

---

# **openfermion Documentation**

*Release 0.8*

**openfermion**

**Aug 17, 2018**



---

## Contents

---

<b>1</b>	<b>Code Documentation</b>	<b>3</b>
1.1	openfermion.hamiltonians . . . . .	3
1.2	openfermion.measurements . . . . .	14
1.3	openfermion.ops . . . . .	16
1.4	openfermion.transforms . . . . .	29
1.5	openfermion.utils . . . . .	38
	<b>Python Module Index</b>	<b>67</b>



## Contents

- *Code Documentation*: The code documentation of OpenFermion.



## 1.1 openfermion.hamiltonians

**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 in 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** `MissingCalculationError` – 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** `MissingCalculationError` – 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  $x\_dimension \times y\_dimension$ . 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

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

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
- $\mu$  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  $\mu$  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

$$H = -t \sum_{\langle i,j \rangle} \sum_{\sigma} (a_{i,\sigma}^{\dagger} a_{j,\sigma} + a_{j,\sigma}^{\dagger} a_{i,\sigma}) + U \sum_i a_{i,\uparrow}^{\dagger} a_{i,\uparrow} a_{i,\downarrow}^{\dagger} a_{i,\downarrow} - \mu \sum_i \sum_{\sigma} a_{i,\sigma}^{\dagger} a_{i,\sigma} - h \sum_i (a_{i,\uparrow}^{\dagger} a_{i,\uparrow} - a_{i,\downarrow}^{\dagger} a_{i,\downarrow}) \quad (1.1)$$

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
- $\mu$  is the chemical potential
- $h$  is the magnetic field

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

$$H = -t \sum_{k=1}^{N-1} (a_k^{\dagger} a_{k+1} + a_{k+1}^{\dagger} a_k) + U \sum_{k=1}^{N-1} a_k^{\dagger} a_k a_{k+1}^{\dagger} a_{k+1} - \mu \sum_{k=1}^N a_k^{\dagger} a_k.$$

### 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  $\mu$  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.hypercube_grid_with_given_wigner_seitz_radius_and_filling` (*dimension*, *grid\_length*, *wigner\_seitz\_radius*, *filling\_fraction*, *spinless=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

$$H = -t \sum_{\langle i,j \rangle} \sum_{\sigma} (a_{i,\sigma}^{\dagger} a_{j,\sigma} + a_{j,\sigma}^{\dagger} a_{i,\sigma}) - \mu \sum_i \sum_{\sigma} a_{i,\sigma}^{\dagger} a_{i,\sigma} \quad (1.3)$$

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

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
- $\Delta_{ij}$  is equal to  $+\Delta/2$  for horizontal edges and  $-\Delta/2$  for vertical edges, where  $\Delta$  is the superconducting gap.
- $\mu$  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  $\Delta$
- **chemical\_potential** (*float*, *optional*) – The chemical potential  $\mu$  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.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.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.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  $|w_0\rangle |w_1\rangle |w_2\rangle \dots |w_{n-1}\rangle$ , on the other hand is described by a length- $n$  binary vector  $w=(w_0, w_1, \dots, w_{n-1})$ . 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  $e''$ :  $v'' \rightarrow e'(e(v''))$  and  $d''$ :  $w'' \rightarrow d(d'(w''))$  where  $n'' = n'$  and  $N'' = N$ , with  $n = N'$  as necessary condition.

Appendage:  $(e'',d'') = (e,d) + (e',d')$  which means  $e''$ :  $(v + v') \rightarrow e(v) + e'(v')$  and  $d''$ :  $(w + w') \rightarrow d(w) + d'(w')$  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  $n \times N$  array or matrix-like nested lists  $A$ , such that  $e(v) = (A v) \bmod 2$ . 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* – list of BinaryPolynomial: Outputs the decoding functions as components.

**encoder**

*scipy.sparse.csc\_matrix* – Outputs  $A$ , the linear matrix that implements the encoding function.

**n\_modes**

*int* – Outputs the number of modes.

**n\_qubits**

*int* – Outputs the number of qubits.

**\_\_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  $w_0$ ,  $w_1$ ,  $w_2$  and so on) that is considered to be evaluated modulo 2. This implies the following set of rules:

the binary addition  $w_1 + w_1 = 0$ , binary multiplication  $w_2 * w_2 = w_2$  and power rule  $w_3 ^ 0 = 1$ , where raising to every other integer power than zero reproduces  $w_3$ .

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.

$1 + w_1 w_2 + w_0 w_1$  .

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  $|w_0\rangle |w_1\rangle |w_2\rangle \dots$  .

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,  $1 + w_1 w_2 + w_0 w_1$  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

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

**\_\_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:  $'i^{\dagger} = b^{\dagger}_i$   $'j = b_j$  where  $['i', 'j^{\dagger}] = \delta_{ij}$  is the commutator.

One can multiply together these bosonic ladder operators to obtain a bosonic term. For instance,  $'2^{\dagger} 1'$  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.  $'3.17 * 2^{\dagger} 1'$ .

The `BosonOperator` class is designed (in general) to store sums of these terms. For instance, an instance of `BosonOperator` might represent  $3.17 2^{\dagger} 1 - 66.2 * 8^{\dagger} 7 6^{\dagger} 2$  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*)  
 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**

*ndarray* – The Hermitian matrix  $T$ .

**two\_body**

*ndarray* – The real symmetric matrix  $V$ .

**constant**

*float* – The constant.

**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:  $'q^\wedge' = a^\dagger_{q}$   $'q' = a_{q}$  where  $\{ 'p^\wedge', 'q' \} = \text{delta}_{pq}$

One can multiply together these fermionic ladder operators to obtain a fermionic term. For instance,  $'2^\wedge 1'$  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.  $'3.17 * 2^\wedge 1'$ .

