PyBlock

Release 0.1

Jun 04, 2021

Contents:

CHAPTER 1

DMRG Theoretical Aspects

1.1 DMRG Quantum Chemistry Hamiltonian in Spatial Orbitals

1.1.1 Hamiltonian

The quantum chemistry Hamiltonian is written as follows

$$\hat{H} = \sum_{ij,\sigma} t_{ij}\ a^\dagger_{i\sigma} a_{j\sigma} + \frac{1}{2} \sum_{ijkl,\sigma\sigma'} v_{ijkl}\ a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}$$

where

$$t_{ij} = t_{(ij)} = \int \mathrm{d}x\ \phi^*_i(x) \left(-\frac{1}{2}\nabla^2 - \sum_a \frac{Z_a}{r_a} \right) \phi_j(x)$$

$$v_{ijkl} = v_{(ij)(kl)} = v_{(kl)(ij)} = \int \mathrm{d}x_1 \mathrm{d}x_2\ \frac{\phi^*_i(x_1)\phi^*_k(x_2)\phi_l(x_2)\phi_j(x_1)}{r_{12}}$$

Note that here the order of $ijkl$ is the same as that in `FCIDUMP` (chemist's notation $[ij|kl]$).

1.1.2 Partitioning in Spatial Orbitals

The partitioning of Hamiltonian in left (L) and right (R) blocks is given by

$$\begin{aligned}
\hat{H} = {} & \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R \\
& + \Big(\sum_{i\in L,\sigma} a^\dagger_{i\sigma} \hat{S}^R_{i\sigma} + h.c. \Big) + \Big(\sum_{i\in L,\sigma} a^\dagger_{i\sigma} \hat{R}^R_{i\sigma} + h.c. + \sum_{i\in R,\sigma} a^\dagger_{i\sigma} \hat{R}^L_{i\sigma} + h.c. \Big) \\
& + \frac{1}{2} \Big(\sum_{ik\in L,\sigma\sigma'} \hat{A}^L_{ik,\sigma\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + h.c. \Big) + \sum_{ij\in L} \hat{B}_{ij} \hat{Q}^R_{ij} - \sum_{il\in L,\sigma\sigma'} \hat{B}'_{il\sigma\sigma'} \hat{Q}'^R_{il\sigma\sigma'}
\end{aligned}$$

where the normal and complementary operators are defined by

$$\begin{aligned}
\hat{S}^{L/R}_{i\sigma} &= \sum_{j\in L/R} t_{ij} a_{j\sigma}, \\
\hat{R}^{L/R}_{i\sigma} &= \sum_{jkl\in L/R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}, \\
\hat{A}_{ik,\sigma\sigma'} &= a^\dagger_{i\sigma} a^\dagger_{k\sigma'}, \\
\hat{B}_{ij} &= \sum_{\sigma} a^\dagger_{i\sigma} a_{j\sigma}, \\
\hat{B}'_{il,\sigma\sigma'} &= a^\dagger_{i\sigma} a_{l\sigma'}, \\
\hat{P}^{R}_{ik,\sigma\sigma'} &= \sum_{jl\in R} v_{ijkl} a_{l\sigma'} a_{j\sigma}, \\
\hat{Q}^{R}_{ij} &= \sum_{kl\in R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}, \\
\hat{Q}'^{R}_{il,\sigma\sigma'} &= \sum_{jk\in R} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}
\end{aligned}$$

Note that we need to move all on-site interaction into local Hamiltonian, so that when construction interaction terms in Hamiltonian, operators anticommute (without giving extra constant terms).

Derivation

First consider one-electron term. ij indices have only two possibilities: i left, j right, or i right, j left. Index i must be associated with creation operator. So the second case is the Hermitian conjugate of the first case. Namely,

$$\sum_{i\in L,\sigma} a^\dagger_{i\sigma}\hat{S}^R_{i\sigma} + h.c. = \sum_{i\in L,\sigma} a^\dagger_{i\sigma}\hat{S}^R_{i\sigma} + \sum_{j\in L,\sigma} \hat{S}^{R\dagger}_{j\sigma} a_{j\sigma} = \sum_{i\in L/R, j\in R/L,\sigma} t_{ij} a^\dagger_{i\sigma} a_{j\sigma}$$

Next consider one of $ijkl$ in left, and three of them in right. These terms are

$$\begin{aligned}
\hat{H}_{1L,3R} &= \frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma} + \frac{1}{2}\sum_{j\in L,ikl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma} + \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma} + \frac{1}{2}_{l\in L,i} \\
&= \left[\frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma} + \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}\right] + \frac{1}{2}\sum_{j\in L,ikl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma} + \frac{1}{2}_{l\in}
\end{aligned}$$

where the terms in bracket equal to first and third terms in left-hand-side. Outside the bracket are second, forth terms.

The conjugate of third term in rhs is second term in rhs

$$\frac{1}{2}\sum_{j\in L,ikl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{j\sigma} a^\dagger_{l\sigma'} a_{k\sigma'} a_{i\sigma} = \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{lkji} a^\dagger_{k\sigma} a^\dagger_{i\sigma'} a_{j\sigma'} a_{l\sigma} = \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma'} a^\dagger_{k\sigma} a_{l\sigma} a_{j\sigma'}$$

The conjugate of forth term in rhs is first term in rhs

$$\frac{1}{2}\sum_{l\in L,ijk\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{j\sigma} a^\dagger_{l\sigma'} a_{k\sigma'} a_{i\sigma} = \frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{lkji} a^\dagger_{k\sigma} a^\dagger_{i\sigma'} a_{j\sigma'} a_{l\sigma} = \frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl} a^\dagger_{i\sigma'} a^\dagger_{k\sigma} a_{l\sigma} a_{j\sigma'}$$

Therefore, using $v_{ijkl} = v_{klij}$

$$\begin{aligned}
\hat{H}_{1L,3R} &= \left[\frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma}\right] + h.c. \\
&= \left[\frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{k\in L,ijl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{k\sigma'}a^\dagger_{i\sigma}a_{j\sigma}a_{l\sigma'}\right] + h.c. \\
&= \left[\frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{i\in L,jkl\in R,\sigma\sigma'} v_{klij}a^\dagger_{i\sigma'}a^\dagger_{k\sigma}a_{l\sigma}a_{j\sigma'}\right] + h.c. \\
&= \sum_{i\in L,jkl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + h.c. \\
&= \sum_{i\in L,\sigma} a^\dagger_{i\sigma}\sum_{jkl\in R,\sigma'} v_{ijkl}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + h.c. = \sum_{i\in L,\sigma} a^\dagger_{i\sigma}R^R_{i\sigma} + h.c.
\end{aligned}$$

Next consider the two creation operators together in left or in together in right. There are two cases. The second case is the conjugate of the first case, namely,

$$\sum_{ik\in R,jl\in L,\sigma\sigma'} a^\dagger_{i\sigma}a^\dagger_{k\sigma'}v_{ijkl}a_{l\sigma'}a_{j\sigma} = \sum_{jl\in R,ik\in L,\sigma\sigma'} a^\dagger_{j\sigma}a^\dagger_{l\sigma'}v_{jilk}a_{k\sigma'}a_{i\sigma} = \sum_{ik\in L,jl\in R,\sigma\sigma'} v_{jilk}a^\dagger_{j\sigma}a^\dagger_{l\sigma'}a_{k\sigma'}a_{i\sigma} = \sum_{ik\in L,jl\in R,\sigma\sigma'} v_{ijkl}\Big(a^\dagger_{i\sigma}a$$

This explains the $\hat{A}\hat{P}$ term. The last situation is, one creation in left and one creation in right. Note that when exchange two elementary operators, one creation and one annihilation, one in left and one in right, they must anticommute.

$$\begin{aligned}
\hat{H}_{2L,2R} &= \frac{1}{2}\sum_{il\in L,jk\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{kl\in L,ij\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{jk\in L} \\
&= -\frac{1}{2}\sum_{il\in L,jk\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{l\sigma'}a^\dagger_{k\sigma'}a_{j\sigma} + \frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} + \frac{1}{2}\sum_{kl\in L,ij\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} - \frac{1}{2}\sum_{jk}
\end{aligned}$$

where the first, forth terms are combing different spins. The second, third terms are for the same spin. First consider the same-spin case

$$\begin{aligned}
&\frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} + \frac{1}{2}\sum_{kl\in L,ij\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} \\
=&\frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} + \frac{1}{2}\sum_{kl\in L,ij\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{k\sigma'}a_{l\sigma'}a^\dagger_{i\sigma}a_{j\sigma} \\
=&\frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} + \frac{1}{2}\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{klij}a^\dagger_{i\sigma'}a_{j\sigma'}a^\dagger_{k\sigma}a_{l\sigma} \\
=&\sum_{ij\in L,kl\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{j\sigma}a^\dagger_{k\sigma'}a_{l\sigma'} = \sum_{ij\in L}\sum_{\sigma} a^\dagger_{i\sigma}a_{j\sigma}\sum_{kl\in R_k}\sum_{\sigma'} v_{ijkl}a^\dagger_{k\sigma'}a_{l\sigma'} = \sum_{ij\in L}\hat{B}_{ij}\hat{Q}^R_{ij}
\end{aligned}$$

For the different-spin case,

$$\begin{aligned}
&-\frac{1}{2}\sum_{il\in L,jk\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{l\sigma'}a^\dagger_{k\sigma'}a_{j\sigma} - \frac{1}{2}\sum_{jk\in L,il\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{l\sigma'}a^\dagger_{k\sigma'}a_{j\sigma} = -\sum_{il\in L,jk\in R,\sigma\sigma'} v_{ijkl}a^\dagger_{i\sigma}a_{l\sigma'}a^\dagger_{k\sigma'}a_{j\sigma} \\
=& -\sum_{il\in L\sigma\sigma'} a^\dagger_{i\sigma}a_{l\sigma'}\sum_{jk\in R} v_{ijkl}a^\dagger_{k\sigma'}a_{j\sigma} = -\sum_{il\in L\sigma\sigma'} \hat{B}'_{il\sigma\sigma'}{\hat{Q}'_{il\sigma\sigma'}}^R
\end{aligned}$$

Normal/Complementary Partitioning

The above version is used when left block is short in length. Note that all terms should be written in a way that operators for particles in left block should appear in the left side of operator string, and operators for particles in right block should appear in the right side of operator string. To write the Hermitian conjugate explicitly, we have

$$\begin{aligned}\hat{H}^{NC} = & \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R \\ & + \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{S}^R_{i\sigma} - a_{i\sigma} \hat{S}^{R\dagger}_{i\sigma} \right) + \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{R}^R_{i\sigma} - a_{i\sigma} \hat{R}^{R\dagger}_{i\sigma} \right) + \sum_{i\in R,\sigma} \left(\hat{R}^{L\dagger}_{i\sigma} a_{i\sigma} - \hat{R}^L_{i\sigma} a^\dagger_{i\sigma} \right) \\ & + \frac{1}{2} \sum_{ik\in L,\sigma\sigma'} \left(\hat{A}_{ik,\sigma\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + \hat{A}^\dagger_{ik,\sigma\sigma'} \hat{P}^{R\dagger}_{ik,\sigma\sigma'} \right) + \sum_{ij\in L} \hat{B}_{ij} \hat{Q}^R_{ij} - \sum_{il\in L,\sigma\sigma'} \hat{B}'_{il\sigma\sigma'} \hat{Q}'^R_{il\sigma\sigma'}\end{aligned}$$

Note that no minus sign for Hermitian conjugate terms with A, P because these are not Fermion operators.

Also note that

$$\sum_{i\in L,\sigma} a^\dagger_{i\sigma} \hat{S}^R_{i\sigma} = \sum_{i\in L, j\in R,\sigma} t_{ij} a^\dagger_{i\sigma} a_{j\sigma} = \sum_{j\in R,\sigma} S^{L\dagger}_{j\sigma} a_{j\sigma}$$

Define

$$\hat{R}'^{L/R}_{i\sigma} = \frac{1}{2} \hat{S}^{L/R}_{i\sigma} + \hat{R}^{L/R}_{i\sigma} = \frac{1}{2} \sum_{j\in L/R} t_{ij} a_{j\sigma} + \sum_{jkl\in L/R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}$$

we have

$$\begin{aligned}\hat{H}^{NC} = & \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R + \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{R}'^R_{i\sigma} - a_{i\sigma} \hat{R}'^{R\dagger}_{i\sigma} \right) + \sum_{i\in R,\sigma} \left(\hat{R}'^{L\dagger}_{i\sigma} a_{i\sigma} - \hat{R}'^L_{i\sigma} a^\dagger_{i\sigma} \right) \\ & + \frac{1}{2} \sum_{ik\in L,\sigma\sigma'} \left(\hat{A}_{ik,\sigma\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + \hat{A}^\dagger_{ik,\sigma\sigma'} \hat{P}^{R\dagger}_{ik,\sigma\sigma'} \right) + \sum_{ij\in L} \hat{B}_{ij} \hat{Q}^R_{ij} - \sum_{il\in L,\sigma\sigma'} \hat{B}'_{il\sigma\sigma'} \hat{Q}'^R_{il\sigma\sigma'}\end{aligned}$$

With this normal/complementary partitioning, the operators required in left block are

$$\left\{ \hat{H}^L, \hat{1}^L, a^\dagger_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^L_{k\sigma}, \hat{A}_{ij,\sigma\sigma'}, \hat{A}^\dagger_{ij,\sigma\sigma'}, \hat{B}_{ij}, \hat{B}'_{ij,\sigma\sigma'} \right\} \quad (i,j \in L,\ k \in R)$$

The operators required in right block are

$$\left\{ \hat{1}^R, \hat{H}^R, \hat{R}'^R_{i\sigma}, \hat{R}'^{R\dagger}_{i\sigma}, a_{k\sigma}, a^\dagger_{k\sigma}, \hat{P}^R_{ij,\sigma\sigma'}, \hat{P}^{R\dagger}_{ij,\sigma\sigma'}, \hat{Q}^R_{ij}, \hat{Q}'^R_{ij,\sigma\sigma'} \right\} \quad (i,j \in L,\ k \in R)$$

Assuming that there are K sites in total, and K_L/K_R sites in left/right block (optimally, $K_L \leq K_R$), the total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{NC} = 1 + 1 + 4K_L + 4K_R + 8K_L^2 + K_L^2 + 4K_L^2 = 13K_L^2 + 4K + 2$$

Complementary/Normal Partitioning

$$\begin{aligned}\hat{H}^{CN} = & \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R + \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{R}'^R_{i\sigma} - a_{i\sigma} \hat{R}'^{R\dagger}_{i\sigma} \right) + \sum_{i\in R,\sigma} \left(\hat{R}'^{L\dagger}_{i\sigma} a_{i\sigma} - \hat{R}'^L_{i\sigma} a^\dagger_{i\sigma} \right) \\ & + \frac{1}{2} \sum_{jl\in R,\sigma\sigma'} \left(\hat{P}^L_{jl,\sigma\sigma'} \hat{A}_{jl,\sigma\sigma'} + \hat{P}^{L\dagger}_{jl,\sigma\sigma'} \hat{A}^\dagger_{jl,\sigma\sigma'} \right) + \sum_{kl\in R} \hat{Q}^L_{kl} \hat{B}_{kl} - \sum_{jk\in R,\sigma\sigma'} \hat{Q}'^L_{jk\sigma\sigma'} \hat{B}'_{jk\sigma\sigma'}\end{aligned}$$

Now the operators required in left block are

$$\left\{ \hat{H}^L, \hat{1}^L, a^\dagger_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^L_{k\sigma}, \hat{P}^L_{kl,\sigma\sigma'}, \hat{P}^{L\dagger}_{kl,\sigma\sigma'}, \hat{Q}^L_{kl}, \hat{Q}'^L_{kl,\sigma\sigma'} \right\} \quad (k,l \in R,\ i \in L)$$

The operators required in right block are

$$\left\{\hat{1}^R, \hat{H}^R, \hat{R}^R_{i\sigma}, \hat{R}^{\prime R\dagger}_{i\sigma}, a_{k\sigma}, a^\dagger_{k\sigma}, \hat{A}_{kl,\sigma\sigma'}, \hat{A}^\dagger_{kl,\sigma\sigma'}, \hat{B}_{kl}, \hat{B}'_{kl,\sigma\sigma'}\right\} \quad (k,l \in R,\ i \in L)$$

The total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{CN} = 1 + 1 + 4K_R + 4K_L + 8K_R^2 + K_R^2 + 4K_R^2 = 13K_R^2 + 4K + 2$$

1.1.3 Blocking

The enlarged left/right block is denoted as $L*/R*$. Make sure that all L operators are to the left of $*$ operators.

$$\begin{aligned}
\hat{R}^{\prime L*}_{i\sigma} &= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{j\in L}\left(\sum_{kl\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} + \sum_{j\in *}\left(\sum_{kl\in L,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in *} v_{ijkl} a_{l\sigma'} a_{j\sigma}\right) + \sum_{k\in *,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in L} v_{ijkl} a_{l\sigma'} a_{j\sigma}\right) - \sum_{l\in L,\sigma'} a_{l\sigma'}\left(\sum_{jk\in *} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}\right) - \sum_{l\in *,\sigma'} a_{l\sigma'}\left(\sum_{jk\in L} \right. \\
&= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{j\in L} a_{j\sigma}\left(\sum_{kl\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}\right) + \sum_{j\in *}\left(\sum_{kl\in L,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in *} v_{ijkl} a_{l\sigma'} a_{j\sigma}\right) + \sum_{k\in *,\sigma'}\left(\sum_{jl\in L} v_{ijkl} a_{l\sigma'} a_{j\sigma}\right) a^\dagger_{k\sigma'} - \sum_{l\in L,\sigma'} a_{l\sigma'}\left(\sum_{jk\in *} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}\right) - \sum_{l\in *,\sigma'}\left(\sum_{jk\in L} v_{ijk}\right.
\end{aligned}$$

Now there are two possibilities. In NC partition, in L we have $A, A^\dagger, B, B'$ and in $*$ we have $P, P^\dagger, Q, Q'$. In CN partition, the opposite is true. Therefore, we have

$$\begin{aligned}
\hat{R}^{\prime L*,NC}_{i\sigma} &= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{j\in L} a_{j\sigma}\hat{Q}^*_{ij} + \sum_{j\in *,kl\in L} v_{ijkl}\hat{B}_{kl} a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\hat{P}^*_{ik,\sigma\sigma'} + \sum_{k\in *,jl\in L,\sigma'} v_{ijkl}\hat{A}^\dagger_{jl,\sigma\sigma'} a^\dagger_{k\sigma'} - \sum_{l\in L,\sigma'} a_{l\sigma'}\hat{Q}^{\prime *}_{il,\sigma\sigma'} - \sum_{l\in *,jk\in L,\sigma'} v_{ijkl}\hat{B}'_{kj,\sigma'\sigma} a_{l\sigma'} \\
&= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\hat{P}^*_{ik,\sigma\sigma'} + \sum_{j\in L} a_{j\sigma}\hat{Q}^*_{ij} - \sum_{l\in L,\sigma'} a_{l\sigma'}\hat{Q}^{\prime *}_{il,\sigma\sigma'} \\
&\quad + \sum_{k\in *,jl\in L,\sigma'} v_{ijkl}\hat{A}^\dagger_{jl,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in *,kl\in L} v_{ijkl}\hat{B}_{kl} a_{j\sigma} - \sum_{l\in *,jk\in L,\sigma'} v_{ijkl}\hat{B}'_{kj,\sigma'\sigma} a_{l\sigma'}
\end{aligned}$$

$$\begin{aligned}
\hat{R}^{\prime L*,CN}_{i\sigma} &= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{j\in L,kl\in *} v_{ijkl} a_{j\sigma}\hat{B}_{kl} + \sum_{j\in *}\hat{Q}^L_{ij} a_{j\sigma} \\
&\quad + \sum_{k\in L,jl\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} + \sum_{k\in *,\sigma'}\hat{P}^L_{ik,\sigma\sigma'} a^\dagger_{k\sigma'} - \sum_{l\in L,jk\in *,\sigma'} v_{ijkl} a_{l\sigma'}\hat{B}'_{kj,\sigma'\sigma} - \sum_{l\in *,\sigma'}\hat{Q}^{\prime L}_{il,\sigma\sigma'} a_{l\sigma'} \\
&= \hat{R}^{\prime L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}^{\prime *}_{i\sigma} + \sum_{k\in L,jl\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} + \sum_{j\in L,kl\in *} v_{ijkl} a_{j\sigma}\hat{B}_{kl} - \sum_{l\in L,jk\in *,\sigma'} v_{ijkl} a_{l\sigma'}\hat{B}'_{kj,\sigma'\sigma} \\
&\quad + \sum_{k\in *,\sigma'}\hat{P}^L_{ik,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in *}\hat{Q}^L_{ij} a_{j\sigma} - \sum_{l\in *,\sigma'}\hat{Q}^{\prime L}_{il,\sigma\sigma'} a_{l\sigma'}
\end{aligned}$$

Similarly,

$$\hat{R}'^{R*,NC}_{i\sigma} = \hat{R}'^{*}_{i\sigma} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{R}'^{R}_{i\sigma} + \sum_{k\in *,\sigma'} a^\dagger_{k\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + \sum_{j\in *} a_{j\sigma} \hat{Q}^R_{ij} - \sum_{l\in *,\sigma'} a_{l\sigma'} \hat{Q}'^{R}_{il,\sigma\sigma'}$$
$$+ \sum_{k\in R,jl\in *,\sigma'} v_{ijkl} \hat{A}^\dagger_{jl,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in R,kl\in *} v_{ijkl} \hat{B}_{kl} a_{j\sigma} - \sum_{l\in R,jk\in *,\sigma'} v_{ijkl} \hat{B}'_{kj,\sigma'\sigma} a_{l\sigma'}$$
$$\hat{R}'^{R*,CN}_{i\sigma} = \hat{R}'^{*}_{i\sigma} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{R}'^{R}_{i\sigma} + \sum_{k\in *,jl\in R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} \hat{A}^\dagger_{jl,\sigma\sigma'} + \sum_{j\in *,kl\in R} v_{ijkl} a_{j\sigma} \hat{B}_{kl} - \sum_{l\in *,jk\in R,\sigma'} v_{ijkl} a_{l\sigma'} \hat{B}'_{kj,\sigma'\sigma}$$
$$+ \sum_{k\in R,\sigma'} \hat{P}^*_{ik,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in R} \hat{Q}^*_{ij} a_{j\sigma} - \sum_{l\in R,\sigma'} \hat{Q}'^{*}_{il,\sigma\sigma'} a_{l\sigma'}$$

Number of terms

$$N_{R',NC} = (2 + 5K_L + 5K_L^2)K_R + (2 + 5 + 5K_R)K_L = 5K_L^2 K_R + 10K_L K_R + 2K + 5K_L$$
$$N_{R',CN} = (2 + 5K_L + 5)K_R + (2 + 5K_R^2 + 5K_R)K_L = 5K_R^2 K_L + 10K_R K_L + 2K + 5K_R$$

Blocking of other complementary operators is straightforward

$$\hat{P}^{L*,CN}_{ik,\sigma\sigma'} = \hat{P}^L_{ik,\sigma\sigma'} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{P}^*_{ik,\sigma\sigma'} + \sum_{j\in L,l\in *} v_{ijkl} a_{l\sigma'} a_{j\sigma} + \sum_{j\in *,l\in L} v_{ijkl} a_{l\sigma'} a_{j\sigma}$$
$$= \hat{P}^L_{ik,\sigma\sigma'} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{P}^*_{ik,\sigma\sigma'} - \sum_{j\in L,l\in *} v_{ijkl} a_{j\sigma} a_{l\sigma'} + \sum_{j\in *,l\in L} v_{ijkl} a_{l\sigma'} a_{j\sigma}$$
$$\hat{P}^{R*,NC}_{ik,\sigma\sigma'} = \hat{P}^*_{ik,\sigma\sigma'} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{P}^R_{ik,\sigma\sigma'} + \sum_{j\in *,l\in R} v_{ijkl} a_{l\sigma'} a_{j\sigma} + \sum_{j\in R,l\in *} v_{ijkl} a_{l\sigma'} a_{j\sigma}$$
$$= \hat{P}^*_{ik,\sigma\sigma'} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{P}^R_{ik,\sigma\sigma'} - \sum_{j\in *,l\in R} v_{ijkl} a_{j\sigma} a_{l\sigma'} + \sum_{j\in R,l\in *} v_{ijkl} a_{l\sigma'} a_{j\sigma}$$

and

$$\hat{Q}^{L*,CN}_{ij} = \hat{Q}^L_{ij} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{Q}^*_{ij} + \sum_{k\in L,l\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} + \sum_{k\in *,l\in L,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}$$
$$= \hat{Q}^L_{ij} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{Q}^*_{ij} + \sum_{k\in L,l\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} - \sum_{k\in *,l\in L,\sigma'} v_{ijkl} a_{l\sigma'} a^\dagger_{k\sigma'}$$
$$\hat{Q}^{R*,NC}_{ij} = \hat{Q}^*_{ij} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{Q}^R_{ij} + \sum_{k\in *,l\in R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} + \sum_{k\in R,l\in *,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'}$$
$$= \hat{Q}^*_{ij} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{Q}^R_{ij} + \sum_{k\in *,l\in R,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} - \sum_{k\in R,l\in *,\sigma'} v_{ijkl} a_{l\sigma'} a^\dagger_{k\sigma'}$$

and

$$\hat{Q}'^{L*,CN}_{il,\sigma\sigma'} = \hat{Q}'^{L}_{il,\sigma\sigma'} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{Q}'^{*}_{il,\sigma\sigma'} + \sum_{j\in L,k\in *} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma} + \sum_{j\in *,k\in L} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}$$
$$= \hat{Q}'^{L}_{il,\sigma\sigma'} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{Q}'^{*}_{il,\sigma\sigma'} - \sum_{j\in L,k\in *} v_{ijkl} a_{j\sigma} a^\dagger_{k\sigma'} + \sum_{j\in *,k\in L} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}$$
$$\hat{Q}'^{R*,NC}_{il,\sigma\sigma'} = \hat{Q}'^{*}_{il,\sigma\sigma'} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{Q}'^{R}_{il,\sigma\sigma'} + \sum_{j\in *,k\in R} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma} + \sum_{j\in R,k\in *} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}$$
$$= \hat{Q}'^{*}_{il,\sigma\sigma'} \otimes \hat{1}^R + \hat{1}^* \otimes \hat{Q}'^{R}_{il,\sigma\sigma'} - \sum_{j\in *,k\in R} v_{ijkl} a_{j\sigma} a^\dagger_{k\sigma'} + \sum_{j\in R,k\in *} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma}$$

1.1.4 Middle-Site Transformation

When the sweep is performed from left to right, passing the middle site, we need to switch from NC partition to CN partition. The cost is $O(K^4/16)$. This happens only once in the sweep. The cost of one blocking procedure is

$O(K_<^2 K_>)$, but there are K blocking steps in one sweep. So the cost for blocking in one sweep is $O(K K_<^2 K_>)$. Note that the most expensive part in the program should be the Hamiltonian step in Davidson, which scales as $O(K_<^2)$.

$$\hat{P}^{L,NC\to CN}_{ik,\sigma\sigma'} = \sum_{jl\in L} v_{ijkl} a_{l\sigma'} a_{j\sigma} = \sum_{jl\in L} v_{ijkl} \hat{A}^\dagger_{jl,\sigma\sigma'}$$
$$\hat{Q}^{L,NC\to CN}_{ij} = \sum_{kl\in L,\sigma'} v_{ijkl} a^\dagger_{k\sigma'} a_{l\sigma'} = \sum_{kl\in L} v_{ijkl} \hat{B}_{kl}$$
$$\hat{Q}'^{L,NC\to CN}_{il,\sigma\sigma'} = \sum_{jk\in L} v_{ijkl} a^\dagger_{k\sigma'} a_{j\sigma} = \sum_{jk\in L} v_{ijkl} \hat{B}'_{kj,\sigma'\sigma}$$

1.2 DMRG Quantum Chemistry Hamiltonian in Unrestricted Spatial Orbitals

1.2.1 Hamiltonian

The quantum chemistry Hamiltonian is written as follows

$$\hat{H} = \sum_{ij,\sigma} t_{ij,\sigma}\, a^\dagger_{i\sigma} a_{j\sigma} + \frac{1}{2} \sum_{ijkl,\sigma\sigma'} v_{ijkl,\sigma\sigma'}\, a^\dagger_{i\sigma} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}$$

where

$$t_{ij,\sigma} = t_{(ij),\sigma} = \int \mathrm{d}x\, \phi^*_{i\sigma}(x) \left(-\frac{1}{2}\nabla^2 - \sum_a \frac{Z_a}{r_a} \right) \phi_{j\sigma}(x)$$
$$v_{ijkl,\sigma\sigma'} = v_{(ij)(kl),\sigma\sigma'} = v_{(kl)(ij),\sigma\sigma'} = \int \mathrm{d}x_1 \mathrm{d}x_2\, \frac{\phi^*_{i\sigma}(x_1)\phi^*_{k\sigma'}(x_2)\phi_{l\sigma'}(x_2)\phi_{j\sigma}(x_1)}{r_{12}}$$

Note that here the order of $ijkl$ is the same as that in `FCIDUMP` (chemist's notation $[ij|kl]$).

