* → Important imports

import PowerExpansion as PE

from scipy.special import hyp1f1, gamma, expi

* scipy.special.hyp1f1(a, b, x)

- confluent hypergeometric function 1F1

→ From Wikipedia:-

$$1F1(a,b;x) = \sum_{n=0}^{\infty} \frac{a^{(n)} x^n}{b^{(n)} n!}$$

where $a^{(n)} = a(a+1)(a+2)\ldots(a+n-1)$

$a^{(0)} = 1$

is called the rising factorial.

This function $1F1(a,b;x)$ is one of the two linearly independent solutions of the Kummer's differential equation, given by:-

$$x\frac{\partial^2 w}{\partial x^2} + (b-x)\frac{dw}{dx} - aw = 0$$

w has a "regular singular point" at $x=0$ and an irregular singular point at $x=\infty$

What is a regular singular point → Ask Dallas if it is necessary.

Also note the following property of $1F1(a,b;x)$

$$1F1(a,b;x) = e^x\, 1F1(b-a, b; -x)$$

* scipy.special.gamma

$$\text{scipy.special.gamma}(z) = \int_0^{\infty} x^{z-1} e^{-x}\, dx = (z-1)!$$

(The Gamma function)

Revise the props. of the gamma function from a mathematical methods book later on.

Mainly :- The Gamma function is defined for $z > 0$

→ for the smallest $\frac{1}{2}$: $\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$

→ Recursion → $\Gamma(z) = (z-1)\,\Gamma(z-1)$

$\Gamma(z) = (z-1)! = (z-1)(z-2)! = (z-1)\,\Gamma(z-1)$

$\therefore \Gamma(z-1) = (z-2)!$

similarly $\Gamma(z+1) = z\,\Gamma(z)$

$\Gamma(3/2) = \Gamma(1/2+1) = \frac{1}{2}\,\Gamma(1/2) = \frac{1}{2}\sqrt{\pi}$

and so on

```
class GFCrystalcalc (object):

    def __init__(self, crys, clum, sitelist, jumpnetwork, Nmax=4, kptwt=None):

        parameters:
            Nmax → maximum range as estimator for k-point mesh
            generator (?).
            Kptwt → To short-circuit kpt mesh generation.

        self.crys = crys
        :
        self.N = sum(len(w) for w in sitelist)
        self.invmap = np.zeros(self.N)
        for ind, w in sitelist
            for i in w:
            self.invmap[i] = ind  # which Wyckoffset the given basis
                                  # site belongs to

        self.Ndiff = self.networkcount(jumpnetwork, self.N)

        self.NG = len(self.crys.G)

        self.grouparray, self.indexpair = self.BreakdownGroups()

    # Kpt mesh generation :-

        bmagn = np.array([np.sqrt(np.dot(crys.reciplatt[:,i],
                                         crys.reciplatt[:,i]))
                          for i in range(3)])

        bmagn /= np.power(np.product(bmagn),1/3)

        bmagn = [ b1/(b1 b2 b3)^1/3 , b2/(b1 b2 b3)^1/3 , b3/(b1 b2 b3)^1/3 ]
```

→ why?

```
self.kptgrid = np.array([2*np.int(np.ceil(2*Nmax*b))
                         for b in bmagn], dtype=int)
if kptwt is None else np.zeros(3, int)

self.kpts, self.wts =
    crys.reducekptmesh(crys.fullkptmesh(self.kptgrid))
if kpt is None
    else dupcopy(kptwt).
```

??

Need to do this later

self.NKpts = self.kpts.shape[0]

#Generate the Fourier transformations for each jump.
#includes the multiplicity for the onsite(?) terms??

site expansions??

self.FTjumps, self.SEjumps = self.FourierTransformJumps(
jumpnetwork, self.N, self.kpts)

#Generate the Taylorexpansion coefficients for each jump:-

self.TaylorJumps = self.TaylorExpandJumps(jumpnetwork,
self.N)

self.jumppairs =
tuple([(self.invmap[jumplist[0][0][0]],
self.invmap[jumplist[0][0][1]])
for jumplist in jumpnetwork])

Which Wyckoff set in "sitelist" the initial & final basis site of a representative jump belong to.

self.jumppairs → stores the Wyckoff sets to which the initial and final basis sites of a representative jump belong to as a tuple of the form:-

(initial site's Wyckoff set index, final site's Wyckoff set index)

⋮

for 1st jump of every sym-unique jump type.

self.D , self.eta = 0, 0.

→ called by self.Ndiff = self.networkcount (jumpnetwork, self.N)

def networkcount (jumpnetwork, N): → @staticmethod

→ Returns a count of how many seperate connected networks there are.

* Lets try to understand how it is done.