The `FermionOperator` class is designed (in general) to store sums of these terms. For instance, an instance of `FermionOperator` might represent  $3.17 2^\wedge 1 - 66.2 * 8^\wedge 7 6^\wedge 2$  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:  $constant + \sum_{p, q} h_{[p, q]} a^\dagger_{p} a_q +$

$\sum_{p, q, r, s} h_{[p, q, r, s]} a^\dagger_{p} a^\dagger_{q} a_r a_s.$



**one\_body\_tensor**

The coefficients of the one-body terms ( $h[p, q]$ ). This is an  $n_{\text{qubits}} \times n_{\text{qubits}}$  numpy array of floats.

**two\_body\_tensor**

The coefficients of the two-body terms ( $h[p, q, r, s]$ ). This is an  $n_{\text{qubits}} \times n_{\text{qubits}} \times n_{\text{qubits}} \times n_{\text{qubits}}$  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  $n_{\text{qubits}} \times n_{\text{qubits}}$  numpy array of floats.
- **two\_body\_tensor** – The coefficients of the two-body terms ( $h[p, q, r, s]$ ). This is an  $n_{\text{qubits}} \times n_{\text{qubits}} \times n_{\text{qubits}} \times n_{\text{qubits}}$  numpy array of floats.

**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  $\langle a^\dagger a \rangle$ .

**two\_body\_tensor**

The expectation values  $\langle a^\dagger a^\dagger a a \rangle$ .

**\_\_init\_\_** (*one\_body\_tensor, two\_body\_tensor*)

Initialize the InteractionRDM class.

**Parameters**

- **one\_body\_tensor** – Expectation values  $\langle a^\dagger a \rangle$ .
- **two\_body\_tensor** – Expectation values  $\langle a^\dagger a^\dagger a a \rangle$ .

**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 Interaction-RDM.

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

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

**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  $n_{\text{qubits}} \times n_{\text{qubits}}$  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 redundant with `FermionOperator` but enables much more efficient numerical computations in many cases, such as basis rotations.

**n\_qubits**

*int* – The number of sites on which the tensor acts.

**n\_body\_tensors**

*dict* – 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  $(n_{\text{qubits}} \times n_{\text{qubits}})$  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  $a_i a^\dagger_j$ .

**\_\_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.

**rotate\_basis** (*rotation\_matrix*)

Rotate the orbital basis of the PolynomialTensor.

**Parameters** `rotation_matrix` – A square numpy array or matrix having dimensions of  $n_{\text{qubits}}$  by  $n_{\text{qubits}}$ . 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:  $q = \sqrt{\hbar/2}(b+b^\dagger)$   $p = -i\sqrt{\hbar/2}(b-b^\dagger)$  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  $n\_qubits \times n\_qubits$  matrix.
- $\Delta$  is an antisymmetric  $n\_qubits \times n\_qubits$  matrix.
- $\mu$  is a real number representing the chemical potential.
- $\delta_{pq}$  is the Kronecker delta symbol.

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

#### **chemical\_potential**

*float* – The chemical potential  $\mu$ .

**\_\_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  $n\_qubits \times n\_qubits$  numpy array of complex numbers.
- **antisymmetric\_part** (*ndarray*) – The matrix  $\Delta$ , which represents the coefficients of the non-particle-number-conserving terms. This is an  $n\_qubits \times n\_qubits$  numpy array of complex numbers.
- **constant** (*float, optional*) – A constant term in the operator.
- **chemical\_potential** (*float, optional*) – The chemical potential  $\mu$ .

**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** ()

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  $W$ .

**Returns** *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  $N \times 2N$ .

#### **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  $(i, j, \theta, \varphi)$ , indicating a Givens rotation of modes  $i$  and  $j$  by angles  $\theta$  and  $\varphi$ .

#### **ground\_energy** ()

Return the ground energy.

#### **hermitian\_part**

The Hermitian part not including the chemical potential.

#### **majorana\_form** ()

Return the Majorana representation 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  $(2 * n\_qubits) \times (2 * n\_qubits)$  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  $\varepsilon_j$  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  $\epsilon_j$
- `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.

---

**static accumulate** (*operators, zero=None*)

Sums over `QubitOperators`.

**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.

**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*([*QubitOperator*]) – A list of operators summing up to self.

**get\_operators** ()

Gets a list of operators with a single term.

**Returns** *operators*([*QubitOperator*]) – A list of operators summing up to self.

**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**

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

**action\_strings**

*tuple* – 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 ('^', '^').

**action\_before\_index**

*bool* – A boolean indicating whether in string representations, the action should come before the index.

**different\_indices\_commute**

*bool* – A boolean indicating whether factors acting on different indices commute.

**terms**

*dict* – **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.

**\_\_init\_\_** (*term=None, coefficient=1.0*)

*x.\_\_init\_\_(...)* initializes *x*; see help(type(*x*)) for signature

**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-12*)

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.

**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  $\sum_j w_j H_j$  where  $w_j$  are scalar coefficients then this norm is  $\left(\sum_j |w_j|^p\right)^{\frac{1}{p}}$  where  $p$  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.general_basis_change` (*general\_tensor, rotation\_matrix, key*)

Change the basis of an general interaction tensor.

$$M'^{\{p_1 p_2 \dots p_n\}} = R^{\{p_1\}_{\{a_1\}}} R^{\{p_2\}_{\{a_2\}}} \dots R^{\{p_n\}_{\{a_n\}}} M^{\{a_1 a_2 \dots a_n\}} \\ R^{\{p_n\}_{\{a_n\}}T} \dots R^{\{p_2\}_{\{a_2\}}T} R_{\{p_1\}_{\{a_1\}}T}$$

where *R* is the rotation matrix, *M* is the general tensor, *M'* is the transformed general tensor, and *a<sub>k</sub>* and *p<sub>k</sub>* 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  $a_p^\dagger a_q$  would have a key of (1, 0) whereas a tensor storing coefficients of  $a_p^\dagger a_q a_r a_s^\dagger$  would have a key of (1, 0, 0, 1).

