Onsager Documentation

Release 1.2.1

Dallas R. Trinkle

Apr 04, 2017

CONTENTS

Documentation for `Onsager` python module for automated computation of diffusivity for interstitial and vacancy mediated diffusion.

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)` or see Onsager github for current version doi information.

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 can be seen in the ipython notebooks in *examples*; the usual import will be:

```
#!/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 available 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 script *CLdiffuser* is a command-line diffuser calculator that is designed to read in an HDF5 file of a diffuser, along with a JSON file that includes the thermal/kinetic data, and computes diffusivity components for different temperatures.

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 format for input/output of crystal structures, and 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. The vacancy-mediated diffuser can be stored as an HDF5 file (which internally stores the crystal structure in YAML format). The thermal/kinetic data is most easily serialized as JSON, but any dictionary-compatible format will do, by making use of tags.

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.
- 1.0 Now including automator for supercell calculations.

- 1.1 Automator update with Makefile; corrections for possible overflow error when omega2 gets very large.
- 1.2 Combined "large omega2" and "non-zero bias" algorithms; notebook for Fe-C added to documentation; cleanup of code and improved testing.
- 1.2.1 Additional notebooks added for vacancy-mediated diffuser.

CHAPTER

TWO

REFERENCES

- Dallas R. Trinkle, "Diffusivity and derivatives for interstitial solutes: Activation energy, volume, and elastodiffusion tensors." Philos. Mag. (2016) [doi:10.1080/14786435.2016.1212175](http://dx.doi.org/10.1080/14786435.2016.1212175); [arXiv:1605.03623](http://arxiv.org/abs/1605.03623)
- Dallas R. Trinkle, "Automatic numerical evaluation of vacancy-mediated transport for arbitrary crystals: Onsager coefficients in the dilute limit using a Green function approach." [arXiv:1608.01252](http://arxiv.org/abs/1608.01252)

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*, *threshold=1e-08*)
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*, *threshold=1e-08*)
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
- **threshold** – threshold for symmetry equivalence (stored in the class)

__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=None*)

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

Parameters threshold – threshold for symmetry comparison; default = self.threshold

Algorithm is slightly complicated: we attempt to identify if there is a internal translation symmetry in the crystal (called t) that applies to all sites. Once identified, we transform the lattice vectors and basis into the "reduced" form of the cell. We use tail recursion to continue until no further reduction is possible. Will usually require some "polishing" on the unit cell after the fact.

We try to do this efficiently: we check the GCD of the site counts (called M); if it's 1, we kick out. We check translations against the smallest site set first.

We try to do this carefully: We make sure that our translation can be expressed rationally with M as the denominator; this helps protect against roundoff error. When we reduce the atomic basis, we *average* the values that match. Finally, as we reduce, we also change the *self.threshold* value accordingly so that recursion uses the same "effective" threshold.

reducekptmesh(*kptfull*, *threshold=None*)
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

`crystal.`**`FourthRankIsotropic`**(*F*)

Returns the average and shear values from orientational averaging of a symmetric fourth-rank tensor.

Parameters **`F[a,b,c,d]`** – symmetric fourth-rank tensor (F[abcd]=F[abcd]=F[bacd]=F[cdab])

Return average average value = (F11+2F12)/3, orientationally averaged

Return shear shear value = F44, orientationally averaged

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.`**`gcdlist`**`(`*lis*`)`
Returns the GCD of a list of integers

`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.`**`isotropicFourthRank`**`(`*average*, *shear*`)`
Returns a symmetrized, isotropic fourth-rank tensor based on an average value and "shear" value

Parameters

- **average** – averaged value = (F11+2F12)/3
- **shear** – shear value = F44 = (F11-F12)/2

Return F[a,b,c,d] isotropic fourth-rank tensor

`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*, *originstates=False*, *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

__contains__(*PS*)
Return true if PS is in the star

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

__init__(*jumpnetwork*, *crys*, *chem*, *Nshells=0*, *originstates=False*, *lattice=False*)
Initiates a star set generator for a given jumpnetwork, crystal, and specified chemical index. Does not include "origin states" by default; these are PairStates that iszero() is True; they are only needed if crystal has a nonzero VectorBasis.

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
- **originstates** – include origin states in 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. 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. Does not include "origin states" by default; these are PairStates that iszero() is True; they are only needed if crystal has 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, and needed to construct delta_omega

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, and needed to construct delta_omega

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.**VectorStarSet**(*starset=None*)
A class to construct vector star sets, and be able to efficiently index.

All based on a StarSet

GFexpansion()
Construct the GF matrix expansion in terms of the star vectors, and indexed to GFstarset.

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

__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*, *omega2=False*)

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
- **omega2** – (optional) are we dealing with the omega2 list, so we need to remove origin states? (default=False)

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

originstateVectorBasisfolddown(*elemtype='solute'*)

Construct the expansion to "fold down" from vector stars to origin states.

Parameters **elemtype** – 'solute' of 'vacancy', depending on which site we need to reduce

Return OSindices list of indices corresponding to origin states

Return folddown [NOS, Nvstars] to map vector stars to origin states

Return OS_VB [NOS, Nsites, 3] mapping of origin state to a vector basis

rateexpansions(*jumpnetwork*, *jumptype*, *omega2=False*)

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
- **omega2** – (optional) are we dealing with the omega2 list, so we need to remove origin states? (default=False)

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]*probfactor0(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]*omega1[k]*probfactor(PS[k]))

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

`crystalStars.`**`zeroclean`**(*x*, *threshold=1e-08*)

Modify x in place, return 0 if x is below a threshold; useful for "symmetrizing" our expansions

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 $x^0 y^0 z^0$ 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*, *pmaxerror=1e-08*)
(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
- **pmaxerror** – parameter controlling error from pmax value. Should be same order as integration error.

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

static **networkcount**(*jumpnetwork*, *N*)

Return a count of how many separate connected networks there are

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

`__str__`()

Human readable version of diffuser

`__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

generatetags()
Create tags for unique interstitial states, and 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.

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

losstensors(*pre*, *betaene*, *dipole*, *preT*, *betaeneT*)
Computes the internal friction loss tensors 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

Return lambdaL list of tuples of (eigenmode, L-tensor) where L-tensor is a 3x3x3x3 loss tensor L-tensor needs to be multiplied by kB T to have proper units of energy.

makesupercells(*super_n*)
Take in a supercell matrix, then generate all of the supercells needed to compute site energies and transitions (corresponding to the representatives).

Parameters **super_n** – 3x3 integer matrix to define our supercell

Return superdict dictionary of `states`, `transitions`, `transmapping`, and `indices` that correspond to dictionaries with tags.

- superdict['states'][i] = supercell of site;
- superdict['transitions'][n] = (supercell initial, supercell final);
- superdict['transmapping'][n] = ((site tag, groupop, mapping), (site tag, groupop, mapping))
- superdict['indices'][tag] = index of tag, where tag is either a state or transition tag.

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*, *NGFmax=4*)
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).

GFcalculator(*NGFmax=0*)
Return the GF calculator; create a new one if NGFmax is being changed

Lij(*bFV*, *bFS*, *bFSV*, *bFT0*, *bFT1*, *bFT2*, *large_om2=100000000.0*)
Calculates the transport coefficients: L0vv, Lss, Lsv, 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.

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)
- **large_om2** – threshold for changing treatment of omega2 contributions (default: 10^8)

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*, *NGFmax=4*)
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)
- **Nthermo** – range of thermodynamic interaction (in successive jumpnetworks)
- **NGFmax** – parameter controlling k-point density of GF calculator; 4 seems reasonably accurate

`__str__()`
Human readable version of diffuser

`__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

`clearcache()`
Clear out the GF cache values

`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

makesupercells(*super_n*)

Take in a supercell matrix, then generate all of the supercells needed to compute site energies and transitions (corresponding to the representatives).

Note: the states are lone vacancy, lone solute, solute-vacancy complexes in our thermodynamic range. Note that there will be escape states are endpoints of some omega1 jumps. They are not relaxed, and have no pre-existing tag. They will only be output as a single endpoint of an NEB run; there may be symmetry equivalent duplicates, as we construct these supercells on an as needed basis.

We've got a few classes of warnings (from most egregious to least) that can issued if the supercell is too small; the analysis will continue despite any warnings:

1. Thermodynamic shell states map to different states in supercell
2. Thermodynamic shell states are not unique in supercell (multiplicity)
3. Kinetic shell states map to different states in supercell
4. Kinetic shell states are not unique in supercell (multiplicity)

The lowest level can still be run reliably but runs the risk of errors in escape transition barriers. Extreme caution should be used if any of the other warnings are raised.

Parameters **super_n** – 3x3 integer matrix to define our supercell

Return superdict dictionary of `states`, `transitions`, `transmapping`, `indices` that correspond to dictionaries with tags; the final tag `reference` is the basesupercell for calculations without defects.

- superdict['states'][i] = supercell of state;
- superdict['transitions'][n] = (supercell initial, supercell final);
- superdict['transmapping'][n] = ((site tag, groupop, mapping), (site tag, groupop, mapping))
- superdict['indices'][tag] = (type, index) of tag, where tag is either a state or transition tag.
- superdict['reference'] = supercell reference, no defects

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 ($\exp(S/k_\mathrm{B})$) and energies, and return βF for energies and transition state energies. Used to provide scaled values to 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

`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

AUTOMATOR

Automator:

The automator module defines functions that create a tarball from a supercell dictionary. Automator code

Functions to convert from a supercell dictionary (output from a Diffuser) into a tarball that contains all of the input files in an organized directory structure to run the atomic-scale transition state calculations. This includes:

1. All positions in POSCAR format (POSCAR files for states to relax, POS as reference for transition endpoints that need to be relaxed)
2. Transformation information from relaxed states to initial states.
3. INCAR files for relaxation and NEB runs; KPOINTS for each.
4. perl script to transform CONTCAR output from a state relaxation to NEB endpoints.
5. perl script to linearly interpolate between NEB endpoints.*
6. Makefile to run NEB construction.

Note: the NEB interpolator script (nebmake.pl) is part of the VTST scripts.

`automator.`**`map2string`**`(`*tag*, *groupop*, *mapping*`)`

Takes in a map: tag, groupop, mapping and constructs a string representation to be dumped to a file. If we want to call using the tuple, `map2string(*(map))` will suffice.

Parameters

- **`tag`** – string of initial state to rotate
- **`groupop`** – see crystal.GroupOp; we use the rot and trans. This is in the supercell coord.
- **`mapping`** – in "chemorder" format; list by chemistry of lists of indices of position in initial cell to use.

Return string_rep string representation (to be used by an external script) `automator.supercelltar`(*tar, superdict, filemode=436, directmode=509, timestamp=None, INCARrelax='SYSTEM = {system}\nPREC = High\nISIF = 2\nEDIFF = 1E-8\nEDIFFG = -10E-3\nIBRION = 2\nNSW = 50\nISMEAR = 1\nSIGMA = 0.1\n# ENCUT =\n# NGX =\n# NGY =\n# NGZ =\n# NGXF =\n# NGYF =\n# NGZF =\n# NPAR =\nLWAVE = .FALSE.\nLCHARG = .FALSE.\nLREAL = .FALSE.\nVOSKOWN = 1\n', INCARNEB='SYSTEM = {system}\nPREC = High\nISIF = 2\nEDIFF = 1E-8\nEDIFFG = -10E-3\nIBRION = 2\nNSW = 50\nISMEAR = 1\nSIGMA = 0.1\n# ENCUT =\n# NGX =\n# NGY =\n# NGZ =\n# NGXF =\n# NGYF =\n# NGZF =\n# NPAR =\nLWAVE = .FALSE.\nLCHARG = .FALSE.\nLREAL = .FALSE.\nVOSKOWN = 1\nIMAGES = 1\nSPRING = -5\nLCLIMB = .TRUE.\nNELMIN = 4\nNFREE = 10\n', KPOINTS='Gamma\n1\nReciprocal\n0. 0. 0. 1.\n', basedir='', statename='relax.', transitionname='neb.', IDformat='{:02d}', JSONdict='tags.json', YAMLdef='supercell.yaml'*)

Takes in a tarfile (needs to be open for writing) and a supercelldict (from a diffuser) and creates the full directory structure inside the tarfile. Best used in a form like

```
with tarfile.open('supercells.tar.gz', mode='w:gz') as tar:
    automator.supercelltar(tar, supercelldict)
```

Parameters

- **tar** – tarfile open for writing; may contain other files in advance.
- **superdict** – dictionary of `states`, `transitions`, `transmapping`, `indices` that correspond to dictionaries with tags; the final tag `reference` is the basesupercell for calculations without defects.
  - superdict['states'][i] = supercell of state;
  - superdict['transitions'][n] = (supercell initial, supercell final);
  - superdict['transmapping'][n] = ((site tag, groupop, mapping), (site tag, groupop, mapping))
  - superdict['indices'][tag] = (type, index) of tag, where tag is either a state or transition tag; or...
  - superdict['indices'][tag] = index of tag, where tag is either a state or transition tag.
  - superdict['reference'] = (optional) supercell reference, no defects
- **filemode** – mode to use for files (default: 664)
- **directmode** – mode to use for directories (default: 775)
- **timestamp** – UNIX time for files; if None, use current time (default)
- **INCARrelax** – contents of INCAR file to use for relaxation; must contain {system} to be replaced by tag value (default: automator.INCARrelax)
- **INCARNEB** – contents of INCAR file to use for NEB; must contain {system} to be replaced by tag value (default: automator.INCARNEB)
- **KPOINTS** – contents of KPOINTS file (default: gamma-point only calculation); if None or empty, no KPOINTS file at all
- **basedir** – prepended to all files/directories (default: '')
- **statename** – prepended to all state names, before 2 digit number (default: relax.)

- **transitionname** – prepended to all transition names, before 2 digit number (default: neb.)
- **IDformat** – format for integer tags (default: {:02d})
- **JSONdict** – name of JSON file storing the tags corresponding to each directory (default: tags.json)
- **YAMLdef** – YAML file containing full definition of supercells, relationship, etc. (default: supercell.yaml); set to None to not output. **may want to change this to None for the future**

CHAPTER TWELVE

NOTEBOOKS

Example notebooks:

Below are several jupyter notebooks with example input and output from `onsager`.

12.1 Fe-C diffusion and elastodiffusivity

Taking data from R.G.A. Veiga, M. Perez, C. Becquart, E. Clouet and C. Domain, Acta mater. **59** (2011) p. 6963 doi:10.1016/j.actamat.2011.07.048

Fe in the body-centered cubic phase, a_0 = 0.28553 nm; C sit at octahedral sites, where the transition states between octahedral sites are represented by tetrahedral sites. The data is obtained from an EAM potential, where C_{11} = 243 GPa, C_{12} = 145 GPa, and C_{44} = 116 GPa. The tetrahedral transition state is 0.816 eV above the octahedral site, and the attempt frequency is taken as 10 THz (10^{13} Hz).

The dipole tensors can be separated into *parallel* and *perpendicular* components; the parallel direction points towards the closest Fe atoms for the C, while the perpendicular components lie in the interstitial plane. For the octahedral, the parallel component is 8.03 eV, and the perpendicular is 3.40 eV; for the tetrahedral transition state, the parallel component is 4.87 eV, and the perpendicular is 6.66 eV.

```
In [1]: import sys
        sys.path.extend(['../'])
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        %matplotlib inline
        import onsager.crystal as crystal
        import onsager.OnsagerCalc as onsager
        from scipy.constants import physical_constants
        kB = physical_constants['Boltzmann constant in eV/K'][0]
```

Create BCC lattice (lattice constant in nm).

```
In [2]: a0 = 0.28553
        Fe = crystal.Crystal.BCC(a0, "Fe")
        print(Fe)
```

```
#Lattice:
  a1 = [-0.142765  0.142765  0.142765]
  a2 = [ 0.142765 -0.142765  0.142765]
  a3 = [ 0.142765  0.142765 -0.142765]
#Basis:
  (Fe) 0.0 = [ 0.  0.  0.]
```

Elastic constants converted from GPa (10^9 J/m^3) to eV/(atomic volume).

