
NuLattice

Release 1.0

M. Rothman, B. Johnson-Toth, G. Hagen, M. Heinz, T. Papenbrock

Sep 10, 2025

CONTENTS:

1	Getting Started	1
1.1	Installation	1
1.2	Usage	1
2	API	3
2.1	General Lattice	3
2.2	Full Configuration Interaction	8
2.3	Hartree Fock	17
2.4	Coupled Cluster	21
2.5	IMSRG	38
	Python Module Index	51
	Index	53

GETTING STARTED

1.1 Installation

Use the requirements.txt file to get the necessary packages using the following command:

```
pip install -r requirements.txt
```

If you want to build the documentation, install sphinx using the following command:

```
pip install sphinx
```

1.2 Usage

Examples for CCSD, Hartree Fock, IMSRG, and FCI can be found in the examples folder. These illustrate how to use the respective methods along with the lattice.

Pre-made hole states can be found in references.py.

2.1 General Lattice

<i>lattice</i>	This module provides functions to define the 3D lattice
<i>constants</i>	Provides useful constants to be used in the rest of the code
<i>references</i>	Provides references states to be used by the lattice

2.1.1 lattice

This module provides functions to define the 3D lattice

Functions

<i>NNNcontact</i> (v3NF, lattice, myL[, spin, isospin])	computes matrix elements for three-body onsite contact
<i>Tkin</i> (lattice, myL[, spin, isospin])	computes 1-body kinetic energy matrix elements.
<i>contacts</i> (vT1, vS1, lattice, myL[, spin, isospin])	computes matrix elements for 2-body onsite contacts
<i>get_lattice</i> (myL)	builds a 3D lattice
<i>get_sp_basis</i> (myL[, spin, isospin])	builds a 3D lattice for nucleons with spin isospin degrees of freedom
<i>left</i> (site, myL)	moves a site to the left in 1D, respecting periodic boundary conditions
<i>makeState</i> (x, y, z, tz, sz)	Takes position in x, y, and z on the lattice as well as the spin and isospin and returns a state
<i>p_x</i> (lattice, myL[, spin, isospin])	computes matrix elements for 1-body momentum operator p_x .
<i>p_y</i> (lattice, myL[, spin, isospin])	computes matrix elements for 1-body momentum operator p_y .
<i>p_z</i> (lattice, myL[, spin, isospin])	computes matrix elements for 1-body momentum operator p_z .
<i>phys_unit</i> (a_lat)	returns the energy unit from basic units
<i>right</i> (site, myL)	moves a site to the right in 1D, respecting periodic boundary conditions
<i>site2index</i> (site, myL)	given a site list [i,j,k] this function returns the index of that state in the list returned by <i>get_lattice</i>
<i>state2index</i> (state, myL[, spin, isospin])	given a state list [i,j,k,tz,sz] this function returns the index of that state in the list returned by <i>get_sp_basis</i>

continues on next page

Table 2 – continued from previous page

<code>states2PHSpace(holeList, myL)</code>	Takes a list of hole states and returns the hole and particle spaces
--	--

`lattice.phys_unit(a_lat)`

returns the energy unit from basic units

Parameters

a_lat (*float*) – lattice spacing in fm

Returns

factor to scale the lattice units to energy units

Return type

float

`lattice.get_sp_basis(myL, spin=2, isospin=2)`

builds a 3D lattice for nucleons with spin isospin degrees of freedom

Parameters

- **myL** (*int*) – number of lattice sites in each direction
- **spin** (*int*) – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returns

List of integer list [i,j,k,tz,sz] where lattice sites are labelled by i, j, k (from 0 to myL-1) in direction 1, 2, 3; tz=0, 1 and sz=0,1 correspond to isospin tz-1/2 and spin sz-1/2, respectively

Return type

list[(int, int, int, int, int)]

`lattice.state2index(state, myL, spin=2, isospin=2)`

given a state list [i,j,k,tz,sz] this function returns the index of that state in the list returned by get_sp_basis

Parameters

- **state** (*list[(int, int, int, int, int)]*) – the list [i,j,k,tz,sz]
- **myL** (*int*) – number of lattice sites in each direction
- **spin** (*int*) – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returns

index as an integer

Return type

int

`lattice.get_lattice(myL)`

builds a 3D lattice

Parameters

myL (*int*) – number of lattice sites in each direction

Returns

List of integer lists [i,j,k] of lattice sites are labelled by i, j, k (from 0 to myL-1) in direction 1, 2, 3

Return type

list[(int, int, int)]

`lattice.site2index(site, myL)`given a site list [i,j,k] this function returns the index of that state in the list returned by `get_lattice`**Parameters**

- **state** (*list[(int, int, int)]*) – the list [i,j,k]
- **myL** (*int*) – number of lattice sites in each direction

Returns

index as an integer

Return type

int

`lattice.right(site, myL)`

moves a site to the right in 1D, respecting periodic boundary conditions

Parameters

- **site** (*int*) – integer location of the site
- **myL** (*int*) – number of lattice sites in each direction

Returns

index of site one to the right of site with index site

Return type

int

`lattice.left(site, myL)`

moves a site to the left in 1D, respecting periodic boundary conditions

Parameters

- **site** (*int*) – integer location of the site
- **myL** (*int*) – number of lattice sites in each direction

Returns

index of site one to the left of site with index site

Return type

int

`lattice.Tkin(lattice, myL, spin=2, isospin=2)`

computes 1-body kinetic energy matrix elements. Really: the negative dimensionless laplacian

Parameters

- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** (*int*) – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of tuples [i, j, value] where i and j are indices in the single-particle basis, and value is the value of the matrix element T_{ij}

Return type

list[(int, int, float)]

`lattice.contacts(vT1, vS1, lattice, myL, spin=2, isospin=2)`

computes matrix elements for 2-body onsite contacts

Parameters

- **vT1** (*float*) – strength of T=1 coupling
- **vS1** (*float*) – strength of S=1 coupling
- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** (*int*) – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of lists [i, j, k, l, value] where i, j and k, l are indices of two particles in the single-particle basis, and value is the value of the matrix element $\langle ij||kl \rangle$. All elements have $i < j$ and $k < l$ **Return type**

list[(int, int, int, int, float)]

`lattice.NNNcontact(v3NF, lattice, myL, spin=2, isospin=2)`

computes matrix elements for three-body onsite contact

Parameters

- **v3NF** (*float*) – strength of the 3 nucleon force
- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of tuples [i1, i2, i3, j1, j2, j3, value] where i1, i2, i3 and j1, j2, j3 are indices of three particles in the single-particle basis, and value is one (unit strength) for the matrix element $\langle i1\ i2\ i3||j1\ j2\ j3 \rangle$. All elements have $i1 < i2 < i3$ and $j1 < j2 < j3$ **Return type**

list[(int, int, int, int, int, int, float)]

`lattice.p_x(lattice, myL, spin=2, isospin=2)`computes matrix elements for 1-body momentum operator `p_x`. Really: $-i$ times d_x **Parameters**

- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of tuples [i, j, value] where i and j are indices in the single-particle basis, and value is the value of the matrix element T_{ij}

Return type

list[(int, int, float)]

`lattice.p_y(lattice, myL, spin=2, isospin=2)`computes matrix elements for 1-body momentum operator p_y . Really: $-i$ times d_y **Parameters**

- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of tuples [i, j, value] where i and j are indices in the single-particle basis, and value is the value of the matrix element T_{ij} **Return type**

list[(int, int, float)]

`lattice.p_z(lattice, myL, spin=2, isospin=2)`computes matrix elements for 1-body momentum operator p_z . Really: $-i$ times d_z **Parameters**

- **lattice** (*list[(int, int, int)]*) – list of lattice sites returned by `get_lattice`
- **myL** (*int*) – number of lattice sites in each direction
- **spin** – Optional; number of spin degrees of freedom
- **isospin** (*int*) – Optional; number of isospin degrees of freedom

Returnslist of tuples [i, j, value] where i and j are indices in the single-particle basis, and value is the value of the matrix element T_{ij} **Return type**

list[(int, int, float)]

`lattice.states2PHSpace(holeList, myL)`

Takes a list of hole states and returns the hole and particle spaces

Parameters

- **holeList** (*list[(int, int, int, int, int)]*) – list of holes and their sites as [i, j, k, tz, sz]
- **myL** (*int*) – number of lattice sites in each direction

Returns

hole and particle space

Return type

tuple(int), tuple(int)

`lattice.makeState(x, y, z, tz, sz)`

Takes position in x, y, and z on the lattice as well as the spin and isospin and returns a state

Parameters

- **x** (*int*) – x position in lattice

- **y** (*int*) – y position in lattice
- **z** (*int*) – z position in lattice
- **tz** (0.5 / -0.5) – isospin
- **sz** (0.5 / -0.5) – spin

Returns

a particle state on the lattice as a list

Return type

list[(int, int, int, int, int)]

2.1.2 constants

Provides useful constants to be used in the rest of the code

2.1.3 references

Provides references states to be used by the lattice

Functions

<code>reference_to_holes</code> (<i>ref</i> , <i>basis</i>)	given a reference state, and a lattice basis, this function returns the corresponding holes as a tuple
---	--

`references.reference_to_holes`(*ref*, *basis*)

given a reference state, and a lattice basis, this function returns the corresponding holes as a tuple

Parameters

- **ref** (`list[list[int, int, int, int, int]]`) – reference state as list of states [lx, ly, lz, tz, sz] where the first three integers lx, ly, lz are the lattice site, and the last two integers are the isospin and spin (with values 0, 1 for -1/2, 1/2)
- **basis** (`list[list[int, int, int, int, int]]`) – list of basis states in the lattice

Returns

tuple of A integers that are the indices of the hole states

Return type

tuple(int, int, ...)

2.2 Full Configuration Interaction

<code>FCI.few_body_diagonalization</code>	module provides functions to build a many-body basis and to construct Hamiltonian matrices stored in compressed storage row csr format.
---	---

2.2.1 FCI.few_body_diagonalization

module provides functions to build a many-body basis and to construct Hamiltonian matrices stored in compressed storage row csr format.

Functions

<code>add_2body_ops</code> (ops, my_basis[, weights])	adds lists of sparse 2-body operators into a single sparse list
<code>add_3body_ops</code> (ops, my_basis[, weights])	adds lists of sparse 3-body operators into a single sparse list
<code>csr_matrix_tolist_2body</code> (op_csr, lookup2b)	converts a scipy.sparse.csr_matrix into a list of elements [p,q,r,s,value]
<code>csr_matrix_tolist_3body</code> (op_csr, lookup3b)	converts a scipy.sparse.csr_matrix into a list of elements [p,q,r,s,u,v,value]
<code>fill_1b_op_in_2b_basis</code> (lookup, operator, nstat)	for a 2-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator
<code>fill_1b_op_in_3b_basis</code> (lookup, operator, nstat)	for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator
<code>fill_1b_op_in_4b_basis</code> (lookup, operator, nstat)	for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator
<code>fill_2b_op_in_2b_basis</code> (lookup, operator)	for a 2-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator
<code>fill_2b_op_in_3b_basis</code> (lookup, operator, nstat)	for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator
<code>fill_2b_op_in_4b_basis</code> (lookup, operator, nstat)	for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator
<code>fill_3b_op_in_3b_basis</code> (lookup, operator)	for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 3-body operator
<code>fill_3b_op_in_4b_basis</code> (lookup, operator, nstat)	for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 3-body operator
<code>fill_shift_op</code> (direc, lookup, myL[, spin, ...])	for a system defined via the lookup table this function returns a list of matrix elements, a list of row indices, and a list of column indices of the shift operator that moves the state by a single lattice unit into direction
<code>get_csr_1b_op_in_2b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 1-body operator into a 2-body basis
<code>get_csr_1b_op_in_3b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 1-body operator into a 3-body basis
<code>get_csr_1b_op_in_4b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 1-body operator into a 4-body basis
<code>get_csr_2b_op_in_2b_basis</code> (lookup, operator)	returns a compressed sparse row 'csr' matrix of a 2-body operator into a 2-body basis
<code>get_csr_2b_op_in_3b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 2-body operator into a 3-body basis
<code>get_csr_2b_op_in_4b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 2-body operator into a 4-body basis
<code>get_csr_3b_op_in_3b_basis</code> (lookup, operator)	returns a compressed sparse row 'csr' matrix of a 3-body operator into a 3-body basis

continues on next page

Table 5 – continued from previous page

<code>get_csr_3b_op_in_4b_basis</code> (lookup, operator, ...)	returns a compressed sparse row 'csr' matrix of a 3-body operator into a 4-body basis
<code>get_csr_matrix_scalar_op</code> (lookup, operator, ...)	returns a scalar operator as a CSR matrix using the lookup dictionary.
<code>get_many_body_states</code> (basis, num_part[, ...])	returns dictionary of many-body states
<code>get_shift_op</code> (direc, lookup, myL[, spin, isospin])	returns a compressed sparse row 'csr' matrix of the shift operator into a few-body basis
<code>num_permutations</code> (my_list)	returns the number of permutations needed to bring a list into order

`FCI.few_body_diagonalization.get_many_body_states(basis, num_part, total_tz=None, total_sz=None)`

returns dictionary of many-body states

Parameters

- **basis** (`list[list[int, int, int, int, int], [...]]`) – the single-particle basis
- **num_part** (`int`) – number of fermions
- **total_tz** (`int`) – total z-component of isospin (twice its value)
- **total_sz** (`int`) – total z-component of spin (twice its value)

Returns

a dictionary where keys are tuples of single-particle states and values are the indices of that many-body state; this serves as a lookup table.