1.2.2 Partitioning in Spatial Orbitals

The partitioning of Hamiltonian in left (L) and right (R) blocks is given by

$$\begin{aligned}\hat{H} = {} & \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R \\ & + \Big(\sum_{i\in L,\sigma} a^\dagger_{i\sigma} \hat{S}^R_{i\sigma} + h.c. \Big) + \Big(\sum_{i\in L,\sigma} a^\dagger_{i\sigma} \hat{R}^R_{i\sigma} + h.c. + \sum_{i\in R,\sigma} a^\dagger_{i\sigma} \hat{R}^L_{i\sigma} + h.c. \Big) \\ & + \frac{1}{2} \Big(\sum_{ik\in L,\sigma\sigma'} \hat{A}^L_{ik,\sigma\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + h.c. \Big) + \sum_{ij\in L,\sigma} \hat{B}_{ij\sigma} \hat{Q}^R_{ij\sigma} - \sum_{il\in L,\sigma\sigma'} \hat{B}'_{il\sigma\sigma'} \hat{Q}'^R_{il\sigma\sigma'}\end{aligned}$$

where the normal and complementary operators are defined by

$$\begin{aligned}
\hat{S}^{L/R}_{i\sigma} &= \sum_{j\in L/R} t_{ij,\sigma} a_{j\sigma}, \\
\hat{R}^{L/R}_{i\sigma} &= \sum_{jkl\in L/R,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}, \\
\hat{A}_{ik,\sigma\sigma'} &= a^\dagger_{i\sigma} a^\dagger_{k\sigma'}, \\
\hat{B}_{ij,\sigma} &= a^\dagger_{i\sigma} a_{j\sigma}, \\
\hat{B}'_{il,\sigma\sigma'} &= a^\dagger_{i\sigma} a_{l\sigma'}, \\
\hat{P}^R_{ik,\sigma\sigma'} &= \sum_{jl\in R} v_{ijkl,\sigma\sigma'} a_{l\sigma'} a_{j\sigma}, \\
\hat{Q}^R_{ij,\sigma} &= \sum_{kl\in R,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'}, \\
\hat{Q}'^R_{il,\sigma\sigma'} &= \sum_{jk\in R} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{j\sigma}
\end{aligned}$$

Note that we need to move all on-site interaction into local Hamiltonian, so that when construction interaction terms in Hamiltonian, operators anticommute (without giving extra constant terms).

Define

$$\hat{R}'^{L/R}_{i\sigma} = \frac{1}{2}\hat{S}^{L/R}_{i\sigma} + \hat{R}^{L/R}_{i\sigma} = \frac{1}{2}\sum_{j\in L/R} t_{ij,\sigma} a_{j\sigma} + \sum_{jkl\in L/R,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'} a_{j\sigma}$$

Then we have

$$\begin{aligned}
\hat{H}^{NC} = \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R &+ \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{R}'^R_{i\sigma} - a_{i\sigma} \hat{R}'^{R\dagger}_{i\sigma} \right) + \sum_{i\in R,\sigma} \left(\hat{R}'^{L\dagger}_{i\sigma} a_{i\sigma} - \hat{R}'^L_{i\sigma} a^\dagger_{i\sigma} \right) \\
&+ \frac{1}{2} \sum_{ik\in L,\sigma\sigma'} \left(\hat{A}_{ik,\sigma\sigma'} \hat{P}^R_{ik,\sigma\sigma'} + \hat{A}^\dagger_{ik,\sigma\sigma'} \hat{P}^{R\dagger}_{ik,\sigma\sigma'} \right) + \sum_{ij\in L,\sigma} \hat{B}_{ij,\sigma} \hat{Q}^R_{ij,\sigma} - \sum_{il\in L,\sigma\sigma'} \hat{B}'_{il\sigma\sigma'} \hat{Q}'^R_{il\sigma\sigma'}
\end{aligned}$$

Normal/Complementary Partitioning

With this normal/complementary partitioning, the operators required in left block are

$$\left\{ \hat{H}^L, \hat{1}^L, a^\dagger_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^L_{k\sigma}, \hat{A}_{ij,\sigma\sigma'}, \hat{A}^\dagger_{ij,\sigma\sigma'}, \hat{B}_{ij,\sigma}, \hat{B}'_{ij,\sigma\sigma'} \right\} \quad (i,j \in L,\ k \in R)$$

The operators required in right block are

$$\left\{ \hat{1}^R, \hat{H}^R, \hat{R}'^R_{i\sigma}, \hat{R}'^{R\dagger}_{i\sigma}, a_{k\sigma}, a^\dagger_{k\sigma}, \hat{P}^R_{ij,\sigma\sigma'}, \hat{P}^{R\dagger}_{ij,\sigma\sigma'}, \hat{Q}^R_{ij,\sigma}, \hat{Q}'^R_{ij,\sigma\sigma'} \right\} \quad (i,j \in L,\ k \in R)$$

Assuming that there are K sites in total, and K_L/K_R sites in left/right block (optimally, $K_L \leq K_R$), the total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{NC} = 1 + 1 + 4K_L + 4K_R + 8K_L^2 + 2K_L^2 + 4K_L^2 = 14K_L^2 + 4K + 2$$

Complementary/Normal Partitioning

$$\begin{aligned}
\hat{H}^{CN} = \hat{H}^L \otimes \hat{1}^R + \hat{1}^L \otimes \hat{H}^R &+ \sum_{i\in L,\sigma} \left(a^\dagger_{i\sigma} \hat{R}'^R_{i\sigma} - a_{i\sigma} \hat{R}'^{R\dagger}_{i\sigma} \right) + \sum_{i\in R,\sigma} \left(\hat{R}'^{L\dagger}_{i\sigma} a_{i\sigma} - \hat{R}'^L_{i\sigma} a^\dagger_{i\sigma} \right) \\
&+ \frac{1}{2} \sum_{jl\in R,\sigma\sigma'} \left(\hat{P}^L_{jl,\sigma\sigma'} \hat{A}_{jl,\sigma\sigma'} + \hat{P}^{L\dagger}_{jl,\sigma\sigma'} \hat{A}^\dagger_{jl,\sigma\sigma'} \right) + \sum_{kl\in R,\sigma} \hat{Q}^L_{kl,\sigma} \hat{B}_{kl,\sigma} - \sum_{jk\in R,\sigma\sigma'} \hat{Q}'^L_{jk\sigma\sigma'} \hat{B}'_{jk\sigma\sigma'}
\end{aligned}$$

Now the operators required in left block are

$$\left\{\hat{H}^L, \hat{1}^L, a^\dagger_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^{L}_{k\sigma}, \hat{P}^L_{kl,\sigma\sigma'}, \hat{P}^{L\dagger}_{kl,\sigma\sigma'}, \hat{Q}^L_{kl,\sigma}, \hat{Q}'^L_{kl,\sigma\sigma'}\right\} \quad (k,l \in R,\ i \in L)$$

The operators required in right block are

$$\left\{\hat{1}^R, \hat{H}^R, \hat{R}'^R_{i\sigma}, \hat{R}'^{R\dagger}_{i\sigma}, a_{k\sigma}, a^\dagger_{k\sigma}, \hat{A}_{kl,\sigma\sigma'}, \hat{A}^\dagger_{kl,\sigma\sigma'}, \hat{B}_{kl,\sigma}, \hat{B}'_{kl,\sigma\sigma'}\right\} \quad (k,l \in R,\ i \in L)$$

The total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{CN} = 1 + 1 + 4K_R + 4K_L + 8K_R^2 + 2K_R^2 + 4K_R^2 = 14K_R^2 + 4K + 2$$

1.2.3 Blocking

The enlarged left/right block is denoted as $L*/R*$. Make sure that all L operators are to the left of $*$ operators.

$$\begin{aligned}
\hat{R}'^{L*}_{i\sigma} &= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{j\in L}\left(\sum_{kl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} + \sum_{j\in *}\left(\sum_{kl\in L,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in *} v_{ijkl,\sigma\sigma'} a_{l\sigma'} a_{j\sigma}\right) + \sum_{k\in *,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in L} v_{ijkl,\sigma\sigma'} a_{l\sigma'} a_{j\sigma}\right) - \sum_{l\in L,\sigma'} a_{l\sigma'}\left(\sum_{jk\in *} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{j\sigma}\right) - \sum_{l\in *,\sigma'} \\
&= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{j\in L} a_{j\sigma}\left(\sum_{kl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'}\right) + \sum_{j\in *}\left(\sum_{kl\in L,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{l\sigma'}\right) a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\left(\sum_{jl\in *} v_{ijkl,\sigma\sigma'} a_{l\sigma'} a_{j\sigma}\right) + \sum_{k\in *,\sigma'}\left(\sum_{jl\in L} v_{ijkl,\sigma\sigma'} a_{l\sigma'} a_{j\sigma}\right) a^\dagger_{k\sigma'} - \sum_{l\in L,\sigma'} a_{l\sigma'}\left(\sum_{jk\in *} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'} a_{j\sigma}\right) - \sum_{l\in *,\sigma'}
\end{aligned}$$

Now there are two possibilities. In NC partition, in L we have $A, A^\dagger, B, B'$ and in $*$ we have $P, P^\dagger, Q, Q'$. In CN partition, the opposite is true. Therefore, we have

$$\begin{aligned}
\hat{R}'^{L*,NC}_{i\sigma} &= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{j\in L} a_{j\sigma}\hat{Q}^*_{ij,\sigma} + \sum_{j\in *,kl\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{B}_{kl,\sigma'} a_{j\sigma} \\
&\quad + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\hat{P}^*_{ik,\sigma\sigma'} + \sum_{k\in *,jl\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} a^\dagger_{k\sigma'} - \sum_{l\in L,\sigma'} a_{l\sigma'}\hat{Q}'^*_{il,\sigma\sigma'} - \sum_{l\in *,jk\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{B}'_{kj,\sigma'\sigma} a_{l\sigma'} \\
&= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{k\in L,\sigma'} a^\dagger_{k\sigma'}\hat{P}^*_{ik,\sigma\sigma'} + \sum_{j\in L} a_{j\sigma}\hat{Q}^*_{ij,\sigma} - \sum_{l\in L,\sigma'} a_{l\sigma'}\hat{Q}'^*_{il,\sigma\sigma'} \\
&\quad + \sum_{k\in *,jl\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in *,kl\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{B}_{kl,\sigma'} a_{j\sigma} - \sum_{l\in *,jk\in L,\sigma'} v_{ijkl,\sigma\sigma'}\hat{B}'_{kj,\sigma'\sigma} a_{l\sigma'}
\end{aligned}$$

$$\begin{aligned}
\hat{R}'^{L*,CN}_{i\sigma} &= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{j\in L,kl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a_{j\sigma}\hat{B}_{kl,\sigma'} + \sum_{j\in *}\hat{Q}^L_{ij,\sigma} a_{j\sigma} \\
&\quad + \sum_{k\in L,jl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} + \sum_{k\in *,\sigma'}\hat{P}^L_{ik,\sigma\sigma'} a^\dagger_{k\sigma'} - \sum_{l\in L,jk\in *,\sigma'} v_{ijkl,\sigma\sigma'} a_{l\sigma'}\hat{B}'_{kj,\sigma'\sigma} - \sum_{l\in *,\sigma'}\hat{Q}'^L_{il,\sigma\sigma'} a_{l\sigma'} \\
&= \hat{R}'^{L}_{i\sigma} \otimes \hat{1}^* + \hat{1}^L \otimes \hat{R}'^{*}_{i\sigma} + \sum_{k\in L,jl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a^\dagger_{k\sigma'}\hat{A}^\dagger_{jl,\sigma\sigma'} + \sum_{j\in L,kl\in *,\sigma'} v_{ijkl,\sigma\sigma'} a_{j\sigma}\hat{B}_{kl,\sigma'} - \sum_{l\in L,jk\in *,\sigma'} v_{ijkl,\sigma\sigma'} a_{l\sigma'}\hat{B}'_{kj,\sigma'\sigma} \\
&\quad + \sum_{k\in *,\sigma'}\hat{P}^L_{ik,\sigma\sigma'} a^\dagger_{k\sigma'} + \sum_{j\in *}\hat{Q}^L_{ij,\sigma} a_{j\sigma} - \sum_{l\in *,\sigma'}\hat{Q}'^L_{il,\sigma\sigma'} a_{l\sigma'}
\end{aligned}$$

1.2.4 Simplified Form

Define

$$\hat{Q}''^{R}_{ij,\sigma\sigma'} = \delta_{\sigma\sigma'}\hat{Q}^{R}_{ij\sigma} - \hat{Q}'^{R}_{ij\sigma\sigma'}$$

we have N/C form

$$\hat{H}^{NC} = \hat{H}^{L}\otimes\hat{1}^{R} + \hat{1}^{L}\otimes\hat{H}^{R} + \sum_{i\in L,\sigma}\left(a^{\dagger}_{i\sigma}\hat{R}'^{R}_{i\sigma} - a_{i\sigma}\hat{R}'^{R\dagger}_{i\sigma}\right) + \sum_{i\in R,\sigma}\left(\hat{R}'^{L\dagger}_{i\sigma}a_{i\sigma} - \hat{R}'^{L}_{i\sigma}a^{\dagger}_{i\sigma}\right)$$
$$+ \frac{1}{2}\sum_{ik\in L,\sigma\sigma'}\left(\hat{A}_{ik,\sigma\sigma'}\hat{P}^{R}_{ik,\sigma\sigma'} + \hat{A}^{\dagger}_{ik,\sigma\sigma'}\hat{P}^{R\dagger}_{ik,\sigma\sigma'}\right) + \sum_{ij\in L,\sigma\sigma'}\hat{B}'_{ij\sigma\sigma'}\hat{Q}''^{R}_{ij\sigma\sigma'}$$

With this normal/complementary partitioning, the operators required in left block are

$$\left\{\hat{H}^{L}, \hat{1}^{L}, a^{\dagger}_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^{L}_{k\sigma}, \hat{A}_{ij,\sigma\sigma'}, \hat{A}^{\dagger}_{ij,\sigma\sigma'}, \hat{B}'_{ij,\sigma\sigma'}\right\} \quad (i,j\in L,\ k\in R)$$

The operators required in right block are

$$\left\{\hat{1}^{R}, \hat{H}^{R}, \hat{R}'^{R}_{i\sigma}, \hat{R}'^{R\dagger}_{i\sigma}, a_{k\sigma}, a^{\dagger}_{k\sigma}, \hat{P}^{R}_{ij,\sigma\sigma'}, \hat{P}^{R\dagger}_{ij,\sigma\sigma'}, \hat{Q}''^{R}_{ij,\sigma\sigma'}\right\} \quad (i,j\in L,\ k\in R)$$

Assuming that there are K sites in total, and K_L/K_R sites in left/right block (optimally, $K_L \le K_R$), the total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{NC} = 1 + 1 + 4K_L + 4K_R + 8K_L^2 + 4K_L^2 = 12K_L^2 + 4K + 2$$

and C/N form

$$\hat{H}^{CN} = \hat{H}^{L}\otimes\hat{1}^{R} + \hat{1}^{L}\otimes\hat{H}^{R} + \sum_{i\in L,\sigma}\left(a^{\dagger}_{i\sigma}\hat{R}'^{R}_{i\sigma} - a_{i\sigma}\hat{R}'^{R\dagger}_{i\sigma}\right) + \sum_{i\in R,\sigma}\left(\hat{R}'^{L\dagger}_{i\sigma}a_{i\sigma} - \hat{R}'^{L}_{i\sigma}a^{\dagger}_{i\sigma}\right)$$
$$+ \frac{1}{2}\sum_{jl\in R,\sigma\sigma'}\left(\hat{P}^{L}_{jl,\sigma\sigma'}\hat{A}_{jl,\sigma\sigma'} + \hat{P}^{L\dagger}_{jl,\sigma\sigma'}\hat{A}^{\dagger}_{jl,\sigma\sigma'}\right) + \sum_{kl\in R,\sigma\sigma'}\hat{Q}''^{L}_{kl\sigma\sigma'}\hat{B}'_{kl\sigma\sigma'}$$

Now the operators required in left block are

$$\left\{\hat{H}^{L}, \hat{1}^{L}, a^{\dagger}_{i\sigma}, a_{i\sigma}, \hat{R}'^{L\dagger}_{k\sigma}, \hat{R}'^{L}_{k\sigma}, \hat{P}^{L}_{kl,\sigma\sigma'}, \hat{P}^{L\dagger}_{kl,\sigma\sigma'}, \hat{Q}''^{L}_{kl,\sigma\sigma'}\right\} \quad (k,l\in R,\ i\in L)$$

The operators required in right block are

$$\left\{\hat{1}^{R}, \hat{H}^{R}, \hat{R}'^{R}_{i\sigma}, \hat{R}'^{R\dagger}_{i\sigma}, a_{k\sigma}, a^{\dagger}_{k\sigma}, \hat{A}_{kl,\sigma\sigma'}, \hat{A}^{\dagger}_{kl,\sigma\sigma'}, \hat{B}'_{kl,\sigma\sigma'}\right\} \quad (k,l\in R,\ i\in L)$$

The total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$N_{CN} = 1 + 1 + 4K_R + 4K_L + 8K_R^2 + 4K_R^2 = 12K_R^2 + 4K + 2$$

Then for blocking

$$\hat{R}'^{L*,NC}_{i\sigma} = \hat{R}'^{L}_{i\sigma}\otimes\hat{1}^{*} + \hat{1}^{L}\otimes\hat{R}'^{*}_{i\sigma} + \sum_{k\in L,\sigma'}a^{\dagger}_{k\sigma'}\hat{P}^{*}_{ik,\sigma\sigma'} + \sum_{j\in L,\sigma'}a_{j\sigma'}\hat{Q}''^{*}_{ij,\sigma\sigma'}$$
$$+ \sum_{k\in *,jl\in L,\sigma'}v_{ijkl,\sigma\sigma'}\hat{A}^{\dagger}_{jl,\sigma\sigma'}a^{\dagger}_{k\sigma'} + \sum_{j\in *,kl\in L,\sigma'}v_{ijkl,\sigma\sigma'}\hat{B}'_{kl,\sigma'\sigma'}a_{j\sigma} - \sum_{l\in *,jk\in L,\sigma'}v_{ijkl,\sigma\sigma'}\hat{B}'_{kj,\sigma'\sigma}a_{l\sigma'}$$

$$\hat{R}'^{L*,CN}_{i\sigma} = \hat{R}'^{L}_{i\sigma}\otimes\hat{1}^{*} + \hat{1}^{L}\otimes\hat{R}'^{*}_{i\sigma} + \sum_{k\in L,jl\in *,\sigma'}v_{ijkl,\sigma\sigma'}a^{\dagger}_{k\sigma'}\hat{A}^{\dagger}_{jl,\sigma\sigma'} + \sum_{j\in L,kl\in *,\sigma'}v_{ijkl,\sigma\sigma'}a_{j\sigma}\hat{B}'_{kl,\sigma'\sigma'}$$
$$- \sum_{l\in L,jk\in *,\sigma'}v_{ijkl,\sigma\sigma'}a_{l\sigma'}\hat{B}'_{kj,\sigma'\sigma} + \sum_{k\in *,\sigma'}\hat{P}^{L}_{ik,\sigma\sigma'}a^{\dagger}_{k\sigma'} + \sum_{j\in *,\sigma'}\hat{Q}'^{L}_{ij,\sigma\sigma'}a_{j\sigma'}$$

1.3 Spin-Adapted DMRG Quantum Chemistry Hamiltonian

1.3.1 Partitioning in SU(2)

The partitioning of Hamiltonian in left (L) and right (R) blocks is given by

$$\begin{aligned}
(\hat{H})^{[0]} = &\left(\hat{H}^L\right)^{[0]} \otimes_{[0]} \left(\hat{1}^R\right)^{[0]} + \left(\hat{1}^L\right)^{[0]} \otimes_{[0]} \left(\hat{H}^R\right)^{[0]} \\
&+ \sqrt{2} \sum_{i \in L} \left[\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{S}_i^R\right)^{[\frac{1}{2}]} + h.c. \right] \\
&+ 2 \sum_{i \in L} \left[\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i^R\right)^{[\frac{1}{2}]} + h.c. \right] + 2 \sum_{i \in R} \left[\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i^L\right)^{[\frac{1}{2}]} + h.c. \right] \\
&- \frac{1}{2} \sum_{ik \in L} \left[\sqrt{3} \left(\hat{A}_{ik}\right)^{[1]} \otimes_{[0]} \left(\hat{P}_{ik}^R\right)^{[1]} + \left(\hat{A}_{ik}\right)^{[0]} \otimes_{[0]} \left(\hat{P}_{ik}^R\right)^{[0]} + h.c. \right] \\
&+ \sum_{ij \in L} \left[\left(\hat{B}_{ij}\right)^{[0]} \otimes_{[0]} \left(2 \left(\hat{Q}_{ij}^R\right)^{[0]} - \left(\hat{Q}_{ij}^{\prime R}\right)^{[0]} \right) + \sqrt{3} \left(\hat{B}^\prime_{ij}\right)^{[1]} \otimes_{[0]} \left(\hat{Q}_{ij}^{\prime R}\right)^{[1]} \right]
\end{aligned}$$

where the normal and complementary operators are defined by

$$\begin{aligned}
\left(\hat{S}_i^{L/R}\right)^{[\frac{1}{2}]} &= \sum_{j \in L/R} t_{ij} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{R}_i^{L/R}\right)^{[\frac{1}{2}]} &= \sum_{jkl \in L/R} v_{ijkl} \left[\left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]} \right] \otimes_{[\frac{1}{2}]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{A}_{ik}\right)^{[0/1]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \\
\left(\hat{P}_{ik}^R\right)^{[0/1]} &= \sum_{jl \in R} v_{ijkl} \left(a_j\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_l\right)^{[\frac{1}{2}]} \\
\left(\hat{B}_{ij}\right)^{[0]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{B}^\prime_{ij}\right)^{[1]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[1]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{Q}_{ij}^R\right)^{[0]} &= \sum_{kl \in R} v_{ijkl} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]} \\
\left(\hat{Q}_{ij}^{\prime R}\right)^{[0/1]} &= \sum_{kl \in R} v_{ilkj} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_l\right)^{[\frac{1}{2}]} \\
\left(\hat{Q}_{ij}^{\prime\prime R}\right)^{[0]} &:= 2 \left(\hat{Q}_{ij}^R\right)^{[0]} - \left(\hat{Q}_{ij}^{\prime R}\right)^{[0]} = \sum_{kl \in R} \left(2 v_{ijkl} - v_{ilkj}\right) \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}
\end{aligned}$$

Derivation

CG Factors

From $j_2 = 1/2$ CG factors

$$\left\langle j_1 \ \left(M - \frac{1}{2}\right) \ \frac{1}{2} \ \frac{1}{2} \middle| \left(j_1 \pm \frac{1}{2}\right) \ M \right\rangle = \pm \sqrt{\frac{1}{2} \left(1 \pm \frac{M}{j_1 + \frac{1}{2}}\right)}$$

$$\left\langle j_1 \ \left(M + \frac{1}{2}\right) \ \frac{1}{2} \ \left(-\frac{1}{2}\right) \middle| \left(j_1 \pm \frac{1}{2}\right) \ M \right\rangle = \sqrt{\frac{1}{2} \left(1 \mp \frac{M}{j_1 + \frac{1}{2}}\right)}$$

and symmetry relation

$$\langle j_1\ m_1\ j_2\ m_2|J\ M\rangle = (-1)^{j_1+j_2-J}\langle j_2\ m_2\ j_1\ m_1|J\ M\rangle$$

and

$$(-1)^{j_1+\frac{1}{2}-j_1\mp\frac{1}{2}} = (-1)^{\frac{1}{2}\mp\frac{1}{2}} = \pm 1$$

we have

$$\left\langle \frac{1}{2}\ \frac{1}{2}\ j_1\ \left(M-\frac{1}{2}\right) \middle| \left(j_1\pm\frac{1}{2}\right)\ M\right\rangle = \sqrt{\frac{1}{2}\left(1\pm\frac{M}{j_1+\frac{1}{2}}\right)}$$
$$\left\langle \frac{1}{2}\ \left(-\frac{1}{2}\right)\ j_1\ \left(M+\frac{1}{2}\right) \middle| \left(j_1\pm\frac{1}{2}\right)\ M\right\rangle = \pm\sqrt{\frac{1}{2}\left(1\mp\frac{M}{j_1+\frac{1}{2}}\right)}$$

let $j_1 = 1$, we have

$$\langle \tfrac{1}{2}\ \tfrac{1}{2}\ 1\ (M-\tfrac{1}{2})|\tfrac{1}{2}\ M\rangle = \sqrt{\tfrac{1}{2}(1-\frac{M}{\frac{3}{2}})}$$
$$\langle \tfrac{1}{2}\ (-\tfrac{1}{2})\ 1\ (M+\tfrac{1}{2})|\tfrac{1}{2}\ M\rangle = -\sqrt{\tfrac{1}{2}(1+\frac{M}{\frac{3}{2}})}$$

So the coefficients for $[\frac{1}{2}]\otimes_{[\frac{1}{2}]}[1]$ are

$$[\tfrac{1}{2}+0=\tfrac{1}{2}] = \sqrt{\tfrac{1}{3}},\quad [-\tfrac{1}{2}+1=\tfrac{1}{2}] = -\sqrt{\tfrac{2}{3}}$$
$$[\tfrac{1}{2}+(-1)=-\tfrac{1}{2}] = \sqrt{\tfrac{2}{3}},\quad [-\tfrac{1}{2}+0=-\tfrac{1}{2}] = -\sqrt{\tfrac{1}{3}}$$

The coefficients for $[1]\otimes_{[\frac{1}{2}]}[\frac{1}{2}]$ are

$$[0+\tfrac{1}{2}=\tfrac{1}{2}] = -\sqrt{\tfrac{1}{3}},\quad [1-\tfrac{1}{2}=\tfrac{1}{2}] = \sqrt{\tfrac{2}{3}}$$
$$[(-1)+\tfrac{1}{2}=-\tfrac{1}{2}] = -\sqrt{\tfrac{2}{3}},\quad [0-\tfrac{1}{2}=-\tfrac{1}{2}] = \sqrt{\tfrac{1}{3}}$$

This means that the SU(2) operator exchange factor for $[\frac{1}{2}]\otimes_{[\frac{1}{2}]}[1]\to[1]\otimes_{[\frac{1}{2}]}[\frac{1}{2}]$ is -1. The fermion factor is $+1$. So the overall exchange factor for this case is -1.

