`b1`** – list of tensors
- **`b2`** – list of tensors

Return tensorbasis list of 2nd-rank symmetric tensors making up the basis

`crystal.`**`CombineVectorBasis`**(*b1*, *b2*)

Combines (intersects) two vector spaces into one.

Parameters

- **`b1`** – (dim, vect) – dimensionality (0..3), vector defining line direction (1) or plane normal (2)
- **`b2`** – (dim, vect)

Return dim dimensionality, 0..3

Return vect vector defining line direction (1) or plane normal (2)

class `crystal.`**`Crystal`**(*lattice*, *basis*, *chemistry=None*, *spins=None*, *NOSYM=False*, *noreduce=False*)

A class that defines a crystal, as well as the symmetry analysis that goes along with it. Now includes optional spins. These can be vectors or "scalar" spins, for which we need to consider a phase factor. In general, they can be complex. Ideally, they should have magnitude either 0 or 1.

Specified by a lattice (3 vectors), a basis (list of lists of positions in direct coordinates). Can also name the elements (chemistry), and specify spin degrees of freedom.

classmethod `BCC`(*a0*, *chemistry=None*)

Create a body-centered cubic crystal with lattice constant a0

Parameters **`a0`** – lattice constant

Return BCC crystal

classmethod `FCC`(*a0*, *chemistry=None*)

Create a face-centered cubic crystal with lattice constant a0

Parameters **a0** – lattice constant

Return FCC crystal

FullVectorBasis(*chem=None*)
Generate our full vector basis, using the information from our crystal

Parameters **chem** – (optional) chemical index to consider; otherwise return a list of such

Return VBfunctions (list) of our unique vector basis lattice functions, normalized; each is an array (NVbasis x Nsites x 3)

Return VVouter (list) of ouf VV "outer" expansion (NVbasis x NVbasis for each chemistry)

classmethod HCP(*a0*, *c_a=1.6329931618554521*, *chemistry=None*)
Create a hexagonal closed packed crystal with lattice constant a0, c/a ratio c_a

Parameters

- **a0** – lattice constant
- **c_a** – (optional) c/a ratio, default=ideal $\sqrt{8/3}$

Return HCP crystal

SymmTensorBasis(*ind*)
Generates the symmetric tensor basis corresponding to an atomic site

Parameters **ind** – tuple index for atom

Return tensorbasis list of 2nd-rank symmetric tensors making up the basis

VectorBasis(*ind*)
Generates the vector basis corresponding to an atomic site

Parameters **ind** – tuple index for atom

Return dim dimensionality, 0..3

Return vect vector defining line direction (1) or plane normal (2)

Wyckoffpos(*uvec*)
Generates all the equivalent Wyckoff positions for a unit cell vector.

Parameters **uvec** – 3-vector (float) vector in direct coordinates

Return Wyckofflist list of equivalent Wyckoff positions

__init__(*lattice*, *basis*, *chemistry=None*, *spins=None*, *NOSYM=False*, *noreduce=False*)
Initialization; starts off with the lattice vector definition and the basis vectors. While it does not explicitly store the specific chemical elements involved, it does store that there are different elements.

Parameters

- **lattice** – array[3,3] or list of array[3] lattice vectors; if [3,3] array, then the vectors need to be in *column* format so that the first lattice vector is lattice[:,0]
- **basis** – list of array[3] or list of list of array[3] crystalline basis vectors, in unit cell coordinates. If a list of lists, then there are multiple chemical elements, with each list corresponding to a unique element
- **chemistry** – (optional) list of names of chemical elements
- **spins** – (optional) list of numbers (complex) / vectors or list of list of same spins for individual atoms; if not None, needs to match the basis. Can either be scalars or vectors, corresponding to collinear or non-collinear magnetism

- **NOSYM** – turn off all symmetry finding (except identity)
- **noreduce** – do not attempt to reduce the atomic basis

__repr__()
String representation of crystal (lattice + basis)

__str__()
Human-readable version of crystal (lattice + basis)

__weakref__
list of weak references to the object (if defined)

addbasis(*basis*, *chemistry=None*, *spins=None*)
Returns a new Crystal object that contains additional sites (assumed to be new chemistry). This is intended to "add in" interstitial sites. Note: if the symmetry is to be maintained, should be the output from Wyckoffpos().

Parameters

- **basis** – list (or list of lists) of new sites
- **chemistry** – (optional) list of chemistry names
- **spins** – (optional) list of spins

Return Crystal new crystal object, with additional sites

calcmetric()
Computes the volume of the cell and the metric tensor

Return volume cell volume

Return metric tensor 3x3

cart2pos(*v*)
Return the lattvec and index corresponding to an atomic position in cartesian coord.

Parameters **v** – 3-vector (float) position in Cartesian coordinates

Return lattvec 3-vector (integer) lattice vector in direct coordinates,

Return (c,i) tuple of matching basis atom; None if no match

cart2unit(*v*)
Return the lattvec and unit cell coord. corresponding to a position in cartesian coord.

Parameters **v** – 3-vector (float) position in Cartesian coordinates

Return lattvec 3-vector (integer) lattice vector in direct coordinates,

Return uvec 3-vector (float) inside unit cell, in direct coordinates

center()
Center the atoms in the cell if there is an inversion operation present.

chemindex(*chemistry*)
Return index corresponding to chemistry; None if not present.

Parameters **chemistry** – value to check

Return index corresponding to chemistry

classmethod fromdict(*yamldict*)
Creates a Crystal object from a *very simple* YAML-created dictionary

Parameters **yamldict** – dictionary; must contain 'lattice' (using *row* vectors!) and 'basis'; can contain optional 'lattice_constant'

Return Crystal(lattice.T, basis) new crystal object

fullkptmesh(*Nmesh*)

Creates a k-point mesh of density given by Nmesh; does not symmetrize but does put the k-points inside the BZ. Does not return any *weights* as every point is equally weighted.

Parameters **Nmesh** – mesh divisions Nmesh[0] x Nmesh[1] x Nmesh[2]

Return kpt array[Nkpt][3] of kpoints

g_cart(*g*, *x*)

Apply a space group operation to a (Cartesian) vector position

Parameters

- **g** – group operation (GroupOp)
- **x** – 3-vector position in space

Return gx 3-vector position in space (Cartesian coordinates)

static g_direc(*g*, *direc*)

Apply a space group operation to a direction

Parameters

- **g** – group operation (GroupOp)
- **direc** – 3-vector direction

Return gdirec 3-vector direction

g_direc_equivalent(*d1*, *d2*, *threshold=1e-08*)

Tells us if two directions are equivalent by according to the space group

Parameters

- **d1** – direction one (array[3])
- **d2** – direction two (array[3])
- **threshold** – threshold for equality

Return equivalent True if equivalent by a point group operation

g_pos(*g*, *lattvec*, *ind*)

Apply a space group operation to an atom position specified by its lattice and index

Parameters

- **g** – group operation (GroupOp)
- **lattvec** – 3-vector (integer) lattice vector in direct coordinates
- **ind** – two-tuple index specifying the atom: (atomtype, atomindex)

Return glatt 3-vector (integer) lattice vector in direct coordinates

Return gindex tuple of new basis atom

static g_tensor(*g*, *tensor*)

Apply a space group operation to a 2nd-rank tensor

Parameters

- **g** – group operation (GroupOp)

- **tensor** – 2nd-rank tensor

Return gtensor 2nd-rank tensor

static g_vect(*g*, *lattvec*, *uvec*)
Apply a space group operation to a vector position specified by its lattice and a location in the unit cell in direct coordinates

Parameters

- **g** – group operation (GroupOp)
- **lattvec** – 3-vector (integer) lattice vector in direct coordinates
- **uvec** – 3-vector (float) vector in direct coordinates
- **guvec** – 3-vector (float) vector in direct coordinates

Return glatt 3-vector (integer) lattice vector in direct coordinates

genBZG()
Generates the reciprocal lattice G points that define the Brillouin zone.

Return Garray array of G vectors that define the BZ, in Cartesian coordinates

genWyckoffsets()
Generate our Wykcoff sets.

Return Wyckoffsets set of sets of tuples of positions that correspond to identical Wyckoff positions

gengroup()
Generate all of the space group operations. Now handles spins! Doesn't store spin phase factors for each group operation, though.

Return Gset frozenset of group operations

genpoint()
Generate our point group indices. Done with crazy list comprehension due to the structure of our basis.

Return Gpointlists list of lists of frozensets of point group operations that leave a site unchanged

inBZ(*vec*, *BZG=None*, *threshold=1e-05*)
Tells us if vec is inside our set of defining points.

Parameters

- **vec** – array [3], vector to be tested
- **BGZ** – array [:,3], optional (default = self.BZG), array of vectors that define the BZ
- **threshold** – double, optional, threshold to use for "equality"

Return inBZ False if outside the BZ, True otherwise

jumpnetwork(*chem*, *cutoff*, *closestdistance=0*)
Generate the full jump network for a specific chemical index, out to a cutoff. Organized by symmetry-unique transitions. Note that i->j and j->i are always related to one-another, but by equivalence of transition state, not symmetry. Now updated with closest-distance parameter.

Parameters

- **chem** – index corresponding to the chemistry to consider
- **cutoff** – distance cutoff

- **closestdistance** – closest distance allowed in transition (can be a list)

Return jumpnetwork list of symmetry-unique transitions; each is a list of tuples: *((i,j), dx)* corresponding to jump from $i \to j$ with vector $\delta\mathbf{x}$

jumpnetwork2lattice(*chem*, *jumpnetwork*)

Convert a "standard" jumpnetwork (that specifies displacement vectors dx) into a lattice representation, where we replace dx with the lattice vector from i to j.

Parameters

- **chem** – index corresponding to the chemistry to consider
- **jumpnetwork** – list of symmetry-unique transitions; each is a list of tuples: *((i,j), dx)* corresponding to jump from $i \to j$ with vector $\delta\mathbf{x}$

Return jumplattice list of symmetry-unique transitions; each is a list of tuples: *((i,j), R)* corresponding to jump from i in unit cell 0 -> j in unit cell R

minlattice()

Try to find the optimal lattice vector definition for a crystal. Our definition of optimal is (a) length of each lattice vector is minimal; (b) the vectors are ordered from shortest to longest; (c) the vectors have minimal dot product; (d) the basis is right-handed.

