The Jordan-Wigner and Bravyi-Kitaev Transforms

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=
```

Ladder operators and the canonical anticommutation relatio

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*. In a finite-dimensional vector space the anticommutation relations have the following consequences:

- The operators $\{a_p^\dagger a_p\}_{p=0}^{N-1}$ commute with each other and have eigenvalues 0 and 1. These are called the *occupation number operators*.
- There is a normalized vector $|\mathbf{vac}\rangle$, called the *vacuum state*, which is a mutual 0-eigenvector of all the $a_p^\dagger a_p$.

- If $|\psi\rangle$ is a 0-eigenvector of $a_p^\dagger a_p$, then $a_p^\dagger|\psi\rangle$ is a 1-eigenvector of $a_p^\dagger a_p$. This explains why we say that $a_p^\dagger$ creates a fermion in mode p.
- If $|\psi\rangle$ is a 1-eigenvector of $a_p^\dagger a_p$, then $a_p|\psi\rangle$ is a 0-eigenvector of $a_p^\dagger a_p$. This explains why we say that a_p annihilates a fermion in mode p.
- $a_p^2 = 0$ for all p. One cannot create or annihilate a fermion in the same mode twice.
- The set of 2^N vectors

$$|n_0, \ldots, n_{N-1}\rangle := (a_0^\dagger)^{n_0} \cdots (a_{N-1}^\dagger)^{n_{N-1}} |\mathrm{vac}\rangle, \qquad n_0, \ldots, n_{N-1} \in \{0, 1\}$$

  are orthonormal. We can assume they form a basis for the entire vector space.

- The annihilation operators a_p act on this basis as follows:

$$a_p|n_0, \ldots, n_{p-1}, 1, n_{p+1}, \ldots, n_{N-1}\rangle = (-1)^{\sum_{q=0}^{p-1} n_q}|n_0, \ldots, n_{p-1}, 0, n_{p+1}, \ldots$$
$$a_p|n_0, \ldots, n_{p-1}, 0, n_{p+1}, \ldots, n_{N-1}\rangle = 0\,.$$

See here (http://michaelnielsen.org/blog/archive/notes/fermions_and_jordan_wigner.pdf) for a derivation and discussion of these consequences.

Mapping fermions to qubits with transforms

To simulate a system of fermions on a quantum computer, we must choose a representation of the ladder operators on the Hilbert space of the qubits. In other words, we must designate a set of qubit operators (matrices) which satisfy the canonical anticommutation relations. Qubit operators are written in terms of the Pauli matrices X, Y, and Z. In OpenFermion a representation is specified by a transform function which maps fermionic operators (typically instances of FermionOperator) to qubit operators (instances of QubitOperator). In this demo we will discuss the Jordan-Wigner and Bravyi-Kitaev transforms, which are implemented by the functions `jordan_wigner` and `bravyi_kitaev`.

The Jordan-Wigner Transform

Under the Jordan-Wigner Transform (JWT), the annihilation operators are mapped to qubit operators as follows:

$$\begin{aligned} a_p &\mapsto \frac{1}{2}(X_p + \mathrm{i}Y_p)Z_1 \cdots Z_{p-1} \\ &= (|0\rangle\langle 1|)_p Z_1 \cdots Z_{p-1} \\ &=: \tilde{a}_p. \end{aligned}$$

This operator has the following action on a computational basis vector $|z_0, \ldots, z_{N-1}\rangle$:

$$\tilde{a}_p |z_0 \ldots, z_{p-1}, 1, z_{p+1}, \ldots, z_{N-1}\rangle = (-1)^{\sum_{q=0}^{p-1} z_q} |z_0 \ldots, z_{p-1}, 0, z_{p+1}, \ldots, z_{N-1}\rangle$$
$$\tilde{a}_p |z_0 \ldots, z_{p-1}, 0, z_{p+1}, \ldots, z_{N-1}\rangle = 0.$$

Note that $|n_0, \ldots, n_{N-1}\rangle$ is a basis vector in the Hilbert space of fermions, while $|z_0, \ldots, z_{N-1}\rangle$ is a basis vector in the Hilbert space of qubits. Similarly, in OpenFermion a_p is a FermionOperator while $\tilde{a}_p$ is a QubitOperator.

Let's instantiate some FermionOperators, map them to QubitOperators using the JWT, and check that the resulting operators satisfy the expected relations.