Tensor Product Formulas

Singlet

$$\left(a_p^\dagger\right)^{[1/2]}\otimes_{[0]}\left(a_q^\dagger\right)^{[1/2]} = \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger\end{pmatrix}^{[1/2]}\otimes_{[0]}\begin{pmatrix} a_{q\alpha}^\dagger \\ a_{q\beta}^\dagger\end{pmatrix}^{[1/2]} = \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\beta}^\dagger - a_{p\beta}^\dagger a_{q\alpha}^\dagger\right)^{[0]}$$
$$\left(a_p^\dagger\right)^{[1/2]}\otimes_{[0]}\left(a_q\right)^{[1/2]} = \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger\end{pmatrix}^{[1/2]}\otimes_{[0]}\begin{pmatrix} -a_{q\beta} \\ a_{q\alpha}\end{pmatrix}^{[1/2]} = \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\alpha} + a_{p\beta}^\dagger a_{q\beta}\right)^{[0]}$$
$$\left(a_p\right)^{[1/2]}\otimes_{[0]}\left(a_q\right)^{[1/2]} = \begin{pmatrix} -a_{p\beta} \\ a_{p\alpha}\end{pmatrix}^{[1/2]}\otimes_{[0]}\begin{pmatrix} -a_{q\beta} \\ a_{q\alpha}\end{pmatrix}^{[1/2]} = \frac{1}{\sqrt{2}}\left(-a_{p\beta}a_{q\alpha} + a_{p\alpha}a_{q\beta}\right)^{[0]}$$

Triplet

$$\left(a_p^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_q^\dagger\right)^{[1/2]} = \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger \end{pmatrix}^{[1/2]} \otimes_{[1]} \begin{pmatrix} a_{q\alpha}^\dagger \\ a_{q\beta}^\dagger \end{pmatrix}^{[1/2]} = \begin{pmatrix} a_{p\alpha}^\dagger a_{q\alpha}^\dagger \\ \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\beta}^\dagger + a_{p\beta}^\dagger a_{q\alpha}^\dagger\right) \\ a_{p\beta}^\dagger a_{q\beta}^\dagger \end{pmatrix}^{[1]}$$

$$\left(a_p^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_q\right)^{[1/2]} = \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger \end{pmatrix}^{[1/2]} \otimes_{[1]} \begin{pmatrix} -a_{q\beta} \\ a_{q\alpha} \end{pmatrix}^{[1/2]} = \begin{pmatrix} -a_{p\alpha}^\dagger a_{q\beta} \\ \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\alpha} - a_{p\beta}^\dagger a_{q\beta}\right) \\ a_{p\beta}^\dagger a_{q\alpha} \end{pmatrix}^{[1]}$$

$$\left(a_p\right)^{[1/2]} \otimes_{[1]} \left(a_q\right)^{[1/2]} = \begin{pmatrix} -a_{p\beta} \\ a_{p\alpha} \end{pmatrix}^{[1/2]} \otimes_{[1]} \begin{pmatrix} -a_{q\beta} \\ a_{q\alpha} \end{pmatrix}^{[1/2]} = \begin{pmatrix} a_{p\beta} a_{q\beta} \\ -\frac{1}{\sqrt{2}}\left(a_{p\beta} a_{q\alpha} + a_{p\alpha} a_{q\beta}\right) \\ a_{p\alpha} a_{q\alpha} \end{pmatrix}^{[1]}$$

Doublet times singlet/triplet

$$U^{[1/2]} = \left(a_p^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_r\right)^{[1/2]} \otimes_{[1]} \left(a_s\right)^{[1/2]}\right] = \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger \end{pmatrix}^{[1/2]} \otimes_{[1/2]} \begin{pmatrix} a_{r\beta} a_{s\beta} \\ -\frac{1}{\sqrt{2}}\left(a_{r\beta} a_{s\alpha} + a_{r\alpha} a_{s\beta}\right) \\ a_{r\alpha} a_{s\alpha} \end{pmatrix}^{[1]}$$

$$= \begin{pmatrix} -\frac{1}{\sqrt{2}}\frac{1}{\sqrt{3}} a_{p\alpha}^\dagger \left(a_{r\beta} a_{s\alpha} + a_{r\alpha} a_{s\beta}\right) - \frac{\sqrt{2}}{\sqrt{3}} a_{p\beta}^\dagger a_{r\beta} a_{s\beta} \\ \frac{\sqrt{2}}{\sqrt{3}} a_{p\alpha}^\dagger a_{r\alpha} a_{s\alpha} + \left(-\frac{1}{\sqrt{3}}\right)\left(-\frac{1}{\sqrt{2}}\right) a_{p\beta}^\dagger \left(a_{r\beta} a_{s\alpha} + a_{r\alpha} a_{s\beta}\right) \end{pmatrix}^{[1/2]} = \frac{1}{\sqrt{6}} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} - a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} - 2a_{p\beta}^\dagger a_{r\beta} a_{s\beta} \\ 2a_{p\alpha}^\dagger a_{r\alpha} a_{s\alpha} + a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} + a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} \end{pmatrix}^{[1/2]}$$

$$V^{[1/2]} = \left(a_p^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_r\right)^{[1/2]} \otimes_{[0]} \left(a_s\right)^{[1/2]}\right] = \frac{1}{\sqrt{2}} \begin{pmatrix} a_{p\alpha}^\dagger \\ a_{p\beta}^\dagger \end{pmatrix}^{[1/2]} \otimes_{[1/2]} \left(-a_{r\beta} a_{s\alpha} + a_{r\alpha} a_{s\beta}\right)^{[0]}$$

$$= \frac{1}{\sqrt{2}} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} + a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} \\ -a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} + a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} \end{pmatrix}^{[1/2]}$$

Therefore,

$$\sqrt{3}U^{[1/2]} - V^{[1/2]} = \frac{1}{\sqrt{2}} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} - a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} - 2a_{p\beta}^\dagger a_{r\beta} a_{s\beta} \\ 2a_{p\alpha}^\dagger a_{r\alpha} a_{s\alpha} + a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} + a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} \end{pmatrix}^{[1/2]} - \frac{1}{\sqrt{2}} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} + a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} \\ -a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} + a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} \end{pmatrix}^{[1/2]}$$

$$= \frac{1}{\sqrt{2}} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} - a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} - 2a_{p\beta}^\dagger a_{r\beta} a_{s\beta} + a_{p\alpha}^\dagger a_{r\beta} a_{s\alpha} - a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} \\ 2a_{p\alpha}^\dagger a_{r\alpha} a_{s\alpha} + a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} + a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} + a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} - a_{p\beta}^\dagger a_{r\alpha} a_{s\beta} \end{pmatrix}^{[1/2]}$$

$$= \sqrt{2} \begin{pmatrix} -a_{p\alpha}^\dagger a_{r\alpha} a_{s\beta} - a_{p\beta}^\dagger a_{r\beta} a_{s\beta} \\ a_{p\alpha}^\dagger a_{r\alpha} a_{s\alpha} + a_{p\beta}^\dagger a_{r\beta} a_{s\alpha} \end{pmatrix}^{[1/2]}$$

Another case

$$S^{[1/2]} = \left(a_r\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_q\right)^{[1/2]}\right] = \begin{pmatrix} -a_{r\beta} \\ a_{r\alpha} \end{pmatrix}^{[1/2]} \otimes_{[1/2]} \begin{pmatrix} -a^\dagger_{p\alpha}a_{q\beta} \\ \frac{1}{\sqrt{2}}\left(a^\dagger_{p\alpha}a_{q\alpha} - a^\dagger_{p\beta}a_{q\beta}\right) \\ a^\dagger_{p\beta}a_{q\alpha} \end{pmatrix}^{[1]}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}}\frac{1}{\sqrt{3}}(-a_{r\beta})\left(a^\dagger_{p\alpha}a_{q\alpha} - a^\dagger_{p\beta}a_{q\beta}\right) + \frac{\sqrt{2}}{\sqrt{3}}a_{r\alpha}a^\dagger_{p\alpha}a_{q\beta} \\ -\frac{\sqrt{2}}{\sqrt{3}}a_{r\beta}a^\dagger_{p\beta}a_{q\alpha} - \frac{1}{\sqrt{2}}\frac{1}{\sqrt{3}}a_{r\alpha}\left(a^\dagger_{p\alpha}a_{q\alpha} - a^\dagger_{p\beta}a_{q\beta}\right) \end{pmatrix}^{[1/2]} = \frac{1}{\sqrt{6}}\begin{pmatrix} -a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\beta}a^\dagger_{p\beta}a_{q\beta} + 2a_{r\alpha}a^\dagger_{p\alpha}a_{q\beta} \\ -2a_{r\beta}a^\dagger_{p\beta}a_{q\alpha} - a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} \end{pmatrix}^{[1/2]}$$

$$T^{[1/2]} = \left(a_r\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_q\right)^{[1/2]}\right] = \frac{1}{\sqrt{2}}\begin{pmatrix} -a_{r\beta} \\ a_{r\alpha} \end{pmatrix}^{[1/2]} \otimes_{[1/2]} \left(a^\dagger_{p\alpha}a_{q\alpha} + a^\dagger_{p\beta}a_{q\beta}\right)^{[0]}$$

$$= \frac{1}{\sqrt{2}}\begin{pmatrix} -a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} - a_{r\beta}a^\dagger_{p\beta}a_{q\beta} \\ a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} \end{pmatrix}^{[1/2]}$$

Therefore,

$$\sqrt{3}S^{[1/2]} - T^{[1/2]} = \frac{1}{\sqrt{6}}\begin{pmatrix} -a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\beta}a^\dagger_{p\beta}a_{q\beta} + 2a_{r\alpha}a^\dagger_{p\alpha}a_{q\beta} \\ -2a_{r\beta}a^\dagger_{p\beta}a_{q\alpha} - a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} \end{pmatrix}^{[1/2]} - \frac{1}{\sqrt{2}}\begin{pmatrix} -a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} - a_{r\beta}a^\dagger_{p\beta}a_{q\beta} \\ a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} \end{pmatrix}^{[1/2]}$$

$$= \frac{1}{\sqrt{2}}\begin{pmatrix} -a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\beta}a^\dagger_{p\beta}a_{q\beta} + 2a_{r\alpha}a^\dagger_{p\alpha}a_{q\beta} + a_{r\beta}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\beta}a^\dagger_{p\beta}a_{q\beta} \\ -2a_{r\beta}a^\dagger_{p\beta}a_{q\alpha} - a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} + a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} - a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} - a_{r\alpha}a^\dagger_{p\beta}a_{q\beta} \end{pmatrix}^{[1/2]}$$

$$= \sqrt{2}\begin{pmatrix} a_{r\beta}a^\dagger_{p\beta}a_{q\beta} + a_{r\alpha}a^\dagger_{p\alpha}a_{q\beta} \\ -a_{r\beta}a^\dagger_{p\beta}a_{q\alpha} - a_{r\alpha}a^\dagger_{p\alpha}a_{q\alpha} \end{pmatrix}^{[1/2]}$$

Triplet times triplet

$$X^{[0]} = \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_q^\dagger\right)^{[1/2]}\right] \otimes_{[0]} \left[\left(a_r\right)^{[1/2]} \otimes_{[1]} \left(a_s\right)^{[1/2]}\right]$$

$$= \begin{pmatrix} a^\dagger_{p\alpha}a^\dagger_{q\alpha} \\ \frac{1}{\sqrt{2}}\left(a^\dagger_{p\alpha}a^\dagger_{q\beta} + a^\dagger_{p\beta}a^\dagger_{q\alpha}\right) \\ a^\dagger_{p\beta}a^\dagger_{q\beta} \end{pmatrix}^{[1]} \otimes_{[0]} \begin{pmatrix} a_{r\beta}a_{s\beta} \\ -\frac{1}{\sqrt{2}}\left(a_{r\beta}a_{s\alpha} + a_{r\alpha}a_{s\beta}\right) \\ a_{r\alpha}a_{s\alpha} \end{pmatrix}^{[1]}$$

$$= \frac{1}{\sqrt{3}}\left(a^\dagger_{p\alpha}a^\dagger_{q\alpha}a_{r\alpha}s_{s\alpha} + \frac{1}{2}\left(a^\dagger_{p\alpha}a^\dagger_{q\beta} + a^\dagger_{p\beta}a^\dagger_{q\alpha}\right)\left(a_{r\beta}a_{s\alpha} + a_{r\alpha}a_{s\beta}\right) + a^\dagger_{p\beta}a^\dagger_{q\beta}a_{r\beta}a_{s\beta}\right)$$

$$Y^{[0]} = \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_q^\dagger\right)^{[1/2]}\right] \otimes_{[0]} \left[\left(a_r\right)^{[1/2]} \otimes_{[0]} \left(a_s\right)^{[1/2]}\right]$$

$$= \frac{1}{\sqrt{2}}\left(a^\dagger_{p\alpha}a^\dagger_{q\beta} - a^\dagger_{p\beta}a^\dagger_{q\alpha}\right)^{[0]} \otimes_{[0]} \frac{1}{\sqrt{2}}\left(-a_{r\beta}a_{s\alpha} + a_{r\alpha}a_{s\beta}\right)^{[0]}$$

$$= \frac{1}{2}\left(a^\dagger_{p\alpha}a^\dagger_{q\beta} - a^\dagger_{p\beta}a^\dagger_{q\alpha}\right)\left(-a_{r\beta}a_{s\alpha} + a_{r\alpha}a_{s\beta}\right)$$

Using

$$(a+b)(c+d) + (a-b)(-c+d) = (a+b)(2d) - 2b(-c+d) = 2(ad+bc)$$

we have

$$\sqrt{3}X^{[0]} + Y^{[0]} = a^\dagger_{p\alpha}a^\dagger_{q\alpha}a_{r\alpha}s_{s\alpha} + a^\dagger_{p\beta}a^\dagger_{q\beta}a_{r\beta}a_{s\beta} + a^\dagger_{p\alpha}a^\dagger_{q\beta}a_{r\alpha}a_{s\beta} + a^\dagger_{p\beta}a^\dagger_{q\alpha}a_{r\beta}a_{s\alpha}$$

$$= \sum_{\sigma\sigma'} a^\dagger_{p\sigma}a^\dagger_{q\sigma'}a_{r\sigma}s_{s\sigma'}$$

Another case

$$
\begin{aligned}
Z^{[0]} &= \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_q\right)^{[1/2]}\right] \otimes_{[0]} \left[\left(a_r^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_s\right)^{[1/2]}\right] \\
&= \begin{pmatrix} -a_{p\alpha}^\dagger a_{q\beta} \\ \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\alpha} - a_{p\beta}^\dagger a_{q\beta}\right) \\ a_{p\beta}^\dagger a_{q\alpha} \end{pmatrix}^{[1]} \otimes_{[0]} \begin{pmatrix} -a_{r\alpha}^\dagger a_{s\beta} \\ \frac{1}{\sqrt{2}}\left(a_{r\alpha}^\dagger a_{s\alpha} - a_{r\beta}^\dagger a_{s\beta}\right) \\ a_{r\beta}^\dagger a_{s\alpha} \end{pmatrix}^{[1]} \\
&= \frac{1}{\sqrt{3}}\left(-a_{p\alpha}^\dagger a_{q\beta} a_{r\beta}^\dagger a_{s\alpha} - \tfrac{1}{2}\left(a_{p\alpha}^\dagger a_{q\alpha} - a_{p\beta}^\dagger a_{q\beta}\right)\left(a_{r\alpha}^\dagger a_{s\alpha} - a_{r\beta}^\dagger a_{s\beta}\right) - a_{p\beta}^\dagger a_{q\alpha} a_{r\alpha}^\dagger a_{s\beta}\right) \\
W^{[0]} &= \left[\left(a_p^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_q\right)^{[1/2]}\right] \otimes_{[0]} \left[\left(a_r^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_s\right)^{[1/2]}\right] \\
&= \frac{1}{\sqrt{2}}\left(a_{p\alpha}^\dagger a_{q\alpha} + a_{p\beta}^\dagger a_{q\beta}\right)^{[0]} \otimes_{[0]} \frac{1}{\sqrt{2}}\left(a_{r\alpha}^\dagger a_{s\alpha} + a_{r\beta}^\dagger a_{s\beta}\right)^{[0]} \\
&= \frac{1}{2}\left(a_{p\alpha}^\dagger a_{q\alpha} + a_{p\beta}^\dagger a_{q\beta}\right)\left(a_{r\alpha}^\dagger a_{s\alpha} + a_{r\beta}^\dagger a_{s\beta}\right)
\end{aligned}
$$

Using

$$(a-b)(c-d) + (a+b)(c+d) = (a+b)(2c) - (2b)(c-d) = 2(ac+bd)$$

we have

$$
\begin{aligned}
-\sqrt{3}Z^{[0]} + W^{[0]} &= a_{p\alpha}^\dagger a_{q\beta} a_{r\beta}^\dagger a_{s\alpha} + a_{p\beta}^\dagger a_{q\alpha} a_{r\alpha}^\dagger a_{s\beta} + a_{p\alpha}^\dagger a_{q\alpha} a_{r\alpha}^\dagger a_{s\alpha} + a_{p\beta}^\dagger a_{q\beta} a_{r\beta}^\dagger a_{s\beta} \\
&= \sum_{\sigma\sigma'} a_{p\sigma}^\dagger a_{q\sigma'} a_{r\sigma'}^\dagger a_{s\sigma}
\end{aligned}
$$

S Term

From second singlet formula we have

$$\sqrt{2}\sum_{i\in L}\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{S}_i^R\right)^{[\frac{1}{2}]} = \sum_{i\in L}\left(t_{ij} a_{i\alpha}^\dagger a_{j\alpha} + t_{ij} a_{i\beta}^\dagger a_{j\beta}\right)$$

R Term

This is the same as the S term. Note that in the expression for $\hat{R}$, we have a $\otimes_{[0]}$, this is because in the original spatial expression there is a summation over σ. Then there is a $[0] \otimes_{[1/2]} [1/2]$, which will not produce any extra coefficients.

AP Term

Using definition

$$
\begin{aligned}
\left(\hat{A}_{ik}\right)^{[0/1]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \\
\left(\hat{P}_{ik}^R\right)^{[0/1]} &= -\sum_{jl\in R} v_{ijkl}\left(a_j\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_l\right)^{[\frac{1}{2}]}
\end{aligned}
$$

We have

$$\sum_{ik\in L}\left[\sqrt{3}\left(\hat{A}_{ik}\right)^{[1]}\otimes_{[0]}\left(\hat{P}^R_{ik}\right)^{[1]}+\left(\hat{A}_{ik}\right)^{[0]}\otimes_{[0]}\left(\hat{P}^R_{ik}\right)^{[0]}\right]$$

$$=\sum_{ik\in L,jl\in R}v_{ijkl}\left[\sqrt{3}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[1]}\left(a_k^\dagger\right)^{[\frac{1}{2}]}\right]\otimes_{[0]}\left[\left(a_j\right)^{[\frac{1}{2}]}\otimes_{[1]}\left(a_l\right)^{[\frac{1}{2}]}\right]+\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(a_k^\dagger\right)^{[\frac{1}{2}]}\right]\otimes_{[0]}\left[\left(a_j\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(a_l\right)^{[\frac{1}{2}]}\right]\right]$$

$$=\sum_{ik\in L,jl\in R}v_{ijkl}\left[\sum_{\sigma\sigma'}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{j\sigma}a_{l\sigma'}\right]=-\sum_{ik\in L,jl\in R,\sigma\sigma'}v_{ijkl}a^\dagger_{i\sigma}a^\dagger_{k\sigma'}a_{l\sigma'}a_{j\sigma}$$

Note that in last step, we can anticommute $a_{l\sigma'}, a_{j\sigma}$ because it's assumed that in the σ summation, when $j=l, \sigma\neq\sigma'$. Otherwise there will be two a operators acting on the same site and the contribution is zero.

BQ Term

In spatial expression, this term is $BQ-B'Q'$. Now $-\sqrt{3}Z^{[0]}+W^{[0]}$ gives $B'Q'$. And $2W^{[0]}$ gives BQ. Therefore,

$$2W^{[0]}-\left(-\sqrt{3}Z^{[0]}+W^{[0]}\right)=\sqrt{3}Z^{[0]}+W^{[0]}$$

This looks like $\hat{A}\hat{P}$ term, but without $\frac{1}{2}$ and $h.c.$. But this is not correct, because the definition of Q, Q' is not equivalent due to the index order in v_{ijkl}. So they will give different $W^{[0]}$. Instead we have (note that $\left(\hat{B}_{ij}\right)^{[0]}=\left(\hat{B}'_{ij}\right)^{[0]}$)

$$\sum_{ij\in L}\left[2\left(\hat{B}_{ij}\right)^{[0]}\otimes_{[0]}\left(\hat{Q}^R_{ij}\right)^{[0]}-\left(\hat{B}'_{ij}\right)^{[0]}\otimes_{[0]}\left(\hat{Q'}^R_{ij}\right)^{[0]}+\sqrt{3}\left(\hat{B}'_{ij}\right)^{[1]}\otimes_{[0]}\left(\hat{Q'}^R_{ij}\right)^{[1]}\right]$$

$$=\sum_{ij\in L}\left[\left(\hat{B}_{ij}\right)^{[0]}\otimes_{[0]}\left(\left(2\hat{Q}^R_{ij}\right)^{[0]}-\left(\hat{Q'}^R_{ij}\right)^{[0]}\right)+\sqrt{3}\left(\hat{B}'_{ij}\right)^{[1]}\otimes_{[0]}\left(\hat{Q'}^R_{ij}\right)^{[1]}\right]$$

Note that B, Q do not have $[1]$ form.

Normal/Complementary Partitioning

Note that

$$\sqrt{2}\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{S}^R_i\right)^{[\frac{1}{2}]}+h.c.\right]=\sqrt{2}\sum_{i\in R}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{S}^L_i\right)^{[\frac{1}{2}]}+h.c.\right]$$

Therefore,

$$\sqrt{2}\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{S}^R_i\right)^{[\frac{1}{2}]}+h.c.\right]+2\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{R}^R_i\right)^{[\frac{1}{2}]}+h.c.\right]+2\sum_{i\in R}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{R}^L_i\right)^{[\frac{1}{2}]}+h.c.\right]$$

$$=\frac{\sqrt{2}}{2}\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{S}^R_i\right)^{[\frac{1}{2}]}+h.c.\right]+\frac{\sqrt{2}}{2}\sum_{i\in R}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{S}^L_i\right)^{[\frac{1}{2}]}+h.c.\right]$$

$$+2\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{R}^R_i\right)^{[\frac{1}{2}]}+h.c.\right]+2\sum_{i\in R}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(\hat{R}^L_i\right)^{[\frac{1}{2}]}+h.c.\right]$$

$$=2\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left[\left(\hat{R}^R_i\right)^{[\frac{1}{2}]}+\frac{\sqrt{2}}{4}\left(\hat{S}^R_i\right)^{[\frac{1}{2}]}\right]+h.c.\right]+2\sum_{i\in R}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left[\left(\hat{R}^L_i\right)^{[\frac{1}{2}]}+\frac{\sqrt{2}}{4}\left(\hat{S}^L_i\right)^{[\frac{1}{2}]}\right]+h.c.\right]$$

So define

$$\left(\hat{R}'^{L/R}_i\right)^{[\frac{1}{2}]}:=\frac{\sqrt{2}}{4}\left(\hat{S}^L_i\right)^{[\frac{1}{2}]}+\left(\hat{R}^L_i\right)^{[\frac{1}{2}]}=\frac{\sqrt{2}}{4}\sum_{j\in L/R}t_{ij}\left(a_j\right)^{[\frac{1}{2}]}+\sum_{jkl\in L/R}v_{ijkl}\left[\left(a_k^\dagger\right)^{[\frac{1}{2}]}\otimes_{[0]}\left(a_l\right)^{[\frac{1}{2}]}\right]\otimes_{[\frac{1}{2}]}\left(a_j\right)^{[\frac{1}{2}]}$$

Here $\frac{\sqrt{2}}{4}$ should be understood as $\frac{1}{2}\cdot\frac{1}{\sqrt{2}}$. The $\frac{1}{2}$ is the same as spatial case, and $\frac{1}{\sqrt{2}}$ is because the expected $\sqrt{2}$ factor is not added for the $\hat{R}$ term.

Operator Exchange factors

Here we consider fermion and SU(2) exchange factors together. From $j_2 = 1/2$ CG factors

$$\left\langle j_1 \ \left(M - \frac{1}{2}\right) \ \frac{1}{2} \ \frac{1}{2} \middle| \left(j_1 \pm \frac{1}{2}\right) \ M \right\rangle = \pm \sqrt{\frac{1}{2}\left(1 \pm \frac{M}{j_1 + \frac{1}{2}}\right)}$$
$$\left\langle j_1 \ \left(M + \frac{1}{2}\right) \ \frac{1}{2} \ \left(-\frac{1}{2}\right) \middle| \left(j_1 \pm \frac{1}{2}\right) \ M \right\rangle = \sqrt{\frac{1}{2}\left(1 \mp \frac{M}{j_1 + \frac{1}{2}}\right)}$$

Let $j_1 = \frac{1}{2}$ we have

$$\left\langle \frac{1}{2} \ \left(-\frac{1}{2}\right) \ \frac{1}{2} \ \frac{1}{2} \middle| \left(\frac{1}{2} \pm \frac{1}{2}\right) \ 0 \right\rangle = \pm \sqrt{\frac{1}{2}}$$
$$\left\langle \frac{1}{2} \ \frac{1}{2} \ \frac{1}{2} \ \left(-\frac{1}{2}\right) \middle| \left(\frac{1}{2} \pm \frac{1}{2}\right) \ 0 \right\rangle = \sqrt{\frac{1}{2}}$$

The exchange factor formula is

$$\begin{aligned}
\left(\hat{X}_1^{[S_1]} \otimes_{[S]} \hat{X}_2^{[S_2]}\right)^{[S_z]} &= \sum_{S_{1z}, S_{2z}} \hat{X}_1^{[S_1][S_{1z}]} \hat{X}_2^{[S_2][S_{2z}]} \langle S S_z | S_1 S_{1z}, \ S_2 S_{2z} \rangle \\
&= \mathrm{P}^{\mathrm{exchange}}_{\mathrm{fermi}}(N_1, N_2) \sum_{S_{1z}, S_{2z}} \hat{X}_2^{[S_2][S_{2z}]} \hat{X}_1^{[S_1][S_{1z}]} \langle S S_z | S_1 S_{1z}, \ S_2 S_{2z} \rangle \\
&= \mathrm{P}^{\mathrm{exchange}}_{\mathrm{fermi}}(N_1, N_2) \frac{\langle S S_z | S_1 S_{1z}, \ S_2 S_{2z} \rangle}{\langle S S_z | S_2 S_{2z}, \ S_1 S_{1z} \rangle} \left(\hat{X}_2^{[S_2]} \otimes_{[S]} \hat{X}_1^{[S_1]}\right)^{[S_z]} \\
\hat{X}_1^{[S_1]} \otimes_{[S]} \hat{X}_2^{[S_2]} &= \mathrm{P}^{\mathrm{exchange}}_{\mathrm{fermi}}(N_1, N_2) \mathrm{P}^{\mathrm{exchange}}_{\mathrm{SU(2)}}(S_1, S_2, S) \hat{X}_2^{[S_2]} \otimes_{[S]} \hat{X}_1^{[S_1]}
\end{aligned}$$

For $[1/2] \otimes_{[0]} [1/2]$, this is

$$\mathrm{P}^{\mathrm{exchange}}(\tfrac{1}{2}, \tfrac{1}{2}, 0) = (-1) \frac{\left\langle \frac{1}{2} \ \frac{1}{2} \ \frac{1}{2} \ \left(-\frac{1}{2}\right) \middle| 0 \ 0 \right\rangle}{\left\langle \frac{1}{2} \ \left(-\frac{1}{2}\right) \ \frac{1}{2} \ \frac{1}{2} \middle| 0 \ 0 \right\rangle} = (-1) \frac{\sqrt{\frac{1}{2}}}{-\sqrt{\frac{1}{2}}} = 1$$

For $[1/2] \otimes_{[1]} [1/2]$, this is

$$\mathrm{P}^{\mathrm{exchange}}(\tfrac{1}{2}, \tfrac{1}{2}, 1) = (-1) \frac{\left\langle \frac{1}{2} \ \frac{1}{2} \ \frac{1}{2} \ \left(-\frac{1}{2}\right) \middle| 1 \ 0 \right\rangle}{\left\langle \frac{1}{2} \ \left(-\frac{1}{2}\right) \ \frac{1}{2} \ \frac{1}{2} \middle| 1 \ 0 \right\rangle} = (-1) \frac{\sqrt{\frac{1}{2}}}{\sqrt{\frac{1}{2}}} = -1$$

