openfermion Documentation

Release 0.11.1.dev

openfermion

May 18, 2020

Contents

Contents

CHAPTER 1

Code Documentation

1.1 openfermion.hamiltonians

class openfermion.hamiltonians.**FermiHubbardModel**(*lattice*, *tunneling_parameters=None*, *interaction_parameters=None*, *potential_parameters=None*, *magnetic_field=0.0*, *particle_hole_symmetry=False*)

A general, parameterized Fermi-Hubbard model.

The general (AKA 'multiband') Fermi-Hubbard model has *k* degrees of freedom per site in a lattice. For a lattice with *n* sites, there are *N = k * n* spatial orbitals. Additionally, in what we call the "spinful" model each spatial orbital is associated with "up" and "down" spin orbitals, for a total of *2N* spin orbitals; in the spinless model, there is only one spin-orbital per site for a total of *N*.

For a lattice with only one type of site and edges from each site only to itself and its neighbors, the Hamiltonian for the spinful model has the form

$$\begin{aligned}
H = &- \sum_{a<b} t^{(\mathrm{onsite})}_{a,b} \sum_i \sum_\sigma (a^\dagger_{i,a,\sigma} a_{i,b,\sigma} + a^\dagger_{i,b,\sigma} a_{i,a,\sigma}) \tag{1.1} \\
&- \sum_{a} t^{(\mathrm{nghbr})}_{a,a} \sum_{\{i,j\}} \sum_\sigma (a^\dagger_{i,a,\sigma} a_{j,a,\sigma} + a^\dagger_{j,a,\sigma} a_{i,a,\sigma}) - \sum_{a<b} t^{(\mathrm{nghbr})}_{a,b} \sum_{(i,j)} \sum_\sigma (a^\dagger_{i,a,\sigma} a_{j,b,\sigma} + a^\dagger_{j,b,\sigma} a_{i,a,\sigma}) \tag{1.2} \\
&+ \sum_{a<b} U^{(\mathrm{onsite},+)}_{a,b} \sum_i \sum_\sigma n_{i,a,\sigma} n_{i,b,\sigma} \tag{1.3} \\
&+ \sum_{a} U^{(\mathrm{nghbr},+)}_{a,a} \sum_{\{i,j\}} \sum_\sigma n_{i,a,\sigma} n_{j,a,\sigma} + \sum_{a<b} U^{(\mathrm{nghbr},+)}_{a,b} \sum_{(i,j)} \sum_\sigma n_{i,a,\sigma} n_{j,b,\sigma} \tag{1.4} \\
&+ \sum_{a\leq b} U^{(\mathrm{onsite},-)}_{a,b} \sum_i \sum_\sigma n_{i,a,\sigma} n_{i,b,-\sigma} \tag{1.5} \\
&+ \sum_{a} U^{(\mathrm{nghbr},-)}_{a,a} \sum_{\{i,j\}} \sum_\sigma n_{i,a,\sigma} n_{j,a,-\sigma} + \sum_{a<b} U^{(\mathrm{nghbr},-)}_{a,b} \sum_{(i,j)} \sum_\sigma n_{i,a,\sigma} n_{j,b,-\sigma} \tag{1.6} \\
&- \sum_{a} \mu_a \sum_i \sum_\sigma n_{i,a,\sigma} \tag{1.7} \\
&- h \sum_i \sum_a (n_{i,a,\uparrow} - n_{i,a,\downarrow}) \tag{1.8}
\end{aligned}$$

where

- The indices (i, j) and $\{i, j\}$ run over ordered and unordered pairs, respectively of sites i and j of neighboring sites in the lattice,
- a and b index degrees of freedom on each site,
- $\sigma \in \{\uparrow, \downarrow\}$ is the spin,
- $t^{(\mathrm{onsite})}_{a,b}$ is the tunneling amplitude between spin orbitals on the same site,
- $t^{(\mathrm{nghbr})}_{a,b}$ is the tunneling amplitude between spin orbitals on neighboring sites,
- $U^{(\mathrm{onsite},\pm)}_{a,b}$ is the Coulomb potential between spin orbitals on the same site with the same (+) or different (-) spins,
- $U^{(\mathrm{nghbr},\pm)}_{a,b}$ is the Coulomb potential betwen spin orbitals on neighborings sites with the same (+) or different (-) spins,
- μ_a is the chemical potential, and
- h is the magnetic field.

One can also construct the Hamiltonian for the spinless model, which has the form

$$\begin{aligned}
H = &- \sum_{a<b} t^{(\mathrm{onsite})}_{a,b} \sum_i (a^\dagger_{i,a} a_{i,b} + a^\dagger_{i,b} a_{i,a}) \tag{1.9} \\
&- \sum_{a} t^{(\mathrm{nghbr})}_{a,a} \sum_{\{i,j\}} (a^\dagger_{i,a} a_{j,a} + a^\dagger_{j,a} a_{i,a}) - \sum_{a<b} t^{(\mathrm{nghbr})}_{a,b} \sum_{(i,j)} (a^\dagger_{i,a} a_{j,b} + a^\dagger_{j,b} a_{i,a}) \tag{1.10} \\
&+ \sum_{a<b} U^{(\mathrm{onsite})}_{a,b} \sum_i n_{i,a} n_{i,b} \tag{1.11} \\
&+ \sum_{a} U^{(\mathrm{nghbr})}_{a,a} \sum_{\{i,j\}} n_{i,a} n_{j,a} + \sum_{a<b} U^{(\mathrm{nghbr})}_{a,b} \sum_{(i,j)} n_{i,a} n_{j,b} \tag{1.12} \\
&- \sum_{a} \mu_a \sum_i n_{i,a} \tag{1.13}
\end{aligned}$$

__init__(*lattice*, *tunneling_parameters=None*, *interaction_parameters=None*, *potential_parameters=None*, *magnetic_field=0.0*, *particle_hole_symmetry=False*)

A Hubbard model defined on a lattice.

Parameters

- **lattice** (*HubbardLattice*) – The lattice on which the model is defined.
- **(Iterable[Tuple[Hashable, Tuple[int, int],** (*interaction_parameters*) – float]], optional): The tunneling parameters.
- **(Iterable[Tuple[Hashable, Tuple[int, int],** – float, int?]], optional): The interaction parameters.
- **potential_parameters** (*Iterable[Tuple[int, float]], optional*) – The potential parameters.
- **magnetic_field** (*float, optional*) – The magnetic field. Default is 0.
- **particle_hole_symmetry** – If true, each number operator n is replaced with $n - 1/2$.

Each group of parameters is specified as an iterable of tuples.

Each tunneling parameter is a tuple `(edge_type, dofs, coefficient)`.

In the spinful, model, the tunneling parameter corresponds to the terms

$$t \sum_{(i,j)\in E^{(\mathrm{edge type})}} \sum_{\sigma} \left(a_{i,a,\sigma}^{\dagger} a_{j,b,\sigma} + a_{j,b,\sigma}^{\dagger} a_{i,a,\sigma} \right)$$

and in the spinless model to

$$-t \sum_{(i,j)\in E^{(\mathrm{edge type})}} \left(a_{i,a}^{\dagger} a_{j,b} + a_{j,b}^{\dagger} a_{i,a} \right),$$

where

- (a, b) is the pair of degrees

of freedom given by `dofs`; - $E^{(\mathrm{edge type})}$ is the set of ordered pairs of

site indices returned by `lattice.site_pairs_iter(edge_type, a != b)`; and

- t is the `coefficient`.

Each interaction parameter is a tuple `(edge_type, dofs, coefficient, spin_pairs)`. The final `spin_pairs` element is optional, and will default to `SpinPairs.ALL`. In any case, it is ignored for spinless lattices.

For example, in the spinful model if *dofs* indicates distinct degrees of freedom then the parameter corresponds to the terms

U sum_{(i, j) in E^{(mathrm{edge type})}} sum_{(sigma, sigma')} n_{i, a, sigma} n_{j, b, sigma'}

where

- (a, b) is the pair of degrees of

freedom given by `dofs`; - $E^{(\mathrm{edge type})}$ is the set of ordered pairs of

site indices returned by `lattice.site_pairs_iter(edge_type)`;

- U is the `coefficient`; and
- (σ, σ') **runs over**
  - all four possible pairs of spins

  if *spin_pairs == SpinPairs.ALL*, - $\{(\uparrow, \downarrow), (\downarrow, \uparrow)\}$ if *spin_pairs == SpinPairs.DIFF*, and - $\{(\uparrow, \uparrow), (\downarrow, \downarrow)\}'if'spin_pairs == SpinPairs.SAME.$

Each potential parameter is a tuple `(dof, coefficient)`. For example, in the spinful model, it corresponds to the terms

$$-\mu \sum_i \sum_\sigma n_{i,a,\sigma},$$

where

- i runs over the sites of the lattice;
- a is the degree of freedom `dof`; and
- μ is the `coefficient`.

In the spinless model, the magnetic field is ignored.

class openfermion.hamiltonians.**HartreeFockFunctional**(*, *one_body_integrals: numpy.ndarray*, *two_body_integrals: numpy.ndarray*, *overlap: numpy.ndarray*, *n_electrons: int*, *model='rhf'*, *nuclear_repulsion: Optional[float] = 0.0*, *initial_orbitals: Union[None, Callable] = None*)

Implementation of the objective function code for Restricted Hartree-Fock

The object transforms a variety of input types into the appropriate output. It does this by analyzing the type and size of the input based on its knowledge of each type.

__init__(*, *one_body_integrals: numpy.ndarray*, *two_body_integrals: numpy.ndarray*, *overlap: numpy.ndarray*, *n_electrons: int*, *model='rhf'*, *nuclear_repulsion: Optional[float] = 0.0*, *initial_orbitals: Union[None, Callable] = None*)

Initialize functional

Parameters

- **one_body_integrals** – integrals in the atomic orbital basis for the one-body potential.
- **two_body_integrals** – integrals in the atomic obrital basis for the two-body potential ordered according to phi_{p}(r1)^{*}phi_{q}^{*}(r2) x phi_{r}(r2)phi_{s}(r1)
- **overlap** – overlap integrals in the atomic orbital basis
- **n_electrons** – number of electrons total
- **model** – Optional flag for performing restricted-, unrestricted-, or generalized- hartree-fock.
- **nuclear_repulsion** – Optional nuclear repulsion term. Energy is shifted by this amount. default is 0.

- **initial_orbitals** – Method for producing the initial orbitals from the atomic orbitals. Default is defining the core orbitals.

energy_from_rhf_opdm(*opdm_aa: numpy.ndarray*) → float

Compute the energy given a spin-up opdm

Parameters **opdm_aa** – spin-up opdm. Should be an n x n matrix where n is the number of spatial orbitals

Returns: RHF energy

rdms_from_rhf_opdm(*opdm_aa: numpy.ndarray*) → openfermion.ops._interaction_rdm.InteractionRDM

Generate spin-orbital InteractionRDM object from the alpha-spin opdm.

Parameters **opdm_aa** – single spin sector of the 1-particle denstiy matrix

Returns: InteractionRDM object for full spin-orbital 1-RDM and 2-RDM

rhf_global_gradient(*params: numpy.ndarray*, *alpha_opdm: numpy.ndarray*)

Compute rhf global gradient

Parameters

- **params** – rhf-parameters for rotation matrix.
- **alpha_opdm** – 1-RDM corresponding to results of basis rotation parameterized by 'params'.

Returns: gradient vector the same size as the input 'params'

class openfermion.hamiltonians.**MolecularData**(*geometry=None*, *basis=None*, *multiplicity=None*, *charge=0*, *description=''*, *filename=''*, *data_directory=None*)

Class for storing molecule data from a fixed basis set at a fixed geometry that is obtained from classical electronic structure packages. Not every field is filled in every calculation. All data that can (for some instance) exceed 10 MB should be saved separately. Data saved in HDF5 format.

geometry

A list of tuples giving the coordinates of each atom. An example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in angstrom. Use atomic symbols to specify atoms.

basis

A string giving the basis set. An example is 'cc-pvtz'.

charge

An integer giving the total molecular charge. Defaults to 0.

multiplicity

An integer giving the spin multiplicity.

description

An optional string giving a description. As an example, for dimers a likely description is the bond length (e.g. 0.7414).

name

A string giving a characteristic name for the instance.

filename

The name of the file where the molecule data is saved.

n_atoms

Integer giving the number of atoms in the molecule.

n_electrons

Integer giving the number of electrons in the molecule.

atoms
List of the atoms in molecule sorted by atomic number.

protons
List of atomic charges in molecule sorted by atomic number.

hf_energy
Energy from open or closed shell Hartree-Fock.

nuclear_repulsion
Energy from nuclei-nuclei interaction.

canonical_orbitals
numpy array giving canonical orbital coefficients.

n_orbitals
Integer giving total number of spatial orbitals.

n_qubits
Integer giving total number of qubits that would be needed.

orbital_energies
Numpy array giving the canonical orbital energies.

fock_matrix
Numpy array giving the Fock matrix.

overlap_integrals
Numpy array of AO overlap integrals

one_body_integrals
Numpy array of one-electron integrals

two_body_integrals
Numpy array of two-electron integrals

mp2_energy
Energy from MP2 perturbation theory.

cisd_energy
Energy from configuration interaction singles + doubles.

cisd_one_rdm
Numpy array giving 1-RDM from CISD calculation.

cisd_two_rdm
Numpy array giving 2-RDM from CISD calculation.

fci_energy
Exact energy of molecule within given basis.

fci_one_rdm
Numpy array giving 1-RDM from FCI calculation.

fci_two_rdm
Numpy array giving 2-RDM from FCI calculation.

ccsd_energy
Energy from coupled cluster singles + doubles.

ccsd_single_amps
Numpy array holding single amplitudes

ccsd_double_amps
Numpy array holding double amplitudes

general_calculations
A dictionary storing general calculation results for this system annotated by the key.

__init__(*geometry=None*, *basis=None*, *multiplicity=None*, *charge=0*, *description="*, *filename="*, *data_directory=None*)
Initialize molecular metadata which defines class.

Parameters

- **geometry** – A list of tuples giving the coordinates of each atom. An example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in angstrom. Use atomic symbols to specify atoms. Only optional if loading from file.
- **basis** – A string giving the basis set. An example is 'cc-pvtz'. Only optional if loading from file.
- **charge** – An integer giving the total molecular charge. Defaults to 0. Only optional if loading from file.
- **multiplicity** – An integer giving the spin multiplicity. Only optional if loading from file.
- **description** – A optional string giving a description. As an example, for dimers a likely description is the bond length (e.g. 0.7414).
- **filename** – An optional string giving name of file. If filename is not provided, one is generated automatically.
- **data_directory** – Optional data directory to change from default data directory specified in config file.

get_active_space_integrals(*occupied_indices=None*, *active_indices=None*)
Restricts a molecule at a spatial orbital level to an active space

This active space may be defined by a list of active indices and doubly occupied indices. Note that one_body_integrals and two_body_integrals must be defined n an orthonormal basis set.

Parameters

- **occupied_indices** – A list of spatial orbital indices indicating which orbitals should be considered doubly occupied.
- **active_indices** – A list of spatial orbital indices indicating which orbitals should be considered active.

Returns

tuple – Tuple with the following entries:

core_constant: Adjustment to constant shift in Hamiltonian from integrating out core orbitals

one_body_integrals_new: one-electron integrals over active space.

two_body_integrals_new: two-electron integrals over active space.

get_from_file(*property_name*)
Helper routine to re-open HDF5 file and pull out single property

Parameters **property_name** – Property name to load from self.filename

Returns

The data located at file[property_name] for the HDF5 file at self.filename. Returns None if the key is not found in the file.

`get_integrals``()`
Method to return 1-electron and 2-electron integrals in MO basis.

Returns

one_body_integrals –

An array of the one-electron integrals having shape of (n_orbitals, n_orbitals).

two_body_integrals: An array of the two-electron integrals having shape of (n_orbitals, n_orbitals, n_orbitals, n_orbitals).

Raises `MisissingCalculationError` – If integrals are not calculated.

`get_molecular_hamiltonian``(`*occupied_indices=None*, *active_indices=None*`)`
Output arrays of the second quantized Hamiltonian coefficients.

Parameters

- **`occupied_indices`** (*`list`*) – A list of spatial orbital indices indicating which orbitals should be considered doubly occupied.
- **`active_indices`** (*`list`*) – A list of spatial orbital indices indicating which orbitals should be considered active.

Returns *molecular_hamiltonian* – An instance of the MolecularOperator class.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`get_molecular_rdm``(`*use_fci=False*`)`
Method to return 1-RDM and 2-RDMs from CISD or FCI.

Parameters **`use_fci`** – Boolean indicating whether to use RDM from FCI calculation.

Returns *rdm* – An instance of the MolecularRDM class.

Raises `MisissingCalculationError` – If the CI calculation has not been performed.

`get_n_alpha_electrons``()`
Return number of alpha electrons.

`get_n_beta_electrons``()`
Return number of beta electrons.

`init_lazy_properties``()`
Initializes properties loaded on demand to None

`save``()`
Method to save the class under a systematic name.

`openfermion.hamiltonians.`**`bose_hubbard`**`(`*x_dimension*, *y_dimension*, *tunneling*, *interaction*, *chemical_potential=0.0*, *dipole=0.0*, *periodic=True*`)`
Return symbolic representation of a Bose-Hubbard Hamiltonian.

In this model, bosons move around on a lattice, and the energy of the model depends on where the bosons are.

The lattice is described by a 2D grid, with dimensions *x_dimension* x *y_dimension*. It is also possible to specify if the grid has periodic boundary conditions or not.

The Hamiltonian for the Bose-Hubbard model has the form

$$H = -t \sum_{\langle i,j \rangle} (b_i^\dagger b_j + b_j^\dagger b_i) + V \sum_{\langle i,j \rangle} b_i^\dagger b_i b_j^\dagger b_j + \frac{U}{2} \sum_i b_i^\dagger b_i (b_i^\dagger b_i - 1) - \mu \sum_i b_i^\dagger b_i.$$

where

- The indices $\langle i, j \rangle$ run over pairs i and j of nodes that are connected to each other in the grid
- t is the tunneling amplitude
- U is the on-site interaction potential
- μ is the chemical potential
- V is the dipole or nearest-neighbour interaction potential

Parameters

- **`x_dimension`** (*`int`*) – The width of the grid.
- **`y_dimension`** (*`int`*) – The height of the grid.
- **`tunneling`** (*`float`*) – The tunneling amplitude t.
- **`interaction`** (*`float`*) – The attractive local interaction strength U.
- **`chemical_potential`** (*`float, optional`*) – The chemical potential μ at each site. Default value is 0.
- **`periodic`** (*`bool, optional`*) – If True, add periodic boundary conditions. Default is True.
- **`dipole`** (*`float`*) – The attractive dipole interaction strength V.

Returns *bose_hubbard_model* – An instance of the BosonOperator class.

`openfermion.hamiltonians.`**`dual_basis_external_potential`**(*grid*, *geometry*, *spinless*, *non_periodic=False*, *period_cutoff=None*)

Return the external potential in the dual basis of arXiv:1706.00023.

The external potential resulting from electrons interacting with nuclei in the plane wave dual basis. Note that a cos term is used which is strictly only equivalent under aliasing in odd grids, and amounts to the addition of an extra term to make the diagonals real on even grids. This approximation is not expected to be significant and allows for use of even and odd grids on an even footing.

Parameters

- **`grid`** (`Grid`) – The discretization to use.
- **`geometry`** – A list of tuples giving the coordinates of each atom. example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in atomic units. Use atomic symbols to specify atoms.
- **`spinless`** (*`bool`*) – Whether to use the spinless model or not.
- **`non_periodic`** (*`bool`*) – If the system is non-periodic, default to False.
- **`period_cutoff`** (*`float`*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions)

Returns *FermionOperator* – The dual basis operator.

`openfermion.hamiltonians.`**`dual_basis_jellium_model`**(*grid*, *spinless=False*, *kinetic=True*, *potential=True*, *include_constant=False*, *non_periodic=False*, *period_cutoff=None*)

Return jellium Hamiltonian in the dual basis of arXiv:1706.00023

Parameters

- **grid** (`Grid`) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **kinetic** (*bool*) – Whether to include kinetic terms.
- **potential** (*bool*) – Whether to include potential terms.
- **include_constant** (*bool*) – Whether to include the Madelung constant. Note constant is unsupported for non-uniform, non-cubic cells with ions.
- **non_periodic** (*bool*) – If the system is non-periodic, default to False.
- **period_cutoff** (*float*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions).

Returns operator (FermionOperator)

`openfermion.hamiltonians.`**`dual_basis_kinetic`**(*grid*, *spinless=False*)

Return the kinetic operator in the dual basis of arXiv:1706.00023.

Parameters

- **grid** (`Grid`) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.

Returns operator (FermionOperator)

`openfermion.hamiltonians.`**`dual_basis_potential`**(*grid*, *spinless=False*, *non_periodic=False*, *period_cutoff=None*)

Return the potential operator in the dual basis of arXiv:1706.00023

Parameters

- **grid** (`Grid`) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **non_periodic** (*bool*) – If the system is non-periodic, default to False.
- **period_cutoff** (*float*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions).

Returns operator (FermionOperator)

`openfermion.hamiltonians.`**`fermi_hubbard`**(*x_dimension*, *y_dimension*, *tunneling*, *coulomb*, *chemical_potential=0.0*, *magnetic_field=0.0*, *periodic=True*, *spinless=False*, *particle_hole_symmetry=False*)

Return symbolic representation of a Fermi-Hubbard Hamiltonian.

The idea of this model is that some fermions move around on a grid and the energy of the model depends on where the fermions are. The Hamiltonians of this model live on a grid of dimensions *x_dimension* x *y_dimension*. The grid can have periodic boundary conditions or not. In the standard Fermi-Hubbard model (which we call the "spinful" model), there is room for an "up" fermion and a "down" fermion at each site on the grid. In this model, there are a total of *2N* spin-orbitals, where *N* = *x_dimension* * *y_dimension* is the number of sites. In the spinless model, there is only one spin-orbital per site for a total of *N*.