**Returns** `transformed_general_tensor` – `general_tensor` in the rotated basis.

## 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 ( $v_0 v_1 v_2 \dots v_{N-1}$ ) that span the fermionic basis, to the qubit basis, spanned by binary vectors ( $w_0, w_1, w_2, \dots, w_{N-1}$ ).

The binary code has to provide an analytic relation between the binary vectors ( $v_0, v_1, \dots, v_{N-1}$ ) and ( $w_0, w_1, \dots, w_{N-1}$ ), 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.  $CPhase(1,2) = 0.5 * (1 + Z_1 + Z_2 - Z_1 Z_2)$ ), 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** – `InteractionOperator`.

**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  $[q_i, p_j] = i \text{hbar} \delta_{ij}$ . 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  $a_p^\dagger a_q$  or  $a_p^\dagger a_q^\dagger a_r a_s$ . 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_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_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  $[q_i, p_j] = i \text{hbar} \delta_{ij}$ . 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:  $a_j^\dagger \rightarrow Z_0 \dots Z_{j-1} (X_j - iY_j) / 2$   $a_j \rightarrow Z_0 \dots Z_{j-1} (X_j + iY_j) / 2$

**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*)

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 ( $\langle Z \rangle = 1$ ) or the 1 subspace ( $\langle Z \rangle = -1$ ).

**Returns** *projected\_operator* – the resultant operator

**Raises** `TypeError` – operator must be a QubitOperator

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

Calculates 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 ( $\langle Z \rangle = 1$ ) or the 1 subspace ( $\langle Z \rangle = -1$ ).

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

**Raises** `TypeError` – operator must be a QubitOperator

`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:  $Z_j \rightarrow I - 2 a^\dagger_j a_j$   $X_j \rightarrow (a^\dagger_j + a_j) Z_{\{j-1\}} Z_{\{j-2\}} \dots Z_0$   
 $Y_j \rightarrow i (a^\dagger_j - a_j) Z_{\{j-1\}} Z_{\{j-2\}} \dots Z_0$

**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)`

Performs the rotation  $e^{-i \text{heta} * P} Q e^{i \text{heta} * P}$  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:

$$q^m p^n \rightarrow (1/2^n) \sum_{r=0}^n \text{Binomial}(n, r) q^r p^m q^{n-r}$$

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;  $S(q^m p^n) = S(q^m p^n)$ , 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 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*, *ac-*  
*tive\_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  $n_{\text{modes}} = 2^{\text{exponent}}$ ,  $n_{\text{qubits}} = \text{exponent}$ .

---

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

---

**Parameters** **exponent** (*int*) – exponent for the number of modes  $n_{\text{modes}} = 2^{\text{exponent}}$

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.  $n_{\text{qubits}} = 2$ ,  $n_{\text{modes}} = 3$ . 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.  $n_{\text{qubits}} = 4$ ,  $n_{\text{modes}} = 5$ . 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  $q^m p^n$ :

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

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 ‘ $qi^m pj^n \dots$ ’ 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:  $\text{linear\_operator} * v - v * \text{lambda}$ .
- **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 parallelepiped 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**

*int* – Number of spatial dimensions the grid occupies

**length**

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

**shifts**

*list of ints* – Integer shifts in position to center grid.

**scale**

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

**volume**

*float* – Total volume of the supercell parallelepiped.

**num\_points**

*int* – Total number of points in the grid.

**reciprocal\_scale**

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

**\_\_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 an 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.LinearQubitOperator` (*qubit\_operator*, *n\_qubits=None*)

A LinearOperator implied from a QubitOperator.

The idea is that a single *i*-th qubit operator, *O<sub>i</sub>*, is a 2-by-2 matrix, to be applied on a vector of length  $n_{\text{hilbert}} / 2^i$ , 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  $n_{\text{hilbert}} / 2^i$ , 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.  $\{O_j \mid j > i\}$ , 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.

`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  $\log[e^{x_1} \dots e^{x_N}]$  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  $\text{trunc} \times \text{trunc}$  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  $[q_i, p_j] = \delta_{ij} \hbar$ . This only applies if calculating the sparse representation of a quadrature operator.

**Returns** The corresponding Scipy sparse matrix of size  $[\text{trunc}, \text{trunc}]$ .

`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.down_index` (*index*)

Function to return down-orbital index given a spatial orbital index.

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

`openfermion.utils.eigspectrum` (*operator*, *n\_qubits=None*)

Compute the eigspectrum 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 eigspectrum.

`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 operator 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*)

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

$$\begin{aligned} \mathcal{B}a_N\mathcal{B}^\dagger &= a_N^\dagger, \\ \mathcal{B}a_j\mathcal{B}^\dagger &= a_j, \quad j = 1, \dots, N \end{aligned} \quad (1.5)$$

or

- a tuple  $(i, j, \theta, \varphi)$ , 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  $\theta$  and  $\varphi$ .

#### 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

- *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  $(i, j, \theta, \varphi)$ , indicating a Givens rotation of modes  $i$  and  $j$  by angles  $\theta$  and  $\varphi$ .
- *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.geometry_from_pubchem(name)`

Function to extract geometry using the molecule's name from the PubChem database.

**Parameters** `name` – a string giving the molecule's name as required by the PubChem database.

**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  $\sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$

We will convert this to the chemistry convention expressing it as  $\sum_{pqrs} g_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$  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  $N \times N \times N \times N$  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  $N \times N$  array of floats giving coefficients of the  $a_p^\dagger a_q$  terms that come out.

**chemist\_two\_body\_coefficients** (`ndarray`): an  $N \times N \times N \times N$  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.haar_random_vector(n, seed=None)`

Generate an *n* dimensional Haar random 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.hermertian_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.

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

### 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 or 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:  $a_j^\dagger \rightarrow Z_0 \dots Z_{j-1} (X_j - iY_j) / 2$   $a_j \rightarrow Z_0 \dots Z_{j-1} (X_j + iY_j) / 2$

### 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*)

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

**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*)

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

**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  $i^{\wedge} i$ , 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  $i^{\wedge} i$ , 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  $\sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$  as  $\sum_l \lambda_l (\sum_{pq} g_{lpq} a_p^\dagger a_q)^2$   $l$  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  $N \times N \times N \times N$  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  $\lambda_l$ .

