Circuits 2: Linear Trotter steps of diagonal Coulomb operators

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

Setup

Install the OpenFermion package:

```
try:
    import openfermion
except ImportError:
    !pip install git+https://github.com/quantumlib/OpenFermion.git@master#egg=
```

Electronic structure Hamiltonians with diagonal Coulomb operators

When expressed in an arbitrary basis the molecular electronic structure Hamiltonian takes the form

$$H = \sum_{pq} h_{pq} a_p^\dagger a_q + \sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$$

where the coefficients h_{pq} and h_{pqrs} are determined by integrals taken over the basis functions. Note that this Hamiltonian has $O(N^4)$ terms which tends to make its simulation challenging on near-term devices.

However, as discussed in Phys. Rev. X 8, 011044 (https://journals.aps.org/prx/abstract/10.1103/PhysRevX.8.011044), by carefully selection of basis function it is possible to obtain a representation that diagonalizes the Coulomb operator, leading to a much simpler Hamiltonian with $O(N^2)$ terms that can be written as

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

This form is derived in Phys. Rev. X 8, 011044 (https://journals.aps.org/prx/abstract/10.1103/PhysRevX.8.011044) by using basis functions that are related to a unitary rotation of plane waves. However, plane waves are not the only basis with this property; e.g., see JCP 147, 244102 (https://aip.scitation.org/doi/10.1063/1.5007066) for a basis that provides the diagonal form and high accuracy representation of single-molecules.

Being a periodic basis, plane waves are particularly well suited to simulating periodic materials (e.g. solid state LiH instead of single molecule LiH in vacuum). One can use plane waves to also simulate single-molecules with a basis set discretization error that is asymptotically equivalent to Gaussian molecular orbitals; however, in practice for simulating single-molecules one often needs a constant factor more plane waves than Gaussians, and sometimes that constant factor is prohibitive for NISQ applications. In Circuits 3 (https://quantumai.google/openfermion/tutorials/circuits_3_arbitrary_basis_trotter), we discuss how a combination of techniques from Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change) and this tutorial enable simulation of arbitrary basis electronic structure in low depth. However, this tutorial will focus on representations of the Hamiltonian with a diagonal Coulomb operator. The techniques discussed in this notebook are applicable to any molecular system, whether periodic or not. However, for simplicity this notebook will focus on the simulation of the uniform electron gas, aka "jellium". Jellium has the same Hamiltonian as an arbitrary molecule but without an external potential (i.e. T_{pp} is uniform for all p).

Generation of a dual basis jellium Hamiltonian

We begin by generating a small two-dimensional jellium model in the "plane wave dual basis" as in Phys. Rev. X 8, 011044 (https://journals.aps.org/prx/abstract/10.1103/PhysRevX.8.011044). Such two-dimensional jellium systems are often studied in the context of the fractional quantum Hall effect.

```
import openfermion

# Set parameters of jellium model.
wigner_seitz_radius = 5. # Radius per electron in Bohr radii.
n_dimensions = 2 # Number of spatial dimensions.
grid_length = 2 # Number of grid points in each dimension.
spinless = True # Whether to include spin degree of freedom or not.
```

```
n_electrons = 2 # Number of electrons.

# Figure out length scale based on Wigner-Seitz radius and construct a basis g
length_scale = openfermion.wigner_seitz_length_scale(
    wigner_seitz_radius, n_electrons, n_dimensions)
grid = openfermion.Grid(n_dimensions, grid_length, length_scale)

# Initialize the model and print out.
fermion_hamiltonian = openfermion.jellium_model(grid, spinless=spinless, plane
print(fermion_hamiltonian)

# Convert to DiagonalCoulombHamiltonian type.
hamiltonian = openfermion.get_diagonal_coulomb_hamiltonian(fermion_hamiltonian
```

```
0.1256637061435917 [0^ 0] +
-0.07957747154594769 [0^ 0 1^ 1] +
-0.07957747154594769 [0^ 0 2^ 2] +
-0.23873241463784306 [0^ 0 3^ 3] +
-0.06283185307179587 [0^ 1] +
-0.06283185307179585 [0^ 2] +
-0.06283185307179587 [1^ 0] +
0.1256637061435917 [1^ 1] +
-0.07957747154594769 [1^ 1 0^ 0] +
-0.23873241463784306 [1^ 1 2^ 2] +
-0.07957747154594769 [1^ 1 3^ 3] +
-0.06283185307179585 [1^ 3] +
-0.06283185307179585 [2^ 0] +
0.1256637061435917 [2^ 2] +
```