The Hamiltonian for the spinful model has the form

$$H = -\,t \sum_{\langle i,j \rangle} \sum_{\sigma} (a^\dagger_{i,\sigma} a_{j,\sigma} + a^\dagger_{j,\sigma} a_{i,\sigma}) + U \sum_{i} a^\dagger_{i,\uparrow} a_{i,\uparrow} a^\dagger_{i,\downarrow} a_{i,\downarrow} \tag{1.14}$$

$$-\,\mu \sum_{i} \sum_{\sigma} a^\dagger_{i,\sigma} a_{i,\sigma} - h \sum_{i} (a^\dagger_{i,\uparrow} a_{i,\uparrow} - a^\dagger_{i,\downarrow} a_{i,\downarrow}) \tag{1.15}$$

where

- The indices $\langle i, j \rangle$ run over pairs i and j of sites that are connected to each other in the grid
- $\sigma \in \{\uparrow, \downarrow\}$ is the spin
- t is the tunneling amplitude
- U is the Coulomb potential
- μ is the chemical potential
- h is the magnetic field

One can also construct the Hamiltonian for the spinless model, which has the form

$$H = -t \sum_{\langle i,j \rangle} (a^\dagger_i a_j + a^\dagger_j a_i) + U \sum_{\langle i,j \rangle} a^\dagger_i a_i a^\dagger_j a_j - \mu \sum_i a_i^\dagger a_i.$$

Parameters

- **`x_dimension`** (*`int`*) – The width of the grid.
- **`y_dimension`** (*`int`*) – The height of the grid.
- **`tunneling`** (*`float`*) – The tunneling amplitude t.
- **`coulomb`** (*`float`*) – The attractive local interaction strength U.
- **`chemical_potential`** (*`float, optional`*) – The chemical potential μ at each site. Default value is 0.
- **`magnetic_field`** (*`float, optional`*) – The magnetic field h at each site. Default value is 0. Ignored for the spinless case.
- **`periodic`** (*`bool, optional`*) – If True, add periodic boundary conditions. Default is True.
- **`spinless`** (*`bool, optional`*) – If True, return a spinless Fermi-Hubbard model. Default is False.
- **`particle_hole_symmetry`** (*`bool, optional`*) – If False, the repulsion term corresponds to:

  $$U \sum_{k=1}^{N-1} a_k^\dagger a_k a_{k+1}^\dagger a_{k+1}$$

  If True, the repulsion term is replaced by:

  $$U \sum_{k=1}^{N-1} (a_k^\dagger a_k - \frac{1}{2})(a_{k+1}^\dagger a_{k+1} - \frac{1}{2})$$

  which is unchanged under a particle-hole transformation. Default is False

Returns *hubbard_model* – An instance of the FermionOperator class.

`openfermion.hamiltonians.`**`get_matrix_of_eigs`**(*w: numpy.ndarray*) → numpy.ndarray

Transform the eigenvalues into a matrix corresponding to summing the adjoint rep.

Parameters **`w`** – eigenvalues of C-matrix

Returns: new array of transformed eigenvalues

`openfermion.hamiltonians.`**`hypercube_grid_with_given_wigner_seitz_radius_and_filling`**(*dimension*, *grid_lengtl* *wigner_sei* *fill-* *ing_fractio* *spin-* *less=True*)

Return a Grid with the same number of orbitals along each dimension with the specified Wigner-Seitz radius.

Parameters

- **`dimension`** (*`int`*) – The number of spatial dimensions.
- **`grid_length`** (*`int`*) – The number of orbitals along each dimension.
- **`wigner_seitz_radius`** (*`float`*) – The Wigner-Seitz radius per particle, in Bohr.
- **`filling_fraction`** (*`float`*) – The average spin-orbital occupation. Specifies the number of particles (rounding down).
- **`spinless`** (*`boolean`*) – Whether to give the system without or with spin.

`openfermion.hamiltonians.`**`jellium_model`**(*grid*, *spinless=False*, *plane_wave=True*, *include_constant=False*, *e_cutoff=None*, *non_periodic=False*, *period_cutoff=None*)

Return jellium Hamiltonian as FermionOperator class.

Parameters

- **`grid`** (`openfermion.utils.Grid`) – The discretization to use.
- **`spinless`** (*`bool`*) – Whether to use the spinless model or not.
- **`plane_wave`** (*`bool`*) – Whether to return in momentum space (True) or position space (False).
- **`include_constant`** (*`bool`*) – Whether to include the Madelung constant. Note constant is unsupported for non-uniform, non-cubic cells with ions.
- **`e_cutoff`** (*`float`*) – Energy cutoff.
- **`non_periodic`** (*`bool`*) – If the system is non-periodic, default to False.
- **`period_cutoff`** (*`float`*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions).

Returns *FermionOperator* – The Hamiltonian of the model.

`openfermion.hamiltonians.`**`jordan_wigner_dual_basis_hamiltonian`**(*grid*, *geometry=None*, *spinless=False*, *include_constant=False*)

Return the dual basis Hamiltonian as QubitOperator.

Parameters

- **`grid`** (`Grid`) – The discretization to use.

- **geometry** – A list of tuples giving the coordinates of each atom. example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in atomic units. Use atomic symbols to specify atoms.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **include_constant** (*bool*) – Whether to include the Madelung constant.

Returns hamiltonian (QubitOperator)

openfermion.hamiltonians.**jordan_wigner_dual_basis_jellium**(*grid*, *spinless=False*, *include_constant=False*)

Return the jellium Hamiltonian as QubitOperator in the dual basis.

Parameters

- **grid** (Grid) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **include_constant** (*bool*) – Whether to include the Madelung constant. Note constant is unsupported for non-uniform, non-cubic cells with ions.

Returns hamiltonian (QubitOperator)

openfermion.hamiltonians.**load_molecular_hamiltonian**(*geometry*, *basis*, *multiplicity*, *description*, *n_active_electrons=None*, *n_active_orbitals=None*)

Attempt to load a molecular Hamiltonian with the given properties.

Parameters

- **geometry** – A list of tuples giving the coordinates of each atom. An example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in angstrom. Use atomic symbols to specify atoms.
- **basis** – A string giving the basis set. An example is 'cc-pvtz'. Only optional if loading from file.
- **multiplicity** – An integer giving the spin multiplicity.
- **description** – A string giving a description.
- **n_active_electrons** – An optional integer specifying the number of electrons desired in the active space.
- **n_active_orbitals** – An optional integer specifying the number of spatial orbitals desired in the active space.

Returns The Hamiltonian as an InteractionOperator.

openfermion.hamiltonians.**make_atom**(*atom_type*, *basis*, *filename="*)

Prepare a molecular data instance for a single element.

Parameters

- **atom_type** – Float giving atomic symbol.
- **basis** – The basis in which to perform the calculation.

Returns *atom* – An instance of the MolecularData class.

openfermion.hamiltonians.**make_atomic_lattice**(*nx_atoms*, *ny_atoms*, *nz_atoms*, *spacing*, *basis*, *atom_type='H'*, *charge=0*, *filename="*)

Function to create atomic lattice with n_atoms.

Parameters

- **nx_atoms** – Integer, the length of lattice (in number of atoms).
- **ny_atoms** – Integer, the width of lattice (in number of atoms).
- **nz_atoms** – Integer, the depth of lattice (in number of atoms).
- **spacing** – The spacing between atoms in the lattice in Angstroms.
- **basis** – The basis in which to perform the calculation.
- **atom_type** – String, the atomic symbol of the element in the ring. this defaults to 'H' for Hydrogen.
- **charge** – An integer giving the total molecular charge. Defaults to 0.
- **filename** – An optional string to give a filename for the molecule.

Returns *molecule* – A an instance of the MolecularData class.

Raises `MolecularLatticeError` – If lattice specification is invalid.

`openfermion.hamiltonians.`**`make_atomic_ring`**(*n_atoms*, *spacing*, *basis*, *atom_type='H'*, *charge=0*, *filename=''*)

Function to create atomic rings with n_atoms.

Note that basic geometry suggests that for spacing L between atoms the radius of the ring should be L / (2 * cos (pi / 2 - theta / 2))

Parameters

- **n_atoms** – Integer, the number of atoms in the ring.
- **spacing** – The spacing between atoms in the ring in Angstroms.
- **basis** – The basis in which to perform the calculation.
- **atom_type** – String, the atomic symbol of the element in the ring. this defaults to 'H' for Hydrogen.
- **charge** – An integer giving the total molecular charge. Defaults to 0.
- **filename** – An optional string to give a filename for the molecule.

Returns *molecule* – A an instance of the MolecularData class.

`openfermion.hamiltonians.`**`mean_field_dwave`**(*x_dimension*, *y_dimension*, *tunneling*, *sc_gap*, *chemical_potential=0.0*, *periodic=True*)

Return symbolic representation of a BCS mean-field d-wave Hamiltonian.

The Hamiltonians of this model live on a grid of dimensions *x_dimension* x *y_dimension*. The grid can have periodic boundary conditions or not. Each site on the grid can have an "up" fermion and a "down" fermion. Therefore, there are a total of *2N* spin-orbitals, where *N = x_dimension * y_dimension* is the number of sites.

The Hamiltonian for this model has the form

$$H = -t \sum_{\langle i,j \rangle} \sum_{\sigma} (a^\dagger_{i,\sigma} a_{j,\sigma} + a^\dagger_{j,\sigma} a_{i,\sigma}) - \mu \sum_i \sum_{\sigma} a^\dagger_{i,\sigma} a_{i,\sigma} \tag{1.16}$$

$$- \sum_{\langle i,j \rangle} \Delta_{ij} (a^\dagger_{i,\uparrow} a^\dagger_{j,\downarrow} - a^\dagger_{i,\downarrow} a^\dagger_{j,\uparrow} + a_{j,\downarrow} a_{i,\uparrow} - a_{j,\uparrow} a_{i,\downarrow}) \tag{1.17}$$

where

- The indices $\langle i, j \rangle$ run over pairs i and j of sites that are connected to each other in the grid
- $\sigma \in \{\uparrow, \downarrow\}$ is the spin
- t is the tunneling amplitude

- Δ_{ij} is equal to $+\Delta/2$ for horizontal edges and $-\Delta/2$ for vertical edges, where Δ is the superconducting gap.
- μ is the chemical potential

Parameters

- **`x_dimension`** (*`int`*) – The width of the grid.
- **`y_dimension`** (*`int`*) – The height of the grid.
- **`tunneling`** (*`float`*) – The tunneling amplitude t.
- **`sc_gap`** (*`float`*) – The superconducting gap Δ
- **`chemical_potential`** (*`float, optional`*) – The chemical potential μ at each site. Default value is 0.
- **`periodic`** (*`bool, optional`*) – If True, add periodic boundary conditions. Default is True.

Returns *mean_field_dwave_model* – An instance of the FermionOperator class.

`openfermion.hamiltonians.`**`plane_wave_external_potential`**(*grid*, *geometry*, *spinless*, *e_cutoff=None*, *non_periodic=False*, *period_cutoff=None*)

Return the external potential operator in plane wave basis.

The external potential resulting from electrons interacting with nuclei. It is defined here as the Fourier transform of the dual basis Hamiltonian such that is spectrally equivalent in the case of both even and odd grids. Otherwise, the two differ in the case of even grids.

Parameters

- **`grid`** (`Grid`) – The discretization to use.
- **`geometry`** – A list of tuples giving the coordinates of each atom. example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in atomic units. Use atomic symbols to specify atoms.
- **`spinless`** – Bool, whether to use the spinless model or not.
- **`e_cutoff`** (*`float`*) – Energy cutoff.
- **`non_periodic`** (*`bool`*) – If the system is non-periodic, default to False.
- **`period_cutoff`** (*`float`*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions)

Returns *FermionOperator* – The plane wave operator.

`openfermion.hamiltonians.`**`plane_wave_hamiltonian`**(*grid*, *geometry=None*, *spinless=False*, *plane_wave=True*, *include_constant=False*, *e_cutoff=None*, *non_periodic=False*, *period_cutoff=None*)

Returns Hamiltonian as FermionOperator class.

Parameters

- **`grid`** (`Grid`) – The discretization to use.
- **`geometry`** – A list of tuples giving the coordinates of each atom. example is [('H', (0, 0, 0)), ('H', (0, 0, 0.7414))]. Distances in atomic units. Use atomic symbols to specify atoms.

- **spinless** (*bool*) – Whether to use the spinless model or not.
- **plane_wave** (*bool*) – Whether to return in plane wave basis (True) or plane wave dual basis (False).
- **include_constant** (*bool*) – Whether to include the Madelung constant.
- **e_cutoff** (*float*) – Energy cutoff.
- **non_periodic** (*bool*) – If the system is non-periodic, default to False.
- **period_cutoff** (*float*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions)

Returns *FermionOperator* – The hamiltonian.

openfermion.hamiltonians.**plane_wave_kinetic**(*grid*, *spinless=False*, *e_cutoff=None*)

Return the kinetic energy operator in the plane wave basis.

Parameters

- **grid** (openfermion.utils.Grid) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **e_cutoff** (*float*) – Energy cutoff.

Returns *FermionOperator* – The kinetic momentum operator.

openfermion.hamiltonians.**plane_wave_potential**(*grid*, *spinless=False*, *e_cutoff=None*, *non_periodic=False*, *period_cutoff=None*)

Return the e-e potential operator in the plane wave basis.

Parameters

- **grid** (Grid) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **e_cutoff** (*float*) – Energy cutoff.
- **non_periodic** (*bool*) – If the system is non-periodic, default to False.
- **period_cutoff** (*float*) – Period cutoff, default to grid.volume_scale() ** (1. / grid.dimensions).

Returns operator (FermionOperator)

openfermion.hamiltonians.**rhf_minimization**(*rhf_object: openfermion.hamiltonians._hartree_fock.HartreeFockFunctional*, *method: Optional[str] = 'CG'*, *initial_guess: Union[None, numpy.ndarray] = None*, *verbose: Optional[bool] = True*, *sp_options: Union[None, Dict[KT, VT]] = None*) → scipy.optimize.optimize.OptimizeResult

Perform Hartree-Fock energy minimization

Parameters

- **rhf_object** – An instantiation of the HartreeFockFunctional
- **method** – (optional) scipy optimization method
- **initial_guess** – (optional) initial rhf parameter vector. If None zero vector is used.
- **verbose** – (optional) turn on printing. This is passed to the scipy 'disp' option.

- **sp_options** –

Returns: scipy.optimize result object

openfermion.hamiltonians.**rhf_params_to_matrix**(*parameters: numpy.ndarray*, *num_orbitals: int*, *occ: Union[None, List[int]] = None*, *virt: Union[None, List[int]] = None*) → numpy.ndarray

For restricted Hartree-Fock we have nocc * nvirt parameters. These are provided as a list that is ordered by (virtuals) imes (occupied).

For example, for H4 we have 2 orbitals occupied and 2 virtuals

occupied = [0, 1] virtuals = [2, 3]

parameters = [(v_{0}, o_{0}), (v_{0}, o_{1}), (v_{1}, o_{0}), (v_{1}, o_{1})] = [(2, 0), (2, 1), (3, 0), (3, 1)]

You can think of the tuples of elements of the upper right triangle of the antihermitian matrix that specifies the c_{b, i} coefficients.

coefficient matrix [[c_{0, 0}, -c_{1, 0}, -c_{2, 0}, -c_{3, 0}],

[c_{1, 0}, c_{1, 1}, -c_{2, 1}, -c_{3, 1}], [c_{2, 0}, c_{2, 1}, c_{2, 2}, -c_{3, 2}], [c_{3, 0}, c_{3, 1}, c_{3, 2}, c_{3, 3}]]

Since we are working with only non-redundant operators we know c_{i, i} = 0 and any c_{i, j} where i and j are both in occupied or both in virtual = 0.

Parameters

- **parameters** – array of parameters for kappa matrix
- **num_orbitals** – total number of spatial orbitals
- **occ** – (Optional) indices for doubly occupied sector
- **virt** – (Optional) indices for virtual sector

Returns: np.ndarray kappa matrix

openfermion.hamiltonians.**wigner_seitz_length_scale**(*wigner_seitz_radius*, *n_particles*, *dimension*)

Function to give length_scale associated with Wigner-Seitz radius.

Parameters

- **wigner_seitz_radius** (*float*) – The radius per particle in atomic units.
- **n_particles** (*int*) – The number of particles in the simulation cell.
- **dimension** (*int*) – The dimension of the system.

Returns *length_scale (float)* – The length scale for the simulation.

Raises `ValueError` – System dimension must be a positive integer.

1.2 openfermion.measurements

openfermion.measurements.**apply_constraints**(*operator*, *n_fermions*)

Function to use linear programming to apply constraints.

Parameters

- **operator** (FermionOperator) – FermionOperator with only 1- and 2-body terms that we wish to vectorize.

- **n_fermions** (*int*) – The number of particles in the simulation.

Returns

modified_operator(FermionOperator) –

The operator with reduced norm that has been modified with equality constraints.

openfermion.measurements.**binary_partition_iterator**(*qubit_list*, *num_iterations=None*)
Generator for a list of 2-partitions of N qubits such that all pairs of qubits are split in at least one partition, This follows a variation on ArXiv:1908.0562 - instead of explicitly partitioning the list based on the binary indices of the qubits, we repeatedly divide the list in two and then zip it back together.

Parameters

- **qubit_list** (*list*) – list of qubits to be partitioned
- **num_iterations** (*int or None*) – number of iterations to perform. If None, will be set to ceil(log2(len(qubit_list)))

Returns *partition(iterator of tuples of lists)* – the required partitioning

openfermion.measurements.**constraint_matrix**(*n_orbitals*, *n_fermions*)
Function to generate matrix of constraints.

Parameters

- **n_orbitals** (*int*) – The number of orbitals in the simulation.
- **n_fermions** (*int*) – The number of particles in the simulation.

Returns *constraint_matrix(scipy.sparse.coo_matrix)* – The matrix of constraints.

openfermion.measurements.**linearize_term**(*term*, *n_orbitals*)
Function to return integer index of term indices.

Parameters

- **term** (*tuple*) – The term indices of a one- or two-body FermionOperator.
- **n_orbitals** (*int*) – The number of orbitals in the simulation.

Returns *index(int)* – The index of the term.

openfermion.measurements.**one_body_fermion_constraints**(*n_orbitals*, *n_fermions*)
Generates one-body positivity constraints on fermionic RDMs.

The specific constraints implemented are known positivity constraints on the one-fermion reduced density matrices. Constraints are generated in the form of FermionOperators whose expectation value is known to be zero for any N-Representable state. Generators are used for efficiency.

Parameters

- **n_orbitals** (*int*) – number of spin-orbitals on which operators act.
- **n_fermions** (*int*) – number of fermions in the system.

Yields Constraint is a FermionOperator with zero expectation value.

openfermion.measurements.**partition_iterator**(*qubit_list*, *partition_size*, *num_iterations=None*)
Generator for a list of k-partitions of N qubits such that all sets of k qubits are perfectly split in at least one partition, following ArXiv:1908.05628

Parameters

- **qubit_list** (*list*) – list of qubits to be partitioned

- **partition_size** (*int*) – the number of sets in the partition.
- **num_iterations** (*int or None*) – the number of iterations in the outer iterator. If None, set to ceil(log2(len(qubit_list)))

Returns *partition(iterator of tuples of lists)* – the required partitioning

openfermion.measurements.**pauli_string_iterator**(*num_qubits*, *max_word_size=2*)
Generates a set of Pauli strings such that each word of k Pauli operators lies in at least one string.

Parameters

- **num_qubits** (*int*) – number of qubits in string
- **max_word_size** (*int*) – maximum required word

Returns

pauli_string(iterator of strings) –

iterator over Pauli strings

openfermion.measurements.**prony**(*signal*)
Estimates amplitudes and phases of a sparse signal using Prony's method.

Single-ancilla quantum phase estimation returns a signal g(k)=sum (aj*exp(i*k*phij)), where aj and phij are the amplitudes and corresponding eigenvalues of the unitary whose phases we wish to estimate. When more than one amplitude is involved, Prony's method provides a simple estimation tool, which achieves near-Heisenberg-limited scaling (error scaling as N^{-1/2}K^{-3/2}).

Parameters **signal** (*1d complex array*) – the signal to fit

Returns

amplitudes(list of complex values) – the amplitudes a_i, in descending order by their complex magnitude phases(list of complex values): the complex frequencies gamma_i,

correlated with amplitudes.

openfermion.measurements.**two_body_fermion_constraints**(*n_orbitals*, *n_fermions*)
Generates two-body positivity constraints on fermionic RDMs.

The specific constraints implemented are known positivity constraints on the two-fermion reduced density matrices. Constraints are generated in the form of FermionOperators whose expectation value is known to be zero for any N-Representable state. Generators are used for efficiency.

Parameters

- **n_orbitals** (*int*) – number of spin-orbitals on which operators act.
- **n_fermions** (*int*) – number of fermions in the system.

Yields Constraint is a FermionOperator with zero expectation value.

openfermion.measurements.**unlinearize_term**(*index*, *n_orbitals*)
Function to return integer index of term indices.

Parameters

- **index** (*int*) – The index of the term.
- **n_orbitals** (*int*) – The number of orbitals in the simulation.

Returns *term(tuple)* – The term indices of a one- or two-body FermionOperator.

1.3 openfermion.ops

class `openfermion.ops.`**`BinaryCode`**(*encoding*, *decoding*)

Bases: `object`

The BinaryCode class provides a representation of an encoding-decoding pair for binary vectors of different lengths, where the decoding is allowed to be non-linear.

As the occupation number of fermionic mode is effectively binary, a length-N vector (v) of binary number can be utilized to describe a configuration of a many-body fermionic state on N modes. An n-qubit product state configuration |w0> |w1> |w2> … |wn-1>, on the other hand is described by a length-n binary vector w=(w0, w1, …, wn-1). To map a subset of N-Orbital Fermion states to n-qubit states we define a binary code, which consists of a (here: linear) encoding (e) and a (non-linear) decoding (d), such that for every v from that subset, w = e(v) is a length-n binary vector with d(w) = v. This can be used to save qubits given a Hamiltonian that dictates such a subset, otherwise n=N.

