FermiLib Documentation

Release 0.1a3

FermiLib

Sep 07, 2017

Contents

Contents

CHAPTER 1

Tutorial

Getting started with FermiLib

Installing FermiLib requires pip. Make sure that you are using an up-to-date version of it. Then, install FermiLib, by running

```
python -m pip install --pre --user fermilib
```

Alternatively, clone/download this repo (e.g., to your /home directory) and run

```
cd /home/fermilib
python -m pip install --pre --user .
```

This will install FermiLib and all its dependencies automatically. In particular, FermiLib requires ProjectQ . It might be useful to install ProjectQ separately before installing FermiLib as it might require setting some manual options such as, e.g., a C++ compiler. Please follow the ProjectQ installation instructions. FermiLib is compatible with both Python 2 and 3.

Basic FermiLib example

To see a basic example with both fermionic and qubit operators as well as whether the installation worked, try to run the following code.

```
from fermilib.ops import FermionOperator, hermitian_conjugated
from fermilib.transforms import jordan_wigner, bravyi_kitaev
from fermilib.utils import eigenspectrum

# Initialize an operator.
fermion_operator = FermionOperator('2^ 0', 3.17)
fermion_operator += hermitian_conjugated(fermion_operator)
print(fermion_operator)
```

```
# Transform to qubits under the Jordan-Wigner transformation and print its spectrum.
jw_operator = jordan_wigner(fermion_operator)
jw_spectrum = eigenspectrum(jw_operator)
print(jw_operator)
print(jw_spectrum)

# Transform to qubits under the Bravyi-Kitaev transformation and print its spectrum.
bk_operator = bravyi_kitaev(fermion_operator)
bk_spectrum = eigenspectrum(bk_operator)
print(bk_operator)
print(bk_spectrum)
```

This code creates the fermionic operator $a_2^\dagger a_0$ and adds its Hermitian conjugate $a_0^\dagger a_2$ to it. It then maps the resulting fermionic operator to qubit operators using two transforms included in FermiLib, the Jordan-Wigner and Bravyi-Kitaev transforms. Despite the different representations, these operators are iso-spectral. The example also shows some of the intuitive string methods included in FermiLib.

Further examples can be found in the docs (*Examples* in the panel on the left) and in the FermiLib examples folder on GitHub.

Plugins

In order to generate molecular hamiltonians in Gaussian basis sets and perform other complicated electronic structure calculations, one can install plugins. We currently support Psi4 (plugin here, recommended) and PySCF (plugin here).

CHAPTER 2

Examples

All of these examples (and more!) are explained in detail in the ipython notebook fermilib_demo.ipynb located at GitHub.

Fermionic Operators

Fermionic systems are often treated in second quantization, where arbitrary operators can be expressed using the fermionic creation and annihilation operators, $a_k^\dagger$ and a_k. Any weighted sum of products of these operators can be represented with the FermionOperator data structure in FermiLib.

```
from fermilib.ops import FermionOperator

my_term = FermionOperator(((3, 1), (1, 0)))
print(my_term)

my_term = FermionOperator('3^ 1')
print(my_term)
```

These two examples yield the same fermionic operator, $a_3^\dagger a_1$.

The preferred way to specify the coefficient in FermiLib is to provide an optional coefficient argument. If not provided, the coefficient defaults to 1. In the code below, the first method is preferred. The multiplication in the last method actually creates a copy of the term, which introduces some additional cost. All inplace operands (such as +=) modify classes whereas binary operands such as + create copies. Important caveats are that the empty tuple FermionOperator(()) and the empty string FermionOperator('') initialize identity. The empty initializer FermionOperator() initializes the zero operator. We demonstrate some of these below.

```
from fermilib.ops import FermionOperator

good_way_to_initialize = FermionOperator('3^ 1', -1.7)
print(good_way_to_initialize)

bad_way_to_initialize = -1.7 * FermionOperator('3^ 1')
```

```
print(bad_way_to_initialize)

identity = FermionOperator('')
print(identity)

zero_operator = FermionOperator()
print(zero_operator)
```

This creates the previous FermionOperator with a coefficient -1.7, as well as the identity and zero operators.

FermionOperator has only one attribute: .terms. This attribute is the dictionary which stores the term tuples.

```
from fermilib.ops import FermionOperator

my_operator = FermionOperator('4^ 1^ 3 9', 1. + 2.j)
print(my_operator)
print(my_operator.terms)
```

FermionOperator supports a wide range of builtins including str(), repr(), =, , /, /=, +, +=, -, -=, - and **. Note that instead of supporting != and ==, we have the method .isclose(), since FermionOperators involve floats.