**one\_body\_squares** (*ndarray of floats*):  $L \times N \times N$  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:  $a_p^\dagger + a_p$

Type 1:  $i(a_p^\dagger - a_p)$

where the  $a_p^\dagger$  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_q^\dagger`.

**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  $\text{tqdm}[p, q, r, s] == a_p a_q a_r^\dagger a_s^\dagger$ .
- **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  $\text{tqdm}[p, q, r, s] == a_p a_q a_r^\dagger a_s^\dagger$ .
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of  $\text{opdm}[p, q] == a_p^\dagger a_q$ .

**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  $\text{tpdm}[p, q, r, s] == a_p^\dagger a_q^\dagger a_r a_s$ .
- **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  $\text{tpdm}[p, q, r, s] == a_p^\dagger a_q^\dagger a_r a_s$ .
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of  $\text{opdm}[p, q] == a_p^\dagger a_q$ .

**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  $\text{tpdm}[p, q, r, s] == a_p^\dagger a_q^\dagger a_r a_s$ .
- **opdm** (*numpy.ndarray*) – The 1-RDM as a 2-index tensor. Indices follow the internal convention of  $\text{opdm}[p, q] == a_p^\dagger a_q$ .

**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.

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, or QuadOperator 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  $\text{parity}=-1$  (default), and returns the bosonic number operator if  $\text{parity}=1$ .

**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:**  
 $\exp(-1.0j * \text{evolution\_time} * \text{op})$ .

#### 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  $(\sum_{pq} h_{pq} a_p^\dagger a_q)^2$  by decomposing as  $R e^{-i \sum_{pq} V_{pq} n_p n_q} R^\dagger$  where : *math* : ‘ $R$  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 coefficients  $V_{pq}$  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*:  $|k\rangle$  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  $[0, \text{len}(lcu\_coefficients) - 1]$ . 2. With probability  $keep\_numers[i] / \text{by } keep\_denom$ , 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\_numbers 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  $n \times n$  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  $n \times n$  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_qubits, conserves_particle_number=False, real=False, seed=None)`

Generate a random instance of QuadraticHamiltonian.

#### Parameters

- **n\_qubits** (`int`) – the number of qubits
- **conserves\_particle\_number** (`bool`) – whether the returned Hamiltonian should conserve particle number
- **real** (`bool`) – whether to use only real numbers

**Returns** QuadraticHamiltonian

`openfermion.utils.random_unitary_matrix` (*n*, *real=False*, *seed=None*)

Obtain a random  $n \times n$  unitary matrix.

`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.

$$S^- = \sum_{i=1}^n a_{i,\beta}^\dagger a_{i,\alpha} \tag{1.7}$$

**Parameters** *n\_spatial\_orbitals* – number of spatial orbitals ( $n_{\text{qubits}} + 1 // 2$ ).

**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.

$$S^+ = \sum_{i=1}^n a_{i,\alpha}^\dagger a_{i,\beta} \tag{1.8}$$

**Parameters** *n\_spatial\_orbitals* – number of spatial orbitals ( $n_{\text{qubits}} + 1 // 2$ ).

**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.

$$S^2 = S^- S^+ + S^z (S^z + 1) \tag{1.9}$$

**Parameters** *n\_spatial\_orbitals* – number of spatial orbitals ( $n_{\text{qubits}} + 1 // 2$ ).

**Returns** *operator (FermionOperator)* – corresponding to the  $s^{\{2\}}$  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  $(i, j, \theta, \varphi)$ , indicating a Givens rotation of modes  $i$  and  $j$  by angles  $\theta$  and  $\varphi$ .

`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.

$$S^x = \frac{1}{2} \sum_{i=1}^n (S^+ + S^-) \tag{1.10}$$

**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.

$$S^y = \frac{-i}{2} \sum_{i=1}^n (S^+ - S^-) \quad (1.11)$$

**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.

$$S^z = \frac{1}{2} \sum_{i=1}^n (n_{i,\alpha} - n_{i,\beta}) \quad (1.12)$$

**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.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 Qubit-Operator. 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  $t_{[i,j]} * (a_i^\dagger a_j - \text{H.C.})$
- **double\_amplitudes** (*ndarray*) – [NxNxNxN] array storing double excitation amplitudes corresponding to  $t_{[i,j,k,l]} * (a_i^\dagger a_j a_k^\dagger a_l - \text{H.C.})$

**Returns**

*single\_amplitudes\_list(list)* –

**list of lists with each sublist storing** a list of indices followed by single excitation amplitudes i.e.  $[[[i,j],t_{ij}], \dots]$

**double\_amplitudes\_list(list): list of lists with each sublist storing** a list of indices followed by double excitation amplitudes i.e.  $[[[i,j,k,l],t_{ijkl}], \dots]$

```
openfermion.utils.uccsd_generator(single_amplitudes, double_amplitudes, anti_hermitian=True)
```

Create a fermionic operator that is the generator of uccsd.

This is 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  $t_{[i,j]} * (a_i^\dagger a_j - \text{H.C.})$
- **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  $t_{[i,j,k,l]} * (a_i^\dagger a_j a_k^\dagger a_l - \text{H.C.})$
- **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  $t_{[i,j]} * (a_i^\dagger a_j - \text{H.C.})$

- **double\_amplitudes** (*ndarray*) – [NxNxNxN] array storing double excitation amplitudes corresponding to  $t_{[i,j,k,l]} * (a_i^\dagger a_j a_k^\dagger a_l - \text{H.C.})$
- **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_index(index)`

Function to return up-orbital index given a spatial orbital index.

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

`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.



