

---

# **Onsager Documentation**

***Release 0.9***

**Dallas R. Trinkle**

May 12, 2016



## CONTENTS

<b>1</b>	<b>Onsager</b>	<b>3</b>
<b>2</b>	<b>References</b>	<b>5</b>
<b>3</b>	<b>Contributors</b>	<b>7</b>
<b>4</b>	<b>Support</b>	<b>9</b>
<b>5</b>	<b>Indices and tables</b>	<b>21</b>
	<b>Python Module Index</b>	<b>23</b>
	<b>Index</b>	<b>25</b>



Contents:



## ONSAGER

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

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

Typical usage looks like:

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

from onsager import crystal
from onsager import OnsagerCalc

...
```

Many of the subpackages within Onsager are support for the main attraction, which is in `OnsagerCalc`. Interstitial calculation examples are 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 tests for the package are include in `test`; `tests.py` will run all of the tests in the directory with verbosity level 2. This can be time-consuming (on the order of several of minutes) to run all tests; coverage is currently >90%.

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

Release 0.9: Full release of Interstitial calculator, along with theory paper (see References below).





## REFERENCES

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



## 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).



## SUPPORT

This work has been supported in part by

- DOE/BES grant DE-FG02-05ER46217,
- ONR grant N000141210752,
- NSF/CDSE grant 1411106.
- 4. (a) 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.

Crystal: 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

**Returns** list of tensors

`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)

**Returns** (dim, vect)

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

A class that defines a crystal, as well as the symmetry analysis that goes along with it.

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

Create a body-centered cubic crystal with lattice constant *a0*

**Parameters** **a0** – lattice constant

**Returns** BCC crystal

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

Create a face-centered cubic crystal with lattice constant a0

**Parameters** **a0** – lattice constant

**Returns** 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

**Returns** (list) of our unique vector basis lattice functions, normalized; each is an array

**Returns** (list) of our VV “outer” expansion

**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** – c/a ratio

**Returns** HCP crystal

**SymmTensorBasis** (*ind*)

Generates the symmetric tensor basis corresponding to an atomic site

**Parameters** **ind** – tuple index for atom

**Returns** (dim, vect) – dimension of basis, vector = normal for plane, direction for line

**VectorBasis** (*ind*)

Generates the vector basis corresponding to an atomic site

**Parameters** **ind** – tuple index for atom

**Returns** (dim, vect) – dimension of basis, vector = normal for plane, direction for line

**Wyckoffpos** (*uvec*)

Generates all the equivalent Wyckoff positions for a unit cell vector.

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

**Returns** list of equivalent Wyckoff positions

**addbasis** (*basis, chemistry=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

**Param chemistry** (optional) list of chemistry names

**Returns** new Crystal object, with additional sites

**calcmetric** ()

Computes the volume of the cell and the metric tensor

**Returns** volume, metric tensor

**cart2pos** (*v*)

Return the lattvec and index corresponding to an atomic position in cartesian coord.

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

**Returns** 3-vector (integer) lattice vector in direct coordinates, index tuple of corresponding atom. Returns None on tuple if no match

**cart2unit** (*v*)

Return the lattice and unit cell coord. corresponding to a position in cartesian coord.

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

**Returns** 3-vector (integer) lattice vector in direct coordinates, 3-vector (float) inside unit cell

**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

**Returns** index corresponding to chemistry

**classmethod fromdict** (*yamldict*)

Creates a Crystal object from a YAML-created dictionary

**Parameters** **yamldict** – dictionary; must contain ‘lattice’ (using *row* vectors!) and ‘basis’;

can contain optional ‘lattice\_constant’:return: Crystal(lattice.T, basis)

**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

**Returns** 3-vector position in space (Cartesian coordinates)

**static g\_direct** (*g*, *direc*)

Apply a space group operation to a direction

**Parameters**

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

**Returns** 3-vector direction

**g\_direct\_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

**Returns** 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)

**Returns** 3-vector (integer) lattice vector in direct coordinates, index

**static g\_tensor** (*g*, *tensor*)

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

