Quantum simulation of electronic structure

Run in Google Colab (https://colab.research.google.com/github/quantumlib/OpenFermion/blob/master/docs/

The quantum simulation of electronic structure is one of the most promising applications of quantum computers. It has potential applications to materials and drug design. This tutorial provides an introduction to OpenFermion, a library for obtaining and manipulating representations of fermionic and qubit Hamiltonians as well as compiling quantum simulation circuits in Cirq.

```
try:
    import openfermion as of
    import openfermionpyscf as ofpyscf
except ImportError:
    print("Installing OpenFermion and OpenFermion-PySCF...")
    !pip install openfermion openfermionpyscf --quiet
```

```
import numpy as np
from scipy.sparse import linalg

import cirq
import openfermion as of
import openfermionpyscf as ofpyscf
```

Background

A system of N fermionic modes is described by a set of fermionic *annihilation operators* $\{a_p\}_{p=0}^{N-1}$ satisfying the *canonical anticommutation relations* $\begin{aligned}\{a_p, a_q\} &= 0,\\ \{a_p, a_q^\dagger\} &= \delta_{pq},\end{aligned}$ where $\{A, B\} := AB + BA$. The adjoint $a_p^\dagger$ of an annihilation operator a_p is called a *creation operator*, and we refer to creation and annihilation operators as fermionic *ladder operators*.

The canonical anticommutation relations impose a number of consequences on the structure of the vector space on which the ladder operators act; see Michael Nielsen's notes (http://michaelnielsen.org/blog/archive/notes/fermions_and_jordan_wigner.pdf) for a good discussion.

The electronic structure Hamiltonian is commonly written in the form

$$\sum_{pq} T_{pq} a_p^\dagger a_q + \sum_{pqrs} V_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$$

where the T_{pq} and V_{pqrs} are coefficients which depend on the physical system being described. We are interested in calculating the lowest eigenvalue of the Hamiltonian. This eigenvalue is also called the ground state energy.

FermionOperator and QubitOperator

`openfermion.FermionOperator`

(https://quantumai.google/reference/python/openfermion/ops/FermionOperator)

- Stores a weighted sum (linear combination) of fermionic terms
- A fermionic term is a product of ladder operators
- Examples of things that can be represented by `FermionOperator`:

$$a_1$$
$$1.7 a_3^\dagger$$
$$-1.7\, a_3^\dagger a_1$$
$$(1+2i)\, a_4^\dagger a_3^\dagger a_9 a_1$$
$$(1+2i)\, a_4^\dagger a_3^\dagger a_9 a_1 - 1.7\, a_3^\dagger a_1$$

- A fermionic term is internally represented as a tuple of tuples
- Each inner tuple represents a single ladder operator as (index, action)
- Examples of fermionic terms:

$$
\begin{aligned}
I &\mapsto () \\
a_1 &\mapsto ((1, 0),) \\
a_3^\dagger &\mapsto ((3, 1),) \\
a_3^\dagger a_1 &\mapsto ((3, 1), (1, 0)) \\
a_4^\dagger a_3^\dagger a_9 a_1 &\mapsto ((4, 1), (3, 1), (9, 0), (1, 0))
\end{aligned}
$$

- `FermionOperator` is a sum of terms, represented as a dictionary from term to coefficient

```
op = of.FermionOperator(((4, 1), (3, 1), (9, 0), (1, 0)), 1 + 2j) + of.Fermio
    ((3, 1), (1, 0)), -1.7
)

print(op.terms)
```

```
{((4, 1), (3, 1), (9, 0), (1, 0)): (1+2j), ((3, 1), (1, 0)): -1.7}
```

Alternative notation, useful when playing around:

$$
\begin{aligned}
I &\mapsto \text{""} \\
a_1 &\mapsto \text{"1"} \\
a_3^\dagger &\mapsto \text{"3^"} \\
a_3^\dagger a_1 &\mapsto \text{"3^ 1"} \\
a_4^\dagger a_3^\dagger a_9 a_1 &\mapsto \text{"4^ 3^ 9 1"}
\end{aligned}
$$

```
op = of.FermionOperator("4^ 3^ 9 1", 1 + 2j) + of.FermionOperator("3^ 1", -1.7

print(op.terms)
```

```
{((4, 1), (3, 1), (9, 0), (1, 0)): (1+2j), ((3, 1), (1, 0)): -1.7}
```