Return type

`dict(tuple(int, int, int, ...): int)`

`FCI.few_body_diagonalization.get_csr_matrix_scalar_op(lookup, operator, num_sp_stat)`

returns a scalar operator as a CSR matrix using the lookup dictionary. The few-body basis can only have A=2, 3, or 4 particles, and the rank of the operator can only be 1, 2, or 3.

Parameters

- **lookup** (`dict(tuple(int, int, ...): int)`) – dictionary of A-body states
- **operator** (`list[list[int, int, int, int, float], [...]]`) – list of matrix elements of the few-body operator
- **num_sp_stat** (`int`) – number of single-particle states

Returns

csr matrix of the operator

Return type

`scipy.sparse.csr_matrix`

`FCI.few_body_diagonalization.fill_1b_op_in_2b_basis(lookup, operator, nstat)`

for a 2-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator

Parameters

- **lookup** (`dict(tuple(int, int): int)`) – dictionary of two-body states
- **operator** (`list[list[int, int, float]]`) – one-body operator as list of [row, col, value]
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**fill_2b_op_in_2b_basis**(lookup, operator)

for a 2-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator

Parameters

- **lookup** (*dict(tuple(int, int): int)*) – dictionary of two-body states
- **operator** (*list[list[int, int, int, int, float]]*) – two-body operator as list of [p, q, r, s, value] where p, q, r, s are one-body states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**fill_1b_op_in_3b_basis**(lookup, operator, nstat)

for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator

Parameters

- **lookup** (*dict(tuple(int, int, int): int)*) – dictionary of three-body states
- **operator** (*list[list[int, int, float]]*) – one-body operator as list of [p, q, value] where p, q are one-body states
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**fill_2b_op_in_3b_basis**(lookup, operator, nstat)

for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator

Parameters

- **lookup** (*dict(tuple(int, int, int): int)*) – dictionary of three-body states where keys are tuples (p q r) of one-body states and values are the index of the corresponding three-body basis state
- **operator** (*list[list[int, int, int, int, float]]*) – two-body operator as list of [p, q, r, s, value] where p, q, r, s are one-body states
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.fill_3b_op_in_3b_basis(*lookup, operator*)

for a 3-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 3-body operator

Parameters

- **lookup** (*dict(tuple(int,int,int): int)*) – dictionary of three-body states where keys are tuples (p q r) of one-body states and values are the index of the corresponding three-body basis state
- **operator** (*list[list[int,int,int,int,int,int,float]]*) – three-body operator as list of [p, q, r, s, u, v, value] where p, q, r, s, u, v are one-body states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.fill_1b_op_in_4b_basis(*lookup, operator, nstat*)

for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 1-body operator

Parameters

- **lookup** (*dict(tuple(int,int,int,int): int)*) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (*list[list[int,int,float]]*) – one-body operator as list of [p, q, value] where p, q are one-body states
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.fill_2b_op_in_4b_basis(*lookup, operator, nstat*)

for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 2-body operator

Parameters

- **lookup** (*dict(tuple(int,int,int,int): int)*) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (*list[list[int,int,int,int,float]]*) – two-body operator as list of [p, q, r, s, value] where p, q, r, s are one-body states
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**fill_3b_op_in_4b_basis**(lookup, operator, nstat)

for a 4-body system this function returns a list of matrix elements, a list of row indices, and a list of column indices of the 3-body operator

Parameters

- **lookup** (*dict(tuple(int,int,int,int): int)*) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (*list[list[int,int,int,int,int,int,float]]*) – three-body operator as list of [p, q, r, s, u, v, value] where p, q, r, s, u, v are one-body states
- **nstat** – number of single-particle states

Returns

operator matrix elements as three lists op_dat, op_row, op_col

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**num_permutations**(my_list)

returns the number of permutations needed to bring a list into order

Parameters

mylist (*list[int]*) – a list of integers

Returns

number of permutations needed to bring a list into order

Return type

int

FCI.few_body_diagonalization.**fill_shift_op**(direc, lookup, myL, spin=2, isospin=2)

for a system defined via the lookup table this function returns a list of matrix elements, a list of row indices, and a list of column indices of the shift operator that moves the state by a single lattice unit into direction

Parameters

- **direc** (*int*) – direction of the shift, must be 1, 2, or 3
- **lookup** (*dict(tuple(int,int,...): int)*) – dictionary of few-body states where keys are tuples of one-body states and values are the index of the corresponding few-body basis state
- **myL** (*int*) – number of lattice sites in each dimension
- **spin** (*int*) – number of spin components
- **isospin** (*int*) – number of isospin components

Returns

a list of matrix elements, a list of row indices, and a list of column indices

Return type

list[float], list[int], list[int]

FCI.few_body_diagonalization.**get_csr_1b_op_in_2b_basis**(lookup, operator, nstat)

returns a compressed sparse row ‘csr’ matrix of a 1-body operator into a 2-body basis

Parameters

- **lookup** (*dict(tuple(int, int): int)*) – dictionary of two-body states where keys are tuples (p q) of one-body states and values are the index of the corresponding two-body basis state
- **operator** (*list[list[int, int, float]]*) – one-body operator as list of [p, q, value] where p, q, are one-body states
- **nstast** – number of single-particle states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.**get_csr_2b_op_in_2b_basis**(*lookup, operator*)

returns a compressed sparse row ‘csr’ matrix of a 2-body operator into a 2-body basis

Parameters

- **lookup** (*dict(tuple(int, int): int)*) – dictionary of two-body states where keys are tuples (p q) of one-body states and values are the index of the corresponding two-body basis state
- **operator** (*list[list[int, int, int, int, float]]*) – one-body operator as list of [p, q, r, s, value] where p, q, r, s are one-body states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.**get_csr_1b_op_in_3b_basis**(*lookup, operator, nstat*)

returns a compressed sparse row ‘csr’ matrix of a 1-body operator into a 3-body basis

Parameters

- **lookup** (*dict(tuple(int, int, int): int)*) – dictionary of three-body states where keys are tuples (p q r) of one-body states and values are the index of the corresponding three-body basis state
- **operator** (*list[list[int, int, float]]*) – one-body operator as list of [p, q, value] where p, q, are one-body states
- **nstast** – number of single-particle states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.**get_csr_2b_op_in_3b_basis**(*lookup, operator, nstat*)

returns a compressed sparse row ‘csr’ matrix of a 2-body operator into a 3-body basis

Parameters

- **lookup** (*dict(tuple(int, int, int): int)*) – dictionary of three-body states where keys are tuples (p q r) of one-body states and values are the index of the corresponding three-body basis state
- **operator** (*list[list[int, int, int, int, float]]*) – two-body operator as list of [p,q,r,s,value] where p,q,r,s are one-body states

- **nstast** – number of single-particle states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.get_csr_3b_op_in_3b_basis(*lookup*, *operator*)

returns a compressed sparse row ‘csr’ matrix of a 3-body operator into a 3-body basis

Parameters

- **lookup** (*dict(tuple(int, int, int): int)*) – dictionary of three-body states where keys are tuples (p q r) of one-body states and values are the index of the corresponding two-body basis state
- **operator** (*list[list[int, int, int, int, int, int, float]]*) – one-body operator as list of [p,q,r,s,u,v, value] where p,q,r,s,u,v are one-body states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.get_csr_1b_op_in_4b_basis(*lookup*, *operator*, *nstat*)

returns a compressed sparse row ‘csr’ matrix of a 1-body operator into a 4-body basis

Parameters

- **lookup** (*dict(tuple(int, int, int, int): int)*) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (*list[list[int, int, float]]*) – one-body operator as list of [p,q,value] where p,q are one-body states
- **nstast** – number of single-particle states

Returns

csr matrix of the operator

Return type

scipy.sparse.csr_matrix

FCI.few_body_diagonalization.get_csr_2b_op_in_4b_basis(*lookup*, *operator*, *nstat*)

returns a compressed sparse row ‘csr’ matrix of a 2-body operator into a 4-body basis

Parameters

- **lookup** (*dict(tuple(int, int, int, int): int)*) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (*list[list[int, int, int, int, float]]*) – two-body operator as list of [p,q,r,s,value] where p,q,r,s are one-body states
- **nstast** – number of single-particle states

Returns

csr matrix of the operator

Return type`scipy.sparse.csr_matrix``FCI.few_body_diagonalization.get_csr_3b_op_in_4b_basis(lookup, operator, nstat)`

returns a compressed sparse row 'csr' matrix of a 3-body operator into a 4-body basis

Parameters

- **lookup** (`dict(tuple(int, int, int, int): int)`) – dictionary of four-body states where keys are tuples (p q r s) of one-body states and values are the index of the corresponding four-body basis state
- **operator** (`list[list[int, int, int, int, int, int, float]]`) – two-body operator as list of [p,q,r,s,u,v,value] where p,q,r,s,u,v are one-body states
- **nstat** – number of single-particle states

Returns

csr matrix of the operator

Return type`scipy.sparse.csr_matrix``FCI.few_body_diagonalization.get_shift_op(direc, lookup, myL, spin=2, isospin=2)`

returns a compressed sparse row 'csr' matrix of the shift operator into a few-body basis

Parameters

- **direc** (`int`) – direction of the shift, must be 1, 2, or 3
- **lookup** (`dict(tuple(int, int, ...): int)`) – dictionary of few-body states where keys are tuples of one-body states and values are the index of the corresponding few-body basis state
- **myL** (`int`) – number of lattice sites in each dimension
- **spin** (`int`) – number of spin components
- **isospin** (`int`) – number of isospin components

Returns

csr matrix of the operator

Return type`scipy.sparse.csr_matrix``FCI.few_body_diagonalization.csr_matrix_tolist_2body(op_csr, lookup2b)`converts a `scipy.sparse.csr_matrix` into a list of elements [p,q,r,s,value]**Parameters**

- **op_csr** (`csr_matrix from scipy.sparse`) – `csr_matrix` from `scipy.sparse`
- **lookup2b** (`dictionary with entries {(int, int): int}`) – dictionary where keys are two-body state tuples (p,q) and values are indices

Returns

list [[p,q,r,s,value],] of two-body matrix elements

Return type

list with elements [int,int,int,int, float]

FCI.few_body_diagonalization.add_2body_ops(*ops*, *my_basis*, *weights=None*)

adds lists of sparse 2-body operators into a single sparse list

Parameters

- **ops** (*[[[int,int,int,int, float], ...], ...]*) – list of lists *[[p,q,r,s, val], ...]* of two-body operators
- **my_basis** (*[int,int,int,int,int]*) – list with elements *[x,y,z,tauz,sz]* that are single-particle states
- **weights** (*[float,float,...]*) – array of weights each operator in ops will be multiplied with

Returns

list *[[p,q,r,s, val], ...]* of a single two-body operator

Return type

[[int,int,int,int, float], ...]

FCI.few_body_diagonalization.csr_matrix_tolist_3body(*op_csr*, *lookup3b*)

converts a scipy.sparse.csr_matrix into a list of elements *[p,q,r,s,u,v,value]*

Parameters

- **op_csr** (*csr_matrix from scipy.sparse*) – csr_matrix from scipy.sparse
- **lookup3b** (*dictionary with entries {(int,int,int): int}*) – dictionary where keys are two-body state tuples (p,q,r) and values are indices

Returns

list *[[p,q,r,s,u,v, value], ...]* of two-body matrix elements

Return type

list with elements *[int,int,int, int,int,int, float]*

FCI.few_body_diagonalization.add_3body_ops(*ops*, *my_basis*, *weights=None*)

adds lists of sparse 3-body operators into a single sparse list

Parameters

- **ops** (*[[[[int,int,int,int,int,int, float], ...], ...]*) – list of lists *[[p,q,r,s,u,v, val], ...]* of three-body operators
- **my_basis** (*[int,int,int,int,int]*) – list with elements *[x,y,z,tauz,sz]* that are single-particle states
- **weights** (*[float,float,...]*) – array of weights each operator in ops will be multiplied with

Returns

list *[[p,q,r,s,u,v, val], ...]* of a single three-body operator

Return type

[[int,int,int,int,int,int, float], ...]