Two binary codes (e,d) and (e',d') can construct a third code (e",d") by two possible operations:

Concatenation: (e",d") = (e,d) * (e',d') which means e": v" -> e'(e(v")) and d": w" -> d(d'(w")) where n" = n' and N" = N, with n = N' as necessary condition.

Appendage: (e",d") = (e,d) + (e',d') which means e": (v + v') -> e(v) + e'(v') and d": (w + w') -> d(w) + d'(w') where the addition is to be understood as appending two vectors together, so N" = N' + N and n" = n + n'.

Appending codes is particularly useful when considering segment codes or segmented transforms.

A BinaryCode-instance is initialized by BinaryCode(A,d), given the encoding (e) as n x N array or matrix-like nested lists A, such that e(v) = (A v) mod 2. The decoding d is an array or a list input of length N, which has entries either of type BinaryPolynomial, or of valid type for an input of the BinaryPolynomial-constructor.

The signs + and *, += and *= are overloaded to implement concatenation and appendage on BinaryCode-objects.

NOTE: multiplication of a BinaryCode with an integer yields a multiple appending of the same code, the multiplication with another BinaryCode their concatenation.

`decoder`

list of BinaryPolynomial: Outputs the decoding functions as components.

Type list

`encoder`

Outputs A, the linear matrix that implements the encoding function.

Type scipy.sparse.csc_matrix

`n_modes`

Outputs the number of modes.

Type int

`n_qubits`

Outputs the number of qubits.

Type int

`__init__`(*encoding*, *decoding*)

Initialization of a binary code.

Parameters

- **`encoding`** (*`np.ndarray or list`*) – nested lists or binary 2D-array
- **`decoding`** (*`array or list`*) – list of BinaryPolynomial (list or str).

Raises

- `TypeError` – non-list, array like encoding or decoding, unsuitable BinaryPolynomial generators,
- `BinaryCodeError` – in case of decoder/encoder size mismatch or decoder size, qubits indexed mismatch

class `openfermion.ops.`**`BinaryPolynomial`**(*term=None*)

Bases: `object`

The BinaryPolynomial class provides an analytic representation of non-linear binary functions. An instance of this class describes a term of binary variables (variables of the values {0,1}, indexed by integers like w0, w1, w2 and so on) that is considered to be evaluated modulo 2. This implies the following set of rules:

the binary addition w1 + w1 = 0, binary multiplication w2 * w2 = w2 and power rule w3 ^ 0 = 1, where raising to every other integer power than zero reproduces w3.

Of course, we can also add a non-trivial constant, which is 1. Due to these binary rules, every function available will be a multinomial like e.g.

1 + w1 w2 + w0 w1 .

These binary functions are used for non-linear binary codes in order to decompress qubit bases back into fermion bases. In that instance, one BinaryPolynomial object characterizes the occupation of single orbital given a multi-qubit state in configuration |w0> |w1> |w2> . . . .

For initialization, the preferred data types is either a string of the multinomial, where each variable and constant is to be well separated by a whitespace, or in its native form of tuples, 1 + w1 w2 + w0 w1 is represented as [(_SYMBOLIC_ONE,),(1,2),(0,1)]

After initialization,BinaryPolynomial terms can be manipulated with the overloaded signs +, * and ^, according to the binary rules mentioned.

Example

```
bin_fun = BinaryPolynomial('1 + w1 w2 + w0 w1')
# Equivalently
bin_fun = BinaryPolynomial(1) + BinaryPolynomial([(1,2),(0,1)])
# Equivalently
bin_fun = BinaryPolynomial([(_SYMBOLIC_ONE,),(1,2),(0,1)])
```

`terms`

a list of tuples. Each tuple represents a summand of the BinaryPolynomial term and each summand can contain multiple tuples representing the factors.

Type list

`__init__`(*term=None*)

Initialize the BinaryPolynomial based on term

Parameters **`term`** (*`str, list, tuple`*) – used for initializing a BinaryPolynomial

Raises `ValueError` – when term is not a string,list or tuple

`enumerate_qubits`()

Enumerates all qubits indexed in a given BinaryPolynomial.

Returns (list): a list of qubits

`evaluate`(*binary_list*)

Evaluates a BinaryPolynomial

Parameters **binary_list** (*list, array, str*) – a list of binary values corresponding each binary variable (in order of their indices) in the expression

Returns (int, 0 or 1): result of the evaluation

Raises BinaryPolynomialError – Length of list provided must match the number of qubits indexed in BinaryPolynomial

classmethod identity()

Returns *multiplicative_identity (BinaryPolynomial)* – A symbolic operator u with the property that u*x = x*u = x for all operators x of the same class.

shift(*const*)

Shift all qubit indices by a given constant.

Parameters **const** (*int*) – the constant to shift the indices by

Raises TypeError – const must be integer

classmethod zero()

Returns *additive_identity (BinaryPolynomial)* – A symbolic operator o with the property that o+x = x+o = x for all operators x of the same class.

class openfermion.ops.**BosonOperator**(*term=None*, *coefficient=1.0*)

Bases: openfermion.ops._symbolic_operator.SymbolicOperator

BosonOperator stores a sum of products of bosonic ladder operators.

In OpenFermion, we describe bosonic ladder operators using the shorthand: 'i^' = b^dagger_i 'j' = b_j where ['i', 'j^'] = delta_ij is the commutator.

One can multiply together these bosonic ladder operators to obtain a bosonic term. For instance, '2^ 1' is a bosonic term which creates at mode 2 and destroys at mode 1. The BosonicOperator class also stores a coefficient for the term, e.g. '3.17 * 2^ 1'.

The BosonOperator class is designed (in general) to store sums of these terms. For instance, an instance of BosonOperator might represent 3.17 2^ 1 - 66.2 * 8^ 7 6^ 2 The Bosonic Operator class overloads operations for manipulation of these objects by the user.

BosonOperator is a subclass of SymbolicOperator. Importantly, it has attributes set as follows:

```
actions = (1, 0)
action_strings = ('^', '')
action_before_index = False
different_indices_commute = True
```

See the documentation of SymbolicOperator for more details.

Example

```
H = (BosonOperator('0^ 3', .5)
       + .5 * BosonOperator('3^ 0'))
# Equivalently
H2 = BosonOperator('0^ 3', 0.5)
H2 += BosonOperator('3^ 0', 0.5)
```

Note: Adding BosonOperator is faster using += (as this is done by in-place addition). Specifying the coefficient during initialization is faster than multiplying a BosonOperator with a scalar.

action_before_index
Whether action comes before index in string representations.

action_strings
The string representations of the allowed actions.

actions
The allowed actions.

different_indices_commute
Whether factors acting on different indices commute.

is_boson_preserving()
Query whether the term preserves particle number.

This is equivalent to requiring the same number of raising and lowering operators in each term.

is_normal_ordered()
Return whether or not term is in normal order.

In our convention, ladder operators come first. Note that unlike the Fermion operator, due to the commutation of ladder operators with different indices, the BosonOperator sorts ladder operators by index.

class openfermion.ops.**DiagonalCoulombHamiltonian**(*one_body*, *two_body*, *constant=0.0*)
Bases: `object`

Class for storing Hamiltonians of the form

$$\sum_{p,q} T_{pq} a_p^\dagger a_q + \sum_{p,q} V_{pq} a_p^\dagger a_p a_q^\dagger a_q + \text{constant}$$

where

- T is a Hermitian matrix.
- V is a real symmetric matrix.

one_body
The Hermitian matrix T.

Type ndarray

two_body
The real symmetric matrix V.

Type ndarray

constant
The constant.

Type float

__init__(*one_body*, *two_body*, *constant=0.0*)
Initialize self. See help(type(self)) for accurate signature.

class openfermion.ops.**FermionOperator**(*term=None*, *coefficient=1.0*)
Bases: `openfermion.ops._symbolic_operator.SymbolicOperator`

FermionOperator stores a sum of products of fermionic ladder operators.

In OpenFermion, we describe fermionic ladder operators using the shorthand: 'q^' = a^dagger_q 'q' = a_q where {'p^', 'q'} = delta_pq

One can multiply together these fermionic ladder operators to obtain a fermionic term. For instance, '2^ 1' is a fermion term which creates at orbital 2 and destroys at orbital 1. The FermionOperator class also stores a coefficient for the term, e.g. '3.17 * 2^ 1'.

The FermionOperator class is designed (in general) to store sums of these terms. For instance, an instance of FermionOperator might represent 3.17 2^ 1 - 66.2 * 8^ 7 6^ 2 The Fermion Operator class overloads operations for manipulation of these objects by the user.

FermionOperator is a subclass of SymbolicOperator. Importantly, it has attributes set as follows:

```
actions = (1, 0)
action_strings = ('^', '')
action_before_index = False
different_indices_commute = False
```

See the documentation of SymbolicOperator for more details.

Example

```
ham = (FermionOperator('0^ 3', .5)
       + .5 * FermionOperator('3^ 0'))
# Equivalently
ham2 = FermionOperator('0^ 3', 0.5)
ham2 += FermionOperator('3^ 0', 0.5)
```

Note: Adding FermionOperators is faster using += (as this is done by in-place addition). Specifying the coefficient during initialization is faster than multiplying a FermionOperator with a scalar.

`action_before_index`
Whether action comes before index in string representations.

`action_strings`
The string representations of the allowed actions.

`actions`
The allowed actions.

`different_indices_commute`
Whether factors acting on different indices commute.

`is_normal_ordered`()
Return whether or not term is in normal order.

In our convention, normal ordering implies terms are ordered from highest tensor factor (on left) to lowest (on right). Also, ladder operators come first.

`is_two_body_number_conserving`(*check_spin_symmetry=False*)
Query whether operator has correct form to be from a molecule.

Require that term is particle-number conserving (same number of raising and lowering operators). Require that term has 0, 2 or 4 ladder operators. Require that term conserves spin (parity of raising operators equals parity of lowering operators).

Parameters **`check_spin_symmetry`** (`bool`) – Whether to check if operator conserves spin.

class `openfermion.ops.`**`InteractionOperator`**(*constant*, *one_body_tensor*, *two_body_tensor*)
Bases: `openfermion.ops._polynomial_tensor.PolynomialTensor`

Class for storing 'interaction operators' which are defined to be fermionic operators consisting of one-body and two-body terms which conserve particle number and spin. The most common examples of data that will use this structure are molecular Hamiltonians. In principle, everything stored in this class could also be represented using the more general FermionOperator class. However, this class is able to exploit specific properties of how fermions interact to enable more numerically efficient manipulation of the data. Note that the operators stored in this class take the form:

$$constant + \sum_{p,q} h_{p,q} a_p^\dagger a_q + \sum_{p,q,r,s} h_{p,q,r,s} a_p^\dagger a_q^\dagger a_r a_s.$$

`one_body_tensor`
The coefficients of the one-body terms

`(:math:`h_{p, q}`). This is an n_qubits x n_qubits`

`numpy array of floats.`

`two_body_tensor`
The coefficients of the two-body terms ($h_{p,q,r,s}$). This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array of floats.

`__init__`(*constant*, *one_body_tensor*, *two_body_tensor*)
Initialize the InteractionOperator class.

Parameters

- **`constant`** – A constant term in the operator given as a float. For instance, the nuclear repulsion energy.
- **`one_body_tensor`** –

  The coefficients of the one-body terms ($h_{p,q}$).

  This is an n_qubits x n_qubits numpy array of floats.
- **`two_body_tensor`** – The coefficients of the two-body terms ($h_{p,q,r,s}$). This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array of floats.

`one_body_tensor`
The value of the one-body tensor.

`two_body_tensor`
The value of the two-body tensor.

`unique_iter`(*complex_valued=False*)
Iterate all terms that are not in the same symmetry group.

Four point symmetry:

1. pq = qp.
2. pqrs = srqp = qpsr = rspq.

Eight point symmetry:

1. pq = qp.
2. pqrs = rqps = psrq = srqp = qpsr = rspq = spqr = qrsp.

Parameters **`complex_valued`** (*`bool`*) – Whether the operator has complex coefficients.

Yields tuple[int]

`class` `openfermion.ops.`**`InteractionRDM`**(*one_body_tensor*, *two_body_tensor*)
Bases: `openfermion.ops._polynomial_tensor.PolynomialTensor`

Class for storing 1- and 2-body reduced density matrices.

`one_body_tensor`
The expectation values <a^dagger_p a_q>.

`two_body_tensor`
The expectation values <a^dagger_p a^dagger_q a_r a_s>.

`__init__`(*one_body_tensor*, *two_body_tensor*)
Initialize the InteractionRDM class.

Parameters

- **`one_body_tensor`** – Expectation values <a^dagger_p a_q>.
- **`two_body_tensor`** – Expectation values <a^dagger_p a^dagger_q a_r a_s>.

`expectation`(*operator*)
Return expectation value of an InteractionRDM with an operator.

Parameters **`operator`** – A QubitOperator or InteractionOperator.

Returns *float* – Expectation value

Raises `InteractionRDMError` – Invalid operator provided.

`get_qubit_expectations`(*qubit_operator*)
Return expectations of QubitOperator in new QubitOperator.

Parameters **`qubit_operator`** – QubitOperator instance to be evaluated on this InteractionRDM.

Returns *QubitOperator* – QubitOperator with coefficients corresponding to expectation values of those operators.

Raises `InteractionRDMError` – Observable not contained in 1-RDM or 2-RDM.

`one_body_tensor`
The value of the one-body tensor.

`two_body_tensor`
The value of the two-body tensor.

class `openfermion.ops.`**`IsingOperator`**(*term=None*, *coefficient=1.0*)
Bases: `openfermion.ops._symbolic_operator.SymbolicOperator`

The IsingOperator class provides an analytic representation of an Ising-type Hamiltonian, i.e. a sum of product of Zs.

IsingOperator is a subclass of SymbolicOperator. Importantly, it has attributes set as follows:

actions = (‘Z’) action_strings = (‘Z’) action_before_index = True different_indices_commute = True

See the documentation of SymbolicOperator for more details.

`action_before_index`
Whether action comes before index in string representations.

`action_strings`
The string representations of the allowed actions.

`actions`
The allowed actions.

`different_indices_commute`
Whether factors acting on different indices commute.

class openfermion.ops.**MajoranaOperator**(*term=None*, *coefficient=1.0*)

Bases: `object`

A linear combination of products of Majorana operators.

A system of N fermionic modes can be described using 2N Majorana operators $\gamma_1, \ldots, \gamma_{2N}$ as an alternative to using N fermionic annihilation operators. The algebra of Majorana operators amounts to the relation

$$\{\gamma_i, \gamma_j\} = \gamma_i \gamma_j + \gamma_j \gamma_i = 2\delta_{ij}$$

Note that this implies $\gamma_i^2 = 1$.

The MajoranaOperator class stores a linear combination of products of Majorana operators. Each product is represented as a tuple of integers representing the indices of the operators. As an example, *MajoranaOperator((2, 3, 5), -1.5)* initializes an operator with a single term which represents the operator $-1.5\gamma_2\gamma_3\gamma_5$. MajoranaOperators can be added, subtracted, multiplied, and divided by scalars. They can be compared for approximate numerical equality using ==.

terms

A dictionary from term, represented by a tuple of integers,

to the coefficient of the term in the linear combination.

__init__(*term=None*, *coefficient=1.0*)

Initialize a MajoranaOperator with a single term.

Parameters

- **term** (*Tuple[int]*) – The indices of a Majorana operator term to start off with
- **coefficient** (*complex*) – The coefficient of the term

Returns MajoranaOperator

commutes_with(*other*)

Test commutation with another MajoranaOperator

static from_dict(*terms*)

Initialize a MajoranaOperator from a terms dictionary.

WARNING: The given dictionary is not validated whatsoever. It's up to you to ensure that it is properly formed.

Parameters terms – A dictionary from Majorana term to coefficient

with_basis_rotated_by(*transformation_matrix*)

Change to a basis of new Majorana operators.

The input to this method is a real orthogonal matrix O. It returns a new MajoranaOperator which is equivalent to the old one but rewritten in terms of a new basis of Majorana operators. Let the original Majorana operators be denoted by γ_i and the new operators be denoted by $\tilde{\gamma_i}$. Then they are related by the equation

$$\tilde{\gamma_i} = \sum_j O_{ij}\gamma_j.$$

Parameters transformation_matrix – A real orthogonal matrix representing the basis transformation.

Returns The rotated operator.

class openfermion.ops.**PolynomialTensor**(*n_body_tensors*)

Bases: `object`

Class for storing tensor representations of operators that correspond with multilinear polynomials in the fermionic ladder operators. For instance, in a quadratic Hamiltonian (degree 2 polynomial) which conserves particle number, there are only terms of the form a^dagger_p a_q, and the coefficients can be stored in an n_qubits x n_qubits matrix. Higher order terms would be described with tensors of higher dimension. Note that each tensor must have an even number of dimensions, since parity is conserved. Much of the functionality of this class is redudant with FermionOperator but enables much more efficient numerical computations in many cases, such as basis rotations.

n_qubits
The number of sites on which the tensor acts.

Type int

n_body_tensors
A dictionary storing the tensors describing n-body interactions. The keys are tuples that indicate the type of tensor. For instance, n_body_tensors[(1, 0)] would be an (n_qubits x n_qubits) numpy array, and it could represent the coefficients of terms of the form a^dagger_i a_j, whereas n_body_tensors[(0, 1)] would be an array of the same shape, but instead representing terms of the form a_i a^dagger_j.

Type dict

__init__(*n_body_tensors*)
Initialize the PolynomialTensor class.

Parameters **n_body_tensors** (*dict*) – A dictionary storing the tensors describing n-body interactions.

constant
The value of the constant term.

projected_n_body_tensors(*selection*, *exact=False*)
Keep only selected elements.

Parameters

- **selection** (*Union[int, Iterable[int]]*) – If int, keeps terms with at most (exactly, if exact is True) that many unique indices. If iterable, keeps only terms containing (all of, if exact is True) the specified indices.
- **exact** (*bool*) – Whether or not the selection is strict.

rotate_basis(*rotation_matrix*)
Rotate the orbital basis of the PolynomialTensor.

Parameters **rotation_matrix** – A square numpy array or matrix having dimensions of n_qubits by n_qubits. Assumed to be real and invertible.

class openfermion.ops.**QuadOperator**(*term=None*, *coefficient=1.0*)
Bases: openfermion.ops._symbolic_operator.SymbolicOperator

QuadOperator stores a sum of products of canonical quadrature operators.

They are defined in terms of the bosonic ladder operators: q = sqrt{hbar/2}(b+b^) p = -isqrt{hbar/2}(b-b^) where hbar is a constant appearing in the commutator of q and p: [q, p] = i hbar

In OpenFermion, we describe the canonical quadrature operators acting on quantum modes 'i' and 'j' using the shorthand: 'qi' = q_i 'pj' = p_j where ['qi', 'pj'] = i hbar delta_ij is the commutator.

The QuadOperator class is designed (in general) to store sums of these terms. For instance, an instance of QuadOperator might represent

```
H = 0.5 * QuadOperator('q0 p5') + 0.3 * QuadOperator('q0')
```

Note for a QuadOperator to be a Hamiltonian which is a hermitian operator, the coefficients of all terms must be real.

QuadOperator is a subclass of SymbolicOperator. Importantly, it has attributes set as follows:

```
actions = ('q', 'p')
action_strings = ('q', 'p')
action_before_index = True
different_indices_commute = True
```

See the documentation of SymbolicOperator for more details.

Example

```
H = (QuadOperator('p0 q3', 0.5)
       + 0.6 * QuadOperator('p3 q0'))
# Equivalently
H2 = QuadOperator('p0 q3', 0.5)
H2 += QuadOperator('p3 q0', 0.6)
```

Note: Adding QuadOperator is faster using += (as this is done by in-place addition). Specifying the coefficient during initialization is faster than multiplying a QuadOperator with a scalar.

action_before_index
Whether action comes before index in string representations.

action_strings
The string representations of the allowed actions.

actions
The allowed actions.

different_indices_commute
Whether factors acting on different indices commute.

is_gaussian()
Query whether the term is quadratic or lower in the quadrature operators.

is_normal_ordered()
Return whether or not term is in normal order.

In our convention, q operators come first. Note that unlike the Fermion operator, due to the commutation of quadrature operators with different indices, the QuadOperator sorts quadrature operators by index.

class openfermion.ops.**QuadraticHamiltonian**(*hermitian_part*, *antisymmetric_part=None*, *constant=0.0*, *chemical_potential=0.0*)
Bases: openfermion.ops._polynomial_tensor.PolynomialTensor

Class for storing Hamiltonians that are quadratic in the fermionic ladder operators. The operators stored in this class take the form

$$\sum_{p,q}(M_{pq} - \mu\delta_{pq})a_p^\dagger a_q + \frac{1}{2}\sum_{p,q}(\Delta_{pq}a_p^\dagger a_q^\dagger + \text{h.c.}) + \text{constant}$$

where

- M is a Hermitian *n_qubits* x *n_qubits* matrix.
- Δ is an antisymmetric *n_qubits* x *n_qubits* matrix.

- μ is a real number representing the chemical potential.
- δ_{pq} is the Kronecker delta symbol.

We separate the chemical potential μ from M so that we can use it to adjust the expectation value of the total number of particles.

`chemical_potential`
The chemical potential μ.

Type float

`__init__`(*hermitian_part*, *antisymmetric_part=None*, *constant=0.0*, *chemical_potential=0.0*)
Initialize the QuadraticHamiltonian class.

Parameters

- **`hermitian_part`** (*`ndarray`*) – The matrix M, which represents the coefficients of the particle-number-conserving terms. This is an *n_qubits* x *n_qubits* numpy array of complex numbers.
- **`antisymmetric_part`** (*`ndarray`*) – The matrix Δ, which represents the coefficients of the non-particle-number-conserving terms. This is an *n_qubits* x *n_qubits* numpy array of complex numbers.
- **`constant`** (*`float, optional`*) – A constant term in the operator.
- **`chemical_potential`** (*`float, optional`*) – The chemical potential μ.

`add_chemical_potential`(*chemical_potential*)
Increase (or decrease) the chemical potential by some value.

`antisymmetric_part`
The antisymmetric part.

`combined_hermitian_part`
The Hermitian part including the chemical potential.

`conserves_particle_number`
Whether this Hamiltonian conserves particle number.