In the last line above we converted the FermionOperator to a class called DiagonalCoulombHamiltonian which is a special data structure in OpenFermion for representing operators that take the form

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

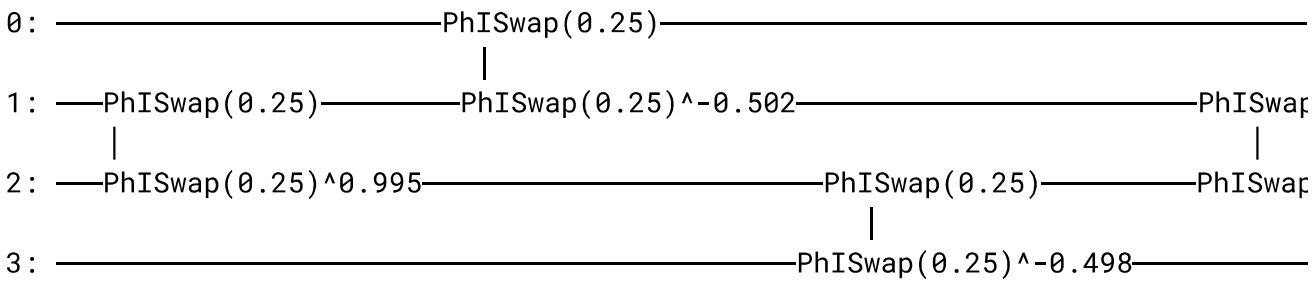
OpenFermion has implemented Hamiltonian simulation algorithms that are optimized specifically for Hamiltonians of this form. They take as input the OpenFermion data structure DiagonalCoulombHamiltonian, which represents such a Hamiltonian in terms of matrices storing T_{pq} and V_{pq}.

Initializing the mean-field state of jellium

Often one would like to begin a simulation of electronic structure in the mean-field state. To do this while keeping operators in the dual basis one needs to apply a rotation of single particle basis functions (see Circuits 1 (https://quantumai.google/openfermion/tutorials/circuits_1_basis_change)). For arbitrary molecules this would necessitate first computing the canonical orbitals using a Hartree-Fock calculation, perhaps by using OpenFermion-PySCF (https://github.com/quantumlib/OpenFermion-PySCF). However, since jellium has no external potential the mean-field state is an eigenstate of the one-body term $\sum_{p,q} T_{pq} a_p^\dagger a_q$. This term is a quadratic Hamiltonian, so its eigenstates can be prepared by applying a Bogoliubov transformation to a computational basis state. The Bogoliubov transformation changes the basis to one in which the quadratic Hamiltonian has the diagonal form $\sum_p \varepsilon_p b_p^\dagger b_p$, where the $b_p^\dagger$ are the creation operators for a new set of orbitals. We'll set the number of electrons to be half the total number of orbitals.

We'll use the OpenFermion class QuadraticHamiltonian to obtain the Bogoliubov transformation matrix. Then, we'll initialize some qubits and create a circuit that applies the transformation to these qubits. Since our algorithms work with linear qubit connectivity, we'll use the `LineQubit` class. We won't bother compiling to Xmon gates for now to keep the circuits simple, but this can be done automatically using the appropriate Cirq methods. We will specify the initial state by passing in a list of the occupied orbitals (which in this case are just the first `n_electron` orbitals).

```
import cirq
import openfermion

# Obtain the Bogoliubov transformation matrix.
quadratic_hamiltonian = openfermion.QuadraticHamiltonian(hamiltonian.one_body)
_, transformation_matrix, _ = quadratic_hamiltonian.diagonalizing_bogoliubov_t

# Create a circuit that prepares the mean-field state
occupied_orbitals = range(n_electrons)
n_qubits = openfermion.count_qubits(quadratic_hamiltonian)
qubits = cirq.LineQubit.range(n_qubits)
state_preparation_circuit = cirq.Circuit(
    openfermion.bogoliubov_transform(
        qubits, transformation_matrix, initial_state=occupied_orbitals))

# Print circuit.
state_preparation_circuit = cirq.drop_negligible_operations(state_preparation_
print(state_preparation_circuit)
```



Hamiltonian simulation via a Trotter-Suzuki product formula

The goal of Hamiltonian time evolution simulation is to apply the unitary operator $\exp(-iHt)$ for some time t. A simulation via a product formula proceeds by dividing the total evolution time t into a finite number of steps r and performing an approximate simulation of $\exp(-iHt/r)$ r times. Each simulation of $\exp(-iHt/r)$ is called a Trotter step. The unitary $\exp(-iHt/r)$ is approximated by interleaving simulations of the terms H_j of a decomposition $H = \sum_{j=1}^{L} H_j$. For example, the first-order symmetric, commonly known as the second-order, Trotter formula is

$$\exp(-iHt) \approx \prod_{j=1}^{L} \exp(-iH_j t/2) \prod_{j=L}^{1} \exp(-iH_j t/2).$$