2.3 Hartree Fock

HF.hartree_fock

functions to perform a Hartree-Fock computation on the lattice

2.3.1 HF.hartree_fock

functions to perform a Hartree-Fock computation on the lattice

Functions

<i>HF_energy</i> (op1, op2, op3, dens)	Computes the Hartree-Fock energy for a given density dens and Hamiltonian consisting of one-body term op1, two-body term op2, and three-body term op3
<i>HF_iter</i> (op1, op2, op3, dens[, mix])	Performs one iteration of the Hartree-Fock procedure
<i>contract_2nf</i> (v2, dens)	takes list of two-body matrix elements and contracts them with the density to get a one-body operator
<i>contract_3nf</i> (w3, dens)	takes list of three-body matrix elements and contracts them with the density to get a one-body operator
<i>get_1body_matrix</i> (myTkin, nstat)	takes the list of one-body matrix elements and turns it into a square matrix
<i>init_density</i> (nstat, hole)	creates a density matrix of dimension nstat x nstat given the hole information
<i>make_HF_ham</i> (op1, op2, op3, dens)	takes Hamiltonian consisting of one-body operator op1, two-body operator op2, and three-body operator op3, and the density matrix and returns the Hartree-Fock Hamiltonian.
<i>solve_HF</i> (op1, op2, op3, dens[, mix, eps, ...])	Solve the Hartree-Fock problem

HF.hartree_fock.get_1body_matrix(myTkin, nstat)

takes the list of one-body matrix elements and turns it into a square matrix

Parameters

- **nstat** (*int*) – dimension of matrix, i.e. the number of 1-body states
- **myTkin** (*list[list[int,int, float]]*) – list of one-body matrix elements
[[p1,q1,value1], [p2,q2,value2], ...]

Returns

nstat x nstat matrix of the list of matrix elements

Return type

numpy.array((:,:), dtype=float)

HF.hartree_fock.contract_2nf(v2, dens)

takes list of two-body matrix elements and contracts them with the density to get a one-body operator

Parameters

- **v2** (*list[list[int,int,int,int, float]]*) – list of two-body matrix elements
[p,q,r,s,value]
- **dens** (*numpy.array((:,:), dtype=float)*) – square density matrix

Returns

one-body operator of the same shape as the density matrix dens

Return type

numpy.array((:,:), dtype=float)

`HF.hartree_fock.contract_3nf(w3, dens)`

takes list of three-body matrix elements and contracts them with the density to get a one-body operator

Parameters

- **w3** (*list[list[int, int, int, int, int, int, float]]*) – list of two-body matrix elements [p,q,r,s,value]
- **dens** (*numpy.array((:,:), dtype=float)*) – square density matrix

Returns

one-body operator of the same shape as the density matrix dens

Return type

numpy.array((:,:), dtype=float)

`HF.hartree_fock.make_HF_ham(op1, op2, op3, dens)`

takes Hamiltonian consisting of one-body operator op1, two-body operator op2, and three-body operator op3, and the density matrix and returns the Hartree-Fock Hamiltonian.

Parameters

- **op1** (*list[list[int, int, float]]*) – list of one-body matrix elements
- **op2** (*list[list[int, int, int, int, float]]*) – list of two-body matrix elements
- **op3** (*list[list[int, int, int, int, int, int, float]]*) – list of three-body matrix elements
- **dens** (*numpy.array((:,:), dtype=float)*) – density matrix (same shape as op1)

Returns

matrix in the shape of op1 and dens that is the Hartree-Fock Hamiltonian

Return type

numpy.array((:,:), dtype=float)

`HF.hartree_fock.init_density(nstat, hole)`

creates a density matrix of dimension nstat x nstat given the hole information

Parameters

- **nstat** (*int*) – dimension of single-particle basis
- **hole** (*tuple(int, int, ...)*) – tuple of occupied single-particle states, as numbers from 0 ... A-1

Returns

density matrix where hole states are occupied (1) and all others not (0)

Return type

numpy.array((nstat,nstat), dtype = float)

`HF.hartree_fock.HF_energy(op1, op2, op3, dens)`

Computes the Hartree-Fock energy for a given density dens and Hamiltonian consisting of one-body term op1, two-body term op2, and three-body term op3

Parameters

- **op1** (*list[list[int, int, float]]*) – list of one-body matrix elements
- **op2** (*list[list[int, int, int, int, float]]*) – list of two-body matrix elements

- **op3** (*list[list[int,int,int,int,int,int, float]]*) – list of three-body matrix elements
- **dens** (*numpy.array(:, :), dtype=float*) – density matrix (same shape as op1)

Returns

Hartree-Fock energy

Return type

float

`HF.hartree_fock.HF_iter(op1, op2, op3, dens, mix=0.5)`

Performs one iteration of the Hartree-Fock procedure

Parameters

- **op1** (*list[list[int,int, float]]*) – list of one-body matrix elements
- **op2** (*list[list[int,int,int,,int, float]]*) – list of two-body matrix elements
- **op3** (*list[list[int,int,int,int,int,int, float]]*) – list of three-body matrix elements
- **dens** (*numpy.array(:, :), dtype=float*) – density matrix (same shape as op1)
- **mix** (*float*) – returned density matrix is $\text{mix} \cdot \text{new_density} + (1 - \text{mix}) \cdot \text{old_density}$

Returns

energy, density, vecs as the current HF energy, current density matrix, and orthogonal transformation matrix that diagonalizes the HF Hamiltonian

Return type

float, *numpy.array(:, :), dtype=float*, *numpy.array(:, :), dtype=float*

`HF.hartree_fock.solve_HF(op1, op2, op3, dens, mix=0.5, eps=1e-08, max_iter=100, verbose=False)`

Solve the Hartree-Fock problem

Parameters

- **op1** (*list[list[int,int, float]]*) – list of one-body matrix elements
- **op2** (*list[list[int,int,int,,int, float]]*) – list of two-body matrix elements
- **op3** (*list[list[int,int,int,int,int,int, float]]*) – list of three-body matrix elements
- **dens** (*numpy.array(:, :), dtype=float*) – density matrix (same shape as op1)
- **mix** (*float*) – parameter used in the mixing: $\text{mix} \cdot \text{new_density} + (1 - \text{mix}) \cdot \text{old_density}$
- **eps** (*float*) – convergence of energy
- **max_iter** (*float*) – maximum number of HF iterations

Returns

energy, orthogonal transformation matrix that diagonalizes the HF Hamiltonian (the first A columns are occupied), converged

Return type

float, *numpy.array(:, :), dtype=float*, boolean

2.4 Coupled Cluster

<code>CCM.coupled_cluster</code>	Provides functions for setting up the coupled cluster equations on the lattice and solving for them
<code>CCM.three_body_utils</code>	Provides functions for all of the 3 body interactions
<code>CCM.ccDgrams</code>	Provides functions for all of the coupled cluster diagrams

2.4.1 CCM.coupled_cluster

Provides functions for setting up the coupled cluster equations on the lattice and solving for them

Functions

<code>ccsd_energy(f_ph, v_pphh, t2, t1)</code>	computes ccsd correlation energy Note: Technically, this would need <code>v_hhpp</code> but this is of course the transpose of <code>v_pphh</code> ; Likewise, <code>f_hp</code> is required, and this is just the transpose of <code>f_ph</code> which we have
<code>ccsd_solver(fock_mats, two_body_int[, ...])</code>	Solves for the correlation energy of a system using the CCSD equations.
<code>get_all_interactions(part, hole, mycontact)</code>	This routine takes the relatively small number of two-body matrix elements in <code>mycontact</code> and sorts them into the four-indexed interaction tensors.
<code>get_fock_matrices(part, hole, myTkin, ...)</code>	constructs fock matrices from one-body interaction and rank-4 tensors of two-body force
<code>get_norm_ord_int(thisL, holes, vT1, vS1[, ...])</code>	Takes all the necessary parameters to generate the fock matrices, two and three body interactions to perform CCM
<code>get_norm_ordered_ham(thisL, holes, myTkin, ...)</code>	Takes all the necessary parameters to generate the fock matrices, two and three body interactions to perform CCM
<code>get_ref_energy(no_1b_hh, no_2b_hhhh, w_hhh_hhh)</code>	Computes the energy of the reference state from normal ordered interactions
<code>t1Init(f_ph, f_pp, f_hh, delta)</code>	Initializes t_i^a based on the perturbation theory guess
<code>t1Iter(t1, t2, f_ph, f_pp, f_hh, v_phph, ...)</code>	iterating $t1$ using the CCSD equations
<code>t2Init(f_pp, f_hh, v_pphh, delta)</code>	Initializes t_{ij}^{ab} based on perturbation theory guess
<code>t2Iter(t1, t2, f_ph, f_hh, f_pp, v_pppp, ...)</code>	iterating $t2$, factoring out terms that look like g_i^i and g_a^a

`CCM.coupled_cluster.get_fock_matrices(part, hole, myTkin, v_phph, v_phhh, v_hhhh)`
constructs fock matrices from one-body interaction and rank-4 tensors of two-body force

Parameters

- **part** (`list[int]`) – list of particle indices
- **hole** (`list[int]`) – list of hole indices
- **myTkin** (`list[(int, int, float)]`) – list of one-body matrix elements
- **v_phph** (`numpy array`) – two body interaction matrix V^{ai}_{bj}
- **v_phhh** (`numpy array`) – two body interaction matrix V^{ai}_{jk}
- **v_hhhh** (`numpy array`) – two body interaction matrix V^{ij}_{kl}

Returns

Fock matrices `f_pp`, `f_ph`, `f_hh`

Return type

numpy array, numpy array, numpy array

`CCM.coupled_cluster.get_all_interactions(part, hole, mycontact, sparse=False)`

This routine takes the relatively small number of two-body matrix elements in `mycontact` and sorts them into the four-indexed interaction tensors. It also anti-symmetrizes the latter when in and out indices run over the same set of particle/hole indices.

Parameters

- **part** (*list[int]*) – list of particle-space indices
- **hole** (*list[int]*) – list of hole-space indices
- **mycontact** (*list[(int, int, int, int, float)]*) – list of two-body matrix elements
- **sparse** (*bool*) – Optional; whether or not `v_pppp` and `v_ppph` should be stored as sparse arrays or not

Returns

Two body matrices `v_pppp`, `v_ppph`, `v_pphh`, `v_phph`, `v_phhh`, `v_hhhh` as rank-four tensors.

Return type

numpy array | sparse array, numpy array | sparse array, numpy array, numpy array, numpy array, numpy array

`CCM.coupled_cluster.ccsd_energy(f_ph, v_pphh, t2, t1)`

computes ccsd correlation energy Note: Technically, this would need `v_hhpp` but this is of course the transpose of `v_pphh`; Likewise, `f_hp` is required, and this is just the transpose of `f_ph` which we have

Parameters

- **f_ph** (*numpy array*) – Fock matrix f^a_i
- **v_pphh** (*numpy array*) – two body interaction matrix V^{ab}_{ij}
- **t1** (*numpy array*) – T^a_i from the coupled cluster equations
- **t2** (*numpy array*) – T^{ab}_{ij} from the coupled cluster equations

Returns

CCSD correlation energy

Return type

float

`CCM.coupled_cluster.get_ref_energy(no_1b_hh, no_2b_hhhh, w_hhh_hhh=None)`

Computes the energy of the reference state from normal ordered interactions

Parameters

- **no_1b_hh** (*numpy array*) – Fock matrix (including contributions from 3NF if `w_hhh_hhh` is not `None`)
- **no_2b_hhhh** (*numpy array*) – numpy array, normal-ordered TBME including contributions from 3NF if `w_3b` is not `None`
- **w_hhh_hhh** (*list[(int, int, int, int, int, int, float)]*) – list of nonzero elements of 3NF or `None` if not using

Returns

energy of reference state

Return type

float

`CCM.coupled_cluster.t1Init(f_ph, f_pp, f_hh, delta)`

Initializes t_i^a based on the perturbation theory guess

Parameters

- **f_ph** (*numpy array*) – Fock matrix f^a_i
- **f_pp** (*numpy array*) – Fock matrix f^a_b
- **f_hh** (*numpy array*) – Fock matrix f^i_j
- **delta** (*float*) – Energy gap to make sure there is no division by 0 errors

