Introduction to the bosonic 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=
```

The BosonOperator

Bosonic systems, like Fermionic systems, are expressed using the bosonic creation and annihilation operators $b_k^\dagger$ and b_k respectively. Unlike fermions, however, which satisfy the Pauli exclusion principle and thus are distinguished by the canonical fermionic anticommutation relations, the bosonic ladder operators instead satisfy a set of commutation relations:

$$[b_i^\dagger, b_j^\dagger] = 0, \quad [b_i, b_j] = 0, \quad [b_i, b_j^\dagger] = \delta_{ij}$$

Any weighted sums of products of these operators are represented with the `BosonOperator` data structure in OpenFermion. Similarly to when we introduced the `FermionOperator`, the following are examples of valid `BosonOperators`:

$$a_1$$
$$1.7 b_3^\dagger$$
$$-1.7\, b_3^\dagger b_1$$
$$(1+2i)\, b_3^\dagger b_4^\dagger b_1 b_9$$
$$(1+2i)\, b_3^\dagger b_4^\dagger b_1 b_9 - 1.7\, b_3^\dagger b_1$$

The `BosonOperator` class is contained in `ops/_boson_operators.py`. The `BosonOperator` is derived from the `SymbolicOperator`, the same class that derives the `FermionOperator`. As such, the details of the class implementation are identical - as in the fermion case, the class is implemented as hash table (python dictionary). The keys of the dictionary encode the strings of ladder operators and values of the dictionary store the coefficients - the strings are subsequently encoded as a tuple of 2-tuples which we refer to as the "terms tuple".

Each ladder operator is represented by a 2-tuple. The first element of the 2-tuple is an int indicating the quantum mode on which the ladder operator acts. The second element of the 2-tuple is Boole: 1 represents raising and 0 represents lowering. For instance, $b_8^\dagger$ is represented in a 2-tuple as $(8, 1)$. Note that indices start at 0 and the identity operator is an empty list.

$$\begin{aligned} I &\mapsto () \\ b_1 &\mapsto ((1, 0),) \\ b_3^\dagger &\mapsto ((3, 1),) \\ b_3^\dagger b_1 &\mapsto ((3, 1), (1, 0)) \\ b_3^\dagger b_4^\dagger b_1 b_9 &\mapsto ((3, 1), (4, 1), (1, 0), (9, 0)) \end{aligned}$$

Alternatively, the `BosonOperator` supports the string-based syntax introduced in the `FermionOperator`; in this case, the terms are separated by spaces, with the integer corresponding to the quantum mode the operator acts on, and `'^'` indicating the Hermitian conjugate:

$$\begin{aligned} I &\mapsto \text{""} \\ b_1 &\mapsto \text{"1"} \\ b_3^\dagger &\mapsto \text{"3\^{}"} \\ b_3^\dagger b_1 &\mapsto \text{"3\^{} 1"} \\ b_3^\dagger b_4^\dagger b_1 b_9 &\mapsto \text{"3\^{} 4\^{} 1 9"} \end{aligned}$$

Note that, unlike the `FermionOperator`, the bosonic creation operators of different indices commute. As a result, the `BosonOperator` automatically sorts groups of annihilation and

creation operators in ascending order of the modes they act on.

Let's initialize our first term! We do it two different ways below.

```
from openfermion.ops import BosonOperator

my_term = BosonOperator(((3, 1), (5, 0), (4, 1), (1, 0)))
print(my_term)

my_term = BosonOperator('3^ 5 4^ 1')
print(my_term)
```

```
1.0 [1 3^ 4^ 5]
1.0 [1 3^ 4^ 5]
```

Note the printed order differs from the code, since bosonic operators of different indices commute past each other.

The preferred way to specify the coefficient in openfermion 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 second 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.

The additive and multiplicative identities can also be created:

- `BosonOperator(())` and `BosonOperator('')` initialises the identity (`BosonOperator.identity()`).
- `BosonOperator()` and `BosonOperator()` initialises the zero operator (`BosonOperator.zero()` (https://quantumai.google/reference/python/openfermion/ops/BosonOperator#zero)).

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

bad_way_to_initialize = -1.7 * BosonOperator('3^ 1')
print(bad_way_to_initialize)

identity = BosonOperator('')
print(identity == BosonOperator.identity())
```

```
print(identity)

zero_operator = BosonOperator()
print(zero_operator == BosonOperator.zero())
print(zero_operator)
```

```
-1.7 [1 3^]
-1.7 [1 3^]
True
1.0 []
True
0
```

Note that `BosonOperator` has only one attribute: .terms. This attribute is the dictionary which stores the term tuples.