Works recursively, and in-place.

nnlist(*ind*, *cutoff*)

Generate the nearest neighbor list for a given cutoff. Only consider neighbor vectors for atoms of the same type. Returns a list of cartesian vectors.

Parameters

- **ind** – tuple index for atom
- **cutoff** – distance cutoff

Return nnlist list of nearest neighbor vectors

pos2cart(*lattvec*, *ind*)

Return the cartesian coordinates of an atom specified by its lattice and index

Parameters

- **lattvec** – 3-vector (integer) lattice vector in direct coordinates
- **ind** – two-tuple index specifying the atom: (atomtype, atomindex)

Return v 3-vector (float) in Cartesian coordinates

reduce(*threshold=1e-08*)

Reduces the lattice and basis, if needed. Works (tail) recursively.

reducekptmesh(*kptfull*, *threshold=1e-08*)

Takes a fully expanded mesh, and reduces it by symmetry. Assumes every point is equally weighted. We would need a different (more complicated) algorithm if not true...

Parameters

- **kptfull** – array[Nkpt][3] of kpoints
- **threshold** – threshold for symmetry equality

Return kptsymm array[Nsymm][3] of kpoints

Return weight array[Nsymm] of weights (integrates to 1)

remapbasis(*supercell*)
Takes the basis definition, and using a supercell definition, returns a new basis

Parameters **supercell** – integer array[3,3]

Return atomic basis list of list of positions

simpleYAML(*a0=1.0*)
Creates a simplified YAML dump, in case we don't want to output the full symmetry analysis

Return YAML string dump

sitelist(*chem*)
Return a list of lists of Wyckoff-related sites for a given chemistry. Done with a single list comprehension– useful as input for diffusion calculation

Parameters **chem** – index corresponding to chemistry to consider

Return symmequivsites list of lists of indices that are equivalent by symmetry

strain(*eps*)
Returns a new Crystal object that is a strained version of the current.

Parameters **eps** – strain tensor

Return Crystal new crystal object, strained

unit2cart(*lattvec*, *uvec*)
Return the cartesian coordinates of a position specified by its lattice and unit cell coordinates

Parameters

- **lattvec** – 3-vector (integer) lattice vector in direct coordinates
- **uvec** – 3-vector (float) unit cell vector in direct coordinates

Return v 3-vector (float) in Cartesian coordinates

static vectlist(*vb*)
Returns a list of orthonormal vectors corresponding to our vector basis.

Parameters **vb** – (dim, v)

Return vlist list of vectors

class crystal.**GroupOp**
A class corresponding to a group operation. Based on namedtuple, so it is immutable.

Intended to be used in combination with Crystal, we have a few operations that can be defined out-of-the-box.

Parameters

- **rot** – np.array(3,3) integer idempotent matrix
- **trans** – np.array(3) real vector
- **cartrot** – np.array(3,3) real unitary matrix
- **indexmap** – tuples of tuples, containing the atom mapping

static GroupOp_constructor(*loader*, *node*)
Construct a GroupOp from YAML

static GroupOp_representer(*dumper*, *data*)
Output a GroupOp

__add__(*other*)
Add a translation to our group operation

__eq__(*other*)
Test for equality–we use numpy.isclose for comparison, since that's what we usually care about

__hash__()
Hash, so that we can make sets of group operations

__mul__(*other*)
Multiply two group operations to produce a new group operation

__ne__(*other*)
Inequality == not __eq__

__sane__()
Return true if the cartrot and rot are consistent and 'sane'

__str__()
Human-readable version of groupop

__sub__(*other*)
Add a (negative) translation to our group operation

eigen()
Returns the type of group operation (single integer) and eigenvectors. 1 = identity 2, 3, 4, 6 = n- fold rotation around an axis negative = rotation + mirror operation, perpendicular to axis "special cases": -1 = mirror, -2 = inversion

eigenvect[0] = axis of rotation / mirror eigenvect[1], eigenvect[2] = orthonormal vectors to define the plane giving a right-handed coordinate system and where rotation around [0] is positive, and the positive imaginary eigenvector for the complex eigenvalue is [1] + i [2].

Return type integer

Return eigenvectors list of [ev0, ev1, ev2]

classmethod ident(*basis*)
Return a group operation corresponding to identity for a given basis

incell()
Return a version of groupop where the translation is in the unit cell

inhalf()
Return a version of groupop where the translation is in the centered unit cell

inv()
Construct and return the inverse of the group operation

static optype(*rot*)
Returns the type of group operation (single integer) and eigenvectors. 1 = identity 2, 3, 4, 6 = n- fold rotation around an axis negative = rotation + mirror operation, perpendicular to axis "special cases": -1 = mirror, -2 = inversion

Parameters **rot** – rotation matrix (can be the integer rot)

Return type integer

crystal.**ProjectTensorBasis**(*tensor*, *basis*)
Given a tensor, project it onto the basis.

Parameters

- **tensor** – tensor
- **basis** – list consisting of an orthonormal basis

Return tensor tensor, projected

crystal.**SymmTensorBasis**(*rottype*, *eigenvect*)
Returns a symmetric second-rank tensor basis corresponding to the optype and eigenvectors for a GroupOp

Parameters

- **rottype** – output from eigen()
- **eigenvect** – eigenvectors

Return tensorbasis list of 2nd-rank symmetric tensors making up the basis

crystal.**VectorBasis**(*rottype*, *eigenvect*)
Returns a vector basis corresponding to the optype and eigenvectors for a GroupOp

Parameters

- **rottype** – output from eigen()
- **eigenvect** – eigenvectors

Return dim dimensionality, 0..3

Return vect vector defining line direction (1) or plane normal (2)

crystal.**Voigtstrain**(*e1*, *e2*, *e3*, *e4*, *e5*, *e6*)
Returns a symmetric strain tensor from the Voigt reduced strain values.

Parameters

- **e1** – xx
- **e2** – yy
- **e3** – zz
- **e4** – yz + zx
- **e5** – zx + xz
- **e6** – xy + yx

Return strain symmetric strain tensor

crystal.**incell**(*vec*)
Returns the vector inside the unit cell (in [0,1)**3)

Parameters **vec** – 3-vector (unit coord)

Returns 3-vector

crystal.**inhalf**(*vec*)
Returns the vector inside the centered cell (in [-0.5,0.5)**3)

Parameters **vec** – 3-vector (unit coord)

Returns 3-vector

crystal.**maptranslation**(*oldpos*, *newpos*, *oldspins=None*, *newspins=None*, *threshold=1e-08*)
Given a list of transformed positions, identify if there's a translation vector that maps from the current positions to the new position.

The mapping specifies the index that the *translated* atom corresponds to in the original position set. If unable to construct a mapping, the mapping return is None; the translation vector will be meaningless.

If old/newspins are given then ONLY mappings that maintain spin are considered. This means that a loop is needed to consider possible spin phase factors.

Parameters

- **oldpos** – list of list of array[3]
- **newpos** – list of list of array[3], same layout as oldpos
- **oldspins** – (optional) list of list of numbers/arrays
- **newspins** – (optional) list of list of numbers/arrays

Return translation array[3]

Return mapping list of list of indices

crystal.**ndarray_representer**(*dumper*, *data*)
Output a numpy array

CHAPTER

SIX

CRYSTALSTARS

CrystalStars:

The crystalStars module defines the classes corresponding to stars (in this case, for solute-vacancy complexes that are equivalent by space group symmetry), and vector stars (the inclusion of a vector basis on the stars). These modules are primarily responsible for all the symmetry analysis, and converting that into matrix forms for rapid numerical evaluation as needed. Stars module, modified to work with crystal class

Classes to generate star sets, double star sets, and vector star sets; a lot of indexing functionality.

NOTE: The naming follows that of stars; the functionality is extremely similar, and this code was modified as little as possible to translate that functionality to *crystals* which possess a basis. In the case of a single atom basis, this should reduce to the stars object functionality.

The big changes are:

- Replacing NNvect star (which represents the jumps) with the jumpnetwork type found in crystal
- Using the jumpnetwork_latt representation from crystal
- Representing a "point" as a solute + vacancy. In this case, it is a tuple (s,v) of unit cell indices and a vector dx or dR (dx = Cartesian vector pointing from solute to vacancy; dR = lattice vector pointing from unit cell of solute to unit cell of vacancy). This is equivalent to our old representation if the tuple (s,v) = (0,0) for all sites. Due to translational invariance, the solute always stays inside the unit cell
- Using indices into the point list rather than just making lists of the vectors themselves. This is because the "points" now have a more complex representation (see above).

crystalStars.**PSlist2array**(*PSlist*)

Take in a list of pair states; return arrays that can be stored in HDF5 format

Parameters **PSlist** – list of pair states

Return ij int_array[N][2] = (i,j)

Return R int[N][3]

Return dx float[N][3]

class crystalStars.**PairState**

A class corresponding to a "pair" state; in this case, a solute-vacancy pair, but can also be a transition state pair. The solute (or initial state) is in unit cell 0, in position indexed i; the vacancy (or final state) is in unit cell R, in position indexed j. The cartesian vector dx connects them. We can add and subtract, negate, and "endpoint" subtract (useful for determining what Green function entry to use)

Parameters

- **i** – index of the first member of the pair (solute)
- **j** – index of the second member of the pair (vacancy)

- **R** – lattice vector pointing from unit cell of i to unit cell of j
- **dx** – Cartesian vector pointing from first to second member of pair

static PairState_constructor(*loader*, *node*)
Construct a GroupOp from YAML

static PairState_representer(*dumper*, *data*)
Output a PairState

__add__(*other*)
Add two states: works if and only if self.j == other.i (i,j) R + (j,k) R' = (i,k) R+R' : works for thinking about transitions... Note: a + b != b + a, and may be that only one of those is even defined

__eq__(*other*)
Test for equality–we don't bother checking dx

__hash__()
Hash, so that we can make sets of states

__ne__(*other*)
Inequality == not __eq__

__neg__()
Negation of state (swap members of pair) - (i,j) R = (j,i) -R Note: a + (-a) == (-a) + a == 0 because we define what "zero" is.