Returns

t_i^a initial guess

Return type

numpy array

`CCM.coupled_cluster.t1Iter(t1, t2, f_ph, f_pp, f_hh, v_phph, v_phhh, v_pphh, v_ppph_results, sparse=True)`

iterating $t1$ using the CCSD equations

Parameters

- **t1** (*numpy array*) – t^a_i from the previous iteration
- **t2** (*numpy array*) – t^{ab}_{ij} from the previous iteration
- **f_ph** (*numpy array*) – Fock matrix f^a_i
- **f_pp** (*numpy array*) – Fock matrix f^a_b
- **f_hh** (*numpy array*) – Fock matrix f^i_j
- **v_phph** (*numpy array*) – two body interaction matrix V^{ai}_{bj}
- **v_phhh** (*numpy array*) – two body interaction matrix V^{ai}_{jk}
- **v_pphh** (*numpy array*) – two body interaction matrix V^{ab}_{ij}
- **v_ppph_results** (*numpy array | list[numpy array]*) – two body interaction matrix V^{ab}_{ci} if *sparse* = False, results from `CCM.ccDgrams.v_ppph_dgrams()` otherwise
- **sparse** (*bool*) – whether or not v_{ppph} and v_{ppph} are stored as sparse arrays or not

Returns

updated t_i^a

Return type

numpy array

`CCM.coupled_cluster.t2Init(f_pp, f_hh, v_pphh, delta)`

Initializes t_{ij}^{ab} based on perturbation theory guess

Parameters

- **f_pp** (*numpy array*) – Fock matrix
- **f_hh** (*numpy array*) – Fock matrix
- **v_pphh** (*numpy array*) – two body interaction matrix V^{ab}_{ij}

- **delta** (*float*) – Energy gap to avoid division by 0 errors

Returns

$t^{\{ab\}}_{ij}$ based on perturbation theory guess

Return type

numpy array

`CCM.coupled_cluster.t2Iter(t1, t2, f_ph, f_hh, f_pp, v_pppp, v_phph, v_phhh, v_pphh, v_ppph_results, v_hhhh, sparse=True)`

iterating t2, factoring out terms that look like g_i^i and g_a^a

Parameters

- **t1** (*numpy array*) – t^a_i from the previous iteration
- **t2** (*numpy array*) – $t^{\{ab\}}_{ij}$ from the previous iteration
- **f_ph** (*numpy array*) – Fock matrix f_i^a
- **f_hh** (*numpy array*) – Fock matrix f_i^j
- **f_pp** (*numpy array*) – Fock matrix f_a^b
- **v_pppp** (*numpy array* | *list[(int, int, int, int, float)]*) – two body interaction matrix $V^{\{ab\}}_{cd}$; optionally stored as a sparse array
- **v_phph** (*numpy array*) – two body interaction matrix $V^{\{ai\}}_{bj}$
- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}}_{jk}$
- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{ij}$
- **v_ppph_results** (*numpy array* | *list[numpy array]*) – if not sparse, the two body interaction matrix $V^{\{ab\}}_{ci}$, but if `sparse=True`, it is the contributions to T2 from the $V^{\{ab\}}_{ci}$ diagrams
- **v_hhhh** (*numpy array*) – two body interaction matrix $V^{\{ij\}}_{kl}$
- **sparse** (*bool*) – whether or not `v_pppp` and `v_ppph` are stored as sparse arrays or not

Returns

updated t_{ij}^{ab}

Return type

numpy array

`CCM.coupled_cluster.ccsd_solver(fock_mats, two_body_int, t1initial=None, eps=1e-08, maxSteps=1000, max_diis=10, delta=0, mixing=0.5, verbose=False, sparse=True, ccs=False)`

Solves for the correlation energy of a system using the CCSD equations. DIIS credit to Daniel G. A. Smith and Lori A. Burns, and The Psi4NumPy Developers, <https://github.com/psi4/psi4numpy>

Parameters

- **fock_mats** (*list[numpy array]*) – Fock matrices
- **two_body_int** (*list[numpy array]*) – two body interaction matrices; if sparse the first two elements will be lists instead of numpy arrays
- **eps** (*float*) – Optional; max relative error
- **maxSteps** (*int*) – Optional; max number of iterations to take
- **max_diis** (*int*) – Optional; size of DIIS array. Set to 0 to not perform DIIS

- **delta** (*float*) – Optional; energy gap for first iteration to avoid division by 0 errors
- **mixing** (*float between 0 and 1*) – Optional; how much of the previous step to mix into the current one. Set to 0 if you don't want any mixing from the previous step
- **verbose** (*bool*) – Optional; whether or not to print steps as they are calculated
- **sparse** (*bool*) – Optional; whether or not `v_pppp` and `v_ppph` are stored as sparse arrays or not
- **ccs** (*bool*) – Optional; whether to perform just the ccs equations or not

Returns

correlation energy, t1, and t2

Return type

float, numpy array, numpy array

`CCM.coupled_cluster.get_norm_ord_int(thisL, holes, vT1, vS1, str_3NF=0, sparse=True)`

Takes all the necessary parameters to generate the fock matrices, two and three body interactions to perform CCM

Parameters

- **thisL** (*int*) – number of lattice sites in each direction
- **holes** (*list[list[int]]*) – list of holes in the format [x, y, z, tz+0.5, sz+0.5]. To generate these, see [lattice.makeState\(\)](#)
- **vT1** (*float*) – strength of T=1 coupling
- **vS1** (*float*) – strength of S=1 coupling
- **str_3NF** (*float*) – Optional strength of T=1 coupling
- **sparse** (*bool*) – Optional whether or not to store `v_pppp` and `v_ppph` as a sparse array or not

Returns

The reference energy and three lists, a list of the three fock matrices in the order `f_pp`, `f_ph`, `f_hh`, all of the two body interactions in the order `v_pppp`, `v_ppph`, `v_pphh`, `v_phph`, `v_phhh`, `v_hhhh`

`CCM.coupled_cluster.get_norm_ordered_ham(thisL, holes, myTkin, mycontact, my3body=None, sparse=True, NO2B=True)`

Takes all the necessary parameters to generate the fock matrices, two and three body interactions to perform CCM

Parameters

- **thisL** (*int*) – number of lattice sites in each direction
- **holes** (*list[list[int]]*) – list of holes in the format [x, y, z, tz+0.5, sz+0.5]. To generate these, see [lattice.makeState\(\)](#)
- **myTkin** (*list[list[int, int, float]]*) – list of one-body matrix elements [[p,q,T]...]
- **mycontact** (*list[list[int, int, int, int, float]]*) – list of two-body matrix elements [[p,q,r,s,V]...]
- **my3body** (*list[list[int, int, int, int, int, int, float]]*) – list of two-body matrix elements [[p,q,r,s,u,v,W]...]
- **sparse** (*bool*) – Optional whether or not to store `v_pppp` and `v_ppph` as a sparse array or not

- **NO2B** (*bool*) – whether or not to apply the normal-order two-body approximation, i.e. to return None for the three body interaction

Returns

The reference energy and three lists, a list of the three fock matrices in the order [f_pp, f_ph, f_hh], all of the two body interactions in the order [v_pppp, v_ppph, v_pphh, v_phph, v_phhh, v_hhhh], and all of the three body interactions in the order [w_ppp_pph, w_ppp_phh, w_pph_pph, w_ppp_hhh, w_pph_phh, w_pph_hhh, w_phh_phh, w_phh_hhh, w_hhh_hhh], or [] if my3body = None or NO2B = True

Return type

float, list[numpy array], list[numpy array], list[numpy array]

2.4.2 CCM.three_body_utils

Provides functions for all of the 3 body interactions

Functions

<code>get_3NF(part, hole, my3body)</code>	This routine takes the relatively small number of three-body matrix elements in mycontact and sorts them into the four-indexed interaction tensors.
<code>get_3NF_Eref(w_hhh_hhh)</code>	returns normal-ordering contributions to the reference energy
<code>get_3NF_fock(hnum, pnum, w_phh_phh, ...)</code>	gets the normal-ordering contributions of the three-body potential to the Fock matrix
<code>get_3NF_tbme(w_pph_pph, w_pph_phh, ..., ...)</code>	Finds the normal-ordering contributions of the three-body potential to the two-body matrix elements
<code>order_state(ket, lookup_p, lookup_h, i1, i2, i3)</code>	orders tuple into right order, i.e. "p" before "h" this function is used by the function get_3NF.

CCM.three_body_utils.**get_3NF**(*part, hole, my3body*)

This routine takes the relatively small number of three-body matrix elements in mycontact and sorts them into the four-indexed interaction tensors. It also anti-symmetrizes the latter whenin and out indices run over the same set of particle/hole indices. The whole thing is a bit tedious but much faster than the function load2bme

Parameters

- **part** (*list[int]*) – list of particle-space indices
- **hole** (*list[int]*) – list of hole-space indices
- **my3body** (*list[(int, int, int, int, int, int, float)]*) – list of three-body matrix elements

Returns

w_ppp_pph, w_ppp_phh, w_pph_pph, w_ppp_hhh, w_pph_phh, w_pph_hhh, w_phh_phh, w_phh_hhh as lists of nonzeros

Return type

9 list[(int, int, int, int, int, int, float)]

CCM.three_body_utils.**order_state**(*ket, lookup_p, lookup_h, i1, i2, i3*)

orders tuple into right order, i.e. "p" before "h" this function is used by the function get_3NF

Parameters

- **ket** ((*str, str, str*)) – ket to be used in finding the right particle hole index

- **lookup_p** (*dict* (*int*, *int*)) – dictionary to lookup the index of particles
- **lookup_h** (*dict* (*int*, *int*)) – dictionary to lookup the index of holes
- **i1** (*int*) – index of first particle/hole
- **i2** (*int*) – index of second particle/hole
- **i3** (*int*) – index of third particle/hole

Returns

the correctly ordered result looking like ket, the sign of the permutation that achieved this, and the list of three single-particle indices

Return type

tuple(str, str, str), float, list[(int, int, int)]

CCM.three_body_utils.get_3NF_Eref(*w_hhh_hhh*)

returns normal-ordering contributions to the reference energy

Parameters

w_hhh_hhh (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction $W^{\{ijk\}}_{\{lmn\}}$

Returns

contribution to the reference energy from the normal ordered three nucleon force

Return type

float

CCM.three_body_utils.get_3NF_fock(*hnum*, *pnum*, *w_phh_phh*, *w_phh_hhh*, *w_hhh_hhh*)

gets the normal-ordering contributions of the three-body potential to the Fock matrix

Parameters

- **hnum** (*int*) – number of hole states
- **pnum** (*int*) – number of particle states
- **w_phh_phh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{aij\}}_{\{bkl\}}$
- **w_phh_hhh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{aij\}}_{\{klm\}}$
- **w_hhh_hhh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{ijk\}}_{\{lmn\}}$

Returns

contributions to the normal ordered one body matrices

Return type

numpy array, numpy array, numpy array

CCM.three_body_utils.get_3NF_tbme(*w_pph_pph*, *w_pph_phh*, *w_pph_hhh*, *w_phh_phh*, *w_phh_hhh*, *w_hhh_hhh*, *pnum*, *hnum*, *sparse_pppp=True*, *sparse_ppph=True*)

Finds the normal-ordering contributions of the three-body potential to the two-body matrix elements

Parameters

- **w_pph_pph** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{abi\}}_{\{cdj\}}$
- **w_pph_phh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{acj\}}_{\{bkl\}}$

- **w_phh_phh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{a_{ij}\}}_{\{b_{kl}\}}$
- **w_phh_hhh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{a_{ij}\}}_{\{klm\}}$
- **w_hhh_hhh** (*list*[(*int*, *int*, *int*, *int*, *int*, *int*, *float*)] – nonzero elements of the three body interaction matrix $W^{\{ijk\}}_{\{lmn\}}$
- **hnum** (*int*) – number of hole states
- **pnum** (*int*) – number of particle states
- **sparse_pppp** (*bool*) – Optional; whether or not v_pppp should be stored as sparse or not
- **sparse_ppph** (*bool*) – Optional; whether or not v_ppph should be stored as sparse or not