```
In [3]: stressconv = 1e9*1e-27*Fe.volume/physical_constants['electron volt'][0]
        c11, c12, c44 = 243*stressconv, 145*stressconv, 116*stressconv
        s11, s12, s44 = (c11+c12)/((c11-c12)*(c11+2*c12)), -c12/((c11-c12)*(c11+2*c12)), 1/c44
        print('S11 = {:.4f} S12 = {:.4f} S44 = {:.4f}'.format(s11, s12, s44))
        stensor = np.zeros((3,3,3,3))
        for a in range(3):
            for b in range(3):
                for c in range(3):
                    for d in range(3):
                        if a==b and b==c and c==d: stensor[a,b,c,d] = s11
                        elif a==b and c==d: stensor[a,b,c,d] = s12
                        elif (a==d and b==c) or (a==c and b==d): stensor[a,b,c,d] = s44/4
```

```
S11 = 0.1023 S12 = -0.0382 S44 = 0.1187
```

Add carbon interstitial sites at octahedral sites in the lattice. This code (1) gets the set of symmetric Wyckoff positions corresponding to the single site $[00\frac{1}{2}]$ (first translated into unit cell coordinates), and then adds that new basis to our Fe crystal to generate a *new* crystal structure, that we name "FeC".

```
In [4]: uoct = np.dot(Fe.invlatt, np.array([0, 0, 0.5*a0]))
        FeC = Fe.addbasis(Fe.Wyckoffpos(uoct), ["C"])
        print(FeC)
```

```
#Lattice:
  a1 = [-0.142765  0.142765  0.142765]
  a2 = [ 0.142765 -0.142765  0.142765]
  a3 = [ 0.142765  0.142765 -0.142765]
#Basis:
  (Fe) 0.0 = [ 0.  0.  0.]
  (C) 1.0 = [ 0.5  0.   0.5]
  (C) 1.1 = [ 0.5  0.5  0. ]
  (C) 1.2 = [ 0.   0.5  0.5]
```

Next, we construct a *diffuser* based on our interstitial. We need to create a `sitelist` (which will be the Wyckoff positions) and a `jumpnetwork` for the transitions between the sites. There are tags that correspond to the unique states and transitions in the diffuser.

```
In [5]: chem = 1  # 1 is the index corresponding to our C atom in the crystal
        sitelist = FeC.sitelist(chem)
        jumpnetwork = FeC.jumpnetwork(chem, 0.6*a0)  # 0.6*a0 is the cutoff distance for finding jumps
        FeCdiffuser = onsager.Interstitial(FeC, chem, sitelist, jumpnetwork)
        print(FeCdiffuser)
```

```
Diffuser for atom 1 (C)
#Lattice:
  a1 = [-0.142765  0.142765  0.142765]
  a2 = [ 0.142765 -0.142765  0.142765]
  a3 = [ 0.142765  0.142765 -0.142765]
#Basis:
  (Fe) 0.0 = [ 0.  0.  0.]
  (C) 1.0 = [ 0.5  0.   0.5]
  (C) 1.1 = [ 0.5  0.5  0. ]
  (C) 1.2 = [ 0.   0.5  0.5]
states:
i:+0.500,+0.000,+0.500
transitions:
i:+0.500,+0.000,+0.500^i:+0.000,-0.500,+0.500
```

Next, we assemble our data: the energies, prefactors, and dipoles for the C atom in Fe, matched to the *representative* states: these are the first states in the lists, which are also identified by the tags above.

A note about units: If ν_0 is in THz, and a_0 is in nm, then $a_0^2\nu_0 = 10^{-2}$ cm^2/s. Thus, we multiply by Dconv = 10^{-2} so that our diffusivity is output in cm2/s.

In [6]:
```
Dconv = 1e-2
vu0 = 10*Dconv
Etrans = 0.816
dipoledict = {'Poctpara': 8.03, 'Poctperp': 3.40,
              'Ptetpara': 4.87, 'Ptetperp': 6.66}
FeCthermodict = {'pre': np.ones(len(sitelist)), 'ene': np.zeros(len(sitelist)),
                 'preT': vu0*np.ones(len(jumpnetwork)),
                 'eneT': Etrans*np.ones(len(jumpnetwork))}
# now to construct the site and transition dipole tensors; we use a "direction"--either
# the site position or the jump direction--to determine the parallel and perpendicular
# directions.
for dipname, Pname, direction in zip(('dipole', 'dipoleT'), ('Poct', 'Ptet'),
                                     (np.dot(FeC.lattice, FeC.basis[chem][sitelist[0][0]]),
                                      jumpnetwork[0][0][1])):
    # identify the non-zero index in our direction:
    paraindex = [n for n in range(3) if not np.isclose(direction[n], 0)][0]
    Ppara, Pperp = dipoledict[Pname + 'para'], dipoledict[Pname + 'perp']
    FeCthermodict[dipname] = np.diag([Ppara if i==paraindex else Pperp
                                      for i in range(3)])
for k,v in FeCthermodict.items():
    print('{}: {}'.format(k, v))
```

```
dipole: [[ 3.4   0.    0.  ]
 [ 0.    8.03  0.  ]
 [ 0.    0.    3.4 ]]
pre: [ 1.]
dipoleT: [[ 6.66  0.    0.  ]
 [ 0.    6.66  0.  ]
 [ 0.    0.    4.87]]
ene: [ 0.]
preT: [ 0.1]
eneT: [ 0.816]
```

We look at the diffusivity D, the elastodiffusivity d, and the activation volume tensor V over a range of temperatures from 300K to 1200K.

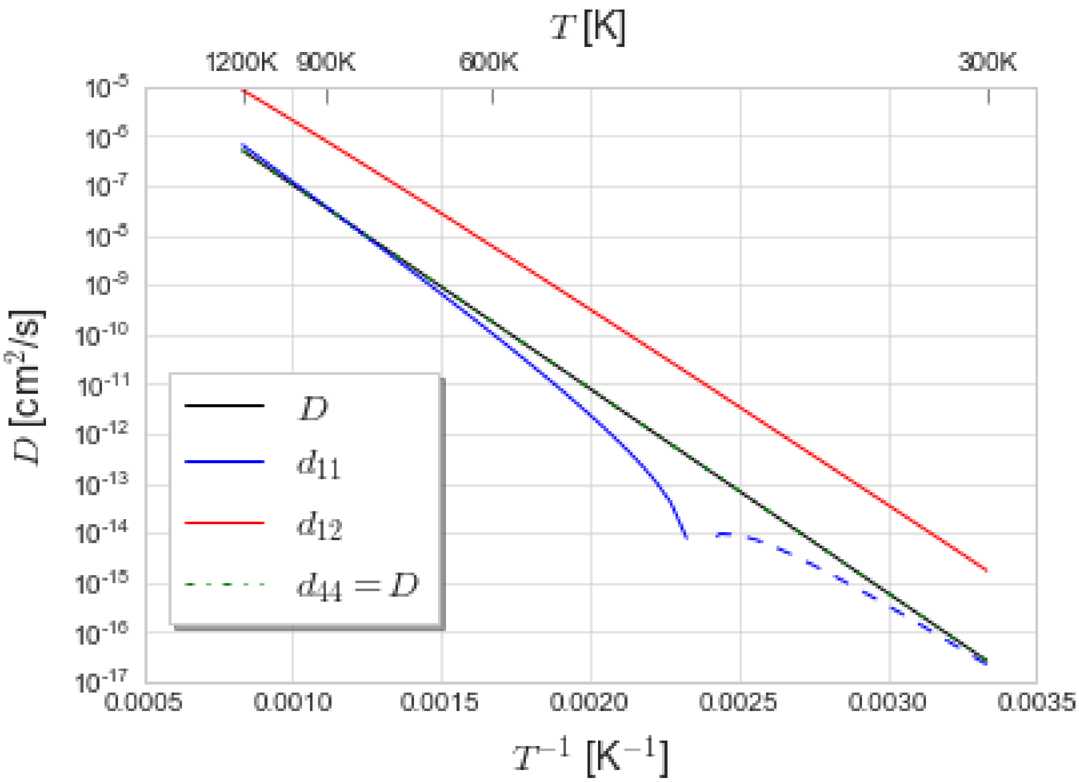
First, we calculate all of the pieces, including the diffusivity prefactor and activation barrier. As we only *have* one barrier, we compute the barrier at $k_B T = 1$.

In [7]:
```
Trange = np.linspace(300, 1200, 91)
Tlabels = Trange[0::30]
Dlist, dDlist, Vlist = [], [], []
for T in Trange:
    beta = 1./(kB*T)
    D, dD = FeCdiffuser.elastodiffusion(FeCthermodict['pre'],
                                        beta*FeCthermodict['ene'],
                                        [beta*FeCthermodict['dipole']],
                                        FeCthermodict['preT'],
                                        beta*FeCthermodict['eneT'],
                                        [beta*FeCthermodict['dipoleT']])
    Dlist.append(D[0,0])
    dDlist.append([dD[0,0,0,0], dD[0,0,1,1], dD[0,1,0,1]])
    Vtensor = (kB*T/(D[0,0]))*np.tensordot(dD, stensor, axes=((2,3),(0,1)))
    Vlist.append([np.trace(np.trace(Vtensor))/3,
                  Vtensor[0,0,0,0], Vtensor[0,0,1,1], Vtensor[0,1,0,1]])
D0 = FeCdiffuser.diffusivity(FeCthermodict['pre'],
                             np.zeros_like(FeCthermodict['ene']),
```

```
                                     FeCthermodict['preT'],
                                     np.zeros_like(FeCthermodict['eneT']))
        D, dbeta = FeCdiffuser.diffusivity(FeCthermodict['pre'],
                                           FeCthermodict['ene'],
                                           FeCthermodict['preT'],
                                           FeCthermodict['eneT'],
                                           CalcDeriv=True)
        print('D0: {:.4e} cm^2/s\nEact: {:.3f} eV'.format(D0[0,0], dbeta[0,0]/D[0,0]))
```

```
D0: 1.3588e-03 cm^2/s
Eact: 0.816 eV
```

```
In [8]: D, dD = np.array(Dlist), np.array(dDlist)
        d11_T = np.vstack((Trange, dD[:,0])).T
        d11pos = np.array([(T,d) for T,d in d11_T if d>=0])
        d11neg = np.array([(T,d) for T,d in d11_T if d<0])
        fig, ax1 = plt.subplots()
        ax1.plot(1./Trange, D, 'k', label='$D$')
        # ax1.plot(1./Trange, dD[:,0], 'b', label='$d_{11}$')
        ax1.plot(1./d11pos[:,0], d11pos[:,1], 'b', label='$d_{11}$')
        ax1.plot(1./d11neg[:,0], -d11neg[:,1], 'b--')
        ax1.plot(1./Trange, dD[:,1], 'r', label='$d_{12}$')
        ax1.plot(1./Trange, dD[:,2], 'g-.', label='$d_{44} = D$')
        ax1.set_yscale('log')
        ax1.set_ylabel('$D$ [cm$^2$/s]', fontsize='x-large')
        ax1.set_xlabel('$T^{-1}$ [K$^{-1}$]', fontsize='x-large')
        ax1.legend(bbox_to_anchor=(0.15,0.15,0.2,0.4), ncol=1,
                   shadow=True, frameon=True, fontsize='x-large')
        ax2 = ax1.twiny()
        ax2.set_xlim(ax1.get_xlim())
        ax2.set_xticks([1./t for t in Tlabels])
        ax2.set_xticklabels(["{:.0f}K".format(t) for t in Tlabels])
        ax2.set_xlabel('$T$ [K]', fontsize='x-large')
        ax2.grid(False)
        ax2.tick_params(axis='x', top='on', direction='in', length=6)
        plt.show()
        # plt.savefig('Fe-C-diffusivity.pdf', transparent=True, format='pdf')
```

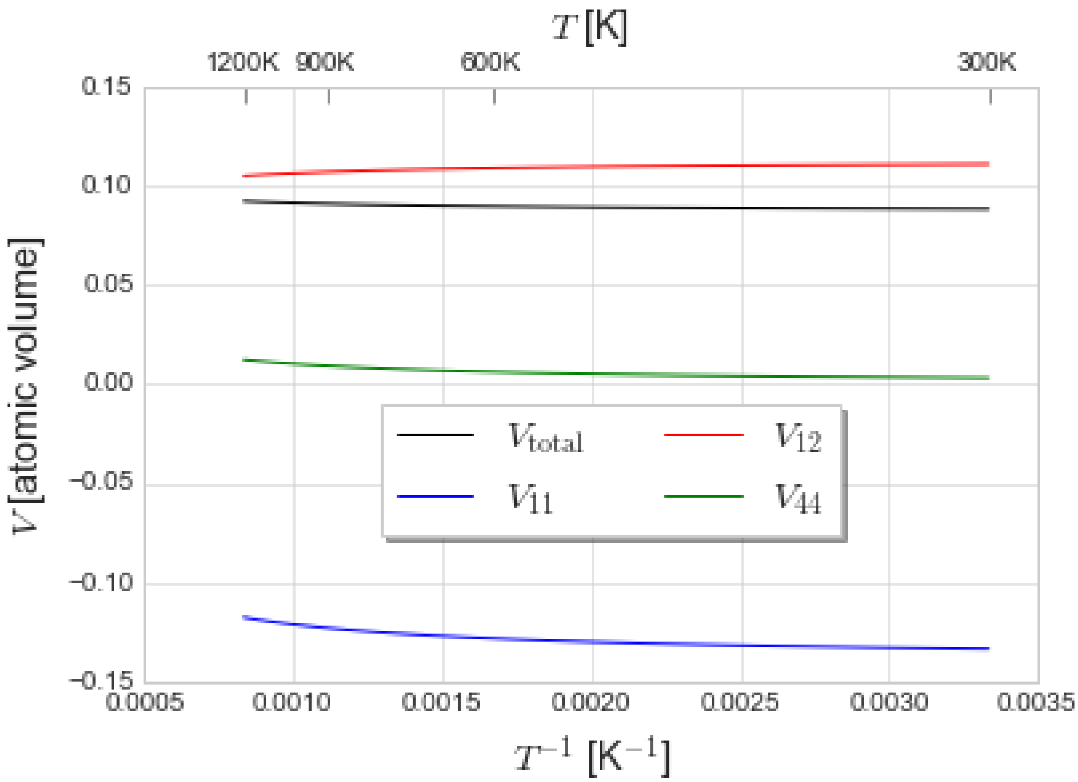



```
In [9]: d11pos[0,0], d11neg[-1,0]
```

```
Out[9]: (430.0, 420.0)
```

Activation volume. We plot the isotropic value (change in diffusivity with respect to pressure), but also the V_{xxxx}, V_{xxyy}, and V_{xyxy} terms. Interestingly, the V_{xxxx} term is negative—which indicates that diffusivity along the [100] direction *increases* with compressive stress in the [100] direction.

```
In [10]: V = np.array(Vlist)
         fig, ax1 = plt.subplots()
         ax1.plot(1./Trange, V[:,0], 'k', label='$V_{\\rm{total}}$')
         ax1.plot(1./Trange, V[:,1], 'b', label='$V_{11}$')
         ax1.plot(1./Trange, V[:,2], 'r', label='$V_{12}$')
         ax1.plot(1./Trange, 2*V[:,3], 'g', label='$V_{44}$')
         ax1.set_yscale('linear')
         ax1.set_ylabel('$V$ [atomic volume]', fontsize='x-large')
         ax1.set_xlabel('$T^{-1}$ [K$^{-1}$]', fontsize='x-large')
         ax1.legend(bbox_to_anchor=(0.3,0.3,0.5,0.2), ncol=2,
                    shadow=True, frameon=True, fontsize='x-large')
         ax2 = ax1.twiny()
         ax2.set_xlim(ax1.get_xlim())
         ax2.set_xticks([1./t for t in Tlabels])
         ax2.set_xticklabels(["{:.0f}K".format(t) for t in Tlabels])
         ax2.set_xlabel('$T$ [K]', fontsize='x-large')
         ax2.grid(False)
         ax2.tick_params(axis='x', top='on', direction='in', length=6)
         plt.show()
         # plt.savefig('Fe-C-activation-volume.pdf', transparent=True, format='pdf')
```



In [11]:
```
print('Total volume: {v[0]:.4f}, {V[0]:.4f}A^3\nV_xxxx: {v[1]:.4f}, {V[1]:.4f}A^3\nV_xxyy: {v[2]:.4f}, {V[
```

```
Total volume: 0.0921, 1.0722A^3
V_xxxx: -0.1175, -1.3681A^3
V_xxyy: 0.1048, 1.2202A^3
V_xyxy: 0.0061, 0.0714A^3
```

In [12]:
```
Vsph = 0.2*(3*V[-1,1] + 2*V[-1,2] + 4*V[-1,3])  # (3V11 + 2V12 + 2V44)/5
print('Spherical average uniaxial activation volume: {:.4f} {:.4f}A^3'.format(Vsph, Vsph*1e3*Fe.volume))
```

```
Spherical average uniaxial activation volume: -0.0237 -0.2757A^3
```

12.2 Convergence of Green function calculation

We check the convergence with $N_{\rm kpt}$ for the calculation of the vacancy Green function for FCC and HCP structures. In particular, we will look at:

1. The $\mathbf{R} = 0$ value,
2. The largest $\mathbf{R}$ value in the calculation of a first neighbor thermodynamic interaction range,
3. The difference of the Green function value for (1) and (2),

with increasing k-point density.

In [1]:
```
import sys
sys.path.extend(['../'])
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
%matplotlib inline
```

```
import onsager.crystal as crystal
import onsager.GFcalc as GFcalc
```

Create an FCC and HCP lattice.