**O**

`openfermion.hamiltonians`, 3  
`openfermion.measurements`, 14  
`openfermion.ops`, 16  
`openfermion.transforms`, 29  
`openfermion.utils`, 38





## Symbols

- `__init__()` (openfermion.hamiltonians.MolecularData method), 5  
`__init__()` (openfermion.ops.BinaryCode method), 16  
`__init__()` (openfermion.ops.BinaryPolynomial method), 17  
`__init__()` (openfermion.ops.InteractionOperator method), 21  
`__init__()` (openfermion.ops.InteractionRDM method), 21  
`__init__()` (openfermion.ops.PolynomialTensor method), 22  
`__init__()` (openfermion.ops.QuadraticHamiltonian method), 24  
`__init__()` (openfermion.ops.SymbolicOperator method), 27  
`__init__()` (openfermion.utils.Davidson method), 38  
`__init__()` (openfermion.utils.DavidsonOptions method), 38  
`__init__()` (openfermion.utils.Grid method), 39  
`__init__()` (openfermion.utils.LinearQubitOperator method), 41  
`__init__()` (openfermion.utils.LinearQubitOperatorOptions method), 41  
`__init__()` (openfermion.utils.ParallelLinearQubitOperator method), 41  
`__init__()` (openfermion.utils.QubitDavidson method), 41  
`__init__()` (openfermion.utils.SparseDavidson method), 42
- A**
- `accumulate()` (openfermion.ops.QubitOperator static method), 26  
`action_before_index` (openfermion.ops.BosonOperator attribute), 19  
`action_before_index` (openfermion.ops.FermionOperator attribute), 20  
`action_before_index` (openfermion.ops.QuadOperator attribute), 23  
`action_before_index` (openfermion.ops.QubitOperator attribute), 26  
`action_before_index` (openfermion.ops.SymbolicOperator attribute), 27, 28  
`actions` (openfermion.ops.BosonOperator attribute), 19  
`actions` (openfermion.ops.FermionOperator attribute), 20  
`actions` (openfermion.ops.QuadOperator attribute), 23  
`actions` (openfermion.ops.QubitOperator attribute), 27  
`actions` (openfermion.ops.SymbolicOperator attribute), 27, 28  
`add_chemical_potential()` (openfermion.ops.QuadraticHamiltonian method), 24  
`all_points_indices()` (openfermion.utils.Grid method), 39  
`amplitude_damping_channel()` (in module openfermion.utils), 42  
`anticommutator()` (in module openfermion.utils), 42  
`antisymmetric_part` (openfermion.ops.QuadraticHamiltonian attribute), 24  
`apply_constraints()` (in module openfermion.measurements), 14  
`atoms` (openfermion.hamiltonians.MolecularData attribute), 4
- B**
- `basis` (openfermion.hamiltonians.MolecularData attribute), 3  
`bch_expand()` (in module openfermion.utils), 42

binary\_code\_transform() (in module openfermion.transforms), 29

BinaryCode (class in openfermion.ops), 16

BinaryPolynomial (class in openfermion.ops), 17

bose\_hubbard() (in module openfermion.hamiltonians), 6

boson\_ladder\_sparse() (in module openfermion.utils), 42

boson\_operator\_sparse() (in module openfermion.utils), 43

BosonOperator (class in openfermion.ops), 18

bravyi\_kitaev() (in module openfermion.transforms), 29

bravyi\_kitaev\_code() (in module openfermion.transforms), 30

bravyi\_kitaev\_fast() (in module openfermion.transforms), 30

bravyi\_kitaev\_tree() (in module openfermion.transforms), 30

## C

canonical\_orbitals (openfermion.hamiltonians.MolecularData attribute), 4

ccsd\_double\_amps (openfermion.hamiltonians.MolecularData attribute), 5

ccsd\_energy (openfermion.hamiltonians.MolecularData attribute), 4

ccsd\_single\_amps (openfermion.hamiltonians.MolecularData attribute), 4

charge (openfermion.hamiltonians.MolecularData attribute), 3

checksum\_code() (in module openfermion.transforms), 30

chemical\_potential (openfermion.ops.QuadraticHamiltonian attribute), 24

chemist\_ordered() (in module openfermion.utils), 43

cisd\_energy (openfermion.hamiltonians.MolecularData attribute), 4

cisd\_one\_rdm (openfermion.hamiltonians.MolecularData attribute), 4

cisd\_two\_rdm (openfermion.hamiltonians.MolecularData attribute), 4

combined\_hermitian\_part (openfermion.ops.QuadraticHamiltonian attribute), 24

commutator() (in module openfermion.utils), 43

compress() (openfermion.ops.SymbolicOperator method), 28

conserves\_particle\_number (openfermion.ops.QuadraticHamiltonian attribute), 24

constant (openfermion.ops.DiagonalCoulombHamiltonian attribute), 19

constant (openfermion.ops.PolynomialTensor attribute), 22

constant (openfermion.ops.SymbolicOperator attribute), 28

constraint\_matrix() (in module openfermion.measurements), 15

count\_qubits() (in module openfermion.utils), 44

## D

Davidson (class in openfermion.utils), 38

DavidsonOptions (class in openfermion.utils), 38

decoder (openfermion.ops.BinaryCode attribute), 16

dephasing\_channel() (in module openfermion.utils), 44

depolarizing\_channel() (in module openfermion.utils), 44

description (openfermion.hamiltonians.MolecularData attribute), 3

DiagonalCoulombHamiltonian (class in openfermion.ops), 19

diagonalizing\_bogoliubov\_transform() (openfermion.ops.QuadraticHamiltonian method), 24