Returns

contributions to the normal ordered two body matrices

Return type

numpy array, numpy array, numpy array, numpy array, numpy array, numpy array

2.4.3 CCM.ccDgrams

Provides functions for all of the coupled cluster diagrams

Functions

<i>dgram_acik_bcjk</i> (v_ppph_res, t2)	Calculates - $P(ij)P(ab)X^{\{ac\}}_{\{ij\}}t^{\{bc\}}_{\{jk\}}$
<i>dgram_akci_ck</i> (v_phph, t1)	Calculates $-V^{\{ak\}}_{\{ci\}}t^{\{c\}}_{\{k\}}$
<i>dgram_bijk_ak1</i> (v_ppph_res, t1)	Calculates $0.5 * P(ab)X^{\{bi\}}_{\{jk\}}t^{\{a\}}_{\{k\}}$
<i>dgram_bijk_ak2</i> (v_ppph_res, t1)	Calculates $0.5 * P(ij)P(ab)X^{\{bi\}}_{\{jk\}}t^{\{a\}}_{\{k\}}$
<i>dgram_bijk_bj</i> (v_phhh, t1)	Calculates $-V^{\{bi\}}_{\{jk\}}t^{\{b\}}_{\{j\}}$
<i>dgram_bkci_ak_cj</i> (v_phph, t1)	Calculates - $P(ij)P(ab)V^{\{bk\}}_{\{ci\}}t^{\{a\}}_{\{k\}}t^{\{c\}}_{\{j\}}$
<i>dgram_bkcj_acik</i> (v_phph, t2)	Calculates $-P(ij)P(ab)V^{\{bk\}}_{\{cj\}}t^{\{cb\}}_{\{ik\}}$
<i>dgram_bkij_ak</i> (v_phhh, t1)	Calculates $P(ab)V^{\{bk\}}_{\{ij\}}t^{\{a\}}_{\{k\}}$
<i>dgram_cdkl_acik_dblj</i> (v_pphh, t2)	Calculates $0.5 * P(ij)P(ab)V^{\{cd\}}_{\{kl\}}t^{\{ac\}}_{\{ik\}}t^{\{db\}}_{\{lj\}}$
<i>dgram_cdkl_ak_bl_cdi</i> (v_pphh, t1, t2)	Calculates $0.25 * P(ab)V^{\{cd\}}_{\{kl\}}t^{\{a\}}_{\{k\}}t^{\{b\}}_{\{l\}}t^{\{cd\}}_{\{ij\}}$
<i>dgram_cdkl_bdkl</i> (v_pphh, t2)	Calculates $-0.5 * V^{\{cd\}}_{\{kl\}}t^{\{bd\}}_{\{kl\}}$
<i>dgram_cdkl_cdi_abkl</i> (v_pphh, t2)	Calculates $0.25 * P(ab)V^{\{cd\}}_{\{kl\}}t^{\{cd\}}_{\{ij\}}t^{\{ab\}}_{\{kl\}}$
<i>dgram_cdkl_cdjl</i> (v_pphh, t2)	Calculates $-0.5 * V^{\{cd\}}_{\{kl\}}t^{\{cd\}}_{\{jl\}}$
<i>dgram_cdkl_ci_ak_dj_bl</i> (v_pphh, t1)	Calculates $0.25 * P(ij)P(ab)V^{\{cd\}}_{\{kl\}}t^{\{c\}}_{\{i\}}t^{\{a\}}_{\{k\}}t^{\{d\}}_{\{j\}}t^{\{b\}}_{\{l\}}$
<i>dgram_cdkl_ci_bl_adkj</i> (v_pphh, t1, t2)	Calculates $P(ij)P(ab)V^{\{cd\}}_{\{kl\}}t^{\{c\}}_{\{i\}}t^{\{b\}}_{\{l\}}t^{\{ad\}}_{\{kj\}}$
<i>dgram_cdkl_ci_dj_abkl</i> (v_pphh, t1, t2)	Calculates $0.25 * P(ij)V^{\{cd\}}_{\{kl\}}t^{\{c\}}_{\{i\}}t^{\{d\}}_{\{j\}}t^{\{ab\}}_{\{kl\}}$
<i>dgram_cdkl_ck_dali</i> (v_pphh, t1, t2)	Calculates $V^{\{cd\}}_{\{kl\}}t^{\{c\}}_{\{k\}}t^{\{da\}}_{\{li\}}$
<i>dgram_cdkl_dk_al</i> (v_pphh, t1)	Calculates $0.5 * V^{\{cd\}}_{\{kl\}}t^{\{d\}}_{\{k\}}t^{\{a\}}_{\{l\}}$
<i>dgram_cdlk_cdli</i> (v_pphh, t2)	Calculates $-0.5 * V^{\{cd\}}_{\{lk\}}t^{\{cd\}}_{\{li\}}$
<i>dgram_cdlk_cl_di</i> (v_pphh, t1)	Calculates $-0.5 * V^{\{cd\}}_{\{lk\}}t^{\{c\}}_{\{l\}}t^{\{d\}}_{\{i\}}$
<i>dgram_cdlk_cl_dj</i> (v_pphh, t1)	Calculates $-V^{\{cd\}}_{\{kl\}}t^{\{c\}}_{\{l\}}t^{\{d\}}_{\{j\}}$
<i>dgram_cdlk_dk_bl</i> (v_pphh, t1)	Calculates $-V^{\{cd\}}_{\{kl\}}t^{\{d\}}_{\{k\}}t^{\{b\}}_{\{l\}}$
<i>dgram_cikl_al_bcjk</i> (v_phhh, t1, t2)	Calculates $-P(ij)P(ab)V^{\{ci\}}_{\{kl\}}t^{\{a\}}_{\{l\}}t^{\{bc\}}_{\{jk\}}$
<i>dgram_cikl_cakl</i> (v_phhh, t2)	Calculates $-0.5 * V^{\{ci\}}_{\{kl\}}t^{\{ca\}}_{\{kl\}}$
<i>dgram_cikl_ck_ablj</i> (v_phhh, t1, t2)	Calculates $-P(ij)V^{\{ci\}}_{\{kl\}}t^{\{c\}}_{\{k\}}t^{\{ab\}}_{\{lj\}}$
<i>dgram_cjkl_ci_abkl</i> (v_phhh, t1, t2)	Calculates $0.5 * P(ij)V^{\{cj\}}_{\{kl\}}t^{\{c\}}_{\{i\}}t^{\{ab\}}_{\{kl\}}$
<i>dgram_cjkl_ci_ak_bl</i> (v_phhh, t1)	Calculates $0.5 * P(ij)P(ab)V^{\{cj\}}_{\{kl\}}t^{\{c\}}_{\{i\}}t^{\{a\}}_{\{k\}}t^{\{b\}}_{\{l\}}$

continues on next page

Table 11 – continued from previous page

<code>dgram_ck_acik(f_ph, t2)</code>	Calculates $f^{\{c\}}_{\{k\}} t^{\{ac\}}_{\{ik\}}$
<code>dgram_ck_ak(f_ph, t1)</code>	Calculates $-0.5 * f^{\{c\}}_{\{k\}} t^{\{a\}}_{\{k\}}$
<code>dgram_ck_bk(f_ph, t1)</code>	Calculates $- * f^{\{c\}}_{\{k\}} t^{\{b\}}_{\{k\}}$
<code>dgram_ck_ci(f_ph, t1)</code>	Calculates $-0.5 * f^{\{c\}}_{\{k\}} t^{\{c\}}_{\{i\}}$
<code>dgram_ck_cj(f_ph, t1)</code>	Calculates $- * f^{\{c\}}_{\{k\}} t^{\{c\}}_{\{j\}}$
<code>dgram_da_dbij(v_ppph_res, t2)</code>	Calculates $- P(ab) X^d_{at} db_{ij}$
<code>dgram_dckl_dakl(v_pphh, t2)</code>	Calculates $-0.5 * V^{\{dc\}}_{\{kl\}} t^{\{da\}}_{\{kl\}}$
<code>dgram_klij_abkl(v_hhhh, t2)</code>	Calculates $0.5 * V^{\{kl\}}_{\{ij\}} t^{\{ab\}}_{\{kl\}}$
<code>dgram_klij_ak_bl(v_hhhh, t1)</code>	Calculates $- 0.5 * P(ab) V^{\{kl\}}_{\{ij\}} t^{\{a\}}_{\{k\}} t^{\{b\}}_{\{l\}}$
<code>pAB(val)</code>	Permutator for ab
<code>pIJ(val)</code>	Permutator for ij
<code>v_ppph_dgrams(v_ppph, t1, t2)</code>	Calculates all 6 of the diagrams that use v_pppp using the fact that it is sparse
<code>v_pppp_dgrams(v_pppp, t1, t2)</code>	Calculates both of the diagrams that use v_pppp using the fact that it is sparse

CCM.ccDgrams.**v_ppph_dgrams**(*v_ppph, t1, t2*)

Calculates all 6 of the diagrams that use v_pppp using the fact that it is sparse

Parameters

- **v_ppph** (*list*[(*int, int, int, int, float*)]) – nonzeros of the two body interaction matrix $V^{\{ab\}}_{\{ck\}}$
- **t1** (*numpy array*) – T1 matrix t_i^a
- **t2** (*numpy array*) – T2 matrix t_{ij}^{ab}

Returns

The result of the 6 diagrams that contribute to t1 and t2

Return type

list[numpy array]

CCM.ccDgrams.**dgram_akci_ck**(*v_phph, t1*)

Calculates $-V^{\{ak\}}_{\{ci\}} t^{\{c\}}_{\{k\}}$

Parameters

- **v_phph** (*numpy array*) – two body interaction matrix $V^{\{ai\}}_{\{bj\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_ck_acik**(*f_ph, t2*)

Calculates $f^{\{c\}}_{\{k\}} t^{\{ac\}}_{\{ik\}}$

Parameters

- **f_ph** (*numpy array*) – Fock matrix $f^{\{a\}}_{\{i\}}$
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cikl_cakl**(*v_phhh*, *t2*)Calculates $-0.5 * V^{\{ci\}}_{\{kl\}} t^{\{ca\}}_{\{kl\}}$ **Parameters**

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}}_{\{jk\}}$
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_ck_dali**(*v_pphh*, *t1*, *t2*)Calculates $V^{\{cd\}}_{\{kl\}} t^{\{c\}}_{\{k\}} t^{\{da\}}_{\{li\}}$ **Parameters**

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{\{ij\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_ck_ci**(*f_ph*, *t1*)Calculates $-0.5 * f^{\{c\}}_{\{k\}} t^{\{c\}}_{\{i\}}$ **Parameters**

- **f_ph** (*numpy array*) – Fock matrix $f^{\{a\}}_{\{i\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_ck_ak**(*f_ph*, *t1*)Calculates $-0.5 * f^{\{c\}}_{\{k\}} t^{\{a\}}_{\{k\}}$ **Parameters**

- **f_ph** (*numpy array*) – Fock matrix $f^{\{a\}}_{\{i\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bijk_bj**(*v_phhh*, *t1*)

Calculates $-V^{\{bi\}}_{\{jk\}}t^{\{b\}}_{\{j\}}$

Parameters

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{bi\}}_{\{jk\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdlk_cdli**(*v_phhh*, *t2*)

Calculates $-0.5 * V^{\{cd\}}_{\{lk\}}t^{\{cd\}}_{\{li\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{\{ij\}}$
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_dckl_dakl**(*v_pphh*, *t2*)

Calculates $-0.5 * V^{\{dc\}}_{\{kl\}}t^{\{da\}}_{\{kl\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{\{ij\}}$
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdlk_cl_di**(*v_pphh*, *t1*)

Calculates $-0.5 * V^{\{cd\}}_{\{lk\}}t^{\{c\}}_{\{l\}}t^{\{d\}}_{\{i\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{\{ij\}}$
- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_dk_al**(*v_pphh*, *t1*)

Calculates $0.5 * V^{\{cd\}}_{\{kl\}}t^{\{d\}}_{\{k\}}t^{\{a\}}_{\{l\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}}_{\{ij\}}$

- **t1** (*numpy array*) – $T^{\{a\}}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**pAB**(*val*)

Permutator for ab

Parameters

val (*numpy array*) – array to be permuted over

Returns

$val^{\{ab\}}_{\{ij\}} - val^{\{ba\}}_{\{ij\}}$

Return type

numpy array

CCM.ccDgrams.**pIJ**(*val*)

Permutator for ij

Parameters

val (*numpy array*) – array to be permuted over

Returns

$val^{\{ab\}}_{\{ij\}} - val^{\{ab\}}_{\{ji\}}$

Return type

numpy array

CCM.ccDgrams.**v_pppp_dgrams**(*v_pppp*, *t1*, *t2*)

Calculates both of the diagrams that use *v_pppp* using the fact that it is sparse

Parameters

- **v_pppp** (*list*[(*int*, *int*, *int*, *int*, *float*)] – nonzeros of the two body interaction matrix $V^{\{ab\}}_{\{cd\}}$
- **t1** (*numpy array*) – T_1 matrix t_i^a
- **t2** (*numpy array*) – T_2 matrix $t_{\{ij\}}^{\{ab\}}$

Returns

The result of the two diagrams that contribute to *t2*

Return type

numpy array, *numpy array*

CCM.ccDgrams.**dgram_klij_abkl**(*v_hhhh*, *t2*)