What is self.N? ↓ total no. of sites

```
jngraph = np.zeros ((N, N), dtype = bool)
for jlist in jumpnetwork:
    for (i,j), dx in jlist:
        jngraph[i, j] = True

connectivity = 0
disconnected = {i for i in range (N)}

While len (disconnected) > 0:
    i = min (disconnected)
    cset = {i}
    disconnected.remove (i)

    While True:
        clen = len (cset)
        for n in cset.copy ():
            for m in disconnected.copy ():
                if jngraph [n, m]:
                    cset.add (m)
                    disconnected.remove (m)
        if clen == len (cset): break.

    connectivity += 1

return connectivity.
```

Note that this indicates that at some point the set must become empty.

but what does this do.

if any basis site i and j are connected by a jump of any distance we set jngraph [i, j] = True.

say N = 3 → disconnected = {0, 1, 2}

⊛ Now observe the code for a few seconds.

"connectivity" increases by one at each iteration whenever the length of disconn. is found to be > 0. But [if say 0, 1, 2 were all connected], i.e., 0 was

connected to 1 and (to 2 so to 2), then in the 1st iteration itself, the length of

→ See the notebook for connectivity.

def FourierTransformJumps:

FTjumps → easy to understand but I don't understand SE jumps.

SE → site expansion, → ?? what does site expansion mean.

→ SEjumps → shape [Nsites][Njumps]

↓

From line 253 in GFCalc, I think this just measures the no. of of jumps of a particular symmetry type originating from a given site.

→ But why do this?

→ This no. will be combined with FTjumps → which stores info only regarding the initial jump.

↳ see how later on.

def TaylorExpandJumps(self, jumpnetwork, N):

Taylor() → what does it mean to initialize a class. This just calls init with all arguments set to default values. But why is this necessary?

```
pre = np.array([1/factorial(n, True)
                for n in range(Taylor.Lmax+1)])
TaylorJumps = []
for jumplist in jumpnetwork:
    # coefficients (??)
    c = [[n, np.zeros((Taylor.powerrange[n], N, N),
                      dtype=complex)
          for n in range(Taylor.Lmax+1)]
    for (i,j), dx in jumpnetwork:
        perp = Taylor.powerexp(dx, normalize=False)
        for n in range(Taylor.Lmax+1):
            (c[n][2])[:, i, j] += pre[n]
```

* [Taylor.powercoeff[n] * pexp)[: Taylor.powlrange[n]]

Taylorjumps.append(Taylor(c))

return Taylorjumps.

Nov 3

Funkin → what is this? → Lature

⇒ TaylorExpandJumps

pre = [$i^n/n!$ for n in range (Lmax+1)]

pre = $[1, -\frac{1}{2}, \frac{-i}{6}, \frac{1}{24}, \cdots]$

Taylorjumps = [ ]

for jumplist in jumpnetwork:

c = [(n, n, up.zeros((Taylor.powlrange[n], N, N), dt = complex)
for n in range (Taylor.Lmax + 1)]

→ shape ?

n

N ---- M

say N = 3.

n = 2.

[0, 2, 1]

[0, 1, 0]

[0, 20]

(2, 2, [[[0+0i, 0+0i, 0+0i] , [0+0i, 0+0i, 0+0i] , [0+0i, 0+0i, 0+0i]] ,

[[0+0i, 0+0i, 0+0i] , [0+0i, 0+0i, 0+0i] , [0+0i, 0+0i, 0+0i]]])

N=3 say

[(0, 0, $(0+0j)_{1\times3\times3}$), (1, 1, $(0+0j)_{4\times3\times3}$), (2, 2, $(0+0j)_{10\times3\times3}$),

c= (3, 3, $(0+0j)_{20\times3\times3}$), (4, 4, $(0+0j)_{35\times3\times3}$),

$(0, 0, [])$]

↳ Remember the last element is 0 in powrange

for (i,j), dx in jmplist:

pexp = Taylor.powexp(dx, normalize = False)

pexp = $\left[1, dx_x^{n_0^1} dx_y^{n_1^1} dx_z^{n_2^1}, dx_x^{n_0^2} dx_y^{n_1^2} dx_z^{n_2^2}, \ldots dx_x^{n_0^{last}} dx_y^{n_1^{last}} dx_z^{n_2^{last}}\right]$

where ind2pow = $\left[[0,0,0], [n_0^1, n_1^1, n_2^1], [n_0^2, n_1^2, n_2^2], \ldots, [n_0^{last}, n_1^{last}, n_2^{last}]\right]$ → 35 entries for Lmax = 4

$\Rightarrow$ for n in range(Lmax + 1):

(c[n][2])[:, i, j] += pre[n] * (powercoeff[n] * pexp)

[: powrange[n]]