__sane__(*crys*, *chem*)
Determine if the dx value makes sense given everything else...

__str__()
Human readable version

__sub__(*other*)
Add a negative: a-b points from initial of a to initial of b if same final state (i,j) R - (k,j) R' = (i,k) R-R' Note: this means that (a-b) + b = a, but b + (a-b) is an error. (b-a) + a = b

__xor__(*other*)
Subtraction on the endpoints (sort of the "opposite" of a-b): a^b points from final of b to final of a if same initial state (i,j) R ^ (i,k) R' = (k,j) R-R' Note: b + (a^b) = a but (a^b) + b is an error. a + (b^a) = b

classmethod fromcrys(*crys*, *chem*, *ij*, *dx*)
Convert (i,j), dx into PairState

classmethod fromcrys_latt(*crys*, *chem*, *ij*, *R*)
Convert (i,j), R into PairState

g(*crys*, *chem*, *g*)
Apply group operation.

Parameters

- **crys** – crystal
- **chem** – chemical index
- **g** – group operation (from crys)

Return g*PairState corresponding to group operation applied to self

iszero()
Quicker than self == PairState.zero()

classmethod zero(*n=0*)
Return a "zero" state

class crystalStars.**StarSet**(*jumpnetwork*, *crys*, *chem*, *Nshells=0*, *lattice=False*)
A class to construct crystal stars, and be able to efficiently index.

Takes in a jumpnetwork, which is used to construct the corresponding stars, a crystal object with which to operate, a specification of the chemical index for the atom moving (needs to be consistent with jumpnetwork and crys), and then the number of shells.

In this case, *shells* = number of successive "jumps" from a state. As an example, in FCC, 1 shell = 1st neighbor, 2 shell = 1-4th neighbors.

__add__(*other*)
Add two StarSets together; done by making a copy of one, and iadding

__iadd__(*other*)
Add another StarSet to this one; very similar to generate()

__init__(*jumpnetwork*, *crys*, *chem*, *Nshells=0*, *lattice=False*)
Initiates a star set generator for a given jumpnetwork, crystal, and specified chemical index.

Parameters

- **jumpnetwork** – list of symmetry unique jumps, as a list of list of tuples; either *((i,j), dx)* for jump from i to j with displacement dx, or *((i,j), R)* for jump from i in unit cell 0 -> j in unit cell R
- **crys** – crystal where jumps take place
- **chem** – chemical index of atom to consider jumps
- **Nshells** – number of shells to generate
- **lattice** – which form does the jumpnetwork take?

__str__()
Human readable version

__weakref__
list of weak references to the object (if defined)

addhdf5(*HDF5group*)
Adds an HDF5 representation of object into an HDF5group (needs to already exist).

Example: if f is an open HDF5, then StarSet.addhdf5(f.create_group('StarSet')) will (1) create the group named 'StarSet', and then (2) put the StarSet representation in that group.

Parameters **HDF5group** – HDF5 group

copy(*empty=False*)
Return a copy of the StarSet; done as efficiently as possible; empty means skip the shells, etc.

diffgenerate(*S1*, *S2*, *threshold=1e-08*)
Construct a starSet using endpoint subtraction from starset S1 to starset S2. Can (will) include zero. Points from vacancy states of S1 to vacancy states of S2.

Parameters

- **S1** – starSet for start
- **S2** – starSet for final
- **threshold** – threshold for sorting magnitudes (can influence symmetry efficiency)

generate(*Nshells*, *threshold=1e-08*, *originstates=False*)
Construct the points and the stars in the set. Now includes "origin states" by default; these are PairStates that iszero() is True; they are only included if they have a nonzero VectorBasis.

Parameters

- **Nshells** – number of shells to generate; this is interpreted as subsequent "sums" of jumplist (as we need the solute to be connected to the vacancy by at least one jump)
- **threshold** – threshold for determining equality with symmetry
- **originstates** – include origin states in generate?

jumpnetwork_omega1()
Generate a jumpnetwork corresponding to vacancy jumping while the solute remains fixed.

Return jumpnetwork list of symmetry unique jumps; list of list of tuples (i,f), dx where i,f index into states for the initial and final states, and dx = displacement of vacancy in Cartesian coordinates. Note: if (i,f), dx is present, so if (f,i), -dx

Return jumptype list of indices corresponding to the (original) jump type for each symmetry unique jump; useful for constructing a LIMB approximation

Return starpair list of tuples of the star indices of the i and f states for each symmetry unique jump

jumpnetwork_omega2()
Generate a jumpnetwork corresponding to vacancy exchanging with a solute.

Return jumpnetwork list of symmetry unique jumps; list of list of tuples (i,f), dx where i,f index into states for the initial and final states, and dx = displacement of vacancy in Cartesian coordinates. Note: if (i,f), dx is present, so if (f,i), -dx

Return jumptype list of indices corresponding to the (original) jump type for each symmetry unique jump; useful for constructing a LIMB approximation

Return starpair list of tuples of the star indices of the i and f states for each symmetry unique jump

classmethod loadhdf5(*crys*, *HDF5group*)
Creates a new StarSet from an HDF5 group.

Parameters

- **crys** – crystal object–MUST BE PASSED IN as it is not stored with the StarSet
- **HDFgroup** – HDF5 group

Return StarSet new StarSet object

starindex(*PS*)
Return the index for the star to which pair state PS belongs; None if not found

stateindex(*PS*)
Return the index of pair state PS; None if not found

symmatch(*PS1*, *PS2*)
True if there exists a group operation that makes PS1 == PS2.

symmequivjumplist(*i*, *f*, *dx*)
Returns a list of tuples of symmetry equivalent jumps

Parameters

- **i** – index of initial state
- **f** – index of final state
- **dx** – displacement vector

Return symmjumplist list of tuples of ((gi, gf), gdx) for every group op

class crystalStars.**StarSetMeta**(*jumpnetwork*, *crys*, *chem*, *Nshells=0*, *lattice=False*, *meta_tags=[]*, *jumpnetwork2=[]*)

Testing meta states with star set

__init__(*jumpnetwork*, *crys*, *chem*, *Nshells=0*, *lattice=False*, *meta_tags=[]*, *jumpnetwork2=[]*)

Initiates a star set generator for a given jumpnetwork, crystal, and specified chemical index.

Parameters

- **jumpnetwork** – list of symmetry unique jumps, as a list of list of tuples; either ((i,j), dx) for jump from i to j with displacement dx, or ((i,j), R) for jump from i in unit cell 0 -> j in unit cell R
- **crys** – crystal where jumps take place
- **chem** – chemical index of atom to consider jumps
- **Nshells** – number of shells to generate
- **lattice** – which form does the jumpnetwork take?

copy(*empty=False*)

Return a copy of the StarSet; done as efficiently as possible; empty means skip the shells, etc.

generate(*Nshells*, *threshold=1e-08*)

Construct the points and the stars in the set. Now includes "origin states" by default; these are PairStates that iszero() is True; they are only included if they have a nonzero VectorBasis.

Parameters

- **Nshells** – number of shells to generate; this is interpreted as subsequent "sums" of jumplist (as we need the solute to be connected to the vacancy by at least one jump)
- **threshold** – threshold for determining equality with symmetry
- **originstates** – include origin states in generate?

jumpnetwork_omega2()

Generate a jumpnetwork corresponding to vacancy exchanging with a solute.

Return jumpnetwork list of symmetry unique jumps; list of list of tuples (i,f), dx where i,f index into states for the initial and final states, and dx = displacement of vacancy in Cartesian coordinates. Note: if (i,f), dx is present, so is (f,i), -dx

Return jumptype list of indices corresponding to the (original) jump type for each symmetry unique jump; useful for constructing a LIMB approximation

Return starpair list of tuples of the star indices of the i and f states for each symmetry unique jump

class crystalStars.**VectorStarSet**(*starset=None*)

A class to construct vector star sets, and be able to efficiently index.

All based on a StarSet

GFexpansion(*VectorBasis=()*)

Construct the GF matrix expansion in terms of the star vectors, and indexed to GFstarset. Now takes in a VectorBasis; if not an empty set, returns the expansion of the vector stars to the "origin states" as represented by the VB.

Parameters **VectorBasis** – (optional) list of [Nsites, 3] the vector basis in the unit cell for the solute states

Return GFexpansion array[Nsv, Nsv, NGFstars] the GF matrix[i, j] = sum(GFexpansion[i, j, k] * GF(starGF[k]))

Return GFstarset starSet corresponding to the GF

Return GFOSexpansion array[NVB, Nsv, NGFstars] the GF matrix[os, i] = sum(GFOSexpansion[os, i, k] * GF(starGF[k]))

__init__ (*starset=None*)
Initiates a vector-star generator; work with a given star.

Parameters **starset** – StarSet, from which we pull nearly all of the info that we need

__weakref__
list of weak references to the object (if defined)

addhdf5 (*HDF5group*)
Adds an HDF5 representation of object into an HDF5group (needs to already exist).

Example: if f is an open HDF5, then StarSet.addhdf5(f.create_group('VectorStarSet')) will (1) create the group named 'VectorStarSet', and then (2) put the VectorStarSet representation in that group.

Parameters **HDF5group** – HDF5 group

bareexpansions (*jumpnetwork*, *jumptype*)
Construct the bare diffusivity expansion in terms of the jumpnetwork. We return the reference (0) contribution so that the change can be determined; this is useful for the vacancy contributions. This saves us from having to deal with issues with our outer shell where we only have a fraction of the escapes, but as long as the kinetic shell is one more than the thermodynamics (so that the interaction energy is 0, hence no change in probability), this will work. The PS (pair stars) is useful for including the probability factor for the endpoint of the jump; we just call it the 'probfactor' below.

Note also: this *currently assumes* that the displacement vector *does not change* between omega0 and omega(1/2).