Calculates $0.5 * V^{\{kl\}}_{\{ij\}} t^{\{ab\}}_{\{kl\}}$

Parameters

- **v_hhhh** (*numpy array*) – two body interaction matrix $V^{\{kl\}}_{\{ij\}}$
- **t2** (*numpy array*) – $T^{\{ab\}}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bkcj_acik**(*v_phph*, *t2*)

Calculates $-P(ij)P(ab)V^{\{bk\}_{cj}}t^{\{cb\}_{ik}}$

Parameters

- **v_phph** (*numpy array*) – two body interaction matrix $V^{\{bk\}_{cj}}$
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bkij_ak**(*v_phhh*, *t1*)

Calculates $P(ab)V^{\{bk\}_{ij}}t^{\{a\}_{k}}$

Parameters

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{bk\}_{ij}}$
- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdki_acik_dblj**(*v_pphh*, *t2*)

Calculates $0.5 * P(ij)P(ab)V^{\{cd\}_{kl}}t^{\{ac\}_{ik}}t^{\{db\}_{lj}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{ij}}$
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdki_cdi_abkl**(*v_pphh*, *t2*)

Calculates $0.25 * P(ab)V^{\{cd\}_{kl}}t^{\{cd\}_{ij}}t^{\{ab\}_{kl}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{ij}}$
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_klij_ak_bl**(*v_hhhh*, *t1*)

Calculates $-0.5 * P(ab)V^{\{kl\}_{ij}}t^{\{a\}_{k}}t^{\{b\}_{l}}$

Parameters

- **v_hhhh** (*numpy array*) – two body interaction matrix $V^{\{kl\}_{ij}}$

- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bkci_ak_cj**(*v_phph*, *t1*)

Calculates - $P(ij)P(ab)V^{\{bk\}_{ci}}t^{\{a\}_{k}}t^{\{c\}_{j}}$

Parameters

- **v_phph** (*numpy array*) – two body interaction matrix $V^{\{ai\}_{bj}}$
- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cikl_ck_ablj**(*v_phhh*, *t1*, *t2*)

Calculates - $P(ij)V^{\{ci\}_{kl}}t^{\{c\}_{k}}t^{\{ab\}_{lj}}$

Parameters

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_da_dbij**(*v_ppph_res*, *t2*)

Calculates - $P(ab)X^{\{d\}_{at}}t^{\{db\}_{ij}}$

Parameters

- **v_ppph_res** (*numpy array*) – intermediate result 3 from [v_ppph_dgrams\(\)](#)
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_acik_bcjk**(*v_ppph_res*, *t2*)

Calculates - $P(ij)P(ab)X^{\{ac\}_{ij}}t^{\{bc\}_{jk}}$

Parameters

- **v_ppph_res** (*numpy array*) – intermediate result 4 from [v_ppph_dgrams\(\)](#)
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cikl_al_bcjk**(*v_phhh*, *t1*, *t2*)Calculates - $P(ij)P(ab)V^{\{ci\}_{\{kl\}}t^{\{a\}_{\{l\}}t^{\{bc\}_{\{jk\}}}$ **Parameters**

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}_{\{jk\}}$
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cjkl_ci_abkl**(*v_phhh*, *t1*, *t2*)Calculates $0.5 * P(ij)V^{\{cj\}_{\{kl\}}t^{\{c\}_{\{i\}}t^{\{ab\}_{\{kl\}}$ **Parameters**

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}_{\{jk\}}$
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{\{ij\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bijk_ak1**(*v_ppph_res*, *t1*)Calculates $0.5 * P(ab)X^{\{bi\}_{\{jk\}}t^{\{a\}_{\{k\}}$ **Parameters**

- **v_ppph_res** (*numpy array*) – intermediate result 5 from [v_ppph_dgrams\(\)](#)
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_bijk_ak2**(*v_ppph_res*, *t1*)Calculates $0.5 * P(ij)P(ab)X^{\{bi\}_{\{jk\}}t^{\{a\}_{\{k\}}$ **Parameters**

- **v_ppph_res** (*numpy array*) – intermediate result 6 from [v_ppph_dgrams\(\)](#)
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cjkl_ci_ak_bl**(*v_phhh*, *t1*)

Calculates $0.5 * P(ij)P(ab)V^{\{cj\}_{kl}t^{\{c\}_i}t^{\{a\}_k}t^{\{b\}_l\}}$

Parameters

- **v_phhh** (*numpy array*) – two body interaction matrix $V^{\{ai\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_i}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_ci_dj_abkl**(*v_phhh*, *t1*, *t2*)

Calculates $0.25 * P(ij)V^{\{cd\}_{kl}t^{\{c\}_i}t^{\{d\}_j}t^{\{ab\}_{kl}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_i}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_ak_bl_cdi_j**(*v_phhh*, *t1*, *t2*)

Calculates $0.25 * P(ab)V^{\{cd\}_{kl}t^{\{a\}_k}t^{\{b\}_l}t^{\{cd\}_{ij}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_i}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_ci_bl_adkj**(*v_phhh*, *t1*, *t2*)

Calculates $P(ij)P(ab)V^{\{cd\}_{kl}t^{\{c\}_i}t^{\{b\}_l}t^{\{ad\}_{kj}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_i}$ from the coupled cluster equations
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_ci_ak_dj_bl**(*v_pphh*, *t1*)

Calculates $0.25 * P(ij)P(ab)V^{\{cd\}_{kl}}t^{\{c\}_{i}}t^{\{a\}_{k}}t^{\{d\}_{j}}t^{\{b\}_{l}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_bdkl**(*v_pphh*, *t2*)

Calculates $-0.5 * V^{\{cd\}_{kl}}t^{\{bd\}_{kl}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdkl_cdjl**(*v_pphh*, *t2*)

Calculates $-0.5 * V^{\{cd\}_{kl}}t^{\{cd\}_{jl}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{jk}}$
- **t2** (*numpy array*) – $T^{\{ab\}_{ij}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_ck_bk**(*f_ph*, *t1*)

Calculates $- * f^{\{c\}_{k}}t^{\{b\}_{k}}$

Parameters

- **f_ph** (*numpy array*) – Fock matrix $f^{\{a\}_{i}}$
- **t1** (*numpy array*) – $T^{\{a\}_{i}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_ck_cj**(*f_ph*, *t1*)

Calculates $- * f^{\{c\}_{k}}t^{\{c\}_{j}}$

Parameters

- **f_ph** (*numpy array*) – Fock matrix $f^{\{a\}_{i}}$

- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdlk_cl_dj**(*v_pphh*, *t1*)

Calculates $-V^{\{cd\}_{\{kl\}}t^{\{c\}_{\{l\}}t^{\{d\}_{\{j\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{\{jk\}}$
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

CCM.ccDgrams.**dgram_cdlk_dk_bl**(*v_pphh*, *t1*)

Calculates $-V^{\{cd\}_{\{kl\}}t^{\{d\}_{\{k\}}t^{\{b\}_{\{l\}}$

Parameters

- **v_pphh** (*numpy array*) – two body interaction matrix $V^{\{ab\}_{\{jk\}}$
- **t1** (*numpy array*) – $T^{\{a\}_{\{i\}}$ from the coupled cluster equations

Returns

result of the contraction

Return type

numpy array

2.5 IMSRG

<i>IMSRG.commutators</i>	Module to evaluate the commutators of the IMSRG.
<i>IMSRG.generator</i>	Module to construct IMSRG generator.
<i>IMSRG.normal_ordering</i>	Module defining normal ordering utilities for the IMSRG.
<i>IMSRG.ode_solver</i>	Module to solve the IMSRG equations via direct integration of flow equations.

2.5.1 IMSRG.commutators

Module to evaluate the commutators of the IMSRG.

Functions

<i>antisymmetrize_2b</i> (a2)	Fully antisymmetrizes a two-body operator with respect to both pairs of indices
-------------------------------	---

continues on next page

Table 13 – continued from previous page

<code>antisymmetrize_2b_pq(a2)</code>	Antisymmetrizes a two-body operator with respect to the first two indices (pq)
<code>antisymmetrize_2b_rs(a2)</code>	Antisymmetrizes a two-body operator with respect to the last two indices (rs)
<code>evaluate_comm_110(occs, a1, b1)</code>	Evaluates the [1,1]->0 commutator contribution
<code>evaluate_comm_111(occs, a1, b1)</code>	Evaluates the [1,1]->1 commutator contribution
<code>evaluate_comm_121(occs, a1, b2)</code>	Evaluates the [1,2]->1 commutator contribution
<code>evaluate_comm_122(occs, a1, b2)</code>	Evaluates the [1,2]->2 commutator contribution
<code>evaluate_comm_220(occs, a2, b2)</code>	Evaluates the [2,2]->0 commutator contribution
<code>evaluate_comm_221(occs, a2, b2)</code>	Evaluates the [2,2]->1 commutator contribution using optimized implementation
<code>evaluate_comm_222(occs, a2, b2)</code>	Evaluates the complete [2,2]->2 commutator contribution
<code>evaluate_comm_222_ph(occs, a2, b2)</code>	Evaluates the particle-hole contribution to the [2,2]->2 commutator
<code>evaluate_comm_222_pphh(occs, a2, b2)</code>	Evaluates the particle-particle hole-hole contribution to the [2,2]->2 commutator
<code>evaluate_imsrg2_commutator(occs, a1, a2, b1, b2)</code>	Evaluates the complete commutator for IMSRG(2) flow equations

IMSRG.commutators.antisymmetrize_2b_pq(a2)

Antisymmetrizes a two-body operator with respect to the first two indices (pq)

Applies the antisymmetrization $A_{\{pq\}} = 1/2(1 - P_{\{pq\}})$ where $P_{\{pq\}}$ exchanges indices p and q

Parameters

a2 (*numpy array*) – Two-body matrix elements with indices pqrs

Returns

Partially antisymmetrized two-body matrix elements

Return type

numpy array

IMSRG.commutators.antisymmetrize_2b_rs(a2)

Antisymmetrizes a two-body operator with respect to the last two indices (rs)

Applies the antisymmetrization $A_{\{rs\}} = 1/2(1 - P_{\{rs\}})$ where $P_{\{rs\}}$ exchanges indices r and s

Parameters

a2 (*numpy array*) – Two-body matrix elements with indices pqrs

Returns

Partially antisymmetrized two-body matrix elements

Return type

numpy array

IMSRG.commutators.antisymmetrize_2b(a2)

Fully antisymmetrizes a two-body operator with respect to both pairs of indices

Applies complete antisymmetrization to both bra and ket indices, equivalent to $A_{\{pq\}}A_{\{rs\}}$ acting on the operator

Parameters

a2 (*numpy array*) – Two-body matrix elements with indices pqrs

Returns

Fully antisymmetrized two-body matrix elements

Return type

numpy array

`IMSRG.commutators.evaluate_comm_110(occs, a1, b1)`

Evaluates the [1,1]->0 commutator contribution

Computes the scalar (0-body) part of the commutator between two one-body operators

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a1** (*numpy array*) – First one-body operator
- **b1** (*numpy array*) – Second one-body operator

Returns

Scalar commutator contribution

Return type

float

`IMSRG.commutators.evaluate_comm_111(occs, a1, b1)`

Evaluates the [1,1]->1 commutator contribution

Computes the one-body part of the commutator between two one-body operators

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a1** (*numpy array*) – First one-body operator
- **b1** (*numpy array*) – Second one-body operator

Returns

One-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_121(occs, a1, b2)`

Evaluates the [1,2]->1 commutator contribution

Computes the one-body part of the commutator between a one-body and two-body operator

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a1** (*numpy array*) – One-body operator
- **b2** (*numpy array*) – Two-body operator

Returns

One-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_122(occs, a1, b2)`

Evaluates the [1,2]->2 commutator contribution

Computes the two-body part of the commutator between a one-body and two-body operator

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a1** (*numpy array*) – One-body operator
- **b2** (*numpy array*) – Two-body operator

Returns

Two-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_220(occs, a2, b2)`

Evaluates the [2,2]->0 commutator contribution

Computes the scalar (0-body) part of the commutator between two two-body operators

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

Scalar commutator contribution

Return type

float

`IMSRG.commutators.__evaluate_comm_221_naive(occs, a2, b2)`

Evaluates the [2,2]->1 commutator contribution using naive implementation

Computes the one-body part of the commutator between two two-body operators using a straightforward but potentially less efficient contraction pattern

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

One-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_221(occs, a2, b2)`

Evaluates the [2,2]->1 commutator contribution using optimized implementation

Computes the one-body part of the commutator between two two-body operators. This implementation pre-computes tensors contracted with occupation numbers for improved efficiency