```
In [2]: a0 = 1.
        FCC, HCP = crystal.Crystal.FCC(a0, "fcc"), crystal.Crystal.HCP(a0, chemistry="hcp")
        print(FCC)
        print(HCP)
```

```
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (fcc) 0.0 = [ 0.  0.  0.]
#Lattice:
  a1 = [ 0.5       -0.8660254  0.       ]
  a2 = [ 0.5        0.8660254  0.       ]
  a3 = [ 0.         0.         1.63299316]
#Basis:
  (hcp) 0.0 = [ 0.33333333  0.66666667  0.25      ]
  (hcp) 0.1 = [ 0.66666667  0.33333333  0.75      ]
```

We will put together our vectors for consideration:

- Maximum $\mathbf{R}$ for FCC = (400), or $\mathbf{x} = 2\hat{j} + 2\hat{k}$.
- Maximum $\mathbf{R}$ for HCP = (440), or $\mathbf{x} = 4\hat{i}$, and (222), or $\mathbf{x} = 2\hat{i} + 2\sqrt{8/3}\hat{k}$.

and our sitelists and jumpnetworks.

```
In [3]: FCCR = np.array([0,2.,2.])
        HCPR1, HCPR2 = np.array([4.,0.,0.]), np.array([2.,0.,2*np.sqrt(8/3)])
```

```
In [4]: FCCsite, FCCjn = FCC.sitelist(0), FCC.jumpnetwork(0, 0.75)
        HCPsite, HCPjn = HCP.sitelist(0), HCP.jumpnetwork(0, 1.01)
```

We use N_{max} parameter, which controls the automated generation of k-points to iterate through successively denser k-point meshes.

```
In [5]: FCCdata = {pmaxerror:[] for pmaxerror in range(-16,0)}
        print('kpt\tNkpt\tG(0)\tG(R)\tG diff')
        for Nmax in range(1,13):
            GFFCC = GFcalc.GFCrystalcalc(FCC, 0, FCCsite, FCCjn, Nmax=Nmax)
            Nreduce, Nkpt, kpt = GFFCC.Nkpt, np.prod(GFFCC.kptgrid), GFFCC.kptgrid
            for pmax in sorted(FCCdata.keys(), reverse=True):
                GFFCC.SetRates(np.ones(1), np.zeros(1), np.ones(1)/12, np.zeros(1), 10**(pmax))
                g0,gR = GFFCC(0,0,np.zeros(3)), GFFCC(0,0,FCCR)
                FCCdata[pmax].append((Nkpt, g0, gR))
            Nkpt,g0,gR = FCCdata[-8][-1]  # print the 10^-8 values
            print("{k[0]}x{k[1]}x{k[2]}\t".format(k=kpt) +
                  " {:5d} ({})\t{:.12f}\t{:.12f}\t{:.12f}".format(Nkpt, Nreduce,
                                                                   g0, gR,g0-gR))
```

```
kpt     Nkpt    G(0)    G(R)    G diff
6x6x6      216 (16)     -1.344901582401 -0.119888361621 -1.225013220779
10x10x10         1000 (48)     -1.344674624975 -0.084566077531 -1.260108547444
14x14x14         2744 (106)    -1.344663672542 -0.084541308263 -1.260122364278
18x18x18         5832 (200)    -1.344661890661 -0.084539383601 -1.260122507060
22x22x22        10648 (337)    -1.344661442418 -0.084538941204 -1.260122501213
26x26x26        17576 (528)    -1.344661295591 -0.084538798573 -1.260122497018
30x30x30        27000 (778)    -1.344661238153 -0.084538742761 -1.260122495392
```

```
34x34x34        39304 (1095)   -1.344661212587 -0.084538717850 -1.260122494737
38x38x38        54872 (1491)   -1.344661200055 -0.084538705591 -1.260122494464
42x42x42        74088 (1971)   -1.344661193423 -0.084538699082 -1.260122494341
46x46x46        97336 (2545)   -1.344661189691 -0.084538695410 -1.260122494281
50x50x50        125000 (3218)  -1.344661187483 -0.084538693232 -1.260122494251
```

In [6]:
```
HCPdata = []
print('kpt\tNkpt\tG(0)\tG(R1)\tG(R2)\tG(R1)-G(0)\tG(R2)-G0')
for Nmax in range(1,13):
    GFHCP = GFcalc.GFCrystalcalc(HCP, 0, HCPsite, HCPjn, Nmax=Nmax)
    GFHCP.SetRates(np.ones(1), np.zeros(1), np.ones(2)/12, np.zeros(2), 1e-8)
    g0,gR1,gR2 = GFHCP(0,0,np.zeros(3)), GFHCP(0,0,HCPR1), GFHCP(0,0,HCPR2)
    Nreduce, Nkpt, kpt = GFHCP.Nkpt, np.prod(GFHCP.kptgrid), GFHCP.kptgrid
    HCPdata.append((Nkpt, g0, gR1, gR2))
    print("{k[0]}x{k[1]}x{k[2]}\t".format(k=kpt) +
          "{:5d} ({})\t{:.12f}\t{:.12f}\t{:.12f}\t{:.12f}\t{:.12f}".format(Nkpt, Nreduce,
                                                                            g0, gR1, gR2,
                                                                            g0-gR1, g0-gR2))
```

```
kpt     Nkpt    G(0)    G(R1)   G(R2)   G(R1)-G(0)      G(R2)-G0
6x6x4     144 (21)      -1.367909503563 -0.192892722514 -0.131552967388 -1.175016781049 -1.236356536175
10x10x6   600 (56)      -1.345034474341 -0.087913619020 -0.089866654871 -1.257120855321 -1.255167819470
16x16x8  2048 (150)     -1.344668575390 -0.084546609595 -0.088212957806 -1.260121965795 -1.256455617584
20x20x12        4800 (308)     -1.344662392185 -0.084539941251 -0.088166498574 -1.260122450934 -1.256495893611
26x26x14        9464 (560)     -1.344661615456 -0.084539088966 -0.088165768509 -1.260122526490 -1.256495846946
30x30x16       14400 (819)     -1.344661401027 -0.084538892419 -0.088165529659 -1.260122508608 -1.256495871368
36x36x20       25920 (1397)    -1.344661260564 -0.084538764009 -0.088165374312 -1.260122496555 -1.256495886252
40x40x22       35200 (1848)    -1.344661230214 -0.084538734661 -0.088165342770 -1.260122495553 -1.256495887444
46x46x24       50784 (2600)    -1.344661210808 -0.084538715598 -0.088165322977 -1.260122495211 -1.256495887832
50x50x28       70000 (3510)    -1.344661197817 -0.084538703416 -0.088165309065 -1.260122494400 -1.256495888752
56x56x30       94080 (4640)    -1.344661192649 -0.084538698279 -0.088165303871 -1.260122494370 -1.256495888778
60x60x32      115200 (5627)    -1.344661189980 -0.084538695678 -0.088165301128 -1.260122494302 -1.256495888852
```

First, look at the behavior of the error with $p_{\max}$(error) parameter. The k-point integration error scales as $N_{\text{kpt}}^{5/3}$, and we see the $p_{\max}$ error is approximately 10^{-8}.

In [7]:
```
print('pmax\tGinf\talpha (Nkpt^-5/3 prefactor)')
Ginflist=[]
for pmax in sorted(FCCdata.keys(), reverse=True):
    data = FCCdata[pmax]
    Nk53 = np.array([N**(5/3) for (N,g0,gR) in data])
    gval = np.array([g0 for (N,g0,gR) in data])
    N10,N5 = np.average(Nk53*Nk53),np.average(Nk53)
    g10,g5 = np.average(gval*Nk53*Nk53),np.average(gval*Nk53)
    denom = N10-N5**2
    Ginf,alpha = (g10-g5*N5)/denom, (g10*N5-g5*N10)/denom
    Ginflist.append(Ginf)
    print('{}\t{}\t{}'.format(pmax, Ginf, alpha))
```

```
pmax    Ginf    alpha (Nkpt^-5/3 prefactor)
-1      -1.3622362852792858     203.75410596197204
-2      -1.345225052792947      24.479334937158068
-3      -1.3446883432274557     3.322356166765774
-4      -1.3446627820660566     1.186618137589885
-5      -1.344661289561045      1.0554418717631806
-6      -1.3446611908160067     1.1378852023509276
-7      -1.3446611836353533     1.2547347330078438
-8      -1.344661182870995      1.403893171115624
-9      -1.344661182337601      1.6081890775249583
-10     -1.3446611814380371     1.8951451743398244
-11     -1.3446611800056545     2.291136009713601
```

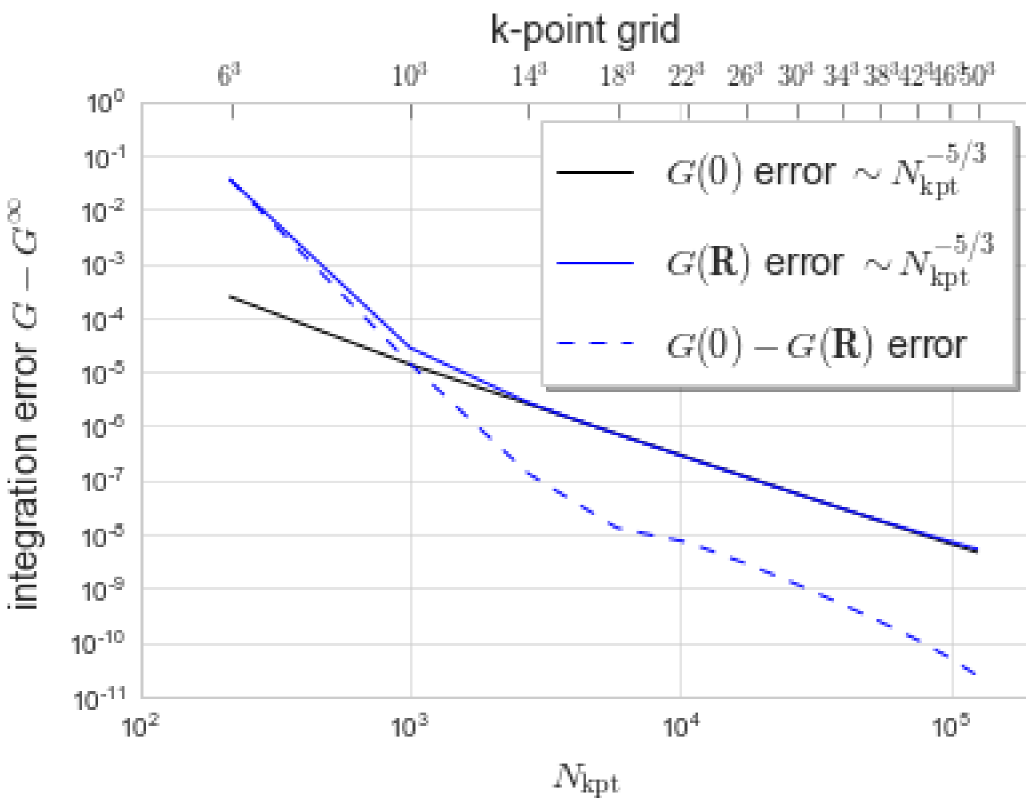
```
-12     -1.344661177927421      2.8185182806123508
-13     -1.3446611751184294     3.4945030859983652
-14     -1.3446611715189616     4.331229854988949
-15     -1.3446611670905142     5.336529960337354
-16     -1.3446611618106175     6.514974709092111
```

Plot the error in the Green function for FCC (at 0, maximum R, and difference between those GF). We extract the infinite value by fitting the error to $N_{\mathrm{kpt}}^{-5/3}$, which empirically matches the numerical error.

In [8]:
```
# plot the errors from pmax = 10^-8
data = FCCdata[-8]
Nk = np.array([N for (N,g0,gR) in data])
g0val = np.array([g0 for (N,g0,gR) in data])
gRval = np.array([gR for (N,g0,gR) in data])

gplot = []
Nk53 = np.array([N**(5/3) for (N,g0,gR) in data])
for gdata, start in zip((g0val, gRval, g0val-gRval), (0,1,2)):
    N10,N5 = np.average(Nk53[start:]*Nk53[start:]),np.average(Nk53[start:])
    denom = N10-N5**2
    g10 = np.average(gdata[start:]*Nk53[start:]*Nk53[start:])
    g5 = np.average(gdata[start:]*Nk53[start:])
    Ginf,alpha = (g10-g5*N5)/denom, (g10*N5-g5*N10)/denom
    gplot.append(np.abs(gdata-Ginf))

fig, ax1 = plt.subplots()
ax1.plot(Nk, gplot[0], 'k', label='$G(\mathbf{0})$ error $\sim N_{\mathrm{kpt}}^{-5/3}$')
ax1.plot(Nk, gplot[1], 'b', label='$G(\mathbf{R})$ error $\sim N_{\mathrm{kpt}}^{-5/3}$')
ax1.plot(Nk, gplot[2], 'b--', label='$G(\mathbf{0})-G(\mathbf{R})$ error')
ax1.set_xlim((1e2,2e5))
ax1.set_ylim((1e-11,1))
ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.set_xlabel('$N_{\mathrm{kpt}}$', fontsize='x-large')
ax1.set_ylabel('integration error $G-G^\infty$', fontsize='x-large')
ax1.legend(bbox_to_anchor=(0.6,0.6,0.4,0.4), ncol=1,
           shadow=True, frameon=True, fontsize='x-large')
ax2 = ax1.twiny()
ax2.set_xscale('log')
ax2.set_xlim(ax1.get_xlim())
ax2.set_xticks([n for n in Nk])
ax2.set_xticklabels(["${:.0f}^3$".format(n**(1/3)) for n in Nk])
ax2.set_xlabel('k-point grid', fontsize='x-large')
ax2.grid(False)
ax2.tick_params(axis='x', top='on', direction='in', length=6)
plt.show()
# plt.savefig('FCC-GFerror.pdf', transparent=True, format='pdf')
```



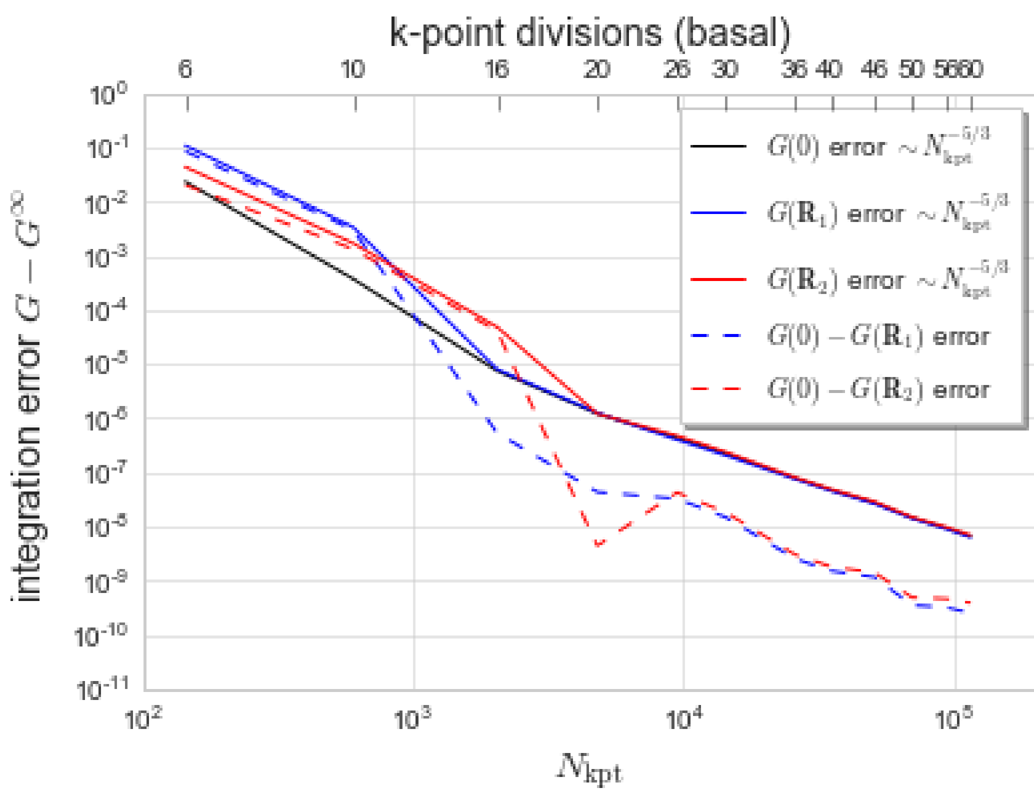
Plot the error in Green function for HCP.

In [9]:
```
# plot the errors from pmax = 10^-8
data = HCPdata
Nk = np.array([N for (N,g0,gR1,gR2) in data])
g0val = np.array([g0 for (N,g0,gR1,gR2) in data])
gR1val = np.array([gR1 for (N,g0,gR1,gR2) in data])
gR2val = np.array([gR2 for (N,g0,gR1,gR2) in data])