Parameters

- **jumpnetwork** – jumpnetwork of symmetry unique omega1-type jumps, corresponding to our starset. List of lists of (IS, FS), dx tuples, where IS and FS are indices corresponding to states in our starset.
- **jumptype** – specific omega0 jump type that the jump corresponds to

Return D0expansion array[3,3, Njump_omega0] the D0[a,b,jt] = sum(D0expansion[a,b, jt] * sqrt(probfactor0[PS[jt][0]]*probfactor0[PS[jt][1]) * omega0[jt])

Return D1expansion array[3,3, Njump_omega1] the D1[a,b,k] = sum(D1expansion[a,b, k] * sqrt(probfactor[PS[k][0]]*probfactor[PS[k][1]) * omega[k])

biasexpansions (*jumpnetwork*, *jumptype*)
Construct the bias1 and bias0 vector expansion in terms of the jumpnetwork. We return the bias0 contribution so that the db = bias1 - bias0 can be determined. This saves us from having to deal with issues with our outer shell where we only have a fraction of the escapes, but as long as the kinetic shell is one more than the thermodynamics (so that the interaction energy is 0, hence no change in probability), this will work. The PS (pair stars) is useful for including the probability factor for the endpoint of the jump; we just call it the 'probfactor' below. *Note:* this used to be separated into bias1expansion, and bias2expansion,and had terms that are now in rateexpansions. Note also that if jumpnetwork_omega2 is passed, it also works for that. However, in that case we have a different approach for the calculation of bias1expansion: if there are origin states, they get the negative summed bias of the others.

Parameters

- **jumpnetwork** – jumpnetwork of symmetry unique omega1-type jumps, corresponding to our starset. List of lists of (IS, FS), dx tuples, where IS and FS are indices corresponding to states in our starset.
- **jumptype** – specific omega0 jump type that the jump corresponds to

Return bias0expansion array[Nsv, Njump_omega0] the gen0 vector[i] = sum(bias0expasion[i, k] * sqrt(probfactor0[PS[k]]) * omega0[k])

Return bias1expansion array[Nsv, Njump_omega1] the gen1 vector[i] = sum(bias1expansion[i, k] * sqrt(probfactor[PS[k]] * omega1[k])

generate(*starset*, *threshold=1e-08*)
Construct the actual vectors stars

Parameters **starset** – StarSet, from which we pull nearly all of the info that we need

generateouter()
Generate our outer products for our star-vectors.

Return outer array [3, 3, Nvstars, Nvstars] outer[:, :, i, j] is the 3x3 tensor outer product for two vector-stars vs[i] and vs[j]

classmethod **loadhdf5**(*SSet*, *HDF5group*)
Creates a new VectorStarSet from an HDF5 group.

Parameters

- **SSet** – StarSet–MUST BE PASSED IN as it is not stored with the VectorStarSet
- **HDFgroup** – HDF5 group

Return VectorStarSet new VectorStarSet object

periodicvectorexpansion(*elemtype='solute'*)
Construct the expansion from vectors on sites in the cell that are periodic to our vectorstar basis. This is used to map from the rate-bias correction vectors into the vectorstar basis, to correct for situations where the vacancy jumps themselves have bias.

Parameters **elemtype** – 'solute' or 'vacancy', depending on which site we need to check.

Return periodicexpansion [Nvstars, Nsites, 3], to map Nsites,3 into Nvstars

rateexpansions(*jumpnetwork*, *jumptype*, *VectorBasis=()*)
Construct the omega0 and omega1 matrix expansions in terms of the jumpnetwork; includes the escape terms separately. The escape terms are tricky because they have probability factors that differ from the transitions; the PS (pair stars) is useful for finding this. We just call it the 'probfactor' below. *Note:* this used to be separated into rate0expansion, and rate1expansion, and partly in bias1expansion. Note also that if jumpnetwork_omega2 is passed, it also works for that. However, in that case we have a different approach for the calculation of rate0expansion: if there are origin states, then we need to "jump" to those; if there is a non-empty VectorBasis we will want to account for them there.

Parameters

- **jumpnetwork** – jumpnetwork of symmetry unique omega1-type jumps, corresponding to our starset. List of lists of (IS, FS), dx tuples, where IS and FS are indices corresponding to states in our starset.
- **jumptype** – specific omega0 jump type that the jump corresponds to
- **VectorBasis** – (optional) list of [Nsites, 3] the vector basis in the unit cell for the solute states

Return rate0expansion array[Nsv, Nsv, Njump_omega0] the omega0 matrix[i, j] = sum(rate0expansion[i, j, k] * omega0[k]); *IF* NVB>0 we "hijack" this and use it for [NVB, Nsv, Njump_omega0], as we're doing an omega2 calc and rate0expansion won't be used *anyway*.

Return rate0escape array[Nsv, Njump_omega0] the escape contributions: omega0[i,i] += sum(rate0escape[i,k]*omega0[k]*probfactor(PS[k]))

Return rate1expansion array[Nsv, Nsv, Njump_omega1] the omega1 matrix[i, j] = sum(rate1expansion[i, j, k] * omega1[k])

Return rate1escape array[Nsv, Njump_omega1] the escape contributions: omega1[i,i] += sum(rate1escape[i,k]*omega0[k]*probfactor(PS[k]))

unitcellVectorBasisfolddown(*VectorBasis*, *elemtype='solute'*)

Construct the expansion to "fold down" from starvector to a VectorBasis in the unit cell :param VectorBasis: list of (N,3) matrices, corresponding to (normalized) vectors :param elemtype: 'solute' of 'vacancy', depending on which site we need to reduce :return: folddown: [NV, Nvstars] to map vstars to VectorBasis

crystalStars.**array2PSlist**(*ij*, *R*, *dx*)

Take in arrays of ij, R, dx (from HDF5), return a list of PairStates

Parameters

- **ij** – int_array[N][2] = (i,j)
- **R** – int[N][3]
- **dx** – float[N][3]

Return PSlist list of pair states

crystalStars.**doublelist2flatlistindex**(*listlist*)

Takes a list of lists, returns a flattened list and an index array

Parameters **listlist** – list of lists of objects

Return flatlist flat list of objects (preserving order)

Return indexarray array indexing which original list it came from

crystalStars.**flatlistindex2doublelist**(*flatlist*, *indexarray*)

Takes a flattened list and an index array, returns a list of lists

Parameters

- **flatlist** – flat list of objects (preserving order)
- **indexarray** – array indexing which original list it came from

Return listlist list of lists of objects

CHAPTER

SEVEN

SUPERCELL

Supercell:

The supercell module defines the `supercell` class for building supercells from `crystal.Crystal` classes.
Supercell class

Class to store supercells of crystals. A supercell is a lattice model of a crystal, with periodically repeating unit cells. In that framework we can

1. add/remove/substitute atoms
2. find the transformation map between two different representations of the same supercell
3. output POSCAR format (possibly other formats?)

class `supercell.`**`Supercell`**(*crys*, *super*, *interstitial=()*, *Nsolute=0*, *empty=False*, *NOSYM=False*)
A class that defines a Supercell of a crystal.

Takes in a crystal, a supercell (3x3 integer matrix). We can identify sites as interstitial sites, and specify if we'll have solutes.

`KrogerVink`()
Attempt to make a "simple" string based on the defectindices, using Kroger-Vink notation. That is, we identify: vacancies, antisites, and interstitial sites, and return a string. NOTE: there is no relative charges, so this is a pseudo-KV notation.

Return KV string representation

`POSCAR`(*name=None*, *stoichiometry=True*)
Return a VASP-style POSCAR, returned as a string.

Parameters

- **`name`** – (optional) name to use for first list
- **`stoichiometry`** – (optional) if True, append stoichiometry to name

Return POSCAR string

`__eq__`(*other*)
Return True if two supercells are equal; this means they should have the same occupancy. *and* the same ordering

Parameters **`other`** – supercell for comparison

Returns True if same crystal, supercell, occupancy, and ordering; False otherwise

`__getitem__`(*key*)
Index into supercell

Parameters **`key`** – index (either an int, a slice, or a position)

Returns chemical occupation at that point

__imul__(*other*)
Multiply by a GroupOp, in place.

Parameters **other** – must be a GroupOp (and *should* be a GroupOp of the supercell!)

Returns self

__init__(*crys*, *super*, *interstitial=()*, *Nsolute=0*, *empty=False*, *NOSYM=False*)
Initialize our supercell to an empty supercell.

Parameters

- **crys** – crystal object
- **super** – 3x3 integer matrix
- **interstitial** – (optional) list/tuple of indices that correspond to interstitial sites
- **Nsolute** – (optional) number of substitutional solute elements to consider; default=0
- **empty** – (optional) designed to allow "copy" to work–skips all derived info
- **NOSYM** – (optional) does not do symmetry analysis (intended ONLY for testing purposes)

__mul__(*other*)
Multiply by a GroupOp; returns a new supercell (constructed via copy).

Parameters **other** – must be a GroupOp (and *should* be a GroupOp of the supercell!)

Returns rotated supercell

__ne__(*other*)
Inequality == not __eq__

__rmul__(*other*)
Multiply by a GroupOp; returns a new supercell (constructed via copy).

Parameters **other** – must be a GroupOp (and *should* be a GroupOp of the supercell!)

Returns rotated supercell

__sane__()
Return True if supercell occupation and chemorder are consistent

__setitem__(*key*, *value*)
Set specific composition for site; uses same indexing as __getitem__

Parameters

- **key** – index (either an int, a slice, or a position)
- **value** – chemical occupation at that point

__str__()
Human readable version of supercell

__weakref__
list of weak references to the object (if defined)

copy()
Make a copy of the supercell; initializes, then copies over *__copyattr__* and *__eqattr__*.

Returns new supercell object, copy of the original

defectindices()
Return a dictionary that corresponds to the "defect" content of the supercell.

Return defects dictionary, keyed by defect type, with a set of indices of corresponding defects

definesolute(*c*, *chemistry*)
Set the name of the chemistry of chemical index c. Only works for substitutional solutes.