Higher-order product formulas are obtained from this one via a recursive construction. There is also a zeroth-order formula, which corresponds to

$$\exp(-iHt) \approx \prod_{j=1}^{L} \exp(-iH_j t).$$

In our case, the H_j have the form $T_{pq} a_p^\dagger a_q + T_{pq}^* a_q^\dagger a_p$ or $V_{pq} a_p^\dagger a_p a_q^\dagger a_q$.

To construct a circuit for performing time evolution via a product formula, we need to specify the total evolution time, the number of steps to use, and the order of the formula to use. For a fixed evolution time, increasing the number of steps and increasing the order of the formula both yield a more accurate simulation at the cost of increasing the gate count of the circuit. We could also specify an asymmetric Trotter step, or a controlled version, but we won't do that here. We will need to specify what algorithm will be used to compile the Trotter step. There are several options appropriate for DiagonalCoulombHamiltonians.

A key result of Phys. Rev. Lett. 120, 110501 (https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.120.110501) was to introduce a linear connectivity swap network which cycles through configurations in which all qubits are adjacent in at most N parallel layers of swaps. As shown in that paper, by using fermionc swap gates instead of regular swap gates, one can use this swap network to simulate Trotter steps of an entire DiagonalCoulombHamiltonian. This effectively simulates $N/2$ different pairs of terms $V_{pq}a_p^\dagger a_p a_q^\dagger a_q$ and $T_{pq}a_p^\dagger a_q + T_{pq}^* a_q^\dagger a_p$ in each layer of gates. By repeating the swap network for N layers, all terms are simulated exactly once. This is referred to as the `LINEAR_SWAP_NETWORK` Trotter step in OpenFermion.

Another approach to implementing the Trotter step involves simulating all $\sum_{pq} V_{pq}a_p^\dagger a_p a_q^\dagger a_q$ terms (which commute and thus, this involves no Trotter error) by using the linear swap networks (with normal swap gates), and then performing a basis transformation which diagonalizes the one-body terms so that all of the $\sum_{pq} T_{pq}a_p^\dagger a_q$ terms can be simulated at once, and then performing another basis transformation to restore the original basis. This method involves more gates in each Trotter step but has less Trotter error, and thus might require fewer Trotter steps to achieve some target accuracy. This is referred to as the `SPLIT_OPERATOR` Trotter step in OpenFermion.

Thus, there are currently two options for simulating DiagonalCoulombHamiltonians, `LINEAR_SWAP_NETWORK` and `SPLIT_OPERATOR`, and they correspond to different orderings of the terms H_j in the product formula. Different orderings give different results because the H_j do not all commute. Let's construct a circuit with the `LINEAR_SWAP_NETWORK` method using just one first order Trotter step. We'll insert operations into the circuit using the strategy `EARLIEST` so the printed output will be most compact. Still, the circuit will be longer than the width of this notebook, so we'll print it out transposed.

```
from openfermion.circuits import trotter

# Set algorithm parameters.
time = 1.0
n_steps = 1
order = 1

# Construct circuit
swap_network_trotter_step = cirq.Circuit(
    openfermion.simulate_trotter(
        qubits, hamiltonian, time, n_steps, order,
        algorithm=trotter.LINEAR_SWAP_NETWORK),
    strategy=cirq.InsertStrategy.EARLIEST)

# Print circuit.
```

```
swap_network_trotter_step=cirq.drop_negligible_operations(swap_network_trotte
print(swap_network_trotter_step.to_text_diagram(transpose=True))
```

```
0          1          2          3
|          |          |          |
iSwap──────iSwap^0.02 iSwap──────iSwap^0.02
|          |          |          |
|          |          |          |
|          |          |          |
@──────────@^0.025    @──────────@^0.025
|          |          |          |
×ᶠ─────────×ᶠ         ×ᶠ─────────×ᶠ
|          |          |          |
|          |          |          |
|          |          |          |
|          |          |          |
|          |          |          |
```