gplot = []
Nk53 = np.array([N**(5/3) for (N,g0,gR1,gR2) in data])
for gdata, start in zip((g0val, gR1val, gR2val, g0val-gR1val, g0val-gR2val), (3,3,3,3,3)):
    N10,N5 = np.average(Nk53[start:]*Nk53[start:]),np.average(Nk53[start:])
    denom = N10-N5**2
    g10 = np.average(gdata[start:]*Nk53[start:]*Nk53[start:])
    g5 = np.average(gdata[start:]*Nk53[start:])
    Ginf,alpha = (g10-g5*N5)/denom, (g10*N5-g5*N10)/denom
    gplot.append(np.abs(gdata-Ginf))

fig, ax1 = plt.subplots()
ax1.plot(Nk, gplot[0], 'k', label='$G(\mathbf{0})$ error $\sim N_{\mathrm{kpt}}^{-5/3}$')
ax1.plot(Nk, gplot[1], 'b', label='$G(\mathbf{R}_1)$ error $\sim N_{\mathrm{kpt}}^{-5/3}$')
ax1.plot(Nk, gplot[2], 'r', label='$G(\mathbf{R}_2)$ error $\sim N_{\mathrm{kpt}}^{-5/3}$')
ax1.plot(Nk, gplot[3], 'b--', label='$G(\mathbf{0})-G(\mathbf{R}_1)$ error')
ax1.plot(Nk, gplot[4], 'r--', label='$G(\mathbf{0})-G(\mathbf{R}_2)$ error')
ax1.set_xlim((1e2,2e5))
ax1.set_ylim((1e-11,1))
ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.set_xlabel('$N_{\mathrm{kpt}}$', fontsize='x-large')
```

```
ax1.set_ylabel('integration error $G-G^\infty$', fontsize='x-large')
ax1.legend(bbox_to_anchor=(0.6,0.6,0.4,0.4), ncol=1,
           shadow=True, frameon=True, fontsize='medium')
ax2 = ax1.twiny()
ax2.set_xscale('log')
ax2.set_xlim(ax1.get_xlim())
ax2.set_xticks([n for n in Nk])
# ax2.set_xticklabels(["${:.0f}$".format((n*1.875)**(1/3)) for n in Nk])
ax2.set_xticklabels(['6','10','16','20','26','30','36','40','46','50','56','60'])
ax2.set_xlabel('k-point divisions (basal)', fontsize='x-large')
ax2.grid(False)
ax2.tick_params(axis='x', top='on', direction='in', length=6)
plt.show()
# plt.savefig('HCP-GFerror.pdf', transparent=True, format='pdf')
```



12.3 Tracer correlation coefficients

We want (for testing purposes) to compute correlation coefficients for tracers for several different crystal structures:

- Simple cubic
- Body-centered cubic
- Face-centered cubic
- Diamond
- Wurtzite

- Hexagonal closed-packed
- NbO
- omega
- octahedral-tetrahedral network in HCP

Some are well-known (previously published) others are new.

```
In [1]: import sys
        sys.path.extend(['../'])
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        %matplotlib inline
        from onsager import crystal, OnsagerCalc
```

Create all of our lattices, with lattice constant a0.

```
In [2]: a0 = 1.
        SC = crystal.Crystal(a0*np.eye(3), [np.array([0.,0.,0.])], ["SC"])
        BCC = crystal.Crystal.BCC(a0, ["BCC"])
        FCC = crystal.Crystal.FCC(a0, ["FCC"])
        diamond = crystal.Crystal(a0*np.array([[0,1/2,1/2],[1/2,0,1/2],[1/2,1/2,0]]),
                                  [np.array([1/8,1/8,1/8]), np.array([-1/8,-1/8,-1/8])],
                                  ["diamond"])
        wurtzite = crystal.Crystal(a0*np.array([[1/2,1/2,0],
                                                [-np.sqrt(3/4),np.sqrt(3/4),0.],
                                                [0.,0.,np.sqrt(8/3)]]),
                                   [np.array([1/3,2/3,1/4-3/16]), np.array([1/3,2/3,1/4+3/16]),
                                    np.array([2/3,1/3,3/4-3/16]), np.array([2/3,1/3,3/4+3/16])],
                                   ["wurtzite"])
        HCP = crystal.Crystal.HCP(a0, np.sqrt(8/3), ["HCP"])
        NbO = crystal.Crystal(a0*np.eye(3),
                              [[np.array([0,1/2,1/2]), np.array([1/2,0,1/2]),np.array([1/2,1/2,0])],
                               [np.array([1/2,0,0]), np.array([0,1/2,0]), np.array([0,0,1/2])]],
                              ['Nb', 'O'])
        omega = crystal.Crystal(a0*np.array([[1/2,1/2,0],
                                             [-np.sqrt(3/4),np.sqrt(3/4),0.],
                                             [0.,0.,np.sqrt(3/8)]]),
                                   [np.array([0.,0.,0.]),
                                    np.array([1/3,2/3,1/2]), np.array([2/3,1/3,1/2])],
                                   ["omega"])
        octtet = crystal.Crystal(a0*np.array([[1/2,1/2,0],
                                              [-np.sqrt(3/4),np.sqrt(3/4),0.],
                                              [0.,0.,np.sqrt(8/3)]]),
                                   [[np.array([0.,0.,0.]), np.array([0.,0.,0.5]),
                                     np.array([1/3,2/3,5/8]), np.array([1/3,2/3,7/8]),
                                     np.array([2/3,1/3,3/8]), np.array([2/3,1/3,1/8])],
                                    [np.array([1/3,2/3,1/4]), np.array([2/3,1/3,3/4])]],
                                   ["O", "Ti"])
        crystallist = [SC, BCC, FCC, diamond, wurtzite, HCP, NbO, omega, octtet]
        crystalnames = ["simple cubic", "body-centered cubic", "face-centered cubic", "diamond",
                        "wurtzite", "hexagonal closed-packed", "NbO", "hexagonal omega",
                        "HCP octahedral-tetrahedral"]
```

```
In [3]: for name, crys in zip(crystalnames, crystallist):
            print(name)
            print(crys)
            print()
```

```
simple cubic
#Lattice:
  a1 = [ 1.  0.  0.]
  a2 = [ 0.  1.  0.]
  a3 = [ 0.  0.  1.]
#Basis:
  (SC) 0.0 = [ 0.  0.  0.]

body-centered cubic
#Lattice:
  a1 = [-0.5  0.5  0.5]
  a2 = [ 0.5 -0.5  0.5]
  a3 = [ 0.5  0.5 -0.5]
#Basis:
  (BCC) 0.0 = [ 0.  0.  0.]

face-centered cubic
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (FCC) 0.0 = [ 0.  0.  0.]

diamond
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (diamond) 0.0 = [ 0.625  0.625  0.625]
  (diamond) 0.1 = [ 0.375  0.375  0.375]

wurtzite
#Lattice:
  a1 = [ 0.5       -0.8660254  0.        ]
  a2 = [ 0.5        0.8660254  0.        ]
  a3 = [ 0.         0.         1.63299316]
#Basis:
  (wurtzite) 0.0 = [ 0.33333333  0.66666667  0.0625    ]
  (wurtzite) 0.1 = [ 0.33333333  0.66666667  0.4375    ]
  (wurtzite) 0.2 = [ 0.66666667  0.33333333  0.5625    ]
  (wurtzite) 0.3 = [ 0.66666667  0.33333333  0.9375    ]

hexagonal closed-packed
#Lattice:
  a1 = [ 0.5       -0.8660254  0.        ]
  a2 = [ 0.5        0.8660254  0.        ]
  a3 = [ 0.         0.         1.63299316]
#Basis:
  (HCP) 0.0 = [ 0.33333333  0.66666667  0.25      ]
  (HCP) 0.1 = [ 0.66666667  0.33333333  0.75      ]

NbO
#Lattice:
  a1 = [ 1.  0.  0.]
  a2 = [ 0.  1.  0.]
  a3 = [ 0.  0.  1.]
#Basis:
```

```
  (Nb) 0.0 = [ 0.   0.5  0.5]
  (Nb) 0.1 = [ 0.5  0.   0.5]
  (Nb) 0.2 = [ 0.5  0.5  0. ]
  (O) 1.0 = [ 0.5  0.   0. ]
  (O) 1.1 = [ 0.   0.5  0. ]
  (O) 1.2 = [ 0.   0.   0.5]

hexagonal omega
#Lattice:
  a1 = [ 0.          0.          0.61237244]
  a2 = [ 0.5        -0.8660254  0.        ]
  a3 = [ 0.5         0.8660254  0.        ]
#Basis:
  (omega) 0.0 = [ 0.  0.  0.]
  (omega) 0.1 = [ 0.5         0.33333333  0.66666667]
  (omega) 0.2 = [ 0.5         0.66666667  0.33333333]

HCP octahedral-tetrahedral
#Lattice:
  a1 = [ 0.5        -0.8660254  0.        ]
  a2 = [ 0.5         0.8660254  0.        ]
  a3 = [ 0.          0.          1.63299316]
#Basis:
  (O) 0.0 = [ 0.  0.  0.]
  (O) 0.1 = [ 0.   0.   0.5]
  (O) 0.2 = [ 0.33333333  0.66666667  0.625     ]
  (O) 0.3 = [ 0.33333333  0.66666667  0.875     ]
  (O) 0.4 = [ 0.66666667  0.33333333  0.375     ]
  (O) 0.5 = [ 0.66666667  0.33333333  0.125     ]
  (Ti) 1.0 = [ 0.33333333  0.66666667  0.25      ]
  (Ti) 1.1 = [ 0.66666667  0.33333333  0.75      ]
```

Now we generate diffusers for *every crystal*. This is fairly automated, where the main input is the cutoff distance.

In [4]:
```
cutoffs = [1.01*a0, 0.9*a0, 0.75*a0, 0.45*a0, 0.62*a0, 1.01*a0, 0.8*a0, 0.66*a0, 0.71*a0]
diffusers = []
for name, crys, cut in zip(crystalnames, crystallist, cutoffs):
    jn = crys.jumpnetwork(0, cut, 0.01)
    print(name)
    print(' Unique jumps:', len(jn))
    for jlist in jn:
        print('  connectivity:', len([i for (i,j), dx in jlist if i==jlist[0][0][0]]))
    diffusers.append(OnsagerCalc.VacancyMediated(crys, 0, crys.sitelist(0), jn, 1, 6))
```

```
simple cubic
 Unique jumps: 1
  connectivity: 6
body-centered cubic
 Unique jumps: 1
  connectivity: 8
face-centered cubic
 Unique jumps: 1
  connectivity: 12
diamond
 Unique jumps: 1
  connectivity: 4
wurtzite
 Unique jumps: 2
```

```
  connectivity: 1
  connectivity: 3
hexagonal closed-packed
 Unique jumps: 2
  connectivity: 6
  connectivity: 6
NbO
 Unique jumps: 1
  connectivity: 8
hexagonal omega
 Unique jumps: 4
  connectivity: 2
  connectivity: 12
  connectivity: 2
  connectivity: 3
HCP octahedral-tetrahedral
 Unique jumps: 3
  connectivity: 6
  connectivity: 1
  connectivity: 3
```

Now run through each, creating the "tracer" and compute the correlation coefficient. We do this by giving all of the vacancy positions the same energy (may not apply for true omega and octahedral-tetrahedral networks, for example), and then assigning the same energy for all transitions (again, may not apply for cases where there is more than one unique jump). We compute the full Onsager matrix, then look at the diagonal of $f = -L_{ss}/L_{sv}$.

```
In [5]: print('crystal\tf_xx\tf_zz')
        for name, diff in zip(crystalnames, diffusers):
            nsites, njumps = len(diff.sitelist), len(diff.om0_jn)
            tdict = {'preV': np.ones(nsites), 'eneV': np.zeros(nsites),
                     'preT0': np.ones(njumps), 'eneT0': np.zeros(njumps)}
            # make a tracer out of it:
            tdict.update(diff.maketracerpreene(**tdict))
            Lss, Lsv = diff.Lij(*diff.preene2betafree(1, **tdict))[1:3]  # just pull out ss and sv
            f = np.diag(-np.dot(Lss, np.linalg.inv(Lsv)))
            print('{name}\t{f[0]:.8f}\t{f[2]:.8f}'.format(name=name, f=f))
```

```
crystal f_xx    f_zz
simple cubic    0.65310884      0.65310884
body-centered cubic     0.72719414      0.72719414
face-centered cubic     0.78145142      0.78145142
diamond 0.50000000      0.50000000
wurtzite        0.50000000      0.50000000
hexagonal closed-packed 0.78120488      0.78145142
NbO     0.68891612      0.68891612
hexagonal omega 0.78122649      0.78157339
HCP octahedral-tetrahedral      0.63052307      0.65230273
```

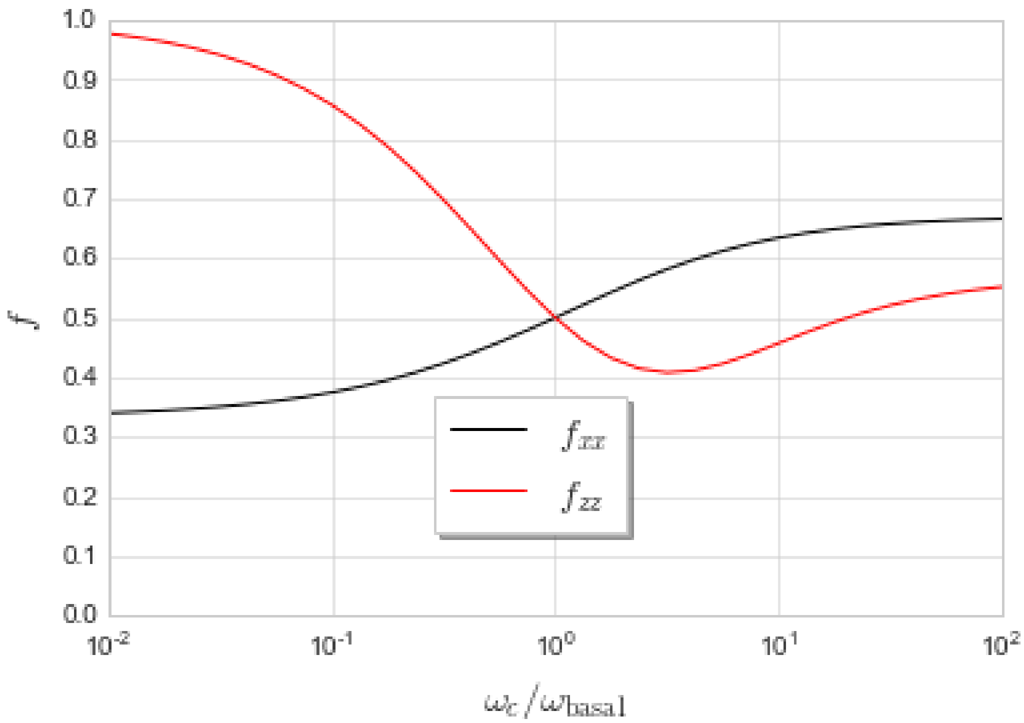
Look at variation in correlation coefficient for wurtzite structure by varying the ratio of the two rates.