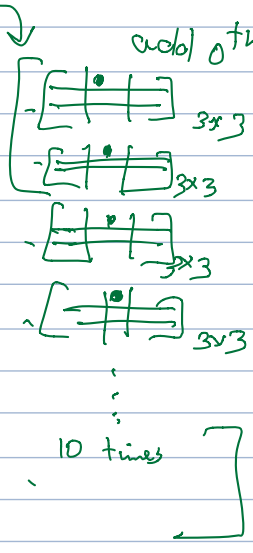
Let's break this down. Say n = 2

then we have the LHS as:-

c[2][2] = $(0+0j)_{10\times3\times3}$ [:, i, j]

↘ add 0th of

say i = 0, j = 1



1, 4, 10

powercoeff = zeros((Lmax+1, Npower))

	0	1	2	3	4	5	6 ... 34
0	1						
1		1					
2	0	0	0	0	1	2	1
3							
4							

for n0 in range (Lmax+1):
  for n1 in range (Lmax+1):
    for n2 in range (Lmax+1):

$n = n_0 + n_1 + n_2$

if $n \le L_{max}$

powercoeff [n, pow2ind[n0, n1, n2]]

$= \frac{n!}{n_0!\, n_1!\, n_2!}$

0	0	0
0	0	1
0	0	2

$n = 2$

$\frac{2!}{2!} = 1$

(powercoeff[n] * pexp) [: pow1range[n]]

pexp = $[1,\ dx_x^0 dx_y^0 dx_z^1,\ dx_x^0 dx_y^1 dx_z^0,\ dx_x^1 dx_y^0 dx_z^0,$
$dx_x^0 dy_y^0 dx_z^2,\ dx_x^0 dx_y^1 dx_z^1,\ dx_x^0 dx_y^2 dx_z^0,\ dx_x^1 dx_y^0 dx_z^1,$
$dx_x^1 dx_y^1 dx_z^0,\ dx_x^2 dx_y^0 dx_z^0,\ dx_x^0 dx_y^0 dx_z^3,\ \dots]$

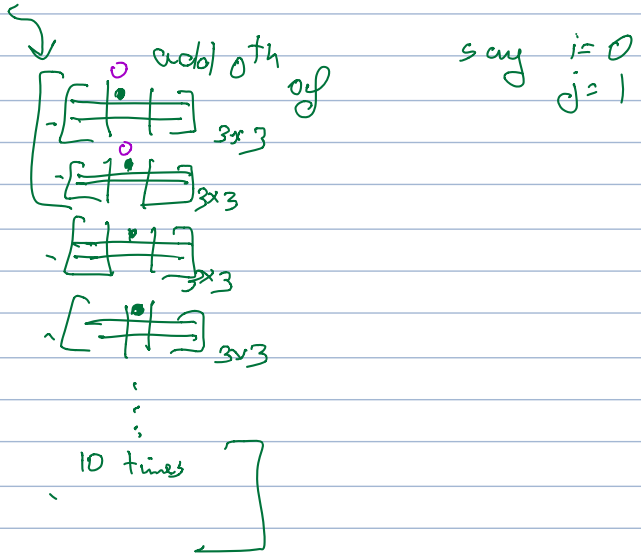
powercoeff[2] * pexp [: 10]

$[0\times 1,\ 0\times dx_x^0 dx_y^0 dx_z^1,\ 0\times dx_x^0 dx_y^1 dx_z^0,\ 0\times dx_x^1 dx_y^0 dx_z^0,$
$1\times dx_x^0 dy_y^0 dx_z^2,\ 2\times dx_x^0 dx_y^1 dx_z^1,\ 1\ dx_x^0 dx_y^2 dx_z^0,$
$0\times dx_x^1 dx_y^0 dx_z^1,\ 0\times dx_x^1 dx_y^1 dx_z^0,\ 0\, dx_x^2 dx_y^0 dx_z^0]$

$= [0, 0, 0, 0, dx_z^2, 2dx_y dx_z, dx_y^2, 0, 0, 0]$

$(c[2][2])[:, i, j]$ +=

$c[2][2] = (0 + 0j)_{10\times3\times3}\ [:, i, j]$



1, 4, 10

$$e^{-i\vec{q}\cdot\delta\vec{x}} = \left[1 - i\vec{q}\cdot\delta\vec{x} - \frac{1}{2}(\vec{q}\cdot\delta x)^2 + \frac{i}{6}(\vec{q}\cdot\delta x)^3 + \frac{1}{24}(\vec{q}\cdot\delta x)^4 + \dots\right]$$

$$\vec{q} = \sum_i d_i^{-1/2} p_i \hat{e}_i \qquad \delta x = \sum_i d_i^{1/2} y_i \hat{e}_i$$

$$\vec{q}\cdot\delta\vec{x} = \sum_i d_i^{-1/2} p_i \hat{e}_i \cdot \sum_j d_j^{1/2} y_j \hat{e}_j$$