From CG factors

$$\langle 1 \ m_1 \ 1 \ (-m_1) | 0 \ 0 \rangle = \frac{(-1)^{1 - m_1}}{\sqrt{3}}$$

we have

$$\mathrm{P}^{\mathrm{exchange}}(1, 1, 0) = (+1) \frac{\left\langle 1 \ 1 \ \ 1 \ -1 \middle| 0 \ 0 \right\rangle}{\left\langle 1 \ -1 \ 1 \ 1 \middle| 0 \ 0 \right\rangle} = (+1) \frac{\frac{(-1)^0}{\sqrt{3}}}{\frac{(-1)^2}{\sqrt{3}}} = 1$$

we have

$$
\begin{aligned}
(\hat{H})^{[0],NC} = & \left(\hat{H}^L\right)^{[0]} \otimes_{[0]} \left(\hat{1}^R\right)^{[0]} + \left(\hat{1}^L\right)^{[0]} \otimes_{[0]} \left(\hat{H}^R\right)^{[0]} \\
& + 2\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i^R\right)^{[\frac{1}{2}]} + \left(a_i\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i'^{R\dagger}\right)^{[\frac{1}{2}]}\right] + 2\sum_{i\in R}\left[\left(\hat{R}_i'^{L\dagger}\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_i\right)^{[\frac{1}{2}]} + \left(\hat{R}_i'^{L}\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_i^\dagger\right)^{[\frac{1}{2}]}\right] \\
& - \frac{1}{2}\sum_{ik\in L}\left[\left(\hat{A}_{ik}\right)^{[0]} \otimes_{[0]} \left(\hat{P}_{ik}^R\right)^{[0]} + \sqrt{3}\left(\hat{A}_{ik}\right)^{[1]} \otimes_{[0]} \left(\hat{P}_{ik}^R\right)^{[1]} + \left(\hat{A}_{ik}^\dagger\right)^{[0]} \otimes_{[0]} \left(\hat{P}_{ik}^{R\dagger}\right)^{[0]} + \sqrt{3}\left(\hat{A}_{ik}^\dagger\right)^{[1]} \otimes_{[0]} \left(\hat{P}_{ik}^{R\dagger}\right)^{[1]}\right] \\
& + \sum_{ij\in L}\left[\left(\hat{B}_{ij}\right)^{[0]} \otimes_{[0]} \left(\hat{Q}_{ij}''^{R}\right)^{[0]} + \sqrt{3}\left(\hat{B}'_{ij}\right)^{[1]} \otimes_{[0]} \left(\hat{Q}_{ij}'^{R}\right)^{[1]}\right]
\end{aligned}
$$

With this normal/complementary partitioning, the operators required in left block are

$$
\left\{\left(\hat{H}^L\right)^{[0]}, \left(\hat{1}^L\right)^{[0]}, \left(a_i^\dagger\right)^{[\frac{1}{2}]}, \left(a_i\right)^{[\frac{1}{2}]}, \left(\hat{R}_k'^{L\dagger}\right)^{[\frac{1}{2}]}, \left(\hat{R}_k'^{L}\right)^{[\frac{1}{2}]}, \left(\hat{A}_{ij}\right)^{[0]}, \left(\hat{A}_{ij}\right)^{[1]}, \left(\hat{A}_{ij}^\dagger\right)^{[0]}, \left(\hat{A}_{ij}^\dagger\right)^{[1]}, \left(\hat{B}_{ij}\right)^{[0]}, \left(\hat{B}'_{ij}\right)^{[1]}\right\} \quad (i,j\in L, k\in R)
$$

The operators required in right block are

$$
\left\{\left(\hat{1}^R\right)^{[0]}, \left(\hat{H}^R\right)^{[0]}, \left(\hat{R}_i'^{R}\right)^{[\frac{1}{2}]}, \left(\hat{R}_i'^{R\dagger}\right)^{[\frac{1}{2}]}, \left(a_k\right)^{[\frac{1}{2}]}, \left(a_k^\dagger\right)^{[\frac{1}{2}]}, \left(\hat{P}_{ij}^R\right)^{[0]}, \left(\hat{P}_{ij}^R\right)^{[1]}, \left(\hat{P}_{ij}^{R\dagger}\right)^{[0]}, \left(\hat{P}_{ij}^{R\dagger}\right)^{[1]}, \left(\hat{Q}_{ij}''^{R}\right)^{[0]}, \left(\hat{Q}_{ij}'^{R}\right)^{[1]}\right\} \quad (i,j\in L, k\in R)
$$

Assuming that there are K sites in total, and K_L/K_R sites in left/right block (optimally, $K_L \leq K_R$), the total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$
N_{NC} = 1 + 1 + 2K_L + 2K_R + 4K_L^2 + 2K_L^2 = 6K_L^2 + 2K + 2
$$

Complementary/Normal Partitioning

Note that due the CG factors, exchange any $\otimes_{[0]}$ product will not produce extra sign.

$$
\begin{aligned}
(\hat{H})^{[0],CN} = & \left(\hat{H}^L\right)^{[0]} \otimes_{[0]} \left(\hat{1}^R\right)^{[0]} + \left(\hat{1}^L\right)^{[0]} \otimes_{[0]} \left(\hat{H}^R\right)^{[0]} \\
& + 2\sum_{i\in L}\left[\left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i'^{R}\right)^{[\frac{1}{2}]} + \left(a_i\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(\hat{R}_i'^{R\dagger}\right)^{[\frac{1}{2}]}\right] + 2\sum_{i\in R}\left[\left(\hat{R}_i'^{L\dagger}\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_i\right)^{[\frac{1}{2}]} + \left(\hat{R}_i'^{L}\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_i^\dagger\right)^{[\frac{1}{2}]}\right] \\
& - \frac{1}{2}\sum_{jl\in R}\left[\left(\hat{P}_{jl}^L\right)^{[0]} \otimes_{[0]} \left(\hat{A}_{jl}\right)^{[0]} + \sqrt{3}\left(\hat{P}_{jl}^L\right)^{[1]} \otimes_{[0]} \left(\hat{A}_{jl}\right)^{[1]} + \left(\hat{P}_{jl}^{L\dagger}\right)^{[0]} \otimes_{[0]} \left(\hat{A}_{jl}^\dagger\right)^{[0]} + \sqrt{3}\left(\hat{P}_{jl}^{L\dagger}\right)^{[1]} \otimes_{[0]} \left(\hat{A}_{jl}^\dagger\right)^{[1]}\right] \\
& + \sum_{kl\in R}\left[\left(\hat{Q}_{kl}''^{L}\right)^{[0]} \otimes_{[0]} \left(\hat{B}_{kl}\right)^{[0]} + \sqrt{3}\left(\hat{Q}_{kl}'^{L}\right)^{[1]} \otimes_{[0]} \left(\hat{B}'_{kl}\right)^{[1]}\right]
\end{aligned}
$$

Now the operators required in left block are

$$
\left\{\left(\hat{H}^L\right)^{[0]}, \left(\hat{1}^L\right)^{[0]}, \left(a_i^\dagger\right)^{[\frac{1}{2}]}, \left(a_i\right)^{[\frac{1}{2}]}, \left(\hat{R}_k'^{L\dagger}\right)^{[\frac{1}{2}]}, \left(\hat{R}_k'^{L}\right)^{[\frac{1}{2}]}, \left(\hat{P}_{kl}^L\right)^{[0]}, \left(\hat{P}_{kl}^L\right)^{[1]}, \left(\hat{P}_{kl}^{L\dagger}\right)^{[0]}, \left(\hat{P}_{kl}^{L\dagger}\right)^{[1]}, \left(\hat{Q}_{kl}''^{L}\right)^{[0]}, \left(\hat{Q}_{kl}'^{L}\right)^{[1]}\right\} \quad (k,l\in R, i\in L)
$$

The operators required in right block are

$$
\left\{\left(\hat{1}^R\right)^{[0]}, \left(\hat{H}^R\right)^{[0]}, \left(\hat{R}_i'^{R}\right)^{[\frac{1}{2}]}, \left(\hat{R}_i'^{R\dagger}\right)^{[\frac{1}{2}]}, \left(a_k\right)^{[\frac{1}{2}]}, \left(a_k^\dagger\right)^{[\frac{1}{2}]}, \left(\hat{A}_{kl}\right)^{[0]}, \left(\hat{A}_{kl}\right)^{[1]}, \left(\hat{A}_{kl}^\dagger\right)^{[0]}, \left(\hat{A}_{kl}^\dagger\right)^{[1]}, \left(\hat{B}_{kl}\right)^{[0]}, \left(\hat{B}'_{kl}\right)^{[1]}\right\} \quad (k,l\in R, i\in L)
$$

The total number of operators (and also the number of terms in Hamiltonian with partition) in left or right block is

$$
N_{CN} = 1 + 1 + 2K_L + 2K_R + 4K_R^2 + 2K_R^2 = 6K_R^2 + 2K + 2
$$

1.3.2 Blocking

The enlarged left/right block is denoted as $L*/R*$. Make sure that all L operators are to the left of $*$ operators. (The exchange factor for this is -1 for doublet $\otimes$ triplet and +1 doublet $\otimes$ singlet.)

First we have

$$\begin{aligned}\left(\hat{R}_i^{L/R}\right)^{[1/2]} &= \sum_{jkl\in L/R} v_{ijkl}\left[\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_l\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_j\right)^{[1/2]} \\ &= \frac{1}{\sqrt{2}} \sum_{jkl\in L/R} v_{ijkl}\left(a_{k\alpha}^\dagger a_{l\alpha} + a_{k\beta}^\dagger a_{l\beta}\right)^{[0]} \otimes_{[1/2]} \left(a_j\right)^{[1/2]} \\ &= \frac{1}{\sqrt{2}} \sum_{jkl\in L/R} v_{ijkl}\begin{pmatrix} -a_{k\alpha}^\dagger a_{l\alpha} a_{j\beta} - a_{k\beta}^\dagger a_{l\beta} a_{j\beta} \\ a_{k\alpha}^\dagger a_{l\alpha} a_{j\alpha} + a_{k\beta}^\dagger a_{l\beta} a_{j\alpha}\end{pmatrix}^{[1/2]}\end{aligned}$$

From the formula $\sqrt{3}U^{[1/2]} - V^{[1/2]}$ we have

$$\left(\hat{R}_i^{L/R}\right)^{[1/2]} = \frac{\sqrt{3}}{2} \sum_{jkl\in L/R} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] - \frac{1}{2} \sum_{jkl\in L/R} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right]$$

From the formula $\sqrt{3}S^{[1/2]} - T^{[1/2]}$ we have (for $k \neq l$)

$$\left(\hat{R}_i^{L/R}\right)^{[1/2]} = \frac{\sqrt{3}}{2} \sum_{jkl\in L/R} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] - \frac{1}{2} \sum_{jkl\in L/R} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right]$$

We have

$$
\begin{aligned}
\left(\hat{R}_i'^{L*}\right)^{[1/2]} &= \left(\hat{R}_i'^{L}\right)^{[1/2]} \otimes_{[1/2]} \left(\hat{1}^{*}\right)^{[0]} + \left(\hat{1}^{L}\right)^{[0]} \otimes_{[1/2]} \left(\hat{R}_i'^{*}\right)^{[1/2]} \\
&+ \sum_{j\in L}\left[\sum_{kl\in *} v_{ijkl}\left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] \otimes_{[\frac{1}{2}]} \left(a_j\right)^{[\frac{1}{2}]} + \sum_{j\in *}\left[\sum_{kl\in L} v_{ijkl}\left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] \otimes_{[\frac{1}{2}]} \left(a_j\right)^{[\frac{1}{2}]} \\
&- \frac{1}{2}\sum_{k\in L}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{k\in L}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&- \frac{1}{2}\sum_{k\in *}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in L} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{k\in *}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in L} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&- \frac{1}{2}\sum_{l\in L}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{l\in L}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&- \frac{1}{2}\sum_{l\in *}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in L} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{l\in *}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in L} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&= \left(\hat{R}_i'^{L}\right)^{[1/2]} \otimes_{[1/2]} \left(\hat{1}^{*}\right)^{[0]} + \left(\hat{1}^{L}\right)^{[0]} \otimes_{[1/2]} \left(\hat{R}_i'^{*}\right)^{[1/2]} \\
&+ \sum_{j\in L}\left(a_j\right)^{[\frac{1}{2}]} \otimes_{[\frac{1}{2}]} \left[\sum_{kl\in *} v_{ijkl}\left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] + \sum_{j\in *}\left[\sum_{kl\in L} v_{ijkl}\left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] \otimes_{[\frac{1}{2}]} \left(a_j\right)^{[\frac{1}{2}]} \\
&- \frac{1}{2}\sum_{k\in L}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{k\in L}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&- \frac{1}{2}\sum_{k\in *}\left[\sum_{jl\in L} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{k\in *}\left[\sum_{jl\in L} v_{ijkl}\left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right) \\
&- \frac{1}{2}\sum_{l\in L}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{l\in L}\left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \\
&- \frac{1}{2}\sum_{l\in *}\left[\sum_{jk\in L} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_l\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{l\in *}\left[\sum_{jk\in L} v_{ijkl}\left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_l\right)
\end{aligned}
$$

After reordering of terms

$$
\begin{aligned}
\left(\hat{R}_i'^{L*}\right)^{[1/2]} &= \left(\hat{R}_i'^{L}\right)^{[1/2]} \otimes_{[1/2]} \left(\hat{1}^{*}\right)^{[0]} + \left(\hat{1}^{L}\right)^{[0]} \otimes_{[1/2]} \left(\hat{R}_i'^{*}\right)^{[1/2]} \\
&- \frac{1}{2}\sum_{k\in L} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{k\in L} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1} \\
&+ \sum_{j\in L} \left(a_j\right)^{[\frac{1}{2}]} \otimes_{[\frac{1}{2}]} \left[\sum_{kl\in *} v_{ijkl} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] \\
&- \frac{1}{2}\sum_{l\in L} \left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{l\in L} \left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1} \\
&- \frac{1}{2}\sum_{k\in *} \left[\sum_{jl\in L} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{k\in *} \left[\sum_{jl\in L} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right) \\
&+ \sum_{j\in *} \left[\sum_{kl\in L} v_{ijkl} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}\right] \otimes_{[\frac{1}{2}]} \left(a_j\right)^{[\frac{1}{2}]} \\
&- \frac{1}{2}\sum_{l\in *} \left[\sum_{jk\in L} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_l\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{l\in *} \left[\sum_{jk\in L} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_l\right. \\
&= \left(\hat{R}_i'^{L}\right)^{[1/2]} \otimes_{[1/2]} \left(\hat{1}^{*}\right)^{[0]} + \left(\hat{1}^{L}\right)^{[0]} \otimes_{[1/2]} \left(\hat{R}_i'^{*}\right)^{[1/2]} \\
&- \frac{1}{2}\sum_{k\in L} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{k\in L} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jl\in *} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1} \\
&+ \frac{1}{2}\sum_{j\in L} \left(a_j\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{kl\in *} (2v_{ijkl} - v_{ilkj}) \left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_l\right)^{[1/2]}\right] + \frac{\sqrt{3}}{2}\sum_{l\in L} \left(a_l\right)^{[1/2]} \otimes_{[1/2]} \left[\sum_{jk\in *} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes\right. \\
&- \frac{1}{2}\sum_{k\in *} \left[\sum_{jl\in L} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[0]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{k\in *} \left[\sum_{jl\in L} v_{ijkl} \left(a_l\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_k^\dagger\right) \\
&+ \frac{1}{2}\sum_{j\in *} \left[\sum_{kl\in L} (2v_{ijkl} - v_{ilkj}) \left(a_k^\dagger\right)^{[1/2]} \otimes_{[0]} \left(a_l\right)^{[1/2]}\right] \otimes_{[1/2]} \left(a_j\right)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{l\in *} \left[\sum_{jk\in L} v_{ijkl} \left(a_k^\dagger\right)^{[1/2]} \otimes_{[1]} \left(a_j\right)^{[1/2]}\right]
\end{aligned}
$$

By definition (The overall exchange factor for $[1/2] \otimes_{[0]} [1/2]$ is 1, and for $[1/2] \otimes_{[1]} [1/2]$ is -1)

$$
\begin{aligned}
\left(\hat{A}_{ik}\right)^{[0/1]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \\
\left(\hat{A}_{ik}^\dagger\right)^{[0]} &= \left(a_i\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_k\right)^{[\frac{1}{2}]} = \left(a_k\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_i\right)^{[\frac{1}{2}]} \\
\left(\hat{A}_{ik}^\dagger\right)^{[1]} &= -\left(a_i\right)^{[\frac{1}{2}]} \otimes_{[1]} \left(a_k\right)^{[\frac{1}{2}]} = \left(a_k\right)^{[\frac{1}{2}]} \otimes_{[1]} \left(a_i\right)^{[\frac{1}{2}]} \\
\left(\hat{P}_{ik}^R\right)^{[0/1]} &= \sum_{jl\in R} v_{ijkl} \left(a_l\right)^{[\frac{1}{2}]} \otimes_{[0/1]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{B}_{ij}\right)^{[0]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{B}_{ij}'\right)^{[1]} &= \left(a_i^\dagger\right)^{[\frac{1}{2}]} \otimes_{[1]} \left(a_j\right)^{[\frac{1}{2}]} \\
\left(\hat{Q}_{ij}'^{R}\right)^{[1]} &= \sum_{kl\in R} v_{ilkj} \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[1]} \left(a_l\right)^{[\frac{1}{2}]} \\
\left(\hat{Q}_{ij}''^{R}\right)^{[0]} &= \sum_{kl\in R} (2v_{ijkl} - v_{ilkj}) \left(a_k^\dagger\right)^{[\frac{1}{2}]} \otimes_{[0]} \left(a_l\right)^{[\frac{1}{2}]}
\end{aligned}
$$

we have

$$
\begin{aligned}
\big(\hat{R}_i'^{L*,NC}\big)^{[1/2]} = {} & \big(\hat{R}_i'^{L}\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[1/2]} \big(\hat{R}_i'^{*}\big)^{[1/2]} \\
& - \frac{1}{2} \sum_{k\in L} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{P}_{ik}^{*}\big)^{[0]} + \frac{\sqrt{3}}{2} \sum_{k\in L} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{P}_{ik}^{*}\big)^{[1]} \\
& + \frac{1}{2} \sum_{j\in L} \big(a_j\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{Q}_{ij}''^{*}\big)^{[0]} + \frac{\sqrt{3}}{2} \sum_{l\in L} \big(a_l\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{Q}_{il}'^{*}\big)^{[1]} \\
& - \frac{1}{2} \sum_{k\in *, jl\in L} v_{ijkl} \big(\hat{A}_{jl}^\dagger\big)^{[0]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} - \frac{\sqrt{3}}{2} \sum_{k\in *, jl\in L} v_{ijkl} \big(\hat{A}_{jl}^\dagger\big)^{[1]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} \\
& + \frac{1}{2} \sum_{j\in *, kl\in L} (2v_{ijkl} - v_{ilkj}) \big(\hat{B}_{kl}\big)^{[0]} \otimes_{[1/2]} \big(a_j\big)^{[1/2]} - \frac{\sqrt{3}}{2} \sum_{l\in *, jk\in L} v_{ijkl} \big(\hat{B}_{kj}'\big)^{[1]} \otimes_{[1/2]} \big(a_l\big)^{[1/2]} \\
\big(\hat{R}_i'^{L*,CN}\big)^{[1/2]} = {} & \big(\hat{R}_i'^{L}\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[1/2]} \big(\hat{R}_i'^{*}\big)^{[1/2]} \\
& - \frac{1}{2} \sum_{k\in L, jl\in *} v_{ijkl} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{A}_{jl}^\dagger\big)^{[0]} + \frac{\sqrt{3}}{2} \sum_{k\in L, jl\in *} v_{ijkl} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{A}_{jl}^\dagger\big)^{[1]} \\
& + \frac{1}{2} \sum_{j\in L, kl\in *} (2v_{ijkl} - v_{ilkj}) \big(a_j\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{B}_{kl}\big)^{[0]} + \frac{\sqrt{3}}{2} \sum_{l\in L, jk\in *} v_{ijkl} \big(a_l\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{B}_{kj}'\big)^{[1]} \\
& - \frac{1}{2} \sum_{k\in *} \big(\hat{P}_{ik}^{L}\big)^{[0]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} - \frac{\sqrt{3}}{2} \sum_{k\in *} \big(\hat{P}_{ik}^{L}\big)^{[1]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} \\
& + \frac{1}{2} \sum_{j\in *} \big(\hat{Q}_{ij}''^{L}\big)^{[0]} \otimes_{[1/2]} \big(a_j\big)^{[1/2]} - \frac{\sqrt{3}}{2} \sum_{l\in *} \big(\hat{Q}_{il}'^{L}\big)^{[1]} \otimes_{[1/2]} \big(a_l\big)^{[1/2]}
\end{aligned}
$$

To generate symmetrized P, we need to change the A line to the following

$$
-\frac{1}{4} \sum_{k\in *, jl\in L} (v_{ijkl} + v_{ilkj}) \big(\hat{A}_{jl}^\dagger\big)^{[0]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} - \frac{\sqrt{3}}{4} \sum_{k\in *, jl\in L} (v_{ijkl} - v_{ilkj}) \big(\hat{A}_{jl}^\dagger\big)^{[1]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]}
$$

Similarly,

$$
\begin{aligned}
\big(\hat{R}_i'^{R*,NC}\big)^{[1/2]} &= \big(\hat{R}_i'^{*}\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{1}^{R}\big)^{[0]} + \big(\hat{1}^{*}\big)^{[0]} \otimes_{[1/2]} \big(\hat{R}_i'^{R}\big)^{[1/2]} \\
&- \frac{1}{2}\sum_{k\in *} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{P}_{ik}^{R}\big)^{[0]} + \frac{\sqrt{3}}{2}\sum_{k\in *} \big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{P}_{ik}^{R}\big)^{[1]} \\
&+ \frac{1}{2}\sum_{j\in *} \big(a_j\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{Q}_{ij}''^{R}\big)^{[0]} + \frac{\sqrt{3}}{2}\sum_{l\in *} \big(a_l\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{Q}_{il}'^{R}\big)^{[1]} \\
&- \frac{1}{2}\sum_{k\in R, jl\in *} v_{ijkl}\big(\hat{A}_{jl}^\dagger\big)^{[0]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{k\in R, jl\in *} v_{ijkl}\big(\hat{A}_{jl}^\dagger\big)^{[1]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} \\
&+ \frac{1}{2}\sum_{j\in R, kl\in *} (2v_{ijkl} - v_{ilkj})\big(\hat{B}_{kl}\big)^{[0]} \otimes_{[1/2]} \big(a_j\big)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{l\in R, jk\in *} v_{ijkl}\big(\hat{B}'_{kj}\big)^{[1]} \otimes_{[1/2]} \big(a_l\big)^{[1/2]} \\
\big(\hat{R}_i'^{R*,CN}\big)^{[1/2]} &= \big(\hat{R}_i'^{*}\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{1}^{R}\big)^{[0]} + \big(\hat{1}^{*}\big)^{[0]} \otimes_{[1/2]} \big(\hat{R}_i'^{R}\big)^{[1/2]} \\
&- \frac{1}{2}\sum_{k\in *, jl\in R} v_{ijkl}\big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{A}_{jl}^\dagger\big)^{[0]} + \frac{\sqrt{3}}{2}\sum_{k\in *, jl\in R} v_{ijkl}\big(a_k^\dagger\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{A}_{jl}^\dagger\big)^{[1]} \\
&+ \frac{1}{2}\sum_{j\in *, kl\in R} (2v_{ijkl} - v_{ilkj})\big(a_j\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{B}_{kl}\big)^{[0]} + \frac{\sqrt{3}}{2}\sum_{l\in *, jk\in R} v_{ijkl}\big(a_l\big)^{[1/2]} \otimes_{[1/2]} \big(\hat{B}'_{kj}\big)^{[1]} \\
&- \frac{1}{2}\sum_{k\in R} \big(\hat{P}_{ik}^{*}\big)^{[0]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{k\in R} \big(\hat{P}_{ik}^{*}\big)^{[1]} \otimes_{[1/2]} \big(a_k^\dagger\big)^{[1/2]} \\
&+ \frac{1}{2}\sum_{j\in R} \big(\hat{Q}_{ij}''^{*}\big)^{[0]} \otimes_{[1/2]} \big(a_j\big)^{[1/2]} - \frac{\sqrt{3}}{2}\sum_{l\in R} \big(\hat{Q}_{il}'^{*}\big)^{[1]} \otimes_{[1/2]} \big(a_l\big)^{[1/2]}
\end{aligned}
$$

Number of terms

$$
\begin{aligned}
N_{R',NC} &= (2 + 4K_L + 4K_L^2)K_R + (2 + 4 + 4K_R)K_L = 4K_L^2K_R + 8K_LK_R + 2K + 4K_L \\
N_{R',CN} &= (2 + 4K_L + 4)K_R + (2 + 4K_R^2 + 4K_R)K_L = 4K_R^2K_L + 8K_RK_L + 2K + 4K_R
\end{aligned}
$$

Blocking of other complementary operators is straightforward

$$
\begin{aligned}
\big(\hat{P}_{ik}^{L*,CN}\big)^{[0/1]} &= \big(\hat{P}_{ik}^{L}\big)^{[0/1]} \otimes_{[0/1]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[0/1]} \big(\hat{P}_{ik}^{*}\big)^{[0/1]} + \sum_{j\in L, l\in *} v_{ijkl}\big(a_l\big)^{[\frac{1}{2}]} \otimes_{[0/1]} \big(a_j\big)^{[\frac{1}{2}]} + \sum_{j\in *, l\in L} v_{ijkl}\big(a_l\big)^{[\frac{1}{2}]} \otimes_{[0/} \\
&= \big(\hat{P}_{ik}^{L}\big)^{[0/1]} \otimes_{[0/1]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[0/1]} \big(\hat{P}_{ik}^{*}\big)^{[0/1]} \pm \sum_{j\in L, l\in *} v_{ijkl}\big(a_j\big)^{[\frac{1}{2}]} \otimes_{[0/1]} \big(a_l\big)^{[\frac{1}{2}]} + \sum_{j\in *, l\in L} v_{ijkl}\big(a_l\big)^{[\frac{1}{2}]} \otimes_{[0/} \\
\big(\hat{P}_{ik}^{R*,NC}\big)^{[0/1]} &= \big(\hat{P}_{ik}^{*}\big)^{[0/1]} \otimes_{[0/1]} \big(\hat{1}^{R}\big)^{[0]} + \big(\hat{1}^{*}\big)^{[0]} \otimes_{[0/1]} \big(\hat{P}_{ik}^{R}\big)^{[0/1]} \pm \sum_{j\in *, l\in R} v_{ijkl}\big(a_j\big)^{[\frac{1}{2}]} \otimes_{[0/1]} \big(a_l\big)^{[\frac{1}{2}]} + \sum_{j\in R, l\in *} v_{ijkl}\big(a_l\big)^{[\frac{1}{2}]} \otimes_{[0/}
\end{aligned}
$$

and

$$
\begin{aligned}
\big(\hat{Q}_{ij}''^{L*,CN}\big)^{[0]} &= \big(\hat{Q}_{ij}''^{L}\big)^{[0]} \otimes_{[0]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[0]} \big(\hat{Q}_{ij}''^{*}\big)^{[0]} + \sum_{k\in L, l\in *} (2v_{ijkl} - v_{ilkj})\big(a_k^\dagger\big)^{[\frac{1}{2}]} \otimes_{[0]} \big(a_l\big)^{[\frac{1}{2}]} + \sum_{k\in *, l\in L} (2v_{ijkl} - v_{ilkj}) \\
&= \big(\hat{Q}_{ij}''^{L}\big)^{[0]} \otimes_{[0]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[0]} \big(\hat{Q}_{ij}''^{*}\big)^{[0]} + \sum_{k\in L, l\in *} (2v_{ijkl} - v_{ilkj})\big(a_k^\dagger\big)^{[\frac{1}{2}]} \otimes_{[0]} \big(a_l\big)^{[\frac{1}{2}]} + \sum_{k\in *, l\in L} (2v_{ijkl} - v_{ilkj}) \\
\big(\hat{Q}_{ij}''^{R*,NC}\big)^{[0]} &= \big(\hat{Q}_{ij}''^{*}\big)^{[0]} \otimes_{[0]} \big(\hat{1}^{R}\big)^{[0]} + \big(\hat{1}^{*}\big)^{[0]} \otimes_{[0]} \big(\hat{Q}_{ij}''^{R}\big)^{[0]} + \sum_{k\in *, l\in R} (2v_{ijkl} - v_{ilkj})\big(a_k^\dagger\big)^{[\frac{1}{2}]} \otimes_{[0]} \big(a_l\big)^{[\frac{1}{2}]} + \sum_{k\in R, l\in *} (2v_{ijkl} - v_{ilkj}
\end{aligned}
$$

and

$$\begin{aligned}
\big(\hat{Q}'^{L*,CN}_{ij}\big)^{[1]} &= \big(\hat{Q}'^{L}_{ij}\big)^{[1]} \otimes_{[1]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[1]} \big(\hat{Q}'^{*}_{ij}\big)^{[1]} + \sum_{k\in L, l\in *} v_{ilkj} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a_{l}\big)^{[\frac{1}{2}]} + \sum_{k\in *, l\in L} v_{ilkj} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a_{l}\big)^{[\frac{1}{2}]} \\
&= \big(\hat{Q}'^{L}_{ij}\big)^{[1]} \otimes_{[1]} \big(\hat{1}^{*}\big)^{[0]} + \big(\hat{1}^{L}\big)^{[0]} \otimes_{[1]} \big(\hat{Q}'^{*}_{ij}\big)^{[1]} + \sum_{k\in L, l\in *} v_{ilkj} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a_{l}\big)^{[\frac{1}{2}]} - \sum_{k\in *, l\in L} v_{ilkj} \big(a_{l}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \\
\big(\hat{Q}'^{R*,CN}_{ij}\big)^{[1]} &= \big(\hat{Q}'^{*}_{ij}\big)^{[1]} \otimes_{[1]} \big(\hat{1}^{R}\big)^{[0]} + \big(\hat{1}^{*}\big)^{[0]} \otimes_{[1]} \big(\hat{Q}'^{R}_{ij}\big)^{[1]} + \sum_{k\in *, l\in R} v_{ilkj} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a_{l}\big)^{[\frac{1}{2}]} - \sum_{k\in R, l\in *} v_{ilkj} \big(a_{l}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]}
\end{aligned}$$