Parameters

- **c** – index
- **chemistry** – string

equivalencemap(*other*)
Given the super *other* we want to find a group operation that transforms *self* into other. This is a GroupOp *along* with an index mapping of chemorder. The index mapping is to get the occposlist to match up: *(g*self).occposlist()[c][mapping[c][i]] == other.occposlist()[c][i]* (We can write a similar expression using chemorder, since chemorder indexes into pos). We're going to return both g and mapping. *Remember: g* does not change the presentation ordering; *mapping* is necessary for full equivalence. If no such equivalence, return None,None.

Parameters **other** – Supercell

Return g GroupOp to transform sites from *self* to *other*

Return mapping list of maps, such that (g*self).chemorder[c][mapping[c][i]] == other.chemorder[c][i]

fillperiodic(*ci*, *Wyckoff=True*)
Occupies all of the (Wyckoff) sites corresponding to chemical index with the appropriate chemistry.

Parameters

- **ci** – tuple of (chem, index) in crystal
- **Wyckoff** – (optional) if False, *only* occupy the specific tuple, but still periodically

Return self

gengroup()
Generate the group operations internal to the supercell

Return G set of GroupOps

index(*pos*, *threshold=1.0*)
Return the index that corresponds to the position *closest* to pos in the supercell. Done in direct coordinates of the supercell, using periodic boundary conditions.

Parameters

- **pos** – 3-vector
- **threshold** – (optional) minimum squared "distance" in supercell for a match; default=1.

Return index index of closest position

makesites()
Generate the array corresponding to the sites; the indexing is based on the translations and the atomindices in crys. These may not all be filled when the supercell is finished.

Return pos array [N*size, 3] of supercell positions in direct coordinates

static maketrans(*super*)
Takes in a supercell matrix, and returns a list of all translations of the unit cell that remain inside the supercell

Parameters **super** – 3x3 integer matrix

Return size integer, corresponding to number of unit cells

Return invsuper integer matrix inverse of supercell (needs to be divided by size)

Return translist list of integer vectors (to be divided by *size*) corresponding to unit cell positions

Return transdict dictionary of tuples and their corresponding index (inverse of trans)

occposlist()
Returns a list of lists of occupied positions, in (chem)order.

Return occposlist list of lists of supercell coord. positions

reorder(*mapping*)
Reorder (in place) the occupied sites. Does not change the occupancies, only the ordering for "presentation".

Parameters **mapping** – list of maps; will make newchemorder[c][i] = chemorder[c][mapping[c][i]]

Return self

If mapping is not a proper permutation, raises ValueError.

setocc(*ind*, *c*)
Set the occupancy of position indexed by ind, to chemistry c. Used by all the other algorithms.

Parameters

- **ind** – integer index
- **c** – chemistry index

stoichiometry()
Return a string representing the current stoichiometry

CHAPTER

EIGHT

POWEREXPANSION

PowerExpansion:

The PowerExpansion module defines the `Taylor3D` class, which is for 3-dimensional (xyz) Taylor expansions of functions. It's primary purpose is to be used in the calculation of the vacancy Green function, as it allows fairly straightforward block evaluation of the small *k* (large distance) transition matrix, and its inverse. This is key to removing the pole in the Green function evaluation. Power expansion class

Class to store and manipulate 3-dimensional Taylor (power) expansions of functions Particularly useful for inverting the FT of the evolution matrix, and subtracting off analytically calculated IFT for the Green function.

Really designed to get used by other code.

class `PowerExpansion.`**`Taylor3D`**(*coefflist=[]*, *Lmax=4*, *nodeepcopy=False*)

Class that stores a Taylor expansion of a function in 3D, and defines some arithmetic

`__add__`(*other*)

Add a set of Taylor expansions

`__call__`(*u*, *fnu=None*)

Method for evaluating our 3D Taylor expansion. We have two approaches: if we are passed a dictionary in fnu that will map (n,l) tuple pairs to either (a) values or (b) functions of a single parameter umagn, then we will compute and return the function value. Otherwise, we return a dictionary mapping (n,l) tuple pairs into values, and leave it at that.

Parameters

- **`u`** – three vector to evaluate; may (or may not) be normalized
- **`fnu`** – dictionary of (n,l): value or function pairs.

Return value or dictionary depending on fnu; default is dictionary

`__getitem__`(*key*)

Indexes (or even slices) into our Taylor expansion.

Parameters **`key`** – indices for our Taylor expansion

Return Taylor3D Taylor expansion after indexing

`__iadd__`(*other*)

Add a set of Taylor expansions

classmethod **`__initTaylor3Dindexing__`**(*Lmax*)

This calls *all* the class methods defined above, and stores them *for the class*. This is intended to be done *once*

Parameters **`Lmax`** – maximum power / orbital angular momentum

__init__(*coefflist=[]*, *Lmax=4*, *nodeepcopy=False*)
Initializes a Taylor3D object, with coefflist (default = empty)

Parameters

- **coefflist** – list((n, lmax, powexpansion)). No type checking; default empty
- **Lmax** – maximum power / orbital angular momentum; can be set only once the first time a Taylor expansion is constructed, and is set for all objects
- **nodeepcopy** – true if we don't want to copy the matrices on creation of object (i.e., deep copy, which is the default) **Note:** deep copy is strongly preferred. The *only* real reason to use nodeepcopy is when returning slices / indexing in arrays, but even then we have to be careful about doing things like reductions, etc., that modify matrices *in place*. We always copy the list, but that doesn't make copies of the underlying matrices.

__isub__(*other*)
Subtract a set of Taylor expansions

__mul__(*other*)
Multiply our expansion

Parameters **other** –

Return Taylor3D expansion of product

__neg__()
Return -T3D

__pos__()
Return +T3D

__radd__(*other*)
Add a set of Taylor expansions

__rmul__(*other*)
Multiply our expansion

Parameters **other** –

Return Taylor3D expansion of product

__rsub__(*other*)
Subtract a set of Taylor expansions

__setitem__(*key*, *value*)
Indexes (or even slices) into our Taylor expansion and "sets"; really only intended to work with another Taylor expansion

Parameters

- **key** – indices for our Taylor expansion
- **value** – assignment value; really, should be

Returns Taylor expansion after indexing

__str__()
Human readable string representation

__sub__(*other*)
Subtract a set of Taylor expansions

__weakref__
list of weak references to the object (if defined)

addhdf5 (*HDF5group*)
Adds an HDF5 representation of object into an HDF5group (needs to already exist). Example: if f is an open HDF5, then T3D.addhdf5(f.create_group('T3D')) will (1) create the group named 'T3D', and then (2) put the T3D representation in that group.

Parameters **HDF5group** – HDF5 group

addterms (*coefflist*)
Add additional coefficients into our object. No type checking. Only works if terms are completely non-overlapping (otherwise, need to use sum).

Parameters **coefflist** – list((n, lmax, powexpansion))

classmethod checkinternalsHDF5 (*HDF5group*)
Reads the power expansion internals into an HDF5group, and performs sanity check

Parameters **HDF5group** – HDF5 group

classmethod coeffproductcoeff (*a*, *b*)
Takes a direction expansion a and b, and returns the product expansion.

Parameters

- **a** – list((n, lmax, powexpansion)
- **b** – list((n, lmax, powexpansion) written as a series of coefficients; n defines the magnitude function, which is additive; lmax is the largest cumulative power of coefficients, and powexpansion is a numpy array that can multiplied. We assume that a and b have consistent shapes throughout–we *do not test this*; runtime will likely fail if not true. The entries in the list are *tuples* of n, lmax, pow

Return c list((n, lmax, powexpansion)), product of *a* and *b*

classmethod collectcoeff (*a*, *inplace=False*, *atol=1e-10*)
Collects coefficients: sums up all the common n values. Best to be done *after* reduce is called.

Parameters

- **a** – list((n, lmax, powexpansion), expansion of function in powers
- **inplace** – modify a in place?

Return coefflist a

classmethod constructexpansion (*basis*, *N=-1*, *pre=None*)
Takes a "basis" for constructing an expansion – list of vectors and matrices – and constructs the expansions up to power N (default = Lmax) Takes a direction expansion a and b, and returns the sum of the expansions.

Parameters

- **= list((coeffmatrix, vect))** (*basis*) – expansions to create; sum(coeffmatrix * (vect*q)^n), for powers n = 0..N
- **N** – maximum power to consider; for N=-1, use Lmax
- **pre** – list of prefactors, defining the Taylor expansion. Default = 1

Return list((n, lmax, powexpansion)),... our expansion, as input to create Taylor3D objects

copy ()
Returns a copy of the current expansion

dumpinternalsHDF5 (*HDF5group*)
Adds the initialized power expansion internals into an HDF5group–should be stored for a sanity check

Parameters **HDF5group** – HDF5 group

ildot(*c*)
Computes $c \cdot self$ in place

inv(*Nmax=0*)
Return the inverse of the expansion, up to order Nmax

Parameters **Nmax** – maximum order in the inverse expansion

Return Taylor3D^-1 Taylor series of inverse

classmethod inversecoeff(*a*, *Nmax=0*)
Takes a direction expansion , and returns the inversion expansion (approximated based on the Taylor expansion of $1/(1-x) = \sum_{i=0}^{\infty} x^i$, or $(A+B)^{-1} = ((1+BA^{-1})A)^{-1} = A^{-1}(1-(-BA^1))^{-1} = A^{-1}\sum_{i=0}(-BA^{-1})^i$

NOTE: assumes SMALLEST n coefficient is the leading order; only works if that coefficient is also isotropic (l=0). Otherwise, raises an error. NOTE: there is no sanity check on whether Nmax is reasonable given the expansion and Lmax values; *caveat emptor*.

Parameters

- **a** – = list((n, lmax, powexpansion) written as a series of coefficients; n defines the magnitude function, which is additive; lmax is the largest cumulative power of coefficients, and powexpansion is a numpy array that can multiplied. We assume that a and b have consistent shapes throughout–we *do not test this*; runtime will likely fail if not true. The entries in the list are *tuples* of n, lmax, pow
- **Nmax** – maximum remaining n value in expansion. Default value of 0 means up to a discontinuity correction in an inversion, but higher (or lower) values are possible.

Return c list((n, lmax, powexpansion)), inverse of a

irdot(*c*)
Computes $self \cdot c$ in place

irotate(*powtrans*)
Rotate in place.