`diagonalizing_bogoliubov_transform`(*spin_sector=None*)
Compute the unitary that diagonalizes a quadratic Hamiltonian.

Any quadratic Hamiltonian can be rewritten in the form

$$\sum_j \varepsilon_j b_j^\dagger b_j + \text{constant},$$

where the $b_j^\dagger$ are a new set fermionic creation operators that satisfy the canonical anticommutation relations. The new creation operators are linear combinations of the original ladder operators. In the most general case, creation and annihilation operators are mixed together:

$$\begin{pmatrix} b_1^\dagger \\ \vdots \\ b_N^\dagger \end{pmatrix} = W \begin{pmatrix} a_1^\dagger \\ \vdots \\ a_N^\dagger \\ a_1 \\ \vdots \\ a_N \end{pmatrix},$$

where W is an $N \times (2N)$ matrix. However, if the Hamiltonian conserves particle number then creation operators don't need to be mixed with annihilation operators and W only needs to be an $N \times N$ matrix:

$$\begin{pmatrix} b_1^\dagger \\ \vdots \\ b_N^\dagger \end{pmatrix} = W \begin{pmatrix} a_1^\dagger \\ \vdots \\ a_N^\dagger \end{pmatrix},$$

This method returns the matrix W.

Parameters **`spin_sector`** (*`optional str`*) – An optional integer specifying a spin sector to restrict to: 0 for spin-up and 1 for spin-down. Should only be specified if the Hamiltonian includes a spin degree of freedom and spin-up modes do not interact with spin-down modes. If specified, the modes are assumed to be ordered so that spin-up orbitals come before spin-down orbitals.

Returns

orbital_energies(ndarray) A one-dimensional array containing the ε_j

diagonalizing_unitary (ndarray): A matrix representing the transformation W of the fermionic ladder operators. If the Hamiltonian conserves particle number then this is $N \times N$; otherwise it is $N \times 2N$. If spin sector is specified, then N here represents the number of spatial orbitals rather than spin orbitals.

constant(float) The constant

`diagonalizing_circuit``()`

Get a circuit for a unitary that diagonalizes this Hamiltonian

This circuit performs the transformation to a basis in which the Hamiltonian takes the diagonal form

$$\sum_j \varepsilon_j b_j^\dagger b_j + \text{constant}.$$

Returns *circuit_description (list[tuple])* – A list of operations describing the circuit. Each operation is a tuple of objects describing elementary operations that can be performed in parallel. Each elementary operation is either the string 'pht' indicating a particle-hole transformation on the last fermionic mode, or a tuple of the form (i, j, θ, φ), indicating a Givens rotation of modes i and j by angles θ and φ.

`ground_energy``()`

Return the ground energy.

`hermitian_part`

The Hermitian part not including the chemical potential.

`majorana_form``()`

Return the Majorana represention of the Hamiltonian.

Any quadratic Hamiltonian can be written in the form

$$\frac{i}{2} \sum_{j,k} A_{jk} f_j f_k + \text{constant}$$

where the f_i are normalized Majorana fermion operators:

$$f_j = \frac{1}{\sqrt{2}}(a_j^\dagger + a_j)$$
$$f_{j+N} = \frac{i}{\sqrt{2}}(a_j^\dagger - a_j)$$

and A is a (2 * *n_qubits*) x (2 * *n_qubits*) real antisymmetric matrix. This function returns the matrix A and the constant.

orbital_energies(*non_negative=False*)
Return the orbital energies.

Any quadratic Hamiltonian is unitarily equivalent to a Hamiltonian of the form

$$\sum_j \varepsilon_j b_j^\dagger b_j + \text{constant}.$$

We call the ε_j the orbital energies. The eigenvalues of the Hamiltonian are sums of subsets of the orbital energies (up to the additive constant).

Parameters **non_negative** (*bool*) – If True, always return a list of orbital energies that are non-negative. This option is ignored if the Hamiltonian does not conserve particle number, in which case the returned orbital energies are always non-negative.

Returns

- *orbital_energies(ndarray)* – A one-dimensional array containing the ε_j
- *constant(float)* – The constant

class openfermion.ops.**QubitOperator**(*term=None*, *coefficient=1.0*)
Bases: `openfermion.ops._symbolic_operator.SymbolicOperator`

A sum of terms acting on qubits, e.g., 0.5 * 'X0 X5' + 0.3 * 'Z1 Z2'.

A term is an operator acting on n qubits and can be represented as:

coefficient * local_operator[0] x … x local_operator[n-1]

where x is the tensor product. A local operator is a Pauli operator ('I', 'X', 'Y', or 'Z') which acts on one qubit. In math notation a term is, for example, 0.5 * 'X0 X5', which means that a Pauli X operator acts on qubit 0 and 5, while the identity operator acts on all other qubits.

A QubitOperator represents a sum of terms acting on qubits and overloads operations for easy manipulation of these objects by the user.

Note for a QubitOperator to be a Hamiltonian which is a hermitian operator, the coefficients of all terms must be real.

```
hamiltonian = 0.5 * QubitOperator('X0 X5') + 0.3 * QubitOperator('Z0')
```

QubitOperator is a subclass of SymbolicOperator. Importantly, it has attributes set as follows:

```
actions = ('X', 'Y', 'Z')
action_strings = ('X', 'Y', 'Z')
action_before_index = True
different_indices_commute = True
```

See the documentation of SymbolicOperator for more details.

Example

```
ham = ((QubitOperator('X0 Y3', 0.5)
        + 0.6 * QubitOperator('X0 Y3')))
# Equivalently
ham2 = QubitOperator('X0 Y3', 0.5)
ham2 += 0.6 * QubitOperator('X0 Y3')
```

Note: Adding QubitOperators is faster using += (as this is done by in-place addition). Specifying the coefficient during initialization is faster than multiplying a QubitOperator with a scalar.

`action_before_index`
Whether action comes before index in string representations.

`action_strings`
The string representations of the allowed actions.

`actions`
The allowed actions.

`different_indices_commute`
Whether factors acting on different indices commute.

`renormalize``()`
Fix the trace norm of an operator to 1

`class` `openfermion.ops.`**`SymbolicOperator`**`(`*term=None*, *coefficient=1.0*`)`
Bases: `object`

Base class for FermionOperator and QubitOperator.

A SymbolicOperator stores an object which represents a weighted sum of terms; each term is a product of individual factors of the form (*index*, *action*), where *index* is a nonnegative integer and the possible values for *action* are determined by the subclass. For instance, for the subclass FermionOperator, *action* can be 1 or 0, indicating raising or lowering, and for QubitOperator, *action* is from the set {'X', 'Y', 'Z'}. The coefficients of the terms are stored in a dictionary whose keys are the terms. SymbolicOperators of the same type can be added or multiplied together.

Note: Adding SymbolicOperators is faster using += (as this is done by in-place addition). Specifying the coefficient during initialization is faster than multiplying a SymbolicOperator with a scalar.

`actions`
A tuple of objects representing the possible actions. e.g. for FermionOperator, this is (1, 0).

Type tuple

`action_strings`
A tuple of string representations of actions. These should be in one-to-one correspondence with actions and listed in the same order. e.g. for FermionOperator, this is ('^', '').

Type tuple

`action_before_index`
A boolean indicating whether in string representations, the action should come before the index.

Type bool

`different_indices_commute`
A boolean indicating whether factors acting on different indices commute.

Type bool

`terms`
key (tuple of tuples): A dictionary storing the coefficients of the terms in the operator. The keys are the terms. A term is a product of individual factors; each factor is represented by a tuple of the form (*index*, *action*), and these tuples are collected into a larger tuple which represents the term as the product of its factors.

Type dict

`__init__`(*term=None*, *coefficient=1.0*)
Initialize self. See help(type(self)) for accurate signature.

classmethod accumulate(*operators*, *start=None*)
Sums over SymbolicOperators.

action_before_index
Whether action comes before index in string representations.

Example: For QubitOperator, the actions are ('X', 'Y', 'Z') and the string representations look something like 'X0 Z2 Y3'. So the action comes before the index, and this function should return True. For FermionOperator, the string representations look like '0^ 1 2^ 3'. The action comes after the index, so this function should return False.

action_strings
The string representations of the allowed actions.

Returns a tuple containing string representations of the possible actions, in the same order as the *actions* property.

actions
The allowed actions.

Returns a tuple of objects representing the possible actions.

compress(*abs_tol=1e-08*)
Eliminates all terms with coefficients close to zero and removes small imaginary and real parts.

Parameters **abs_tol** (*float*) – Absolute tolerance, must be at least 0.0

constant
The value of the constant term.

different_indices_commute
Whether factors acting on different indices commute.

get_operator_groups(*num_groups*)
Gets a list of operators with a few terms. :param num_groups: How many operators to get in the end. :type num_groups: int

Returns

operators([self.__class__]) –

A list of operators summing up to self.

get_operators()
Gets a list of operators with a single term.

Returns *operators([self.__class__])* – A generator of the operators in self.

classmethod identity()

Returns *multiplicative_identity (SymbolicOperator)* – A symbolic operator u with the property that u*x = x*u = x for all operators x of the same class.

induced_norm(*order=1*)
Compute the induced p-norm of the operator.

If we represent an operator as :math: *sum_{j} w_j H_j* where :math: *w_j* are scalar coefficients then this norm is :math: *left(sum_{j} | w_j |^p right)^{frac{1}{p}} where :math: 'p* is the order of the induced norm

Parameters **order** (*int*) – the order of the induced norm.

many_body_order()
Compute the many-body order of a SymbolicOperator.

The many-body order of a SymbolicOperator is the maximum length of a term with nonzero coefficient.

Returns int

classmethod **zero**()

Returns *additive_identity (SymbolicOperator)* – A symbolic operator o with the property that o+x = x+o = x for all operators x of the same class.

openfermion.ops.**down_index**(*index*)
Function to return down-orbital index given a spatial orbital index.

Parameters **index** (*int*) – spatial orbital index

Returns An integer representing the index of the associated spin-down orbital

openfermion.ops.**general_basis_change**(*general_tensor*, *rotation_matrix*, *key*)
Change the basis of an general interaction tensor.

M'^{p_1p_2…p_n} = R^{p_1}_{a_1} R^{p_2}_{a_2} … R^{p_n}_{a_n} M^{a_1a_2…a_n} R^{p_n}_{a_n}^T … R^{p_2}_{a_2}^T R_{p_1}_{a_1}^T

where R is the rotation matrix, M is the general tensor, M' is the transformed general tensor, and a_k and p_k are indices. The formula uses the Einstein notation (implicit sum over repeated indices).

In case R is complex, the k-th R in the above formula need to be conjugated if key has a 1 in the k-th place (meaning that the corresponding operator is a creation operator).

Parameters

- **general_tensor** – A square numpy array or matrix containing information about a general interaction tensor.
- **rotation_matrix** – A square numpy array or matrix having dimensions of n_qubits by n_qubits. Assumed to be unitary.
- **key** – A tuple indicating the type of general_tensor. Assumed to be non-empty. For example, a tensor storing coefficients of $a^\dagger_p a_q$ would have a key of (1, 0) whereas a tensor storing coefficients of $a^\dagger_p a_q a_r a^\dagger_s$ would have a key of (1, 0, 0, 1).

Returns *transformed_general_tensor* – general_tensor in the rotated basis.

openfermion.ops.**up_index**(*index*)
Function to return up-orbital index given a spatial orbital index.

Parameters **index** (*int*) – spatial orbital index

Returns An integer representing the index of the associated spin-up orbital

1.4 openfermion.transforms

openfermion.transforms.**binary_code_transform**(*hamiltonian*, *code*)
Transforms a Hamiltonian written in fermionic basis into a Hamiltonian written in qubit basis, via a binary code.

The role of the binary code is to relate the occupation vectors (v0 v1 v2 … vN-1) that span the fermionic basis, to the qubit basis, spanned by binary vectors (w0, w1, w2, …, wn-1).

The binary code has to provide an analytic relation between the binary vectors (v0, v1, …, vN-1) and (w0, w1, …, wn-1), and possibly has the property N>n, when the Fermion basis is smaller than the fermionic Fock space. The binary_code_transform function can transform Fermion operators to qubit operators for custom- and qubit-saving mappings.

Note: Logic multi-qubit operators are decomposed into Pauli-strings (e.g. CPhase(1,2) = 0.5 * (1 + Z1 + Z2 - Z1 Z2)), which might increase the number of Hamiltonian terms drastically.

Parameters

- **`hamiltonian`** (`FermionOperator`) – the fermionic Hamiltonian
- **`code`** (`BinaryCode`) – the binary code to transform the Hamiltonian

Returns (QubitOperator): the transformed Hamiltonian

Raises

- `TypeError` – if the hamiltonian is not a FermionOperator or code is not
- a BinaryCode

`openfermion.transforms.`**`bravyi_kitaev`**(*operator*, *n_qubits=None*)

Apply the Bravyi-Kitaev transform.

Implementation from arXiv:quant-ph/0003137 and "A New Data Structure for Cumulative Frequency Tables" by Peter M. Fenwick.

Note that this implementation is equivalent to the one described in arXiv:1208.5986, and is different from the one described in arXiv:1701.07072. The one described in arXiv:1701.07072 is implemented in OpenFermion as *bravyi_kitaev_tree*.

Parameters

- **`operator`** (`openfermion.ops.FermionOperator`) – A FermionOperator to transform.
- **`n_qubits`** (*`int | None`*) – Can force the number of qubits in the resulting operator above the number that appear in the input operator.

Returns *transformed_operator* – An instance of the QubitOperator class.

Raises `ValueError` – Invalid number of qubits specified.

`openfermion.transforms.`**`bravyi_kitaev_code`**(*n_modes*)

The Bravyi-Kitaev transform as binary code. The implementation follows arXiv:1208.5986.

Parameters **`n_modes`** (*`int`*) – number of modes

Returns (BinaryCode): The Bravyi-Kitaev BinaryCode

`openfermion.transforms.`**`bravyi_kitaev_fast`**(*operator*)

Find the Pauli-representation of InteractionOperator for Bravyi-Kitaev Super fast (BKSF) algorithm. Pauli-representation of general FermionOperator is not possible in BKSF. Also, the InteractionOperator given as input must be Hermitian. In future we might provide a transformation for a restricted set of fermion operator.

Parameters **`operator`** – Interaction Operator.

Returns *transformed_operator* – An instance of the QubitOperator class.

Raises `TypeError` – If operator is not an InteractionOperator

`openfermion.transforms.`**`bravyi_kitaev_tree`**(*operator*, *n_qubits=None*)

Apply the "tree" Bravyi-Kitaev transform.

Implementation from arxiv:1701.07072

Note that this implementation is different from the one described in arXiv:quant-ph/0003137. In particular, it gives different results when the total number of modes is not a power of 2. The one described in arXiv:quant-ph/0003137 is the same as the one described in arXiv:1208.5986, and it is implemented in OpenFermion under the name *bravyi_kitaev*.

Parameters

- **`operator`** (`openfermion.ops.FermionOperator`) – A FermionOperator to transform.
- **`n_qubits`** (*`int | None`*) – Can force the number of qubits in the resulting operator above the number that appear in the input operator.

Returns *transformed_operator* – An instance of the QubitOperator class.

Raises `ValueError` – Invalid number of qubits specified.

`openfermion.transforms.`**`checksum_code`**(*n_modes*, *odd*)

Checksum code for either even or odd Hamming weight. The Hamming weight is defined such that it yields the total occupation number for a given basis state. A Checksum code with odd weight will encode all states with odd occupation number. This code saves one qubit: n_qubits = n_modes - 1.

Parameters

- **`n_modes`** (*`int`*) – number of modes
- **`odd`** (*`int or bool`*) – 1 (True) or 0 (False), if odd, we encode all states with odd Hamming weight

Returns (BinaryCode): The checksum BinaryCode

`openfermion.transforms.`**`dissolve`**(*term*)

Decomposition helper. Takes a product of binary variables and outputs the Pauli-string sum that corresponds to the decomposed multi-qubit operator.

Parameters **`term`** (*`tuple`*) – product of binary variables, i.e.: ‘w0 w2 w3’

Returns (QubitOperator): superposition of Pauli-strings

Raises `ValueError` – if the variable in term is not integer

`openfermion.transforms.`**`edit_hamiltonian_for_spin`**(*qubit_hamiltonian*, *spin_orbital*, *orbital_parity*)

Removes the Z terms acting on the orbital from the Hamiltonian.

`openfermion.transforms.`**`get_boson_operator`**(*operator*, *hbar=1.0*)

Convert to BosonOperator.

Parameters

- **`operator`** – QuadOperator.
- **`hbar`** (*`float`*) – the value of hbar used in the definition of the commutator [q_i, p_j] = i hbar delta_ij. By default hbar=1.

Returns *boson_operator* – An instance of the BosonOperator class.

`openfermion.transforms.`**`get_diagonal_coulomb_hamiltonian`**(*fermion_operator*, *n_qubits=None*, *ignore_incompatible_terms=False*)

Convert a FermionOperator to a DiagonalCoulombHamiltonian.

Parameters

- **fermion_operator** (FermionOperator) – The operator to convert.
- **n_qubits** (*int*) – Optionally specify the total number of qubits in the system
- **ignore_incompatible_terms** (*bool*) – This flag determines the behavior of this method when it encounters terms that are not represented by the DiagonalCoulombHamiltonian class, namely, terms that are not quadratic and not quartic of the form a^dagger_p a_p a^dagger_q a_q. If set to True, this method will simply ignore those terms. If False, then this method will raise an error if it encounters such a term. The default setting is False.

openfermion.transforms.**get_fermion_operator**(*operator*)

Convert to FermionOperator.

Returns *fermion_operator* – An instance of the FermionOperator class.

openfermion.transforms.**get_interaction_operator**(*fermion_operator*, *n_qubits=None*)

Convert a 2-body fermionic operator to InteractionOperator.

This function should only be called on fermionic operators which consist of only a_p^dagger a_q and a_p^dagger a_q^dagger a_r a_s terms. The one-body terms are stored in a matrix, one_body[p, q], and the two-body terms are stored in a tensor, two_body[p, q, r, s].

Returns *interaction_operator* – An instance of the InteractionOperator class.

Raises

- TypeError – Input must be a FermionOperator.
- TypeError – FermionOperator does not map to InteractionOperator.

Warning: Even assuming that each creation or annihilation operator appears at most a constant number of times in the original operator, the runtime of this method is exponential in the number of qubits.

openfermion.transforms.**get_interaction_rdm**(*qubit_operator*, *n_qubits=None*)

Build an InteractionRDM from measured qubit operators.

Returns: An InteractionRDM object.

openfermion.transforms.**get_majorana_operator**(*operator: Union[openfermion.ops._polynomial_tensor.PolynomialTe openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulon openfermion.ops._fermion_operator.FermionOperator]*) → openfermion.ops._majorana_operator.MajoranaOperator

Convert to MajoranaOperator.

Uses the convention of even + odd indexing of Majorana modes derived from a fermionic mode:

fermion annhil. c_k -> (gamma_{2k} + 1.j * gamma_{2k+1}) / 2 fermion creation c^_k -> (gamma_{2k} - 1.j * gamma_{2k+1}) / 2

Parameters (PolynomialTensor, (*operator*) – DiagonalCoulombHamiltonian or FermionOperator): Operator to write as Majorana Operator.

Returns *majorana_operator* – An instance of the MajoranaOperator class.

Raises TypeError – If operator is not of PolynomialTensor, DiagonalCoulombHamiltonian or FermionOperator.

openfermion.transforms.**get_molecular_data**(*interaction_operator*, *geometry=None*, *basis=None*, *multiplicity=None*, *n_electrons=None*, *reduce_spin=True*, *data_directory=None*)

Output a MolecularData object generated from an InteractionOperator

Parameters

- **interaction_operator** (InteractionOperator) – two-body interaction operator defining the "molecular interaction" to be simulated.
- **geometry** (*string or list of atoms*) –
- **basis** (*string*) – String denoting the basis set used to discretize the system.
- **multiplicity** (*int*) – Spin multiplicity desired in the system.
- **n_electrons** (*int*) – Number of electrons in the system
- **reduce_spin** (*bool*) – True if one wishes to perform spin reduction on integrals that are given in interaction operator. Assumes spatial (x) spin structure generically.

Returns *molecule(MolecularData)* – Instance that captures the interaction_operator converted into the format that would come from an electronic structure package adorned with some meta-data that may be useful.

openfermion.transforms.**get_number_preserving_sparse_operator**(*fermion_op*, *num_qubits*, *num_electrons*, *spin_preserving=False*, *reference_determinant=None*, *excitation_level=None*)

Initialize a Scipy sparse matrix in a specific symmetry sector.

This method initializes a Scipy sparse matrix from a FermionOperator, explicitly working in a particular particle number sector. Optionally, it can also restrict the space to contain only states with a particular Sz.

Finally, the Hilbert space can also be restricted to only those states which are reachable by excitations up to a fixed rank from an initial reference determinant.

Parameters

- **fermion_op** (FermionOperator) – An instance of the FermionOperator class. It should not contain terms which do not preserve particle number. If spin_preserving is set to True it should also not contain terms which do not preserve the Sz (it is assumed that the ordering of the indices goes alpha, beta, alpha, beta, . . .).
- **num_qubits** (*int*) – The total number of qubits / spin-orbitals in the system.
- **num_electrons** (*int*) – The number of particles in the desired Hilbert space.
- **spin_preserving** (*bool*) – Whether or not the constructed operator should be defined in a space which has support only on states with the same Sz value as the reference_determinant.
- **reference_determinant** (*list(bool)*) – A list, whose length is equal to num_qubits, which specifies which orbitals should be occupied in the reference state. If spin_preserving is set to True then the Sz value of this reference state determines the Sz value of the symmetry sector in which the generated operator acts. If a value for excitation_level is provided then the excitations are generated with respect to the reference state. In any case, the ordering of the states in the matrix representation of the operator depends

on reference_determinant and the state corresponding to reference_determinant is the vector [1.0, 0.0, 0.0 … 0.0]. Can be set to None in order to take the first num_electrons orbitals to be the occupied orbitals.