Qubit Operators

The QubitOperator data structure is another essential part of FermiLib. While the QubitOperator was originally developed for FermiLib, it is now part of the core ProjectQ library so that it can be interpreted by the ProjectQ compiler using the TimeEvolution gate. As the name suggests, QubitOperator is used to store qubit operators in almost exactly the same way that FermionOperator is used to store fermion operators. For instance $X_0 Z_3 Y_4$ is a QubitOperator. The internal representation of this as a terms tuple would be $((0, X), (3, Z), (4, Y))$. Note that one important difference between QubitOperator and FermionOperator is that the terms in QubitOperator are always sorted in order of tensor factor. In some cases, this enables faster manipulation. We initialize some QubitOperators below.

```
from projectqtemp.ops import QubitOperator

my_first_qubit_operator = QubitOperator('X1 Y2 Z3')
print(my_first_qubit_operator)
print(my_first_qubit_operator.terms)

operator_2 = QubitOperator('X3 Z4', 3.17)
operator_2 -= 77. * my_first_qubit_operator
print(operator_2)
```

Transformations

FermiLib also provides functions for mapping FermionOperators to QubitOperators, including the Jordan-Wigner and Bravyi-Kitaev transforms.

```
from fermilib.ops import FermionOperator, hermitian_conjugated
from fermilib.transforms import jordan_wigner, bravyi_kitaev
from fermilib.utils import eigenspectrum

# Initialize an operator.
fermion_operator = FermionOperator('2^ 0', 3.17)
fermion_operator += hermitian_conjugated(fermion_operator)
```

```
print(fermion_operator)

# Transform to qubits under the Jordan-Wigner transformation and print its spectrum.
jw_operator = jordan_wigner(fermion_operator)
jw_spectrum = eigenspectrum(jw_operator)
print(jw_operator)
print(jw_spectrum)

# Transform to qubits under the Bravyi-Kitaev transformation and print its spectrum.
bk_operator = bravyi_kitaev(fermion_operator)
bk_spectrum = eigenspectrum(bk_operator)
print(bk_operator)
print(bk_spectrum)
```

We see that despite the different representation, these operators are iso-spectral. We can also apply the Jordan-Wigner transform in reverse to map arbitrary QubitOperators to FermionOperators. Note that we also demonstrate the .compress() method (a method on both FermionOperators and QubitOperators) which removes zero entries.

```
from projectqtemp.ops import QubitOperator
from fermilib.transforms import jordan_wigner, reverse_jordan_wigner

# Initialize QubitOperator.
my_operator = QubitOperator('X0 Y1 Z2', 88.)
my_operator += QubitOperator('Z1 Z4', 3.17)
print(my_operator)

# Map QubitOperator to a FermionOperator.
mapped_operator = reverse_jordan_wigner(my_operator)
print(mapped_operator)

# Map the operator back to qubits and make sure it is the same.
back_to_normal = jordan_wigner(mapped_operator)
back_to_normal.compress()
print(back_to_normal)
```

Sparse matrices and the Hubbard model

Often, one would like to obtain a sparse matrix representation of an operator which can be analyzed numerically. There is code in both fermilib.transforms and fermilib.utils which facilitates this. The function get_sparse_operator converts either a FermionOperator, a QubitOperator or other more advanced classes such as InteractionOperator to a scipy.sparse.csc matrix. There are numerous functions in fermilib.utils which one can call on the sparse operators such as "get_gap", "get_hartree_fock_state", "get_ground_state", ect. We show this off by computing the ground state energy of the Hubbard model. To do that, we use code from the fermilib.utils module which constructs lattice models of fermions such as Hubbard models.

```
from fermilib.transforms import get_sparse_operator, jordan_wigner
from fermilib.utils import fermi_hubbard, get_ground_state

# Set model.
x_dimension = 2
y_dimension = 2
tunneling = 2.
coulomb = 1.
magnetic_field = 0.5
chemical_potential = 0.25
```

```
periodic = 1
spinless = 1

# Get fermion operator.
hubbard_model = fermi_hubbard(
    x_dimension, y_dimension, tunneling, coulomb, chemical_potential,
    magnetic_field, periodic, spinless)
print(hubbard_model)

# Get qubit operator under Jordan-Wigner.
jw_hamiltonian = jordan_wigner(hubbard_model)
jw_hamiltonian.compress()
print(jw_hamiltonian)

# Get scipy.sparse.csc representation.
sparse_operator = get_sparse_operator(hubbard_model)
print(sparse_operator)
print('\nEnergy of the model is {} in units of T and J.'.format(
    get_ground_state(sparse_operator)[0]))
```

Basics of MolecularData class

Data from electronic structure calculations can be saved in a FermiLib data structure called MolecularData, which makes it easy to access within our library. Often, one would like to analyze a chemical series or look at many different Hamiltonians and sometimes the electronic structure calculations are either expensive to compute or difficult to converge (e.g. one needs to mess around with different types of SCF routines to make things converge). Accordingly, we anticipate that users will want some way to automatically database the results of their electronic structure calculations so that important data (such as the SCF intergrals) can be looked up on-the-fly if the user has computed them in the past. FermiLib supports a data provenance strategy which saves key results of the electronic structure calculation (including pointers to files containing large amounts of data, such as the molecular integrals) in an HDF5 container.

The MolecularData class stores information about molecules. One initializes a MolecularData object by specifying parameters of a molecule such as its geometry, basis, multiplicity, charge and an optional string describing it. One can also initialize MolecularData simply by providing a string giving a filename where a previous MolecularData object was saved in an HDF5 container. One can save a MolecularData instance by calling the class's .save() method. This automatically saves the instance in a data folder specified during FermiLib installation. The name of the file is generated automatically from the instance attributes and optionally provided description. Alternatively, a filename can also be provided as an optional input if one wishes to manually name the file.