1.3.3 Middle-Site Transformation

$$\begin{aligned}
\big(\hat{P}^{L,NC\rightarrow CN}_{ik}\big)^{[0/1]} &= \sum_{jl\in L} v_{ijkl} \big(a_{l}\big)^{[\frac{1}{2}]} \otimes_{[0/1]} \big(a_{j}\big)^{[\frac{1}{2}]} = \sum_{jl\in L} v_{ijkl} \big(\hat{A}^{\dagger}_{jl}\big)^{[0/1]} \\
\big(\hat{Q}''^{L,NC\rightarrow CN}_{ij}\big)^{[0]} &= \sum_{kl\in R} (2v_{ijkl} - v_{ilkj}) \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[0]} \big(a_{l}\big)^{[\frac{1}{2}]} = \sum_{kl\in R} (2v_{ijkl} - v_{ilkj}) \big(\hat{B}_{kl}\big)^{[0]} \\
\big(\hat{Q}'^{L,NC\rightarrow CN}_{ij}\big)^{[1]} &= \sum_{kl\in R} v_{ilkj} \big(a^{\dagger}_{k}\big)^{[\frac{1}{2}]} \otimes_{[1]} \big(a_{l}\big)^{[\frac{1}{2}]} = \sum_{kl\in R} v_{ilkj} \big(\hat{B}'_{kl}\big)^{[1]}
\end{aligned}$$

1.4 Diagonal Two-Particle Density Matrix

1.4.1 PDM Definition

One-particle density matrix

$$\langle a^{\dagger}_{p\sigma} a_{q\tau} \rangle$$

Two-particle density matrix

$$\langle a^{\dagger}_{p\sigma} a^{\dagger}_{q\tau} a_{r\gamma} a_{s\lambda} \rangle$$

Spatial one-particle density matrix

$$E_{pq} \equiv \sum_{\sigma} \langle a^{\dagger}_{p\sigma} a_{q\sigma} \rangle$$

Spatial two-particle density matrix

$$e_{pqrs} \equiv \sum_{\sigma\tau} \langle a^{\dagger}_{p\sigma} a^{\dagger}_{q\tau} a_{r\tau} a_{s\sigma} \rangle$$

Spatial two-spin density matrix

$$s_{pqrs} \equiv \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle a^{\dagger}_{p\sigma} a^{\dagger}_{q\tau} a_{r\tau} a_{s\sigma} \rangle$$

where

$$(-1)^{1+\delta_{\sigma\tau}} = \begin{cases} 1 & \sigma = \tau \\ -1 & \sigma \neq \tau \end{cases}$$

1.4.2 NPC Definition

Number of particle correlation (pure spin)

$$\langle n_{p\sigma} n_{q\tau} \rangle = \langle a^\dagger_{p\sigma} a_{p\sigma} a^\dagger_{q\tau} a_{q\tau} \rangle$$

Number of particle correlation (mixed spin)

$$\langle a^\dagger_{p\sigma} a_{p\tau} a^\dagger_{q\tau} a_{q\sigma} \rangle$$

1.4.3 Spin/Charge Correlation

Spin correlation

$$S_{pq} = \langle (n_{p\alpha} - n_{p\beta})(n_{q\alpha} - n_{q\beta}) \rangle = \langle n_{p\alpha} n_{q\alpha} \rangle - \langle n_{p\alpha} n_{q\beta} \rangle - \langle n_{p\beta} n_{q\alpha} \rangle + \langle n_{p\beta} n_{q\beta} \rangle = \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle n_{p\sigma} n_{q\tau} \rangle$$

Charge correlation

$$C_{pq} = \langle (n_{p\alpha} + n_{p\beta})(n_{q\alpha} + n_{q\beta}) \rangle = \langle n_{p\alpha} n_{q\alpha} \rangle + \langle n_{p\alpha} n_{q\beta} \rangle + \langle n_{p\beta} n_{q\alpha} \rangle + \langle n_{p\beta} n_{q\beta} \rangle = \sum_{\sigma\tau} \langle n_{p\sigma} n_{q\tau} \rangle$$

1.4.4 Diagonal Spatial Two-Particle Density Matrix (Pure Spin)

Using anticommutation relation

$$a^\dagger_{q\tau} a_{p\sigma} = -a_{p\sigma} a^\dagger_{q\tau} + \delta_{pq} \delta_{\sigma\tau}$$

We have

$$\langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{q\tau} a_{p\sigma} \rangle = -\langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{p\sigma} a_{q\tau} \rangle = \langle a^\dagger_{p\sigma} a_{p\sigma} a^\dagger_{q\tau} a_{q\tau} \rangle - \delta_{pq} \delta_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{q\tau} \rangle$$

Then

$$\begin{aligned} e_{pqqp} \equiv \sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{q\tau} a_{p\sigma} \rangle &= -\sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{p\sigma} a_{q\tau} \rangle = \sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{p\sigma} a^\dagger_{q\tau} a_{q\tau} \rangle - \delta_{pq} \sum_{\sigma} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \\ &= \sum_{\sigma\tau} \langle n_{p\sigma} n_{q\tau} \rangle - \delta_{pq} \sum_{\sigma} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \end{aligned}$$

Therefore,

$$\boxed{C_{pq} \equiv \sum_{\sigma\tau} \langle n_{p\sigma} n_{q\tau} \rangle = e_{pqqp} + \delta_{pq} E_{pq}}$$

Similarly,

$$\begin{aligned} s_{pqqp} \equiv \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{q\tau} a_{p\sigma} \rangle &= -\sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{p\sigma} a_{q\tau} \rangle \\ &= \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle a^\dagger_{p\sigma} a_{p\sigma} a^\dagger_{q\tau} a_{q\tau} \rangle - \delta_{pq} \sum_{\sigma} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \\ &= \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle n_{p\sigma} n_{q\tau} \rangle - \delta_{pq} \sum_{\sigma} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \end{aligned}$$

Therefore,

$$\boxed{S_{pq} \equiv \sum_{\sigma\tau} (-1)^{1+\delta_{\sigma\tau}} \langle n_{p\sigma} n_{q\tau} \rangle = s_{pqqp} + \delta_{pq} E_{pq}}$$

1.4.5 Diagonal Spatial Two-Particle Density Matrix (Mixed Spin)

Using anticommutation relation

$$a^\dagger_{q\tau} a_{p\tau} = -a_{p\tau} a^\dagger_{q\tau} + \delta_{pq}$$

we have

$$\begin{aligned} e_{pqpq} \equiv \sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a^\dagger_{q\tau} a_{p\tau} a_{q\sigma} \rangle &= -\sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{p\tau} a^\dagger_{q\tau} a_{q\sigma} \rangle + \delta_{pq} \sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \\ &= -\sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{p\tau} a^\dagger_{q\tau} a_{q\sigma} \rangle + 2\delta_{pq} \sum_{\sigma} \langle a^\dagger_{p\sigma} a_{q\sigma} \rangle \end{aligned}$$

Therefore,

$$\boxed{\sum_{\sigma\tau} \langle a^\dagger_{p\sigma} a_{p\tau} a^\dagger_{q\tau} a_{q\sigma} \rangle = -e_{pqpq} + 2\delta_{pq} E_{pq}}$$

CHAPTER 2

pyblock API References

2.1 pyblock.symmetry

2.1.1 pyblock.symmetry.symmetry

Symmetry related data structures.

class pyblock.symmetry.symmetry.**DirectProdGroup**(**args*)
Bases: object

Irreducible representation for symmetry formed by direct product of sub-symmetries.

Attributes:

irs [list(Group)] A list of irreducible representations for sub-symmetries.

ng [int] Number of sub-symmetries.

sub_group(*idx*)
Return a subgroup of this object, given a list of indices.

class pyblock.symmetry.symmetry.**HashIrrep**(*ir*)
Bases: object

Base class for irreducible representation, supporting hashing.

class pyblock.symmetry.symmetry.**PGC1**(*ir*)
Bases: *pyblock.symmetry.symmetry.PointGroup*

C_1 point group.

Cached = [None]

InverseElem = array([0])

IrrepNames = ['A']

Table = array([[0]])

class pyblock.symmetry.symmetry.**PGC2**(*ir*)
Bases: *pyblock.symmetry.symmetry.PGCI*

C_2 point group.

IrrepNames = ['A', 'B']

class pyblock.symmetry.symmetry.**PGC2H**(*ir*)
Bases: *pyblock.symmetry.symmetry.PGC2V*

C_{2h} point group.

IrrepNames = ['Ag', 'Au', 'Bu', 'Bg']

class pyblock.symmetry.symmetry.**PGC2V**(*ir*)
Bases: *pyblock.symmetry.symmetry.PointGroup*

D_{c2v} point group.

Cached = [None, None, None, None]

InverseElem = array([0, 1, 2, 3])

IrrepNames = ['A1', 'B1', 'B2', 'A2']

Table = array([[0, 1, 2, 3], [1, 0, 3, 2], [2, 3, 0, 1], [3, 2, 1, 0]])

class pyblock.symmetry.symmetry.**PGCI**(*ir*)
Bases: *pyblock.symmetry.symmetry.PointGroup*

C_i point group.

Cached = [None, None]

InverseElem = array([0, 1])

IrrepNames = ['Ag', 'Au']

Table = array([[0, 1], [1, 0]])

class pyblock.symmetry.symmetry.**PGCS**(*ir*)
Bases: *pyblock.symmetry.symmetry.PGCI*

C_s point group.

IrrepNames = ["A'", "A''"]

class pyblock.symmetry.symmetry.**PGD2**(*ir*)
Bases: *pyblock.symmetry.symmetry.PGC2V*

D_2 point group.

IrrepNames = ['A1', 'B3', 'B2', 'B1']

class pyblock.symmetry.symmetry.**PGD2H**(*ir*)
Bases: *pyblock.symmetry.symmetry.PointGroup*

D_{2h} point group.

Cached = [None, None, None, None, None, None, None, None]

InverseElem = array([0, 1, 2, 3, 4, 5, 6, 7])

IrrepNames = ['Ag', 'B3u', 'B2u', 'B1g', 'B1u', 'B2g', 'B3g', 'Au']

Table = array([[0, 1, 2, 3, 4, 5, 6, 7], [1, 0, 3, 2, 5, 4, 7, 6], [2, 3, 0, 1, 6, 7,

class pyblock.symmetry.symmetry.**ParticleN**(*ir*)
Bases: *pyblock.symmetry.symmetry.HashIrrep*

Irreducible representation for particle number symmetry.

CachedM = []

CachedP = []

static clebsch_gordan(*a*, *b*, *c*)

multiplicity

class pyblock.symmetry.symmetry.**PointGroup**(*ir*)
Bases: *pyblock.symmetry.symmetry.HashIrrep*

Base class for irreducible representation for point group symmetry.

Attributes:

Table [rank-2 numpy.ndarray] Mutiplication table of the group.

InverseElem [rank-1 numpy.ndarray] Inverse Element of each element in the group.

IrrepNames [list(str)] Name of each irreducible representation in the group.

InverseElem = array([], dtype=int64)

IrrepNames = []

Table = array([], shape=(0, 0), dtype=int64)

static clebsch_gordan(*a*, *b*, *c*)

multiplicity

class pyblock.symmetry.symmetry.**SU2**(*s*)
Bases: *pyblock.symmetry.symmetry.HashIrrep*

Irreducible representation for SU(2) spin symmetry.

Cached = []

static clebsch_gordan(*a*, *b*, *c*)
Return rank-3 numpy.ndarray for CG coefficients with all possible projected quantum numbers.

j
SU(2) spin quantum number.

multiplicity

to_multi()

class pyblock.symmetry.symmetry.**SU2Proj**(*s*, *sz*)
Bases: *pyblock.symmetry.symmetry.SU2*

Irreducible representation for SU(2) spin symmetry with extra projected spin label.

Cached = []

copy()

jz
SU(2) projected spin quantum number.

multiplicity

static random_from_multi(*multi*)

to_multi()

class pyblock.symmetry.symmetry.**SZ**(*sz*)
Bases: *pyblock.symmetry.symmetry.HashIrrep*

Irreducible representation for projected spin symmetry.

CachedM = []

CachedP = []

static clebsch_gordan(*a*, *b*, *c*)

multiplicity

pyblock.symmetry.symmetry.**point_group**(*pg_name*)
Return point group class corresponding to the point group name.

2.1.2 pyblock.symmetry.cg

Clebsch-Gordan coefficients.

class pyblock.symmetry.cg.**CG**
Bases: object

Precomputed numerical constants.

Attributes:

NSqrtFact [int] Number of constants to generate.

SqrtFact [[float]] A list of size *NSqrtFact*, with element at index i being $\sqrt{i!}$.

NSqrtFact = 200

SqrtFact = [1.0, 1.0, 1.4142135623730951, 2.4494897427831783, 4.898979485566357, 10.95

class pyblock.symmetry.cg.**SU2CG**
Bases: object

SU(2) Clebsch-Gordan coefficients.

static clebsch_gordan(*ja*, *jb*, *jc*, *ma*, *mb*, *mc*)

static sqrt_delta(*ja*, *jb*, *jc*)

static wigner_3j(*ja*, *jb*, *jc*, *ma*, *mb*, *mc*)

2.1.3 pyblock.symmetry.basis

Basis transformation for site operators.

pyblock.symmetry.basis.**basis_transform**(*mat*, *q_label*, *old_basis*, *new_basis*)
Transform the matrix representation of an site operator from one basis to another basis, using Wigner-Eckart theorem.

2.2 pyblock.tensor

2.2.1 pyblock.tensor.tensor

Block-sparse tensor and tensor network.

class pyblock.tensor.tensor.**SubTensor**(*q_labels=None*, *reduced=None*, *cgs=None*)
Bases: object

A block in block-sparse tensor.

Attributes:

q_labels [tuple(DirectProdGroup..)] Quantum labels for this sub-tensor block. Each element in the tuple corresponds one rank of the tensor.

rank [int] Rank of the tensor. `rank == len(q_labels)`.

ng [int] Number of sub-symmetry groups. `ng == q_labels[0].ng`.

reduced [numpy.ndarray] Rank-`rank` dense reduced matrix.

reduced_shape [tuple(int..)] Shape of `reduced`

cgs [list(numpy.ndarray)] A list of CG factors of length `ng` with each element being Rank-`rank` dense reduced matrix representing CG coefficients for projected quantum numbers in each sub-symmetry group.

T
Transpose.

add_noise(*noise*)
Add noise to reduced matrix by random numbers in [-0.5 * noise, 0.5 * noise).

Args:

noise [float] prefactor for the noise.

build_random()
Set reduced matrix with random numbers in [0, 1).

build_rank3_cg()
Generate `cgs` for rank-3 tensor.

build_zero()
Set reduced matrix to zero.

equal_shape(*o*)

class pyblock.tensor.tensor.**Tensor**(*blocks=None*, *tags=None*, *contractor=None*)
Bases: object

Block-sparse tensor.

Attributes:

blocks [list(SubTensor)] A list of (non-zero) blocks.

tags [set(str or int)] Tags of the tensor for labeling the type or site index of the tensor.

contractor : If not None, this is used to perform specialized tensor contraction and diagonalization.

add_noise(*noise*)
Add noise to reduced matrix by random numbers in [-0.5 * noise, 0.5 * noise).

Args:

noise [float] prefactor for the noise.

add_tags(*tags*)
Add the tags, return `self` for chain notation.

align(*o*)

build_identity()
Set the reduced matrix to identity. Not work for general situations.

build_random()
Fill the reduced matrix with random numbers in [0, 1).

build_rank3_cg()
Generate CG coefficients for rank-3 tensor.

build_zero()
Fill the reduced matrix with zero.

static contract(*tsa*, *tsb*, *idxa*, *idxb*, *target_q_labels=None*)
Contract two Tensor to form a new Tensor.

Args:

tsa [Tensor] Tensor a, as left operand.

tsb [Tensor] Tensor b, as right operand.

idxa [list(int)] Indices to be contracted in tensor a.

idxb [list(int)] Indices to be contracted in tensor b.

target_q_labels [None or DirectProdGroup or [DirectProdGroup]] If None, this is contraction of states. For all other cases, this is contraction of operators. If DirectProdGroup, the resulting direct product operator will have the specific quantum label. If [DirectProdGroup], the resulting direct product operator will have mixed quantum labels (not very useful).

Returns:

tensor [Tensor] The contracted Tensor.

copy()
Shallow copy.

deep_copy()
Deep copy.

diag_eigs(*k=-1*, *limit=None*)

equal_shape(*o*)

fit(*o*, *v*)

fuse_index(*idxl*, *fused*, *target=None*, *equal=None*)

get_diag_density_matrix(*trace_right*, *noise=0.0*)

left_canonicalize(*mode='reduced'*)
Left canonicalization (using QR factorization).

Returns:

r_blocks [dict((DirectProdGroup,) -> numpy.ndarray)] The R matrix for each right-index quantum label.

left_multiply(*mats*)
Left Multiplication. Currently only used for multiplying R obtained from right-canonicalization.

Args:

mats [dict((DirectProdGroup,) -> numpy.ndarray)] The R matrix for each right-index quantum label.

modify(*other*)
Modify the blocks according to another Tensor's blocks.

n_blocks
Number of (non-zero) blocks.

ng
Number of sub-symmetry groups.

norm()

static operator_init(*basis*, *repr*, *op_q_labels*)
Build tensor for single-site primiary operator. When this is a tensor operator, `repr` and `op_q_labels` will contain multiple items. `cgs` and `reduced` will be calculated.

Args:

basis [list(DirectProdGroup)] Site basis.

repr [list(numpy.ndarray)] A list of dense matrices for different `op_q_label` s.

op_q_labels [list(DirectProdGroup)] A list of operator quantum labels.

Returns:

tensor [Tensor] Operator Tensor.

static partial_trace(*ts*, *idx_l*, *idx_r*, *target_q_labels=None*)
Partial trace of a tensor. The indices in `idx_l` will be combined. The indices in `idx_r` will also be combined. Then for each tensor block with q_label[idx_l] == q_label[idx_r], the term will be included, with its reduced matrix traced in corresponding indices.

Args:

ts [Tensor] Tensor to be traced.

idx_l [list(int)] Left traced indices.

idx_r [list(int)] Right traced indices.

target_q_labels [None] Defaults to None. Currently only the default is implemented.

Returns:

tensor [Tensor] The resulting Tensor with indices `idx_l` and `idx_r` are traced.

rank
Rank of the tensor.

static rank2_init_target(*left*, *right*, *target*)

static rank3_init_left(*pre*, *basis*, *post*)
Build the MPS rank-3 tensor by coupling states in `pre` and `basis` into states in `post`. `cgs` and `reduced` will not be calculated.

Args:

pre [dict(DirectProdGroup -> int)] Left basis.

basis [dict(DirectProdGroup -> int)] Site basis.

post [dict(DirectProdGroup -> int)] Right basis. Right basis should be obtained from truncating the direct product of left and site basis.

Returns: tensor : Tensor

static rank3_init_right(*pre*, *basis*, *post*)
Build the MPS rank-3 tensor by coupling states in `pre` and `basis` into states in `post`. `cgs` and `reduced` will not be calculated.

Args:

pre [dict(DirectProdGroup -> int)] Right basis.

basis [dict(DirectProdGroup -> int)] Site basis.

post [dict(DirectProdGroup -> int)] Left basis. Left basis should be obtained from truncating the direct product of right and site basis.

Returns: tensor : Tensor

right_canonicalize(*mode='reduced'*)
Right canonicalization (using LQ factorization).

Returns:

l_blocks [dict((DirectProdGroup,) -> numpy.ndarray)] The L matrix for each left-index quantum label.

right_multiply(*mats*)
Right Multiplication. Currently only used for multiplying L obtained from right-canonicalization.

Args:

mats [dict((DirectProdGroup,) -> numpy.ndarray)] The L matrix for each left-index quantum label.

set_contractor(*contractor*)
Change the contractor, return `self` for chain notation.

set_tags(*tags*)
Change the tags, return `self` for chain notation.

sort()
Sort non-zero blocks in increasing order of quantum labels.

split(*absorb_right*, *k=-1*)
Split rank-2 block-diagonal Tensor to two tensors (using SVD).

Args:

absorb_right [bool] If absorb_right is True, singlular values will be multiplied into right Tensor. Otherwise, They will be multiplied into left Tensor.

k [int] Maximal bond length. Defaults to -1 (no truncation).

Returns:

left_tensor [Tensor] Left tensor.

right_tensor [Tensor] Right tensor.

error [float] Sum of Discarded weights.

split_using_density_matrix(*dm*, *absorb_right*, *k=-1*, *limit=None*)

svd(*q_label_left*, *k=-1*)
Singular Value Decomposition of rank-2 block-diagonal Tensor.

Args:

k [int] Maximal bond length. Defaults to -1 (no truncation).

Returns:

left_tensor [Tensor] Left tensor.

right_tensor [Tensor] Right tensor.

svd_s [list(numpy.ndarray)] A list including singular values for each tensor block.

error [float] Sum of Discarded weights (square of singular values).

to_dict(*idx*)

unfuse_index(*idx*, *left*, *right*)

zero_copy()
A deep copy with zero reduced matrices.

class pyblock.tensor.tensor.**TensorNetwork**(*tensors=None*)
Bases: object

An inefficient implementation for Quimb TensorNetwork.

Attributes:

tensors [list(Tensor)] List of tensors in the network.

add(*tn*)
Add a Tensor or TensorNetwork to the tensor network.

add_tags(*tags*)
Add tags to all tensors in the tensor network.

contract(*tags*, *in_place=False*)
Contract a sub tensor network specified by tags. If any Tensor in the sub tensor network has a `contractor`, the specialized contraction will be used. Currently the general contraction is not implemented.

Args:

tags [tuple(str or int..)] Tags of sub tensor network.

Kwargs:

in_place [bool] Defaults to False. Indicating whether the current tensor network should be changed.

Returns:

tn [TensorNetwork] The tensor network with selected sub tensor network replaced by its contraction.

copy()
Shallow copy.

deep_copy()
Deep copy.

remove(*tags*, *which='all'*, *in_place=False*)
Remove a sub tensor network specified by tags.

Args:

tags [set(str or int)] Tags of sub tensor network.

Kwargs:

which ['all' or 'any' or 'exact'] Defaults to 'all'. Indicating how the `tags` should be applied for selection.

in_place [bool] Defaults to False. Indicating whether the current tensor network should be changed.

Returns:

tn [TensorNetwork] The remaining tensor network.

remove_tags(*tags*)
Remove tags from all tensors in the tensor network.

replace(*tags*, *tn*, *which='all'*)

select(*tags*, *which='all'*, *inverse=False*)
Extract a sub tensor network specified by tags.

Args:

tags [set(str or int)] Tags of sub tensor network.

Kwargs:

which ['all' or 'any' or 'exact'] Defaults to 'all'. Indicating how the `tags` should be applied for selection.

inverse [bool] Defaults to False. Indicating whether the selection should be inversed.

Returns:

tn [TensorNetwork] The selected tensor network.

set_contractor(*contractor*)
Set contractor for all tensors in the tensor network.

tags
Return a list of tags collected from all tensors in the tensor network.

zero_copy()
Deep copy with zeros.

exception pyblock.tensor.tensor.**TensorNetworkError**
Bases: `Exception`

2.3 pyblock.qchem

2.3.1 pyblock.qchem.core

Translation of low-level objects in Block code.

exception pyblock.qchem.core.**BlockError**
Bases: `Exception`

class pyblock.qchem.core.**BlockEvaluation**
Bases: `object`

classmethod eigen_values(*mat*)
Return all eigenvalues of a StackSparseMatrix.

classmethod expr_diagonal_eval(*expr*, *a*, *b*, *sts*)
Evaluate the diagonal elements of the result of a symbolic operator expression. The diagonal elements are required for perconditioning in Davidson algorithm.

Args:

expr [OpString or OpCollection or ParaOpCollection] The operator expression to evaluate.

a [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in left block to its matrix representation.

b [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in right block to its matrix representation.

sts [VectorStateInfo] StateInfo in which the result of the operator expression is represented.

Returns: diag : DiagonalMatrix

classmethod expr_eval(*expr*, *a*, *b*, *sts*, *q_label*, *nmat=0*)
Evaluate the result of a symbolic operator expression. The operator expression is usually a sum of direct products of operators in left and right blocks.

Args:

expr [OpString or OpSum] The operator expression to evaluate.

a [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in left block to its matrix representation.

b [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in right block to its matrix representation.

sts [VectorStateInfo] StateInfo in which the result of the operator expression is represented.

q_label [DirectProdGroup] Quantum label of the result operator (indicating how it changes the state quantum labels).

Returns: nmat : StackSparseMatrix

classmethod expr_expectation(*expr*, *a*, *b*, *ket*, *bra*, *work*, *sts*)

classmethod expr_multiply_eval(*expr*, *a*, *b*, *c*, *nwave*, *sts*)
Evaluate the result of a symbolic operator expression applied on a wavefunction.

Args:

expr [OpString or OpCollection or ParaOpCollection] The operator expression.

a [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in left block to its matrix representation.

b [dict(OpElement -> StackSparseMatrix)] A map from operator symbol in right block to its matrix representation.

c [Wavefunction] The input wavefuction.

nwave [Wavefunction] The output wavefuction.

sts [VectorStateInfo] StateInfo in which the wavefuction is represented.

expr_perturbative_density_eval(*expr*, *a*, *b*, *c*, *d*, *forward*, *sts*)

classmethod left_contract(*i*, *optl*, *optd*, *mpo_info*, *mps_info*, *bra_mps_info=None*)
Perform blocking MPO x MPO for left block.

Args:

i [int] Site index.

optl: OperatorTensor or DualOperatorTensor Contracted MPO operator tensor at previous left block.

optd [OperatorTensor or DualOperatorTensor] MPO operator tensor at dot block.

mpo_info [MPOInfo] MPOInfo object.

mps_info [MPSInfo] MPSInfo object.

bra_mps_info [MPSInfo or None (if same as mps_info)] MPSInfo object for bra state.

Returns:

new_opt [OperatorTensor or DualOperatorTensor] Operator tensor in untruncated basis in current left block.

classmethod left_right_contract(*i*, *optl*, *optr*, *mpo_info*, *tag*)
Symbolically construct the super block MPO.

Args:

i [int] Site index of first/left dot block.

optl: OperatorTensor Contracted MPO operator at (enlarged) left block.

optr: OperatorTensor Contracted MPO operator at (enlarged) right block.

mpo_info [MPOInfo] MPOInfo object.

tag [str] Extra tag for caching.

Returns:

new_opt [OperatorTensor] Operator tensor for super block. This method does not evaluate the super block operator experssion.

classmethod left_rotate(*i*, *opt*, *mpst*, *mps_info*, *bra_mpst=None*, *bra_mps_info=None*)
Perform rotation <MPS|MPO|MPS> for left block.

Args:

i [int] Site index.

opt [OperatorTensor or DualOperatorTensor] Operator tensor in (untruncated) old basis.

mpst [Tensor] MPS tensor defining rotation in ket side.

mps_info [MPSInfo] MPSInfo object for ket state.

bra_mpst [Tensor or None (if same as mpst)] MPS tensor defining rotation in bra side.

bra_mps_info [MPSInfo] MPSInfo object for bra state.