diagonalizing\_circuit() (openfermion.ops.QuadraticHamiltonian method), 25

different\_indices\_commute (openfermion.ops.BosonOperator attribute), 19

different\_indices\_commute (openfermion.ops.FermionOperator attribute), 20

different\_indices\_commute (openfermion.ops.QuadOperator attribute), 23

different\_indices\_commute (openfermion.ops.QubitOperator attribute), 27

different\_indices\_commute (openfermion.ops.SymbolicOperator attribute), 27, 28

dimensions (openfermion.utils.Grid attribute), 39

dissolve() (in module openfermion.transforms), 30

double\_commutator() (in module openfermion.utils), 44

down\_index() (in module openfermion.utils), 45

dual\_basis\_external\_potential() (in module openfermion.hamiltonians), 7

dual\_basis\_jellium\_model() (in module openfermion.hamiltonians), 8

dual\_basis\_kinetic() (in module openfermion.hamiltonians), 8

dual\_basis\_potential() (in module openfermion.hamiltonians), 8

## E

edit\_hamiltonian\_for\_spin() (in module openfermion.transforms), 31

eigenspectrum() (in module openfermion.utils), 45

encoder (openfermion.ops.BinaryCode attribute), 16

- enumerate\_qubits() (openfermion.ops.BinaryPolynomial method), 17  
 error\_bound() (in module openfermion.utils), 45  
 error\_operator() (in module openfermion.utils), 45  
 evaluate() (openfermion.ops.BinaryPolynomial method), 17  
 expectation() (in module openfermion.utils), 46  
 expectation() (openfermion.ops.InteractionRDM method), 21  
 expectation\_computational\_basis\_state() (in module openfermion.utils), 46
- ## F
- fci\_energy (openfermion.hamiltonians.MolecularData attribute), 4  
 fci\_one\_rdm (openfermion.hamiltonians.MolecularData attribute), 4  
 fci\_two\_rdm (openfermion.hamiltonians.MolecularData attribute), 4  
 fermi\_hubbard() (in module openfermion.hamiltonians), 8  
 FermionOperator (class in openfermion.ops), 19  
 filename (openfermion.hamiltonians.MolecularData attribute), 3  
 fock\_matrix (openfermion.hamiltonians.MolecularData attribute), 4  
 fourier\_transform() (in module openfermion.utils), 46  
 freeze\_orbitals() (in module openfermion.utils), 46
- ## G
- gaussian\_state\_preparation\_circuit() (in module openfermion.utils), 47  
 general\_basis\_change() (in module openfermion.ops), 28  
 general\_calculations (openfermion.hamiltonians.MolecularData attribute), 5  
 generate\_linear\_qubit\_operator() (in module openfermion.utils), 47  
 geometry (openfermion.hamiltonians.MolecularData attribute), 3  
 geometry\_from\_pubchem() (in module openfermion.utils), 48  
 get\_active\_space\_integrals() (openfermion.hamiltonians.MolecularData method), 5  
 get\_boson\_operator() (in module openfermion.transforms), 31  
 get\_chemist\_two\_body\_coefficients() (in module openfermion.utils), 48  
 get\_diagonal\_coulomb\_hamiltonian() (in module openfermion.transforms), 31  
 get\_fermion\_operator() (in module openfermion.transforms), 31  
 get\_file\_path() (in module openfermion.utils), 48  
 get\_from\_file() (openfermion.hamiltonians.MolecularData method), 5  
 get\_gap() (in module openfermion.utils), 48  
 get\_ground\_state() (in module openfermion.utils), 49  
 get\_integrals() (openfermion.hamiltonians.MolecularData method), 6  
 get\_interaction\_operator() (in module openfermion.transforms), 31  
 get\_interaction\_rdm() (in module openfermion.transforms), 32  
 get\_linear\_qubit\_operator\_diagonal() (in module openfermion.utils), 49  
 get\_lowest\_n() (openfermion.utils.Davidson method), 38  
 get\_molecular\_data() (in module openfermion.transforms), 32  
 get\_molecular\_hamiltonian() (openfermion.hamiltonians.MolecularData method), 6  
 get\_molecular\_rdm() (openfermion.hamiltonians.MolecularData method), 6  
 get\_n\_alpha\_electrons() (openfermion.hamiltonians.MolecularData method), 6  
 get\_n\_beta\_electrons() (openfermion.hamiltonians.MolecularData method), 6  
 get\_operator\_groups() (openfermion.ops.QubitOperator method), 27  
 get\_operators() (openfermion.ops.QubitOperator method), 27  
 get\_pool() (openfermion.utils.LinearQubitOperatorOptions method), 41  
 get\_processes() (openfermion.utils.LinearQubitOperatorOptions method), 41  
 get\_quad\_operator() (in module openfermion.transforms), 32  
 get\_quadratic\_hamiltonian() (in module openfermion.transforms), 32  
 get\_qubit\_expectations() (openfermion.ops.InteractionRDM method), 22  
 get\_sparse\_operator() (in module openfermion.transforms), 33  
 Grid (class in openfermion.utils), 39  
 grid\_indices() (openfermion.utils.Grid method), 39  
 ground\_energy() (openfermion.ops.QuadraticHamiltonian method), 25
- ## H
- haar\_random\_vector() (in module openfermion.utils), 49  
 hartree\_fock\_state\_jellium() (in module openfermion.utils), 49  
 hermitian\_conjugated() (in module openfermion.utils), 49