$$= \sum_{ij} p_i y_j d_i^{-1/2} d_j^{1/2} \hat{e}_i \cdot \hat{e}_j$$

$$= \sum_{ij} p_i y_j d_i^{-1/2} d_j^{1/2} \delta_{ij}$$

$$= \sum_i p_i y_i$$

$$= \vec{p}\cdot\vec{y}$$

$$e^{-i\vec{q}\cdot\delta\vec{x}} = e^{-i\vec{p}\cdot\vec{y}}$$

20 rows

$(c[3][2])(:, i, j)$

↳ 10^{th} to 13^{th}

$$= [0, 0, 0, \ldots 0 \; dx_3^3, \; 3dx_y \, dx_3^2, \; 3dx_y^2 dx_3, \; dx_y^3,$$
$$0, 1, 2, \ldots, 9, \quad 10 \quad 11 \quad 12 \quad 13$$

$$0, 0, 0, 0 \ldots, 0 \;]$$
$$14 \quad 15 \quad 16 \quad 17 \ldots, 19$$

Taylor jumps. append (Taylor (c))

$e^{-i\vec{q}\cdot dx}$

$(\vec{q}\cdot\delta x)^2 = q^2(\hat{q}\cdot\delta\vec{x})^2$

Solution

$q^2(\hat{q}\cdot\delta\vec{x}_{ij})^2 = q^2(\hat{q}_x dx_x + \hat{q}_y dx_y + \hat{q}_3 dx_3)^2$

$= q^2\left(\hat{q}_x^2 dx_x^2 + \hat{q}_y^2 dx_y^2 + \hat{q}_3^2 dx_3^2 + 2\hat{q}_x\hat{q}_y dx_x dx_y \right.$
$\left. + 2\hat{q}_x\hat{q}_3 dx_x dx_3 + 2\hat{q}_y\hat{q}_3 dx_y dx_3\right)$

coefficients → $dx_x^2, \; dx_y^2, \; dx_3^2, \; 2q_x q_y, \; 2q_x q_3, \; 2q_y q_3$

why are only three of these being stored?

$E_g(x) = \frac{w_b}{2}\left(1 - \cos\left(\frac{2\pi g(x)}{b}\right)\right)$

$w_0 = \alpha G b^2$

$w \approx b\sqrt{\frac{w_0}{w_b}}$

$$0 = \frac{2b}{\pi}\tan^{-1}\left(e^{2\pi x/b\sqrt{w_b/w_0}}\right)$$

$$e^{2\pi x/b\sqrt{w_b/w_0}} = 0$$

SymmRates (pre, betaene, preT, betaeneT, pmaxerror = 1e-8)

self.jumppairs = tuple ((self.invmap [jumplist[0][0][0]],
self.invmap [jumplist[0][0][1]])
for jumplist in jumpnetwork.

self.invmap

self.invmap = zeros (N)
for ind, we in enumerate (sitelist):
for i in w:
self.invmap [i] = ind

invmap → stores which wyckoff set each jump belongs to.

$\{0.5 \times betaene[w_0] + 0.5\ betaene[w_1] - beT\}$

$$\frac{pt * e}{\sqrt{pre[w_0]} \times \sqrt{pre[w_1]}}$$ ← symmrates

↓

for (w_0, w_1), pT, beT in zip (jumppairs, preT, betaeneT)

$i \in w_0$, $j \in w_i$ → wyckoff sets.

$$(p^{0,i})^{1/2} \left(pt\ e^{-[betaeneT - betaene_i]} \right) (p^{0,j})^{-1/2}$$

$$\frac{e^{-\frac{1}{2} betaene_i}}{\sqrt{pre_i}} \quad pt\ e^{-betaeneT + betaene_i} \quad \frac{e^{+betaene_j/2}}{\sqrt{pre_j}}$$

---

sum_ar = []
for i, w_i in enumerate (self.invmap):
sum = 0
for J, pretrans, BeT in zip (count(), preT, betaeneT):
sum += SEjumps [i, J] * pretrans / pre [w_i] × $e^{(BE[w_i] - BeT)}$
sum_ar.append (sum) ← sum of the escape rates out of state i

dimension of
Sum_arr → N x 1

may be J.