Returns:

new_opt [OperatorTensor or DualOperatorTensor] Operator tensor in (truncated) new basis.

parallelizer = None
Explicit evaluation of symbolic expression for operators.

classmethod right_contract(*i*, *optr*, *optd*, *mpo_info*, *mps_info*, *bra_mps_info=None*)
Perform blocking MPO x MPO for right block.

Args:

i [int] Site index.

optr: OperatorTensor or DualOperatorTensor Contracted MPO operator tensor at previous right block.

optd [OperatorTensor or DualOperatorTensor] MPO operator tensor at dot block.

mpo_info [MPOInfo] MPOInfo object.

mps_info [MPSInfo] MPSInfo object.

bra_mps_info [MPSInfo or None (if same as mps_info)] MPSInfo object for bra state.

Returns:

new_opt [OperatorTensor or DualOperatorTensor] Operator tensor in untruncated basis in current right block.

classmethod right_rotate(*i*, *opt*, *mpst*, *mps_info*, *bra_mpst=None*, *bra_mps_info=None*)
Perform rotation <MPS|MPO|MPS> for right block.

Args:

i [int] Site index.

opt [OperatorTensor or DualOperatorTensor] Operator tensor in (untruncated) old basis.

mpst [Tensor] MPS tensor defining rotation in ket side.

mps_info [MPSInfo] MPSInfo object for ket state.

bra_mpst [Tensor or None (if same as mpst)] MPS tensor defining rotation in bra side.

bra_mps_info [MPSInfo or None (if same as mps_info)] MPSInfo object for bra state.

Returns:

new_opt [OperatorTensor or DualOperatorTensor] Operator tensor in (truncated) new basis.

simplifier = <pyblock.qchem.simplifier.NoSimplifier object>

classmethod tensor_rotate(*opt*, *sts*, *rmats*)
Transform basis of MPO using rotation matrix.

Args:

opt [OperatorTensor or DualOperatorTensor] Operator tensor in (untruncated) old basis.

sts [VectorStateInfo] Old (untruncated) and new (truncated) basis.

rmats [VectorVectorMatrix] Rotation matrices for ket (or bra and ket).

Returns:

new_opt [OperatorTensor or DualOperatorTensor] Operator tensor in (truncated) new basis.

class pyblock.qchem.core.**BlockHamiltonian**(**args*, ***kwargs*)
Bases: object

Initialization of block code.

Attributes:

output_level [int] Output level of block code.

point_group [str] Point group of molecule.

n_sites [int] Number of sites/orbitals.

n_electrons [int] Number of electrons.

target_s [Fraction] SU(2) quantum number of target state.

spatial_syms [[int]] point group irrep number at each site.

target_spatial_sym [int] point group irrep number of target state.

dot [int] two-dot (2) or one-dot (1) scheme.

t [TInt] One-electron integrals.

v [VInt] Two-electron integrals.

e [float] Const term in energy.

static block_operator_summary(*block*)
Return a summary str of operators included in one Block (block code).

static get(*fcidump*, *pg*, *su2*, *dot=2*, *output_level=0*, *memory=2000*, *page=None*, *omp_threads=1*, ***kwargs*)

static get_current_memory()
Return current stack memory position.

get_site_operators(*m*, *op_set*)
Return operator representations dict(OpElement -> StackSparseMatrix) at site m.

static release_memory()

static set_current_memory(*m*)
Reset current stack memory to given position.

class pyblock.qchem.core.**BlockSymmetry**
Bases: object

Including functions for translating quantum label related objects.

classmethod from_spin_quantum(*sq*)
Translate from SpinQuantum (block code) to DirectProdGroup.

classmethod from_state_info(*state_info*)
Translate from StateInfo (block code) to [(DirectProdGroup, int)].

classmethod initial_state_info(*i=0*)
Return StateInfo for site basis at site i.

classmethod to_spin_quantum(*dpg*)
Translate from DirectProdGroup to SpinQuantum (block code).

classmethod to_state_info(*states*)
Translate from [(DirectProdGroup, int)] to StateInfo (block code).

2.3.2 pyblock.qchem.fcidump

FCIDUMP file and storage of integrals.

class pyblock.qchem.fcidump.**GVInt**(*n*)
Bases: *pyblock.qchem.fcidump.TInt*

General rank-4 array for two-electron integral storage.

Attributes:

n [int] Number of orbitals.

data [numpy.ndarray] 1D flat array of size n^4.

class pyblock.qchem.fcidump.**TInt**(*n*)
Bases: object

Symmetric rank-2 array ($T_{ij} = T_{ji}$) for one-electron integral storage.

Attributes:

n [int] Number of orbitals.

data [numpy.ndarray] 1D flat array of size $n(n+1)/2$.

copy()

find_index(*i*, *j*)
Find linear index from full indices (i, j).

class pyblock.qchem.fcidump.**UVInt**(*n*)
Bases: *pyblock.qchem.fcidump.TInt*

Symmetric rank-4 array ($V_{ijkl} = V_{jikl} = V_{ijlk}$) for two-electron integral storage.

Attributes:

n [int] Number of orbitals.

data [numpy.ndarray] 1D flat array of size m^2 where $m = n(n+1)/2$.

find_index(*i*, *j*, *k*, *l*)
Find linear index from full indices (i, j, k, l).

class pyblock.qchem.fcidump.**VInt**(*n*)
Bases: *pyblock.qchem.fcidump.TInt*

Symmetric rank-4 array ($V_{ijkl} = V_{jikl} = V_{ijlk} = V_{klij}$) for two-electron integral storage.

Attributes:

n [int] Number of orbitals.

data [numpy.ndarray] 1D flat array of size $m(m+1)/2$ where $m = n(n+1)/2$.

find_index(*i*, *j*, *k*, *l*)
Find linear index from full indices (i, j, k, l).

pyblock.qchem.fcidump.**read_fcidump**(*filename*)
Read FCI options and integrals from FCIDUMP file.

Args: filename : str

Returns:

cont_dict [dict] FCI options or input parameters.

(t, v, e) [(TInt, VInt, float)] One- and two-electron integrals and const energy.

pyblock.qchem.fcidump.**write_fcidump**(*filename*, *h1e*, *h2e*, *nmo*, *nelec*, *nuc*, *ms*, *isym=1*, *orbsym=None*, *tol=1e-15*)

2.3.3 pyblock.qchem.operator

Symbolic operators.

class pyblock.qchem.operator.**OpElement**(*name*, *site_index*, *factor=1*, *q_label=None*)
Bases: *pyblock.qchem.operator.OpExpression*

Single operator symbol.

Attributes:

name [*OpNames*] Type of the operator.

site_index [() or tuple(int..)] Site indices of the operator.

factor [float] scalar factor.

q_label [DirectProdGroup] Quantum label of the operator.

Cached = {}

factor

name

static parse(*expr*)
Parse a str to operator symbol.

static parse_site_index(*expr*)

q_label

site_index

class pyblock.qchem.operator.**OpExpression**
Bases: object

class pyblock.qchem.operator.**OpNames**
Bases: enum.Enum

Operator Names.

A = 12

AD = 13

B = 16

C = 6

D = 7

H = 1

I = 2

N = 3

NN = 4

NUD = 5

P = 14

PD = 15

PDM1 = 18

Q = 17

R = 10

RD = 11

S = 8

SD = 9

class pyblock.qchem.operator.**OpString**(*ops*, *factor=1*)
Bases: *pyblock.qchem.operator.OpExpression*

String of operator symbols representing direct product of single operator symbols.

Attributes:

ops [list(*OpElement*)] A list of single operator symbols.

sign [int (1 or -1)] Sign factor. With SU(2) factor considered

factor

op

ops

class pyblock.qchem.operator.**OpSum**(*strings*)
Bases: *pyblock.qchem.operator.OpExpression*

Sum of direct product of single operator symbols.

Attributes: strings : list(*OpString*)

strings

2.3.4 pyblock.qchem.mps

Matrix Product State for quantum chemistry calculations.

class pyblock.qchem.mps.**LineCoupling**(*n_sites*, *basis*, *empty*, *target*)
Bases: object

set_bond_dimension(*m*, *exact=False*)
Truncate the renormalized basis, using the given bond dimension. Note that the ceiling is used for rounding for each quantum number, so the actual bond dimension is often larger than the given value.

set_bond_dimension_using_occ(*m*, *occ*, *bias=1*)

tensor_product(*p*, *q*)

class pyblock.qchem.mps.**MPS**(*lcp*, *center*, *dot=2*, *iprint=False*, *tensors=None*)
Bases: *pyblock.tensor.tensor.TensorNetwork*

Matrix Product State.

canonicalize(*random=False*)
Canonicalization.

deep_copy()
Deep copy.

fill_identity()
Fill MPS reduced matrices with identity matrices whenever possible.

fit(*o*, *v*)

static from_tensor_network(*tn*, *mps_info*, *center*, *dot=2*)

randomize()
Fill MPS reduced matrices with random numbers in [0, 1).

update_line_coupling()

zero_copy()
Deep copy with zeros.

class pyblock.qchem.mps.**MPSInfo**(*lcp*)
Bases: object

from_left_rotation_matrix(*i*, *rot*)
Translate rotation matrix (block code) in left block to rank-3 Tensor.

Args:

i [int] Site index.

rot [VectorMatrix] Rotation matrix, defining the transformation from untruncated (but collected) basis to truncated basis.

Returns: tensor : class:*Tensor*

from_right_rotation_matrix(*i*, *rot*)
Translate rotation matrix (block code) in right block to rank-3 `Tensor`.

Args:

i [int] Site index.

rot [VectorMatrix] Rotation matrix, defining the transformation from untruncated (but collected) basis to truncated basis.

Returns: tensor : class:*Tensor*

from_wavefunction_fused(*i*, *wfn*, *sts=None*)
Construct rank-2 `Tensor` with fused indices from Wavefunction (block code).

Args:

i [int] Site index of first/left dot.

wfn [Wavefunction] Wavefunction.

Returns:

tensor [class:*Tensor*] In two-dot scheme, the rank-2 tensor representing two-dot object. Both left and right rank indices are fused. No CG factor are generated. One-dot scheme is not implemented.

get_left_rotation_matrix(*i*, *tensor*)
Translate rank-3 `Tensor` to rotation matrix (block code) in left block.

Args:

i [int] Site index.

tensor [class:*Tensor*] MPS tensor.

Returns:

rot [VectorMatrix] Rotation matrix, defining the transformation from untruncated (but collected) basis to truncated basis.

get_left_state_info(*i*, *left=None*)
Construct StateInfo for left block [0..i] (used internally)

get_right_rotation_matrix(*i*, *tensor*)
Translate rank-3 `Tensor` to rotation matrix (block code) in right block.

Args:

i [int] Site index.

tensor [class:*Tensor*] MPS tensor.

Returns:

rot [VectorMatrix] Rotation matrix, defining the transformation from untruncated (but collected) basis to truncated basis.

get_right_state_info(*i*, *right=None*)
Construct StateInfo for right block [i..:attr:*n_sites*-1] (used internally)

get_wavefunction_fused(*i*, *tensor*, *dot*, *sts=None*)
Construct Wavefunction (block code) from rank-2 `Tensor`.

Args:

i [int] Site index of first/left dot.

tensors [[Tensor]] Rank-2 `Tensor` with fused indices.

dot [int] One-dot or two-dot (default) scheme.

Returns: wfn : Wavefunction

update_local_left_block_basis(*i*, *left_block_basis*)
Update renormalized basis at site i and associated StateInfo objects.

This will update for left block with sites [0..i] and [0..i+1]

Args:

i [int] Center site, for determining assocated left and right blocks.

left_block_basis [[(DirectProdGroup, int)]] Renormalized basis for left block with sites [0..i].

update_local_left_state_info(*i*, *left=None*)
Update StateInfo objects for left block ending at site i.

Args:

i [int] Last site in the left block.

Kwargs:

left [StateInfo] The (optional) StateInfo object for previous left block. Defaults to None.

Returns:

right [StateInfo] The StateInfo object for current left block.

update_local_right_block_basis(*i*, *right_block_basis*)
Update renormalized basis at site i and associated StateInfo objects.

This will update for right block with sites [i+1..:attr:*n_sites*-1] and [i..:attr:*n_sites*-1].

Args:

i [int] Center site, for determining assocated left and right blocks.

right_block_basis [[(DirectProdGroup, int)]] Renormalized basis for right block with sites [i+1..:attr:*n_sites*-1].

update_local_right_state_info(*i*, *right=None*)
Update StateInfo objects for right block starting at site i.

Args:

i [int] First site in the right block.

Kwargs:

right [StateInfo] The (optional) StateInfo object for previous right block. Defaults to None.

Returns:

left [StateInfo] The StateInfo object for current right block.

pyblock.qchem.mps.**random_choice**(*data*, *m*)

2.3.5 pyblock.qchem.mpo

Matrix Product Operator for quantum chemistry calculations.

class pyblock.qchem.mpo.**DualOperatorTensor**(*lmat=None*, *rmat=None*, *ops=None*, *tags=None*, *contractor=None*)

Bases: *pyblock.tensor.tensor.Tensor*

MPO tensor or contracted MPO tensor with dual (left and right) representation.

copy()

Return shallow copy of this object.

class pyblock.qchem.mpo.**IdentityMPO**(*hamil*, *iprint=False*)

Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.mpo.**IdentityMPOInfo**(*hamil*, *cache_contraction=True*)

Bases: *pyblock.qchem.mpo.MPOInfo*

class pyblock.qchem.mpo.**LocalMPO**(*hamil*, *op_name*, *site_index=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.mpo.**LocalMPOInfo**(*hamil*, *op_name*, *site_index=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPOInfo*

class pyblock.qchem.mpo.**MPO**(*hamil*, *iprint=False*)

Bases: *pyblock.tensor.tensor.TensorNetwork*

class pyblock.qchem.mpo.**MPOInfo**(*hamil*, *cache_contraction=True*)

Bases: `object`

class pyblock.qchem.mpo.**OperatorTensor**(*mat*, *ops*, *tags=None*, *contractor=None*)

Bases: *pyblock.tensor.tensor.Tensor*

Represent MPO tensor or contracted MPO tensor.

Attributes:

mat [numpy.ndarray(dtype=OpExpression)] 2-D array of Symbolic operator expressions.

ops [dict(OpElement -> StackSparseMatrix)] Numeric representation of operator symbols. When the object is the super block MPO, `ops` is a pair of dicts representing operator symbols for left and right blocks, respectively.

copy()

Return shallow copy of this object.

class pyblock.qchem.mpo.**ProdMPO**(*hamil*, *opa_name*, *opb_name*, *opab_name*, *site_index_a=()*, *site_index_b=()*, *site_index_ab=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.mpo.**ProdMPOInfo**(*hamil*, *opa_name*, *opb_name*, *opab_name*, *site_index_a=()*, *site_index_b=()*, *site_index_ab=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPOInfo*

class pyblock.qchem.mpo.**SquareMPO**(*hamil*, *op_name*, *opsq_name*, *site_index=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.mpo.**SquareMPOInfo**(*hamil*, *op_name*, *opsq_name*, *site_index=()*, ***kwargs*)

Bases: *pyblock.qchem.mpo.MPOInfo*

2.3.6 pyblock.qchem.contractor

Specialized MPS/MPO operations for DMRG.

class pyblock.qchem.contractor.**BlockMultiplyH**(*opt*, *sts*, *diag=True*)
Bases: object

A wrapper of Block.MultiplyH (block code) for Davidson algorithm.

Attributes:

opt [OperatorTensor] The (symbolic) super block Hamiltonian.

st [StateInfo] StateInfo of super block.

diag_mat [DiagonalMatrix] Diagonal elements of super block Hamiltonian, in flatten form with no quantum labels.

apply(*other*, *result*)
Perform $\hat{H}|\psi\rangle$.

Args:

other [BlockWavefunction] Input vector/wavefunction.

result [BlockWavefunction] Output vector/wavefunction.

diag()
Returns Diagonal elements (for preconditioning).

diag_norm()

expect(*ket*, *bra*)

class pyblock.qchem.contractor.**BlockWavefunction**(*wave*, *factor=1.0*)
Bases: object

A wrapper of Wavefunction (block code) for Davidson algorithm.

clear_copy()
Return a deep copy of this object, but all the matrix elements are set to zero.

copy()
Return a deep copy of this object.

copy_data(*other*)
Fill the matrix elements in this object with data from another *BlockWavefunction* object.

deallocate()
Deallocate the memory associated with this object.

dot(*other*)
Return dot product of two *BlockWavefunction*.

normalize()
Normalization.

precondition(*ld*, *diag*)
Apply precondition on this object.

Args:

ld [float] Eigenvalue.

diag [DiagonalMatrix] Diagonal elements of Hamiltonian.

ref

exception pyblock.qchem.contractor.**ContractionError**
Bases: Exception

class pyblock.qchem.contractor.**DMRGContractor**(*mps_info*, *mpo_info*, *simplifier=None*, *parallelizer=None*, *davidson_tol=5e-06*)
Bases: object

bra_mps_info is MPSInfo of some constant MPS.

apply(*opt*, *mpst*)

bond_left(*tags={}*)

bond_right(*tags={}*)

bond_upper_limit_left(*tags={}*)

bond_upper_limit_right(*tags={}*)

contract(*tn*, *tags*)
Tensor network contraction.

Args:

tn [TensorNetwork] Part of tensor network to be contracted.

tags [(str, int) or (str,)] Tags of the tensor network to be contracted. If tags = ('_LEFT', i), the contraction is corresponding to blocking and renormalizing left block at site i. If tags = ('_RIGHT', i), the contraction is corresponding to blocking and renormalizing right block at site i. If tags = ('_HAM'), the contraction is corresponding to blocking both left and right block and forming the super block hamiltonian.

Returns:

mpo [OperatorTensor] The contracted MPO tensor.

eigs(*opt*, *mpst*)
Davidson diagonalization.

Args:

opt [OperatorTensor] Super block contracted operator tensor.

mpst [Tensor] Contracted MPS tensor in dot blocks.

Returns:

energy [float] Ground state energy.

v [class:*Tensor*] In two-dot scheme, the rank-2 tensor representing two-dot object. Both left and right rank indices are fused. One-dot scheme is not implemented.

ndav [int] Number of Davidson iterations.

expect(*opt*, *brat*, *kett*)

expo_apply(*opt*, *mpst*, *beta*)

fuse_left(*i*, *tensor*, *original_form*)

fuse_right(*i*, *tensor*, *original_form*)

perturbative_noise(*opt*, *mpst*)

post_sweep()
Operations performed at the end of each DMRG sweep.

pre_sweep()
Operations performed at the beginning of each DMRG sweep.

unfuse_left(*i*, *tensor*)

unfuse_right(*i*, *tensor*)

update_local_left_mps_info(*i*, *l_fused*)
Update `info` for site i using the left tensor from SVD.

update_local_right_mps_info(*i*, *r_fused*)
Update `info` for site i using the right tensor from SVD.

class pyblock.qchem.contractor.**DMRGDataPage**(*save_dir='node0'*, *n_frames=1*)
Bases: *pyblock.qchem.contractor.DataPage*

Determine how to swap data between disk and memory for DMRG calculation.

activate(*tags*, *reset=False*)
Activate one data page in memory for writing data.

clean()
Delete all temporary files.

get()

initialize()
Allocate memory for all pages.

load(*tags*)
Load data page from disk to memory, for reading data.

release()
Deallocate memory for all pages.

save(*tags*)
Save the data page in memory to disk.

unload(*tags*)
Unload data page in memory for reading data.

class pyblock.qchem.contractor.**DataPage**
Bases: `object`

activate(*tags*, *reset=None*)

get()

initialize(*tags*)

load(*tags*)

release(*tags*)

save(*tags*)

unload(*tags*)

2.3.7 pyblock.qchem.simplifier

Rules for simplifying symbolic operator expressions.

class pyblock.qchem.simplifier.**AllRules**(*su2=True*)
Bases: *pyblock.qchem.simplifier.Rule*

class pyblock.qchem.simplifier.**NoSimplifier**
Bases: object

No simplification is performed.

simplify(*zipped*)

class pyblock.qchem.simplifier.**NoTransposeRules**(*su2=True*, *rule=None*)
Bases: *pyblock.qchem.simplifier.Rule*

class pyblock.qchem.simplifier.**OpCollection**(*uniq*, *linked=None*)
Bases: object

class pyblock.qchem.simplifier.**OpLink**(*op*, *trans*, *scale*)
Bases: object

class pyblock.qchem.simplifier.**OpShell**(*data*)
Bases: object

class pyblock.qchem.simplifier.**PDM1Rules**(*su2=True*)
Bases: *pyblock.qchem.simplifier.Rule*

class pyblock.qchem.simplifier.**Rule**(*f=<function Rule.<lambda>>*)
Bases: object

class pyblock.qchem.simplifier.**RuleSU2**
Bases: object

class A(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class B(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class D(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class P(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class PDM1(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class Q(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class R(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class pyblock.qchem.simplifier.**RuleSZ**
Bases: object

class A(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class B(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class D(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class P(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class PDM1(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class Q(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class R(*f=<function Rule.<lambda>>*)
Bases: *pyblock.qchem.simplifier.Rule*

class pyblock.qchem.simplifier.**Simplifier**(*rule*)
Bases: object

Simplify complementary operators using symmetry properties.

simplify(*zipped*)

2.3.8 pyblock.qchem.parallelizer

MPI parallelization.

class pyblock.qchem.parallelizer.**ParaOpCollection**(*uniq*, *linked=None*, *partial=None*, *collect=None*, *broadcast=None*, *bcast_all=False*)
Bases: *pyblock.qchem.simplifier.OpCollection*

class pyblock.qchem.parallelizer.**ParaProperty**(*owner*, *repeated*, *repeated_num*, *partial*)
Bases: object

avail

class pyblock.qchem.parallelizer.**ParaRule**(*size=<sphinx.ext.autodoc.importer._MockObject object>*)
Bases: object

class pyblock.qchem.parallelizer.**Parallelizer**(*rule*, *rank=<sphinx.ext.autodoc.importer._MockObject object>*)
Bases: object

parallelize(*op_coll*, *do_partial=False*, *bcast_all=False*)

2.3.9 pyblock.qchem.thermal

Setting up integral for calculating thermal quantities.

class pyblock.qchem.thermal.**FreeEnergy**(*hamil*)
Bases: object

set_energy()

set_free_energy(*mu*)

set_particle_number()

2.3.10 pyblock.qchem.occupation

Initialize quantum numbers using occupation numbers.

class pyblock.qchem.occupation.**Occupation**(*occ*, *n_sites*, *basis*, *empty*, *target*, *bias=1*)
Bases: object

set_bond_dimension(*fci_l*, *fci_r*, *m*)

tensor_product(*p*, *q*)

2.3.11 pyblock.qchem.ancilla

Ancilla approach for finite temperature simulation.

class pyblock.qchem.ancilla.mps.**AncillaLineCoupling**(*n_sites*, *basis*, *empty*, *target*)
Bases: *pyblock.qchem.mps.LineCoupling*

set_thermal_limit()

class pyblock.qchem.ancilla.mps.**AncillaMPS**(*lcp*, *center*, *dot=2*, *iprint=False*, *tensors=None*)
Bases: *pyblock.qchem.mps.MPS*

fill_thermal_limit()

class pyblock.qchem.ancilla.mpo.**Ancilla**(*cls*, *npdm=False*)
Bases: object

MPO/MPOInfo Class decorator for adding ancilla sites.

static NPDM(*cls*)

2.3.12 pyblock.qchem.npdm

N-particle density matrix.

MPO for N-particle density matrix.

class pyblock.qchem.npdm.mpo.**NRMMPO**(*hamil*, *iprint=False*)
Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.npdm.mpo.**NRMMPOInfo**(*hamil*, *cache_contraction=True*)
Bases: *pyblock.qchem.mpo.MPOInfo*

class pyblock.qchem.npdm.mpo.**PDM1MPO**(*hamil*, *iprint=False*)
Bases: *pyblock.qchem.mpo.MPO*

class pyblock.qchem.npdm.mpo.**PDM1MPOInfo**(*hamil*, *cache_contraction=True*)
Bases: *pyblock.qchem.mpo.MPOInfo*

2.4 pyblock.legacy

2.4.1 pyblock.legacy.block_dmrg

Python wrapper for high-level DMRG algorithm in block code.

class pyblock.legacy.block_dmrg.**DMRG**(*input_file*, *output_level=0*)
Bases: object

Block DMRG in its original workflow.

block_and_decimate(*warm_up*, *dot_with_sys*)
Block and renormalize operators in Block.

dmrg(*gen_block=False*, *rot_mats=None*, *tol=None*, *forward=True*)
Perform DMRG.

do_one(*warm_up*, *forward*, *restart=False*, *restart_size=0*)
Perform one sweep.

finalize()
Release stack memory.

gen_block_block_and_decimate(*dot_with_sys*, *rot_mats=None*, *forward=True*, *implicit_trans=True*, *do_norms=None*, *do_comp=None*)
Blocking and renormalization step for generating operators from rotation matrix.

gen_block_do_one(*rot_mats=None*, *forward=True*, *implicit_trans=True*, *do_norms=None*, *do_comp=None*)
Perform one sweep for generating blocks from rotation matrices.

integral_index = 0

output_level = 0

sweep_params = None

system = None

2.5 pyblock.numerical

2.5.1 pyblock.numerical.davidson

Davidson diagonalization algorithm.

exception pyblock.numerical.davidson.**DavidsonError**
Bases: Exception

class pyblock.numerical.davidson.**Matrix**(*arr*)
Bases: object

General interface of Matrix for Davidson algorithm.

apply(*other*, *result*)
Perform $\hat{H}|\psi\rangle$.

Args:

other [Vector] Input vector.

result [Vector] Output vector.

diag()
Diagonal elements.

diag_norm()

class pyblock.numerical.davidson.**Vector**(*arr*, *factor=1.0*)
Bases: object

General interface of Vector for Davidson algorithm

clear_copy()
Return a deep copy of this object, but all the matrix elements are set to zero.

copy()
Return a deep copy of this object.

copy_data(*other*)
Fill the matrix elements in this object with data from another *Vector* object.

deallocate()
Deallocate the memory associated with this object. This is no-op for numpy.ndarray backend used here.

dot(*other*)
Dot product.

normalize()
Normalization.

precondition(*ld*, *diag*)
Apply precondition on this object.

Args:

ld [float] Eigenvalue.

diag [numpy.ndarray] Diagonal elements of Hamiltonian, stored in 1D array.

ref

pyblock.numerical.davidson.**davidson**(*a*, *b*, *k*, *max_iter=500*, *conv_thold=5e-06*, *deflation_min_size=2*, *deflation_max_size=30*, *iprint=False*, *mpi=False*)
Davidson diagonalization.

Args:

a [Matrix] The matrix to diagonalize.

b [list(Vector)] The initial guesses for eigenvectors.

Kwargs:

max_iter [int] Maximal number of davidson iteration.

conv_thold [float] Convergence threshold for squared norm of eigenvector.

deflation_min_size [int] Sub-space size after deflation.

deflation_max_size [int] Maximal sub-space size before deflation.

iprint [bool] Indicate whether davidson iteration information should be printed.

Returns:

ld [list(float)] List of eigenvalues.

b [list(Vector)] List of eigenvectors.

pyblock.numerical.davidson.**olsen_precondition**(*q*, *c*, *ld*, *diag*)
Olsen precondition.

2.5.2 pyblock.numerical.expo

pyblock.numerical.expo.**expo**(*a*, *b*, *beta*, *const_a=0.0*, *expo_tol=0*, *deflation_max_size=20*)
Calculate exp(-beta (a + const_a)) b.

pyblock.numerical.expo.**stdout_redirected**(*to='/dev/null'*)

2.6 pyblock.algorithm

2.6.1 pyblock.algorithm.dmrg

DMRG algorithm.

class pyblock.algorithm.dmrg.**DMRG**(*mpo*, *mps*, *bond_dims*, *noise=0.0*, *contractor=None*)
Bases: object

DMRG algorithm.

Attributes:

n_sites [int] Number of sites/orbitals

dot [int] Two-dot (2) or one-dot (1) scheme.

bond_dims [list(int) or int] Bond dimension for each sweep.

noise [list(float) or float] Noise prefactor for each sweep.

energies [list(float)] Energies collected for all sweeps.

blocking(*i*, *forward*, *bond_dim*, *noise*)
Perform one blocking iteration.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

Returns:

energy [float] Ground state energy.

error [float] Sum of discarded weights.

ndav [int] Number of Davidson iterations.

construct_envs()

set_mps(*tags*, *wfn*)

solve(*n_sweeps*, *tol*, *forward=True*, *two_dot_to_one_dot=-1*)
Perform DMRG algorithm.