Just print the operator for a nice readable representation:

```
print(op)
```

```
-1.7 [3^ 1] +
(1+2j) [4^ 3^ 9 1]
```

openfermion.QubitOperator

(https://quantumai.google/reference/python/openfermion/ops/QubitOperator)

Same as `FermionOperator`, but the possible actions are 'X', 'Y', and 'Z' instead of 1 and 0.

```
op = of.QubitOperator(((1, "X"), (2, "Y"), (3, "Z")))
op += of.QubitOperator("X3 Z4", 3.0)

print(op)
```

```
1.0 [X1 Y2 Z3] +
3.0 [X3 Z4]
```

`FermionOperator` and `QubitOperator` actually inherit from the same parent class: `openfermion.SymbolicOperator` (https://quantumai.google/reference/python/openfermion/ops/SymbolicOperator).

The Jordan-Wigner and Bravyi-Kitaev transforms

A fermionic transform maps `FermionOperator`s to `QubitOperator`s in a way that preserves the canonical anticommutation relations. The most basic transforms are the Jordan-Wigner transform (JWT) and Bravyi-Kitaev transform (BKT). Note that the BKT requires the total number of qubits to be predetermined. Whenever a fermionic transform is being applied implicitly, it is the JWT.

```
op = of.FermionOperator("2^ 15")

print(of.jordan_wigner(op))
print()
```

```
print(of.bravyi_kitaev(op, n_qubits=16))
```

```
(0.25+0j) [X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 X15] +
0.25j [X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Y15] +
-0.25j [Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 X15] +
(0.25+0j) [Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Y15]

(-0.25+0j) [Z1 X2 X3 X7 Z15] +
-0.25j [Z1 X2 X3 Y7 Z11 Z13 Z14] +
0.25j [Z1 Y2 X3 X7 Z15] +
(-0.25+0j) [Z1 Y2 X3 Y7 Z11 Z13 Z14]
```

Exercise

Below are some examples of how `FermionOperator`s are mapped to `QubitOperator`s by the Jordan-Wigner transform (the notation 'h.c.' stands for 'hermitian conjugate'):

$$a_p^\dagger \mapsto \frac{1}{2}(X_p - iY_p)Z_0 \cdots Z_{p-1}$$
$$a_p^\dagger a_p \mapsto \frac{1}{2}(I - Z_p)$$
$$(\beta a_p^\dagger a_q + \text{h.c.}) \mapsto \frac{1}{2}[\text{Re}(\beta)(X_p ZZ \cdots ZZ X_q + Y_p ZZ \cdots ZZ Y_q) + \text{Im}(\beta)(Y_p$$

Verify these mappings for $p = 2$ and $q = 7$. The openfermion.hermitian_conjugated (https://quantumai.google/reference/python/openfermion/utils/hermitian_conjugated) function may be useful here.

```
a2 = of.FermionOperator("2")
print(of.jordan_wigner(a2))
print()

a2dag = of.FermionOperator("2^")
print(of.jordan_wigner(a2dag * a2))
print()

a7 = of.FermionOperator("7")
a7dag = of.FermionOperator("7^")
print(of.jordan_wigner((1 + 2j) * (a2dag * a7) + (1 - 2j) * (a7dag * a2)))
```

```
0.5 [Z0 Z1 X2] +
0.5j [Z0 Z1 Y2]

(0.5+0j) [] +
(-0.5+0j) [Z2]

(0.5+0j) [X2 Z3 Z4 Z5 Z6 X7] +
(-1+0j) [X2 Z3 Z4 Z5 Z6 Y7] +
(1+0j) [Y2 Z3 Z4 Z5 Z6 X7] +
(0.5+0j) [Y2 Z3 Z4 Z5 Z6 Y7]
```