- **`excitation_level`** (*`int`*) – The number of excitations from the reference state which should be included in the generated operator's matrix representation. Can be set to None to include all levels of excitation.

Returns

sparse_op(scipy.sparse.csc_matrix) –

A sparse matrix representation of fermion_op in the basis set by the arguments.

`openfermion.transforms.`**`get_quad_operator`**(*operator*, *hbar=1.0*)

Convert to QuadOperator.

Parameters

- **`operator`** – BosonOperator.
- **`hbar`** (*`float`*) – the value of hbar used in the definition of the commutator [q_i, p_j] = i hbar delta_ij. By default hbar=1.

Returns *quad_operator* – An instance of the QuadOperator class.

`openfermion.transforms.`**`get_quadratic_hamiltonian`**(*fermion_operator*, *chemical_potential=0.0*, *n_qubits=None*, *ignore_incompatible_terms=False*)

Convert a quadratic fermionic operator to QuadraticHamiltonian.

Parameters

- **`fermion_operator`** (`FermionOperator`) – The operator to convert.
- **`chemical_potential`** (*`float`*) – A chemical potential to include in the returned operator
- **`n_qubits`** (*`int`*) – Optionally specify the total number of qubits in the system
- **`ignore_incompatible_terms`** (*`bool`*) – This flag determines the behavior of this method when it encounters terms that are not quadratic that is, terms that are not of the form a^dagger_p a_q. If set to True, this method will simply ignore those terms. If False, then this method will raise an error if it encounters such a term. The default setting is False.

Returns *quadratic_hamiltonian* – An instance of the QuadraticHamiltonian class.

Raises

- `TypeError` – Input must be a FermionOperator.
- `TypeError` – FermionOperator does not map to QuadraticHamiltonian.

Warning: Even assuming that each creation or annihilation operator appears at most a constant number of times in the original operator, the runtime of this method is exponential in the number of qubits.

`openfermion.transforms.`**`get_sparse_operator`**(*operator*, *n_qubits=None*, *trunc=None*, *hbar=1.0*)

Map an operator to a sparse matrix.

If the input is not a QubitOperator, the Jordan-Wigner Transform is used.

Parameters

- **operator** – Currently supported operators include: FermionOperator, QubitOperator, DiagonalCoulombHamiltonian, PolynomialTensor, BosonOperator, QuadOperator.
- **n_qubits** (*int*) – Number qubits in the system Hilbert space. Applicable only to fermionic systems.
- **trunc** (*int*) – The size at which the Fock space should be truncated. Applicable only to bosonic systems.
- **hbar** (*float*) – the value of hbar to use in the definition of the canonical commutation relation [q_i, p_j] = delta_{ij} i hbar. Applicable only to the QuadOperator.

openfermion.transforms.**interleaved_code**(*modes*)

Linear code that reorders orbitals from even-odd to up-then-down. In up-then-down convention, one can append two instances of the same code 'c' in order to have two symmetric subcodes that are symmetric for spin-up and -down modes: ' c + c '. In even-odd, one can concatenate with the interleaved_code to have the same result:' interleaved_code * (c + c)'. This code changes the order of modes from (0, 1 , 2, … , modes-1) to (0, modes/2, 1 modes/2+1, … , modes-1, modes/2 - 1). n_qubits = n_modes.

Args: modes (int): number of modes, must be even

Returns (BinaryCode): code that interleaves orbitals

openfermion.transforms.**jordan_wigner**(*operator*)

Apply the Jordan-Wigner transform to a FermionOperator, InteractionOperator, or DiagonalCoulombHamiltonian to convert to a QubitOperator.

Operators are mapped as follows: a_j^dagger -> Z_0 .. Z_{j-1} (X_j - iY_j) / 2 a_j -> Z_0 .. Z_{j-1} (X_j + iY_j) / 2

Returns *transformed_operator* – An instance of the QubitOperator class.

Warning: The runtime of this method is exponential in the maximum locality of the original FermionOperator.

Raises `TypeError` – Operator must be a FermionOperator, DiagonalCoulombHamiltonian, or InteractionOperator.

openfermion.transforms.**jordan_wigner_code**(*n_modes*)

The Jordan-Wigner transform as binary code.

Parameters **n_modes** (*int*) – number of modes

Returns (BinaryCode): The Jordan-Wigner BinaryCode

openfermion.transforms.**linearize_decoder**(*matrix*)

Outputs linear decoding function from input matrix

Parameters **matrix** (*np.ndarray or list*) – list of lists or 2D numpy array to derive the decoding function from

Returns (list): list of BinaryPolynomial

openfermion.transforms.**parity_code**(*n_modes*)

The parity transform (arXiv:1208.5986) as binary code. This code is very similar to the Jordan-Wigner transform, but with long update strings instead of parity strings. It does not save qubits: n_qubits = n_modes.

Parameters **n_modes** (*int*) – number of modes

Returns (BinaryCode): The parity transform BinaryCode `openfermion.transforms.`**`project_onto_sector`**(*operator*, *qubits*, *sectors*)

Remove qubit by projecting onto sector.

Takes a QubitOperator, and projects out a list of qubits, into either the +1 or -1 sector. Note - this requires knowledge of which sector we wish to project into.

Parameters

- **`operator`** – the QubitOperator to work on
- **`qubits`** – a list of indices of qubits in operator to remove
- **`sectors`** – for each qubit, whether to project into the 0 subspace (<Z>=1) or the 1 subspace (<Z>=-1).

Returns *projected_operator* – the resultant operator

Raises

- `TypeError` – operator must be a QubitOperator.
- `TypeError` – qubits and sector must be an array-like.
- `ValueError` – If qubits and sectors have different length.
- `ValueError` – If sector are not specified as 0 or 1.

`openfermion.transforms.`**`projection_error`**(*operator*, *qubits*, *sectors*)

Calculate the error from the project_onto_sector function.

Parameters

- **`operator`** – the QubitOperator to work on
- **`qubits`** – a list of indices of qubits in operator to remove
- **`sectors`** – for each qubit, whether to project into the 0 subspace (<Z>=1) or the 1 subspace (<Z>=-1).

Returns *error* – the trace norm of the removed term.

Raises

- `TypeError` – operator must be a QubitOperator.
- `TypeError` – qubits and sector must be an array-like.
- `ValueError` – If qubits and sectors have different length.
- `ValueError` – If sector are not specified as 0 or 1.

`openfermion.transforms.`**`reverse_jordan_wigner`**(*qubit_operator*, *n_qubits=None*)

Transforms a QubitOperator into a FermionOperator using the Jordan-Wigner transform.

Operators are mapped as follows: Z_j -> I - 2 a^dagger_j a_j X_j -> (a^dagger_j + a_j) Z_{j-1} Z_{j-2} .. Z_0 Y_j -> i (a^dagger_j - a_j) Z_{j-1} Z_{j-2} .. Z_0

Parameters

- **`qubit_operator`** – the QubitOperator to be transformed.
- **`n_qubits`** – the number of qubits term acts on. If not set, defaults to the maximum qubit number acted on by term.

Returns *transformed_term* – An instance of the FermionOperator class.

Raises

- `TypeError` – Input must be a QubitOperator.

- `TypeError` – Invalid number of qubits specified.
- `TypeError` – Pauli operators must be X, Y or Z.

`openfermion.transforms.`**`rotate_qubit_by_pauli`**(*qop*, *pauli*, *angle*)

Rotate qubit operator by exponential of Pauli.

Perform the rotation e^{-i theta * P}Qe^{i theta * P} on a qubitoperator Q and a Pauli operator P.

Parameters

- **`qop`** – the QubitOperator to be rotated
- **`pauli`** – a single Pauli operator - a QubitOperator with a single term, and a coefficient equal to 1.
- **`angle`** – the angle to be rotated by.

Returns

rotated_op - the rotated QubitOperator following the above formula.

Raises

- `TypeError` – qop must be a QubitOperator
- `TypeError` – pauli must be a Pauli Operator (QubitOperator with single term and coefficient equal to 1).

`openfermion.transforms.`**`symmetric_ordering`**(*operator*, *ignore_coeff=True*, *ignore_identity=True*)

Apply the symmetric ordering to a BosonOperator or QuadOperator.

The symmetric ordering is performed by applying McCoy's formula directly to polynomial terms of quadrature operators:

q^m p^n -> (1/ 2^n) sum_{r=0}^{n} Binomial(n, r) q^r p^m q^{n-r}

Note: in general, symmetric ordering is performed on a single term containing the tensor product of various operators. However, this function can also be applied to a sum of these terms, and the symmetric product is distributed over the summed terms.

In this case, Hermiticity cannot be guaranteed - as such, by default term coefficients and identity operators are ignored. However, this behavior can be modified via keyword arguments describe below if necessary.

Parameters

- **`operator`** – either a BosonOperator or QuadOperator.
- **`ignore_coeff`** (*`bool`*) – By default, the coefficients for each term are ignored; S(a q^m p^n) = S(q^m p^n), and the returned operator is always Hermitian. If set to False, then instead the coefficients are taken into account; S(q^m p^n) = a S(q^m p^n). In this case, if a is a complex coefficient, it is not guaranteed that the the returned operator will be Hermitian.
- **`ignore_identity`** (*`bool`*) – By default, identity terms are ignore; S(I) = 0. If set to False, then instead S(I) = I.

Returns *transformed_operator* – an operator of the same class as in the input.

Warning: The runtime of this method is exponential in the maximum locality of the original operator.

openfermion.transforms.**symmetry_conserving_bravyi_kitaev**(*fermion_hamiltonian*, *active_orbitals*, *active_fermions*)

Returns the qubit Hamiltonian for the fermionic Hamiltonian supplied, with two qubits removed using conservation of electron spin and number, as described in arXiv:1701.08213.

Parameters

- **fermion_hamiltonian** – A fermionic hamiltonian obtained using OpenFermion. An instance of the FermionOperator class.
- **active_orbitals** – Int type object. The number of active orbitals being considered for the system.
- **active_fermions** – Int type object. The number of active fermions being considered for the system (note, this is less than the number of electrons in a molecule if some orbitals have been assumed filled).

Returns

qubit_hamiltonian –

The qubit Hamiltonian corresponding to the supplied fermionic Hamiltonian, with two qubits removed using spin symmetries.

Warning: Reorders orbitals from the default even-odd ordering to all spin-up orbitals, then all spin-down orbitals.

Raises

- ValueError if fermion_hamiltonian isn't of the type
- FermionOperator, or active_orbitals isn't an integer,
- or active_fermions isn't an integer.

Notes: This function reorders the spin orbitals as all spin-up, then all spin-down. It uses the OpenFermion bravyi_kitaev_tree mapping, rather than the bravyi-kitaev mapping. Caution advised when using with a Fermi-Hubbard Hamiltonian; this technique correctly reduces the Hamiltonian only for the lowest energy even and odd fermion number states, not states with an arbitrary number of fermions.

openfermion.transforms.**verstraete_cirac_2d_square**(*operator*, *x_dimension*, *y_dimension*, *add_auxiliary_hamiltonian=True*, *snake=False*)

Apply the Verstraete-Cirac transform on a 2-d square lattice.

Note that this transformation adds one auxiliary fermionic mode for each mode already present, and hence it doubles the number of qubits needed to represent the system.

Currently only supports even values of x_dimension and only works for spinless models.

Parameters

- **operator** (`FermionOperator`) – The operator to transform.
- **x_dimension** (*int*) – The number of columns of the grid.
- **y_dimension** (*int*) – The number of rows of the grid.

- **snake** (*bool, optional*) – Indicates whether the fermions are already ordered according to the 2-d "snake" ordering. If False, we assume they are in "lexicographic" order by row and column index. Default is False.

Returns *transformed_operator* – A QubitOperator.

openfermion.transforms.**weight_one_binary_addressing_code**(*exponent*)

Weight-1 binary addressing code (arXiv:1712.07067). This highly non-linear code works for a number of modes that is an integer power of two. It encodes all possible vectors with Hamming weight 1, which corresponds to all states with total occupation one. The amount of qubits saved here is maximal: for a given argument 'exponent', we find n_modes = 2 ^ exponent, n_qubits = exponent.

Note: This code is highly non-linear and might produce a lot of terms.

Parameters **exponent** (*int*) – exponent for the number of modes n_modes = 2 ^ exponent

Returns (BinaryCode): the weight one binary addressing BinaryCode

openfermion.transforms.**weight_one_segment_code**()

Weight-1 segment code (arXiv:1712.07067). Outputs a 3-mode, 2-qubit code, which encodes all the vectors (states) with Hamming weight (occupation) 0 and 1. n_qubits = 2, n_modes = 3. A linear amount of qubits can be saved appending several instances of this code.

Note: This code is highly non-linear and might produce a lot of terms.

Returns (BinaryCode): weight one segment code

openfermion.transforms.**weight_two_segment_code**()

Weight-2 segment code (arXiv:1712.07067). Outputs a 5-mode, 4-qubit code, which encodes all the vectors (states) with Hamming weight (occupation) 2 and 1. n_qubits = 4, n_modes = 5. A linear amount of qubits can be saved appending several instances of this code.

Note: This code is highly non-linear and might produce a lot of terms.

Returns (BinaryCode): weight-2 segment code

openfermion.transforms.**weyl_polynomial_quantization**(*polynomial*)

Apply the Weyl quantization to a phase space polynomial.

The Weyl quantization is performed by applying McCoy's formula directly to a polynomial term of the form q^m p^n:

q^m p^n -> (1/ 2^n) sum_{r=0}^{n} Binomial(n, r) hat{q}^r hat{p}^m q^{n-r}

where q and p are phase space variables, and hat{q} and hat{p} are quadrature operators.

The input is provided in the form of a string, for example

```
weyl_polynomial_quantization('q0^2 p0^3 q1^3')
```

where 'q' or 'p' is the phase space quadrature variable, the integer directly following is the mode it is with respect to, and '^2' is the polynomial power.

Parameters **polynomial** (*str*) – polynomial function of q and p of the form 'qi^m pj^n . . . ' where i,j are the modes, and m, n the powers.

Returns *QuadOperator* – the Weyl quantization of the phase space function.

> **Warning:** The runtime of this method is exponential in the maximum locality of the original operator.

1.5 openfermion.utils

class openfermion.utils.**Davidson**(*linear_operator*, *linear_operator_diagonal*, *options=None*)
Davidson algorithm to get the n states with smallest eigenvalues.

__init__(*linear_operator*, *linear_operator_diagonal*, *options=None*)

Parameters

- **linear_operator** (*scipy.sparse.linalg.LinearOperator*) – The linear operator which defines a dot function when applying on a vector.
- **linear_operator_diagonal** (*numpy.ndarray*) – The linear operator's diagonal elements.
- **options** (DavidsonOptions) – Iteration options.

get_lowest_n(*n_lowest=1*, *initial_guess=None*, *max_iterations=None*)

Returns *n* smallest eigenvalues and corresponding eigenvectors for linear_operator.

Parameters

- **n** (*int*) – The number of states corresponding to the smallest eigenvalues and associated eigenvectors for the linear_operator.
- **initial_guess** (*numpy.ndarray[complex]*) – Initial guess of eigenvectors associated with the *n* smallest eigenvalues.
- **max_iterations** (*int*) – Max number of iterations when not converging.

Returns

success(bool) –

Indicates whether it converged, i.e. max elementwise error is smaller than eps.

eigen_values(numpy.ndarray[complex]): The smallest n eigenvalues. eigen_vectors(numpy.ndarray[complex]): The smallest n eigenvectors

corresponding with those eigen values.

class openfermion.utils.**DavidsonOptions**(*max_subspace=100*, *max_iterations=300*, *eps=1e-06*, *real_only=False*)
Davidson algorithm iteration options.

__init__(*max_subspace=100*, *max_iterations=300*, *eps=1e-06*, *real_only=False*)

Parameters

- **max_subspace** (*int*) – Max number of vectors in the auxiliary subspace.
- **max_iterations** (*int*) – Max number of iterations.
- **eps** (*float*) – The max error for eigen vector error's elements during iterations: linear_operator * v - v * lambda.

- **real_only** (*bool*) – Desired eigenvectors are real only or not. When one specifies the real_only to be true but it only has complex ones, no matter it converges or not, the returned vectors will be complex.

set_dimension(*dimension*)

Parameters **dimension** (*int*) – Dimension of the matrix, which sets a upper limit on the work space.

class openfermion.utils.**Grid**(*dimensions*, *length*, *scale*)

A multi-dimension grid of points with an assigned length scale.

This grid acts as a helper class for parallelpiped super cells. It tracks a mapping from indices to grid points and stores the associated reciprocal lattice with respect to the original real-space lattice. This enables calculations with non-trivial unit cells.

dimensions

Number of spatial dimensions the grid occupys

Type int

length

d-length tuple specifying number of points along each dimension.

Type tuple of ints

shifts

Integer shifts in position to center grid.

Type list of ints

scale

Vectors defining the super cell being simulated, vectors are stored as columns in the matrix.

Type ndarray

volume

Total volume of the supercell parallelpiped.

Type float

num_points

Total number of points in the grid.

Type int

reciprocal_scale

Vectors defining the reciprocal lattice. The vectors are stored as the columns in the matrix.

Type ndarray

__init__(*dimensions*, *length*, *scale*)

Parameters

- **dimensions** (*int*) – The number of dimensions the grid lives in.
- **length** (*int or tuple*) – The number of points along each grid axis that will be taken in both reciprocal and real space. If tuple, it is read for each dimension, otherwise assumed uniform.
- **scale** (*float or ndarray*) – The total length of each grid dimension. If a float is passed, the uniform cubic unit cell is assumed. For an ndarray, dimensions independent vectors of the correct dimension must be passed. We assume column vectors define the supercell vectors.

all_points_indices()

Returns *iterable[tuple[int]]* – The index-coordinate tuple of each point in the grid.

grid_indices(*qubit_id*, *spinless*)

This function is the inverse of orbital_id.

Parameters

- **qubit_id** (*int*) – The tensor factor to map to grid indices.
- **spinless** (*bool*) – Whether to use the spinless model or not.

Returns *grid_indices (numpy.ndarray[int])* – The location of the qubit on the grid.

index_to_momentum_ints(*index*)

Parameters **index** (*tuple*) – d-dimensional tuple specifying index in the grid

Returns Integer momentum vector

momentum_ints_to_index(*momentum_ints*)

Parameters **momentum_ints** (*tuple*) – d-dimensional tuple momentum integers

Returns d-dimensional tuples of indices

momentum_ints_to_value(*momentum_ints*, *periodic=True*)

Parameters

- **momentum_ints** (*tuple*) – d-dimensional tuple momentum integers
- **periodic** (*bool*) – Alias the momentum

Returns ndarray containing the momentum vector.

momentum_vector(*momentum_indices*, *periodic=True*)

Given grid point coordinate, return momentum vector with dimensions.

Parameters

- **momentum_indices** (*list*) – integers giving momentum indices. Allowed values are ints in [0, grid_length).
- **periodic** (*bool*) – Wrap the momentum indices according to periodicity
- **Returns** –

  momentum_vector: A numpy array giving the momentum vector with dimensions.

orbital_id(*grid_coordinates*, *spin=None*)

Return the tensor factor of a orbital with given coordinates and spin.

Parameters

- **grid_coordinates** – List or tuple of ints giving coordinates of grid element. Acceptable to provide an int(instead of tuple or list) for 1D case.
- **spin** (*bool*) – 0 means spin down and 1 means spin up. If None, assume spinless model.

Returns *tensor_factor (int)* – tensor factor associated with provided orbital label.

position_vector(*position_indices*)

Given grid point coordinate, return position vector with dimensions.

Parameters **position_indices** (*int | iterable[int]*) – List or tuple of integers giving grid point coordinate. Allowed values are ints in [0, grid_length).

Returns position_vector (numpy.ndarray[float])

`volume_scale`()

Returns *float* – The volume of a length-scale hypercube within the grid.

class openfermion.utils.**HubbardSquareLattice**(*x_dimension*, *y_dimension*, *n_dofs=1*, *spinless=False*, *periodic=True*)

A square lattice for a Hubbard model.

Valid edge types are:

- 'onsite'
- 'horizontal_neighbor'
- 'vertical_neighbor'
- 'neighbor': union of 'horizontal_neighbor' and 'vertical_neighbor'
- 'diagonal_neighbor'

`__init__`(*x_dimension*, *y_dimension*, *n_dofs=1*, *spinless=False*, *periodic=True*)

Parameters

- **x_dimension** (*int*) – The width of the grid.
- **y_dimension** (*int*) – The height of the grid.
- **n_dofs** (*int, optional*) – The number of degrees of freedom per site (and spin if applicable). Defaults is 1.
- **periodic** (*bool, optional*) – If True, add periodic boundary conditions. Default is True.
- **spinless** (*bool, optional*) – If True, return a spinless Fermi-Hubbard model. Default is False.

`delta_mag`(*X*, *Y*, *by_index=False*)
The distance between sites X and Y in each dimension.

`edge_types`
The types of edges that a term could correspond to.

Examples include 'onsite', 'neighbor', 'diagonal_neighbor', etc.

`n_dofs`
The number of degrees of freedom per site (and spin if applicable).

`n_horizontal_neighbor_pairs`(*ordered=True*)
Number of horizontally neighboring (unordered) pairs of sites.

`n_neighbor_pairs`(*ordered=True*)
Number of neighboring (unordered) pairs of sites.

`n_sites`
The number of sites in the lattice.

`n_vertical_neighbor_pairs`(*ordered=True*)
Number of vertically neighboring (unordered) pairs of sites.

`onsite_edge_types`
The edge types that connect sites to themselves.

`site_pairs_iter`(*edge_type*, *ordered=True*)
Iterable over pairs of sites corresponding to the given edge type.

spinless
Whether or not the fermion has spin (False if so).

to_site_index(*site*)
The index of a site.

class openfermion.utils.**LinearQubitOperator**(*qubit_operator*, *n_qubits=None*)
A LinearOperator implied from a QubitOperator.

The idea is that a single i_th qubit operator, O_i, is a 2-by-2 matrix, to be applied on a vector of length n_hilbert / 2^i, performs permutations and/ or adds an extra factor for its first half and the second half, e.g. a *Z* operator keeps the first half unchanged, while adds a factor of -1 to the second half, while an *I* keeps it both components unchanged.