When electronic structure calculations are run, the data files for the molecule can be automatically updated. If one wishes to later use that data they either initialize MolecularData with the instance filename or initialize the instance and then later call the .load() method.

Basis functions are provided to initialization using a string such as "6-31g". Geometries can be specified using a simple txt input file (see geometry_from_file function in molecular_data.py) or can be passed using a simple python list format demonstrated below. Atoms are specified using a string for their atomic symbol. Distances should be provided in atomic units (Bohr). Below we initialize a simple instance of MolecularData without performing any electronic structure calculations.

```
from fermilib.utils import MolecularData

# Set parameters to make a simple molecule.
diatomic_bond_length = .7414
geometry = [('H', (0., 0., 0.)), ('H', (0., 0., diatomic_bond_length))]
basis = 'sto-3g'
```

```
multiplicity = 1
charge = 0
description = str(diatomic_bond_length)

# Make molecule and print out a few interesting facts about it.
molecule = MolecularData(geometry, basis, multiplicity,
                         charge, description)
print('Molecule has automatically generated name {}'.format(
    molecule.name))
print('Information about this molecule would be saved at:\n{}\n'.format(
    molecule.filename))
print('This molecule has {} atoms and {} electrons.'.format(
    molecule.n_atoms, molecule.n_electrons))
for atom, atomic_number in zip(molecule.atoms, molecule.protons):
    print('Contains {} atom, which has {} protons.'.format(
        atom, atomic_number))
```

If we had previously computed this molecule using an electronic structure package, we can call molecule.load() to populate all sorts of interesting fields in the data structure. Though we make no assumptions about what electronic structure packages users might install, we assume that the calculations are saved in Fermilib's MolecularData objects. There may be plugins available in future. For the purposes of this example, we will load data that ships with FermiLib to make a plot of the energy surface of hydrogen. Note that helper functions to initialize some interesting chemical benchmarks are found in fermilib.utils.

```
# Set molecule parameters.
basis = 'sto-3g'
multiplicity = 1
bond_length_interval = 0.1
n_points = 25

# Generate molecule at different bond lengths.
hf_energies = []
fci_energies = []
bond_lengths = []
for point in range(3, n_points + 1):
    bond_length = bond_length_interval * point
    bond_lengths += [bond_length]
    description = str(round(bond_length,2))
    print(description)
    geometry = [('H', (0., 0., 0.)), ('H', (0., 0., bond_length))]
    molecule = MolecularData(
        geometry, basis, multiplicity, description=description)

    # Load data.
    molecule.load()

    # Print out some results of calculation.
    print('\nAt bond length of {} Bohr, molecular hydrogen has:'.format(
        bond_length))
    print('Hartree-Fock energy of {} Hartree.'.format(molecule.hf_energy))
    print('MP2 energy of {} Hartree.'.format(molecule.mp2_energy))
    print('FCI energy of {} Hartree.'.format(molecule.fci_energy))
    print('Nuclear repulsion energy between protons is {} Hartree.'.format(
        molecule.nuclear_repulsion))
    for orbital in range(molecule.n_orbitals):
        print('Spatial orbital {} has energy of {} Hartree.'.format(
            orbital, molecule.orbital_energies[orbital]))
```

```
    hf_energies += [molecule.hf_energy]
    fci_energies += [molecule.fci_energy]
```

InteractionOperator and InteractionRDM for efficient numerical representations

Fermion Hamiltonians can be expressed as $H = h_0 + \sum_{pq} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$, where h_0 is a constant shift due to the nuclear repulsion and h_{pq} and h_{pqrs} are the famous molecular integrals. Since fermions interact pairwise, their energy is thus a unique function of the one-particle and two-particle reduced density matrices which are expressed in second quantization as $\rho_{pq} = \langle p|a_p^\dagger a_q|q\rangle$ and $\rho_{pqrs} = \langle pq|a_p^\dagger a_q^\dagger a_r a_s|rs\rangle$, respectively.

Because the RDMs and molecular Hamiltonians are both compactly represented and manipulated as 2- and 4- index tensors, we can represent them in a particularly efficient form using similar data structures. The InteractionOperator data structure can be initialized for a Hamiltonian by passing the constant h0h0 (or 0), as well as numpy arrays representing h_{pq} (or ρ_{pq}) and h_{pqrs} (or ρ_{pqrs}). Importantly, InteractionOperators can also be obtained by calling MolecularData.get_molecular_hamiltonian() or by calling the function get_interaction_operator() (found in fermilib.utils) on a FermionOperator. The InteractionRDM data structure is similar but represents RDMs. For instance, one can get a molecular RDM by calling MolecularData.get_molecular_rdm(). When generating Hamiltonians from the MolecularData class, one can choose to restrict the system to an active space.

These classes inherit from the same base class, InteractionTensor. This data structure overloads the slice operator [] so that one can get or set the key attributes of the InteractionOperator: .constant, .one_body_coefficients and .two_body_coefficients. For instance, InteractionOperator[p,q,r,s] would return h_{pqrs} and InteractionRDM would return ρ_{pqrs}. Importantly, the class supports fast basis transformations using the method InteractionTensor.rotate_basis(rotation_matrix). But perhaps most importantly, one can map the InteractionOperator to any of the other data structures we've described here.