Solution

```
a2 = of.FermionOperator("2")
a2dag = of.FermionOperator("2^")
a7 = of.FermionOperator("7")
a7dag = of.FermionOperator("7^")

print(of.jordan_wigner(a2dag))
print()
print(of.jordan_wigner(a2dag * a2))
print()

op = (2 + 3j) * a2dag * a7
op += of.hermitian_conjugated(op)
print(of.jordan_wigner(op))
```

```
0.5 [Z0 Z1 X2] +
-0.5j [Z0 Z1 Y2]

(0.5+0j) [] +
(-0.5+0j) [Z2]

(1+0j) [X2 Z3 Z4 Z5 Z6 X7] +
(-1.5+0j) [X2 Z3 Z4 Z5 Z6 Y7] +
(1.5+0j) [Y2 Z3 Z4 Z5 Z6 X7] +
(1+0j) [Y2 Z3 Z4 Z5 Z6 Y7]
```

Exercise

Use the + and * operators to verify that after applying the JWT to ladder operators, the resulting `QubitOperator`s satisfy

$$a_2 a_7 + a_7 a_2 = 0$$
$$a_2 a_7^\dagger + a_7^\dagger a_2 = 0$$
$$a_2 a_2^\dagger + a_2^\dagger a_2 = 1$$

Solution

```
a2_jw = of.jordan_wigner(a2)
a2dag_jw = of.jordan_wigner(a2dag)
a7_jw = of.jordan_wigner(a7)
a7dag_jw = of.jordan_wigner(a7dag)

print(a2_jw * a7_jw + a7_jw * a2_jw)
print(a2_jw * a7dag_jw + a7dag_jw * a2_jw)
print(a2_jw * a2dag_jw + a2dag_jw * a2_jw)
```

```
0
0
(1+0j) []
```

Array data structures

- When `FermionOperator`s have specialized structure we can store coefficients in numpy arrays, enabling fast numerical manipulation.
- Array data structures can always be converted to `FermionOperator` using `openfermion.get_fermion_operator` (https://quantumai.google/reference/python/openfermion/transforms/get_fermion_operator).

InteractionOperator

- Stores the one- and two-body tensors T_{pq} and V_{pqrs} of the molecular Hamiltonian

$$\sum_{pq} T_{pq} a_p^\dagger a_q + \sum_{pqrs} V_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$$

- Default data structure for molecular Hamiltonians
- Convert from `FermionOperator` using `openfermion.get_interaction_operator` (https://quantumai.google/reference/python/openfermion/transforms/get_interaction_operator)

DiagonalCoulombHamiltonian

- Stores the one- and two-body coefficient matrices T_{pq} and V_{pq} of a Hamiltonian with a diagonal Coulomb term:

$$\sum_{pq} T_{pq} a_p^\dagger a_q + \sum_{pq} V_{pq} a_p^\dagger a_p a_q^\dagger a_q$$

- Leads to especially efficient algorithms for quantum simulation
- Convert from `FermionOperator` using `openfermion.get_diagonal_coulomb_hamiltonian` (https://quantumai.google/reference/python/openfermion/transforms/get_diagonal_coulomb_hamiltonian)

QuadraticHamiltonian

- Stores the Hermitian matrix M_{pq} and antisymmetric matrix Δ_{pq} describing a general quadratic Hamiltonian

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

- Routines included for efficient diagonalization (can handle thousands of fermionic modes)
- Convert from `FermionOperator` using `openfermion.get_quadratic_hamiltonian` (https://quantumai.google/reference/python/openfermion/transforms/get_quadratic_hamiltonian)