Note that the vector length is n_hilbert / 2^i, therefore when one works on i monotonically (in increasing order), one keeps splitting the vector to the right size and then apply O_i on them independently.

Also note that operator O_i, is an *envelop operator* for all operators after it, i.e. {O_j | j > i}, which implies that starting with i = 0, one can split the vector, apply O_i, split the resulting vector (cached) again for the next operator.

__init__(*qubit_operator*, *n_qubits=None*)

Parameters

- **qubit_operator** (`QubitOperator`) – A qubit operator to be applied on vectors.
- **n_qubits** (*int*) – The total number of qubits

class openfermion.utils.**LinearQubitOperatorOptions**(*processes=10*, *pool=None*)
Options for LinearQubitOperator.

__init__(*processes=10*, *pool=None*)

Parameters

- **processes** (*int*) – Number of processors to use.
- **pool** (*multiprocessing.Pool*) – A pool of workers.

get_pool(*num=None*)
Gets a pool of workers to do some parallel work.

pool will be cached, which implies that one should be very clear how many processes one needs, as it's allocated at most once. Subsequent calls of get_pool() will reuse the cached pool.

Parameters **num** (*int*) – Number of workers one needs.

Returns *pool(multiprocessing.Pool)* – A pool of workers.

get_processes(*num*)
Number of real processes to use.

class openfermion.utils.**ParallelLinearQubitOperator**(*qubit_operator*, *n_qubits=None*, *options=None*)
A LinearOperator from a QubitOperator with multiple processors.

__init__(*qubit_operator*, *n_qubits=None*, *options=None*)

Parameters

- **qubit_operator** (`QubitOperator`) – A qubit operator to be applied on vectors.
- **n_qubits** (*int*) – The total number of qubits
- **options** (`LinearQubitOperatorOptions`) – Options for the LinearOperator.

class openfermion.utils.**QubitDavidson**(*qubit_operator*, *n_qubits=None*, *options=None*)
Davidson algorithm applied to a QubitOperator.

__init__(*qubit_operator*, *n_qubits=None*, *options=None*)

Parameters

- **qubit_operator** (QubitOperator) – A qubit operator which is a linear operator as well.
- **n_qubits** (*int*) – Number of qubits.
- **options** (DavidsonOptions) – Iteration options.

class openfermion.utils.**SparseDavidson**(*sparse_matrix*, *options=None*)
Davidson algorithm for a sparse matrix.

__init__(*sparse_matrix*, *options=None*)

Parameters

- **sparse_matrix** (*scipy.sparse.spmatrix*) – A sparse matrix in scipy.
- **options** (DavidsonOptions) – Iteration options.

class openfermion.utils.**Spin**
An enumeration.

class openfermion.utils.**SpinPairs**
The spin orbitals corresponding to a pair of spatial orbitals.

openfermion.utils.**amplitude_damping_channel**(*density_matrix*, *probability*, *target_qubit*, *transpose=False*)
Apply an amplitude damping channel

Applies an amplitude damping channel with a given probability to the target qubit in the density_matrix.

Parameters

- **density_matrix** (*numpy.ndarray*) – Density matrix of the system
- **probability** (*float*) – Probability error is applied p in [0, 1]
- **target_qubit** (*int*) – target for the channel error.
- **transpose** (*bool*) – Conjugate transpose channel operators, useful for acting on Hamiltonians in variational channel state models

Returns

new_density_matrix(numpy.ndarray) –

Density matrix with the channel applied.

openfermion.utils.**anticommutator**(*operator_a*, *operator_b*)
Compute the anticommutator of two operators.

Parameters **operator_b** (*operator_a,*) – Operators in anticommutator. Any operators are accepted so long as implicit addition and multiplication are supported; e.g. QubitOperators, FermionOperators, BosonOperators, or Scipy sparse matrices. 2D Numpy arrays are also supported.

Raises TypeError – operator_a and operator_b are not of the same type.

openfermion.utils.**bch_expand**(**ops*, ***kwargs*)
Compute log[e^{x_1} … e^{x_N}] using the BCH formula.

This implementation is explained in arXiv:1712.01348.

Parameters

- **ops** – A sequence of operators of the same type for which multiplication and addition are supported. For instance, QubitOperators, FermionOperators, or Scipy sparse matrices.
- **arguments** (*keyword*) –

  order(int): The max degree of monomial with respect to X and Y to truncate the BCH expansions. Defaults to 6

Returns The truncated BCH operator.

Raises

- `ValueError` – invalid order parameter.
- `TypeError` – operator types are not all the same.

`openfermion.utils.`**`boson_ladder_sparse`**(*n_modes*, *mode*, *ladder_type*, *trunc*)

Make a matrix representation of a singular bosonic ladder operator in the Fock space.

Since the bosonic operator lies in an infinite Fock space, a truncation value needs to be provide so that a sparse matrix of finite size can be returned.

Parameters

- **n_modes** (*int*) – Number of modes in the system Hilbert space.
- **mode** (*int*) – The mode the ladder operator targets.
- **ladder_type** (*int*) – This is a nonzero integer. 0 indicates a lowering operator, 1 a raising operator.
- **trunc** (*int*) – The size at which the Fock space should be truncated when returning the matrix representing the ladder operator.

Returns The corresponding trunc x trunc Scipy sparse matrix.

`openfermion.utils.`**`boson_operator_sparse`**(*operator*, *trunc*, *hbar=1.0*)

Initialize a Scipy sparse matrix in the Fock space from a bosonic operator.

Since the bosonic operators lie in an infinite Fock space, a truncation value needs to be provide so that a sparse matrix of finite size can be returned.

Parameters

- **operator** – One of either BosonOperator or QuadOperator.
- **trunc** (*int*) – The size at which the Fock space should be truncated when returning the matrix representing the ladder operator.
- **hbar** (*float*) – the value of hbar to use in the definition of the canonical commutation relation [q_i, p_j] = delta_{ij} i hbar. This only applies if calcualating the sparse representation of a quadrature operator.

Returns The corresponding Scipy sparse matrix of size [trunc, trunc].

`openfermion.utils.`**`chemist_ordered`**(*fermion_operator*)

Puts a two-body fermion operator in chemist ordering.

The normal ordering convention for chemists is different. Rather than ordering the two-body term as physicists do, as $a^\dagger a^\dagger a a$ the chemist ordering of the two-body term is $a^\dagger a a^\dagger a$

TODO: This routine can be made more efficient.

Parameters **fermion_operator** (`FermionOperator`) – a fermion operator guarenteed to have number conserving one- and two-body fermion terms only.

Returns

chemist_ordered_operator (FermionOperator) –

the input operator ordered in the chemistry convention.

Raises `OperatorSpecificationError` – Operator is not two-body number conserving.

`openfermion.utils.`**`commutator`**(*operator_a*, *operator_b*)

Compute the commutator of two operators.

Parameters **`operator_b`** (*`operator_a,`*) – Operators in commutator. Any operators are accepted so long as implicit subtraction and multiplication are supported; e.g. QubitOperators, FermionOperators, BosonOperators, or Scipy sparse matrices. 2D Numpy arrays are also supported.

Raises `TypeError` – operator_a and operator_b are not of the same type.

`openfermion.utils.`**`count_qubits`**(*operator*)

Compute the minimum number of qubits on which operator acts.

Parameters **`operator`** – FermionOperator, QubitOperator, DiagonalCoulombHamiltonian, or PolynomialTensor.

Returns *num_qubits (int)* – The minimum number of qubits on which operator acts.

Raises `TypeError` – Operator of invalid type.

`openfermion.utils.`**`dephasing_channel`**(*density_matrix*, *probability*, *target_qubit*, *transpose=False*)

Apply a dephasing channel

Applies an amplitude damping channel with a given probability to the target qubit in the density_matrix.

Parameters

- **`density_matrix`** (*`numpy.ndarray`*) – Density matrix of the system
- **`probability`** (*`float`*) – Probability error is applied p in [0, 1]
- **`target_qubit`** (*`int`*) – target for the channel error.
- **`transpose`** (*`bool`*) – Conjugate transpose channel operators, useful for acting on Hamiltonians in variational channel state models

Returns

new_density_matrix (numpy.ndarray) –

Density matrix with the channel applied.

`openfermion.utils.`**`depolarizing_channel`**(*density_matrix*, *probability*, *target_qubit*, *transpose=False*)

Apply a depolarizing channel

Applies an amplitude damping channel with a given probability to the target qubit in the density_matrix.

Parameters

- **`density_matrix`** (*`numpy.ndarray`*) – Density matrix of the system
- **`probability`** (*`float`*) – Probability error is applied p in [0, 1]
- **`target_qubit`** (*`int/str`*) – target for the channel error, if given special value "all", then a total depolarizing channel is applied.
- **`transpose`** (*`bool`*) – Dummy parameter to match signature of other channels but depolarizing channel is symmetric under conjugate transpose.

Returns

new_density_matrix (numpy.ndarray) –

Density matrix with the channel applied.

`openfermion.utils.`**`double_commutator`**(*op1*, *op2*, *op3*, *indices2=None*, *indices3=None*, *is_hopping_operator2=None*, *is_hopping_operator3=None*)

Return the double commutator [op1, [op2, op3]].

Parameters

- **`op2, op3`** (*`op1,`*) – operators for the commutator. All three operators must be of the same type.
- **`indices3`** (*`indices2,`*) – The indices op2 and op3 act on.
- **`is_hopping_operator2`** (*`bool`*) – Whether op2 is a hopping operator.
- **`is_hopping_operator3`** (*`bool`*) – Whether op3 is a hopping operator.

Returns The double commutator of the given operators.

`openfermion.utils.`**`eigenspectrum`**(*operator*, *n_qubits=None*)

Compute the eigenspectrum of an operator.

WARNING: This function has cubic runtime in dimension of Hilbert space operator, which might be exponential.

NOTE: This function does not currently support QuadOperator and BosonOperator.

Parameters

- **`operator`** – QubitOperator, InteractionOperator, FermionOperator, PolynomialTensor, or InteractionRDM.
- **`n_qubits`** (*`int`*) – number of qubits/modes in operator. if None, will be counted.

Returns *spectrum* – dense numpy array of floats giving eigenspectrum.

`openfermion.utils.`**`error_bound`**(*terms*, *tight=False*)

Numerically upper bound the error in the ground state energy for the second order Trotter-Suzuki expansion.

Parameters

- **`terms`** – a list of single-term QubitOperators in the Hamiltonian to be simulated.
- **`tight`** – whether to use the triangle inequality to give a loose upper bound on the error (default) or to calculate the norm of the error operator.

Returns A float upper bound on norm of error in the ground state energy.

Notes: follows Poulin et al.'s work in "The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry". In particular, Equation 16 is used for a loose upper bound, and the norm of Equation 9 is calculated for a tighter bound using the error operator from error_operator.

Possible extensions of this function would be to get the expectation value of the error operator with the Hartree-Fock state or CISD state, which can scalably bound the error in the ground state but much more accurately than the triangle inequality.

`openfermion.utils.`**`error_operator`**(*terms*, *series_order=2*)

Determine the difference between the exact generator of unitary evolution and the approximate generator given by Trotter-Suzuki to the given order.

Parameters

- **terms** – a list of QubitTerms in the Hamiltonian to be simulated.
- **series_order** – the order at which to compute the BCH expansion. Only the second order formula is currently implemented (corresponding to Equation 9 of the paper).

Returns

The difference between the true and effective generators of time evolution for a single Trotter step.

Notes: follows Equation 9 of Poulin et al.'s work in "The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry".

openfermion.utils.**expectation**(*operator*, *state*)

Compute the expectation value of an operator with a state.

Parameters

- **operator** (*scipy.sparse.spmatrix or scipy.sparse.linalg.LinearOperator*) – The operator whose expectation value is desired.
- **state** (*numpy.ndarray or scipy.sparse.spmatrix*) – A numpy array representing a pure state or a sparse matrix representing a density matrix. If *operator* is a LinearOperator, then this must be a numpy array.

Returns A complex number giving the expectation value.

Raises `ValueError` – Input state has invalid format.

openfermion.utils.**expectation_computational_basis_state**(*operator*, *computational_basis_state*)

Compute expectation value of operator with a state.

Parameters

- **operator** – Qubit or FermionOperator to evaluate expectation value of. If operator is a FermionOperator, it must be normal-ordered.
- **computational_basis_state** (*scipy.sparse vector / list*) – normalized computational basis state (if scipy.sparse vector), or list of occupied orbitals.

Returns A real float giving expectation value.

Raises `TypeError` – Incorrect operator or state type.

openfermion.utils.**fourier_transform**(*hamiltonian*, *grid*, *spinless*)

Apply Fourier transform to change hamiltonian in plane wave basis.

$$c_v^\dagger = \sqrt{1/N} \sum_m a_m^\dagger \exp(-ik_v r_m) c_v = \sqrt{1/N} \sum_m a_m \exp(ik_v r_m)$$

Parameters

- **hamiltonian** (`FermionOperator`) – The hamiltonian in plane wave basis.
- **grid** (`Grid`) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.

Returns *FermionOperator* – The fourier-transformed hamiltonian.

openfermion.utils.**freeze_orbitals**(*fermion_operator*, *occupied*, *unoccupied=None*, *prune=True*)

Fix some orbitals to be occupied and others unoccupied.

Removes all operators acting on the specified orbitals, and renumbers the remaining orbitals to eliminate unused indices. The sign of each term is modified according to the ladder uperator anti-commutation relations in order to preserve the expectation value of the operator.

Parameters

- **occupied** – A list containing the indices of the orbitals that are to be assumed to be occupied.
- **unoccupied** – A list containing the indices of the orbitals that are to be assumed to be unoccupied.

openfermion.utils.**gaussian_state_preparation_circuit**(*quadratic_hamiltonian*, *occupied_orbitals=None*, *spin_sector=None*)

Obtain the description of a circuit which prepares a fermionic Gaussian state.

Fermionic Gaussian states can be regarded as eigenstates of quadratic Hamiltonians. If the Hamiltonian conserves particle number, then these are just Slater determinants. See arXiv:1711.05395 for a detailed description of how this procedure works.

The circuit description is returned as a sequence of elementary operations; operations that can be performed in parallel are grouped together. Each elementary operation is either

- the string 'pht', indicating the particle-hole transformation on the last fermionic mode, which is the operator $\mathcal{B}$ such that

$$\mathcal{B} a_N \mathcal{B}^\dagger = a_N^\dagger, \tag{1.18}$$

$$\mathcal{B} a_j \mathcal{B}^\dagger = a_j, \quad j = 1, \ldots, N - 1, \tag{1.19}$$

  or

- a tuple (i, j, θ, φ), indicating the operation

$$\exp[i\varphi a_j^\dagger a_j] \exp[\theta(a_i^\dagger a_j - a_j^\dagger a_i)],$$

  a Givens rotation of modes i and j by angles θ and φ.

Parameters

- **quadratic_hamiltonian** (`QuadraticHamiltonian`) – The Hamiltonian whose eigenstate is desired.
- **occupied_orbitals** (`list`) – A list of integers representing the indices of the occupied orbitals in the desired Gaussian state. If this is None (the default), then it is assumed that the ground state is desired, i.e., the orbitals with negative energies are filled.
- **spin_sector** (`optional str`) – An optional integer specifying a spin sector to restrict to: 0 for spin-up and 1 for spin-down. If specified, the returned circuit acts on modes indexed by spatial indices (rather than spin indices). Should only be specified if the Hamiltonian includes a spin degree of freedom and spin-up modes do not interact with spin-down modes.

Returns

- *circuit_description (list[tuple])* – A list of operations describing the circuit. Each operation is a tuple of objects describing elementary operations that can be performed in parallel. Each elementary operation is either the string 'pht', indicating a particle-hole transformation on the last fermionic mode, or a tuple of the form (i, j, θ, φ), indicating a Givens rotation of modes i and j by angles θ and φ.

- *start_orbitals (list)* – The occupied orbitals to start with. This describes the initial state that the circuit should be applied to: it should be a Slater determinant (in the computational basis) with these orbitals filled.

`openfermion.utils.`**`generate_linear_qubit_operator`**(*qubit_operator*, *n_qubits=None*, *options=None*)

Generates a LinearOperator from a QubitOperator.

Parameters

- **`qubit_operator`** (`QubitOperator`) – A qubit operator to be applied on vectors.
- **`n_qubits`** (*`int`*) – The total number of qubits
- **`options`** (`LinearQubitOperatorOptions`) – Options for the ParallelLinearQubitOperator.

Returns *linear_operator(scipy.sparse.linalg.LinearOperator)* – A linear operator.

`openfermion.utils.`**`generate_parity_permutations`**(*seq*)

Generates the permutations and sign of a sequence by constructing a tree where the nth level contains all n-permutations of m (n < m) objects.

At the last level where n == m all permutations are generated. The sign is kept updated by determining where the next number is inserted into the current leaf's set of numbers.

Example

Constructing the permutations of the sequence [A, B, C] constructs the following tree:

[[(A, +1)], [(AB, +1), (BA, -1)], [(ABC, +1), (ACB, -1), (CAB, +1], [(BAC, -1), (BCA, +1), (CBA, -1)]]

Parameters **`seq`** – a sequence of a string to provide permutations

Returns a permutation list with the elements of the seq permuted and a sign associated with the permutation.

`openfermion.utils.`**`geometry_from_pubchem`**(*name: str*, *structure: str = None*)

Function to extract geometry using the molecule's name from the PubChem database. The 'structure' argument can be used to specify which structure info to use to extract the geometry. If structure=None, the geometry will be constructed based on 3D info, if available, otherwise on 2D (to keep backwards compatibility with the times when the argument 'structure' was not implemented).

Parameters

- **`name`** – a string giving the molecule's name as required by the PubChem database.
- **`structure`** – a string '2d' or '3d', to specify a specific structure information to be retrieved from pubchem. The default is None. Recommended value is '3d'.

Returns *geometry* – a list of tuples giving the coordinates of each atom with distances in Angstrom.

`openfermion.utils.`**`get_chemist_two_body_coefficients`**(*two_body_coefficients*, *spin_basis=True*)

Convert two-body operator coefficients to low rank tensor.

The input is a two-body fermionic Hamiltonian expressed as $\sum_{pqrs} h_{pqrs} a^\dagger_p a^\dagger_q a_r a_s$

We will convert this to the chemistry convention expressing it as $\sum_{pqrs} g_{pqrs} a^\dagger_p a_q a^\dagger_r a_s$ but without the spin degree of freedom.

In the process of performing this conversion, constants and one-body terms come out, which will be returned as well.

Parameters

- **two_body_coefficients** (*ndarray*) – an N x N x N x N numpy array giving the h_{pqrs} tensor.
- **spin_basis** (*bool*) – True if the two-body terms are passed in spin orbital basis. False if already in spatial orbital basis.

Returns

one_body_correction (ndarray) –

an N x N array of floats giving coefficients of the $a_p^\dagger a_q$ terms that come out.

chemist_two_body_coefficients (ndarray): an N x N x N x N numpy array giving the g_{pqrs} tensor in chemist notation.

Raises `TypeError` – Input must be two-body number conserving FermionOperator or InteractionOperator.

`openfermion.utils.`**`get_file_path`**(*file_name*, *data_directory*)

Compute file_path for the file that stores operator.

Parameters

- **file_name** – The name of the saved file.
- **data_directory** – Optional data directory to change from default data directory specified in config file.

Returns *file_path (string)* – File path.

Raises `OperatorUtilsError` – File name is not provided.

`openfermion.utils.`**`get_gap`**(*sparse_operator*, *initial_guess=None*)

Compute gap between lowest eigenvalue and first excited state.

Parameters

- **sparse_operator** (*LinearOperator*) – Operator to find the ground state of.
- **initial_guess** (*ndarray*) – Initial guess for eigenspace. A good guess dramatically reduces the cost required to converge.

Returns: A real float giving eigenvalue gap.

`openfermion.utils.`**`get_ground_state`**(*sparse_operator*, *initial_guess=None*)

Compute lowest eigenvalue and eigenstate.

Parameters

- **sparse_operator** (*LinearOperator*) – Operator to find the ground state of.
- **initial_guess** (*ndarray*) – Initial guess for ground state. A good guess dramatically reduces the cost required to converge.

Returns

- *eigenvalue* – The lowest eigenvalue, a float.
- *eigenstate* – The lowest eigenstate in scipy.sparse csc format.

openfermion.utils.**get_linear_qubit_operator_diagonal**(*qubit_operator*, *n_qubits=None*)

Return a linear operator's diagonal elements.

The main motivation is to use it for Davidson's algorithm, to find out the lowest n eigenvalues and associated eigenvectors.

Qubit terms with X or Y operators will contribute nothing to the diagonal elements, while I or Z will contribute a factor of 1 or -1 together with the coefficient.

Parameters **qubit_operator** (QubitOperator) – A qubit operator.

Returns

linear_operator_diagonal(numpy.ndarray) –

The diagonal elements for LinearQubitOperator(qubit_operator).

openfermion.utils.**group_into_tensor_product_basis_sets**(*operator*, *seed=None*)

Split an operator (instance of QubitOperator) into *sub-operator* QubitOperators, where each sub-operator has terms that are diagonal in the same tensor product basis.

Each *sub-operator* can be measured using the same qubit post-rotations in expectation estimation. Grouping into these tensor product basis sets has been found to improve the efficiency of expectation estimation significantly for some Hamiltonians in the context of VQE (see section V(A) in the supplementary material of https://arxiv.org/pdf/1704.05018v2.pdf). The more general problem of grouping operators into commutitative groups is discussed in section IV (B2) of https://arxiv.org/pdf/1509.04279v1.pdf. The original input operator is the union of all output sub-operators, and all sub-operators are disjoint (do not share any terms).

Parameters

- **operator** (QubitOperator) – the operator that will be split into sub-operators (tensor product basis sets).
- **seed** (*int*) – default None. Random seed used to initialize the numpy.RandomState pseudo-random number generator.