Now let's do the same, but using the `SPLIT_OPERATOR` method.

```
split_operator_trotter_step = cirq.Circuit(
    openfermion.simulate_trotter(
        qubits, hamiltonian, time, n_steps, order,
        algorithm=trotter.SPLIT_OPERATOR),
    strategy=cirq.InsertStrategy.EARLIEST)
split_operator_trotter_step=cirq.drop_negligible_operations(split_operator_tr
print(split_operator_trotter_step.to_text_diagram(transpose=True))
```

```
0             1                   2                   3
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |
|             |                   PhISwap(0.25)───────PhISwap(0.25)^0.5
|             |                   |                   |
|             |                   |                   |
|             |                   |                   |
|             PhISwap(0.25)───────PhISwap(0.25)^0.608  |
|             |                   |                   |
|             |                   PhISwap(0.25)───────PhISwap(0.25)^-0.004
|             |                   |                   |
PhISwap(0.25)─PhISwap(0.25)^(2/3) |                   |
|             |                   |                   |
```

Let's run these circuits on the simulator that comes with Cirq and compute the energy of the resulting states.

```
# Initialize Cirq simulator.
simulator = cirq.Simulator()

# Convert the Hamiltonian to a sparse matrix.
hamiltonian_sparse = openfermion.get_sparse_operator(hamiltonian)

# Obtain initial state vector as integer.
initial_state = sum(2 ** (n_qubits - 1 - i) for i in occupied_orbitals)

# Construct and simulate circuit using the swap network method.
circuit = state_preparation_circuit + swap_network_trotter_step
result = simulator.simulate(circuit, initial_state=initial_state)
final_state = result.final_state_vector

print('Energy of state obtained with swap network method: {}'.format(
    openfermion.expectation(hamiltonian_sparse, final_state).real))

# Construct and simulate circuit using the split-operator method.
circuit = state_preparation_circuit + split_operator_trotter_step
result = simulator.simulate(circuit, initial_state=initial_state)
final_state = result.final_state_vector

print('Energy of state obtained with split-operator method: {}'.format(
    openfermion.expectation(hamiltonian_sparse, final_state).real))
```

```
Energy of state obtained with swap network method: -0.19257483118852697
Energy of state obtained with split-operator method: -0.1925762536825269
```

Increasing the number of Trotter steps will cause both methods to converge to the same operation, corresponding to an exact simulation. You can play around with the number of Trotter steps to confirm. Note that for NISQ applications one will often be interested in using the zeroth-order Trotter step, also known as the first-order asymmetric Trotter step. We can implement these Trotter steps by setting the order to zero, as we do below.

```
# Set algorithm parameters.
time = 1.0
n_steps = 1
order = 0
```

```
# Construct circuit
swap_network_trotter_step = cirq.Circuit(
    openfermion.simulate_trotter(
        qubits, hamiltonian, time, n_steps, order,
        algorithm=trotter.LINEAR_SWAP_NETWORK),
    strategy=cirq.InsertStrategy.EARLIEST)
swap_network_trotter_step=cirq.drop_negligible_operations(swap_network_trotte
print(swap_network_trotter_step.to_text_diagram(transpose=True))
```

```
0          1          2          3
|          |          |          |
iSwap──────iSwap^0.04 iSwap──────iSwap^0.04
|          |          |          |
|          |          |          |
|          |          |          |
@──────────@^0.051    @──────────@^0.051
|          |          |          |
×ᶠ─────────×ᶠ         ×ᶠ─────────×ᶠ
|          |          |          |
|          |          |          |
|          |          |          |
|          |          |          |
|          |          |          |
```

Note the unusual pattern of fermionic swap networks towards the end. What is happening there is that in the zeroth order step of a **`LINEAR_SWAP_NETWORK`** style Trotter step, the qubit order is reversed upon output. To avoid this one needs to set an option called **`omit_final_swaps`**, e.g.

```
swap_network_trotter_step = cirq.Circuit(
    openfermion.simulate_trotter(
        qubits, hamiltonian, time, n_steps, order,
        algorithm=trotter.LINEAR_SWAP_NETWORK,
        omit_final_swaps=True),
    strategy=cirq.InsertStrategy.EARLIEST)
swap_network_trotter_step=cirq.drop_negligible_operations(swap_network_trotte
print(swap_network_trotter_step.to_text_diagram(transpose=True))
```

```
0          1          2          3
|          |          |          |
iSwap──────iSwap^0.04 iSwap──────iSwap^0.04
|          |          |          |
```

```
|           |           |           |
|           |           |           |
@———————————@^0.051     @———————————@^0.051
|           |           |           |
×ᶠ——————————×ᶠ          ×ᶠ——————————×ᶠ
|           |           |           |
|           |           |           |
|           |           |           |
|           |           |           |
|           |           |           |
¦           ¦           ¦           ¦
```