→ np.diag(sum_arr) = N x N → self.escape,

→ what is omega_qij

omega_qij = np.tensordot(self.symmrate, self.FTjumps, axes=(0,0))

now this is equivalent to :-

$$\text{omega\_q}_{ij} = \sum_{j=1}^{J} \text{symmrate}[j] \times \text{FTjumps}[j, :, :, :]$$

FTjumps[j, :, :, :] = matrix NKpts x N x N

$$\text{FTjumps}[j, Kpt, i, j] = \sum_{jmp \in j} e^{i \vec{Kpt} \cdot \vec{dx}_{ij}^{jmp}}$$

$$\text{symmrate}[j]\,[j, Kpt, i, j] = w_j \sum_{jmp \in j} e^{i \vec{kpt} \cdot \vec{Sx}_{ij}^{jmp}}$$

$$\text{omega\_qij}[Kpt][i][j] = \sum_{j=1}^{J} \sum_{jmp \in j} w_j e^{i \vec{Kpt} \cdot Sx_{ij}^{jmp}}$$

expanding RHS out fully, we get :-

$$\sum_{x, ab} w_{\vec{0}a, \vec{x}b} \; e^{i \vec{q} \cdot \vec{Sx}_{0a, xb}}$$

→ Now recall that $\sum_{x, ab} s_a^{\alpha} \, w_{0a, xb} \, s_b^{\beta} \, e^{i \vec{q} \cdot \vec{Sx}_{0a, xb}} = w^{\alpha\beta}(q)$

so except for s_a^{α} s_b^{β}, this seems complete.

* How is self.escape added to omega_qij ?

code → omega_qij[:] += self.escape.

for any Kpoint kpt, Omg_qij.[kpt] = N x N matrix

self.escape = N x N matrix, diagonal -

note that before self.escape is added, omega_qij[kpt][i][i] for any i = 0.

After adding the escape rates :-

$\Rightarrow$ omega_qij [Rpt] [i][i] = $-\sum_{\text{jumps}}^{\text{total}}$ escape rate from i.

→ self.escape [i][i]

= sum (self.SEjumps [i, J] * pT * $e^{BE(w_i) - BeT}$

sum of escape rates out of state 'i' due to all possible jumps out of state i.

for J, pT, BeT in zip (count , boeT, BetaemeT))

---

* self.omega_Taylor = sum (symmrate * expansion for symmrate, expansion in zip (self. symmrate, self.Taylor jumps))

self. omega_Taylor += self. escape.

Omega_Taylor

Taylorjumps = []

for jumplist in jumpnetwork :

C=[(n, n, np.zeros ([Taylor.powlrange [n] , N, N)), dt = complex) for n in range (Taylor. Lmax + 1)]

* Diagonalize the Gamma point value :-

self. r, self. vr = self. Diaggamma ()

↳ omega_qij for q = 0 → diagonalized, eigenvalues r and eigenvectors vr are returned.

if not np. allclose (self.r [: self. Ndiff], 0):

raise ArithmeticError ("did not find Ndiff eqb solution to rates.")

(1) Things to look up in Taylor_3D :-

{ ⇒ __mul__ with an array of rates.
⇒ __add__ with another Taylor object

→ Can you write the math down and see what's going on?

These two together will give us an idea regarding what is going on in omega_Taylor (line 331)

⇒ What happens when we multiply a Taylor3D object with a coefficient list C, that sums up the Taylor expansion terms of all the jumps in a given jump list?

(*) structure of the coefficient list 'C' :-

in a given jump list

→ consider a jump dx_{ij}

we have $\omega_{ij}(\vec{q}) = \sum_{J=1}^{jtypes} \sum_{jmp \in J} \omega_J \, e^{i \vec{q} \cdot \delta \vec{x}_{jmp}}$

now for a given k^{th} jump $jmp \equiv (i_k, j_k)$, $dx_{(ij)_k}$

$$e^{i \vec{q} \cdot \delta \vec{x}_{jmp}} = 1 + i \vec{q} \cdot \delta \vec{x}_{(ij)_k} - \frac{i}{2} \left(\vec{q} \cdot \delta \vec{x}_{(ij)_k} \right)^2 - \frac{i}{6} \left(\vec{q} \cdot \delta \vec{x}_{(ij)_k} \right)^3 + \frac{1}{24} \left(\vec{q} \cdot \delta \vec{x}_{(ij)_k} \right)^4$$