- hermitian\_part (openfermion.ops.QuadraticHamiltonian attribute), 25
- hf\_energy (openfermion.hamiltonians.MolecularData attribute), 4
- hypercube\_grid\_with\_given\_wigner\_seitz\_radius\_and\_filling\_factor() (in module openfermion.hamiltonians), 10
- ## I
- identity() (openfermion.ops.BinaryPolynomial class method), 18
- identity() (openfermion.ops.SymbolicOperator class method), 28
- index\_to\_momentum\_ints() (openfermion.utils.Grid method), 40
- induced\_norm() (openfermion.ops.SymbolicOperator method), 28
- init\_lazy\_properties() (openfermion.hamiltonians.MolecularData method), 6
- inline\_sum() (in module openfermion.utils), 49
- inner\_product() (in module openfermion.utils), 50
- InteractionOperator (class in openfermion.ops), 20
- InteractionRDM (class in openfermion.ops), 21
- interleaved\_code() (in module openfermion.transforms), 33
- inverse\_fourier\_transform() (in module openfermion.utils), 50
- is\_boson\_preserving() (openfermion.ops.BosonOperator method), 19
- is\_gaussian() (openfermion.ops.QuadOperator method), 23
- is\_hermitian() (in module openfermion.utils), 50
- is\_identity() (in module openfermion.utils), 50
- is\_normal\_ordered() (openfermion.ops.BosonOperator method), 19
- is\_normal\_ordered() (openfermion.ops.FermionOperator method), 20
- is\_normal\_ordered() (openfermion.ops.QuadOperator method), 23
- is\_two\_body\_number\_conserving() (openfermion.ops.FermionOperator method), 20
- ## J
- jellium\_model() (in module openfermion.hamiltonians), 10
- jordan\_wigner() (in module openfermion.transforms), 33
- jordan\_wigner\_code() (in module openfermion.transforms), 33
- jordan\_wigner\_dual\_basis\_hamiltonian() (in module openfermion.hamiltonians), 10
- jordan\_wigner\_dual\_basis\_jellium() (in module openfermion.hamiltonians), 11
- jordan\_wigner\_sparse() (in module openfermion.utils), 50
- jw\_configuration\_state() (in module openfermion.utils), 50
- jw\_get\_gaussian\_state() (in module openfermion.utils), 50
- jw\_get\_ground\_state\_at\_particle\_number() (in module openfermion.utils), 51
- jw\_hartree\_fock\_state() (in module openfermion.utils), 51
- jw\_number\_restrict\_operator() (in module openfermion.utils), 51
- jw\_number\_restrict\_state() (in module openfermion.utils), 51
- jw\_slater\_determinant() (in module openfermion.utils), 52
- jw\_sz\_restrict\_operator() (in module openfermion.utils), 52
- jw\_sz\_restrict\_state() (in module openfermion.utils), 52
- ## L
- lambda\_norm() (in module openfermion.utils), 52
- length (openfermion.utils.Grid attribute), 39
- linearize\_decoder() (in module openfermion.transforms), 34
- linearize\_term() (in module openfermion.measurements), 15
- LinearQubitOperator (class in openfermion.utils), 40
- LinearQubitOperatorOptions (class in openfermion.utils), 41
- load\_molecular\_hamiltonian() (in module openfermion.hamiltonians), 11
- load\_operator() (in module openfermion.utils), 52
- low\_depth\_second\_order\_trotter\_error\_bound() (in module openfermion.utils), 53
- low\_depth\_second\_order\_trotter\_error\_operator() (in module openfermion.utils), 53
- low\_rank\_two\_body\_decomposition() (in module openfermion.utils), 54
- ## M
- majorana\_form() (openfermion.ops.QuadraticHamiltonian method), 25
- majorana\_operator() (in module openfermion.utils), 54
- make\_atom() (in module openfermion.hamiltonians), 11
- make\_atomic\_lattice() (in module openfermion.hamiltonians), 11
- make\_atomic\_ring() (in module openfermion.hamiltonians), 12
- many\_body\_order() (openfermion.ops.SymbolicOperator method), 28
- map\_one\_hole\_dm\_to\_one\_pdm() (in module openfermion.utils), 55
- map\_one\_pdm\_to\_one\_hole\_dm() (in module openfermion.utils), 55

- map\_particle\_hole\_dm\_to\_one\_pdm() (in module openfermion.utils), 55
- map\_particle\_hole\_dm\_to\_two\_pdm() (in module openfermion.utils), 55
- map\_two\_hole\_dm\_to\_one\_hole\_dm() (in module openfermion.utils), 55
- map\_two\_hole\_dm\_to\_two\_pdm() (in module openfermion.utils), 56
- map\_two\_pdm\_to\_one\_pdm() (in module openfermion.utils), 56
- map\_two\_pdm\_to\_particle\_hole\_dm() (in module openfermion.utils), 56
- map\_two\_pdm\_to\_two\_hole\_dm() (in module openfermion.utils), 56
- mean\_field\_dwave() (in module openfermion.hamiltonians), 12
- module\_importable() (in module openfermion.utils), 56
- MolecularData (class in openfermion.hamiltonians), 3
- momentum\_ints\_to\_index() (openfermion.utils.Grid method), 40
- momentum\_ints\_to\_value() (openfermion.utils.Grid method), 40
- momentum\_vector() (openfermion.utils.Grid method), 40
- mp2\_energy (openfermion.hamiltonians.MolecularData attribute), 4
- multiplicity (openfermion.hamiltonians.MolecularData attribute), 3
- ## N
- n\_atoms (openfermion.hamiltonians.MolecularData attribute), 3
- n\_body\_tensors (openfermion.ops.PolynomialTensor attribute), 22
- n\_electrons (openfermion.hamiltonians.MolecularData attribute), 3
- n\_modes (openfermion.ops.BinaryCode attribute), 16
- n\_orbitals (openfermion.hamiltonians.MolecularData attribute), 4
- n\_qubits (openfermion.hamiltonians.MolecularData attribute), 4
- n\_qubits (openfermion.ops.BinaryCode attribute), 16
- n\_qubits (openfermion.ops.PolynomialTensor attribute), 22
- name (openfermion.hamiltonians.MolecularData attribute), 3
- normal\_ordered() (in module openfermion.utils), 56
- nuclear\_repulsion (openfermion.hamiltonians.MolecularData attribute), 4
- num\_points (openfermion.utils.Grid attribute), 39
- number\_operator() (in module openfermion.utils), 57
- ## O
- one\_body (openfermion.ops.DiagonalCoulombHamiltonian attribute), 19
- one\_body\_fermion\_constraints() (in module openfermion.measurements), 15
- one\_body\_integrals (openfermion.hamiltonians.MolecularData attribute), 4
- one\_body\_tensor (openfermion.ops.InteractionOperator attribute), 21
- one\_body\_tensor (openfermion.ops.InteractionRDM attribute), 21
- openfermion.hamiltonians (module), 3
- openfermion.measurements (module), 14
- openfermion.ops (module), 16
- openfermion.transforms (module), 29
- openfermion.utils (module), 38
- orbital\_energies (openfermion.hamiltonians.MolecularData attribute), 4
- orbital\_energies() (openfermion.ops.QuadraticHamiltonian method), 25
- orbital\_id() (openfermion.utils.Grid method), 40
- overlap\_integrals (openfermion.hamiltonians.MolecularData attribute), 4
- ## P
- ParallelLinearQubitOperator (class in openfermion.utils), 41
- parity\_code() (in module openfermion.transforms), 34
- pauli\_exp\_to\_qasm() (in module openfermion.utils), 57
- plane\_wave\_external\_potential() (in module openfermion.hamiltonians), 13
- plane\_wave\_hamiltonian() (in module openfermion.hamiltonians), 13
- plane\_wave\_kinetic() (in module openfermion.hamiltonians), 14
- plane\_wave\_potential() (in module openfermion.hamiltonians), 14
- PolynomialTensor (class in openfermion.ops), 22
- position\_vector() (openfermion.utils.Grid method), 40
- prepare\_one\_body\_squared\_evolution() (in module openfermion.utils), 58
- preprocess\_lcu\_coefficients\_for\_reversible\_sampling() (in module openfermion.utils), 58
- project\_onto\_sector() (in module openfermion.transforms), 34
- projection\_error() (in module openfermion.transforms), 34
- protons (openfermion.hamiltonians.MolecularData attribute), 4
- prune\_unused\_indices() (in module openfermion.utils), 59