**Parameters**

- **g** – group operation (GroupOp)
- **tensor** – 2nd-rank tensor

**Returns** 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

**Returns** 3-vector (integer) lattice vector in direct coordinates, location in unit cell in direct coordinates

**genBZG** ()

Generates the reciprocal lattice G points that define the Brillouin zone.

**Returns** array of G vectors that define the BZ, in Cartesian coordinates

**genWyckoffsets** ()

Generate our Wyckoff sets.

**Returns** set of sets of tuples of positions that correspond to identical Wyckoff positions

**gengroup** ()

Generate all of the space group operations.

**Returns** list of group operations

**genpoint** ()

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

**Returns** list of sets 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”

**Returns** 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  $i \rightarrow j$  and  $j \rightarrow i$  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)

**Returns** list of symmetry-unique transitions; each is a list of tuples:  $((i,j), dx)$  corresponding to jump from  $i \rightarrow j$  with vector  $dx$

**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  $i \rightarrow j$  with vector  $dx$

**Returns** list of symmetry-unique transitions; each is a list of tuples:  $((i,j), R)$  corresponding to jump from  $i$  in unit cell 0  $\rightarrow j$  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.

**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

**Returns** 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)

**Returns** 3-vector (float) in Cartesian coordinates

**reduce** (*threshold=1e-08*)

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

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

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

**Parameters**

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

**Return kptsymm** array[Nsymm][3] of kpoints

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

**remapbasis** (*supercell*)

Takes the basis definition, and using a supercell definition, returns a new basis

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

**Returns** atomic basis

**simpleYAML** (*a0=1.0*)

Creates a simplified YAML dump, in case we don't want to output the full symmetry analysis

**Returns** YAML 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

**Returns** 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

**Returns** 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

**Returns** 3-vector (float) in Cartesian coordinates

**static vectlist** (*vb*)

Returns a list of orthonormal vectors corresponding to our vector basis.

**Parameters** **vb** – (dim, v)

**Returns** list of vectors

**class** `crystal.GroupOp`

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

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

**Parameters**

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

**static GroupOp\_constructor** (*loader, node*)

Construct a GroupOp from YAML

**static GroupOp\_representer** (*dumper, data*)

Output a GroupOp

**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].

**Returns** type (integer)

**Returns** 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

**crystal.ProjectTensorBasis** (*tensor, basis*)

Given a tensor, project it onto the basis.

**Parameters**

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

**Returns** tensor, projected

**crystal.SymmTensorBasis** (*rotype, eigenvect*)

Returns a symmetric second-rank tensor basis corresponding to the otype and eigenvectors for a GroupOp

**Parameters**

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

**Returns** list of 2nd-rank symmetric tensors making up the basis

**crystal.VectorBasis** (*rotype, eigenvect*)

Returns a vector basis corresponding to the otype and eigenvectors for a GroupOp

**Parameters**

- **rotype** – output from eigen()

- **eigenvect** – eigenvectors

**Returns** (dim, vect) – dimensionality (0..3), 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

**Returns** symmetric strain tensor

`crystal.incell(vec)`

Returns the vector inside the unit cell (in [0,1]\*\*3)

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

**Returns** 3-vector

`crystal.inhalf(vec)`

Returns the vector inside the centered cell (in [-0.5,0.5]\*\*3)

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

**Returns** 3-vector

`crystal.maptranslation(oldpos, newpos, 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.

**Parameters**

- **oldpos** – list of list of array[3]
- **newpos** – list of list of array[3], same layout as oldpos

**Returns** translation (array[3]), mapping (list of list of indices)

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.

`crystal.ndarray_representer(dumper, data)`

Output a numpy array

CrystalStars:

PowerExpansion: 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, nodeeepcopy=False)