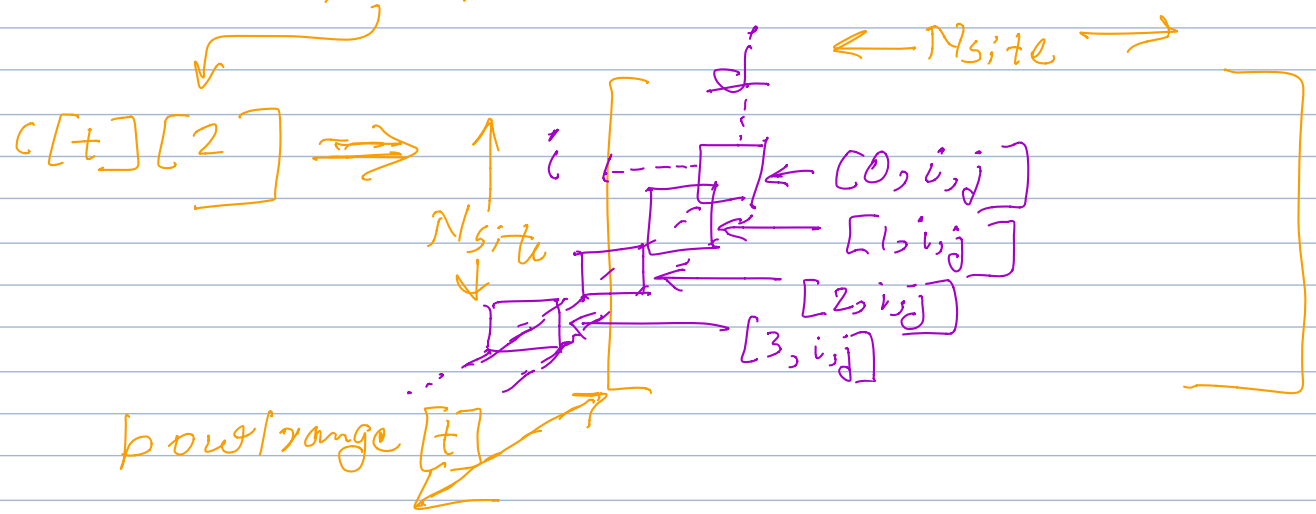
C[0] C[1] C[2] C[3] C[4]

For a given jump list, indexed by J :-

$C[t] = [t, t, (\text{array of coefficients for } t^{th} \text{ term in the above expansions summed over all of the jumps in } J)]$

(array of coefficients for t^{th} terms in the above expansion, summed over all of the jumps in $\mathcal{J}$)

↳ shape



Say, $t = 0$

* → Length of first axis powlrange[0] = 1

$$\square \leftarrow [0, i, j] = \sum_{jump \in \mathcal{J}} \delta(jump.i - i)\, \delta(jump.j - j) + 0_j$$

Store this $1 \times N_{site} \times N_{site}$ matrix in $c[1][2]$

$\Rightarrow$ $t = 1$, ie, we are storing expansion of $+i(\vec{q} \cdot \delta x)$

* → Length of first axis powlrange[1] = 4

$$\square \leftarrow [0, i, j] = 0 \quad \leftarrow \text{The previous terms are neglected}$$

$$\square \leftarrow [1, i, j] = \sum_{jump \in \mathcal{J}} (1 + 0_j)\, \delta x_x\, \delta(jump.i - i)\, \delta(jump.j - j)$$

$$\square \leftarrow [2, i, j] = \sum_{jump \in \mathcal{J}} (1 + 0_j)\, \delta x_y\, \delta(jump.i - i)\, \delta(jump.j - j)$$

$$\square \leftarrow [3, i, j] = \sum_{jump \in \mathcal{J}} (1 + 0_j)\, \delta x_z\, \delta(jump.i - i)\, \delta(jump.j - j)$$

Store these $4 \times N_{site} \times N_{site}$ matrix in $c[1][2][:, i, j]$

(See the Jupyter notebook for the exact order of the terms in terms of the components of dx).

Similarly for $t=2$,

* → length of first axis bounds range [2] = 10

$\square \leftarrow [0, i, j] = 0$ ← The previous terms are neglected

$\square \leftarrow [1, i, j] = 0$

$\square \leftarrow [2, i, j] = 0$

$\square \leftarrow [3, i, j] = 0$

$$\square \leftarrow [4, i, j] = \sum_{jump \in J} (1 + 0_j) \frac{\delta x^2}{3} \delta(jump.i - i) \delta(jump.j - j)$$

$$\square \leftarrow [5, i, j] = \sum_{jump \in J} (1 + 0_j) \delta x \delta x \delta x \delta(jump.i - i) \delta(jump.j - j)$$

⋮

[see the Jupyter notebook for the rest of the terms].

The final array of self.Taylorjumps contains these coefficient containers C for each type of jump in the jumpnetwork.

⊛ Now, what happens when we multiply an expansion with a rate, and add another Taylor Taylor expansion.

Looking at __mul__ and __rmul__ for Taylor3D object

7 steps of the setRates function

⊛ Step 0 - construct the Taylor series expansions of the jumps.
⊛ Step 1 - Diagonalize the Fourier Transform at the gamma point.
⊛ Step 2 - Calculate eta and diffusivity.
⊛ Step 3 - Spatially rotate the Taylor expansion.
⊛ Step 4 - Invert the Taylor expansion using the block inversion formula and truncate at $n=0$.
⊛ Step 5 - Invert the Fourier expansion.
⊛ Step 6 - Slice for faster evaluation.