```
In [6]: print('w(c)/w(basal)\tf_xx\tf_zz')
        crysindex = crystalnames.index('wurtzite')
        diff = diffusers[crysindex]
        nsites, njumps = len(diff.sitelist), len(diff.om0_jn)
        freq_list, correl_xx_list, correl_zz_list = [], [], []
        for i, w0_w1 in enumerate(np.linspace(-2,2,num=33)):
            w0 = 10**(w0_w1)
            w1 = 1
            tdict = {'preV': np.ones(nsites), 'eneV': np.zeros(nsites),
```

```
                'preT0': np.array([w0,w1]), 'eneT0': np.zeros(njumps)}
        # make a tracer out of it:
        tdict.update(diff.maketracerpreene(**tdict))
        Lss, Lsv = diff.Lij(*diff.preene2betafree(1, **tdict))[1:3]  # just pull out ss and sv
        f = np.diag(-np.dot(Lss, np.linalg.inv(Lsv)))
        freq_list.append(w0)
        correl_xx_list.append(f[0])
        correl_zz_list.append(f[2])
        if i%4==0:
            print('10^{w0_w1:+.2f}\t{f[0]:.8f}\t{f[2]:.8f}'.format(w0_w1=w0_w1, f=f))
```

```
w(c)/w(basal)   f_xx    f_zz
10^-2.00        0.34028415      0.97703525
10^-1.50        0.35071960      0.94102894
10^-1.00        0.37474153      0.85697089
10^-0.50        0.42323211      0.69772423
10^+0.00        0.50000000      0.50000000
10^+0.50        0.58129067      0.40813890
10^+1.00        0.63424155      0.45691154
10^+1.50        0.65763077      0.52060064
10^+2.00        0.66602090      0.55182811
```

```
In [7]: freq, correl_xx, correl_zz = np.array(freq_list), np.array(correl_xx_list), np.array(correl_zz_list)
        fig, ax1 = plt.subplots()
        ax1.plot(freq, correl_xx, 'k', label='$f_{xx}$')
        ax1.plot(freq, correl_zz, 'r', label='$f_{zz}$')
        ax1.set_xscale('log')
        ax1.set_ylabel('$f$', fontsize='x-large')
        ax1.set_xlabel('$\omega_{c}/\omega_{\mathrm{basal}}$', fontsize='x-large')
        ax1.set_ylim((0,1))
        ax1.set_yticks(np.linspace(0,1,11))
        ax1.legend(bbox_to_anchor=(0.4,0.1,0.2,0.3), ncol=1,
                   shadow=True, frameon=True, fontsize='x-large')
        plt.show()
        # plt.savefig('wurtzite-correlation.pdf', transparent=True, format='pdf')
```



12.4 Garnet correlation coefficients

Comparing to correlation coefficients from William D. Carlson and Clark R. Wilson, Phys Chem Minerals **43**, 363-369 (2016) doi:10.1007/s00269-016-0800-2

Garnet structure includes pyrope, which we use as our example structure, with space group 230 (Ia3d) with stoichiometry Mg3Al2Si3O12. The occupied Wyckoff positions for this are (lattice constant a_0=1.1459 nm):

Wykcoff site	chemistry	position
24c	Mg	1/8 0 1/4
16a	Al	0 0 0
24d	Si	3/8 0 1/4
96h	O	.03284 .05014 .65330

Data from G. V. Gibbs and J. V. Smith, "Refinement of the crystal structure of synthetic pyrope." American Mineralogist **50** 2023-2039 (1965), PDF.

```
In [1]: import sys
        sys.path.extend(['../'])
        import numpy as np
        import onsager.crystal as crystal
        import onsager.OnsagerCalc as onsager
```

Create garnet crystal (lattice constant in nm). Wyckoff positions cut and pasted from Bilbao crystallographic server.

```
In [2]: # a0 = 1.1459
        # alatt = a0*np.eye(3)
        a0 = 1.
        alatt = a0*np.array([[-0.5,0.5,0.5],[0.5,-0.5,0.5],[0.5,0.5,-0.5]])
        invlatt = np.array([[0,1,1],[1,0,1],[1,1,0]])
```

```
x,y,z = (.03284,.05014,.65330)
uMg = ((1/8,0,1/4),(3/8,0,3/4),(1/4,1/8,0),(3/4,3/8,0),
       (0,1/4,1/8),(0,3/4,3/8),(7/8,0,3/4),(5/8,0,1/4),
       (3/4,7/8,0),(1/4,5/8,0),(0,3/4,7/8),(0,1/4,5/8))
uAl = ((0,0,0),(1/2,0,1/2),(0,1/2,1/2),(1/2,1/2,0),
       (3/4,1/4,1/4),(3/4,3/4,3/4),(1/4,1/4,3/4),(1/4,3/4,1/4))
uSi = ((3/8,0,1/4),(1/8,0,3/4),(1/4,3/8,0),(3/4,1/8,0),
       (0,1/4,3/8),(0,3/4,1/8),(3/4,5/8,0),(3/4,3/8,1/2),
       (1/8,1/2,1/4),(7/8,0,1/4),(0,1/4,7/8),(1/2,1/4,1/8))
uO = ((x,y,z),(-x+1/2,-y,z+1/2),(-x,y+1/2,-z+1/2),(x+1/2,-y+1/2,-z),
      (z,x,y),(z+1/2,-x+1/2,-y),(-z+1/2,-x,y+1/2),(-z,x+1/2,-y+1/2),
      (y,z,x),(-y,z+1/2,-x+1/2),(y+1/2,-z+1/2,-x),(-y+1/2,-z,x+1/2),
      (y+3/4,x+1/4,-z+1/4),(-y+3/4,-x+3/4,-z+3/4),(y+1/4,-x+1/4,z+3/4),(-y+1/4,x+3/4,z+1/4),
      (x+3/4,z+1/4,-y+1/4),(-x+1/4,z+3/4,y+1/4),(-x+3/4,-z+3/4,-y+3/4),(x+1/4,-z+1/4,y+3/4),
      (z+3/4,y+1/4,-x+1/4),(z+1/4,-y+1/4,x+3/4),(-z+1/4,y+3/4,x+1/4),(-z+3/4,-y+3/4,-x+3/4),
      (-x,-y,-z),(x+1/2,y,-z+1/2),(x,-y+1/2,z+1/2),(-x+1/2,y+1/2,z),
      (-z,-x,-y),(-z+1/2,x+1/2,y),(z+1/2,x,-y+1/2),(z,-x+1/2,y+1/2),
      (-y,-z,-x),(y,-z+1/2,x+1/2),(-y+1/2,z+1/2,x),(y+1/2,z,-x+1/2),
      (-y+1/4,-x+3/4,z+3/4),(y+1/4,x+1/4,z+1/4),(-y+3/4,x+3/4,-z+1/4),(y+3/4,-x+1/4,-z+3/4),
      (-x+1/4,-z+3/4,y+3/4),(x+3/4,-z+1/4,-y+3/4),(x+1/4,z+1/4,y+1/4),(-x+3/4,z+3/4,-y+1/4),
      (-z+1/4,-y+3/4,x+3/4),(-z+3/4,y+3/4,-x+1/4),(z+3/4,-y+1/4,-x+3/4),(z+1/4,y+1/4,x+1/4))
# tovec = lambda x: np.array(x)
# tovec2 = lambda x: np.array((x[0]+1/2,x[1]+1/2,x[2]+1/2))
tovec = lambda x: np.dot(invlatt, x)
pyrope = crystal.Crystal(alatt, [[vec(w) for w in ulist for vec in (tovec,)]
                                 for ulist in (uMg, uAl, uSi, uO)],
                         ['Mg','Al','Si','O'])
# print(pyrope)
```

Next, we construct a *diffuser* based on vacancies for our Mg ion. We need to create a `sitelist` (which will be the Wyckoff positions) and a `jumpnetwork` for the transitions between the sites. There are tags that correspond to the unique states and transitions in the diffuser. The first cutoff is $\sim 0.31a_0$, but that connects half of the Mg cation sites to each other; increasing the cutoff to $\sim 0.51a_0$ introduces a second network that completes the connections.

In [3]:
```
chem = 0  # 0 is the index corresponding to our Mg atom in the crystal
cutoff = 0.31*a0  # had been 0.51*a0
sitelist = pyrope.sitelist(chem)
jumpnetwork = pyrope.jumpnetwork(chem, cutoff)
Mgdiffuser = onsager.VacancyMediated(pyrope, chem, sitelist, jumpnetwork, 1)
print(Mgdiffuser)
```

```
Diffuser for atom 0 (Mg), Nthermo=1
#Lattice:
  a1 = [-0.5  0.5  0.5]
  a2 = [ 0.5 -0.5  0.5]
  a3 = [ 0.5  0.5 -0.5]
#Basis:
  (Mg) 0.0 = [ 0.25   0.375  0.125]
  (Mg) 0.1 = [ 0.75   0.125  0.375]
  (Mg) 0.2 = [ 0.125  0.25   0.375]
  (Mg) 0.3 = [ 0.375  0.75   0.125]
  (Mg) 0.4 = [ 0.375  0.125  0.25 ]
  (Mg) 0.5 = [ 0.125  0.375  0.75 ]
  (Mg) 0.6 = [ 0.75   0.625  0.875]
  (Mg) 0.7 = [ 0.25   0.875  0.625]
  (Mg) 0.8 = [ 0.875  0.75   0.625]
  (Mg) 0.9 = [ 0.625  0.25   0.875]
  (Mg) 0.10 = [ 0.625  0.875  0.75 ]
  (Mg) 0.11 = [ 0.875  0.625  0.25 ]
```

```
(Al) 1.0 = [ 0.  0.  0.]
(Al) 1.1 = [ 0.5  0.   0.5]
(Al) 1.2 = [ 0.   0.5  0.5]
(Al) 1.3 = [ 0.5  0.5  0. ]
(Al) 1.4 = [ 0.5  0.   0. ]
(Al) 1.5 = [ 0.5  0.5  0.5]
(Al) 1.6 = [ 0.   0.   0.5]
(Al) 1.7 = [ 0.   0.5  0. ]
(Si) 2.0 = [ 0.25   0.625  0.375]
(Si) 2.1 = [ 0.75   0.875  0.125]
(Si) 2.2 = [ 0.375  0.25   0.625]
(Si) 2.3 = [ 0.125  0.75   0.875]
(Si) 2.4 = [ 0.625  0.375  0.25 ]
(Si) 2.5 = [ 0.875  0.125  0.75 ]
(Si) 2.6 = [ 0.625  0.75   0.375]
(Si) 2.7 = [ 0.875  0.25   0.125]
(Si) 2.8 = [ 0.75   0.375  0.625]
(Si) 2.9 = [ 0.25   0.125  0.875]
(Si) 2.10 = [ 0.125  0.875  0.25 ]
(Si) 2.11 = [ 0.375  0.625  0.75 ]
(O) 3.0 = [ 0.70344  0.68614  0.08298]
(O) 3.1 = [ 0.10316  0.62046  0.41702]
(O) 3.2 = [ 0.39684  0.81386  0.5173 ]
(O) 3.3 = [ 0.79656  0.87954  0.9827 ]
(O) 3.4 = [ 0.08298  0.70344  0.68614]
(O) 3.5 = [ 0.41702  0.10316  0.62046]
(O) 3.6 = [ 0.5173   0.39684  0.81386]
(O) 3.7 = [ 0.9827   0.79656  0.87954]
(O) 3.8 = [ 0.68614  0.08298  0.70344]
(O) 3.9 = [ 0.62046  0.41702  0.10316]
(O) 3.10 = [ 0.81386  0.5173   0.39684]
(O) 3.11 = [ 0.87954  0.9827   0.79656]
(O) 3.12 = [ 0.87954  0.39684  0.08298]
(O) 3.13 = [ 0.81386  0.79656  0.41702]
(O) 3.14 = [ 0.62046  0.70344  0.5173 ]
(O) 3.15 = [ 0.68614  0.10316  0.9827 ]
(O) 3.16 = [ 0.10316  0.9827   0.68614]
(O) 3.17 = [ 0.70344  0.5173   0.62046]
(O) 3.18 = [ 0.79656  0.41702  0.81386]
(O) 3.19 = [ 0.39684  0.08298  0.87954]
(O) 3.20 = [ 0.5173   0.62046  0.70344]
(O) 3.21 = [ 0.9827   0.68614  0.10316]
(O) 3.22 = [ 0.08298  0.87954  0.39684]
(O) 3.23 = [ 0.41702  0.81386  0.79656]
(O) 3.24 = [ 0.29656  0.31386  0.91702]
(O) 3.25 = [ 0.89684  0.37954  0.58298]
(O) 3.26 = [ 0.60316  0.18614  0.4827 ]
(O) 3.27 = [ 0.20344  0.12046  0.0173 ]
(O) 3.28 = [ 0.91702  0.29656  0.31386]
(O) 3.29 = [ 0.58298  0.89684  0.37954]
(O) 3.30 = [ 0.4827   0.60316  0.18614]
(O) 3.31 = [ 0.0173   0.20344  0.12046]
(O) 3.32 = [ 0.31386  0.91702  0.29656]
(O) 3.33 = [ 0.37954  0.58298  0.89684]
(O) 3.34 = [ 0.18614  0.4827   0.60316]
(O) 3.35 = [ 0.12046  0.0173   0.20344]
(O) 3.36 = [ 0.12046  0.60316  0.91702]
(O) 3.37 = [ 0.18614  0.20344  0.58298]
(O) 3.38 = [ 0.37954  0.29656  0.4827 ]
```

```
  (O) 3.39 = [ 0.31386  0.89684  0.0173 ]
  (O) 3.40 = [ 0.89684  0.0173   0.31386]
  (O) 3.41 = [ 0.29656  0.4827   0.37954]
  (O) 3.42 = [ 0.20344  0.58298  0.18614]
  (O) 3.43 = [ 0.60316  0.91702  0.12046]
  (O) 3.44 = [ 0.4827   0.37954  0.29656]
  (O) 3.45 = [ 0.0173   0.31386  0.89684]
  (O) 3.46 = [ 0.91702  0.12046  0.60316]
  (O) 3.47 = [ 0.58298  0.18614  0.20344]
vacancy configurations:
v:+0.250,+0.375,+0.125
solute configurations:
s:+0.250,+0.375,+0.125
solute-vacancy configurations:
s:+0.375,+0.125,+0.250-v:+0.750,+0.125,+0.375
omega0 jumps:
omega0:v:+0.625,+0.250,+0.875^v:+0.250,-0.125,+0.625
omega1 jumps:
omega1:s:+0.875,+0.625,+0.250-v:+0.625,+0.250,-0.125^v:+0.250,-0.125,-0.375
omega1:s:+0.750,+0.625,+0.875-v:+0.625,+0.250,+0.875^v:+0.250,-0.125,+0.625
omega1:s:+0.625,+0.875,+0.750-v:+0.625,+1.250,+0.875^v:+0.250,+0.875,+0.625
omega2 jumps:
omega2:s:+0.250,+0.875,+0.625-v:+0.625,+1.250,+0.875^s:+0.625,+0.250,+0.875-v:+0.250,-0.125,+0.625
```

Quick analysis on our jump network:

1. What is the connectivity, Z?
2. What is the individual contribution to $\delta\mathbf{x}\otimes\delta\mathbf{x}$? And 1/3 Tr (which will be the symmetrized contribution)?
3. What is the squared magnitude δx^2?

In [4]:
```
for jlist in jumpnetwork:
    Z = 0
    dx2 = np.zeros((3,3))
    for (i,j), dx in jlist:
        if i==0:
            Z += 1
            dx2 += np.outer(dx,dx)
    print("coordination number:", Z)
    print(dx2)
    print("1/3 Tr dx dx:", dx2.trace()/3)
    print("dx^2:", np.dot(dx,dx))
```

```
coordination number: 4
[[ 0.0625   0.       0.     ]
 [ 0.       0.15625 -0.125  ]
 [ 0.      -0.125    0.15625]]
1/3 Tr dx dx: 0.125
dx^2: 0.09375
```

Next, we assemble our data: the energies and prefactors, for a VMg in pyrope for our *representative* states and transitions: these are the first states in the lists, which are also identified by the tags above. As we are computing a tracer, we make the choice to set $\nu_0 = 1/Z$ where $Z = 4$ is the coordination number.

In [5]:
```
nu0 = 0.25
Etrans = 0.
# we don't need to use the tags, since there's only one site and jump type, and
# we want to build a tracer.
Mgthermodict = {'preV': np.ones(len(sitelist)),
```

```
                        'eneV': np.zeros(len(sitelist)),
                        'preT0': nu0*np.ones(len(jumpnetwork)),
                        'eneT0': Etrans*np.ones(len(jumpnetwork))}
        Mgthermodict.update(Mgdiffuser.maketracerpreene(**Mgthermodict))
        for k,v in Mgthermodict.items():
            print('{}: {}'.format(k, v))
```

```
eneSV: [ 0.]
eneT0: [ 0.]
preS: [ 1.]
preT1: [ 0.25  0.25  0.25]
preSV: [ 1.]
preT0: [ 0.25]
eneV: [ 0.]
eneT2: [ 0.]
eneS: [ 0.]
preV: [ 1.]
eneT1: [ 0.  0.  0.]
preT2: [ 0.25]
```

We compute the Onsager matrices, and look at $-L_{\mathrm{ss}}/L_{\mathrm{sv}}$ to get our correlation coefficient.

Note: we can define f (for our tracer) as the ratio of L_{ss} to $Z(\delta x)^2 w_2 c_{\mathrm{v}} c_{\mathrm{s}}/6 = \frac{1}{16}\nu_0 a_0^2$ in this case, the same as what we get for L_{vv} and $-L_{\mathrm{sv}}$.