```
from openfermion import *

# Create some ladder operators
annihilate_2 = FermionOperator('2')
create_2 = FermionOperator('2^')
annihilate_5 = FermionOperator('5')
create_5 = FermionOperator('5^')

# Construct occupation number operators
num_2 = create_2 * annihilate_2
num_5 = create_5 * annihilate_5

# Map FermionOperators to QubitOperators using the JWT
annihilate_2_jw = jordan_wigner(annihilate_2)
create_2_jw = jordan_wigner(create_2)
annihilate_5_jw = jordan_wigner(annihilate_5)
create_5_jw = jordan_wigner(create_5)
num_2_jw = jordan_wigner(num_2)
num_5_jw = jordan_wigner(num_5)

# Create QubitOperator versions of zero and identity
zero = QubitOperator()
identity = QubitOperator(())

# Check the canonical anticommutation relations
assert anticommutator(annihilate_5_jw, annihilate_2_jw) == zero
assert anticommutator(annihilate_5_jw, annihilate_5_jw) == zero
assert anticommutator(annihilate_5_jw, create_2_jw) == zero
```

```
assert anticommutator(annihilate_5_jw, create_5_jw) == identity

# Check that the occupation number operators commute
assert commutator(num_2_jw, num_5_jw) == zero

# Print some output
print("annihilate_2_jw = \n{}".format(annihilate_2_jw))
print('')
print("create_2_jw = \n{}".format(create_2_jw))
print('')
print("annihilate_5_jw = \n{}".format(annihilate_5_jw))
print('')
print("create_5_jw = \n{}".format(create_5_jw))
print('')
print("num_2_jw = \n{}".format(num_2_jw))
print('')
print("num_5_jw = \n{}".format(num_5_jw))
```

```
0.5 [Z0 Z1 Z2 Z3 Z4 X5] +
0.5j [Z0 Z1 Z2 Z3 Z4 Y5]

create_5_jw =
0.5 [Z0 Z1 Z2 Z3 Z4 X5] +
-0.5j [Z0 Z1 Z2 Z3 Z4 Y5]

num_2_jw =
(0.5+0j) [] +
(-0.5+0j) [Z2]

num_5_jw =
(0.5+0j) [] +
(-0.5+0j) [Z5]
```

The parity transform

By comparing the action of $\tilde{a}_p$ on $|z_0, \ldots, z_{N-1}\rangle$ in the JWT with the action of a_p on $|n_0, \ldots, n_{N-1}\rangle$ (described in the first section of this demo), we can see that the JWT is associated with a particular mapping of bitstrings $e : \{0,1\}^N \to \{0,1\}^N$, namely, the identity map $e(x) = x$. In other words, under the JWT, the fermionic basis vector $|n_0, \ldots, n_{N-1}\rangle$ is represented by the computational basis vector $|z_0, \ldots, z_{N-1}\rangle$, where $z_p = n_p$ for all p. We can write this as

$$|x\rangle \mapsto |e(x)\rangle,$$

where the vector on the left is fermionic and the vector on the right is qubit. We call the mapping e an *encoder*.

There are other transforms which are associated with different encoders. To see why we might be interested in these other transforms, observe that under the JWT, $\tilde{a}_p$ acts not only on qubit p but also on qubits $0, \ldots, p-1$. This means that fermionic operators with low weight can get mapped to qubit operators with high weight, where by weight we mean the number of modes or qubits an operators acts on. There are some disadvantages to having high-weight operators; for instance, they may require more gates to simulate and are more expensive to measure on some near-term hardware platforms. In the worst case, the annihilation operator on the last mode will map to an operator which acts on all the qubits. To emphasize this point let's apply the JWT to the annihilation operator on mode 99:

```
print(jordan_wigner(FermionOperator('99')))
```

```
0.5 [Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 Z17 Z18 Z19 Z2(
0.5j [Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 Z17 Z18 Z19 Z2
```

The purpose of the string of Pauli Z's is to introduce the phase factor $(-1)^{\sum_{q=0}^{p-1} n_q}$ when acting on a computational basis state; when e is the identity encoder, the modulo-2 sum $\sum_{q=0}^{p-1} n_q$ is computed as $\sum_{q=0}^{p-1} z_q$, which requires reading p bits and leads to a Pauli Z string with weight p. A simple solution to this problem is to consider instead the encoder defined by

$$e(x)_p = \sum_{q=0}^{p} x_q \pmod 2,$$

which is associated with the mapping of basis vectors $|n_0, \ldots, n_{N-1}\rangle \mapsto |z_0, \ldots, z_{N-1}\rangle$, where $z_p = \sum_{q=0}^{p} n_q$ (again addition is modulo 2). With this encoding, we can compute the sum $\sum_{q=0}^{p-1} n_q$ by reading just one bit because this is the value stored by z_{p-1}. The associated transform is called the parity transform because the p-th qubit is storing the parity (modulo-2 sum) of modes $0, \ldots, p$. Under the parity transform, annihilation operators are mapped as follows:

$$
\begin{aligned}
a_p &\mapsto \frac{1}{2}(X_p Z_{p-1} + \mathrm{i} Y_p) X_{p+1} \cdots X_N \\
&= \frac{1}{4}[(X_p + \mathrm{i} Y_p)(I + Z_{p-1}) - (X_p - \mathrm{i} Y_p)(I - Z_{p-1})] X_{p+1} \cdots X_N \\
&= [(|0\rangle\langle 1|)_p (|0\rangle\langle 0|)_{p-1} - (|0\rangle\langle 1|)_p (|1\rangle\langle 1|)_{p-1}] X_{p+1} \cdots X_N
\end{aligned}
$$

The term in brackets in the last line means "if $z_p = n_p$ then annihilate in mode p; otherwise, create in mode p and attach a minus sign". The value stored by z_{p-1} contains the information needed to determine whether a minus sign should be attached or not. However, now there is a string of Pauli X's acting on modes $p+1, \ldots, N-1$ and hence using the parity transform also yields operators with high weight. These Pauli X's perform the necessary update to $z_{p+1}, \ldots, z_{N-1}$ which is needed if the value of n_p changes. In the worst case, the annihilation operator on the first mode will map to an operator which acts on all the qubits.

Since the parity transform does not offer any advantages over the JWT, OpenFermion does not include a standalone function to perform it. However, there is functionality for defining new transforms by specifying an encoder and decoder pair, also known as a binary code (in our examples the decoder is simply the inverse mapping), and the binary code which defines the parity transform is included in the library as an example. See Lowering qubit requirements using binary codes (https://quantumai.google/openfermion/tutorials/binary_code_transforms_demo) for a demonstration of this functionality and how it can be used to reduce the qubit resources required for certain applications.

Let's use this functionality to map our previously instantiated FermionOperators to QubitOperators using the parity transform with 10 total modes and check that the resulting operators satisfy the expected relations.

```
# Set the number of modes in the system
n_modes = 10

# Define a function to perform the parity transform
def parity(fermion_operator, n_modes):
    return binary_code_transform(fermion_operator, parity_code(n_modes))

# Map FermionOperators to QubitOperators using the parity transform
annihilate_2_parity = parity(annihilate_2, n_modes)
create_2_parity = parity(create_2, n_modes)
annihilate_5_parity = parity(annihilate_5, n_modes)
create_5_parity = parity(create_5, n_modes)
num_2_parity = parity(num_2, n_modes)
num_5_parity = parity(num_5, n_modes)
```

```
# Check the canonical anticommutation relations
assert anticommutator(annihilate_5_parity, annihilate_2_parity) == zero
assert anticommutator(annihilate_5_parity, annihilate_5_parity) == zero
assert anticommutator(annihilate_5_parity, create_2_parity) == zero
assert anticommutator(annihilate_5_parity, create_5_parity) == identity

# Check that the occupation number operators commute
assert commutator(num_2_parity, num_5_parity) == zero

# Print some output
print("annihilate_2_parity = \n{}".format(annihilate_2_parity))
print('')
print("create_2_parity = \n{}".format(create_2_parity))
print('')
print("annihilate_5_parity = \n{}".format(annihilate_5_parity))
print('')
print("create_5_parity = \n{}".format(create_5_parity))
print('')
print("num_2_parity = \n{}".format(num_2_parity))
print('')
print("num_5_parity = \n{}".format(num_5_parity))
```

```
annihilate_2_parity =
0.5 [Z1 X2 X3 X4 X5 X6 X7 X8 X9] +
0.5j [Y2 X3 X4 X5 X6 X7 X8 X9]

create_2_parity =
0.5 [Z1 X2 X3 X4 X5 X6 X7 X8 X9] +
(-0-0.5j) [Y2 X3 X4 X5 X6 X7 X8 X9]

annihilate_5_parity =
0.5 [Z4 X5 X6 X7 X8 X9] +
0.5j [Y5 X6 X7 X8 X9]

create_5_parity =
0.5 [Z4 X5 X6 X7 X8 X9] +
```

Now let's map one of the FermionOperators again but with the total number of modes set to 100.