Generating the Hamiltonian for a molecule

The cell below demonstrates using one of our electronic structure package plugins, OpenFermion-PySCF, to generate a molecular Hamiltonian for a hydrogen molecule. Note that the Hamiltonian is returned as an `InteractionOperator`. We'll convert it to a `FermionOperator` and print the result.

```
# Set molecule parameters
geometry = [("H", (0.0, 0.0, 0.0)), ("H", (0.0, 0.0, 0.8))]
basis = "sto-3g"
multiplicity = 1
charge = 0

# Perform electronic structure calculations and
# obtain Hamiltonian as an InteractionOperator
hamiltonian = ofpyscf.generate_molecular_hamiltonian(
    geometry, basis, multiplicity, charge
)

# Convert to a FermionOperator
hamiltonian_ferm_op = of.get_fermion_operator(hamiltonian)

print(hamiltonian_ferm_op)
```

```
0.66147151365 [] +
-1.2178260299951058 [0^ 0] +
0.3316650744318082 [0^ 0^ 0 0] +
0.09231339177803066 [0^ 0^ 2 2] +
0.3316650744318082 [0^ 1^ 1 0] +
0.09231339177803066 [0^ 1^ 3 2] +
0.09231339177803066 [0^ 2^ 0 2] +
0.3267206861819477 [0^ 2^ 2 0] +
0.09231339177803066 [0^ 3^ 1 2] +
0.3267206861819477 [0^ 3^ 3 0] +
0.3316650744318082 [1^ 0^ 0 1] +
0.09231339177803066 [1^ 0^ 2 3] +
-1.2178260299951058 [1^ 1] +
0.3316650744318082 [1^ 1^ 1 1] +
```

Let's calculate the ground energy (lowest eigenvalue) of the Hamiltonian. First, we'll map the `FermionOperator` to a `QubitOperator` using the JWT. Then, we'll convert the `QubitOperator` to a SciPy sparse matrix and get its lowest eigenvalue.

```
# Map to QubitOperator using the JWT
hamiltonian_jw = of.jordan_wigner(hamiltonian_ferm_op)

# Convert to Scipy sparse matrix
hamiltonian_jw_sparse = of.get_sparse_operator(hamiltonian_jw)

# Compute ground energy
eigs, _ = linalg.eigsh(hamiltonian_jw_sparse, k=1, which="SA")
```

```
ground_energy = eigs[0]

print("Ground_energy: {}".format(ground_energy))
print("JWT transformed Hamiltonian:")
print(hamiltonian_jw)
```

```
Ground_energy: -1.134147666677095
JWT transformed Hamiltonian:
(-0.16733398905695201+0j) [] +
(-0.04615669588901533+0j) [X0 X1 Y2 Y3] +
(0.04615669588901533+0j) [X0 Y1 Y2 X3] +
(0.04615669588901533+0j) [Y0 X1 X2 Y3] +
(-0.04615669588901533+0j) [Y0 Y1 X2 X3] +
(0.16251648748871642+0j) [Z0] +
(0.1658325372159041+0j) [Z0 Z1] +
(0.11720364720195856+0j) [Z0 Z2] +
(0.1633603430909739+0j) [Z0 Z3] +
(0.16251648748871636+0j) [Z1] +
(0.1633603430909739+0j) [Z1 Z2] +
(0.11720364720195856+0j) [Z1 Z3] +
```

Exercise

Compute the ground energy of the same Hamiltonian, but via the Bravyi-Kitaev transform. Verify that you get the same value.

```
# Map to QubitOperator using the JWT
hamiltonian_bk = of.bravyi_kitaev(hamiltonian_ferm_op)

# Convert to Scipy sparse matrix
hamiltonian_bk_sparse = of.get_sparse_operator(hamiltonian_bk)

# Compute ground energy
eigs, _ = linalg.eigsh(hamiltonian_bk_sparse, k=1, which="SA")
ground_energy = eigs[0]

print("Ground_energy: {}".format(ground_energy))
print("BK transformed Hamiltonian:")
print(hamiltonian_bk)
```

```
Ground_energy: -1.1341476666770918
BK transformed Hamiltonian:
```

```
(-0.16733398905695201+0j) [] +
(0.04615669588901533+0j) [X0 Z1 X2] +
(0.04615669588901533+0j) [X0 Z1 X2 Z3] +
(0.04615669588901533+0j) [Y0 Z1 Y2] +
(0.04615669588901533+0j) [Y0 Z1 Y2 Z3] +
(0.16251648748871642+0j) [Z0] +
(0.16251648748871636+0j) [Z0 Z1] +
(0.1633603430909739+0j) [Z0 Z1 Z2] +
(0.1633603430909739+0j) [Z0 Z1 Z2 Z3] +
(0.11720364720195856+0j) [Z0 Z2] +
(0.11720364720195856+0j) [Z0 Z2 Z3] +
```