```
In [6]: Lvv, Lss, Lsv, L1vv = Mgdiffuser.Lij(*Mgdiffuser.preene2betafree(1, **Mgthermodict))
        print(Lvv)
        print(Lss)
        print(Lsv)
        print(L1vv)
        print("Correlation coefficient:", -Lss[0,0]/Lsv[0,0])
```

```
[[ 0.015625 -0.       -0.      ]
 [-0.        0.015625 -0.      ]
 [-0.       -0.        0.015625]]
[[ 0.00585895  0.          0.        ]
 [ 0.          0.00585895  0.        ]
 [ 0.          0.          0.00585895]]
[[-0.015625  0.        0.      ]
 [ 0.       -0.015625  0.      ]
 [ 0.        0.       -0.015625]]
[[ -1.17108470e-34   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00  -1.17108470e-34   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00  -1.17108470e-34]]
Correlation coefficient: 0.374972670783
```

Compare with tabulated GF data from Carlson and Wilson paper. They use the notation (l, m, n) for a $\delta\mathbf{x}$ vector that is $a_0(l\hat{x} + m\hat{y} + n\hat{z})/8$. We will need to find a corresponding site that lands at that displacement from our origin site.

Unfortunately, it looks like in two cases ((800), (444)) there are two distinct sites that are mapped in that displacement vector, which have different GF values; the CW reported values appear to be the averaged values. In two other cases, ((640), (420)) the reported values are half of what the computed values are here.

As Carlson and Wilson used a stochastic approach to compute their GF values, all of their other data has errors $\sim 10^{-4}$.

```
In [7]: # tabulated data from paper
        CarlsonWilsonGFdata = \
        {(0,0,0): 2.30796022, (2,1,1): 1.30807261, (3,3,2): 0.80669536,
         (4,2,0): 0.40469085, (4,4,4): 0.50242046, (5,3,2): 0.56195744,
```

```
 (6,1,1): 0.56071092, (6,4,0): 0.22460654, (6,5,3): 0.42028488,
 (6,5,5): 0.40137897, (7,2,1): 0.44437878, (8,0,0): 0.41938675}
```

In [8]:
```
print('CW index\tdx match\tGF (FT eval)\tGF(CW stoch.)\terror')
GF = Mgdiffuser.GFcalc  # get our GF calculator; should already have rates set
basis = pyrope.basis[chem]
x0 = np.dot(alatt, basis[0])
for vec,gCW in CarlsonWilsonGFdata.items():
    dx0 = np.array(vec,dtype=float)/8
    nmatch, Gave, Gmatch = 0, 0, {}
    for g in pyrope.G:
        dx = np.dot(g.cartrot, dx0)
        j = pyrope.cart2pos(x0+dx)[1]
        if j is not None and j[0]==chem and j[1]<6:
            G = GF(0, j[1], dx)
            Gmatch[tuple((8*dx).astype(int))] = G
            nmatch += 1
            Gave += G
    Gave /= nmatch
    for t,G in Gmatch.items():
        print('{}\t{}\t{:.12f}\t{:.8f}\t{:.4e}'.format(vec, t, -G, gCW, abs(G+gCW)))
    print('{}\taverage value\t{:.12f}\t{:.8f}\t{:.4e}'.format(vec, -Gave, gCW, abs(Gave+gCW)))
```

```
CW index        dx match        GF (FT eval)    GF(CW stoch.)   error
(8, 0, 0)       (0, 0, -8)      0.427361034009  0.41938675      7.9743e-03
(8, 0, 0)       (8, 0, 0)       0.403566247455  0.41938675      1.5821e-02
(8, 0, 0)       (0, 8, 0)       0.427361034009  0.41938675      7.9743e-03
(8, 0, 0)       (-8, 0, 0)      0.403566247455  0.41938675      1.5821e-02
(8, 0, 0)       (0, -8, 0)      0.427361034009  0.41938675      7.9743e-03
(8, 0, 0)       (0, 0, 8)       0.427361034009  0.41938675      7.9743e-03
(8, 0, 0)       average value   0.419429438491  0.41938675      4.2688e-05
(6, 1, 1)       (-1, 6, 1)      0.560766700022  0.56071092      5.5780e-05
(6, 1, 1)       (-1, -6, -1)    0.560766700022  0.56071092      5.5780e-05
(6, 1, 1)       (1, 1, 6)       0.560766700022  0.56071092      5.5780e-05
(6, 1, 1)       (1, -1, -6)     0.560766700022  0.56071092      5.5780e-05
(6, 1, 1)       average value   0.560766700022  0.56071092      5.5780e-05
(3, 3, 2)       (-3, 3, -2)     0.806767995595  0.80669536      7.2636e-05
(3, 3, 2)       (3, -2, 3)      0.806767995595  0.80669536      7.2636e-05
(3, 3, 2)       (3, 2, -3)      0.806767995595  0.80669536      7.2636e-05
(3, 3, 2)       (-3, -3, 2)     0.806767995595  0.80669536      7.2636e-05
(3, 3, 2)       average value   0.806767995595  0.80669536      7.2636e-05
(2, 1, 1)       (-1, -2, 1)     1.308081132926  1.30807261      8.5229e-06
(2, 1, 1)       (-1, 2, -1)     1.308081132926  1.30807261      8.5229e-06
(2, 1, 1)       (1, 1, -2)      1.308081132926  1.30807261      8.5229e-06
(2, 1, 1)       (1, -1, 2)      1.308081132926  1.30807261      8.5229e-06
(2, 1, 1)       average value   1.308081132926  1.30807261      8.5229e-06
(6, 5, 3)       (3, -6, 5)      0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (-3, -5, 6)     0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (5, -3, -6)     0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (-3, 5, -6)     0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (-5, -6, -3)    0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (-5, 6, 3)      0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (5, 3, 6)       0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       (3, 6, -5)      0.420386782427  0.42028488      1.0190e-04
(6, 5, 3)       average value   0.420386782427  0.42028488      1.0190e-04
(6, 4, 0)       (-6, 0, 4)      0.449091350780  0.22460654      2.2448e-01
(6, 4, 0)       (6, 4, 0)       0.449091350780  0.22460654      2.2448e-01
(6, 4, 0)       (6, -4, 0)      0.449091350780  0.22460654      2.2448e-01
(6, 4, 0)       (-6, 0, -4)     0.449091350780  0.22460654      2.2448e-01
(6, 4, 0)       average value   0.449091350780  0.22460654      2.2448e-01
```

```
(0, 0, 0)       (0, 0, 0)       2.308081141615  2.30796022      1.2092e-04
(0, 0, 0)       average value   2.308081141615  2.30796022      1.2092e-04
(4, 2, 0)       (2, 0, -4)      0.809394258097  0.40469085      4.0470e-01
(4, 2, 0)       (-2, -4, 0)     0.809394258097  0.40469085      4.0470e-01
(4, 2, 0)       (-2, 4, 0)      0.809394258097  0.40469085      4.0470e-01
(4, 2, 0)       (2, 0, 4)       0.809394258097  0.40469085      4.0470e-01
(4, 2, 0)       average value   0.809394258097  0.40469085      4.0470e-01
(5, 3, 2)       (-5, 2, -3)     0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (3, 2, 5)       0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (3, -2, -5)     0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (-5, -2, 3)     0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (5, 3, -2)      0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (5, -3, 2)      0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (-3, -5, -2)    0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       (-3, 5, 2)      0.561961239416  0.56195744      3.7994e-06
(5, 3, 2)       average value   0.561961239416  0.56195744      3.7994e-06
(7, 2, 1)       (-1, -2, -7)    0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (1, 7, 2)       0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (7, 2, -1)      0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (-7, 1, -2)     0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (-7, -1, 2)     0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (7, -2, 1)      0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (-1, 2, 7)      0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       (1, -7, -2)     0.444350262895  0.44437878      2.8517e-05
(7, 2, 1)       average value   0.444350262895  0.44437878      2.8517e-05
(4, 4, 4)       (-4, 4, 4)      0.457297218361  0.50242046      4.5123e-02
(4, 4, 4)       (-4, 4, -4)     0.547635344309  0.50242046      4.5215e-02
(4, 4, 4)       (-4, -4, -4)    0.457297218361  0.50242046      4.5123e-02
(4, 4, 4)       (4, 4, -4)      0.547635344309  0.50242046      4.5215e-02
(4, 4, 4)       (-4, -4, 4)     0.547635344309  0.50242046      4.5215e-02
(4, 4, 4)       (4, -4, -4)     0.457297218361  0.50242046      4.5123e-02
(4, 4, 4)       (4, -4, 4)      0.547635344309  0.50242046      4.5215e-02
(4, 4, 4)       (4, 4, 4)       0.457297218361  0.50242046      4.5123e-02
(4, 4, 4)       average value   0.502466281335  0.50242046      4.5821e-05
(6, 5, 5)       (-5, 6, -5)     0.401425331863  0.40137897      4.6362e-05
(6, 5, 5)       (-5, -6, 5)     0.401425331863  0.40137897      4.6362e-05
(6, 5, 5)       (5, -5, 6)      0.401425331863  0.40137897      4.6362e-05
(6, 5, 5)       (5, 5, -6)      0.401425331863  0.40137897      4.6362e-05
(6, 5, 5)       average value   0.401425331863  0.40137897      4.6362e-05
```

12.5 Large ω^2 correction

In the limit of large ω^2, large roundoff error can become problematic as the correlation almost exactly matches the uncorrelated contribution to solute diffusion, and so it becomes necessary to introduce an alternative treatment specific to the large ω^2 limit. We will show the range of roundoff error by using FCC as an example.

In [1]:
```
import sys
sys.path.extend(['../'])
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
%matplotlib inline
from onsager import crystal, OnsagerCalc
```

Create FCC crystal, and diffuser with first neighbor range.

```
In [2]: a0 = 1.
        FCC = crystal.Crystal.FCC(a0, ["FCC"])
        diffuser = OnsagerCalc.VacancyMediated(FCC, 0, FCC.sitelist(0),
                                               FCC.jumpnetwork(0, 0.75*a0), 1)
        print(diffuser)
```

```
Diffuser for atom 0 (FCC), Nthermo=1
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (FCC) 0.0 = [ 0.  0.  0.]
vacancy configurations:
v:+0.000,+0.000,+0.000
solute configurations:
s:+0.000,+0.000,+0.000
solute-vacancy configurations:
s:+0.000,+0.000,+0.000-v:-1.000,+0.000,+0.000
omega0 jumps:
omega0:v:+0.000,+0.000,+0.000^v:+1.000,+0.000,-1.000
omega1 jumps:
omega1:s:+0.000,+0.000,+0.000-v:+1.000,+0.000,-1.000^v:+2.000,+0.000,-2.000
omega1:s:+0.000,+0.000,+0.000-v:-1.000,+1.000,+0.000^v:+0.000,+1.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+1.000,-1.000,+0.000^v:+2.000,-1.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+0.000,-1.000,+0.000^v:+1.000,-1.000,-1.000
omega2 jumps:
omega2:s:+0.000,+0.000,+0.000-v:-1.000,+0.000,+1.000^s:+0.000,+0.000,+0.000-v:+1.000,+0.000,-1.000
```

Next, we fill out our thermodynamic dictionary.

```
In [3]: tdict = {'preV': np.ones(1), 'eneV': np.zeros(1), 'preT0': np.ones(1), 'eneT0': np.zeros(1)}
        tdict.update(diffuser.maketracerpreene(**tdict))
        for k,v in tdict.items():
            print(k, v)
```

```
preV [ 1.]
preT1 [ 1.  1.  1.  1.]
eneT2 [ 0.]
eneSV [ 0.]
preSV [ 1.]
eneV [ 0.]
eneT1 [ 0.  0.  0.  0.]
eneS [ 0.]
preT2 [ 1.]
preT0 [ 1.]
preS [ 1.]
eneT0 [ 0.]
```

Now, to loop through a range of ω^2 values from 10^{-17} to 10^{17}, and evaluate the L_{ss} in three different ways:

1. Never using the large ω^2 treatment (should fail for large ω^2).
2. Always using the large ω^2 treatment (should fail for small ω^2).
3. Automatically switching treatment depending on ω^2 value (should be accurate over entire range).

Because the failure can be pretty spectacular, we check for NaN, Inf, or 0 values.

```
In [4]: print('omega2\tno large\tall large\tautomatic')
        om2_list, correl_list = [], []
        for om2pow in np.concatenate((np.linspace(-17,-13,num=17), np.linspace(13,17,num=17))):
```