```
print(parity(annihilate_2, 100))
```

```
0.5 [Z1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15 X16 X17 X18 X19 X20 X
0.5j [Y2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15 X16 X17 X18 X19 X20 X21
```

Note that with the JWT, it is not necessary to specify the total number of modes in the system because $\tilde{a}_p$ only acts on qubits $0, \ldots, p$ and not any higher ones.

The Bravyi-Kitaev transform

The discussion above suggests that we can think of the action of a transformed annihilation operator $\tilde{a}_p$ on a computational basis vector $|z\rangle$ as a 4-step classical algorithm:

1. Check if $n_p = 0$. If so, then output the zero vector. Otherwise,
2. Update the bit stored by z_p.
3. Update the rest of the bits z_q, $q \neq p$.
4. Multiply by the parity $\sum_{q=0}^{p-1} n_p$.

Under the JWT, Steps 1, 2, and 3 are represented by the operator $(|0\rangle\langle 1|)_p$ and Step 4 is accomplished by the operator $Z_0 \cdots Z_{p-1}$ (Step 3 actually requires no action). Under the parity transform, Steps 1, 2, and 4 are represented by the operator $(\lvert{0}\rangle\langle{1}\rvert)_p$ $(\lvert{0}\rangle\langle{0}\rvert)$_{p - 1} - $(\lvert{0}\rangle\langle{1}\rvert)_p$ $(\lvert{1}\rangle\langle{1}\rvert)$_{p - 1} $and Step 3 is accomplished by the operator X_{p+1} \cdots X_{N-1}$.

To obtain a simpler description of these and other transforms (with an aim at generalizing), it is better to put aside the ladder operators and work with an alternative set of $2N$ operators defined by

$$c_p = a_p + a_p^\dagger, \qquad d_p = -\mathrm{i}(a_p - a_p^\dagger).$$

These operators are known as Majorana operators. Note that if we describe how Majorana operators should be transformed, then we also know how the annihilation operators should be transformed, since

$$a_p = \frac{1}{2}(c_p + \mathrm{i}d_p).$$

For simplicity, let's consider just the c_p; the d_p are treated similarly. The action of c_p on a fermionic basis vector is given by

$$c_p|n_0,\ldots,n_{p-1},n_p,n_{p+1},\ldots,n_{N-1}\rangle = (-1)^{\sum_{q=0}^{p-1} n_q}|n_0,\ldots,n_{p-1},1-n_p,n_{p+1},.$$

In words, c_p flips the occupation of mode p and multiplies by the ever-present parity factor. If we transform c_p to a qubit operator $\tilde{c}_p$, we should be able to describe the action of $\tilde{c}_p$ on a computational basis vector $|z\rangle$ with a 2-step classical algorithm:

1. Update the string z to a new string z'.
2. Multiply by the parity $\sum_{q=0}^{p-1} n_q$.

Step 1 amounts to flipping some bits, so it will be performed by some Pauli X's, and Step 2 will be performed by some Pauli Z's. So $\tilde{c}_p$ should take the form

$$\tilde{c}_p = X_{U(p)} Z_{P(p-1)},$$

where $U(j)$ is the set of bits that need to be updated upon flipping n_j, and $P(j)$ is a set of bits that stores the sum $\sum_{q=0}^{j} n_q$ (let's define $P(-1)$ to be the empty set). Let's see how this looks under the JWT and parity transforms.

```
# Create a Majorana operator from our existing operators
c_5 = annihilate_5 + create_5

# Set the number of modes (required for the parity transform)
n_modes = 10

# Transform the Majorana operator to a QubitOperator in two different ways
c_5_jw = jordan_wigner(c_5)
c_5_parity = parity(c_5, n_modes)

# Print some output
print("c_5_jw = \n{}".format(c_5_jw))
print('')
print("c_5_parity = \n{}".format(c_5_parity))
```

```
c_5_jw =
1.0 [Z0 Z1 Z2 Z3 Z4 X5]

c_5_parity =
1.0 [Z4 X5 X6 X7 X8 X9]
```

For the JWT, $U(j) = \{j\}$ and $P(j) = \{0, \ldots, j\}$, whereas for the parity transform, $U(j) = \{j, \ldots, N-1\}$ and $P(j) = \{j\}$. The size of these sets can be as large as N, the total number of modes. These sets are determined by the encoding function e.

It is possible to pick a clever encoder with the property that these sets have size $O(\log N)$. The corresponding transform will map annihilation operators to qubit operators with weight $O(\log N)$, which is much smaller than the $\Omega(N)$ weight associated with the JWT and parity transforms. This fact was noticed by Bravyi and Kitaev (https://arxiv.org/abs/quant-ph/0003137), and later Havlíček and others (https://arxiv.org/abs/1701.07072) pointed out that the encoder which achieves this is implemented by a classical data structure called a Fenwick tree. The transforms described in these two papers actually correspond to different variants of the Fenwick tree data structure and give different results when the total number of modes is not a power of 2. OpenFermion implements the one from the first paper as **`bravyi_kitaev`** and the one from the second paper as **`bravyi_kitaev_tree`**. Generally, the first one (**`bravyi_kitaev`**) is preferred because it results in operators with lower weight and is faster to compute.

Let's transform our previously instantiated Majorana operator using the Bravyi-Kitaev transform.