Returns

sub_operators (dict) –

a dictionary where each key defines a tensor product basis, and each corresponding value is a QubitOperator with terms that are all diagonal in that basis. **key** (tuple of tuples): Each key is a term, which defines

a tensor product basis. A term is a product of individual factors; each factor is represented by a tuple of the form (*index*, *action*), and these tuples are collected into a larger tuple which represents the term as the product of its factors. *action* is from the set {'X', 'Y', 'Z'} and *index* is a non-negative integer corresponding to the index of a qubit.

value (QubitOperator): A QubitOperator with terms that are diagonal in the basis defined by the key it is stored in.

Raises TypeError – Operator of invalid type.

openfermion.utils.**haar_random_vector**(*n*, *seed=None*)

Generate an n dimensional Haar randomd vector.

openfermion.utils.**hartree_fock_state_jellium**(*grid*, *n_electrons*, *spinless=True*, *plane_wave=False*)

Give the Hartree-Fock state of jellium.

Parameters

- **grid** (Grid) – The discretization to use.
- **n_electrons** (*int*) – Number of electrons in the system.
- **spinless** (*bool*) – Whether to use the spinless model or not.
- **plane_wave** (*bool*) – Whether to return the Hartree-Fock state in the plane wave (True) or dual basis (False).

Notes

The jellium model is built up by filling the lowest-energy single-particle states in the plane-wave Hamiltonian until n_electrons states are filled.

`openfermion.utils.`**`hermitian_conjugated`**(*operator*)
Return Hermitian conjugate of operator.

`openfermion.utils.`**`inline_sum`**(*summands*, *seed*)
Computes a sum, using the __iadd__ operator. :param seed: The starting total. The zero value. :type seed: T :param summands: Values to add (with +=) into the total. :type summands: iterable[T]

Returns *T* – The result of adding all the factors into the zero value.

`openfermion.utils.`**`inner_product`**(*state_1*, *state_2*)
Compute inner product of two states.

`openfermion.utils.`**`inverse_fourier_transform`**(*hamiltonian*, *grid*, *spinless*)
Apply inverse Fourier transform to change hamiltonian in plane wave dual basis.

$$a^\dagger_v = \sqrt{1/N} \sum_m c^\dagger_m \exp(i k_v r_m) a_v = \sqrt{1/N} \sum_m c_m \exp(-i k_v r_m)$$

Parameters

- **hamiltonian** (FermionOperator) – The hamiltonian in plane wave dual basis.
- **grid** (Grid) – The discretization to use.
- **spinless** (*bool*) – Whether to use the spinless model or not.

Returns *FermionOperator* – The inverse-fourier-transformed hamiltonian.

`openfermion.utils.`**`is_hermitian`**(*operator*)
Test if operator is Hermitian.

`openfermion.utils.`**`is_identity`**(*operator*)
Check whether QubitOperator of FermionOperator is identity.

Parameters **operator** – QubitOperator, FermionOperator, BosonOperator, or QuadOperator.

Raises `TypeError` – Operator of invalid type.

`openfermion.utils.`**`jordan_wigner_sparse`**(*fermion_operator*, *n_qubits=None*)
Initialize a Scipy sparse matrix from a FermionOperator.

Operators are mapped as follows: a_j^dagger -> Z_0 .. Z_{j-1} (X_j - iY_j) / 2 a_j -> Z_0 .. Z_{j-1} (X_j + iY_j) / 2

Parameters

- **fermion_operator** (FermionOperator) – instance of the FermionOperator class.
- **n_qubits** (*int*) – Number of qubits.

Returns The corresponding Scipy sparse matrix.

openfermion.utils.**jw_configuration_state**(*occupied_orbitals*, *n_qubits*)
Function to produce a basis state in the occupation number basis.

Parameters

- **occupied_orbitals** (*list*) – A list of integers representing the indices of the occupied orbitals in the desired basis state
- **n_qubits** (*int*) – The total number of qubits

Returns *basis_vector(sparse)* – The basis state as a sparse matrix

openfermion.utils.**jw_get_gaussian_state**(*quadratic_hamiltonian*, *occupied_orbitals=None*)
Compute an eigenvalue and eigenstate of a quadratic Hamiltonian.

Eigenstates of a quadratic Hamiltonian are also known as fermionic Gaussian states.

Parameters

- **quadratic_hamiltonian** (QuadraticHamiltonian) – The Hamiltonian whose eigenstate is desired.
- **occupied_orbitals** (*list*) – A list of integers representing the indices of the occupied orbitals in the desired Gaussian state. If this is None (the default), then it is assumed that the ground state is desired, i.e., the orbitals with negative energies are filled.

Returns

- *energy (float)* – The eigenvalue.
- *state (sparse)* – The eigenstate in scipy.sparse csc format.

openfermion.utils.**jw_get_ground_state_at_particle_number**(*sparse_operator*, *particle_number*)
Compute ground energy and state at a specified particle number.

Assumes the Jordan-Wigner transform. The input operator should be Hermitian and particle-number-conserving.

Parameters

- **sparse_operator** (*sparse*) – A Jordan-Wigner encoded sparse matrix.
- **particle_number** (*int*) – The particle number at which to compute the ground energy and states

Returns

ground_energy(float) –

The lowest eigenvalue of sparse_operator within the eigenspace of the number operator corresponding to particle_number.

ground_state(ndarray): The ground state at the particle number

openfermion.utils.**jw_hartree_fock_state**(*n_electrons*, *n_orbitals*)
Function to produce Hartree-Fock state in JW representation.

openfermion.utils.**jw_number_restrict_operator**(*operator*, *n_electrons*, *n_qubits=None*)
Restrict a Jordan-Wigner encoded operator to a given particle number

Parameters

- **sparse_operator** (*ndarray or sparse*) – Numpy operator acting on the space of n_qubits.
- **n_electrons** (*int*) – Number of particles to restrict the operator to

- **n_qubits** (*int*) – Number of qubits defining the total state

Returns

new_operator(ndarray or sparse) –

Numpy operator restricted to acting on states with the same particle number.

openfermion.utils.**jw_number_restrict_state**(*state*, *n_electrons*, *n_qubits=None*)

Restrict a Jordan-Wigner encoded state to a given particle number

Parameters

- **state** (*ndarray or sparse*) – Numpy vector in the space of n_qubits.
- **n_electrons** (*int*) – Number of particles to restrict the state to
- **n_qubits** (*int*) – Number of qubits defining the total state

Returns

new_operator(ndarray or sparse) –

Numpy vector restricted to states with the same particle number. May not be normalized.

openfermion.utils.**jw_slater_determinant**(*slater_determinant_matrix*)

Obtain a Slater determinant.

The input is an $N_f \times N$ matrix Q with orthonormal rows. Such a matrix describes the Slater determinant

$$b_1^\dagger \cdots b_{N_f}^\dagger |\text{vac}\rangle,$$

where

$$b_j^\dagger = \sum_{k=1}^{N} Q_{jk} a_k^\dagger.$$

Parameters **slater_determinant_matrix** – The matrix Q which describes the Slater determinant to be prepared.

Returns The Slater determinant as a sparse matrix.

openfermion.utils.**jw_sz_restrict_operator**(*operator*, *sz_value*, *n_electrons=None*, *n_qubits=None*, *up_index=<function up_index>*, *down_index=<function down_index>*)

Restrict a Jordan-Wigner encoded operator to a given Sz value

Parameters

- **operator** (*ndarray or sparse*) – Numpy operator acting on the space of n_qubits.
- **sz_value** (*float*) – Desired Sz value. Should be an integer or half-integer.
- **n_electrons** (*int, optional*) – Number of particles to restrict the operator to, if such a restriction is desired.
- **n_qubits** (*int, optional*) – Number of qubits defining the total state
- **up_index** (*Callable, optional*) – Function that maps a spatial index to the index of the corresponding up site
- **down_index** (*Callable, optional*) – Function that maps a spatial index to the index of the corresponding down site

Returns

new_operator(ndarray or sparse) –

Numpy operator restricted to acting on states with the desired Sz value.

`openfermion.utils.`**`jw_sz_restrict_state`**(*state*, *sz_value*, *n_electrons=None*, *n_qubits=None*, *up_index=<function up_index>*, *down_index=<function down_index>*)

Restrict a Jordan-Wigner encoded state to a given Sz value

Parameters

- **`state`** (*`ndarray or sparse`*) – Numpy vector in the space of n_qubits.
- **`sz_value`** (*`float`*) – Desired Sz value. Should be an integer or half-integer.
- **`n_electrons`** (*`int, optional`*) – Number of particles to restrict the operator to, if such a restriction is desired.
- **`n_qubits`** (*`int, optional`*) – Number of qubits defining the total state
- **`up_index`** (*`Callable, optional`*) – Function that maps a spatial index to the index of the corresponding up site
- **`down_index`** (*`Callable, optional`*) – Function that maps a spatial index to the index of the corresponding down site

Returns

new_operator(ndarray or sparse) –

Numpy vector restricted to states with the desired Sz value. May not be normalized.

`openfermion.utils.`**`lambda_norm`**(*diagonal_operator*)

Computes the lambda norm relevant to LCU algorithms.

Parameters **`diagonal_operator`** – instance of DiagonalCoulombHamiltonian.

Returns *lambda_norm* – A float giving the lambda norm.

`openfermion.utils.`**`load_operator`**(*file_name=None*, *data_directory=None*, *plain_text=False*)

Load FermionOperator or QubitOperator from file.

Parameters

- **`file_name`** – The name of the saved file.
- **`data_directory`** – Optional data directory to change from default data directory specified in config file.
- **`plain_text`** – Whether the input file is plain text

Returns

operator –

The stored FermionOperator, BosonOperator, QuadOperator, or QubitOperator

Raises `TypeError` – Operator of invalid type.

`openfermion.utils.`**`low_depth_second_order_trotter_error_bound`**(*terms*, *indices=None*, *is_hopping_operator=None*, *jellium_only=False*, *verbose=False*)

Numerically upper bound the error in the ground state energy for the second-order Trotter-Suzuki expansion.

Parameters

- **terms** – a list of single-term FermionOperators in the Hamiltonian to be simulated.
- **indices** – a set of indices the terms act on in the same order as terms.
- **is_hopping_operator** – a list of whether each term is a hopping operator.
- **jellium_only** – Whether the terms are from the jellium Hamiltonian only, rather than the full dual basis Hamiltonian (i.e. whether c_i = c for all number operators i^ i, or whether they depend on i as is possible in the general case).
- **verbose** – Whether to print percentage progress.

Returns A float upper bound on norm of error in the ground state energy.

Notes

Follows Equation 9 of Poulin et al.'s work in "The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry" to calculate the error operator, for the "stagger"-based Trotter step for detailed in Kivlichan et al., "Quantum Simulation of Electronic Structure with Linear Depth and Connectivity", arxiv:1711.04789.

openfermion.utils.**low_depth_second_order_trotter_error_operator**(*terms*, *indices=None*, *is_hopping_operator=None*, *jellium_only=False*, *verbose=False*)

Determine the difference between the exact generator of unitary evolution and the approximate generator given by the second-order Trotter-Suzuki expansion.

Parameters

- **terms** – a list of FermionOperators in the Hamiltonian in the order in which they will be simulated.
- **indices** – a set of indices the terms act on in the same order as terms.
- **is_hopping_operator** – a list of whether each term is a hopping operator.
- **jellium_only** – Whether the terms are from the jellium Hamiltonian only, rather than the full dual basis Hamiltonian (i.e. whether c_i = c for all number operators i^ i, or whether they depend on i as is possible in the general case).
- **verbose** – Whether to print percentage progress.

Returns

The difference between the true and effective generators of time evolution for a single Trotter step.

Notes: follows Equation 9 of Poulin et al.'s work in "The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry", applied to the "stagger"-based Trotter step for detailed in Kivlichan et al., "Quantum Simulation of Electronic Structure with Linear Depth and Connectivity", arxiv:1711.04789.

openfermion.utils.**low_rank_two_body_decomposition**(*two_body_coefficients*, *truncation_threshold=1e-08*, *final_rank=None*, *spin_basis=True*)

Convert two-body operator into sum of squared one-body operators.

As in arXiv:1808.02625, this function decomposes $\sum_{pqrs} h_{pqrs} a^\dagger_p a^\dagger_q a_r a_s$ as $\sum_l \lambda_l (\sum_{pq} g_{lpq} a^\dagger_p a_q)^2$ l is truncated to take max value L so that $\sum_{l=0}^{L-1} (\sum_{pq} |g_{lpq}|)^2 |\lambda_l| < x$

Parameters

- **two_body_coefficients** (*ndarray*) – an N x N x N x N numpy array giving the h_{pqrs} tensor. This tensor must be 8-fold symmetric (real integrals).
- **truncation_threshold** (*optional Float*) – the value of x, above.
- **final_rank** (*optional int*) – if provided, this specifies the value of L at which to truncate. This overrides truncation_threshold.
- **spin_basis** (*bool*) – True if the two-body terms are passed in spin orbital basis. False if already in spatial orbital basis.

Returns

eigenvalues (ndarray of floats) –

length L array giving the λ_l.

one_body_squares (ndarray of floats): L x N x N array of floats corresponding to the value of g_{pql}.

one_body_correction (ndarray): One-body correction terms that result from reordering to chemist ordering, in spin-orbital basis.

truncation_value (float): after truncation, this is the value $\sum_{l=0}^{L-1} (\sum_{pq} |g_{lpq}|)^2 |\lambda_l| < x$

Raises TypeError – Invalid two-body coefficient tensor specification.

openfermion.utils.**majorana_operator**(*term=None*, *coefficient=1.0*)

Initialize a Majorana operator.

Parameters

- **term** (*tuple or string*) – The first element of the tuple indicates the mode on which the Majorana operator acts, starting from zero. The second element of the tuple is an integer, either 0 or 1, indicating which type of Majorana operator it is:

  Type 0: $a^\dagger_p + a_p$

  Type 1: $i(a^\dagger_p - a_p)$

  where the $a^\dagger_p$ and a_p are the usual fermionic ladder operators. Alternatively, one can provide a string such as 'c2', which is a Type 0 operator on mode 2, or 'd3', which is a Type 1 operator on mode 3. Default will result in the zero operator.
- **coefficient** (*complex or float, optional*) – The coefficient of the term. Default value is 1.0.

Returns FermionOperator

openfermion.utils.**map_one_hole_dm_to_one_pdm**(*oqdm*)

Convert a 1-hole-RDM to a 1-RDM

Parameters **oqdm** (*numpy.ndarray*) – The 1-hole-RDM as a 2-index tensor. Indices follow the internal convention of oqdm[p, q] == $a_p a^\dagger_q$.

Returns *oqdm (numpy.ndarray)* – the 1-hole-RDM transformed from a 1-RDM.

openfermion.utils.**map_one_pdm_to_one_hole_dm**(*opdm*)

Convert a 1-RDM to a 1-hole-RDM

Parameters **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of opdm[p, q] == $a_p^\dagger a_q$.

Returns *oqdm (numpy.ndarray)* – the 1-hole-RDM transformed from a 1-RDM.

openfermion.utils.**map_particle_hole_dm_to_one_pdm**(*phdm*, *num_particles*, *num_basis_functions*)

Map the particle-hole-RDM to the 1-RDM

Parameters

- **phdm** (*numpy.ndarray*) – The 2-particle-hole-RDM as a 4-index tensor. Indices follow the internal convention of phdm[p, q, r, s] == $a_p^\dagger a_q a_r^\dagger a_s$.
- **num_particles** – number of particles in the system.
- **num_basis_functions** – number of spin-orbitals (usually the number of qubits)

Returns *opdm (numpy.ndarray)* – the 1-RDM transformed from a 1-RDM.

openfermion.utils.**map_particle_hole_dm_to_two_pdm**(*phdm*, *opdm*)

Map the 2-RDM to the particle-hole-RDM

Parameters

- **phdm** (*numpy.ndarray*) – The 2-particle-hole-RDM as a 4-index tensor. Indices follow the internal convention of phdm[p, q, r, s] == $a_p^\dagger a_q a_r^\dagger a_s$.
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of opdm[p, q] == $a_p^\dagger a_q$.

Returns *tpdm (numpy.ndarray)* – The 2-RDM matrix.

openfermion.utils.**map_two_hole_dm_to_one_hole_dm**(*tqdm*, *hole_number*)

Map from 2-hole-RDM to 1-hole-RDM

Parameters

- **tqdm** (*numpy.ndarray*) – The 2-hole-RDM as a 4-index tensor. Indices follow the internal convention of tqdm[p, q, r, s] == $a_p a_q a_r^\dagger a_s^\dagger$.
- **hole_number** (*float*) – Number of holes in the system. For chemical systems this is usually the number of spin orbitals minus the number of electrons.

Returns *oqdm (numpy.ndarray)* – The 1-hole-RDM contracted from the tqdm.

openfermion.utils.**map_two_hole_dm_to_two_pdm**(*tqdm*, *opdm*)

Map from the 2-hole-RDM to the 2-RDM

Parameters

- **tqdm** (*numpy.ndarray*) – The 2-hole-RDM as a 4-index tensor. Indices follow the internal convention of tqdm[p, q, r, s] == $a_p a_q a_r^\dagger a_s^\dagger$.
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of opdm[p, q] == $a_p^\dagger a_q$.

Returns *tpdm (numpy.ndarray)* – The 2-RDM matrix.

openfermion.utils.**map_two_pdm_to_one_pdm**(*tpdm*, *particle_number*)

Contract a 2-RDM to a 1-RDM

Parameters

- **tpdm** (*numpy.ndarray*) – The 2-RDM as a 4-index tensor. Indices follow the internal convention of tpdm[p, q, r, s] == $a_p^\dagger a_q^\dagger a_r a_s$.
- **particle_number** (*float*) – number of particles in the system

Returns *opdm (numpy.ndarray)* – The 1-RDM contracted from the tpdm.

openfermion.utils.**map_two_pdm_to_particle_hole_dm**(*tpdm*, *opdm*)
Map the 2-RDM to the particle-hole-RDM

Parameters

- **tpdm** (*numpy.ndarray*) – The 2-RDM as a 4-index tensor. Indices follow the internal convention of tpdm[p, q, r, s] == $a_p^\dagger a_q^\dagger a_r a_s$.
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of opdm[p, q] == $a_p^\dagger a_q$.

Returns *phdm (numpy.ndarray)* – The particle-hole matrix.

openfermion.utils.**map_two_pdm_to_two_hole_dm**(*tpdm*, *opdm*)
Map from the 2-RDM to the 2-hole-RDM

Parameters

- **tpdm** (*numpy.ndarray*) – The 2-RDM as a 4-index tensor. Indices follow the internal convention of tpdm[p, q, r, s] == $a_p^\dagger a_q^\dagger a_r a_s$.
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of opdm[p, q] == $a_p^\dagger a_q$.

Returns *tqdm (numpy.ndarray)* – The 2-hole matrix.

openfermion.utils.**module_importable**(*module*)
Without importing it, returns whether python module is importable.

Parameters **module** (*string*) – Name of module.

Returns bool

openfermion.utils.**normal_ordered**(*operator*, *hbar=1.0*)
Compute and return the normal ordered form of a FermionOperator, BosonOperator, QuadOperator, or InteractionOperator.

Due to the canonical commutation/anticommutation relations satisfied by these operators, there are multiple forms that the same operator can take. Here, we define the normal ordered form of each operator, providing a distinct representation for distinct operators.

In our convention, normal ordering implies terms are ordered from highest tensor factor (on left) to lowest (on right). In addition:

- FermionOperators: a^dagger comes before a
- BosonOperators: b^dagger comes before b
- QuadOperators: q operators come before p operators,

Parameters

- **operator** – an instance of the FermionOperator, BosonOperator, QuadOperator, or InteractionOperator classes.
- **hbar** (*float*) – the value of hbar used in the definition of the commutator [q_i, p_j] = i hbar delta_ij. By default hbar=1. This argument only applies when normal ordering QuadOperators.

`openfermion.utils.`**`number_operator`**(*n_modes*, *mode=None*, *coefficient=1.0*, *parity=-1*)
Return a fermionic or bosonic number operator.

Parameters

- **n_modes** (*int*) – The number of modes in the system.
- **mode** (*int, optional*) – The mode on which to return the number operator. If None, return total number operator on all sites.
- **coefficient** (*float*) – The coefficient of the term.
- **parity** (*int*) – Returns the fermionic number operator if parity=-1 (default), and returns the bosonic number operator if parity=1.

Returns operator (BosonOperator or FermionOperator)

`openfermion.utils.`**`pauli_exp_to_qasm`**(*qubit_operator_list*, *evolution_time=1.0*, *qubit_list=None*, *ancilla=None*)
Exponentiate a list of QubitOperators to a QASM string generator.

Exponentiates a list of QubitOperators, and yields string generators in QASM format using the formula: exp(-1.0j * evolution_time * op).

Parameters

- **qubit_operator_list** (*list of QubitOperators*) – list of single Pauli-term QubitOperators to be exponentiated
- **evolution_time** (*float*) – evolution time of the operators in the list
- **qubit_list** – (list/tuple or None)Specifies the labels for the qubits to be output in qasm. If a list/tuple, must have length greater than or equal to the number of qubits in the QubitOperator. Entries in the list must be castable to string. If None, qubits are labeled by index (i.e. an integer).
- **ancilla** (*string or None*) – if any, an ancilla qubit to perform the rotation conditional on (for quantum phase estimation)

Yields string

`openfermion.utils.`**`prepare_one_body_squared_evolution`**(*one_body_matrix*, *spin_basis=True*)
Get Givens angles and DiagonalHamiltonian to simulate squared one-body.

The goal here will be to prepare to simulate evolution under $(\sum_{pq} h_{pq} a^\dagger_p a_q)^2$ by decomposing as $R e^{-i \sum_{pq} V_{pq} n_p n_q} R^\dagger' where : math :$ is a basis transformation matrix.

TODO: Add option for truncation based on one-body eigenvalues.

Parameters

- **one_body_matrix** (*ndarray of floats*) – an N by N array storing the coefficients of a one-body operator to be squared. For instance, in the above the elements of this matrix are h_{pq}.
- **spin_basis** (*bool*) – Whether the matrix is passed in the spin orbital basis.

Returns

density_density_matrix(ndarray of floats) an N by N array storing the diagonal two-body coefficeints V_{pq} above.

basis_transformation_matrix (ndarray of floats) an N by N array storing the values of the basis transformation.