```
        om2 = 10.**(om2pow)
        tdict['preT2'] = np.array([om2])
        correl = []
        for large_om2 in (1e33, 1e-33, 1e8):
            Lss, Lsv = diffuser.Lij(*diffuser.preene2betafree(1., **tdict),
                                    large_om2=large_om2)[1:3]
            if Lsv[0,0] is np.nan or Lsv[0,0] is np.inf or Lsv[0,0]==0 :
                c = 1
            else:
                c = -Lss[0,0]/Lsv[0,0]
            correl.append(c)
        om2_list.append(om2)
        correl_list.append(correl)
        print('10^{:+.2f}\t{:.8e}\t{:.8e}\t{:.16e}'.format(om2pow,
                                                    correl[0], correl[1], correl[2]))
```

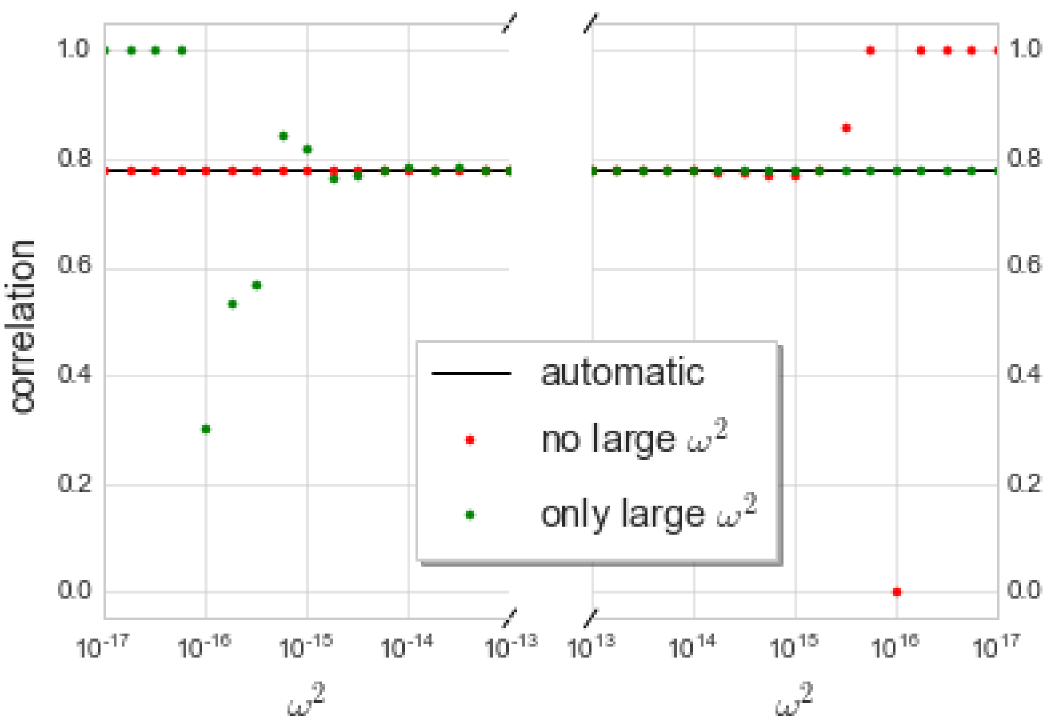
```
omega2  no large        all large       automatic
10^-17.00       7.81451419e-01  1.00000000e+00  7.8145141885543312e-01
10^-16.75       7.81451419e-01  1.00000000e+00  7.8145141885543312e-01
10^-16.50       7.81451419e-01  1.00000000e+00  7.8145141885543301e-01
10^-16.25       7.81451419e-01  1.00000000e+00  7.8145141885543301e-01
10^-16.00       7.81451419e-01  3.00239975e-01  7.8145141885543301e-01
10^-15.75       7.81451419e-01  5.33910566e-01  7.8145141885543312e-01
10^-15.50       7.81451419e-01  5.69665300e-01  7.8145141885543301e-01
10^-15.25       7.81451419e-01  8.44186728e-01  7.8145141885543312e-01
10^-15.00       7.81451419e-01  8.18836296e-01  7.8145141885543312e-01
10^-14.75       7.81451419e-01  7.62729380e-01  7.8145141885543312e-01
10^-14.50       7.81451419e-01  7.69817973e-01  7.8145141885543312e-01
10^-14.25       7.81451419e-01  7.79249287e-01  7.8145141885543301e-01
10^-14.00       7.81451419e-01  7.83234718e-01  7.8145141885543301e-01
10^-13.75       7.81451419e-01  7.81332535e-01  7.8145141885543312e-01
10^-13.50       7.81451419e-01  7.86830524e-01  7.8145141885543312e-01
10^-13.25       7.81451419e-01  7.81654377e-01  7.8145141885543312e-01
10^-13.00       7.81451419e-01  7.81874935e-01  7.8145141885543301e-01
10^+13.00       7.81383433e-01  7.81451419e-01  7.8145141885543312e-01
10^+13.25       7.81196581e-01  7.81451419e-01  7.8145141885543323e-01
10^+13.50       7.80341880e-01  7.81451419e-01  7.8145141885543312e-01
10^+13.75       7.81196581e-01  7.81451419e-01  7.8145141885543323e-01
10^+14.00       7.80068729e-01  7.81451419e-01  7.8145141885543312e-01
10^+14.25       7.76223776e-01  7.81451419e-01  7.8145141885543323e-01
10^+14.50       7.74647887e-01  7.81451419e-01  7.8145141885543323e-01
10^+14.75       7.71428571e-01  7.81451419e-01  7.8145141885543323e-01
10^+15.00       7.71428571e-01  7.81451419e-01  7.8145141885543301e-01
10^+15.25       7.77777778e-01  7.81451419e-01  7.8145141885543312e-01
10^+15.50       8.57142857e-01  7.81451419e-01  7.8145141885543312e-01
10^+15.75       1.00000000e+00  7.81451419e-01  7.8145141885543323e-01
10^+16.00       0.00000000e+00  7.81451419e-01  7.8145141885543323e-01
10^+16.25       1.00000000e+00  7.81451419e-01  7.8145141885543301e-01
10^+16.50       1.00000000e+00  7.81451419e-01  7.8145141885543301e-01
10^+16.75       1.00000000e+00  7.81451419e-01  7.8145141885543323e-01
10^+17.00       1.00000000e+00  7.81451419e-01  7.8145141885543323e-01
```

```
In [5]: om2, correl = np.array(om2_list), np.array(correl_list)
        f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
        for ax in (ax1, ax2):
            ax.plot(om2, correl[:,2], 'k', label='automatic')
            ax.plot(om2, correl[:,0], 'r.', label='no large $\omega^2$')
            ax.plot(om2, correl[:,1], 'g.', label='only large $\omega^2$')
        ax1.set_xlim((1e-17,1e-13))
        ax2.set_xlim((1e13,1e17))
```

```
ax1.set_ylim((-0.05,1.05))
ax2.set_ylim((-0.05,1.05))
ax1.set_xscale('log')
ax2.set_xscale('log')
ax1.set_xlabel('$\omega^2$', fontsize='x-large')
ax2.set_xlabel('$\omega^2$', fontsize='x-large')
ax1.set_ylabel('correlation', fontsize='x-large')
ax2.legend(bbox_to_anchor=(0,0.3,0.5,0.2), ncol=1,
           shadow=True, frameon=True, fontsize='x-large')
ax1.yaxis.tick_left()
ax1.tick_params(labelright='off')
ax2.yaxis.tick_right()
ax1.spines['right'].set_visible(False)
ax2.spines['left'].set_visible(False)

d = .015 # how big to make the diagonal lines in axes coordinates
# arguments to pass plot, just so we don't keep repeating them
kwargs = dict(transform=ax1.transAxes, color='k', clip_on=False)
ax1.plot((1-d,1+d), (-d,+d), **kwargs)
ax1.plot((1-d,1+d),(1-d,1+d), **kwargs)

kwargs.update(transform=ax2.transAxes)  # switch to the bottom axes
ax2.plot((-d,+d), (1-d,1+d), **kwargs)
ax2.plot((-d,+d), (-d,+d), **kwargs)
plt.show()
# plt.savefig('largeomega2.pdf', transparent=True, format='pdf')
```



12.6 Si in FCC Ni

Based on data in hdl.handle.net/11115/239, "Data Citation: Diffusion of Si impurities in Ni under stress: A first-principles study" by T. Garnier, V. R. Manga, P. Bellon, and D. R. Trinkle (2014). The transport coefficient results, using the self-consistent mean-field method, appear in T. Garnier, V. R. Manga, D. R. Trinkle, M. Nastar, and P. Bellon, "Stress-induced anisotropic diffusion in alloys: Complex Si solute flow near a dislocation core in Ni," *Phys. Rev. B* **88**, 134108 (2013), doi:10.1103/PhysRevB.88.134108.

```
In [1]: import sys
        sys.path.append('../')
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        %matplotlib inline
        import onsager.crystal as crystal
        import onsager.OnsagerCalc as onsager
        from scipy.constants import physical_constants
        kB = physical_constants['Boltzmann constant in eV/K'][0]
        import h5py, json
```

Create an FCC Ni crystal.

```
In [2]: a0 = 0.343
        Ni = crystal.Crystal.FCC(a0, chemistry="Ni")
        print(Ni)
```

```
#Lattice:
  a1 = [ 0.       0.1715  0.1715]
  a2 = [ 0.1715  0.       0.1715]
  a3 = [ 0.1715  0.1715  0.    ]
#Basis:
  (Ni) 0.0 = [ 0.  0.  0.]
```

Next, we construct our diffuser. For this problem, our thermodynamic range is out to the fourth neighbor; hence, we construct a *two shell* thermodynamic range (that is, sums of two $\frac{a}{2}\langle 110\rangle$ vectors. That is, $N_{\text{thermo}} = 2$ gives 4 stars: $\frac{a}{2}\langle 110\rangle$, $a\langle 100\rangle$, $\frac{a}{2}\langle 112\rangle$, and $a\langle 110\rangle$. For Si in Ni, the first three have non-zero interaction energies, while the fourth is zero. The states, as written, are the solute (basis index + lattice position) : vacancy (basis index + lattice position), and *dx* is the (Cartesian) vector separating them.

```
In [3]: chemistry = 0 # only one sublattice anyway
        Nthermo = 2
        NiSi = onsager.VacancyMediated(Ni, chemistry, Ni.sitelist(chemistry),
                                       Ni.jumpnetwork(chemistry, 0.75*a0), Nthermo)
        print(NiSi)
```

```
Diffuser for atom 0 (Ni), Nthermo=2
#Lattice:
  a1 = [ 0.       0.1715  0.1715]
  a2 = [ 0.1715  0.       0.1715]
  a3 = [ 0.1715  0.1715  0.    ]
#Basis:
  (Ni) 0.0 = [ 0.  0.  0.]
vacancy configurations:
v:+0.000,+0.000,+0.000
solute configurations:
s:+0.000,+0.000,+0.000
solute-vacancy configurations:
s:+0.000,+0.000,+0.000-v:+1.000,-1.000,+0.000
s:+0.000,+0.000,+0.000-v:-1.000,-1.000,+1.000
s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-2.000
```

```
s:+0.000,+0.000,+0.000-v:-2.000,+0.000,+0.000
omega0 jumps:
omega0:v:+0.000,+0.000,+0.000^v:+0.000,+0.000,-1.000
omega1 jumps:
omega1:s:+0.000,+0.000,+0.000-v:-1.000,+0.000,+0.000^v:-1.000,+0.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+1.000,+0.000,+0.000^v:+1.000,+0.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:-1.000,+1.000,+0.000^v:-1.000,+1.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+0.000,+0.000,-1.000^v:+0.000,+0.000,-2.000
omega1:s:+0.000,+0.000,+0.000-v:+1.000,-1.000,-1.000^v:+1.000,-1.000,-2.000
omega1:s:+0.000,+0.000,+0.000-v:-1.000,-1.000,+1.000^v:-1.000,-1.000,+0.000
omega1:s:+0.000,+0.000,+0.000-v:-1.000,+0.000,-1.000^v:-1.000,+0.000,-2.000
omega1:s:+0.000,+0.000,+0.000-v:-2.000,+1.000,+0.000^v:-2.000,+1.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+1.000,+1.000,-2.000^v:+1.000,+1.000,-3.000
omega1:s:+0.000,+0.000,+0.000-v:+2.000,+0.000,-1.000^v:+2.000,+0.000,-2.000
omega1:s:+0.000,+0.000,+0.000-v:-2.000,+1.000,+1.000^v:-2.000,+1.000,+0.000
omega1:s:+0.000,+0.000,+0.000-v:-2.000,+0.000,+0.000^v:-2.000,+0.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+2.000,-2.000,+0.000^v:+2.000,-2.000,-1.000
omega1:s:+0.000,+0.000,+0.000-v:+0.000,+0.000,-2.000^v:+0.000,+0.000,-3.000
omega2 jumps:
omega2:s:+0.000,+0.000,+0.000-v:+0.000,+0.000,+1.000^s:+0.000,+0.000,+0.000-v:+0.000,+0.000,-1.000
```

Below is an example of the above data translated into a dictionary corresponding to the data for Ni-Si; it is output into a `JSON` compliant file for reference. The strings are the corresponding tags in the diffuser. The first entry in each list is the prefactor (in THz) and the second is the corresponding energy (in eV). **Note:** all jumps are defined as *transition state energies*, hence the reference energy is added / subtracted as needed. Also, there are "missing" transition states; these will have there energies defined using the LIMB (linear interpolation of migration barriers) approximation. This introduces an error of no more than 10 meV in any activation barrier.

```
In [4]: NiSidata={
        "v:+0.000,+0.000,+0.000": [1., 0.],
        "s:+0.000,+0.000,+0.000": [1., 0.],
        "s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000": [1., -0.108],
        "s:+0.000,+0.000,+0.000-v:-1.000,-1.000,+1.000": [1., +0.004],
        "s:+0.000,+0.000,+0.000-v:+1.000,-2.000,+0.000": [1., +0.037],
        "s:+0.000,+0.000,+0.000-v:+0.000,-2.000,+0.000": [1., -0.008],
        "omega0:v:+0.000,+0.000,+0.000^v:+0.000,+1.000,-1.000": [4.8, 1.074],
        "omega1:s:+0.000,+0.000,+0.000-v:-1.000,+0.000,+0.000^v:-1.000,+1.000,-1.000": [5.2, 1.213-0.108],
        "omega1:s:+0.000,+0.000,+0.000-v:+0.000,-1.000,+0.000^v:+0.000,+0.000,-1.000": [5.2, 1.003-0.108],
        "omega1:s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000^v:+0.000,+2.000,-2.000": [4.8, 1.128-0.108],
        "omega1:s:+0.000,+0.000,+0.000-v:-1.000,+1.000,+0.000^v:-1.000,+2.000,-1.000": [5.2, 1.153-0.108],
        "omega1:s:+0.000,+0.000,+0.000-v:+1.000,-1.000,-1.000^v:+1.000,+0.000,-2.000": [4.8, 1.091+0.004],
        "omega2:s:+0.000,+0.000,+0.000-v:+0.000,-1.000,+1.000^s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000": [5.1,
        }
        NiSi2013data={
        "v:+0.000,+0.000,+0.000": [1., 0.],
        "s:+0.000,+0.000,+0.000": [1., 0.],
        "s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000": [1., -0.100],
        "s:+0.000,+0.000,+0.000-v:-1.000,-1.000,+1.000": [1., +0.011],
        "s:+0.000,+0.000,+0.000-v:+1.000,-2.000,+0.000": [1., +0.045],
        "s:+0.000,+0.000,+0.000-v:+0.000,-2.000,+0.000": [1., 0],
        "omega0:v:+0.000,+0.000,+0.000^v:+0.000,+1.000,-1.000": [4.8, 1.074],
        "omega1:s:+0.000,+0.000,+0.000-v:-1.000,+0.000,+0.000^v:-1.000,+1.000,-1.000": [5.2, 1.213-0.100],
        "omega1:s:+0.000,+0.000,+0.000-v:+0.000,-1.000,+0.000^v:+0.000,+0.000,-1.000": [5.2, 1.003-0.100],
        "omega1:s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000^v:+0.000,+2.000,-2.000": [4.8, 1.128-0.100],
        "omega1:s:+0.000,+0.000,+0.000-v:-1.000,+1.000,+0.000^v:-1.000,+2.000,-1.000": [5.2, 1.153-0.100],
        "omega1:s:+0.000,+0.000,+0.000-v:+1.000,-1.000,-1.000^v:+1.000,+0.000,-2.000": [4.8, 1.091+0.011],
        "omega2:s:+0.000,+0.000,+0.000-v:+0.000,-1.000,+1.000^s:+0.000,+0.000,+0.000-v:+0.000,+1.000,-1.000": [5.1,
```

```
    }
    print(json.dumps(NiSi2013data, sort_keys=True, indent=4))
```

"omega0:v:+0.000,+0.000,+0.000^v:+0.000,+1.000,-1.000": [4.8,   1.074   ],   "omega1:s:+0.000,+0

Next, we convert our dictionary into the simpler form used by the diffuser.

```
In [5]: preenedict = NiSi.tags2preene(NiSi2013data)
        preenedict
```

```
Out[5]: {'eneS': array([ 0.]),
         'eneSV': array([-0.1  ,  0.011,  0.045,  0.   ]),
         'eneT0': array([ 1.074]),
         'eneT1': array([ 1.053 ,  0.903 ,  1.113 ,  1.028 ,  1.0795,  1.102 ,  1.0965,
                 1.0965,  1.0965,  1.0965,  1.119 ,  1.074 ,  1.074 ,  1.074 ]),
         'eneT2': array([ 0.791]),
         'eneV': array([ 0.]),
         'preS': array([ 1.]),
         'preSV': array([ 1.,  1.,  1.,  1.]),
         'preT0': array([ 4.8]),
         'preT1': array([ 5.2,  5.2,  5.2,  4.8,  4.8,  4.8,  4.8,  4.8,  4.8,  4.8,  4.8,
                 4.8,  4.8,  4.8]),
         'preT2': array([ 5.1]),
         'preV': array([ 1.])}
```

We can now calculate the diffusion coefficients and drag ratio. **Note:** the diffusion coefficients L_{ss} and L_{sv} both need to be multiplied by $c_s c_v / k_B T$ where c_s is the solute concentration, c_v the (equilibrium) vacancy concentration, and $k_B T$ is the thermal energy of the system. The current units shown below are in $\mathrm{nm}^2 \cdot \mathrm{THz}$.