```
c_5_bk = bravyi_kitaev(c_5, n_modes)
print("c_5_bk = \n{}".format(c_5_bk))
```

```
c_5_bk =
1.0 [Z3 Z4 X5 X7]
```

The advantage of the Bravyi-Kitaev transform is not apparent in a system with so few modes. Let's look at a system with 100 modes.

```
n_modes = 100

# Initialize some Majorana operators
c_17 = FermionOperator('[17] + [17^]')
c_50 = FermionOperator('[50] + [50^]')
c_73 = FermionOperator('[73] + [73^]')

# Map to QubitOperators
c_17_jw = jordan_wigner(c_17)
c_50_jw = jordan_wigner(c_50)
c_73_jw = jordan_wigner(c_73)
```

```
c_17_parity = parity(c_17, n_modes)
c_50_parity = parity(c_50, n_modes)
c_73_parity = parity(c_73, n_modes)
c_17_bk = bravyi_kitaev(c_17, n_modes)
c_50_bk = bravyi_kitaev(c_50, n_modes)
c_73_bk = bravyi_kitaev(c_73, n_modes)

# Print some output
print("Jordan-Wigner\n"
      "-------------")
print("c_17_jw = \n{}".format(c_17_jw))
print('')
print("c_50_jw = \n{}".format(c_50_jw))
print('')
print("c_73_jw = \n{}".format(c_73_jw))
print('')
print("Parity\n"
      "------")
print("c_17_parity = \n{}".format(c_17_parity))
print('')
print("c_50_parity = \n{}".format(c_50_parity))
print('')
print("c_73_parity = \n{}".format(c_73_parity))
print('')
print("Bravyi-Kitaev\n"
      "-------------")
print("c_17_bk = \n{}".format(c_17_bk))
print('')
print("c_50_bk = \n{}".format(c_50_bk))
print('')
print("c_73_bk = \n{}".format(c_73_bk))
```

```
Jordan-Wigner
-------------
c_17_jw =
1.0 [Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 X17]

c_50_jw =
1.0 [Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 Z17 Z18 Z19 Z2

c_73_jw =
1.0 [Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Z11 Z12 Z13 Z14 Z15 Z16 Z17 Z18 Z19 Z2

Parity
------
c_17_parity =
```

Now let's go back to a system with 10 modes and check that the Bravyi-Kitaev transformed operators satisfy the expected relations.

```
# Set the number of modes in the system
n_modes = 10

# Map FermionOperators to QubitOperators using the Bravyi-Kitaev transform
annihilate_2_bk = bravyi_kitaev(annihilate_2, n_modes)
create_2_bk = bravyi_kitaev(create_2, n_modes)
annihilate_5_bk = bravyi_kitaev(annihilate_5, n_modes)
create_5_bk = bravyi_kitaev(create_5, n_modes)
num_2_bk = bravyi_kitaev(num_2, n_modes)
num_5_bk = bravyi_kitaev(num_5, n_modes)

# Check the canonical anticommutation relations
assert anticommutator(annihilate_5_bk, annihilate_2_bk) == zero
assert anticommutator(annihilate_5_bk, annihilate_5_bk) == zero
assert anticommutator(annihilate_5_bk, create_2_bk) == zero
assert anticommutator(annihilate_5_bk, create_5_bk) == identity

# Check that the occupation number operators commute
assert commutator(num_2_bk, num_5_bk) == zero

# Print some output
print("annihilate_2_bk = \n{}".format(annihilate_2_bk))
print('')
print("create_2_bk = \n{}".format(create_2_bk))
print('')
print("annihilate_5_bk = \n{}".format(annihilate_5_bk))
print('')
print("create_5_bk = \n{}".format(create_5_bk))
print('')
print("num_2_bk = \n{}".format(num_2_bk))
print('')
print("num_5_bk = \n{}".format(num_5_bk))
```

```
annihilate_2_bk =
0.5 [Z1 X2 X3 X7] +
0.5j [Z1 Y2 X3 X7]

create_2_bk =
0.5 [Z1 X2 X3 X7] +
-0.5j [Z1 Y2 X3 X7]

annihilate_5_bk =
```

```
0.5 [Z3 Z4 X5 X7] +
0.5j [Z3 Y5 X7]

create_5_bk =
0.5 [Z3 Z4 X5 X7] +
```


Last updated 2023-01-18 UTC.