Args:

n_sweeps [int] Maximum number of sweeps.

tol [float] Energy convergence threshold.

forward [bool] Direction of first sweep. If True, sweep is performed from left to right.

two_dot_to_one_dot [int or -1] Indicating when to switch to one-dot scheme. If -1, no switching.

Returns:

energy [float] Final ground state energy.

sweep(*forward*, *bond_dim*, *noise*)
Perform one sweep iteration.

Args:

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dims [int] Bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

Returns:

energy [float] Ground state energy.

update_one_dot(*i*, *forward*, *bond_dim*, *noise*)
Update local site in one-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

Returns:

energy [float] Ground state energy.

error [float] Sum of discarded weights.

ndav [int] Number of Davidson iterations.

update_two_dot(*i*, *forward*, *bond_dim*, *noise*)
Update local sites in two-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

Returns:

energy [float] Ground state energy.

error [float] Sum of discarded weights.

ndav [int] Number of Davidson iterations.

exception pyblock.algorithm.dmrg.**DMRGError**
Bases: Exception

class pyblock.algorithm.dmrg.**MovingEnvironment**(*n_sites*, *center*, *dot*, *tn*, *iprint=True*)
Bases: object

Environment blocks in DMRG.

Attributes:

n_sites [int] Number of sites/orbitals.

dot [int] Two-dot (2) or one-dot (1) scheme.

tnc [TensorNetwork] The tensor network <bra|H|ket> before contraction.

pos [int] Current site position of left dot.

envs [dict(int -> TensorNetwork)] DMRG Environment for different positions of left dot.

init_environments()
Initialize DMRG Environment blocks by contraction.

move_to(*i*)
Change the current left dot site to `i` (by zero or one site).

prepare_sweep(*dot*, *pos*)
Prepare environment for next sweep.

pyblock.algorithm.dmrg.**pprint**(**args*, ***kwargs*)

2.6.2 pyblock.algorithm.time_evolution

Imaginary time evolution algorithm.

class pyblock.algorithm.time_evolution.**ExpoApply**(*mpo*, *mps*, *beta*, *bond_dims*, *contractor=None*, *canonical_form=None*)

Bases: object

Apply exp(beta H) to MPS (tangent space approach).

Attributes:

n_sites [int] Number of sites/orbitals

dot [int] Two-dot (2) or one-dot (1) scheme.

bond_dims [list(int) or int] Bond dimension for each sweep.

energies [list(float)] Energies collected for all sweeps.

canonical_form [list(str)] The canonical form of initial MPS. If None, assumming it is LL..CC..RR (dot=2) or LL..C..RR (dot=1)

blocking(*i*, *forward*, *bond_dim*, *beta*)
Perform one blocking iteration.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

beta [float] Beta parameter in $\exp(-\beta H)$.

Returns:

energy [float] Energy of state $\exp(-\beta H)|\psi\rangle$.

normsq [float] Self inner product of state $\exp(-\beta H)|\psi\rangle$.

error [float] Sum of discarded weights.

nexpos [(int, int)] Number of operator multiplication in forward and backward exponential step.

construct_envs()

set_mps(*tags*, *wfn*)

solve(*n_sweeps*, *forward=True*, *two_dot_to_one_dot=-1*, *current_beta=0.0*, *iprint=True*)
Perform time evolution algorithm.

Args:

n_sweeps [int] Maximum number of sweeps (two sweeps will calculate one exp(-beta H) application)

forward [bool] Direction of first sweep. If True, sweep is performed from left to right.

two_dot_to_one_dot [int or -1] Indicating when to switch to one-dot scheme. If -1, no switching.

Returns:

energy [float] Energy of state $\exp(-\beta * (n_{sweeps}/2)H)|\psi\rangle$.

sweep (*forward*, *bond_dim*, *beta*)
Perform one sweep iteration.

Args:

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

beta [float] Beta parameter in $\exp(-\beta H)$.

Returns:

energy [float] Energy of state $\exp(-\beta H)|\psi\rangle$.

normsq [float] Self inner product of state $\exp(-\beta H)|\psi\rangle$.

error [float] Largest sum of discarded weights.

update_one_dot (*i*, *forward*, *bond_dim*, *beta*)
Update local site in one-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

beta [float] Beta parameter in $\exp(-\beta H)$.

Returns:

energy [float] Energy of state $\exp(-\beta H)|\psi\rangle$.

normsq [float] Self inner product of state $\exp(-\beta H)|\psi\rangle$.

error [float] Sum of discarded weights.

nexpos [(int, int)] Number of operator multiplication in forward and backward exponential step.

update_two_dot (*i*, *forward*, *bond_dim*, *beta*)
Update local sites in two-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bond dimension of current sweep.

beta [float] Beta parameter in $\exp(-\beta H)$.

Returns:

energy [float] Energy of state $\exp(-\beta H)|\psi\rangle$.

normsq [float] Self inner product of state $\exp(-\beta H)|\psi\rangle$.

error [float] Sum of discarded weights.

nexpos [(int, int)] Number of operator multiplication in forward and backward exponential step.

exception pyblock.algorithm.time_evolution.**TEError**
Bases: Exception

pyblock.algorithm.time_evolution.**pprint**(**args*, ***kwargs*)

2.6.3 pyblock.algorithm.expectation

Expectation calculation for <MPS1|MPO|MPS2>.

class pyblock.algorithm.expectation.**Expect**(*mpo*, *bra_mps*, *ket_mps*, *bra_canonical_form=None*, *ket_canonical_form=None*, *contractor=None*)
Bases: object

Calculation of expectation value <MPS1|MPO|MPS2>. The expectation value can be evaluated at current canonical form, when *forward is None*. Otherwise, it is recommended that *bond_dim* (bond dimension) of the MPS is given, and one sweep will be performed and expectation value will be evaluated at each canonical form. The sweep will thus change the canonical form of MPS and MPSInfo in contractor. Therefore, it is recommended that a copy of MPS and MPSInfo is used here.

Attributes:

n_sites [int] Number of sites/orbitals

dot [int] Two-dot (2) or one-dot (1) scheme.

blocking(*i*, *forward*, *bond_dim*)
Perform one blocking iteration.

Args:

i [int] Site index of left dot

forward [bool or None] Direction of current sweep. If True, sweep is performed from left to right. If None, no sweep is performed (local evaluation).

bond_dim [int] Bond dimension of current sweep.

Returns:

result [float] Expectation value.

construct_envs()

get_1pdm(*normsq=1*)
1-particle density matrix.

get_1pdm_spatial(*normsq=1*)
Spatial 1-particle density matrix.

solve(*forward=None*, *bond_dim=-1*)
Calculate expectation value.

Args:

forward [bool or None] Direction of current sweep. If True, sweep is performed from left to right. If None, no sweep is performed (local evaluation).

bond_dim [int] Bond dimension of current sweep.

Returns:

expect [float] Expectation value.

update_one_dot(*i*, *forward*, *bond_dim*)
Update local sites in one-dot scheme.

Args:

i [int] Site index of left dot

forward [bool or None] Direction of current sweep. If True, sweep is performed from left to right. If None, no sweep is performed (local evaluation).

bond_dim [int] Bond dimension of current sweep.

Returns:

expect [float] Expectation value.

update_two_dot(*i*, *forward*, *bond_dim*)
Update local sites in two-dot scheme.

Args:

i [int] Site index of left dot

forward [bool or None] Direction of current sweep. If True, sweep is performed from left to right. If None, no sweep is performed (local evaluation).

bond_dim [int] Bond dimension of current sweep.

Returns:

expect [float] Expectation value.

exception pyblock.algorithm.expectation.**ExpectationError**
Bases: Exception

pyblock.algorithm.expectation.**pprint**(**args*, ***kwargs*)

2.6.4 pyblock.algorithm.compress

Compression algorithm.

class pyblock.algorithm.compress.**Compress**(*mpo*, *mps*, *ket_mps*, *bond_dims*, *noise*, *ket_bond_dim=-1*, *bra_canonical_form=None*, *ket_canonical_form=None*, *contractor=None*)
Bases: object

Compression after apply MPO on MPS.

Attributes:

n_sites [int] Number of sites/orbitals

dot [int] Two-dot (2) or one-dot (1) scheme.

bond_dims [list(int) or int] Bond dimension for each sweep.

blocking(*i*, *forward*, *bond_dim*, *ket_bond_dim*, *noise*, *beta*)
Perform one blocking iteration.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bra bond dimension of current sweep.

ket_bond_dim [int] Ket bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

beta [float] Not used.

Returns:

norm [float] Norm of compressed state.

error [float] Sum of discarded weights.

nexpo [int] Number of operator multiplication steps.

construct_envs()

set_mps(*tags*, *wfn*)

solve(*n_sweeps*, *tol*, *forward=True*, *two_dot_to_one_dot=-1*)
Perform Compression algorithm.

Args:

n_sweeps [int] Maximum number of sweeps.

tol [float] Norm convergence threshold.

forward [bool] Direction of first sweep. If True, sweep is performed from left to right.

two_dot_to_one_dot [int or -1] Indicating when to switch to one-dot scheme. If -1, no switching.

Returns:

nrom [float] Final compressed stae norm.

sweep(*forward*, *bond_dim*, *ket_bond_dim*, *noise*, *beta*)
Perform one sweep iteration.

Args:

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bra bond dimension of current sweep.

ket_bond_dim [int] Ket bond dimension of current sweep.

noise [float] Noise prefactor of current sweep.

beta [float] Not used.

Returns:

norm [float] Norm of compressed state.

update_one_dot(*i*, *forward*, *bond_dim*, *ket_bond_dim*, *noise*, *beta*)
Update local sites in one-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bra bond dimension of current sweep.

ket_bond_dim [int] Ket bond dimension of current sweep.

beta [float] Not used.

Returns:

norm [float] Norm of compressed state.

error [float] Sum of discarded weights.

nexpos [(int, int)] Number of operator multiplication steps.

update_two_dot(*i*, *forward*, *bond_dim*, *ket_bond_dim*, *noise*, *beta*)
Update local sites in two-dot scheme.

Args:

i [int] Site index of left dot

forward [bool] Direction of current sweep. If True, sweep is performed from left to right.

bond_dim [int] Bra bond dimension of current sweep.

ket_bond_dim [int] Ket bond dimension of current sweep.

beta [float] Not used.

Returns:

norm [float] Norm of compressed state.

error [float] Sum of discarded weights.

nexpos [(int, int)] Number of operator multiplication steps.

exception pyblock.algorithm.compress.**CompressionError**
Bases: Exception

pyblock.algorithm.compress.**pprint**(**args*, ***kwargs*)

CHAPTER 3

Block API References

3.1 block module

Python3 wrapper for block 1.5.3 (spin adapted).

class block.**DiagonalMatrix**(**args*, ***kwargs*)
Bases: object

NEWMAT10 diagonal matrix.

cols

ref
A numpy.ndarray reference.

resize(*self: block.DiagonalMatrix*, *nr: int*) → None

rows

class block.**Matrix**(**args*, ***kwargs*)
Bases: object

NEWMAT10 matrix.

cols

ref
A numpy.ndarray reference.

rows

class block.**VectorBool**(**args*, ***kwargs*)
Bases: object

append(*self: block.VectorBool*, *x: bool*) → None
Add an item to the end of the list

count(*self: block.VectorBool*, *x: bool*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorBool, L: block.VectorBool) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorBool, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorBool*, *i: int*, *x: bool*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorBool) -> bool

Remove and return the last item

2. pop(self: block.VectorBool, i: int) -> bool

Remove and return the item at index `i`

remove(*self: block.VectorBool*, *x: bool*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.**VectorDouble**(**args*, ***kwargs*)
Bases: `object`

append(*self: block.VectorDouble*, *x: float*) → None
Add an item to the end of the list

count(*self: block.VectorDouble*, *x: float*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorDouble, L: block.VectorDouble) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorDouble, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorDouble*, *i: int*, *x: float*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorDouble) -> float

Remove and return the last item

2. pop(self: block.VectorDouble, i: int) -> float

Remove and return the item at index `i`

remove(*self: block.VectorDouble*, *x: float*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.**VectorInt**(**args*, ***kwargs*)
Bases: `object`

append(*self: block.VectorInt*, *x: int*) → None
Add an item to the end of the list

count(*self: block.VectorInt*, *x: int*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorInt, L: block.VectorInt) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorInt, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorInt*, *i: int*, *x: int*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorInt) -> int

Remove and return the last item

2. pop(self: block.VectorInt, i: int) -> int

Remove and return the item at index `i`

remove(*self: block.VectorInt*, *x: int*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.**VectorMatrix**(**args*, ***kwargs*)
Bases: `object`

append(*self: block.VectorMatrix*, *x: block.Matrix*) → None
Add an item to the end of the list

count(*self: block.VectorMatrix*, *x: block.Matrix*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorMatrix, L: block.VectorMatrix) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorMatrix, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorMatrix*, *i: int*, *x: block.Matrix*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorMatrix) -> block.Matrix

Remove and return the last item

2. pop(self: block.VectorMatrix, i: int) -> block.Matrix

Remove and return the item at index `i`

remove(*self: block.VectorMatrix*, *x: block.Matrix*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.**VectorVectorInt**(**args*, ***kwargs*)
Bases: object

append(*self: block.VectorVectorInt*, *x: block.VectorInt*) → None
Add an item to the end of the list

count(*self: block.VectorVectorInt*, *x: block.VectorInt*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorVectorInt, L: block.VectorVectorInt) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorVectorInt, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorVectorInt*, *i: int*, *x: block.VectorInt*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorVectorInt) -> block.VectorInt

Remove and return the last item

2. pop(self: block.VectorVectorInt, i: int) -> block.VectorInt

Remove and return the item at index i

remove(*self: block.VectorVectorInt*, *x: block.VectorInt*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.**VectorVectorMatrix**(**args*, ***kwargs*)
Bases: object

append(*self: block.VectorVectorMatrix*, *x: block.VectorMatrix*) → None
Add an item to the end of the list

count(*self: block.VectorVectorMatrix*, *x: block.VectorMatrix*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.VectorVectorMatrix, L: block.VectorVectorMatrix) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.VectorVectorMatrix, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.VectorVectorMatrix*, *i: int*, *x: block.VectorMatrix*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.VectorVectorMatrix) -> block.VectorMatrix

Remove and return the last item

2. pop(self: block.VectorVectorMatrix, i: int) -> block.VectorMatrix

Remove and return the item at index `i`

remove (*self: block.VectorVectorMatrix*, *x: block.VectorMatrix*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

`block.`**load_rotation_matrix** (*arg0: block.VectorInt*, *arg1: block.VectorMatrix*, *arg2: int*) → None
Load rotation matrix.

`block.`**save_rotation_matrix** (*arg0: block.VectorInt*, *arg1: block.VectorMatrix*, *arg2: int*) → None
Save rotation matrix.

3.2 block.io

Contains Input/Output related interfaces.

class `block.io.`**AlgorithmTypes**
Bases: `enum.Enum`

Types of algorithm: one-dot or two-dot or other types.

Members:

OneDot

TwoDot

TwoDotToOneDot

PartialSweep

OneDot = 1

PartialSweep = 4

TwoDot = 2

TwoDotToOneDot = 3

class `block.io.`**CumulTimer** (**args*, ***kwargs*)
Bases: `object`

reset (*self: block.io.CumulTimer*) → None

start (*self: block.io.CumulTimer*) → None

stop (*self: block.io.CumulTimer*) → None

class `block.io.`**Global**
Bases: `object`

Wrapper for global variables.

dmrginp = None

non_abelian_sym = None

point_group = None

class `block.io.`**Input** (**args*, ***kwargs*)
Bases: `object`

algorithm_type
Algorithm type: one-dot or two-dot or other types.

effective_molecule_quantum_vec(*self: block.io.Input*) → block.symmetry.VectorSpinQuantum
Often this simply returns a vector containing one `molecule_quantum`. For non-interacting orbitals or Bogoliubov algorithm, this may be more than that.

hf_occupancy

is_spin_adapted
Indicates whether SU(2) symmetry is utilized. If SU(2) is not used, The Abelian subgroup of SU(2) (Sz symmetry) is used.

molecule_quantum
Symmetry of target state.

n_electrons
Number of electrons

n_max_iters
The maximal number of sweep iterations (outer loop).

n_roots(*self: block.io.Input*, *sweep_iter: int*) → int
Get number of states to solve for given sweep iteration.

output_level

slater_size
Number of spin-orbitals

spin_orbs_symmetry
Spatial symmetry (irrep number) of each spin-orbital.

sweep_tol

timer_guessgen
Timer for generating or loading dot blocks and environment block.

timer_multiplier
Timer for blocking.

timer_operrot
Timer for operator rotation.

twodot_to_onedot_iter
Indicating at which sweep iteration the switching from two-dot to one-dot will happen.

class block.io.**Timer**(**args*, ***kwargs*)
Bases: `object`

elapsed_cputime(*self: block.io.Timer*) → float

elapsed_walltime(*self: block.io.Timer*) → int

start(*self: block.io.Timer*) → None

block.io.**get_current_stack_memory**() → int

block.io.**init_stack_memory**() → None

block.io.**read_input**(*arg0: str*) → None

block.io.**release_stack_memory**() → None

block.io.**set_current_stack_memory**(*arg0: int*) → None

3.3 block.dmrg

DMRG calculations.

class block.dmrg.**MPS**(**args*, ***kwargs*)
Bases: object

get_site_tensors(*self: block.dmrg.MPS*, *arg0: int*) → block.VectorMatrix

get_w(*self: block.dmrg.MPS*) → block.operator.Wavefunction

n_sweep_iters = None

site_blocks = None

write_to_disk(*self: block.dmrg.MPS*, *state_index: int*, *write_state_average: bool = False*) → None

block.dmrg.**MPS_init**(*arg0: bool*) → None
Initialize the single site blocks *MPS.site_blocks*.

class block.dmrg.**SweepParams**(**args*, ***kwargs*)
Bases: object

additional_noise

backward_starting_size
Initial size of system block if in backward direction.

block_iter
Counter for controlling the blocking iteration (inner loop).

current_root

davidson_tol

env_add
The dot block size near environment block.

forward_starting_size
Initial size of system block if in forward direction.

guess_type

largest_dw
Largest discarded weight (or largest error).

lowest_energy

lowest_energy_spin

lowest_error

n_block_iters
The number of blocking iterations (inner loops) needed in one sweep.

n_keep_qstates
(May not be useful.)

n_keep_states
The bond dimension for states in current sweep.

noise

one_dot
Whether it is the one-dot scheme.

save_state (*self: block.dmrg.SweepParams*, *forward: bool*, *size: int*) → None
Save the sweep direction and number of sites in system block into the disk file 'statefile.*.tmp'.

set_sweep_parameters (*self: block.dmrg.SweepParams*) → None

sweep_iter
Counter for controlling the sweep iteration (outer loop).

sys_add
The dot block size near system block.

block.dmrg.**block_and_decimate** (*sweep_params: block.dmrg.SweepParams*, *system: block.block.Block*, *new_system: block.block.Block*, *use_slater: bool*, *dot_with_sys: bool*) → None
Block and decimate to generate the new system block.

block.dmrg.**calldmrg** (*input_file_name: str*) → None
Global driver.

block.dmrg.**dmrg** (*sweep_tol: float*) → None
Perform DMRG calculation.

block.dmrg.**do_one** (*sweep_params: block.dmrg.SweepParams*, *warm_up: bool*, *forward: bool*, *restart: bool*, *restart_size: int*) → float
Perform one sweep procedure.

block.dmrg.**get_dot_with_sys** (*system: block.block.Block*, *one_dot: bool*, *forward: bool*) → bool
Return the *dot_with_sys* variable, determing whether the complementary operators should be defined based on system block indicies.

block.dmrg.**guess_wavefunction** (*solution: block.operator.Wavefunction*, *e: block.DiagonalMatrix*, *big: block.block.Block*, *guess_wave_type: block.block.GuessWaveTypes*, *one_dot: bool*, *state: int*, *transpose_guess_wave: bool*, *additional_noise: float = 0.0*) → None

block.dmrg.**make_system_environment_big_overlap_blocks** (*system_sites: block.VectorInt*, *system_dot: block.block.Block*, *environment_dot: block.block.Block*, *system: block.block.Block*, *new_system: block.block.Block*, *environment: block.block.Block*, *new_environment: block.block.Block*, *big: block.block.Block*, *sweep_params: block.dmrg.SweepParams*, *dot_with_sys: bool*, *use_slater: bool*, *integral_index: int*, *bra_state: int*, *ket_state: int*) → None

3.4 block.block

Block definition and operator operations.

class block.block.**Block**(**args*, ***kwargs*)

Bases: object

add_additional_ops(*self: block.block.Block*) → None

add_all_comp_ops(*self: block.block.Block*) → None

bra_state_info

clear(*self: block.block.Block*) → None

deallocate(*self: block.block.Block*) → None

diagonal_h(*self: block.block.Block*, *arg0: block.DiagonalMatrix*) → None

ket_state_info

left_block

If this is a sum block, return left sub-block for building it.

loop_block

Whether the block is loop block.

move_and_free_memory(*self: block.block.Block*, *arg0: block.block.Block*) → None

If the parameter `system` is allocated before `this` object, but we need to free `system`. Then we have to move the memory of `this` to `system` then clear `system`.

multiply_overlap(*self: block.block.Block*, *c: block.operator.Wavefunction*, *v: block.operator.Wavefunction*, *num_threads: int = 1*) → None

name

A random integer.

ops

Map from operator types to matrix representation of operators.

print_operator_summary(*self: block.block.Block*) → None

Print operator summary when *block.io.Input.output_level* at least = 2.

remove_additional_ops(*self: block.block.Block*) → None

renormalize_from(*self: block.block.Block*, *energies: block.VectorDouble*, *spins: block.VectorDouble*, *error: float*, *rotate_matrix: block.VectorMatrix*, *kept_states: int*, *kept_qstates: int*, *tol: float*, *big: block.block.Block*, *guess_wave_type: block.block.GuessWaveTypes*, *noise: float*, *additional_noise: float*, *one_dot: bool*, *system: block.block.Block*, *system_dot: block.block.Block*, *environment: block.block.Block*, *dot_with_sys: bool*, *warm_up: bool*, *sweep_iter: int*, *current_root: int*, *lower_states: block.operator.VectorWavefunction*) → float

restore(*self: block.block.Block*, *forward: bool*, *sites: block.VectorInt*, *left: int*, *right: int*) → None

Read a *Block* from disk.

Args:

forward [bool] The direction of sweep.

sites [*block.VectorInt*] List of indices of sites contained in the block. left : int Bra state (-1 for normal case).

right [int] Ket state (-1 for normal case).

right_block

If this is a sum block, return right sub-block for building it.

sites
List of indices of sites contained in the block.

store(*self: block.block.Block*, *forward: bool*, *sites: block.VectorInt*, *left: int*, *right: int*) → None
Store a *Block* into disk.

Args:

forward [bool] The direction of sweep.

sites [*block.VectorInt*] List of indices of sites contained in the block. This is kind of redundant and can be obtained from *Block.sites*.

left [int] Bra state (-1 for normal case).

right [int] Ket state (-1 for normal case).

transform_operators(*self: block.block.Block*, *arg0: block.VectorMatrix*) → None

transform_operators_2(*self: block.block.Block*, *left_rotate_matrix: block.VectorMatrix*, *right_rotate_matrix: block.VectorMatrix*, *clear_right_block: bool = True*, *clear_left_block: bool = True*) → None

class block.block.**GuessWaveTypes**
Bases: enum.Enum

Types of guess wavefunction for initialize Davidson algorithm (enumerator).

Members:

Basic

Transform

Transpose

Basic = 1

Transform = 2

Transpose = 3

class block.block.**MapOperators**(**args*, ***kwargs*)
Bases: object

items(*self: block.block.MapOperators*) → iterator

class block.block.**StorageTypes**
Bases: enum.Enum

Types of storage (enumerator).

Members:

LocalStorage

DistributedStorage

DistributedStorage = 2

LocalStorage = 1

class block.block.**VectorBlock**(**args*, ***kwargs*)
Bases: object

append(*self: block.block.VectorBlock*, *x: block.block.Block*) → None
Add an item to the end of the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.block.VectorBlock, L: block.block.VectorBlock) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.block.VectorBlock, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.block.VectorBlock*, *i: int*, *x: block.block.Block*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.block.VectorBlock) -> block.block.Block

Remove and return the last item

2. pop(self: block.block.VectorBlock, i: int) -> block.block.Block

Remove and return the item at index `i`

block.block.**init_big_block**(*left_block: block.block.Block*, *right_block: block.block.Block*, *big_block: block.block.Block*, *bra_quanta: block.symmetry.VectorSpinQuantum = VectorSpinQuantum[]*, *ket_quanta: block.symmetry.VectorSpinQuantum = VectorSpinQuantum[]*) → None
Initialize big (super) block.

block.block.**init_new_environment_block**(*environment: block.block.Block*, *environment_dot: block.block.Block*, *new_environment: block.block.Block*, *system: block.block.Block*, *system_dot: block.block.Block*, *left_state: int*, *right_state: int*, *sys_add: int*, *env_add: int*, *forward: bool*, *direct: bool*, *one_dot: bool*, *use_slater: bool*, *integral_index: int*, *have_norm_ops: bool*, *have_comp_ops: bool*, *dot_with_sys: bool*) → None
Initialize new environment block

block.block.**init_new_system_block**(*system: block.block.Block*, *system_dot: block.block.Block*, *new_system: block.block.Block*, *left_state: int*, *right_state: int*, *sys_add: int*, *direct: bool*, *integral_index: int*, *storage: block.block.StorageTypes*, *have_norm_ops: bool*, *have_comp_ops: bool*) → None
Initialize new system block

block.block.**init_starting_block**(*starting_block: block.block.Block*, *forward: bool*, *left_state: int*, *right_state: int*, *forward_starting_size: int*, *backward_starting_size: int*, *restart_size: int*, *restart: bool*, *warm_up: bool*, *integral_index: int*, *bra_quanta: block.symmetry.VectorSpinQuantum = VectorSpinQuantum[]*, *ket_quanta: block.symmetry.VectorSpinQuantum = VectorSpinQuantum[]*) → None
Initialize starting block

3.5 block.operator

Classes for operator matrices and operations.

class block.operator.**DensityMatrix**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**MapPairInt**(**args*, ***kwargs*)
Bases: object

items(*self: block.operator.MapPairInt*) → iterator

class block.operator.**OpTypes**
Bases: enum.Enum

Types of operators (enumerator).