```
In [6]: print("#T #Lss #Lsv #drag")
        for T in np.linspace(300, 1400, 23):
            L0vv, Lss, Lsv, L1vv = NiSi.Lij(*NiSi.preene2betafree(kB*T, **preenedict))
            print(T, Lss[0,0], Lsv[0,0], Lsv[0,0]/Lss[0,0])
```

```
#T #Lss #Lsv #drag
300.0 4.1020388689e-16 4.03521345382e-16 0.983709219436
350.0 5.99654580387e-14 5.75933473853e-14 0.960442048956
400.0 2.51914470587e-12 2.32702939103e-12 0.923737880405
450.0 4.61249420384e-11 4.03116417036e-11 0.873966230027
500.0 4.7250491963e-10 3.84170227063e-10 0.813050216206
550.0 3.17382569388e-09 2.36031280433e-09 0.743680665541
600.0 1.55354740581e-08 1.03884191885e-08 0.668690195716
650.0 5.96097883468e-08 3.52092669306e-08 0.590662505388
700.0 1.88868586771e-07 9.66525181976e-08 0.511744805477
750.0 5.13340570374e-07 2.22584238868e-07 0.43359954719
800.0 1.23159242387e-06 4.40215392835e-07 0.357435937654
850.0 2.66586838346e-06 7.57314596438e-07 0.28407801418
900.0 5.29556637614e-06 1.13347630469e-06 0.214042507294
950.0 9.78435004446e-06 1.44428949825e-06 0.147612206399
1000.0 1.69973273335e-05 1.44304978386e-06 0.084898628799
1050.0 2.80063788635e-05 7.25153043356e-07 0.0258924242542
1100.0 4.4083410105e-05 -1.3003604007e-06 -0.0294977270951
1150.0 6.66826933232e-05 -5.4290391372e-06 -0.0814160146604
1200.0 9.74143994545e-05 -1.26676005168e-05 -0.130038275529
1250.0 0.000138011882666 -2.42289344792e-05 -0.175556872431
1300.0 0.000190295346612 -4.15167679854e-05 -0.21817016929
1350.0 0.000256134304169 -6.61019634203e-05 -0.258075401632
1400.0 0.000337410855954 -9.96927651132e-05 -0.295464011765
```

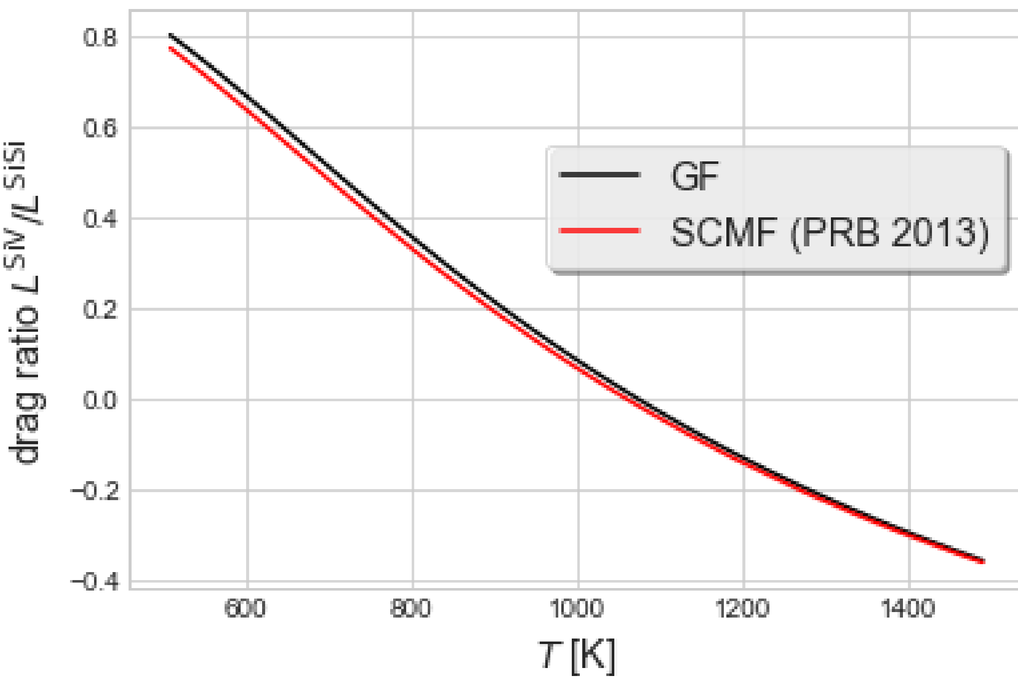
For direct comparison with the SCMF data in the 2013 *Phys. Rev. B* paper, we evaluate at 960K, 1060K (the predicted crossover temperature), and 1160K. The reported data is in units of mol/eV Å ns.

```
In [7]: volume = 0.25*a0**3
        conv = 1e3*0.1/volume  # 10^3 for THz->ns^-1, 10^-1 for nm^-1 ->Ang^-1
        # T: (L0vv, Lsv, Lss)
        PRBdata = {960: (1.52e-1, 1.57e-1, 1.29e0),
                   1060: (4.69e-1, 0., 3.27e0),
                   1160: (1.18e0, -7.55e-1, 7.02e0)}
        print("#T #Lvv #Lsv #Lss")
        for T in (960, 1060, 1160):
            c = conv/(kB*T)
            L0vv, Lss, Lsv, L1vv = NiSi.Lij(*NiSi.preene2betafree(kB*T, **preenedict))
            vv, sv, ss = L0vv[0,0]*c, Lsv[0,0]*c, Lss[0,0]*c
            vvref, svref, ssref = PRBdata[T]
            print("{} {:.4g} ({:.4g}) {:.4g} ({:.4g}) {:.4g} ({:.4g})".format(T, vv, vvref/vv, sv, svref/sv, ss, ss
```

```
#T #Lvv #Lsv #Lss
960 0.1556 (0.9766) 0.1773 (0.8856) 1.315 (0.9807)
1060 0.4797 (0.9777) 0.04852 (0) 3.339 (0.9792)
1160 1.208 (0.9769) -0.6537 (1.155) 7.152 (0.9815)
```

```
In [8]: # raw comparison data from 2013 paper
        Tval = np.array([510, 530, 550, 570, 590, 610, 630, 650, 670, 690,
                         710, 730, 750, 770, 790, 810, 830, 850, 870, 890,
                         910, 930, 950, 970, 990, 1010, 1030, 1050, 1070, 1090,
                         1110, 1130, 1150, 1170, 1190, 1210, 1230, 1250, 1270, 1290,
                         1310, 1330, 1350, 1370, 1390, 1410, 1430, 1450, 1470, 1490])
        fluxval = np.array([0.771344, 0.743072, 0.713923, 0.684066, 0.653661, 0.622858,
                            0.591787, 0.560983, 0.529615, 0.498822, 0.467298, 0.436502,
                            0.406013, 0.376193, 0.346530, 0.316744, 0.288483, 0.260656,
                            0.232809, 0.205861, 0.179139, 0.154038, 0.128150, 0.103273,
                            0.079025, 0.055587, 0.032558, 0.010136, -0.011727, -0.033069,
                            -0.053826, -0.074061, -0.093802, -0.113075, -0.132267, -0.149595,
                            -0.167389, -0.184604, -0.202465, -0.218904, -0.234157, -0.250360,
                            -0.265637, -0.280173, -0.294940, -0.308410, -0.322271, -0.335809,
                            -0.349106, -0.361605])
        # Trange = np.linspace(300, 1500, 121)
        Draglist = []
        for T in Tval:
            L0vv, Lss, Lsv, L1vv = NiSi.Lij(*NiSi.preene2betafree(kB*T, **preenedict))
            Draglist.append(Lsv[0,0]/Lss[0,0])
        Drag = np.array(Draglist)
```

```
In [9]: fig, ax1 = plt.subplots()
        ax1.plot(Tval, Drag, 'k', label='GF')
        ax1.plot(Tval, fluxval, 'r', label='SCMF (PRB 2013)')
        ax1.set_ylabel('drag ratio $L^{\\rm{SiV}}/L^{\\rm{SiSi}}$', fontsize='x-large')
        ax1.set_xlabel('$T$ [K]', fontsize='x-large')
        ax1.legend(bbox_to_anchor=(0.5,0.6,0.5,0.2), ncol=1,
                   shadow=True, frameon=True, fontsize='x-large')
        plt.show()
        # plt.savefig('NiSi-drag.pdf', transparent=True, format='pdf')
```



12.7 Split oxygen-vacancy defects in Co

We want to work out the symmetry analysis for our split oxygen-vacancy (V-O-V) defects α-Co (HCP) and β-Co (FCC).

The split defects can be represented simply as crowdion interstitial sites, for the purposes of symmetry analysis. We're interested in extracting the tensor expansions around those sites, and (eventually) computing the damping coefficients from the DFT data.

```
In [1]: import sys
        sys.path.extend(['../'])
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('seaborn-whitegrid')
        %matplotlib inline
        import onsager.crystal as crystal
        import onsager.OnsagerCalc as onsager
        from scipy.constants import physical_constants
        kB = physical_constants['Boltzmann constant in eV/K'][0]
```

```
In [2]: betaCo = crystal.Crystal.FCC(1.0, 'Co')
        print(betaCo)
```

```
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (Co) 0.0 = [ 0.  0.  0.]
```

```
In [3]: betaCo.Wyckoffpos(np.array([0.5,0.,0.]))
```

```
Out[3]: [array([ 0. ,  0. ,  0.5]),
         array([ 0.5,  0. ,  0.5]),
```

```
 array([ 0.5,  0. ,  0. ]),
 array([ 0. ,  0.5,  0. ]),
 array([ 0.5,  0.5,  0. ]),
 array([ 0. ,  0.5,  0.5])]
```

```
In [4]: betaCoO = betaCo.addbasis(betaCo.Wyckoffpos(np.array([0.5,0.,0.])), ['O'])
        print(betaCoO)
```

```
#Lattice:
  a1 = [ 0.   0.5  0.5]
  a2 = [ 0.5  0.   0.5]
  a3 = [ 0.5  0.5  0. ]
#Basis:
  (Co) 0.0 = [ 0.  0.  0.]
  (O) 1.0 = [ 0.   0.   0.5]
  (O) 1.1 = [ 0.5  0.   0.5]
  (O) 1.2 = [ 0.5  0.   0. ]
  (O) 1.3 = [ 0.   0.5  0. ]
  (O) 1.4 = [ 0.5  0.5  0. ]
  (O) 1.5 = [ 0.   0.5  0.5]
```

```
In [5]: Ojumpnetwork = betaCoO.jumpnetwork(1,0.5)
```

```
In [6]: Odiffuser = onsager.Interstitial(betaCoO, 1, betaCoO.sitelist(1), Ojumpnetwork)
```

We need to analyze the geometry of our representative site; we get the position, then find the zero entry in the position vector, and work from there.

```
In [7]: Ppara, Pperp, Pshear = -2.70, -4.30, 0.13
        reppos = betaCoO.pos2cart(np.zeros(3), (1, Odiffuser.sitelist[0][0]))
        perpindex = [n for n in range(3) if np.isclose(reppos[n], 0)][0]
        paraindex = [n for n in range(3) if n != perpindex]
        shearsign = 1 if reppos[paraindex[0]]*reppos[paraindex[1]] > 0 else -1
        Pdipole = np.diag([Pperp if n == perpindex else Ppara for n in range(3)])
        Pdipole[paraindex[0], paraindex[1]] = shearsign*Pshear
        Pdipole[paraindex[1], paraindex[0]] = shearsign*Pshear
        Pdipole
```

```
Out[7]: array([[-2.7 ,  0.13,  0.  ],
               [ 0.13, -2.7 ,  0.  ],
               [ 0.  ,  0.  , -4.3 ]])
```

```
In [8]: nu0, Emig = 1e13, 0.91
        nsites, njumps = len(Odiffuser.sitelist), len(Odiffuser.jumpnetwork)
        betaCoOthermodict = {'pre': np.ones(nsites), 'ene': np.zeros(nsites),
                             'preT': nu0*np.ones(nsites), 'eneT': Emig*np.ones(nsites)}
        beta = 1./(kB*300)  # 300K
        Llamb = Odiffuser.losstensors(betaCoOthermodict['pre'], beta*betaCoOthermodict['ene'],
                                      [Pdipole],
                                      betaCoOthermodict['preT'], beta*betaCoOthermodict['eneT'])
```

```
In [9]: for (lamb, Ltens) in Llamb:
            print(lamb, crystal.FourthRankIsotropic(Ltens))
```

```
0.0619225494951 (0.0, 0.17066666666666686)
0.0412816996634 (2.4132664014743868e-32, 0.0033799999999999629)
```

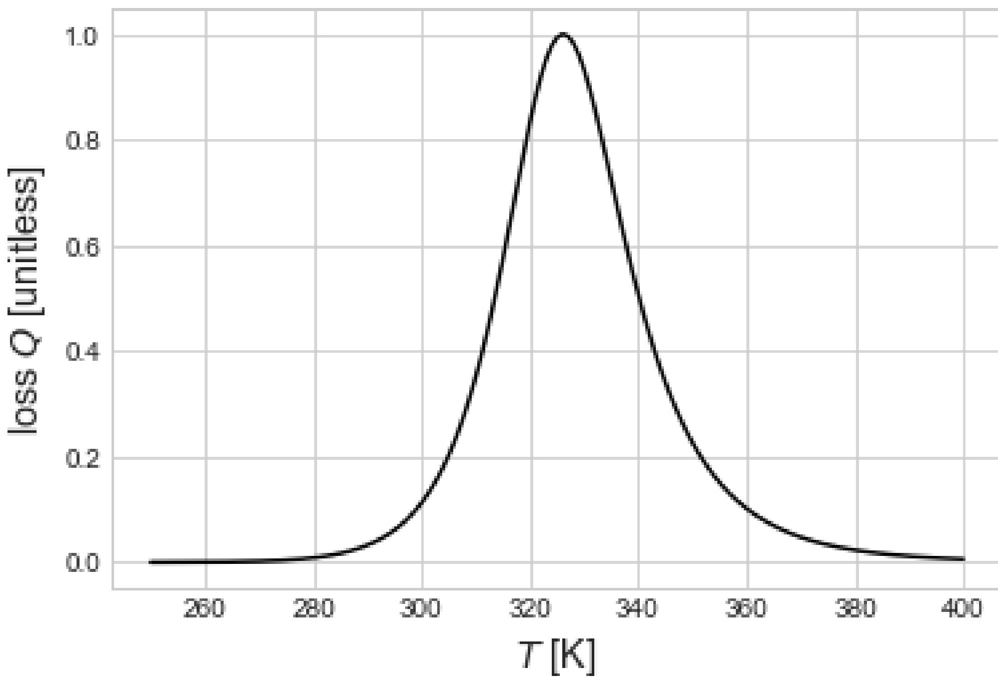
```
In [10]: sh1 = crystal.FourthRankIsotropic(Llamb[0][1])[1]
         sh2 = crystal.FourthRankIsotropic(Llamb[1][1])[1]
         print(sh2/sh1)
```

```
0.0198046875
```

Internal friction resonance. We do loading at a frequency of 1 Hz.

```
In [11]: nuIF = 1.
         Trange = np.linspace(250,400,151)
         shlist = []
         for T in Trange:
             beta = 1./(kB*T)
             Llamb = Odiffuser.losstensors(betaCoOthermodict['pre'], beta*betaCoOthermodict['ene'],
                                           [Pdipole],
                                           betaCoOthermodict['preT'], beta*betaCoOthermodict['eneT'])
             f1,L1,f2,L2 = Llamb[0][0], Llamb[0][1], Llamb[1][0], Llamb[1][1]
             sh = crystal.FourthRankIsotropic(L1*nuIF*f1/(nuIF**2+f1**2) +
                                              L2*nuIF*f2/(nuIF**2+f2**2))[1]
             shlist.append(sh*kB*T)
         shear = np.array(shlist)
```

```
In [12]: fig, ax1 = plt.subplots()
         ax1.plot(Trange, shear/np.max(shear), 'k')
         ax1.set_ylabel('loss $Q$ [unitless]', fontsize='x-large')
         ax1.set_xlabel('$T$ [K]', fontsize='x-large')
         plt.show()
         # plt.savefig('FCC-Co-O-loss.pdf', transparent=True, format='pdf')
```



Temperature where peak maximum is found?

```
In [13]: Trange[np.argmax(shear)]
```

```
Out[13]: 326.0
```

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z