Parameters **powtrans** – Npow x Npow matrix, of [oldpow,newpow] corresponding to the rotation

Returns self

ldot(*c*)
Returns $c \cdot self$

classmethod loadhdf5(*HDF5group*)
Creates a new T3D from an HDF5 group.

Parameters **HDFgroup** – HDF5 group

Return T3D new T3D object

classmethod makeLprojections()
Constructs a series of projection matrices for each l component in our power series

Returns projL[l][p][p'] projection of powers containing *only* l component. -1 component = sum(l=0..Lmax, projL[l]) = simplification projection

classmethod makeYlmpow()
Construct the expansion of the Ylm's in powers of x,y,z. Done via brute force.

Return Ylmpow[lm, p] expansion of each Ylm in powers

classmethod makedirectmult()

Return direcmult[p][p'] index that corresponds to the multiplication of power indices p and p'

static **makeindexPowerYlm**(*Lmax*)

Analyzes the spherical harmonics and powers for a given Lmax; returns a series of index functions.

Parameters **Lmax** – maximum l value to consider; equal to the sum of powers

Return NYlm number of Ylm coefficients

Return Npower number of power coefficients

Return pow2ind[n1][n2][n3] powers to index

Return ind2pow[n] powers for a given index

Return Ylm2ind[l][m] (l,m) to index

Return ind2Ylm[lm] (l,m) for a given index

Return powlrange[l] upper limit of power indices for a given l value; note: [-1] = 0

classmethod **makepowYlm**()

Construct the expansion of the powers in Ylm's. Done using recursion relations instead of direct calculation. Note: an alternative approach would be Gaussian quadrature.

Return powYlm[p][lm] expansion of powers in Ylm; uses indexing scheme above

classmethod **makepowercoeff**()

Make our power coefficients for our construct expansion method

Return powercoeff[n][p] vector we multiply by our power expansion to get the n'th coefficients

classmethod **negcoeff**(*a*)

Negates a coefficient expansion a

Parameters **= list((n, lmax, powexpansion)** (*a*) – expansion of function in powers

Return coefflist -a

nl()

Returns a list of (n,l) pairs in the coefflist

Return nl_list all of the (n,l) pairs that are present in our coefflist

classmethod **powexp**(*u*, *normalize=True*)

Given a vector u, normalize it and return the power expansion of uvec

Parameters

- **u[3]** – vector to apply
- **normalize** – do we normalize u first?

Return upow[Npower] ux uy uz products of powers

Return umagn magnitude of u (if normalized)

rdot(*c*)

Returns $self \cdot c$

reduce()

Reduce the coefficients: eliminate any n that has zero coefficients, collect all of the same values of n together. Done in place.

classmethod **reducecoeff**(*a*, *inplace=False*, *atol=1e-10*)

Projects coefficients through Ylm space, then eliminates any zero contributions (including possible reduction in l values, too).

Parameters

- **a** – list((n, lmax, powexpansion), expansion of function in powers
- **inplace** – modify a in place?

Return coefflist a

rotate(*powtrans*)
Return a rotated version of the expansion.

Parameters **powtrans** – Npow x Npow matrix, of [oldpow,newpow] corresponding to the rotation

Return rTaylor3D Taylor expansion, rotated

classmethod rotatecoeff(*a*, *npowtrans*, *inplace=False*)
Return a rotated version of the expansion. Needs to use pad to work with reduced representations.

Parameters

- **a** – coefficiant list
- **npowtrans** – Lmax+1 x Npow x Npow matrix, of [n,oldpow,newpow] corresponding to the rotation

Return rcoeff coefficient list, rotated

classmethod rotatedirections(*qptrans*)
Takes a transformation matrix qptrans, where q[i] = sum_j qptrans[i][j] p[j], and returns the Npow x Npow transformation matrix for the new components in terms of the old. NOTE: This is more complex than one might first realize. If we only work with cases where all of the entries for a given power n have those same n (that is, not reduced), then this is straightforward. However, we run into problems with *reductions*: e.g., for n=2, the power $x^0y^0z^0$ is, in reality, $x^2 + y^2 + z^2$, and hence *it must be transformed* because we allow non-orthogonal transformation matrices.

Parameters **qptrans** – 3x3 matrix

Return npowtrans [Lmax +1][Npow][Npow] transformation matrix [n][original pow][new pow] for each n from 0 up to Lmax

classmethod scalarproductcoeff(*c*, *a*, *inplace=False*)
Multiplies an coefficient expansion a by a scalar c

Parameters

- **c** – scalar *or* dictionary mapping (n,l) to scalars
- **= list((n, lmax, powexpansion)** (a) – expansion of function in powers
- **inplace** – modify a in place?

Return coefflist c*a

separate()
Separate out the coefficients into (n,l) terms where *only* l contributions appear in each.

classmethod separatecoeff(*a*, *inplace=False*, *atol=1e-10*)
Projects coefficients through Ylm space, one by one. Assumes they've already been reduced and collected first; if not, could lead to duplicated (n,l) entries in list, which is inefficient (should still *evaluate* the same, just with extra steps). After this, each (n,l) term *only* contains terms equal to l, rather than terms <= l.

Parameters

- **a** – list((n, lmax, powexpansion), expansion of function in powers

- **inplace** – modify a in place?

Return coefflist a

classmethod sumcoeff(*a*, *b*, *alpha=1*, *beta=1*, *inplace=False*)
Takes Taylor3D expansion a and b, and returns the sum of the expansions.

Param a, b = list((n, lmax, powexpansion) written as a series of coefficients; n defines the magnitude function, which is additive; lmax is the largest cumulative power of coefficients, and powexpansion is a numpy array that can multiplied. We assume that a and b have consistent shapes throughout–we *do not test this*; runtime will likely fail if not true. The entries in the list are *tuples* of n, lmax, pow

Parameters

- **beta** (*alpha,*) – optional scalars: c = alpha*a + beta*b; allows for more efficient expansions
- **inplace** – True if the summation should modify a in place

Return c coeff of sum of a and b (! NOTE ! does not return the class!) sum of a and b

classmethod tensorproductcoeff(*c*, *a*, *leftmultiply=True*)
Multiplies an coefficient expansion a by a scalar c

Parameters

- **c** – array *or* dictionary mapping (n,l) to arrays
- **= list((n, lmax, powexpansion)** (*a*) – expansion of function in powers
- **leftmultiply** – tensordot(c,a) vs. tensordot(a,c)

Return coefflist c.a (or a.c)

truncate(*Nmax*, *inplace=False*)
Remove the coefficients above a given Nmax; normally returns a new object

Parameters

- **Nmax** – maximum coefficient to include
- **inplace** – do it in place?

classmethod truncatecoeff(*a*, *Nmax*, *inplace=False*)
Remove the coefficients above a given Nmax; normally returns a new object

Parameters

- **Nmax** – maximum coefficient to include
- **a** – list((n, lmax, powexpansion), expansion of function in powers
- **inplace** – do it in place?

classmethod zeros(*nmin*, *nmax*, *shape*, *dtype=<class 'complex'>*)
Constructs (and returns) a "zero" Taylor expansion with the prescribed shape. This will be useful for doing slicing assignments. Because of the manner in which slicing works for assignment, we create what looks like a *lot* of zeros, by explicitly making the full range of l values.

Parameters

- **nmin** – minimum value of n
- **nmax** – maximum value of n (inclusive)
- **shape** – shape of matrix, as zeros would expect.

Return Taylor3D Taylor3D, with a zero coefficient list

CHAPTER

NINE

GFCALC

GFcalc:

The GFcalc module defines the `GFCrystalcalc` class for the evaluation of the vacancy Green function. GFcalc module

Code to compute the lattice Green function for diffusion; this entails inverting the "diffusion" matrix, which is infinite, singular, and has translational invariance. The solution involves fourier transforming to reciprocal space, inverting, and inverse fourier transforming back to real (lattice) space. The complication is that the inversion produces a second order pole which must be treated analytically. Subtracting off the pole then produces a discontinuity at the gamma-point (q=0), which also should be treated analytically. Then, the remaining function can be numerically inverse fourier transformed.

class `GFcalc.`**`GFCrystalcalc`**(*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nmax=4*)

Class calculator for the Green function, designed to work with the Crystal class.

This computes the bare vacancy GF. It requires a crystal, chemical identity for the vacancy, list of symmetry unique sites (to define energies / entropies uniquely), and a corresponding jumpnetwork for that vacancy.

`BlockInvertOmegaTaylor`(*dd*, *dr*, *rd*, *rr*, *D*)

Returns block inverted omega as a Taylor expansion, up to Nmax = 0 (discontinuity correction). Needs to be rotated such that leading order of D is isotropic.

Parameters

- **`dd`** – diffusive/diffusive block (upper left)
- **`dr`** – diffusive/relaxive block (lower left)
- **`rd`** – relaxive/diffusive block (upper right)
- **`rr`** – relaxive/relaxive block (lower right)
- **`D`** – $dd - dr(rr)^{-1}rd$ (diffusion)

Return gT Taylor expansion of g in block form, and reduced (collected terms)

`BlockRotateOmegaTaylor`(*omega_Taylor_rotate*)

Returns block partitioned Taylor expansion of a rotated omega Taylor expansion.

Parameters

- **`omega_Taylor_rotate`** – rotated into diffusive [0] / relaxive [1:] basis
- **`dd`** – diffusive/diffusive block (upper left)
- **`dr`** – diffusive/relaxive block (lower left)
- **`rd`** – relaxive/diffusive block (upper right)
- **`rr`** – relaxive/relaxive block (lower right)

- **D** – $dd - dr(rr)^{-1}rd$ (diffusion)

BreakdownGroups()
Takes in a crystal, and a chemistry, and constructs the indexing breakdown for each (i,j) pair. :return grouparray: array[NG][3][3] of the NG group operations :return indexpair: array[N][N][NG][2] of the index pair for each group operation

DiagGamma(*omega=None*)
Diagonalize the gamma point (q=0) term

Parameters **omega** – optional; the Taylor expansion to use. If None, use self.omega_Taylor

Return r array of eigenvalues, sorted from 0 to decreasing values.