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

One-body commutator contribution

Return type

numpy array

`IMSRG.commutators.__evaluate_comm_222_naive(occs, a2, b2)`

Evaluates the [2,2]->2 commutator contribution using naive implementation

Computes the two-body part of the commutator between two two-body operators using a straightforward contraction approach that may be less computationally efficient

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

Two-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_222_pphh(occs, a2, b2)`

Evaluates the particle-particle hole-hole contribution to the [2,2]->2 commutator

Computes the specific part of the two-body commutator involving contractions between particle-particle and hole-hole index pairs

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

Particle-particle hole-hole commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_222_ph(occs, a2, b2)`

Evaluates the particle-hole contribution to the [2,2]->2 commutator

Computes the specific part of the two-body commutator involving contractions between particle-hole index pairs with proper antisymmetrization

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

Particle-hole commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_comm_222(occs, a2, b2)`

Evaluates the complete [2,2]->2 commutator contribution

Computes the full two-body part of the commutator between two two-body operators by combining particle-particle hole-hole and particle-hole contributions

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a2** (*numpy array*) – First two-body operator
- **b2** (*numpy array*) – Second two-body operator

Returns

Complete two-body commutator contribution

Return type

numpy array

`IMSRG.commutators.evaluate_imsrg2_commutator(occs, a1, a2, b1, b2)`

Evaluates the complete commutator for IMSRG(2) flow equations

Computes all IMSRG(2) contributions to the commutator $C = [A, B]$ where A and B each contain one- and two-body parts, returning the 0-body, 1-body, and 2-body contributions to the result C

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **a1** (*numpy array*) – One-body part of first operator
- **a2** (*numpy array*) – Two-body part of first operator
- **b1** (*numpy array*) – One-body part of second operator
- **b2** (*numpy array*) – Two-body part of second operator

Returns

Zero-body, one-body, and two-body commutator contributions

Return type

float, numpy array, numpy array

2.5.2 IMSRG.generator

Module to construct IMSRG generator.

Functions

<code>build_1b_arctan_generator(occs, f[, delta])</code>	Constructs the one-body part of the arctan IMSRG generator
<code>build_1b_energy_difference(occs, f[, delta])</code>	Constructs one-body energy differences for IMSRG generator denominators
<code>build_2b_arctan_generator(occs, f, gamma[, ...])</code>	Constructs the two-body part of the arctan IMSRG generator

continues on next page

Table 14 – continued from previous page

<code>build_2b_energy_difference(occs, f[, delta])</code>	Constructs two-body energy differences for IMSRG generator denominators
<code>get_hole_spes(occs, f)</code>	Extracts single-particle energies for hole states from the Fock matrix
<code>get_particle_spes(occs, f[, delta])</code>	Extracts single-particle energies for particle states from the Fock matrix

`IMSRG.generator.get_hole_spes(occs, f)`

Extracts single-particle energies for hole states from the Fock matrix

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – Fock matrix containing single-particle energies on the diagonal

Returns

Single-particle energies for occupied (hole) states

Return type

numpy array

`IMSRG.generator.get_particle_spes(occs, f, delta=0.0)`

Extracts single-particle energies for particle states from the Fock matrix

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – Fock matrix containing single-particle energies on the diagonal
- **delta** (*float*) – Optional; energy shift to avoid degeneracies in denominators

Returns

Single-particle energies for unoccupied (particle) states

Return type

numpy array

`IMSRG.generator.build_1b_energy_difference(occs, f, delta=0.0)`

Constructs one-body energy differences for IMSRG generator denominators

Computes the energy differences $e_i - e_a$ between hole and particle states, which appear in the denominators of the IMSRG generator. A small regularization term prevents division by zero

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – Fock matrix containing single-particle energies
- **delta** (*float*) – Optional; energy shift to avoid degeneracies in denominators

Returns

Matrix of energy differences for one-body generator denominators

Return type

numpy array

`IMSRG.generator.build_2b_energy_difference(occs, f, delta=0.0)`

Constructs two-body energy differences for IMSRG generator denominators

Computes the energy differences $e_i + e_j - e_a - e_b$ between hole and particle state pairs, which appear in the denominators of the IMSRG generator. A small regularization term prevents division by zero

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – Fock matrix containing single-particle energies
- **delta** (*float*) – Optional; energy shift to avoid degeneracies in denominators

Returns

Tensor of energy differences for two-body generator denominators

Return type

numpy array

`IMSRG.generator.build_1b_arctan_generator(occs, f, delta=0.0)`

Constructs the one-body part of the arctan IMSRG generator

Builds the generator $\eta^{\{1\}} = (1/2) \arctan(2f^{\{ai\}}/e_{\{ai\}})$ that drives the one-body part of the IMSRG flow equations, ensuring decoupling of particle-hole excitations from the reference state. An optional delta pushes up the energies of the particle states to prevent vanishing energy denominators and ensure numerical stability. Most calculations should not need this

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – One-body Fock matrix
- **delta** (*float*) – Optional; energy shift to avoid degeneracies in denominators

Returns

One-body arctangent generator matrix elements

Return type

numpy array

`IMSRG.generator.build_2b_arctan_generator(occs, f, gamma, delta=0.0)`

Constructs the two-body part of the arctan IMSRG generator

Builds the generator $\eta^{\{2\}} = (1/2) \arctan(2\Gamma^{\{abij\}}/e_{\{abij\}})$ that drives the two-body part of the IMSRG flow equations, ensuring decoupling of particle-hole excitations from the reference state. An optional delta pushes up the energies of the particle states to prevent vanishing energy denominators and ensure numerical stability. Most calculations should not need this

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state
- **f** (*numpy array*) – One-body Fock matrix for constructing energy denominators
- **gamma** (*numpy array*) – Two-body interaction matrix
- **delta** (*float*) – Optional; energy shift to avoid degeneracies in denominators

Returns

Two-body arctangent generator matrix elements

Return type

numpy array

2.5.3 IMSRG.normal_ordering

Module defining normal ordering utilities for the IMSRG.

Functions

<code>compute_normal_ordered_hamiltonian_no2b(...)</code>	Computes the normal-ordered Hamiltonian with respect to a reference state
<code>create_occupations(basis, ref)</code>	Creates an array of occupation numbers indicating which states in the basis are occupied in the reference state
<code>expand_h2(h2)</code>	Expands two-body matrix elements by applying anti-symmetrization relations
<code>expand_h3(h3)</code>	Expands three-body matrix elements by applying anti-symmetrization relations
<code>get_three_body_permutations(pp, qq, rr)</code>	Generates all antisymmetrized permutations of three indices with appropriate signs

`IMSRG.normal_ordering.create_occupations(basis, ref)`

Creates an array of occupation numbers indicating which states in the basis are occupied in the reference state

Parameters

- **basis** (`list[tuple[int, int, int, int, int]]`) – List of all possible single-particle states in the basis
- **ref** (`list[tuple[int, int, int, int, int]]`) – List of occupied single-particle states in the reference configuration

Returns

Array with 1.0 for occupied states and 0.0 for unoccupied states

Return type

numpy array

`IMSRG.normal_ordering.expand_h2(h2)`

Expands two-body matrix elements by applying antisymmetrization relations

Generates all antisymmetrized matrix elements from the input two-body interactions using the relations $\langle pq|rs \rangle = -\langle qp|rs \rangle = -\langle pq|sr \rangle = \langle qp|sr \rangle$

Parameters

h2 (`list[tuple[int, int, int, int, float]]`) – List of two-body matrix elements in format [p, q, r, s, matrix_element]

Returns

Expanded list with all antisymmetrized two-body matrix elements

Return type

`list[tuple[int, int, int, int, float]]`

`IMSRG.normal_ordering.get_three_body_permutations(pp, qq, rr)`

Generates all antisymmetrized permutations of three indices with appropriate signs

Computes the six permutations of three indices with the correct antisymmetrization factors for fermionic systems

Parameters

- **pp** (`int`) – First index
- **qq** (`int`) – Second index

- **rr** (*int*) – Third index

Returns

List of permutations with format [index1, index2, index3, sign_factor]

Return type

list[tuple[int, int, int, float]]

IMSRG.normal_ordering.expand_h3(*h3*)

Expands three-body matrix elements by applying antisymmetrization relations

Generates all antisymmetrized three-body matrix elements from the input interactions (with $p < q < r$ and $s < t < u$) by applying permutations to both bra and ket indices with appropriate sign factors

Parameters

h3 (*list[tuple[int, int, int, int, int, int, float]]*) – List of three-body matrix elements in format [p, q, r, s, t, u, matrix_element]

Returns

Expanded list with all antisymmetrized three-body matrix elements

Return type

list[tuple[int, int, int, int, int, int, float]]

IMSRG.normal_ordering.compute_normal_ordered_hamiltonian_no2b(*occs, h1, h2, h3=None*)

Computes the normal-ordered Hamiltonian with respect to a reference state

Transforms the Hamiltonian to normal-ordered form by summing over occupied states, yielding the reference-state energy, effective one-body (Fock) operator, and effective two-body interactions

Parameters

- **occs** (*numpy array*) – Occupation numbers for each single-particle state (1.0 if occupied, 0.0 if empty)
- **h1** (*list[tuple[int, int, float]]*) – One-body matrix elements in format [p, q, matrix_element]
- **h2** (*list[tuple[int, int, int, int, float]]*) – Two-body matrix elements in format [p, q, r, s, matrix_element]
- **h3** (*list[tuple[int, int, int, int, int, int, float]] | None*) – Optional; three-body matrix elements in format [p, q, r, s, t, u, matrix_element]

Returns

Reference state energy, normal-ordered one-body operator (Fock matrix), normal-ordered two-body operator

Return type

float, numpy array, numpy array

2.5.4 IMSRG.ode_solver

Module to solve the IMSRG equations via direct integration of flow equations.

Functions

<i>flatten_hamiltonian</i> (dim, e0, f, gamma)	Flattens Hamiltonian components into a single array for ODE integration
<i>imsrg_rhs</i> (s, packed, occs, delta, dim, ...)	Right-hand side function for IMSRG flow equations

continues on next page

Table 16 – continued from previous page

<code>norm(x)</code>	Computes the Frobenius norm of a tensor
<code>solve_imsrg2(occs, e0, f, gamma[, s_init, ...])</code>	Solves IMSRG(2) flow equations in a single integration step
<code>unflatten_hamiltonian(dim, packed)</code>	Reconstructs Hamiltonian components from flattened array
<code>write_op_1b(dim, op, fname)</code>	Writes one-body operator matrix elements to file
<code>write_op_2b(dim, op, fname)</code>	Writes two-body operator matrix elements to file

`IMSRG.ode_solver.flatten_hamiltonian(dim, e0, f, gamma)`

Flattens Hamiltonian components into a single array for ODE integration

Packs the scalar energy, one-body operator, and two-body operator into a single one-dimensional array suitable for scipy ODE solvers

Parameters

- **dim** (*int*) – Dimension of the single-particle basis
- **e0** (*float*) – Zero-body (scalar) energy contribution
- **f** (*numpy array*) – One-body operator matrix
- **gamma** (*numpy array*) – Two-body operator tensor

Returns

Flattened array containing all Hamiltonian components

Return type

numpy array

`IMSRG.ode_solver.unflatten_hamiltonian(dim, packed)`

Reconstructs Hamiltonian components from flattened array

Unpacks a one-dimensional array back into scalar energy, one-body matrix, and two-body tensor components for IMSRG calculations

Parameters

- **dim** (*int*) – Dimension of the single-particle basis
- **packed** (*numpy array*) – Flattened array containing Hamiltonian components

Returns

Zero-body energy, one-body matrix, and two-body tensor

Return type

float, numpy array, numpy array

`IMSRG.ode_solver.write_op_1b(dim, op, fname)`

Writes one-body operator matrix elements to file

Outputs non-zero matrix elements of a one-body operator to a text file in the format: index_p index_q matrix_element

Parameters

- **dim** (*int*) – Dimension of the operator matrix
- **op** (*numpy array*) – One-body operator matrix
- **fname** (*str*) – Output filename

IMSRG.ode_solver.**write_op_2b**(*dim, op, fname*)

Writes two-body operator matrix elements to file

Outputs non-zero matrix elements of a two-body operator to a text file in the format: index_p index_q index_r index_s matrix_element

Parameters

- **dim** (*int*) – Dimension of each operator index
- **op** (*numpy array*) – Two-body operator tensor
- **fname** (*str*) – Output filename

IMSRG.ode_solver.**norm**(*x*)

Computes the Frobenius norm of a tensor

Parameters

x (*numpy array*) – Input tensor

Returns

Frobenius norm (sum of squared elements)