Below, we load MolecularData from a saved calculation of LiH. We then obtain an InteractionOperator representation of this system in an active space. We then map that operator to qubits. We then demonstrate that one can rotate the orbital basis of the InteractionOperator using random angles to obtain a totally different operator that is still isospectral.

```
from fermilib.transforms import get_fermion_operator, get_sparse_operator, jordan_
↪wigner
from fermilib.utils import get_ground_state, MolecularData
import numpy
import scipy
import scipy.linalg

# Load saved file for LiH.
diatomic_bond_length = 1.45
geometry = [('Li', (0., 0., 0.)), ('H', (0., 0., diatomic_bond_length))]
basis = 'sto-3g'
multiplicity = 1

# Set Hamiltonian parameters.
active_space_start = 1
active_space_stop = 3

# Generate and populate instance of MolecularData.
molecule = MolecularData(geometry, basis, multiplicity, description=\"1.45\")
molecule.load()

# Get the Hamiltonian in an active space.
```

```
molecular_hamiltonian = molecule.get_molecular_hamiltonian(
    occupied_indices=range(active_space_start),
    active_indices=range(active_space_start, active_space_stop))

# Map operator to fermions and qubits.
fermion_hamiltonian = get_fermion_operator(molecular_hamiltonian)
qubit_hamiltonian = jordan_wigner(fermion_hamiltonian)
qubit_hamiltonian.compress()
print('The Jordan-Wigner Hamiltonian in canonical basis follows:\n{}'.format(qubit_
↪hamiltonian))

# Get sparse operator and ground state energy.
sparse_hamiltonian = get_sparse_operator(qubit_hamiltonian)
energy, state = get_ground_state(sparse_hamiltonian)
print('Ground state energy before rotation is {} Hartree.\n'.format(energy))

# Randomly rotate.
n_orbitals = molecular_hamiltonian.n_qubits // 2
n_variables = int(n_orbitals * (n_orbitals - 1) / 2)
random_angles = numpy.pi * (1. - 2. * numpy.random.rand(n_variables))
kappa = numpy.zeros((n_orbitals, n_orbitals))
index = 0
for p in range(n_orbitals):
    for q in range(p + 1, n_orbitals):
        kappa[p, q] = random_angles[index]
        kappa[q, p] = -numpy.conjugate(random_angles[index])
        index += 1

    # Build the unitary rotation matrix.
    difference_matrix = kappa + kappa.transpose()
    rotation_matrix = scipy.linalg.expm(kappa)

    # Apply the unitary.
    molecular_hamiltonian.rotate_basis(rotation_matrix)

# Get qubit Hamiltonian in rotated basis.
qubit_hamiltonian = jordan_wigner(molecular_hamiltonian)
qubit_hamiltonian.compress()
print('The Jordan-Wigner Hamiltonian in rotated basis follows:\n{}'.format(qubit_
↪hamiltonian))

# Get sparse Hamiltonian and energy in rotated basis.
sparse_hamiltonian = get_sparse_operator(qubit_hamiltonian)
energy, state = get_ground_state(sparse_hamiltonian)
print('Ground state energy after rotation is {} Hartree.'.format(energy))
```

Simulating a variational quantum eigensolver using ProjectQ

We now demonstrate how one can use both FermiLib and ProjectQ to run a simple VQE example using a Unitary Coupled Cluster ansatz. It demonstrates a simple way to evaluate the energy, optimize the energy with respect to the ansatz and build the corresponding compiled quantum circuit. It utilizes ProjectQ to build and simulate the circuit.

```
from numpy import array, concatenate, zeros
from numpy.random import randn
from scipy.optimize import minimize
```

```
from fermilib.config import *
from fermilib.utils import *
from fermilib.transforms import jordan_wigner

from projectq.ops import X, All, Measure
from projectq.backends import CommandPrinter, CircuitDrawer
```

Here we load H_2 from a precomputed molecule file found in the test data directory, and initialize the ProjectQ circuit compiler to a standard setting that uses a first-order Trotter decomposition to break up the exponentials of non-commuting operators.

```
# Load the molecule.
import os
filename = os.path.join(DATA_DIRECTORY, 'H2_sto-3g_singlet_0.7414')
molecule = MolecularData(filename=filename)

# Use a Jordan-Wigner encoding, and compress to remove 0 imaginary components
qubit_hamiltonian = jordan_wigner(molecule.get_molecular_hamiltonian())
qubit_hamiltonian.compress()
compiler_engine = uccsd_trotter_engine()
```