Return vr array of eigenvectors where vr[:,i] is the vector for eigenvalue r[i]

Diffusivity(*omega_Taylor_D=None*)
Return the diffusivity, or compute it if it's not already known. Uses omega_Taylor_D to compute with maximum efficiency.

Parameters **omega_Taylor_D** – Taylor expansion of the diffusivity component

Return D diffusivity [3,3] array

FourierTransformJumps(*jumpnetwork*, *N*, *kpts*)
Generate the Fourier transform coefficients for each jump

Parameters

- **jumpnetwork** – list of unique transitions, as lists of ((i,j), dx)
- **N** – number of sites
- **kpts** – array[Nkpt][3], in Cartesian (same coord. as dx)

Return FTjumps array[Njump][Nkpt][Nsite][Nsite] of FT of the jump network

Return SEjumps array[Nsite][Njump] multiplicity of jump on each site

SetRates(*pre*, *betaene*, *preT*, *betaeneT*)
(Re)sets the rates, given the prefactors and Arrhenius factors for the sites and transitions, using the ordering according to sitelist and jumpnetwork. Initiates all of the calculations so that GF calculation is (fairly) efficient for each input.

Parameters

- **pre** – list of prefactors for site probabilities
- **betaene** – list of beta*E (energy/kB T) for each site
- **preT** – list of prefactors for transition states
- **betaeneT** – list of beta*ET (energy/kB T) for each transition state

SymmRates(*pre*, *betaene*, *preT*, *betaeneT*)
Returns a list of lists of symmetrized rates, matched to jumpnetwork

TaylorExpandJumps(*jumpnetwork*, *N*)
Generate the Taylor expansion coefficients for each jump

Parameters

- **jumpnetwork** – list of unique transitions, as lists of ((i,j), dx)
- **N** – number of sites

Return T3Djumps list of Taylor3D expansions of the jump network

__call__(*i*, *j*, *dx*)
Evaluate the Green function from site i to site j, separated by vector dx

Parameters

- **i** – site index
- **j** – site index
- **dx** – vector pointing from i to j (can include lattice contributions)

Return G Green function value

__init__(*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nmax=4*)
Initializes our calculator with the appropriate topology / connectivity. Doesn't require, at this point, the site probabilities or transition rates to be known.

Parameters

- **crys** – Crystal object
- **chem** – index identifying the diffusing species
- **sitelist** – list, grouped into Wyckoff common positions, of unique sites
- **jumpnetwork** – list of unique transitions as lists of ((i,j), dx)
- **Nmax** – maximum range as estimator for kpt mesh generation

__weakref__
list of weak references to the object (if defined)

addhdf5(*HDF5group*)
Adds an HDF5 representation of object into an HDF5group (needs to already exist).

Example: if f is an open HDF5, then GFcalc.addhdf5(f.create_group('GFcalc')) will (1) create the group named 'GFcalc', and then (2) put the GFcalc representation in that group.

Parameters **HDF5group** – HDF5 group

biascorrection(*etav=None*)
Return the bias correction, or compute it if it's not already known. Uses etav to compute.

Parameters **etav** – Taylor expansion of the bias correction

Return eta [N,3] array

exp_dxq(*dx*)
Return the array of exp(-i q.dx) evaluated over the q-points, and accounting for symmetry

Parameters **dx** – vector

Return exp(-i q.dx) array of $\exp(-i \cdot dx)$

classmethod loadhdf5(*crys*, *HDF5group*)
Creates a new GFcalc from an HDF5 group.

Parameters

- **crys** – crystal object–MUST BE PASSED IN as it is not stored with the GFcalc
- **HDFgroup** – HDF5 group

Return GFcalc new GFcalc object

CHAPTER

TEN

ONSAGERCALC

OnsagerCalc:

The OnsagerCalc module defines the `Intersitial` class (for computation of interstitial-mediated diffusion), and `VacancyMediated` class (for computation of vacancy-mediated diffusion). Onsager calculator module: Interstitialcy mechanism and Vacancy-mediated mechanism

Class to create an Onsager "calculator", which brings two functionalities: 1. determines *what* input is needed to compute the Onsager (mobility, or L) tensors 2. constructs the function that calculates those tensors, given the input values.

This class is designed to be combined with code that can, e.g., automatically run some sort of atomistic-scale (DFT, classical potential) calculation of site energies, and energy barriers, and then in concert with scripts to convert such data into rates and probabilities, this will allow for efficient evaluation of transport coefficients.

This implementation will be for vacancy-mediated solute diffusion assumes the dilute limit. The mathematics is based on a Green function solution for the vacancy diffusion. The computation of the GF is included in the GFcalc module.

Now with HDF5 write / read capability for VacancyMediated module

class `OnsagerCalc.`**`Interstitial`**(*crys*, *chem*, *sitelist*, *jumpnetwork*)

A class to compute interstitial diffusivity; uses structure of crystal to do most of the heavy lifting in terms of symmetry.

Takes in a crystal that contains the interstitial as one of the chemical elements, to be specified by *chem*, the sitelist (list of symmetry equivalent sites), and jumpnetwork. Both of the latter can be computed automatically from *crys* methods, but as they are lists, can also be editted or constructed by hand.

`__init__`(*crys*, *chem*, *sitelist*, *jumpnetwork*)

Initialization; takes an underlying crystal, a choice of atomic chemistry, a corresponding Wyckoff site list and jump network.

Parameters

- **`crys`** – Crystal object
- **`chem`** – integer, index into the basis of crys, corresponding to the chemical element that hops
- **`sitelist`** – list of lists of indices, site indices where the atom may hop; grouped by symmetry equivalency
- **`jumpnetwork`** – list of lists of tuples: ((i, j), dx) symmetry unique transitions; each list is all of the possible transitions from site i to site j with jump vector dx; includes i->j and j->i

`__weakref__`

list of weak references to the object (if defined)

diffusivity(*pre*, *betaene*, *preT*, *betaeneT*, *CalcDeriv=False*)
Computes the diffusivity for our element given prefactors and energies/kB T. Also returns the negative derivative of diffusivity with respect to beta (used to compute the activation barrier tensor) if CalcDeriv = True The input list order corresponds to the sitelist and jumpnetwork

Parameters

- **pre** – list of prefactors for unique sites
- **betaene** – list of site energies divided by kB T
- **preT** – list of prefactors for transition states
- **betaeneT** – list of transition state energies divided by kB T

Return D[3,3] diffusivity as a 3x3 tensor

Return DE[3,3] diffusivity times activation barrier (if CalcDeriv == True)

elastodiffusion(*pre*, *betaene*, *dipole*, *preT*, *betaeneT*, *dipoleT*)
Computes the elastodiffusion tensor for our element given prefactors, energies/kB T, and elastic dipoles/kB T The input list order corresponds to the sitelist and jumpnetwork

Parameters

- **pre** – list of prefactors for unique sites
- **betaene** – list of site energies divided by kB T
- **dipole** – list of elastic dipoles divided by kB T
- **preT** – list of prefactors for transition states
- **betaeneT** – list of transition state energies divided by kB T
- **dipoleT** – list of elastic dipoles divided by kB T

Return D[3,3] diffusivity as 3x3 tensor

Return dD[3,3,3,3] elastodiffusion tensor as 3x3x3x3 tensor

generateJumpGroupOps()
Generates a list of group operations that transform the first jump in the jump network into all of the other members; one group operation for each.

Return siteGroupOps list of list of group ops that mirrors the structure of jumpnetwork.

generateJumpSymmTensorBasis()
Generates a list of symmetric tensor bases for the first representative transition in our jump network

Return TensorSet list of list of symmetric tensors

generateSiteGroupOps()
Generates a list of group operations that transform the first site in each site list into all of the other members; one group operation for each.

Return siteGroupOps list of list of group ops that mirrors the structure of site list

generateSiteSymmTensorBasis()
Generates a list of symmetric tensor bases for the first representative site in our site list.

Return TensorSet list of symmetric tensors

jumpDipoles(*dipoles*)
Returns a list of the elastic dipole for each transition, given the dipoles for the representatives. ("populating" the full set of dipoles)

Parameters **dipoles** – list of dipoles for the first representative transition

Return dipolelist list of lists of dipole for each jump[site][3][3]

static **jumpnetworkYAML**(*jumpnetwork*)
Dumps a "sample" YAML formatted version of the jumpnetwork with data to be entered

ratelist(*pre*, *betaene*, *preT*, *betaeneT*)
Returns a list of lists of rates, matched to jumpnetwork

siteDipoles(*dipoles*)
Returns a list of the elastic dipole on each site, given the dipoles for the representatives. ("populating" the full set of dipoles)

Parameters **dipoles** – list of dipoles for the first representative site

Return dipolelist array of dipole for each site [site][3][3]

static **sitelistYAML**(*sitelist*)
Dumps a "sample" YAML formatted version of the sitelist with data to be entered

siteprob(*pre*, *betaene*)
Returns our site probabilities, normalized, as a vector

symmratelist(*pre*, *betaene*, *preT*, *betaeneT*)
Returns a list of lists of symmetrized rates, matched to jumpnetwork

class OnsagerCalc.**VacancyMediated**(*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nthermo=0*)
A class to compute vacancy-mediated solute transport coefficients, specifically L_vv (vacancy diffusion), L_ss (solute), and L_sv (off-diagonal). As part of that, it determines *what* quantities are needed as inputs in order to perform this calculation.

Based on crystal class. Also now includes its own GF calculator and cacheing, and storage in HDF5 format.

Requires a crystal, chemical identity of vacancy, list of symmetry-equivalent sites for that chemistry, and a jumpnetwork for the vacancy. The thermodynamic range (number of "shells" – see *crystalStars.StarSet* for precise definition).

Lij(*bFV*, *bFS*, *bFSV*, *bFT0*, *bFT1*, *bFT2*)
Calculates the transport coefficients: Lvv, L0ss, L2ss, L1sv, L1vv from the scaled free energies. The Green function entries are calculated from the omega0 info. As this is the most time-consuming part of the calculation, we cache these values with a dictionary and hash function. Used by Lij.