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

**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). :param coefflist: list((n, lmax, powexpansion))

**classmethod checkinternalsHDF5** (*HDF5group*)

Reads the power expansion internals into an HDF5group, and performs sanity check :param HDF5group:

**classmethod coeffproductcoeff** (*a, b*)

Takes a direction expansion a and b, and returns the product expansion. :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

**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. :param a = list((n, lmax, powexpansion)): expansion of function in powers :param 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. :param basis = list((coeffmatrix, vect)): expansions to create;

sum(coeffmatrix \* (vect\*q)^n), for powers n = 0..N

**Parameters**

- **N** – maximum power to consider; for N=-1, use Lmax
- **pre** – list of prefactors, defining the Taylor expansion. Default = 1

**:returns** **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 :param HDF5group:

**ildot** (*c*)

Computes c.self in place

**inv** (*Nmax=0*)

Return the inverse of the expansion, up to order Nmax :param Nmax: maximum order in the inverse expansion :return: 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  $(A+B)^{-1} = ((1+BA^{-1})A)^{-1} = A^{-1}(1-(-BA^{-1}))^{-1} = A^{-1} \sum_{i=0}^{\infty} (-BA^{-1})^i$  :param *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 be 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** **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*

**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

**irdot** (*c*)

Computes self.*c* in place

**irotate** (*powtrans*)

Rotate in place. :param *powtrans*: *Npow* x *Npow* matrix, of [*oldpow*,*newpow*] corresponding to the rotation :return: self

**ldot** (*c*)

Returns *c*.self

**classmethod** `loadhdf5` (*HDF5group*)

Creates a new T3D from an HDF5 group. :param *HDFgroup*: HDF5 group :return: new T3D object

**classmethod** `makeLprojections` ()

Constructs a series of projection matrices for each *l* component in our power series :return: *projL[l][p][p']*  
projection of powers containing *only* *l* component. -1 component =  $\sum_{l=0..Lmax} \text{projL}[l] =$  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** *directmult[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. :param *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 *powrange[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 :param a = list((n, lmax, powexpansion): 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 :param u[3]: vector to apply :param 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.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). :param a = list((n, lmax, powexpansion): expansion of function in powers :param inplace: modify a in place? :return coefflist: a

**rotate (powtrans)**

Return a rotated version of the expansion. :param powtrans: Npow x Npow matrix, of [oldpow,newpow] corresponding to the rotation :return: coefficient list, rotated

**classmethod rotatecoeff (a, npowtrans, inplace=False)**

Return a rotated version of the expansion. Needs to use pad to work with reduced representations. :param a: coefficient list :param npowtrans: Lmax+1 x Npow x Npow matrix, of [n,oldpow,newpow] corresponding to the rotation :return: 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. :param qptrans: 3x3 matrix :return: Lmax +1 x Npow x 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 :param c: scalar or dictionary mapping (n,l) to scalars :param a = list((n, lmax, powexpansion): expansion of function in powers :param 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 separaterecoeff (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  $\leq l$ . :param a = list((n, lmax, powexpansion): expansion of function in powers :param 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 be 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 :param c: array *or* dictionary mapping (n,l) to arrays :param a = list((n, lmax, powexpansion): expansion of function in powers :param leftmultiply: tensor-dot(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 :param Nmax: maximum coefficient to include :param inplace: do it in place?

**classmethod truncatecoeff** (*a, Nmax, inplace=False*)

Remove the coefficients above a given Nmax; normally returns a new object :param Nmax: maximum coefficient to include :param a = list((n, lmax, powexpansion): expansion of function in powers :param 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. :param nmin: minimum value of n :param nmax: maximum value of n (inclusive) :param shape: shape of matrix, as zeros would expect. :return: Taylor3D, with a zero coefficient list

GFcalc:

OnsagerCalc:



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### c

`crystal`, 9

### p

`PowerExpansion`, 16



## A

addbasis() (crystal.Crystal method), 10  
 addhdf5() (PowerExpansion.Taylor3D method), 16  