The Variational Quantum Eigensolver (or VQE), works by parameterizing a wavefunction $|\Psi(\theta)\rangle$ through some quantum circuit, and minimzing the energy with respect to that angle, which is defined by

$$E(\theta) = \langle\Psi(\theta)|H|\Psi(\theta)\rangle$$

To perform the VQE loop with a simple molecule, it helps to wrap the evaluation of the energy into a simple objective function that takes the parameters of the circuit and returns the energy. Here we define that function using ProjectQ to handle the qubits and the simulation.

```
def energy_objective(packed_amplitudes):
    """Evaluate the energy of a UCCSD singlet wavefunction with packed_amplitudes
    Args:
        packed_amplitudes(ndarray): Compact array that stores the unique
            amplitudes for a UCCSD singlet wavefunction.

    Returns:
        energy(float): Energy corresponding to the given amplitudes
    """
    # Set Jordan-Wigner initial state with correct number of electrons
    wavefunction = compiler_engine.allocate_qureg(molecule.n_qubits)
    for i in range(molecule.n_electrons):
        X | wavefunction[i]

    # Build the circuit and act it on the wavefunction
    evolution_operator = uccsd_singlet_evolution(packed_amplitudes,
                                                 molecule.n_qubits,
                                                 molecule.n_electrons)
    evolution_operator | wavefunction
    compiler_engine.flush()

    # Evaluate the energy and reset wavefunction
    energy = compiler_engine.backend.get_expectation_value(qubit_hamiltonian,␣
↪wavefunction)
    All(Measure) | wavefunction
    compiler_engine.flush()
    return energy
```

While we could plug this objective function into any optimizer, SciPy offers a convenient framework within the Python ecosystem. We'll choose as starting amplitudes the classical CCSD values that can be loaded from the molecule if desired. The optimal energy is found and compared to the exact values to verify that our simulation was successful.

```
n_amplitudes = uccsd_singlet_paramsize(molecule.n_qubits, molecule.n_electrons)
initial_amplitudes = [0, 0.05677]
initial_energy = energy_objective(initial_amplitudes)

# Run VQE Optimization to find new CCSD parameters
opt_result = minimize(energy_objective, initial_amplitudes,
                      method="CG", options={'disp':True})

opt_energy, opt_amplitudes = opt_result.fun, opt_result.x
print("\nOptimal UCCSD Singlet Energy: {}".format(opt_energy))
print("Optimal UCCSD Singlet Amplitudes: {}".format(opt_amplitudes))
print("Classical CCSD Energy: {} Hartrees".format(molecule.ccsd_energy))
print("Exact FCI Energy: {} Hartrees".format(molecule.fci_energy))
print("Initial Energy of UCCSD with CCSD amplitudes: {} Hartrees".format(initial_
↪energy))
```

As we can see, the optimization terminates extremely quickly because the classical coupled cluster amplitudes were (for this molecule) already optimal. We can now use ProjectQ to compile this simulation circuit to a set of two-body quanutm gates.

```
compiler_engine = uccsd_trotter_engine(CommandPrinter())
wavefunction = compiler_engine.allocate_qureg(molecule.n_qubits)
for i in range(molecule.n_electrons):
    X | wavefunction[i]

# Build the circuit and act it on the wavefunction
evolution_operator = uccsd_singlet_evolution(opt_amplitudes,
                                             molecule.n_qubits,
                                             molecule.n_electrons)
evolution_operator | wavefunction
compiler_engine.flush()
```

For more, see the GitHub examples.

CHAPTER 3

Code Documentation

fermilib.transforms

fermilib.transforms.**bravyi_kitaev**(*operator*, *n_qubits=None*)
Apply the Bravyi-Kitaev transform and return qubit operator.

Parameters

- **operator** (fermilib.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 An instance of the QubitOperator class.

Return type transformed_operator

Raises ValueError – Invalid number of qubits specified.

fermilib.transforms.**get_fermion_operator**(*interaction_operator*)
Output InteractionOperator as instance of FermionOperator class.

Returns An instance of the FermionOperator class.

Return type fermion_operator

fermilib.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 An instance of the InteractionOperator class.

Return type interaction_operator

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.

`fermilib.transforms.`**`get_interaction_rdm`**(*qubit_operator*, *n_qubits=None*)
Build an InteractionRDM from measured qubit operators.

Returns: An InteractionRDM object.

`fermilib.transforms.`**`get_sparse_operator`**(*operator*, *n_qubits=None*)
Map a Fermion, Qubit, or InteractionOperator to a SparseOperator.

`fermilib.transforms.`**`jordan_wigner`**(*operator*)
Apply the Jordan-Wigner transform to a FermionOperator or InteractionOperator to convert to a QubitOperator.

Returns An instance of the QubitOperator class.

Return type transformed_operator

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

`fermilib.transforms.`**`reverse_jordan_wigner`**(*qubit_operator*, *n_qubits=None*)
Transforms a QubitOperator into a FermionOperator using the Jordan-Wigner transform.

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

Parameters

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

Returns An instance of the FermionOperator class.

Return type transformed_term

Raises

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

fermilib.ops

class `fermilib.ops.`**`FermionOperator`**(*term=None*, *coefficient=1.0*)
FermionOperator stores a sum of products of fermionic ladder operators.

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

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

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

terms
dict – **key** (tuple of tuples): Each tuple represents a fermion term, i.e. a tensor product of fermion ladder operators with a coefficient. The first element is an integer indicating the mode on which a ladder operator acts and the second element is a bool, either '0' indicating annihilation, or '1' indicating creation in that mode; for example, '2^ 5' is ((2, 1), (5, 0)). **value** (complex float): The coefficient of term represented by key.

__init__(*term=None*, *coefficient=1.0*)
Initializes a FermionOperator.

The init function only allows to initialize a FermionOperator consisting of a single term. If one desires to initialize a FermionOperator consisting of many terms, one must add those terms together by using either += (which is fast) or using +.