Step 0 - construct Taylor series expansions of the jumps.
- Lines 319 - 339

self.symmrate = self.symmrates(pre, betaeme, preT, betaemeT)
↦ Get the symmetrized jump rates.

self.maxrate = self.symmrate.max()

-- initTay for 3D indexing --

self.symmrate /= self.maxrate → normalize all the rates with respect to the maximum rate.

self.escape = -np.diag([sum(self.SEjumps[i,J] * pretrans / pre[wi] * exp(betaeme[wi] - beT) for J, pretrans, beT in zip(count(), preT, betaemeT)) for i, wi in enumerate(self.invmap)]) / self.maxrate

↦ write in expanded terms :-

First, self.invmap :-
for ind, w in enumerate(site list) :
for i in w : #i → state (in this case site) index
self.invmap[i] = w
→ stores which Wyckoff list a given site belongs to.

list = []
for i, wi in enumerate(sitelist) : i → site index
sum = 0 w → Wyckoff list index where i belongs.
for J, pretrans, beT in zip(count(), preT, betaemeT) :
sum += self.escape[i,J] * $\frac{\text{pretrans} \times \exp(\text{betaeme}[wi] - beT)}{\text{pre}[wi]}$
list.append(sum)

→ so what are the elements of list :-
* We have :-

$$a_{ii} = -\sum_j W_{ij}$$

$$\Rightarrow a_{ii} = -(W_{i1} + W_{i2} + W_{i3} + \dots)$$

$$= -\sum_{\text{symmjumpgroup}} \sum_j W_{ij}\, \delta(W_{ij} \in \text{symmjumpgroup})$$

$$= \sum_j W_{ij}^1 + \sum_j W_{ij}^2 + \sum_j W_{ij}^3 + \dots$$

where $w_{ij}^{x} = w_{ij} \; \forall \; (w_{ij} \in \text{symmjumpgroup}\, x)$

now $\sum_j w_{ij}^{x} = w_x \times N_{w_{ij} \in x}$

(w_x: The rate of the jumpgroup; $N_{w_{ij} \in x}$: The no. of jumps in that jumpgroup)

$$\Rightarrow \sum_j w_{ij}^{x} = N_{w_{ij} \in x} \times \frac{preT[x]}{pre[w_{yckoff_i}]} \times \exp\left(betaene[w_{yckoff_i}] - beT[x]\right)$$

$$\therefore w_{ii} = \sum_x \sum_j w_{ij}^{x}$$

$$= \sum_x N_{w_{ij} \in x} \times \frac{preT[x]}{pre[w_{yckoff_i}]} \times \exp\left(betaene[w_{yckoff_i}] - beT[x]\right)$$

where the sum is over all jump types.

$\Rightarrow$ In the notation of the array SEjumps :-

$$w_{ii} = -\sum_x SEjumps[i, x] \times \frac{preT[x]}{pre[w_{yckoff_i}]} \times \exp\left(betaene[w_{yckoff_i}] - beT[x]\right)$$

$\therefore$ self.escape = $\begin{bmatrix} w_{11} & & & \\ & w_{22} & & \\ & & w_{33} & \\ & & & \ddots \end{bmatrix}$ ← diagonal.

self.escape != self.maxrate.

now, let's write this back in the compact form.

self.escape = -np.diag([sum([self.SEjumps[i, J] * bretrans / pre[wi] × exp(betaene[wi] - beT)
for J, bretrans, beT in zip(count, preT, betaeneT)])
for i, wi in enumerate(self.indexmap)])
/ self.maxrate

self.escape = -np.diag([sum([self.SEjumps[i, J] × bt / b[wi] × exp(betaene[wi] - beT)
for J, bt, beT in zip(count(), preT, betaeneT)])
for i, wi in enumerate(indexmap)])

= np.diag([w_{11}, w_{22}, w_{33}, ...])

where $\omega_{11} = -\sum_{x} N_{\omega_j \in -x} \times \frac{pre_T[x]}{pre[\omega_1]} \times exp\left(beta_{ene}[\omega_1] - beta_{ene_T}[x]\right)$

$\omega_{22} = -\sum_{x} N_{\omega_2 j \in -x} \times \frac{pre_T[x]}{pre[\omega_2]} \times exp\left(beta_{ene}[\omega_2] - beta_{ene_T}[x]\right)$

$\therefore$ self.escape = $\begin{bmatrix} \omega_{11} & & & & \\ & \omega_{22} & & & \\ & & \omega_{33} & & \\ & & & \ddots & \\ & & & & \omega_{NN} \end{bmatrix}$

N → no of sites of ce given chemistry.