Solution

```
# Map to QubitOperator using the BKT
hamiltonian_bk = of.bravyi_kitaev(hamiltonian_ferm_op)

# Convert to Scipy sparse matrix
hamiltonian_bk_sparse = of.get_sparse_operator(hamiltonian_bk)

# Compute ground state energy
eigs, _ = linalg.eigsh(hamiltonian_bk_sparse, k=1, which="SA")
ground_energy = eigs[0]

print("Ground_energy: {}".format(ground_energy))
print("BKT transformed Hamiltonian:")
print(hamiltonian_bk)
```

```
Ground_energy: -1.134147666677097
BKT transformed Hamiltonian:
(-0.16733398905695201+0j) [] +
(0.04615669588901533+0j) [X0 Z1 X2] +
(0.04615669588901533+0j) [X0 Z1 X2 Z3] +
(0.04615669588901533+0j) [Y0 Z1 Y2] +
(0.04615669588901533+0j) [Y0 Z1 Y2 Z3] +
(0.16251648748871642+0j) [Z0] +
(0.16251648748871636+0j) [Z0 Z1] +
(0.1633603430909739+0j) [Z0 Z1 Z2] +
(0.1633603430909739+0j) [Z0 Z1 Z2 Z3] +
(0.11720364720195856+0j) [Z0 Z2] +
(0.11720364720195856+0j) [Z0 Z2 Z3] +
(0.1658325372159041+0j) [Z1] +
```

Exercise

- The BCS mean-field d-wave model of superconductivity has the Hamiltonian

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

Use the `mean_field_dwave` function to generate an instance of this model with dimensions 10x10.

- Convert the Hamiltonian to a `QubitOperator` with the JWT. What is the length of the longest Pauli string that appears?
- Convert the Hamiltonian to a `QubitOperator` with the BKT. What is the length of the longest Pauli string that appears?
- Convert the Hamiltonian to a `QuadraticHamiltonian`. Get its ground energy using the `ground_energy` method of `QuadraticHamiltonian`. What would happen if you tried to compute the ground energy by converting to a sparse matrix?

Hamiltonian simulation with Trotter formulas

- Goal: apply $\mathbf{exp}(-iHt)$ where $H = \sum_j H_j$
- Use an approximation such as $\mathbf{exp}(-iHt) \approx (\prod_{j=1} \mathbf{exp}(-iH_j t/r))^r$
- Exposed via the `openfermion.simulate_trotter` (https://quantumai.google/reference/python/openfermion/circuits/simulate_trotter) function
- Currently implemented algorithms are from arXiv:1706.00023 (https://arxiv.org/pdf/1706.00023.pdf), arXiv:1711.04789 (https://arxiv.org/pdf/1711.04789.pdf), and arXiv:1808.02625 (https://arxiv.org/pdf/1808.02625.pdf), and are based on the JWT
- Currently supported Hamiltonian types: `DiagonalCoulombHamiltonian` and `InteractionOperator`

As a demonstration, we'll simulate time evolution under the hydrogen molecule Hamiltonian we generated earlier.

First, let's create a random initial state and apply the exact time evolution by matrix exponentiation:

$$|\psi\rangle \mapsto \exp(-iHt)|\psi\rangle$$

```
# Create a random initial state
n_qubits = of.count_qubits(hamiltonian)
initial_state = of.haar_random_vector(2**n_qubits, seed=7)

# Set evolution time
time = 1.0

# Apply exp(-i H t) to the state
exact_state = linalg.expm_multiply(-1j * hamiltonian_jw_sparse * time, initia
```