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 terms to FermionOperator is faster using += (as this is done by in-place addition). Specifying the coefficient in the __init__ is faster than by multiplying a QubitOperator with a scalar as calls an out-of-place multiplication.

Parameters

- **term** (*tuple of tuples, a string, or optional*) –
  1. A tuple of tuples. The first element of each tuple is an integer indicating the mode on which a fermion ladder operator acts, starting from zero. The second element of each tuple is an integer, either 1 or 0, indicating whether creation or annihilation acts on that mode.
  2. A string of the form '0^ 2', indicating creation in mode 0 and annihilation in mode 2.
  3. default will result in the zero operator.
- **coefficient** (*complex float, optional*) – The coefficient of the term. Default value is 1.0.

Raises FermionOperatorError – Invalid term provided to FermionOperator.

compress(*abs_tol=1e-12*)
Eliminates all terms with coefficients close to zero and removes imaginary parts of coefficients that are close to zero.

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

static identity()

Returns A fermion operator u with the property that u*x = x*u = x for all fermion operators x.

Return type multiplicative_identity (*FermionOperator*)

is_molecular_term()
Query whether term has correct form to be from a molecular.

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).

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.

isclose(*other*, *rel_tol=1e-12*, *abs_tol=1e-12*)
Returns True if other (FermionOperator) is close to self.

Comparison is done for each term individually. Return True if the difference between each terms in self and other is less than the relative tolerance w.r.t. either other or self (symmetric test) or if the difference is less than the absolute tolerance.

Parameters

- **other** (FermionOperator) – FermionOperator to compare against.
- **rel_tol** (*float*) – Relative tolerance, must be greater than 0.0
- **abs_tol** (*float*) – Absolute tolerance, must be at least 0.0

static zero()

Returns A fermion operator o with the property that o+x = x+o = x for all fermion operators x.

Return type additive_identity (*FermionOperator*)

class fermilib.ops.**InteractionOperator**(*constant*, *one_body_tensor*, *two_body_tensor*)
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.

n_qubits
An int giving the number of qubits.

constant
A constant term in the operator given as a float. For instance, the nuclear repulsion energy.

one_body_tensor
The coefficients of the one-body terms (h[p, q]). This is an n_qubits x n_qubits numpy array of floats.

two_body_tensor
The coefficients of the two-body terms (h[p, q, r, s]). This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array of floats.

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

Parameters

- **constant** – A constant term in the operator given as a float. For instance, the nuclear repulsion energy.
- **one_body_tensor** – The coefficients of the one-body terms (h[p,q]). This is an n_qubits x n_qubits numpy array of floats.
- **two_body_tensor** – The coefficients of the two-body terms (h[p, q, r, s]). This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array of floats.

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 fermilib.ops.**InteractionRDM**(*one_body_tensor*, *two_body_tensor*)
Class for storing 1- and 2-body reduced density matrices.

one_body_tensor
The expectation values <a^dagger_p a_q>.

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

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

Parameters

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

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

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

Returns Expectation value

Return type float

Raises InteractionRDMError – Invalid operator provided.

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

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

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

Return type QubitOperator

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

class `fermilib.ops.`**`InteractionTensor`**(*constant*, *one_body_tensor*, *two_body_tensor*)

Class for storing data about the interactions between orbitals. Because electrons interact pairwise, in second-quantization, all Hamiltonian terms have either the form of a^dagger_p a_q or a^dagger_p a^dagger_q a_r a_s. The first of these terms is associated with the one-body Hamiltonian and 1-RDM and its information is stored in one_body_tensor. The second of these terms is associated with the two-body Hamiltonian and 2-RDM and its information is stored in two_body_tensor. Much of the functionality of this class is redudant with FermionOperator but enables much more efficient numerical computations in many cases, such as basis rotations.

`n_qubits`

The number of qubits on which the tensor acts.

`constant`

A constant term in the operator given as a float. For instance, the nuclear repulsion energy.

`one_body_tensor`

The coefficients of the 2D matrix terms. This is an n_qubits x n_qubits numpy array of floats. For instance, the one body term of MolecularOperator.

`two_body_tensor`

The coefficients of the 4D matrix terms. This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array offloats. For instance, the two body term of MolecularOperator.

`__init__`(*constant*, *one_body_tensor*, *two_body_tensor*)

Initialize the InteractionTensor 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 2D matrix terms. This is an n_qubits x n_qubits numpy array of floats. For instance, the one body term of MolecularOperator.
- **`two_body_tensor`** – The coefficients of the 4D matrix terms. This is an n_qubits x n_qubits x n_qubits x n_qubits numpy array of floats. For instance, the two body term of MolecularOperator.

`rotate_basis`(*rotation_matrix*)

Rotate the orbital basis of the InteractionTensor.

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

`fermilib.ops.`**`hermitian_conjugated`**(*fermion_operator*)

Return Hermitian conjugate of fermionic operator.

`fermilib.ops.`**`normal_ordered`**(*fermion_operator*)

Compute and return the normal ordered form of a FermionOperator.

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

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

fermilib.ops.**number_operator**(*n_orbitals*, *orbital=None*, *coefficient=1.0*)
Return a number operator.