Raises `ValueError` – one_body_matrix is not Hermitian.

`openfermion.utils.`**`preprocess_lcu_coefficients_for_reversible_sampling`**(*lcu_coefficients*, *epsilon*)

Prepares data used to perform efficient reversible roulette selection.

Treats the coefficients of unitaries in the linear combination of unitaries decomposition of the Hamiltonian as probabilities in order to decompose them into a list of alternate and keep numerators allowing for an efficient preparation method of a state where the computational basis state :math. |*k*> has an amplitude proportional to the coefficient.

It is guaranteed that following the following sampling process will sample each index k with a probability within epsilon of lcu_coefficients[k] / sum(lcu_coefficients) and also, 1. Uniformly sample an index i from [0, len(lcu_coefficients) - 1]. 2. With probability keep_numers[i] / by keep_denom, return i. 3. Otherwise return alternates[i].

Parameters

- **`lcu_coefficients`** – A list of non-negative floats, with the i'th float corresponding to the i'th coefficient of an LCU decomposition of the Hamiltonian (in an ordering determined by the caller).
- **`epsilon`** – Absolute error tolerance.

Returns

alternates (list[int]) –

A python list of ints indicating alternative indices that may be switched to after generating a uniform index. The int at offset k is the alternate to use when the initial index is k.

keep_numers (list[int]): A python list of ints indicating the numerators of the probability that the alternative index should be used instead of the initial index.

sub_bit_precision (int): A python int indicating the exponent of the denominator to divide the items in keep_numers by in order to get a probability. The actual denominator is 2**sub_bit_precision.

`openfermion.utils.`**`prune_unused_indices`**(*symbolic_operator*)

Remove indices that do not appear in any terms.

Indices will be renumbered such that if an index i does not appear in any terms, then the next largest index that appears in at least one term will be renumbered to i.

`openfermion.utils.`**`qubit_operator_sparse`**(*qubit_operator*, *n_qubits=None*)

Initialize a Scipy sparse matrix from a QubitOperator.

Parameters

- **`qubit_operator`** (`QubitOperator`) – instance of the QubitOperator class.
- **`n_qubits`** (*`int`*) – Number of qubits.

Returns The corresponding Scipy sparse matrix.

`openfermion.utils.`**`random_antisymmetric_matrix`**(*n*, *real=False*, *seed=None*)

Generate a random n x n antisymmetric matrix.

`openfermion.utils.`**`random_diagonal_coulomb_hamiltonian`**(*n_qubits*, *real=False*, *seed=None*)

Generate a random instance of DiagonalCoulombHamiltonian.

Parameters

- **n_qubits** – The number of qubits
- **real** – Whether to use only real numbers in the one-body term

openfermion.utils.**random_hermitian_matrix**(*n*, *real=False*, *seed=None*)
Generate a random n x n Hermitian matrix.

openfermion.utils.**random_interaction_operator**(*n_orbitals*, *expand_spin=False*, *real=True*, *seed=None*)
Generate a random instance of InteractionOperator.

Parameters

- **n_orbitals** – The number of orbitals.
- **expand_spin** – Whether to expand each orbital symmetrically into two spin orbitals. Note that if this option is set to True, then the total number of orbitals will be doubled.
- **real** – Whether to use only real numbers.
- **seed** – A random number generator seed.

openfermion.utils.**random_quadratic_hamiltonian**(*n_orbitals*, *conserves_particle_number=False*, *real=False*, *expand_spin=False*, *seed=None*)
Generate a random instance of QuadraticHamiltonian.

Parameters

- **n_orbitals** (*int*) – the number of orbitals
- **conserves_particle_number** (*bool*) – whether the returned Hamiltonian should conserve particle number
- **real** (*bool*) – whether to use only real numbers
- **expand_spin** – Whether to expand each orbital symmetrically into two spin orbitals. Note that if this option is set to True, then the total number of orbitals will be doubled.

Returns QuadraticHamiltonian

openfermion.utils.**random_unitary_matrix**(*n*, *real=False*, *seed=None*)
Obtain a random n x n unitary matrix.

openfermion.utils.**reduce_number_of_terms**(*operator*, *stabilizers*, *maintain_length=False*, *output_fixed_positions=False*, *manual_input=False*, *fixed_positions=None*)
Reduce the number of Pauli strings of operator using stabilizers.

This function reduces the number of terms in a string by merging terms that are identical by the multiplication of stabilizers. The resulting Pauli strings maintain their length, unless specified otherwise. In the latter case, a list of indices can be passed to manually indicate the qubits to be fixed.

It is possible to reduce the number of terms in a Hamiltonian by merging Pauli strings H_1, H_2 that are related by a stabilizer S such that $H_1 = H_2 \cdot S$. Given a stabilizer generator $\pm X \otimes p$ this algorithm fixes the first qubit, such that every Pauli string in the Hamiltonian acts with either Z or the identity on it. Where necessary, this is achieved by multiplications with $\pm X \otimes p$: a string $Y \otimes h$, for instance, is turned into $Z \otimes (\mp ih \cdot p)$. Qubits on which a generator acts as Y (Z) are constrained to be acted on by the Hamiltonian as Z (X) or the identity. Fixing a different qubit for every stabilizer generator eliminates all redundant strings. The fixed representations are in the end re-expressed as the shortest of the original strings, H_1 or H_2.

Parameters

- **operator** (QubitOperator) – Operator of which the number of terms will be reduced.
- **stabilizers** (QubitOperator) – Stabilizer generators used for the reduction. Can also be passed as a list of QubitOperator.
- **maintain_length** (*Boolean*) – Option deciding whether the fixed Pauli strings are re-expressed in their original form. Set to False by default.
- **output_fixed_positions** (*Boolean*) – Option deciding whether to return the list of fixed qubit positions. Set to False by default.
- **manual_input** (*Boolean*) – Option to pass the list of fixed qubits positions manually. Set to False by default.
- **fixed_positions** (*list*) – (optional) List of fixed qubit positions. Passing a list is only effective if manual_input is True.

Returns

reduced_operator (QubitOperator) –

Operator with reduced number of terms.

fixed_positions (list): (optional) Fixed qubits.

Raises

- TypeError – Input terms must be QubitOperator.
- TypeError – Input stabilizers must be QubitOperator or list.
- StabilizerError – Trivial stabilizer (identity).
- StabilizerError – Stabilizer with complex coefficient.
- TypeError – List of fixed qubits required if manual input True.
- StabilizerError – The number of stabilizers must be equal to the number of qubits manually fixed.
- StabilizerError – All qubit positions must be different.

openfermion.utils.**reorder**(*operator*, *order_function*, *num_modes=None*, *reverse=False*)

Changes the ladder operator order of the Hamiltonian based on the provided order_function per mode index.

Parameters

- **operator** (SymbolicOperator) – the operator that will be reordered. must be a SymbolicOperator or any type of operator that inherits from SymbolicOperator.
- **order_function** (*func*) – a function per mode that is used to map the indexing. must have arguments mode index and num_modes.
- **num_modes** (*int*) – default None. User can provide the number of modes assumed for the system. if None, the number of modes will be calculated based on the Operator.
- **reverse** (*bool*) – default False. if set to True, the mode mapping is reversed. reverse = True will not revert back to original if num_modes calculated differs from original and reverted.

Note: Every order function must take in a mode_idx and num_modes.

openfermion.utils.**s_minus_operator**(*n_spatial_orbitals*)

Return the s+ operator.

$$S^- = \sum_{i=1}^{n} a_{i,\beta}^\dagger a_{i,\alpha} \tag{1.20}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits + 1 // 2).

Returns *operator (FermionOperator)* – corresponding to the s- operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`s_plus_operator`**(*n_spatial_orbitals*)

Return the s+ operator.

$$S^{+} = \sum_{i=1}^{n} a_{i,\alpha}^{\dagger} a_{i,\beta} \tag{1.21}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits + 1 // 2).

Returns *operator (FermionOperator)* – corresponding to the s+ operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`s_squared_operator`**(*n_spatial_orbitals*)

Return the s^{2} operator.

$$S^{2} = S^{-}S^{+} + S^{z}(S^{z} + 1) \tag{1.22}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits + 1 // 2).

Returns *operator (FermionOperator)* – corresponding to the s+ operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`save_operator`**(*operator*, *file_name=None*, *data_directory=None*, *allow_overwrite=False*, *plain_text=False*)

Save FermionOperator or QubitOperator to file.

Parameters

- **`operator`** – An instance of FermionOperator, BosonOperator, or QubitOperator.
- **`file_name`** – The name of the saved file.
- **`data_directory`** – Optional data directory to change from default data directory specified in config file.
- **`allow_overwrite`** – Whether to allow files to be overwritten.
- **`plain_text`** – Whether the operator should be saved to a plain-text format for manual analysis

Raises

- `OperatorUtilsError` – Not saved, file already exists.
- `TypeError` – Operator of invalid type.

`openfermion.utils.`**`slater_determinant_preparation_circuit`**`(`*slater_determinant_matrix*`)`

Obtain the description of a circuit which prepares a Slater determinant.

The input is an $N_f \times N$ matrix Q with orthonormal rows. Such a matrix describes the Slater determinant

$$b_1^\dagger \cdots b_{N_f}^\dagger |\text{vac}\rangle,$$

where

$$b_j^\dagger = \sum_{k=1}^N Q_{jk} a_k^\dagger.$$

The output is the description of a circuit which prepares this Slater determinant, up to a global phase. The starting state which the circuit should be applied to is a Slater determinant (in the computational basis) with the first N_f orbitals filled.

Parameters **`slater_determinant_matrix`** – The matrix Q which describes the Slater determinant to be prepared.

Returns *circuit_description* – A list of operations describing the circuit. Each operation is a tuple of elementary operations that can be performed in parallel. Each elementary operation is a tuple of the form (i, j, θ, φ), indicating a Givens rotation of modes i and j by angles θ and φ.

`openfermion.utils.`**`sparse_eigenspectrum`**`(`*sparse_operator*`)`

Perform a dense diagonalization.

Returns *eigenspectrum* – The lowest eigenvalues in a numpy array.

`openfermion.utils.`**`sx_operator`**`(`*n_spatial_orbitals*`)`

Return the sx operator.

$$S^x = \frac{1}{2}\sum_{i=1}^n (S^+ + S^-) \tag{1.23}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits // 2).

Returns *operator (FermionOperator)* – corresponding to the sx operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`sy_operator`**`(`*n_spatial_orbitals*`)`

Return the sy operator.

$$S^y = \frac{-i}{2}\sum_{i=1}^n (S^+ - S^-) \tag{1.24}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits // 2).

Returns *operator (FermionOperator)* – corresponding to the sx operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`sz_operator`**`(`*n_spatial_orbitals*`)`

Return the sz operator.

$$S^z = \frac{1}{2}\sum_{i=1}^n (n_{i,\alpha} - n_{i,\beta}) \tag{1.25}$$

Parameters **`n_spatial_orbitals`** – number of spatial orbitals (n_qubits // 2).

Returns *operator (FermionOperator)* – corresponding to the sz operator over n_spatial_orbitals.

Note: The indexing convention used is that even indices correspond to spin-up (alpha) modes and odd indices correspond to spin-down (beta) modes.

`openfermion.utils.`**`taper_off_qubits`**(*operator*, *stabilizers*, *manual_input=False*, *fixed_positions=None*, *output_tapered_positions=False*)

Remove qubits from given operator.

Qubits are removed by eliminating an equivalent number of stabilizer conditions. Which qubits that are can either be determined automatically or their positions can be set manually.

Qubits can be disregarded from the Hamiltonian when the effect of all its terms on them is rendered trivial. This algorithm employs a stabilizers like $\pm X \otimes p$ to fix the action of every Pauli string on the first qubit to Z or the identity. A string $X \otimes h$ would for instance be multiplied with the stabilizer to obtain $1 \otimes (\pm h \cdot p)$ while a string $Z \otimes h'$ would pass without correction. The first qubit can subsequently be removed as it must be in the computational basis in Hamiltonian eigenstates. For stabilizers acting as Y (Z) on selected qubits, the algorithm would fix the action of every Hamiltonian string to Z (X). Updating also the list of remaining stabilizer generators, the algorithm is run iteratively.

Parameters

- **`operator`** (`QubitOperator`) – Operator of which qubits will be removed.
- **`stabilizers`** (`QubitOperator`) – Stabilizer generators for the tapering. Can also be passed as a list of QubitOperator.
- **`manual_input`** (*`Boolean`*) – Option to pass the list of fixed qubits positions manually. Set to False by default.
- **`fixed_positions`** (*`list`*) – (optional) List of fixed qubit positions. Passing a list is only effective if manual_input is True.
- **`output_tapered_positions`** (*`Boolean`*) – Option to output the positions of qubits that have been removed.

Returns

skimmed_operator (QubitOperator) – Operator with fewer qubits. removed_positions (list): (optional) List of removed qubit positions.

For the qubits to be gone in the qubit count, the remaining qubits have been moved up to those indices.

`openfermion.utils.`**`trotter_operator_grouping`**(*hamiltonian*, *trotter_number=1*, *trotter_order=1*, *term_ordering=None*, *k_exp=1.0*)

Trotter-decomposes operators into groups without exponentiating.

Operators are still Hermitian at the end of this method but have been multiplied by k_exp.

Note: The default term_ordering is simply the ordered keys of the QubitOperators.terms dict.

Parameters

- **`hamiltonian`** (`QubitOperator`) – full hamiltonian
- **`trotter_number`** (*`int`*) – optional number of trotter steps - default is 1

- **trotter_order** (*int*) – optional order of trotterization as an integer from 1-3 - default is 1
- **term_ordering** (*list of (tuples of tuples)*) – optional list of QubitOperator terms dictionary keys that specifies order of terms when trotterizing
- **k_exp** (*float*) – optional exponential factor to all terms when trotterizing

Yields QubitOperator generator

Raises

- ValueError if order > 3 or order <= 0,
- TypeError for incorrect types

openfermion.utils.**trotterize_exp_qubop_to_qasm**(*hamiltonian*, *evolution_time=1*, *trotter_number=1*, *trotter_order=1*, *term_ordering=None*, *k_exp=1.0*, *qubit_list=None*, *ancilla=None*)

Trotterize a Qubit hamiltonian and write it to QASM format.

Assumes input hamiltonian is still hermitian and -1.0j has not yet been applied. Therefore, signs of coefficients should reflect this. Returns a generator which generates a QASM file.

Parameters

- **hamiltonian** (QubitOperator) – hamiltonian
- **trotter_number** (*int*) – optional number of trotter steps (slices) for trotterization as an integer - default = 1
- **trotter_order** – optional order of trotterization as an integer - default = 1
- **term_ordering** (*list of (tuples of tuples)*) – list of tuples (QubitOperator terms dictionary keys) that specifies order of terms when trotterizing
- **qubit_list** – (list/tuple or None)Specifies the labels for the qubits to be output in qasm. If a list/tuple, must have length greater than or equal to the number of qubits in the QubitOperator. Entries in the list must be castable to string. If None, qubits are labeled by index (i.e. an integer).
- **k_exp** (*float*) – optional exponential factor to all terms when trotterizing
- **Yields** – string generator

openfermion.utils.**uccsd_convert_amplitude_format**(*single_amplitudes*, *double_amplitudes*)

Re-format single_amplitudes and double_amplitudes from ndarrays to lists.

Parameters

- **single_amplitudes** (*ndarray*) – [NxN] array storing single excitation amplitudes corresponding to t[i,j] * (a_i^dagger a_j - H.C.)
- **double_amplitudes** (*ndarray*) – [NxNxNxN] array storing double excitation amplitudes corresponding to t[i,j,k,l] * (a_i^dagger a_j a_k^dagger a_l - H.C.)

Returns

single_amplitudes_list(list) –

list of lists with each sublist storing a list of indices followed by single excitation amplitudes i.e. [[[i,j],t_ij], . . .]

double_amplitudes_list(list): list of lists with each sublist storing a list of indices followed by double excitation amplitudes i.e. [[[i,j,k,l],t_ijkl], . . .]

`openfermion.utils.`**`uccsd_generator`**(*single_amplitudes*, *double_amplitudes*, *anti_hermitian=True*)

Create a fermionic operator that is the generator of uccsd.

This a the most straight-forward method to generate UCCSD operators, however it is slightly inefficient. In particular, it parameterizes all possible excitations, so it represents a generalized unitary coupled cluster ansatz, but also does not explicitly enforce the uniqueness in parametrization, so it is redundant. For example there will be a linear dependency in the ansatz of single_amplitudes[i,j] and single_amplitudes[j,i].

Parameters

- **`single_amplitudes`** (*`list or ndarray`*) – list of lists with each sublist storing a list of indices followed by single excitation amplitudes i.e. [[[i,j],t_ij], . . .] OR [NxN] array storing single excitation amplitudes corresponding to t[i,j] * (a_i^dagger a_j - H.C.)
- **`double_amplitudes`** (*`list or ndarray`*) – list of lists with each sublist storing a list of indices followed by double excitation amplitudes i.e. [[[i,j,k,l],t_ijkl], . . .] OR [NxNxNxN] array storing double excitation amplitudes corresponding to t[i,j,k,l] * (a_i^dagger a_j a_k^dagger a_l - H.C.)
- **`anti_hermitian`** (*`Bool`*) – Flag to generate only normal CCSD operator rather than unitary variant, primarily for testing

Returns *uccsd_generator(FermionOperator)* – Anti-hermitian fermion operator that is the generator for the uccsd wavefunction.

`openfermion.utils.`**`uccsd_singlet_generator`**(*packed_amplitudes*, *n_qubits*, *n_electrons*, *anti_hermitian=True*)

Create a singlet UCCSD generator for a system with n_electrons

This function generates a FermionOperator for a UCCSD generator designed to act on a single reference state consisting of n_qubits spin orbitals and n_electrons electrons, that is a spin singlet operator, meaning it conserves spin.

Parameters

- **`packed_amplitudes`** (*`list`*) – List storing the unique single and double excitation amplitudes for a singlet UCCSD operator. The ordering lists unique single excitations before double excitations.
- **`n_qubits`** (*`int`*) – Number of spin-orbitals used to represent the system, which also corresponds to number of qubits in a non-compact map.
- **`n_electrons`** (*`int`*) – Number of electrons in the physical system.
- **`anti_hermitian`** (*`Bool`*) – Flag to generate only normal CCSD operator rather than unitary variant, primarily for testing

Returns

generator(FermionOperator) –

Generator of the UCCSD operator that builds the UCCSD wavefunction.

`openfermion.utils.`**`uccsd_singlet_get_packed_amplitudes`**(*single_amplitudes*, *double_amplitudes*, *n_qubits*, *n_electrons*)

Convert amplitudes for use with singlet UCCSD

The output list contains only those amplitudes that are relevant to singlet UCCSD, in an order suitable for use with the function *uccsd_singlet_generator*.

Parameters

- **single_amplitudes** (*ndarray*) – [NxN] array storing single excitation amplitudes corresponding to t[i,j] * (a_i^dagger a_j - H.C.)
- **double_amplitudes** (*ndarray*) – [NxNxNxN] array storing double excitation amplitudes corresponding to t[i,j,k,l] * (a_i^dagger a_j a_k^dagger a_l - H.C.)
- **n_qubits** (*int*) – Number of spin-orbitals used to represent the system, which also corresponds to number of qubits in a non-compact map.
- **n_electrons** (*int*) – Number of electrons in the physical system.

Returns

packed_amplitudes(list) –

List storing the unique single and double excitation amplitudes for a singlet UCCSD operator. The ordering lists unique single excitations before double excitations.

openfermion.utils.**uccsd_singlet_paramsize**(*n_qubits*, *n_electrons*)

Determine number of independent amplitudes for singlet UCCSD

Parameters

- **n_qubits** (*int*) – Number of qubits/spin-orbitals in the system
- **n_electrons** (*int*) – Number of electrons in the reference state

Returns Number of independent parameters for singlet UCCSD with a single reference.

openfermion.utils.**up_then_down**(*mode_idx*, *num_modes*)

up then down reordering, given the operator has the default even-odd ordering. Otherwise this function will reorder indices where all even indices now come before odd indices.

Example: 0,1,2,3,4,5 -> 0,2,4,1,3,5

The function takes in the index of the mode that will be relabeled and the total number modes.

Parameters

- **mode_idx** (*int*) – the mode index that is being reordered
- **num_modes** (*int*) – the total number of modes of the operator.

Returns (int): reordered index of the mode.

openfermion.utils.**variance**(*operator*, *state*)

Compute variance of operator with a state.

Parameters

- **operator** (*scipy.sparse.spmatrix or scipy.sparse.linalg.LinearOperator*) – The operator whose expectation value is desired.
- **state** (*numpy.ndarray or scipy.sparse.spmatrix*) – A numpy array representing a pure state or a sparse matrix representing a density matrix.

Returns A complex number giving the variance.

Raises ValueError – Input state has invalid format.

openfermion.utils.**wedge**(*left_tensor*, *right_tensor*, *left_index_ranks*, *right_index_ranks*)

Implement the wedge product between left_tensor and right_tensor

The wedge product is defined as

$$a^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_p} \wedge b^{i_{p+1},i_{p+2},\ldots,i_N}_{j_{p+1},j_{p+2},\ldots,j_N} = \left(\frac{1}{N!}\right)^2 = \sum_{\pi,\sigma} \epsilon(\pi)\epsilon(\sigma)\pi\sigma a^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_p} b^{i_{p+1},i_{p+2},\ldots,i_N}_{j_{p+1},j_{p+2},\ldots,j_N} \tag{1.26}$$

The top indices are those that transform contravariently. The bottom indices transform covariently.

The tensor storage convention for marginals follows the OpenFermion convention. tpdm[i, j, k, l] = <i^ j^ k l>, rtensor[u1, u2, u3, d1] = <u1^ u2^ u3^ d1>

Parameters

- **left_tensor** – left tensor to wedge product
- **right_tensor** – right tensor to wedge product
- **left_index_ranks** – tuple of number of indices that transform contravariently and covariently
- **right_index_ranks** – tuple of number of indices that transform contravariently and covariently

Returns new tensor constructed as the wedge product of the left_tensor and right_tensor

Python Module Index

Index