Return type

float

IMSRG.ode_solver.**imsrg_rhs**(*s, packed, occs, delta, dim, data_tracking, eta_criterion=0.001*)

Right-hand side function for IMSRG flow equations

Computes the derivatives dH/ds for the IMSRG flow equations using the commutator $[eta, H]$ where eta is the arctan generator. This is not a strictly pure function; *data_tracking* is modified by the function to track integration data

Parameters

- **s** (*float*) – Flow parameter
- **packed** (*numpy array*) – Flattened Hamiltonian components
- **occs** (*numpy array*) – Occupation numbers for reference state
- **delta** (*float*) – Energy shift parameter for generator denominators
- **dim** (*int*) – Dimension of single-particle basis
- **data_tracking** (*list*) – List for storing flow data during integration
- **eta_criterion** (*float*) – Optional; Generator norm at which solution should truncate.
Default = $1e-3$

Returns

Flattened derivatives for ODE integration

Return type

numpy array

IMSRG.ode_solver.**solve_imsrg2**(*occs, e0, f, gamma, s_init=0.0, s_max=40, delta=0.0, eta_criterion=0.001*)

Solves IMSRG(2) flow equations in a single integration step

Integrates the IMSRG flow equations from initial to final flow parameter values, returning the converged energy (and flow data for possible further analysis)

Parameters

- **occs** (*numpy array*) – Occupation numbers for reference state

- **e0** (*float*) – Initial zero-body energy
- **f** (*numpy array*) – Initial one-body operator
- **gamma** (*numpy array*) – Initial two-body operator
- **s_init** (*float*) – Optional; initial flow parameter value, default = 0.0
- **s_max** (*float*) – Optional; final flow parameter value, default = 40.0
- **delta** (*float*) – Optional; energy shift for generator denominators
- **eta_criterion** (*float*) – Optional; Generator norm at which solution should truncate.
Default = 1e-3

Returns

Final IMSRG energy and flow tracking data

Return type

float, list[tuple[float, float, float, float]]

PYTHON MODULE INDEX

C

CCM.ccDgrams, 28
CCM.coupled_cluster, 21
CCM.three_body_utils, 26
constants, 8

f

FCI.few_body_diagonalization, 8

h

HF.hartree_fock, 18

i

IMSRG.commutators, 38
IMSRG.generator, 43
IMSRG.normal_ordering, 46
IMSRG.ode_solver, 47

l

lattice, 3

r

references, 8

Symbols

`__evaluate_comm_221_naive()` (in module *IM-SRG.commutators*), 41
`__evaluate_comm_222_naive()` (in module *IM-SRG.commutators*), 42

A

`add_2body_ops()` (in module *FCI.few_body_diagonalization*), 16
`add_3body_ops()` (in module *FCI.few_body_diagonalization*), 17
`antisymmetrize_2b()` (in module *IM-SRG.commutators*), 39
`antisymmetrize_2b_pq()` (in module *IM-SRG.commutators*), 39
`antisymmetrize_2b_rs()` (in module *IM-SRG.commutators*), 39

B

`build_1b_arctan_generator()` (in module *IM-SRG.generator*), 45
`build_1b_energy_difference()` (in module *IM-SRG.generator*), 44
`build_2b_arctan_generator()` (in module *IM-SRG.generator*), 45
`build_2b_energy_difference()` (in module *IM-SRG.generator*), 44

C

CCM.ccDgrams
 module, 28
CCM.coupled_cluster
 module, 21
CCM.three_body_utils
 module, 26
`ccsd_energy()` (in module *CCM.coupled_cluster*), 22
`ccsd_solver()` (in module *CCM.coupled_cluster*), 24
`compute_normal_ordered_hamiltonian_no2b()` (in module *IMSRG.normal_ordering*), 47
 constants
 module, 8
`contacts()` (in module *lattice*), 6

`contract_2nf()` (in module *HF.hartree_fock*), 18
`contract_3nf()` (in module *HF.hartree_fock*), 19
`create_occupations()` (in module *IM-SRG.normal_ordering*), 46
`csr_matrix_tolist_2body()` (in module *FCI.few_body_diagonalization*), 16
`csr_matrix_tolist_3body()` (in module *FCI.few_body_diagonalization*), 17

D

`dgram_acik_bcjk()` (in module *CCM.ccDgrams*), 34
`dgram_akci_ck()` (in module *CCM.ccDgrams*), 29
`dgram_bijk_ak1()` (in module *CCM.ccDgrams*), 35
`dgram_bijk_ak2()` (in module *CCM.ccDgrams*), 35
`dgram_bijk_bj()` (in module *CCM.ccDgrams*), 30
`dgram_bkci_ak_cj()` (in module *CCM.ccDgrams*), 34
`dgram_bkcj_acik()` (in module *CCM.ccDgrams*), 32
`dgram_bkij_ak()` (in module *CCM.ccDgrams*), 33
`dgram_cdkl_acik_dblj()` (in module *CCM.ccDgrams*), 33
`dgram_cdkl_ak_bl_cdi_j()` (in module *CCM.ccDgrams*), 36
`dgram_cdkl_bdkl()` (in module *CCM.ccDgrams*), 37
`dgram_cdkl_cdi_j_abkl()` (in module *CCM.ccDgrams*), 33
`dgram_cdkl_cdj_l()` (in module *CCM.ccDgrams*), 37
`dgram_cdkl_ci_ak_dj_bl()` (in module *CCM.ccDgrams*), 36
`dgram_cdkl_ci_bl_adkj()` (in module *CCM.ccDgrams*), 36
`dgram_cdkl_ci_dj_abkl()` (in module *CCM.ccDgrams*), 36
`dgram_cdkl_ck_dali()` (in module *CCM.ccDgrams*), 30
`dgram_cdkl_dk_al()` (in module *CCM.ccDgrams*), 31
`dgram_cdlk_cdli()` (in module *CCM.ccDgrams*), 31
`dgram_cdlk_cl_di()` (in module *CCM.ccDgrams*), 31
`dgram_cdlk_cl_dj()` (in module *CCM.ccDgrams*), 38
`dgram_cdlk_dk_bl()` (in module *CCM.ccDgrams*), 38
`dgram_cikl_al_bcjk()` (in module *CCM.ccDgrams*), 35
`dgram_cikl_cakl()` (in module *CCM.ccDgrams*), 30

dgram_cikl_ck_ablj() (in module CCM.ccDgrams), 34
 dgram_cjkl_ci_abkl() (in module CCM.ccDgrams), 35
 dgram_cjkl_ci_ak_bl() (in module CCM.ccDgrams), 35
 dgram_ck_acik() (in module CCM.ccDgrams), 29
 dgram_ck_ak() (in module CCM.ccDgrams), 30
 dgram_ck_bk() (in module CCM.ccDgrams), 37
 dgram_ck_ci() (in module CCM.ccDgrams), 30
 dgram_ck_cj() (in module CCM.ccDgrams), 37
 dgram_da_dbij() (in module CCM.ccDgrams), 34
 dgram_dckl_dakl() (in module CCM.ccDgrams), 31
 dgram_klij_abkl() (in module CCM.ccDgrams), 32
 dgram_klij_ak_bl() (in module CCM.ccDgrams), 33

E

evaluate_comm_110() (in module IM-SRG.commutators), 40
 evaluate_comm_111() (in module IM-SRG.commutators), 40
 evaluate_comm_121() (in module IM-SRG.commutators), 40
 evaluate_comm_122() (in module IM-SRG.commutators), 40
 evaluate_comm_220() (in module IM-SRG.commutators), 41
 evaluate_comm_221() (in module IM-SRG.commutators), 41
 evaluate_comm_222() (in module IM-SRG.commutators), 43
 evaluate_comm_222_ph() (in module IM-SRG.commutators), 42
 evaluate_comm_222_pphh() (in module IM-SRG.commutators), 42
 evaluate_imsrg2_commutator() (in module IM-SRG.commutators), 43
 expand_h2() (in module IMSRG.normal_ordering), 46
 expand_h3() (in module IMSRG.normal_ordering), 47

F

FCI.few_body_diagonalization module, 8
 fill_1b_op_in_2b_basis() (in module FCI.few_body_diagonalization), 10
 fill_1b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 11
 fill_1b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 12
 fill_2b_op_in_2b_basis() (in module FCI.few_body_diagonalization), 11
 fill_2b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 11

fill_2b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 12
 fill_3b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 11
 fill_3b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 12
 fill_shift_op() (in module FCI.few_body_diagonalization), 13
 flatten_hamiltonian() (in module IM-SRG.ode_solver), 48

G

get_1body_matrix() (in module HF.hartree_fock), 18
 get_3NF() (in module CCM.three_body_utils), 26
 get_3NF_Eref() (in module CCM.three_body_utils), 27
 get_3NF_fock() (in module CCM.three_body_utils), 27
 get_3NF_tbme() (in module CCM.three_body_utils), 27
 get_all_interactions() (in module CCM.coupled_cluster), 22
 get_csr_1b_op_in_2b_basis() (in module FCI.few_body_diagonalization), 13
 get_csr_1b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 14
 get_csr_1b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 15
 get_csr_2b_op_in_2b_basis() (in module FCI.few_body_diagonalization), 14
 get_csr_2b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 14
 get_csr_2b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 15
 get_csr_3b_op_in_3b_basis() (in module FCI.few_body_diagonalization), 15
 get_csr_3b_op_in_4b_basis() (in module FCI.few_body_diagonalization), 16
 get_csr_matrix_scalar_op() (in module FCI.few_body_diagonalization), 10
 get_fock_matrices() (in module CCM.coupled_cluster), 21
 get_hole_spes() (in module IMSRG.generator), 44
 get_lattice() (in module lattice), 4
 get_many_body_states() (in module FCI.few_body_diagonalization), 10
 get_norm_ord_int() (in module CCM.coupled_cluster), 25
 get_norm_ordered_ham() (in module CCM.coupled_cluster), 25
 get_particle_spes() (in module IMSRG.generator), 44
 get_ref_energy() (in module CCM.coupled_cluster), 22
 get_shift_op() (in module FCI.few_body_diagonalization), 16
 get_sp_basis() (in module lattice), 4

get_three_body_permutations() (in module *IMSRG.normal_ordering*), 46

H

HF.hartree_fock
module, 18

HF_energy() (in module *HF.hartree_fock*), 19

HF_iter() (in module *HF.hartree_fock*), 20

I

IMSRG.commutators
module, 38

IMSRG.generator
module, 43

IMSRG.normal_ordering
module, 46

IMSRG.ode_solver
module, 47

imsrg_rhs() (in module *IMSRG.ode_solver*), 49

init_density() (in module *HF.hartree_fock*), 19

L

lattice
module, 3

left() (in module *lattice*), 5

M

make_HF_ham() (in module *HF.hartree_fock*), 19

makeState() (in module *lattice*), 7

module

CCM.ccDgrams, 28

CCM.coupled_cluster, 21

CCM.three_body_utils, 26

constants, 8

FCI.few_body_diagonalization, 8

HF.hartree_fock, 18

IMSRG.commutators, 38

IMSRG.generator, 43

IMSRG.normal_ordering, 46

IMSRG.ode_solver, 47

lattice, 3

references, 8

N

NNNcontact() (in module *lattice*), 6

norm() (in module *IMSRG.ode_solver*), 49

num_permutations() (in module *FCI.few_body_diagonalization*), 13

O

order_state() (in module *CCM.three_body_utils*), 26

P

p_x() (in module *lattice*), 6

p_y() (in module *lattice*), 7

p_z() (in module *lattice*), 7

pAB() (in module *CCM.ccDgrams*), 32

phys_unit() (in module *lattice*), 4

pIJ() (in module *CCM.ccDgrams*), 32

R

reference_to_holes() (in module *references*), 8

references
module, 8

right() (in module *lattice*), 5

S

site2index() (in module *lattice*), 5

solve_HF() (in module *HF.hartree_fock*), 20

solve_imsrg2() (in module *IMSRG.ode_solver*), 49

state2index() (in module *lattice*), 4

states2PHSpace() (in module *lattice*), 7

T

t1Init() (in module *CCM.coupled_cluster*), 23

t1Iter() (in module *CCM.coupled_cluster*), 23

t2Init() (in module *CCM.coupled_cluster*), 23

t2Iter() (in module *CCM.coupled_cluster*), 24

Tkin() (in module *lattice*), 5

U

unflatten_hamiltonian() (in module *IMSRG.ode_solver*), 48

V

v_ppph_dgrams() (in module *CCM.ccDgrams*), 29

v_pppp_dgrams() (in module *CCM.ccDgrams*), 32

W

write_op_1b() (in module *IMSRG.ode_solver*), 48

write_op_2b() (in module *IMSRG.ode_solver*), 48