Parameters

- **bFV[NWyckoff]** – beta*eneV - ln(preV) (relative to minimum value)
- **bFS[NWyckoff]** – beta*eneS - ln(preS) (relative to minimum value)
- **bFSV[Nthermo]** – beta*eneSV - ln(preSV) (excess)
- **bFT0[Nomega0]** – beta*eneT0 - ln(preT0) (relative to minimum value of bFV)
- **bFT1[Nomega1]** – beta*eneT1 - ln(preT1) (relative to minimum value of bFV + bFS)
- **bFT2[Nomega2]** – beta*eneT2 - ln(preT2) (relative to minimum value of bFV + bFS)

Return Lvv[3, 3] vacancy-vacancy; needs to be multiplied by cv/kBT

Return Lss[3, 3] solute-solute; needs to be multiplied by cv*cs/kBT

Return Lsv[3, 3] solute-vacancy; needs to be multiplied by cv*cs/kBT

Return Lvv1[3, 3] vacancy-vacancy correction due to solute; needs to be multiplied by cv*cs/kBT

__init__ (*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nthermo=0*)
Create our diffusion calculator for a given crystal structure, chemical identity, jumpnetwork (for the vacancy) and thermodynamic shell.

Parameters

- **crys** – Crystal object
- **chem** – index identifying the diffusing species
- **sitelist** – list, grouped into Wyckoff common positions, of unique sites
- **jumpnetwork** – list of unique transitions as lists of ((i,j), dx)

__weakref__
list of weak references to the object (if defined)

addhdf5 (*HDF5group*)
Adds an HDF5 representation of object into an HDF5group (needs to already exist).

Example: if f is an open HDF5, then VacancyMediated.addhdf5(f.create_group('Diffuser')) will (1) create the group named 'Diffuser', and then (2) put the VacancyMediated representation in that group.

Parameters **HDF5group** – HDF5 group

generate (*Nthermo*)
Generate the necessary stars, vector-stars, and jump networks based on the thermodynamic range.

Parameters **Nthermo** – range of thermodynamic interactions, in terms of "shells", which is multiple summations of jumpvect

generatematrices ()
Generates all the matrices and "helper" pieces, based on our jump networks. This has been separated out in case the user wants to, e.g., prune / modify the networks after they've been created with generate(), then generatematrices() can be rerun.

generatetags ()
Create tags for vacancy states, solute states, solute-vacancy complexes; omega0, omega1, and omega2 transition states.

Return tags dictionary of tags; each is a list-of-lists

Return tagdict dictionary that maps tag into the index of the corresponding list.

Return tagdicttype dictionary that maps tag into the key for the corresponding list.

interactlist ()
Return a list of solute-vacancy configurations for interactions. The points correspond to a vector between a solute atom and a vacancy. Defined by Stars.

Return statelist list of PairStates for the solute-vacancy interactions

classmethod **loadhdf5** (*HDF5group*)
Creates a new VacancyMediated diffuser from an HDF5 group.

Parameters **HDFgroup** – HDF5 group

Return VacancyMediated new VacancyMediated diffuser object from HDF5

makeLIMBpreene (*preS*, *eneS*, *preSV*, *eneSV*, *preT0*, *eneT0*, ***ignoredextraarguments*)
Generates corresponding energies / prefactors for corresponding to LIMB (Linearized interpolation of migration barrier approximation). Returns a dictionary. (we ignore extra arguments so that a dictionary including additional entries can be passed)

Parameters

- **preS[NWyckoff]** – prefactor for solute formation
- **eneS[NWyckoff]** – solute formation energy
- **preSV[Nthermo]** – prefactor for solute-vacancy interaction
- **eneSV[Nthermo]** – solute-vacancy binding energy
- **preT0[Nomeg0]** – prefactor for vacancy jump transitions (follows jumpnetwork)
- **eneT0[Nomega0]** – transition energy for vacancy jumps

Return preT1[Nomega1] prefactor for omega1-style transitions (follows om1_jn)

Return eneT1[Nomega1] transition energy/kBT for omega1-style jumps

Return preT2[Nomega2] prefactor for omega2-style transitions (follows om2_jn)

Return eneT2[Nomega2] transition energy/kBT for omega2-style jumps

maketracerpreene(*preT0*, *eneT0*, ***ignoredextraarguments*)

Generates corresponding energies / prefactors for an isotopic tracer. Returns a dictionary. (we ignore extra arguments so that a dictionary including additional entries can be passed)

Parameters

- **preT0[Nomeg0]** – prefactor for vacancy jump transitions (follows jumpnetwork)
- **eneT0[Nomega0]** – transition energy state for vacancy jumps

Return preS[NWyckoff] prefactor for solute formation

Return eneS[NWyckoff] solute formation energy

Return preSV[Nthermo] prefactor for solute-vacancy interaction

Return eneSV[Nthermo] solute-vacancy binding energy

Return preT1[Nomega1] prefactor for omega1-style transitions (follows om1_jn)

Return eneT1[Nomega1] transition energy for omega1-style jumps

Return preT2[Nomega2] prefactor for omega2-style transitions (follows om2_jn)

Return eneT2[Nomega2] transition energy for omega2-style jumps

omegalist(*fivefreqindex=1*)

Return a list of pairs of endpoints for a vacancy jump, corresponding to omega1 or omega2 Solute at the origin, vacancy hopping between two sites. Defined by om1_jumpnetwork

Parameters **fivefreqindex** – 1 or 2, corresponding to omega1 or omega2

Return omegalist list of tuples of PairStates

Return omegajumptype index of corresponding omega0 jumptype

static preene2betafree(*kT*, *preV*, *eneV*, *preS*, *eneS*, *preSV*, *eneSV*, *preT0*, *eneT0*, *preT1*, *eneT1*, *preT2*, *eneT2*, ***ignoredextraarguments*)

Read in a series of prefactors ($\exp(S/k_B)$) and energies, and return βF for energies and transition state energies. Used to provide scaled values to Lij() and _lij(). Can specify all of the entries using a dictionary; e.g., *preene2betafree(kT, **data_dict)* and then send that output as input to Lij: *Lij(*preene2betafree(kT, **data_dict))* (we ignore extra arguments so that a dictionary including additional entries can be passed)

Parameters

- **kT** – temperature times Boltzmann's constant kB
- **preV** – prefactor for vacancy formation (prod of inverse vibrational frequencies)

- **eneV** – vacancy formation energy
- **preS** – prefactor for solute formation (prod of inverse vibrational frequencies)
- **eneS** – solute formation energy
- **preSV** – excess prefactor for solute-vacancy binding
- **eneSV** – solute-vacancy binding energy
- **preT0** – prefactor for vacancy transition state
- **eneT0** – energy for vacancy transition state (relative to eneV)
- **preT1** – prefactor for vacancy swing transition state
- **eneT1** – energy for vacancy swing transition state (relative to eneV + eneS + eneSV)
- **preT2** – prefactor for vacancy exchange transition state
- **eneT2** – energy for vacancy exchange transition state (relative to eneV + eneS + eneSV)

Return bFV beta*eneV - ln(preV) (relative to minimum value)

Return bFS beta*eneS - ln(preS) (relative to minimum value)

Return bFSV beta*eneSV - ln(preSV) (excess)

Return bFT0 beta*eneT0 - ln(preT0) (relative to minimum value of bFV)

Return bFT1 beta*eneT1 - ln(preT1) (relative to minimum value of bFV + bFS)

Return bFT2 beta*eneT2 - ln(preT2) (relative to minimum value of bFV + bFS)

tags2preene(*usertagdict*, *VERBOSE=False*)
Generates energies and prefactors based on a dictionary of tags.

Parameters

- **usertagdict** – dictionary where the keys are tags, and the values are tuples: (pre, ene)
- **VERBOSE** – (optional) if True, also return a dictionary of missing tags, duplicate tags, and bad tags

Return thermodict dictionary of ene's and pre's corresponding to usertagdict

Return missingdict dictionary with keys corresponding to tag types, and the values are lists of lists of symmetry equivalent tags that are missing

Return duplicatelist list of lists of tags in usertagdict that are (symmetry) duplicates

Return badtaglist list of all tags in usertagdict that aren't found in our dictionary

class OnsagerCalc.**VacancyMediatedMeta**(*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nthermo=0*, *meta_tags=[]*, *jumpnetwork2=[]*)
Trying out metastable preening

__init__(*crys*, *chem*, *sitelist*, *jumpnetwork*, *Nthermo=0*, *meta_tags=[]*, *jumpnetwork2=[]*)
will fill it later

generatekineticmeta(*Nthermo*)
no idea. will fill it later

generatethermometa(*Nthermo*)
will fill it later

gfconstruct(*Nthermo*)
no idea. will fill it later

OnsagerCalc.**arrays2vTKdict**(*vTKarray*, *valarray*, *vTKsplits*)
Takes two arrays of vTK keys and values, and the splits to separate vTKarray back into vTK and returns a dictionary indexed by the vTK.

Parameters

- **vTKarray** – array of vTK entries
- **valarray** – array of values
- **vTKsplits** – split placement for vTK entries

Return vTKdict dictionary, indexed by vTK objects, whose entries are arrays

OnsagerCalc.**vTKdict2arrays**(*vTKdict*)
Takes a dictionary indexed by vTK objects, returns two arrays of vTK keys and values, and the splits to separate vTKarray back into vTK

Parameters **vTKdict** – dictionary, indexed by vTK objects, whose entries are arrays

Return vTKarray array of vTK entries

Return valarray array of values

Return vTKsplits split placement for vTK entries

class OnsagerCalc.**vacancyThermoKinetics**
Class to store (in a hashable manner) the thermodynamics and kinetics for the vacancy

Parameters

- **pre** – prefactors for sites
- **betaene** – energy for sites / kBT
- **preT** – prefactors for transition states
- **betaeneT** – transition state energy for sites / kBT

static vacancyThermoKinetics_constructor(*loader*, *node*)
Construct a GroupOp from YAML

static vacancyThermoKinetics_representer(*dumper*, *data*)
Output a PairState

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

S

T

U

V

W

Z