Parameters

- **n_orbitals** (*int*) – The number of spin-orbitals in the system.
- **orbital** (*int, optional*) – The orbital on which to return the number operator. If None, return total number operator on all sites.
- **coefficient** (*float*) – The coefficient of the term.

Returns operator (FermionOperator)

fermilib.utils

class fermilib.utils.**Grid**(*dimensions*, *length*, *scale*)
A multi-dimensional grid of points.

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

Parameters

- **dimensions** (*int*) – The number of dimensions the grid lives in.
- **length** (*int*) – The number of points along each grid axis.
- **scale** (*float*) – The total length of each grid dimension.

all_points_indices()

Returns The index-coordinate tuple of each point in the grid.

Return type iterable[tuple[int]]

num_points()

Returns The number of points in the grid.

Return type int

volume_scale()

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

Return type float

class fermilib.utils.**MolecularData**(*geometry=None*, *basis=None*, *multiplicity=None*, *charge=0*, *description=u''*, *filename=u''*, *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 atomic units. 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.

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

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

Parameters

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

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

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

Parameters

- **occupied_indices** (*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

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.

Return type tuple

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

Parameters **property_name** (*string*) – 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

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).

Return type *one_body_integrals*

Raises MisissingCalculationError – If SCF calculation has not been performed.

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

Parameters

- **rotation_matrix** – A square numpy array or matrix having dimensions of n_orbitals by n_orbitals. Assumed real and invertible.
- **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 An instance of the MolecularOperator class.

Return type molecular_hamiltonian

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 An instance of the MolecularRDM class.

Return type rdm

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

get_n_alpha_electrons()
Return number of alpha electrons.

get_n_beta_electrons()
Return number of beta electrons.

init_lazy_properties()
Initializes properties loaded on demand to None

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

fermilib.utils.**commutator**(*operator_a*, *operator_b*)
Compute the commutator of two QubitOperators or FermionOperators.

Parameters **operator_b** (*operator_a,*) – Operators in commutator.

fermilib.utils.**count_qubits**(*operator*)
Compute the minimum number of qubits on which operator acts.

Parameters **operator** – QubitOperator, InteractionOperator, FermionOperator, InteractionTensor, or InteractionRDM.

Returns The minimum number of qubits on which operator acts.

Return type *n_qubits* (int)

Raises TypeError – Operator of invalid type.

fermilib.utils.**dual_basis_error_bound**(*terms*, *indices=None*, *is_hopping_operator=None*, *jellium_only=False*, *verbose=False*)
Numerically upper bound the error in the ground state energy for the second-order Trotter-Suzuki expansion.

Parameters

- **terms** – a list of single-term FermionOperators in the Hamiltonian to be simulated.
- **indices** – a set of indices the terms act on in the same order as terms.
- **is_hopping_operator** – a list of whether each term is a hopping operator.
- **jellium_only** – Whether the terms are from the jellium Hamiltonian only, rather than the full dual basis Hamiltonian (i.e. whether c_i = c for all number operators i^ i, or whether they depend on i as is possible in the general case).
- **verbose** – Whether to print percentage progress.

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

Notes

Follows Equation 9 of Poulin et al.'s work in "The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry" to calculate the error operator.

fermilib.utils.**dual_basis_error_operator**(*terms*, *indices=None*, *is_hopping_operator=None*, *jellium_only=False*, *verbose=False*)
Determine the difference between the exact generator of unitary evolution and the approximate generator given by the second-order Trotter-Suzuki expansion.

Parameters

- **terms** – a list of FermionOperators in the Hamiltonian in the order in which they will be simulated.
- **indices** – a set of indices the terms act on in the same order as terms.
- **is_hopping_operator** – a list of whether each term is a hopping operator.

- **jellium_only** – Whether the terms are from the jellium Hamiltonian only, rather than the full dual basis Hamiltonian (i.e. whether c_i = c for all number operators i^ i, or whether they depend on i as is possible in the general case).
- **verbose** – Whether to print percentage progress.

Returns

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

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

fermilib.utils.**dual_basis_external_potential**(*grid*, *geometry*, *spinless*)
Return the external potential in the dual basis of arXiv:1706.00023.

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.

Returns The dual basis operator.

Return type *FermionOperator*

fermilib.utils.**dual_basis_jellium_model**(*grid*, *spinless=False*, *kinetic=True*, *potential=True*, *include_constant=False*)
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.

Returns operator (FermionOperator)

fermilib.utils.**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)

fermilib.utils.**dual_basis_potential**(*grid*, *spinless=False*)
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.

Returns operator (FermionOperator)

fermilib.utils.**eigenspectrum**(*operator*)

Compute the eigenspectrum of an operator.

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

Parameters **operator** – QubitOperator, InteractionOperator, FermionOperator, InteractionTensor, or InteractionRDM.

Returns dense numpy array of floats giving eigenspectrum.

Return type *eigenspectrum*

fermilib.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.

fermilib.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".

fermilib.utils.**expectation**(*sparse_operator*, *state*)

Compute expectation value of operator with a state.

Parameters **state** – scipy.sparse.csc vector representing a pure state, or, a scipy.sparse.csc matrix representing a density matrix.