One can also have fun compiling arbitrary high-order formulas. Here's the third-order symmetric formula:

```
order=3
n_steps=1
swap_network_trotter_step = cirq.Circuit(
    openfermion.simulate_trotter(
        qubits, hamiltonian, time, n_steps, order,
        algorithm=trotter.LINEAR_SWAP_NETWORK),
    strategy=cirq.InsertStrategy.EARLIEST)
swap_network_trotter_step = cirq.drop_negligible_operations(swap_network_trot
print(swap_network_trotter_step.to_text_diagram(transpose=True))
```

```
0           1            2           3
|           |            |           |
iSwap———————iSwap^0.003  iSwap———————iSwap^0.003
|           |            |           |
|           |            |           |
|           |            |           |
@———————————@^0.004      @———————————@^0.004
|           |            |           |
×ᶠ——————————×ᶠ           ×ᶠ——————————×ᶠ
|           |            |           |
|           |            |           |
|           |            |           |
|           |            |           |
|           |            |           |
```

Application to phase estimation

Phase estimation is a procedure that, given access to a controlled unitary and one of its eigenvectors, estimates the phase of the eigenvalue corresponding to that eigenvector. In

the context of quantum simulation, this unitary is usually the time evolution operator e^{-iHt}. Thus if $H|n\rangle = E_n|n\rangle$, and we initialize the system in state $|n\rangle$, phase estimation would estimate the value $E_n t/(2\pi)$. To avoid aliasing of phases, t should be chosen to be smaller than $2\pi/|E_n|$.

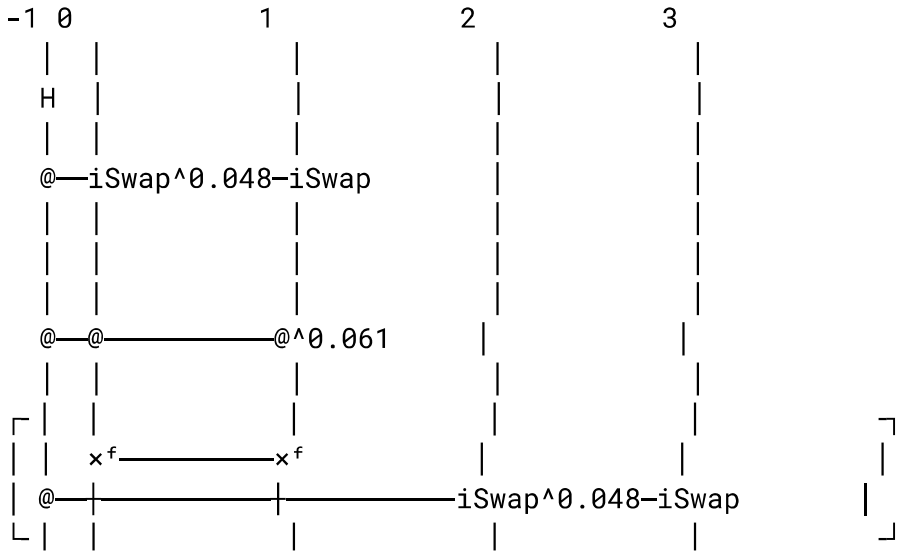
The simplest phase estimation circuit measures one bit of the phase in four steps:

1. Perform a Hadamard transform on the control qubit.
2. Apply the controlled unitary.
3. Perform a Hadamard transform on the control qubit.
4. Measure the control qubit.

Below, we demonstrate the construction of this circuit where the controlled unitary is a controlled Trotter step of our jellium Hamiltonian. This circuit can be used as a building block of a larger phase estimation circuit.

```
# Define a phase estimation circuit.
def measure_bit_of_phase(system_qubits,
                         control_qubit,
                         controlled_unitary):
    yield cirq.H(control_qubit)
    yield controlled_unitary
    yield cirq.H(control_qubit)
    yield cirq.measure(control_qubit)

# Get an upper bound on the Hamiltonian norm.
import numpy
bound = numpy.sum(numpy.abs(hamiltonian.one_body)) + numpy.sum(numpy.abs(hami

# Construct phase estimation circuit.
time = 2 * numpy.pi / bound
control = cirq.LineQubit(-1)

controlled_unitary = openfermion.simulate_trotter(
    qubits, hamiltonian, time,
    n_steps=1,
    order=1,
    algorithm=trotter.LINEAR_SWAP_NETWORK,
    control_qubit=control)

circuit = cirq.Circuit(
    measure_bit_of_phase(
        qubits,
        control,
```

```
        controlled_unitary))

# Print the circuit.
circuit=cirq.drop_negligible_operations(circuit)
print(circuit.to_text_diagram(transpose=True))
```



Last updated 2023-10-27 UTC.