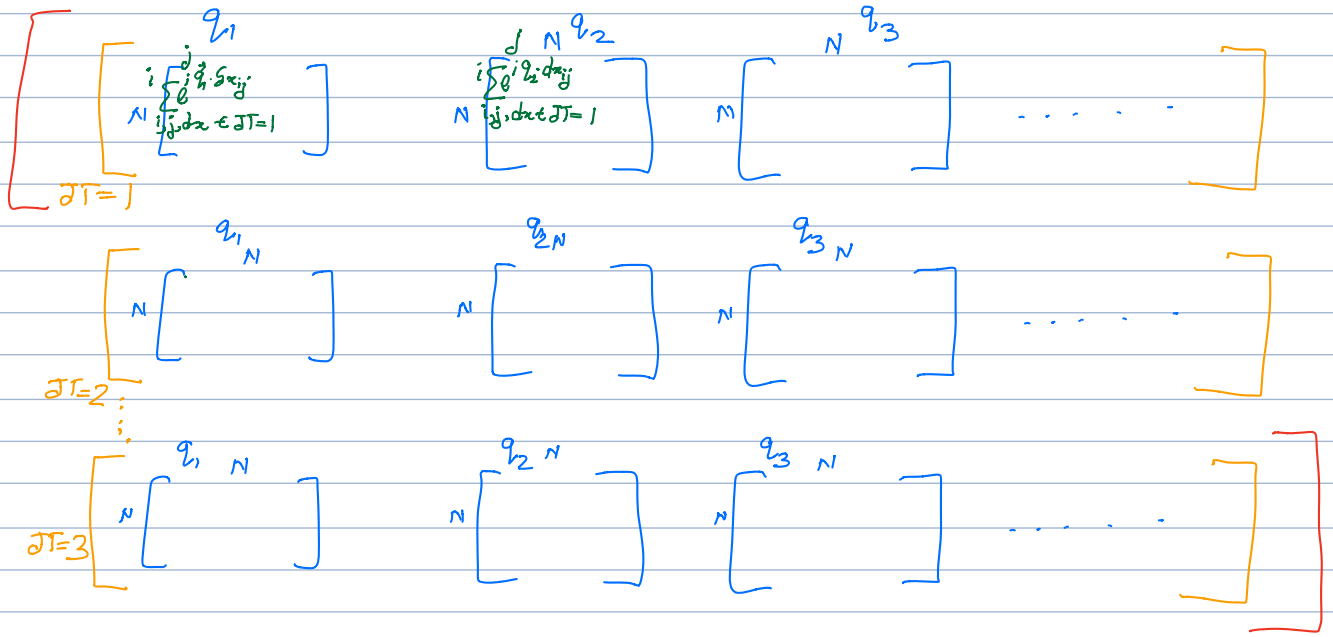
Next → Form the Fourier Transform Matrix.

```
self.omega_qij = np.tensordot(self.symmrates, self.FTjumps, axis=(0,0))
self.omega_qij[:] += self.escape
```

[symmrate_type1 , symmrate_type2 , symmrate_type3, ...]

```
self.FTJumps = zeros((lenGjumpnetwork), Nkpt, N, N), dtype=complex)
for J, jumplist in enumerate(jumpnetwork):
    for (i,j), dx in jumplist:
        FTjumps[J, :, i, j] += exp(1j * np.dot(kpts, dx))
        STjumps[i, J] += 1.
```

tensordot, axis=(0,0) ↦ Recall.



Tensordot :- np.tensordot(a, b, axes)

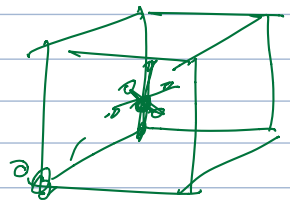
→ given two tensors (arrays with dimension greater than or equal to one),

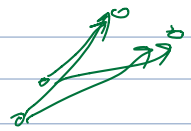
and an object 'axes' (which may be array-like), that specifies (a_axes, b_axes) sum the products of a's and b's elements over the axes specified by a_axes and b_axes.

np.tensordot(a, b, axes) :

three common use cases :-

* axes = 0 → $a \otimes b$ $\quad a \otimes b = [A_{ij} = a_i b_j]$
* axes = 1 → $a \cdot b$
* axes = 2 → contraction (?) → tensor double contraction.





db10 = 110

db.R = 000

db2.R = 111

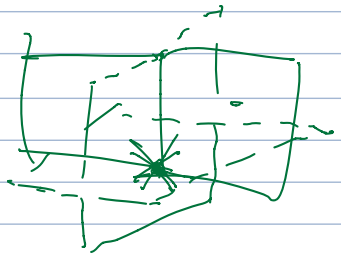
4 x 2 x 3 = 24

no. of 110 type :-

xy - 2

yz - 2 } to → 6

xz - 2



each has 24 escapes to [111]

24 x 6 = 144 escapes to [111]

```
ProbVsqrt = np.array([np.sqrt(ProbV[self.kin2vacancy[starindex]])
                      for starindex in self.vstar2kin])
```

self.kin2vacancy, self.vstar2kin.

```
self.kin2vacancy = [self.invmap[self.kinetic.states[s[0]].j] for s
                    in self.kinetic.stars]
```