Members:

Hamiltonian

Cre

CreCre

DesDesComp

CreDes

CreDesComp

CreCreDesComp

Des

DesDes

CreCreComp

DesCre

DesCreComp

CreDesDesComp

Overlap

Cre = 2

CreCre = 3

CreCreComp = 10

CreCreDesComp = 7

CreDes = 5

CreDesComp = 6

CreDesDesComp = 13

Des = 8

DesCre = 11

DesCreComp = 12

DesDes = 9

DesDesComp = 4

Hamiltonian = 1

Overlap = 14

class block.operator.**OperatorArrayBase**
Bases: object

class block.operator.**OperatorArrayCre**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of Cre operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayCre*, *arg0: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayCre*, *arg0: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreCre**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreCre operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCreCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global (*self: block.operator.OperatorArrayCreCre*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local (*self: block.operator.OperatorArrayCreCre*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element (*self: block.operator.OperatorArrayCreCre*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear (*self: block.operator.OperatorArrayCreCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreCreComp** (**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreCreComp operators defined at different sites.

global_element_linear (*self: block.operator.OperatorArrayCreCreComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global (*self: block.operator.OperatorArrayCreCreComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local (*self: block.operator.OperatorArrayCreCreComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element (*self: block.operator.OperatorArrayCreCreComp*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear (*self: block.operator.OperatorArrayCreCreComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreCreDesComp**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreCreDesComp operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCreCreDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayCreCreDesComp*, *arg0: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayCreCreDesComp*, *arg0: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayCreCreDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayCreCreDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreDes**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreDes operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCreDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayCreDes*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayCreDes*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayCreDes*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayCreDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreDesComp**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreDesComp operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCreDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayCreDesComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayCreDesComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayCreDesComp*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayCreDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayCreDesDesComp**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of CreDesDesComp operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayCreDesDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayCreDesDesComp*, *arg0: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayCreDesDesComp*, *arg0: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayCreDesDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayCreDesDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayDes**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of Des operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayDes*, *arg0: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayDes*, *arg0: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayDesCre**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of DesCre operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayDesCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayDesCre*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayDesCre*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayDesCre*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local

storage).

local_element_linear(*self: block.operator.OperatorArrayDesCre*, *arg0: int*) → block.operator.VectorStackSparseMatrix

Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices

A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz

Number of non-zero elements in global storage.

n_local_nz

Number of non-zero elements in local storage.

op_string

Name of the type of operators contained in this array.

class block.operator.**OperatorArrayDesCreComp**(**args*, ***kwargs*)

Bases: *block.operator.OperatorArrayBase*

An array of DesCreComp operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayDesCreComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix

Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices

A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayDesCreComp*, *arg0: int*, *arg1: int*) → bool

Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayDesCreComp*, *arg0: int*, *arg1: int*) → bool

Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayDesCreComp*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix

Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayDesCreComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix

Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices

A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz

Number of non-zero elements in global storage.

n_local_nz

Number of non-zero elements in local storage.

op_string

Name of the type of operators contained in this array.

class block.operator.**OperatorArrayDesDes**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of DesDes operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayDesDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayDesDes*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayDesDes*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayDesDes*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayDesDes*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayDesDesComp**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of DesDesComp operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayDesDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayDesDesComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero (in local or global storage). The parameters are site indices.

has_local(*self: block.operator.OperatorArrayDesDesComp*, *arg0: int*, *arg1: int*) → bool
Query whether the element is non-zero in local storage. The parameters are site indices.

local_element(*self: block.operator.OperatorArrayDesDesComp*, *arg0: int*, *arg1: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given site indices (in local storage).

local_element_linear(*self: block.operator.OperatorArrayDesDesComp*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayHamiltonian**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of Hamiltonian operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayHamiltonian*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayHamiltonian*) → bool
Query whether the element is non-zero (in local or global storage).

has_local(*self: block.operator.OperatorArrayHamiltonian*) → bool
Query whether the element is non-zero in local storage.

local_element(*self: block.operator.OperatorArrayHamiltonian*) → block.operator.VectorStackSparseMatrix
Get the array of operators (for different spin quantum numbers, in local storage).

local_element_linear(*self: block.operator.OperatorArrayHamiltonian*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorArrayOverlap**(**args*, ***kwargs*)
Bases: *block.operator.OperatorArrayBase*

An array of Overlap operators defined at different sites.

global_element_linear(*self: block.operator.OperatorArrayOverlap*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in global storage).

global_indices
A 1d array contains the site indices of non-zero elements (in local or global storage). It gives a map from flattened single index to multiple site indices. Then this array itself is flattened.

has_global(*self: block.operator.OperatorArrayOverlap*) → bool
Query whether the element is non-zero (in local or global storage).

has_local(*self: block.operator.OperatorArrayOverlap*) → bool
Query whether the element is non-zero in local storage.

local_element(*self: block.operator.OperatorArrayOverlap*) → block.operator.VectorStackSparseMatrix
Get the array of operators (for different spin quantum numbers, in local storage).

local_element_linear(*self: block.operator.OperatorArrayOverlap*, *arg0: int*) → block.operator.VectorStackSparseMatrix
Get an array of operators (for different spin quantum numbers) defined for the given (flattened) linear index (in local storage).

local_indices
A 2d array contains the site indices of non-zero elements in local storage. It gives a map from flattened single index to multiple site indices (which is represented as an array).

n_global_nz
Number of non-zero elements in global storage.

n_local_nz
Number of non-zero elements in local storage.

op_string
Name of the type of operators contained in this array.

class block.operator.**OperatorCre**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreCre**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreCreComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreCreDesComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreDes**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreDesComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorCreDesDesComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorDes**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorDesCre**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorDesCreComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorDesDes**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorDesDesComp**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorHamiltonian**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**OperatorOverlap**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**StackMatrix**(**args*, ***kwargs*)
Bases: object

Very simple Matrix class that provides a Matrix type interface for a double array. It does not own its own data.

Note that the C++ class used indices counting from 1. Here we count from 0. Row-major (C) storage.

cols

ref
A numpy.ndarray reference.

rows

class block.operator.**StackSparseMatrix**(**args*, ***kwargs*)
Bases: object

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

allocate(*self: block.operator.StackSparseMatrix*, *arg0: block.symmetry.VectorStateInfo*) → None

allocate_memory(*self: block.operator.StackSparseMatrix*, *arg0: int*) → None

allowed(*self: block.operator.StackSparseMatrix*, *arg0: int*, *arg1: int*) → bool

clear(*self: block.operator.StackSparseMatrix*) → None

cols

conjugacy

deallocate(*self: block.operator.StackSparseMatrix*) → None

deep_clear_copy(*self: block.operator.StackSparseMatrix*, *arg0: block.operator.StackSparseMatrix*) → None

deep_copy(*self: block.operator.StackSparseMatrix*, *arg0: block.operator.StackSparseMatrix*) → None

delta_quantum
Allowed change of quantum numbers between states.

fermion

get_scaling(*self: block.operator.StackSparseMatrix*, *leftq: block.symmetry.SpinQuantum*, *rightq: block.symmetry.SpinQuantum*) → float

initialized

map_to_non_zero_blocks
A map from pair of bra and ket indices, to the index in *StackSparseMatrix.non_zero_blocks*.

non_zero_blocks
A list of non zero blocks. Each element in the list is a pair of a pair of bra and ket indices, and *StackMatrix*.

operator_element(*self: block.operator.StackSparseMatrix*, *arg0: int*, *arg1: int*) → block.operator.StackMatrix

ref
A numpy.ndarray reference.

rows

shallow_copy(*self: block.operator.StackSparseMatrix*, *arg0: block.operator.StackSparseMatrix*) → None

symm_scale

total_memory

transpose(*self: block.operator.StackSparseMatrix*) → SpinAdapted::StackTransposeview

class block.operator.**StackTransposeView**(**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

class block.operator.**VectorCre**(**args*, ***kwargs*)
Bases: object

append(*self: block.operator.VectorCre*, *x: block.operator.OperatorCre*) → None
Add an item to the end of the list

count(*self: block.operator.VectorCre*, *x: block.operator.OperatorCre*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorCre, L: block.operator.VectorCre) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCre, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorCre*, *i: int*, *x: block.operator.OperatorCre*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorCre) -> block.operator.OperatorCre

Remove and return the last item

2. pop(self: block.operator.VectorCre, i: int) -> block.operator.OperatorCre

Remove and return the item at index i

remove(*self: block.operator.VectorCre*, *x: block.operator.OperatorCre*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreCre**(**args*, ***kwargs*)
Bases: object

append(*self: block.operator.VectorCreCre*, *x: block.operator.OperatorCreCre*) → None
Add an item to the end of the list

count(*self: block.operator.VectorCreCre*, *x: block.operator.OperatorCreCre*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorCreCre, L: block.operator.VectorCreCre) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreCre, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorCreCre*, *i: int*, *x: block.operator.OperatorCreCre*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorCreCre) -> block.operator.OperatorCreCre

Remove and return the last item

2. pop(self: block.operator.VectorCreCre, i: int) -> block.operator.OperatorCreCre

Remove and return the item at index `i`

remove (*self: block.operator.VectorCreCre*, *x: block.operator.OperatorCreCre*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreCreComp** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorCreCreComp*, *x: block.operator.OperatorCreCreComp*) → None
Add an item to the end of the list

count (*self: block.operator.VectorCreCreComp*, *x: block.operator.OperatorCreCreComp*) → int
Return the number of times `x` appears in the list

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorCreCreComp, L: block.operator.VectorCreCreComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreCreComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorCreCreComp*, *i: int*, *x: block.operator.OperatorCreCreComp*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorCreCreComp) -> block.operator.OperatorCreCreComp

Remove and return the last item

2. pop(self: block.operator.VectorCreCreComp, i: int) -> block.operator.OperatorCreCreComp

Remove and return the item at index `i`

remove (*self: block.operator.VectorCreCreComp*, *x: block.operator.OperatorCreCreComp*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreCreDesComp** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorCreCreDesComp*, *x: block.operator.OperatorCreCreDesComp*) → None
Add an item to the end of the list

count (*self: block.operator.VectorCreCreDesComp*, *x: block.operator.OperatorCreCreDesComp*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorCreCreDesComp, L: block.operator.VectorCreCreDesComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreCreDesComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorCreCreDesComp*, *i: int*, *x: block.operator.OperatorCreCreDesComp*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorCreCreDesComp) -> block.operator.OperatorCreCreDesComp

Remove and return the last item

2. pop(self: block.operator.VectorCreCreDesComp, i: int) -> block.operator.OperatorCreCreDesComp

Remove and return the item at index `i`

remove(*self: block.operator.VectorCreCreDesComp*, *x: block.operator.OperatorCreCreDesComp*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreDes**(**args*, ***kwargs*)
Bases: `object`

append(*self: block.operator.VectorCreDes*, *x: block.operator.OperatorCreDes*) → None
Add an item to the end of the list

count(*self: block.operator.VectorCreDes*, *x: block.operator.OperatorCreDes*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorCreDes, L: block.operator.VectorCreDes) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreDes, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorCreDes*, *i: int*, *x: block.operator.OperatorCreDes*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorCreDes) -> block.operator.OperatorCreDes

Remove and return the last item

2. pop(self: block.operator.VectorCreDes, i: int) -> block.operator.OperatorCreDes

Remove and return the item at index `i`

remove(*self: block.operator.VectorCreDes*, *x: block.operator.OperatorCreDes*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreDesComp**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorCreDesComp*, *x: block.operator.OperatorCreDesComp*) → None

Add an item to the end of the list

count(*self: block.operator.VectorCreDesComp*, *x: block.operator.OperatorCreDesComp*) → int

Return the number of times x appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorCreDesComp, L: block.operator.VectorCreDesComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreDesComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorCreDesComp*, *i: int*, *x: block.operator.OperatorCreDesComp*) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: block.operator.VectorCreDesComp) -> block.operator.OperatorCreDesComp

Remove and return the last item

2. pop(self: block.operator.VectorCreDesComp, i: int) -> block.operator.OperatorCreDesComp

Remove and return the item at index i

remove(*self: block.operator.VectorCreDesComp*, *x: block.operator.OperatorCreDesComp*) → None

Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorCreDesDesComp**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorCreDesDesComp*, *x: block.operator.OperatorCreDesDesComp*) → None

Add an item to the end of the list

count(*self: block.operator.VectorCreDesDesComp*, *x: block.operator.OperatorCreDesDesComp*) → int

Return the number of times x appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorCreDesDesComp, L: block.operator.VectorCreDesDesComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorCreDesDesComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorCreDesDesComp*, *i: int*, *x: block.operator.OperatorCreDesDesComp*) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: block.operator.VectorCreDesDesComp) -> block.operator.OperatorCreDesDesComp

Remove and return the last item

2. pop(self: block.operator.VectorCreDesDesComp, i: int) -> block.operator.OperatorCreDesDesComp

Remove and return the item at index `i`

remove(*self: block.operator.VectorCreDesDesComp*, *x: block.operator.OperatorCreDesDesComp*) → None

Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorDes**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorDes*, *x: block.operator.OperatorDes*) → None

Add an item to the end of the list

count(*self: block.operator.VectorDes*, *x: block.operator.OperatorDes*) → int

Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorDes, L: block.operator.VectorDes) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorDes, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorDes*, *i: int*, *x: block.operator.OperatorDes*) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: block.operator.VectorDes) -> block.operator.OperatorDes

Remove and return the last item

2. pop(self: block.operator.VectorDes, i: int) -> block.operator.OperatorDes

Remove and return the item at index `i`

remove(*self: block.operator.VectorDes*, *x: block.operator.OperatorDes*) → None

Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorDesCre**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorDesCre*, *x: block.operator.OperatorDesCre*) → None

Add an item to the end of the list

count(*self: block.operator.VectorDesCre*, *x: block.operator.OperatorDesCre*) → int

Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorDesCre, L: block.operator.VectorDesCre) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorDesCre, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorDesCre*, *i: int*, *x: block.operator.OperatorDesCre*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorDesCre) -> block.operator.OperatorDesCre

Remove and return the last item

2. pop(self: block.operator.VectorDesCre, i: int) -> block.operator.OperatorDesCre

Remove and return the item at index `i`

remove (*self: block.operator.VectorDesCre*, *x: block.operator.OperatorDesCre*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorDesCreComp** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorDesCreComp*, *x: block.operator.OperatorDesCreComp*) → None
Add an item to the end of the list

count (*self: block.operator.VectorDesCreComp*, *x: block.operator.OperatorDesCreComp*) → int
Return the number of times `x` appears in the list

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorDesCreComp, L: block.operator.VectorDesCreComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorDesCreComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorDesCreComp*, *i: int*, *x: block.operator.OperatorDesCreComp*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorDesCreComp) -> block.operator.OperatorDesCreComp

Remove and return the last item

2. pop(self: block.operator.VectorDesCreComp, i: int) -> block.operator.OperatorDesCreComp

Remove and return the item at index `i`

remove (*self: block.operator.VectorDesCreComp*, *x: block.operator.OperatorDesCreComp*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorDesDes** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorDesDes*, *x: block.operator.OperatorDesDes*) → None
Add an item to the end of the list

count (*self: block.operator.VectorDesDes*, *x: block.operator.OperatorDesDes*) → int
Return the number of times `x` appears in the list

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorDesDes, L: block.operator.VectorDesDes) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorDesDes, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorDesDes*, *i: int*, *x: block.operator.OperatorDesDes*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorDesDes) -> block.operator.OperatorDesDes

Remove and return the last item

2. pop(self: block.operator.VectorDesDes, i: int) -> block.operator.OperatorDesDes

Remove and return the item at index `i`

remove (*self: block.operator.VectorDesDes*, *x: block.operator.OperatorDesDes*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorDesDesComp** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorDesDesComp*, *x: block.operator.OperatorDesDesComp*) → None
Add an item to the end of the list

count (*self: block.operator.VectorDesDesComp*, *x: block.operator.OperatorDesDesComp*) → int
Return the number of times `x` appears in the list

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorDesDesComp, L: block.operator.VectorDesDesComp) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorDesDesComp, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorDesDesComp*, *i: int*, *x: block.operator.OperatorDesDesComp*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorDesDesComp) -> block.operator.OperatorDesDesComp

Remove and return the last item

2. pop(self: block.operator.VectorDesDesComp, i: int) -> block.operator.OperatorDesDesComp

Remove and return the item at index `i`

remove (*self: block.operator.VectorDesDesComp*, *x: block.operator.OperatorDesDesComp*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorHamiltonian** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.operator.VectorHamiltonian*, *x: block.operator.OperatorHamiltonian*) → None
Add an item to the end of the list

count(*self: block.operator.VectorHamiltonian*, *x: block.operator.OperatorHamiltonian*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorHamiltonian, L: block.operator.VectorHamiltonian) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorHamiltonian, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorHamiltonian*, *i: int*, *x: block.operator.OperatorHamiltonian*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorHamiltonian) -> block.operator.OperatorHamiltonian

Remove and return the last item

2. pop(self: block.operator.VectorHamiltonian, i: int) -> block.operator.OperatorHamiltonian

Remove and return the item at index i

remove(*self: block.operator.VectorHamiltonian*, *x: block.operator.OperatorHamiltonian*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorNonZeroStackMatrix**(**args*, ***kwargs*)
Bases: object

append(*self: block.operator.VectorNonZeroStackMatrix, x: Tuple[Tuple[int, int], block.operator.StackMatrix]*) → None
Add an item to the end of the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorNonZeroStackMatrix, L: block.operator.VectorNonZeroStackMatrix) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorNonZeroStackMatrix, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorNonZeroStackMatrix, i: int, x: Tuple[Tuple[int, int], block.operator.StackMatrix]*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorNonZeroStackMatrix) -> Tuple[Tuple[int, int], block.operator.StackMatrix]

Remove and return the last item

2. pop(self: block.operator.VectorNonZeroStackMatrix, i: int) -> Tuple[Tuple[int, int], block.operator.StackMatrix]

Remove and return the item at index i

class block.operator.**VectorOperatorArrayBase**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorOperatorArrayBase*, *x: block.operator.OperatorArrayBase*) → None

Add an item to the end of the list

count(*self: block.operator.VectorOperatorArrayBase*, *x: block.operator.OperatorArrayBase*) → int

Return the number of times x appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorOperatorArrayBase, L: block.operator.VectorOperatorArrayBase) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorOperatorArrayBase, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorOperatorArrayBase*, *i: int*, *x: block.operator.OperatorArrayBase*) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: block.operator.VectorOperatorArrayBase) -> block.operator.OperatorArrayBase

Remove and return the last item

2. pop(self: block.operator.VectorOperatorArrayBase, i: int) -> block.operator.OperatorArrayBase

Remove and return the item at index i

remove(*self: block.operator.VectorOperatorArrayBase*, *x: block.operator.OperatorArrayBase*) → None

Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorOverlap**(**args*, ***kwargs*)

Bases: object

append(*self: block.operator.VectorOverlap*, *x: block.operator.OperatorOverlap*) → None

Add an item to the end of the list

count(*self: block.operator.VectorOverlap*, *x: block.operator.OperatorOverlap*) → int

Return the number of times x appears in the list

extend(**args*, ***kwargs*)

Overloaded function.

1. extend(self: block.operator.VectorOverlap, L: block.operator.VectorOverlap) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorOverlap, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorOverlap*, *i: int*, *x: block.operator.OperatorOverlap*) → None

Insert an item at a given position.

pop(**args*, ***kwargs*)

Overloaded function.

1. pop(self: block.operator.VectorOverlap) -> block.operator.OperatorOverlap

Remove and return the last item

2. pop(self: block.operator.VectorOverlap, i: int) -> block.operator.OperatorOverlap

Remove and return the item at index `i`

remove(*self: block.operator.VectorOverlap*, *x: block.operator.OperatorOverlap*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorStackSparseMatrix**(**args*, ***kwargs*)
Bases: object

append(*self: block.operator.VectorStackSparseMatrix*, *x: block.operator.StackSparseMatrix*) → None
Add an item to the end of the list

count(*self: block.operator.VectorStackSparseMatrix*, *x: block.operator.StackSparseMatrix*) → int
Return the number of times `x` appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorStackSparseMatrix, L: block.operator.VectorStackSparseMatrix) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorStackSparseMatrix, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: block.operator.VectorStackSparseMatrix*, *i: int*, *x: block.operator.StackSparseMatrix*) → None
Insert an item at a given position.

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorStackSparseMatrix) -> block.operator.StackSparseMatrix

Remove and return the last item

2. pop(self: block.operator.VectorStackSparseMatrix, i: int) -> block.operator.StackSparseMatrix

Remove and return the item at index `i`

remove(*self: block.operator.VectorStackSparseMatrix*, *x: block.operator.StackSparseMatrix*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.operator.**VectorWavefunction**(**args*, ***kwargs*)
Bases: object

append(*self: block.operator.VectorWavefunction*, *x: block.operator.Wavefunction*) → None
Add an item to the end of the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.operator.VectorWavefunction, L: block.operator.VectorWavefunction) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.operator.VectorWavefunction, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.operator.VectorWavefunction*, *i: int*, *x: block.operator.Wavefunction*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.operator.VectorWavefunction) -> block.operator.Wavefunction

Remove and return the last item

2. pop(self: block.operator.VectorWavefunction, i: int) -> block.operator.Wavefunction

Remove and return the item at index `i`

class block.operator.**Wavefunction** (**args*, ***kwargs*)
Bases: *block.operator.StackSparseMatrix*

Block-sparse matrix. Non-zero blocks are identified by symmetry (quantum numbers) requirements and stored as *StackMatrix* objects

copy_data (*self: block.operator.Wavefunction*, *arg0: block.operator.Wavefunction*) → None

initialize (*self: block.operator.Wavefunction*, *arg0: block.symmetry.VectorSpinQuantum*, *arg1: block.symmetry.StateInfo*, *arg2: block.symmetry.StateInfo*, *arg3: bool*) → None

initialize_from (*self: block.operator.Wavefunction*, *arg0: block.operator.Wavefunction*) → None

onedot

save_wavefunction_info (*self: block.operator.Wavefunction*, *arg0: block.symmetry.StateInfo*, *arg1: block.VectorInt*, *arg2: int*) → None

block.operator.**multiply_with_own_transpose** (*arg0: block.operator.StackSparseMatrix*, *arg1: block.operator.StackSparseMatrix*, *arg2: float*) → None

3.6 block.symmetry

Classes for handling symmetries and quantum numbers.

class block.symmetry.**IrrepSpace** (**args*, ***kwargs*)
Bases: object

A wrapper class for molecular point group symmetry irrep.

irrep

class block.symmetry.**SpinQuantum** (**args*, ***kwargs*)
Bases: object

A collection of quantum numbers associated with a specific state (irreducible representation). One such collection defines a specific sector in the state space.

n
Particle number.

s
Irreducible representation for spin symmetry (S or S_z).

symm
Irreducible representation for molecular point group symmetry.

class block.symmetry.**SpinSpace**(**args*, ***kwargs*)
Bases: object

A wrapper class for the spin irrep.

In S_z symmetry, the irrep is $2S_z$. In SU(2) symmetry, the irrep is $2S$. The behaviour is toggled checking block.io.Global.dmrginp.spin_adapted.

irrep

class block.symmetry.**StateInfo**(**args*, ***kwargs*)
Bases: object

A collection of symmetry sectors. Each sector can contain several internal states (which can no longer be differentiated by symmetries), the number of which is also stored.

collect_quanta(*self: block.symmetry.StateInfo*) → None

copy(*self: block.symmetry.StateInfo*) → block.symmetry.StateInfo

left_state_info

left_unmap_quanta
Index in left StateInfo, for each combined state.

n_states
Number of states per (combined) quantum number.

n_total_states

old_to_new_state
old_to_new_state[i] = [k1, k2, k3, . . .] where i is the index in the collected StateInfo and k's are indices in the uncollected StateInfo.

quanta
Quantum numbers for a set of sites.

right_state_info

right_unmap_quanta
Index in right StateInfo, for each combined state.

set_left_state_info(*self: block.symmetry.StateInfo*, *arg0: block.symmetry.StateInfo*) → None

set_right_state_info(*self: block.symmetry.StateInfo*, *arg0: block.symmetry.StateInfo*) → None

set_uncollected_state_info(*self: block.symmetry.StateInfo*, *arg0: block.symmetry.StateInfo*) → None

uncollected_state_info

class block.symmetry.**VectorSpinQuantum**(**args*, ***kwargs*)
Bases: object

append(*self: block.symmetry.VectorSpinQuantum*, *x: block.symmetry.SpinQuantum*) → None
Add an item to the end of the list

count(*self: block.symmetry.VectorSpinQuantum*, *x: block.symmetry.SpinQuantum*) → int
Return the number of times x appears in the list

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.symmetry.VectorSpinQuantum, L: block.symmetry.VectorSpinQuantum) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.symmetry.VectorSpinQuantum, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.symmetry.VectorSpinQuantum*, *i: int*, *x: block.symmetry.SpinQuantum*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.symmetry.VectorSpinQuantum) -> block.symmetry.SpinQuantum

Remove and return the last item

2. pop(self: block.symmetry.VectorSpinQuantum, i: int) -> block.symmetry.SpinQuantum

Remove and return the item at index `i`

remove (*self: block.symmetry.VectorSpinQuantum*, *x: block.symmetry.SpinQuantum*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

class block.symmetry.**VectorStateInfo** (**args*, ***kwargs*)
Bases: `object`

append (*self: block.symmetry.VectorStateInfo*, *x: block.symmetry.StateInfo*) → None
Add an item to the end of the list

count (*self: block.symmetry.VectorStateInfo*, *x: block.symmetry.StateInfo*) → int
Return the number of times `x` appears in the list

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: block.symmetry.VectorStateInfo, L: block.symmetry.VectorStateInfo) -> None

Extend the list by appending all the items in the given list

2. extend(self: block.symmetry.VectorStateInfo, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: block.symmetry.VectorStateInfo*, *i: int*, *x: block.symmetry.StateInfo*) → None
Insert an item at a given position.

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: block.symmetry.VectorStateInfo) -> block.symmetry.StateInfo

Remove and return the last item

2. pop(self: block.symmetry.VectorStateInfo, i: int) -> block.symmetry.StateInfo

Remove and return the item at index `i`

remove (*self: block.symmetry.VectorStateInfo*, *x: block.symmetry.StateInfo*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

block.symmetry.**get_commute_parity** (*a: block.symmetry.SpinQuantum*, *b: block.symmetry.SpinQuantum*, *c: block.symmetry.SpinQuantum*) → float

block.symmetry.**state_tensor_product** (*arg0: block.symmetry.StateInfo*, *arg1: block.symmetry.StateInfo*) → block.symmetry.StateInfo

block.symmetry.**state_tensor_product_target**(*arg0: block.symmetry.StateInfo*, *arg1: block.symmetry.StateInfo*) → block.symmetry.StateInfo

block.symmetry.**wigner_9j**(*arg0: int*, *arg1: int*, *arg2: int*, *arg3: int*, *arg4: int*, *arg5: int*, *arg6: int*, *arg7: int*, *arg8: int*) → float

3.7 block.rev

Revised Block functions.

block.rev.**product**(*a: block.operator.StackSparseMatrix*, *b: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*, *state_info: block.symmetry.StateInfo*, *scale: float = 1.0*) → None

block.rev.**tensor_dot_product**(*a: block.operator.StackSparseMatrix*, *b: block.operator.StackSparseMatrix*) → float

block.rev.**tensor_precondition**(*a: block.operator.StackSparseMatrix*, *e: float*, *diag: block.DiagonalMatrix*) → None

block.rev.**tensor_product**(*a: block.operator.StackSparseMatrix*, *b: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*, *state_info: block.symmetry.VectorStateInfo*, *scale: float = 1.0*) → None

block.rev.**tensor_product_diagonal**(*a: block.operator.StackSparseMatrix*, *b: block.operator.StackSparseMatrix*, *c: block.DiagonalMatrix*, *state_info: block.symmetry.VectorStateInfo*, *scale: float = 1.0*) → None

block.rev.**tensor_product_multiply**(*a: block.operator.StackSparseMatrix*, *b: block.operator.StackSparseMatrix*, *c: block.operator.Wavefunction*, *v: block.operator.Wavefunction*, *state_info: block.symmetry.VectorStateInfo*, *op_q: block.symmetry.SpinQuantum*, *scale: float*) → None

block.rev.**tensor_rotate**(*a: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*, *state_info: block.symmetry.VectorStateInfo*, *rotate_matrices: block.VectorVectorMatrix*, *scale: float = 1.0*) → None

block.rev.**tensor_scale**(*scale: float*, *a: block.operator.StackSparseMatrix*) → None

block.rev.**tensor_scale_add**(*scale: float*, *a: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*, *state_info: block.symmetry.VectorStateInfo*) → None

block.rev.**tensor_scale_add_no_trans**(*scale: float*, *a: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*) → None

block.rev.**tensor_trace**(*a: block.operator.StackSparseMatrix*, *c: block.operator.StackSparseMatrix*, *state_info: block.symmetry.VectorStateInfo*, *trace_right: bool*, *scale: float = 1.0*) → None

block.rev.**tensor_trace_diagonal**(*a: block.operator.StackSparseMatrix*, *c: block.DiagonalMatrix*, *state_info: block.symmetry.VectorStateInfo*, *trace_right: bool*, *scale: float = 1.0*) → None

block.rev.**tensor_trace_multiply**(*a: block.operator.StackSparseMatrix*, *c: block.operator.Wavefunction*, *v: block.operator.Wavefunction*, *state_info: block.symmetry.StateInfo*, *trace_right: bool*, *scale: float*) → None

3.8 block.data_page

Revised data page functions.

block.data_page.**activate_data_page**(*ip: int*) → None

block.data_page.**get_data_page_pointer**(*ip: int*) → int

block.data_page.**init_data_pages**(*n_pages: int*) → None

block.data_page.**load_data_page**(*ip: int*, *filename: str*) → None

block.data_page.**release_data_pages**() → None

block.data_page.**save_data_page**(*ip: int*, *filename: str*) → None

block.data_page.**set_data_page_pointer**(*ip: int*, *offset: int*) → None

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

p

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z