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

```
(1+2j) [1^ 3 4^ 9]
{((1, 1), (3, 0), (4, 1), (9, 0)): (1+2j)}
```

Methods and functions that act on the BosonOperator

There are various functions and methods that act on the `BosonOperator`; these include the ability to normal order, double check if the operator is Hermitian, and calculate the Hermitian conjugate.

```
from openfermion.utils import hermitian_conjugated, is_hermitian
from openfermion.transforms import normal_ordered
```

`normal_ordered_boson` applies the bosonic commutation relations to write the operator using only normal-ordered terms; that is, that all creation operators are to the left of annihilation operators:

```
H = BosonOperator('0 0^', 1. + 2.j)
H.is_normal_ordered()
```

```
False
```

```
normal_ordered(BosonOperator('0 0^', 1. + 2.j))
```

```
(1+2j) [] +
(1+2j) [0^ 0]
```

We can also use a boson operator method to check if the operator conserves the particle number - that is, for each qumode, the number of annihilation operators equals the number of creation operators.

```
H.is_boson_preserving()
```

```
True
```

```
H = BosonOperator('0 0^ 1^ ', 1. + 2.j)
H.is_boson_preserving()
```

```
False
```

The Hermitian conjugated function returns the Hermitian conjugate of the operator, and its hermiticity can be checked using `is_hermitian`:

```
is_hermitian(H)
```

```
False
```

```
hermitian_conjugated(H)
```

```
(1-2j) [0 0^ 1]
```

```
H = BosonOperator('0 1^', 1/2.)
H += BosonOperator('1 0^', 1/2.)
print(is_hermitian(H))
print(hermitian_conjugated(H))
```

```
True
0.5 [0 1^] +
0.5 [0^ 1]
```

The QuadOperator

Using the bosonic ladder operators, it is common to define the canonical position and momentum operators $\hat{q}$ and $\hat{p}$:

$$\hat{q}_i = \sqrt{\frac{\hbar}{2}}(\hat{b}_i + \hat{b}_i^\dagger), \quad \hat{p}_i = -i\sqrt{\frac{\hbar}{2}}(\hat{b}_i - \hat{b}_i^\dagger)$$

These operators are Hermitian, and are referred to as the phase space quadrature operators. They satisfy the canonical commutation relation

$$[\hat{q}_i, \hat{p}_j] = \delta_{ij} i\hbar$$

where the value of $\hbar$ depends on convention, often taking values $\hbar = 0.5$, 1, or 2.

In OpenFermion, the quadrature operators are represented by the `QuadOperator` class, and stored as a dictionary of tuples (as keys) and coefficients (as values). For example, the multi-mode quadrature operator $q_0 p_1 q_3$ is represented internally as `((0, 'q'), (1,`

`'p'), (3, 'q'))`. Alternatively, `QuadOperators` also support string input - using string input, the same operator is described by `'q0 p1 q3'`.

```
from openfermion.ops import QuadOperator

H = QuadOperator('q0 p1 q3')
print(H)
print(H.terms)

H2 = QuadOperator('q3 p4', 3.17)
H2 -= 77. * H
print('')
print(H2)
```

```
1.0 [q0 p1 q3]
{((0, 'q'), (1, 'p'), (3, 'q')): 1.0}

-77.0 [q0 p1 q3] +
3.17 [q3 p4]
```

Note that quadrature operators of different indices commute; as such, like the `BosonOperator`, by default we sort quadrature operators such that the operators acting on the lowest numbered mode appear to the left.

Methods and functions that act on the QuadOperator

Like the `BosonOperator`, there are various functions and methods that act on the `QuadOperator`; these include the ability to normal order, double check if the operator is Hermitian, and calculate the Hermitian conjugate.

```
from openfermion.utils import hermitian_conjugated, is_hermitian
```

`normal_ordered_quad` is an arbitrary convention chosen in OpenFermion that allows us to compare two quadrature operators that might be equivalent, but written in different forms. It is simply defined as a quadrature operator that has all of the position operators $\hat{q}$ to the left of the momentum operators $\hat{q}$. All quadrature operators can be placed in this 'normal form' by making use of the canonical commutation relation.

```
H = QuadOperator('p0 q0', 1. + 2.j)
H.is_normal_ordered()
```

```
False
```

```
normal_ordered(H)
```

```
(2-1j) [] +
(1+2j) [q0 p0]
```