Returns A real float giving expectation value.

Raises `ValueError` – Input state has invalid format.

`fermilib.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.

`fermilib.utils.`**`fermi_hubbard`**(*x_dimension*, *y_dimension*, *tunneling*, *coulomb*, *chemical_potential=None*, *magnetic_field=None*, *periodic=True*, *spinless=False*)

Return symbolic representation of a Fermi-Hubbard Hamiltonian.

Parameters

- **`x_dimension`** – An integer giving the number of sites in width.
- **`y_dimension`** – An integer giving the number of sites in height.
- **`tunneling`** – A float giving the tunneling amplitude.
- **`coulomb`** – A float giving the attractive local interaction strength.
- **`chemical_potential`** – An optional float giving the potential of each site. Default value is None.
- **`magnetic_field`** – An optional float giving a magnetic field at each site. Default value is None.
- **`periodic`** – If True, add periodic boundary conditions.
- **`spinless`** – An optional Boolean. If False, each site has spin up orbitals and spin down orbitals. If True, return a spinless Fermi-Hubbard model.
- **`verbose`** – An optional Boolean. If True, print all second quantized terms.

Returns An instance of the FermionOperator class.

Return type hubbard_model

`fermilib.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 The fourier-transformed hamiltonian.

Return type *FermionOperator*

fermilib.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.

Return type file_path (string)

Raises OperatorUtilsError – File name is not provided.

fermilib.utils.**get_gap**(*sparse_operator*)
Compute gap between lowest eigenvalue and first excited state.

Returns: A real float giving eigenvalue gap.

fermilib.utils.**get_ground_state**(*sparse_operator*)
Compute lowest eigenvalue and eigenstate.

Returns The lowest eigenvalue, a float. eigenstate: The lowest eigenstate in scipy.sparse csc format.

Return type eigenvalue

fermilib.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 The inverse-fourier-transformed hamiltonian.

Return type *FermionOperator*

fermilib.utils.**is_hermitian**(*sparse_operator*)
Test if matrix is Hermitian.

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

Parameters **operator** – QubitOperator or FermionOperator.

Raises TypeError – Operator of invalid type.

fermilib.utils.**jellium_model**(*grid*, *spinless=False*, *plane_wave=True*, *include_constant=False*)
Return jellium Hamiltonian as FermionOperator class.

Parameters

- **grid** (fermilib.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.

Returns The Hamiltonian of the model.

Return type *FermionOperator*

fermilib.utils.**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)

fermilib.utils.**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.

Returns hamiltonian (QubitOperator)

fermilib.utils.**jordan_wigner_sparse**(*fermion_operator*, *n_qubits=None*)

Initialize a SparseOperator from a FermionOperator.

Parameters

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

Returns The corresponding SparseOperator.

fermilib.utils.**jw_hartree_fock_state**(*n_electrons*, *n_orbitals*)

Function to product Hartree-Fock state in JW representation.

fermilib.utils.**load_operator**(*file_name=None*, *data_directory=None*)

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.

Returns The stored FermionOperator or QubitOperator

Return type operator

Raises TypeError – Operator of invalid type.

fermilib.utils.**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 An instance of the MolecularData class.

Return type atom

fermilib.utils.**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 A an instance of the MolecularData class.

Return type molecule

Raises `MolecularLatticeError` – If lattice specification is invalid.

fermilib.utils.**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 A an instance of the MolecularData class.

Return type molecule

fermilib.utils.**plane_wave_external_potential**(*grid*, *geometry*, *spinless*)
Return the external potential operator in plane wave basis.

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.

Returns The plane wave operator.

Return type *FermionOperator*

fermilib.utils.**plane_wave_hamiltonian**(*grid*, *geometry=None*, *spinless=False*, *plane_wave=True*, *include_constant=False*)
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.

Returns The hamiltonian.

Return type *FermionOperator*

fermilib.utils.**plane_wave_kinetic**(*grid*, *spinless=False*)
Return the kinetic energy operator in the plane wave basis.

Parameters

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

Returns The kinetic momentum operator.

Return type *FermionOperator*

fermilib.utils.**plane_wave_potential**(*grid*, *spinless=False*)
Return the potential operator in the plane wave basis.

Parameters

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

Returns operator (FermionOperator)

fermilib.utils.**qubit_operator_sparse**(*qubit_operator*, *n_qubits=None*)
Initialize a SparseOperator from a QubitOperator.

Parameters

- **qubit_operator** (*QubitOperator*) – instance of the QubitOperator class.

- **n_qubits** (*int*) – Number of qubits.

Returns The corresponding SparseOperator.

fermilib.utils.**save_operator**(*operator*, *file_name=None*, *data_directory=None*)
Save FermionOperator or QubitOperator to file.

Parameters

- **operator** – An instance of FermionOperator 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.

Raises

- OperatorUtilsError – Not saved, file already exists.
- TypeError – Operator of invalid type.

fermilib.utils.**sparse_eigenspectrum**(*sparse_operator*)
Perform a dense diagonalization.

Returns The lowest eigenvalues in a numpy array.

Return type *eigenspectrum*

fermilib.utils.**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 The length scale for the simulation.

Return type length_scale (float)

Raises ValueError – System dimension must be a positive integer.

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z