 addterms() (PowerExpansion.Taylor3D method), 17

## B

BCC() (crystal.Crystal class method), 9

## C

calcmetric() (crystal.Crystal method), 10  
 cart2pos() (crystal.Crystal method), 10  
 cart2unit() (crystal.Crystal method), 11  
 center() (crystal.Crystal method), 11  
 checkinternalsHDF5() (PowerExpansion.Taylor3D class method), 17  
 chemindex() (crystal.Crystal method), 11  
 coeffproductcoeff() (PowerExpansion.Taylor3D class method), 17  
 collectcoeff() (PowerExpansion.Taylor3D class method), 17  
 CombineTensorBasis() (in module crystal), 9  
 CombineVectorBasis() (in module crystal), 9  
 constructexpansion() (PowerExpansion.Taylor3D class method), 17  
 copy() (PowerExpansion.Taylor3D method), 17  
 Crystal (class in crystal), 9  
 crystal (module), 9

## D

dumpinternalsHDF5() (PowerExpansion.Taylor3D method), 17

## E

eigen() (crystal.GroupOp method), 15

## F

FCC() (crystal.Crystal class method), 9  
 fromdict() (crystal.Crystal class method), 11  
 fullkptmesh() (crystal.Crystal method), 11  
 FullVectorBasis() (crystal.Crystal method), 10

## G

g\_cart() (crystal.Crystal method), 11  
 g\_direct() (crystal.Crystal static method), 11  
 g\_direct\_equivalent() (crystal.Crystal method), 11  
 g\_pos() (crystal.Crystal method), 12  
 g\_tensor() (crystal.Crystal static method), 12  
 g\_vect() (crystal.Crystal static method), 12  
 genBZG() (crystal.Crystal method), 12  
 gengroup() (crystal.Crystal method), 12  
 genpoint() (crystal.Crystal method), 12  
 genWyckoffsets() (crystal.Crystal method), 12  
 GroupOp (class in crystal), 14  
 GroupOp\_constructor() (crystal.GroupOp static method), 15  
 GroupOp\_representer() (crystal.GroupOp static method), 15

## H

HCP() (crystal.Crystal class method), 10

## I

ident() (crystal.GroupOp class method), 15  
 ildot() (PowerExpansion.Taylor3D method), 17  
 inBZ() (crystal.Crystal method), 12  
 incell() (crystal.GroupOp method), 15  
 incell() (in module crystal), 16  
 inhalf() (crystal.GroupOp method), 15  
 inhalf() (in module crystal), 16  
 inv() (crystal.GroupOp method), 15  
 inv() (PowerExpansion.Taylor3D method), 17  
 inversecoeff() (PowerExpansion.Taylor3D class method), 17  
 irdot() (PowerExpansion.Taylor3D method), 18  
 irotate() (PowerExpansion.Taylor3D method), 18

## J

jumpnetwork() (crystal.Crystal method), 13  
 jumpnetwork2lattice() (crystal.Crystal method), 13

## L

ldot() (PowerExpansion.Taylor3D method), 18  
 loadhdf5() (PowerExpansion.Taylor3D class method), 18

## M

`makedirectmult()` (`PowerExpansion.Taylor3D` class method), 18  
`makeindexPowerYlm()` (`PowerExpansion.Taylor3D` static method), 18  
`makeLprojections()` (`PowerExpansion.Taylor3D` class method), 18  
`makepowercoeff()` (`PowerExpansion.Taylor3D` class method), 18  
`makepowYlm()` (`PowerExpansion.Taylor3D` class method), 18  
`makeYImpow()` (`PowerExpansion.Taylor3D` class method), 18  
`maptranslation()` (in module `crystal`), 16  
`minlattice()` (`crystal.Crystal` method), 13

## N

`ndarray_representer()` (in module `crystal`), 16  
`negcoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`nl()` (`PowerExpansion.Taylor3D` method), 19  
`nnlist()` (`crystal.Crystal` method), 13

## P

`pos2cart()` (`crystal.Crystal` method), 13  
`PowerExpansion` (module), 16  
`powexp()` (`PowerExpansion.Taylor3D` class method), 19  
`ProjectTensorBasis()` (in module `crystal`), 15

## R

`rdot()` (`PowerExpansion.Taylor3D` method), 19  
`reduce()` (`crystal.Crystal` method), 13  
`reduce()` (`PowerExpansion.Taylor3D` method), 19  
`reducecoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`reducekptmesh()` (`crystal.Crystal` method), 14  
`remapbasis()` (`crystal.Crystal` method), 14  
`rotate()` (`PowerExpansion.Taylor3D` method), 19  
`rotatecoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`rotatedirections()` (`PowerExpansion.Taylor3D` class method), 19

## S

`scalarproductcoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`separate()` (`PowerExpansion.Taylor3D` method), 19  
`separatecoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`simpleYAML()` (`crystal.Crystal` method), 14  
`sitelist()` (`crystal.Crystal` method), 14  
`strain()` (`crystal.Crystal` method), 14  
`sumcoeff()` (`PowerExpansion.Taylor3D` class method), 19  
`SymmTensorBasis()` (`crystal.Crystal` method), 10

`SymmTensorBasis()` (in module `crystal`), 15

## T

`Taylor3D` (class in `PowerExpansion`), 16  
`tensorproductcoeff()` (`PowerExpansion.Taylor3D` class method), 20  
`truncate()` (`PowerExpansion.Taylor3D` method), 20  
`truncatecoeff()` (`PowerExpansion.Taylor3D` class method), 20

## U

`unit2cart()` (`crystal.Crystal` method), 14

## V

`vectlist()` (`crystal.Crystal` static method), 14  
`VectorBasis()` (`crystal.Crystal` method), 10  
`VectorBasis()` (in module `crystal`), 15  
`Voigtstrain()` (in module `crystal`), 16

## W

`Wyckoffpos()` (`crystal.Crystal` method), 10

## Z

`zeros()` (`PowerExpansion.Taylor3D` class method), 20