Now, let's create a circuit to perform the evolution and compare the fidelity of the resulting state with the one from exact evolution. The fidelity can be increased by increasing the number of Trotter steps. Note that the Hamiltonian input to `openfermion.simulate_trotter` (https://quantumai.google/reference/python/openfermion/circuits/simulate_trotter) should be an `InteractionOperator`, not a `FermionOperator`.

```
# Initialize qubits
qubits = cirq.LineQubit.range(n_qubits)

# Create circuit
circuit = cirq.Circuit(
    of.simulate_trotter(
        qubits, hamiltonian, time, n_steps=10, order=0, algorithm=of.LOW_RANK
    )
)

# Apply the circuit to the initial state
result = circuit.final_state_vector(initial_state)

# Compute the fidelity with the final state from exact evolution
fidelity = abs(np.dot(exact_state, result.conj())) ** 2

print(fidelity)
```

```
0.9999820449924582
```

```
print(circuit.to_text_diagram(transpose=True))
```

```
0             1                  2                 3
|             |                  |                 |
Rz(0)         Rz(0)              Rz(0)             Rz(0)
|             |                  |                 |
Rz(0.052π)    Rz(0.052π)         Rz(0.03π)         Rz(0.03π)
|             |                  |                 |
Rz(0)         Rz(0)              Rz(0)             Rz(0)
|             |                  |                 |
|             PhISwap(0.25)──────PhISwap(0.25)^-1  |
|             |                  |                 |
PhISwap(0.25)─PhISwap(0.25)^-1   Z^0               |
|             |                  |                 |
|             Z^0                PhISwap(0.25)─────PhISwap(0.25)^-1
|             |                  |                 |
```

Bogoliubov transformation

- Single-particle orbital basis change
- In the particle-conserving case, takes the form

$$U a_p^\dagger U^\dagger = b_p^\dagger, \quad b_p^\dagger = \sum_q u_{pq} a_q^\dagger$$

and u is unitary.

- Can be used to diagonalize any quadratic Hamiltonian:

$$\sum_{p,q} T_{pq} a_p^\dagger a_q \mapsto \sum_j \varepsilon_j b_j^\dagger b_j + \text{constant}$$

- Implementation from arXiv:1711.05395 (https://arxiv.org/pdf/1711.05395.pdf); uses linear depth and linear connectivity

As an example, we'll prepare the ground state of a random particle-conserving quadratic Hamiltonian.

```
n_qubits = 5
quad_ham = of.random_quadratic_hamiltonian(
```

```
    n_qubits, conserves_particle_number=True, seed=7
)

print(of.get_fermion_operator(quad_ham))
```

```
1.690525703800356 [] +
(0.5315776978980016+0j) [0^ 0] +
(-1.347208023348913+2.7004721387490935j) [0^ 1] +
(-0.28362365442898696-1.8784499457335426j) [0^ 2] +
(0.12594647819298657-1.3106154125325498j) [0^ 3] +
(-0.3880303291443195-1.1751249212322041j) [0^ 4] +
(-1.347208023348913-2.7004721387490935j) [1^ 0] +
(2.5012533818678193+0j) [1^ 1] +
(0.3391421007279024-3.8305756810505094j) [1^ 2] +
(-0.3509690502067961+0.090677856754656j) [1^ 3] +
(1.8575239595653907-1.4736314761076197j) [1^ 4] +
(-0.28362365442898696+1.8784499457335426j) [2^ 0] +
(0.3391421007279024+3.8305756810505094j) [2^ 1] +
(-0.019560786804260433+0j) [2^ 2] +
```

Now we construct a circuit which maps computational basis states to eigenstates of the Hamiltonian.

```
_, basis_change_matrix, _ = quad_ham.diagonalizing_bogoliubov_transform()

qubits = cirq.LineQubit.range(n_qubits)
circuit = cirq.Circuit(of.bogoliubov_transform(qubits, basis_change_matrix))

print(circuit.to_text_diagram(transpose=True))
```

```
0             1                   2                   3
|             |                   |                   |
Rz(0)         Rz(0)               Rz(π)               Rz(-π)
|             |                   |                   |
|             |                   |                   PhISwap(0.25)────────P
|             |                   |                   |
|             |                   PhISwap(0.25)───────PhISwap(0.25)^(5/11)  Z
|             |                   |                   |
|             PhISwap(0.25)───────PhISwap(0.25)^0.814  Z^-0.508
|             |                   |                   |
PhISwap(0.25)─PhISwap(0.25)^0.703 Z^-0.304            PhISwap(0.25)────────P
|             |                   |                   |
|             Z^-0.767            PhISwap(0.25)───────PhISwap(0.25)^(13/16) Z
```