## Q

QuadOperator (class in openfermion.ops), 22  
 QuadraticHamiltonian (class in openfermion.ops), 23  
 qubit\_operator\_sparse() (in module openfermion.utils), 59  
 QubitDavidson (class in openfermion.utils), 41  
 QubitOperator (class in openfermion.ops), 26

## R

random\_antisymmetric\_matrix() (in module openfermion.utils), 59  
 random\_diagonal\_coulomb\_hamiltonian() (in module openfermion.utils), 59  
 random\_hermitian\_matrix() (in module openfermion.utils), 59  
 random\_interaction\_operator() (in module openfermion.utils), 59  
 random\_quadratic\_hamiltonian() (in module openfermion.utils), 59  
 random\_unitary\_matrix() (in module openfermion.utils), 59  
 reciprocal\_scale (openfermion.utils.Grid attribute), 39  
 renormalize() (openfermion.ops.QubitOperator method), 27  
 reorder() (in module openfermion.utils), 60  
 reverse\_jordan\_wigner() (in module openfermion.transforms), 34  
 rotate\_basis() (openfermion.ops.PolynomialTensor method), 22  
 rotate\_qubit\_by\_pauli() (in module openfermion.transforms), 35

## S

s\_minus\_operator() (in module openfermion.utils), 60  
 s\_plus\_operator() (in module openfermion.utils), 60  
 s\_squared\_operator() (in module openfermion.utils), 60  
 save() (openfermion.hamiltonians.MolecularData method), 6  
 save\_operator() (in module openfermion.utils), 61  
 scale (openfermion.utils.Grid attribute), 39  
 set\_dimension() (openfermion.utils.DavidsonOptions method), 39  
 shift() (openfermion.ops.BinaryPolynomial method), 18  
 shifts (openfermion.utils.Grid attribute), 39  
 slater\_determinant\_preparation\_circuit() (in module openfermion.utils), 61  
 sparse\_eigenspectrum() (in module openfermion.utils), 61  
 SparseDavidson (class in openfermion.utils), 42  
 sx\_operator() (in module openfermion.utils), 61  
 sy\_operator() (in module openfermion.utils), 62  
 SymbolicOperator (class in openfermion.ops), 27  
 symmetric\_ordering() (in module openfermion.transforms), 35

symmetry\_conserving\_bravyi\_kitaev() (in module openfermion.transforms), 36  
 sz\_operator() (in module openfermion.utils), 62

## T

terms (openfermion.ops.BinaryPolynomial attribute), 17  
 terms (openfermion.ops.SymbolicOperator attribute), 27  
 trotter\_operator\_grouping() (in module openfermion.utils), 62  
 trotterize\_exp\_qubop\_to\_qasm() (in module openfermion.utils), 63  
 two\_body (openfermion.ops.DiagonalCoulombHamiltonian attribute), 19  
 two\_body\_fermion\_constraints() (in module openfermion.measurements), 15  
 two\_body\_integrals (openfermion.hamiltonians.MolecularData attribute), 4  
 two\_body\_tensor (openfermion.ops.InteractionOperator attribute), 21  
 two\_body\_tensor (openfermion.ops.InteractionRDM attribute), 21

## U

uccsd\_convert\_amplitude\_format() (in module openfermion.utils), 63  
 uccsd\_generator() (in module openfermion.utils), 63  
 uccsd\_singlet\_generator() (in module openfermion.utils), 64  
 uccsd\_singlet\_get\_packed\_amplitudes() (in module openfermion.utils), 64  
 uccsd\_singlet\_paramsize() (in module openfermion.utils), 65  
 unique\_iter() (openfermion.ops.InteractionOperator method), 21  
 unlinearize\_term() (in module openfermion.measurements), 15  
 up\_index() (in module openfermion.utils), 65  
 up\_then\_down() (in module openfermion.utils), 65

## V

variance() (in module openfermion.utils), 65  
 verstraete\_cirac\_2d\_square() (in module openfermion.transforms), 36  
 volume (openfermion.utils.Grid attribute), 39  
 volume\_scale() (openfermion.utils.Grid method), 40

## W

weight\_one\_binary\_addressing\_code() (in module openfermion.transforms), 37  
 weight\_one\_segment\_code() (in module openfermion.transforms), 37  
 weight\_two\_segment\_code() (in module openfermion.transforms), 37

weyl\_polynomial\_quantization() (in module openfermion.transforms), 37

wigner\_seitz\_length\_scale() (in module openfermion.hamiltonians), 14

## Z

zero() (openfermion.ops.BinaryPolynomial class method), 18

zero() (openfermion.ops.SymbolicOperator class method), 28