In the rotated basis, the quadratic Hamiltonian takes the form

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

We can get the ε_j and the constant using the `orbital_energies` method of `QuadraticHamiltonian`.

```
orbital_energies, constant = quad_ham.orbital_energies()

print(orbital_energies)
print(constant)
```

```
[-6.25377614 -1.2291963   0.71202361  5.0062515   8.20078604]
1.690525703800356
```

The ground state of the Hamiltonian is prepared by filling in the orbitals with negative energy.

```
# Apply the circuit with initial state having the first two modes occupied.
result = circuit.final_state_vector(initial_state=0b11000)

# Compute the expectation value of the final state with the Hamiltonian
quad_ham_sparse = of.get_sparse_operator(quad_ham)
print(of.expectation(quad_ham_sparse, result))

# Print out the ground state energy; it should match
print(quad_ham.ground_energy())
```

```
(-5.792446738060052+1.1102230246251565e-16j)
-5.792446738060049
```

Recall that the Jordan-Wigner transform of $b_j^\dagger b_j$ is $\frac{1}{2}(I - Z)$. Therefore, $\exp(-i\varepsilon_j b_j^\dagger b_j)$ is equivalent to a single-qubit Z rotation under the JWT. Since the operators $b_j^\dagger b_j$ commute, we have

$$\exp(-iHt) = \exp(-i \sum_j \varepsilon_j b_j^\dagger b_j t) = \prod_j \exp(-i \varepsilon_j b_j^\dagger b_j t)$$

This gives a method for simulating time evolution under a quadratic Hamiltonian:

- Use a Bogoliubov transformation to change to the basis in which the Hamiltonian is diagonal (Note: this transformation might be the inverse of what you expect. In that case, use `cirq.inverse` (https://quantumai.google/reference/python/cirq/inverse))
- Apply single-qubit Z-rotations with angles proportional to the orbital energies
- Undo the basis change

The code cell below creates a random initial state and applies time evolution by direct matrix exponentiation.

```
# Create a random initial state
initial_state = of.haar_random_vector(2**n_qubits)

# Set evolution time
time = 1.0

# Apply exp(-i H t) to the state
final_state = linalg.expm_multiply(-1j * quad_ham_sparse * time, initial_state
```

Exercise

Fill in the code cell below to construct a circuit which applies $\exp(-iHt)$ using the method described above

```
# Initialize qubits
qubits = cirq.LineQubit.range(n_qubits)

# Write code below to create the circuit
# You should define the `circuit` variable here
# ------------------------------------------


# ------------------------------------------

# Apply the circuit to the initial state
result = circuit.final_state_vector(initial_state)
```

```
# Compute the fidelity with the correct final state
fidelity = abs(np.dot(final_state, result.conj())) ** 2

# Print fidelity; it should be 1
print(fidelity)
```

```
0.08926042490120051
```

Solution

```
# Initialize qubits
qubits = cirq.LineQubit.range(n_qubits)

# Write code below to create the circuit
# You should define the `circuit` variable here
# ------------------------------------------
def exponentiate_quad_ham(qubits, quad_ham):
    _, basis_change_matrix, _ = quad_ham.diagonalizing_bogoliubov_transform()
    orbital_energies, _ = quad_ham.orbital_energies()

    yield cirq.inverse(of.bogoliubov_transform(qubits, basis_change_matrix))
    for i in range(len(qubits)):
        yield cirq.rz(rads=-orbital_energies[i]).on(qubits[i])
    yield of.bogoliubov_transform(qubits, basis_change_matrix)


circuit = cirq.Circuit(exponentiate_quad_ham(qubits, quad_ham))
# ------------------------------------------

# Apply the circuit to the initial state
result = circuit.final_state_vector(initial_state)

# Compute the fidelity with the correct final state
fidelity = abs(np.dot(final_state, result.conj())) ** 2

# Print fidelity; it should be 1
print(fidelity)
```

```
0.999999999999994
```

Last updated 2022-06-10 UTC.