By default, we assume the value $\hbar = 1$ in the canonical commutation relation, but this can be modified by passing the `hbar` keyword argument to the function:

```
normal_ordered(H, hbar=2)
```

```
(4-2j) [] +
(1+2j) [q0 p0]
```

We can also use a quad operator method to check if the operator is **Gaussian** - that is, all terms in the quad operator are of quadratic order or lower:

```
H = QuadOperator('p0 q0', 1. + 2.j)
H.is_gaussian()
```

```
True
```

```
H = QuadOperator('p0 q0 q1', 1. + 2.j)
H.is_gaussian()
```

```
False
```

The Hermitian conjugated function returns the Hermitian conjugate of the operator, and its hermiticity can be checked using `is_hermitian`:

```
H = QuadOperator('p0 q1 p1', 1-2j)
hermitian_conjugated(H)
```

```
(1+2j) [p0 p1 q1]
```

```
H = QuadOperator('p0 q0', 1/2.)
H += QuadOperator('q0 p0', -1/2.)
print(is_hermitian(H))
print(hermitian_conjugated(H))
```

```
False
-0.5 [p0 q0] +
0.5 [q0 p0]
```

```
H = QuadOperator('p0 q0', 1/2.)
H += QuadOperator('q0 p0', 1/2.)
print(is_hermitian(H))
print(hermitian_conjugated(H))
```

```
True
0.5 [p0 q0] +
0.5 [q0 p0]
```

```
hermitian_conjugated(H)
```

```
0.5 [p0 q0] +
0.5 [q0 p0]
```

Converting between quadrature operators and bosonic operators

Converting between bosonic ladder operators and quadrature operators is simple - we just apply the definition of the $\hat{q}$ and $\hat{p}$ operators in terms of $\hat{b}$ and $\hat{b}^\dagger$. Two functions are provided to do this automatically; `get_quad_operator` and `get_boson_operator`:

```
from openfermion.transforms import get_boson_operator, get_quad_operator
```

```
H = QuadOperator('p0 q0', 1/2.)
H += QuadOperator('q0 p0', 1/2.)
H
```

```
0.5 [p0 q0] +
0.5 [q0 p0]
```

```
get_boson_operator(H)
```

```
-0.5000000000000001j [0 0] +
0.5000000000000001j [0^ 0^]
```

Note that, since these conversions are dependent on the value of $\hbar$ chosen, both accept a `hbar` keyword argument. As before, if not specified, the default value of $\hbar$ is `hbar=1`.

```
H = BosonOperator('0 0^')
normal_ordered(get_quad_operator(H, hbar=0.5), hbar=0.5)
```

```
(0.5+0j) [] +
(1+0j) [p0 p0] +
1.0 [q0 q0]
```

Weyl quantization and symmetric ordering

We also provide support for the Weyl quantization - this maps a polynomial function of the form

$$f(q_0, \ldots, q_{N-1}, p_0 \ldots, p_{N-1}) = q_0^{m_0} \cdots q_{N-1}^{m_{N-1}} p_0^{m_0} \cdots p_{N-1}^{m_{N-1}}$$

on the phase space to the corresponding combination of quadrature operators $\hat{q}$ and $\hat{p}$. To do so, we make use of the McCoy formula,

$$q^m p^n \to \frac{1}{2^n} \sum_{r=0}^{n} \binom{n}{r} q^r p^m q^{n-r}.$$

```
from openfermion.transforms import weyl_polynomial_quantization, symmetric_or
```

For `weyl_polynomial_quantization`, the polynomial function in the phase space is provided in the form of a string, where 'q' or 'p' is the phase space quadrature variable, the integer directly following is the mode it is with respect to, and '^2' is the polynomial power. If the power is not provided, it is assumed to be '^1'.

```
weyl_polynomial_quantization('q0 p0')
```

```
0.5 [p0 q0] +
0.5 [q0 p0]
```

```
weyl_polynomial_quantization('q0^2 p0^3 q1^3')
```

```
0.125 [p0 p0 p0 q0 q0 q1 q1 q1] +
0.375 [p0 p0 q0 q0 p0 q1 q1 q1] +
0.375 [p0 q0 q0 p0 p0 q1 q1 q1] +
0.125 [q0 q0 p0 p0 p0 q1 q1 q1]
```

McCoy's formula is also used to provide a function that returns the symmetric ordering of a `BosonOperator` or `QuadOperator`, $S(\hat{O})$. Note that $S(\hat{O}) \neq \hat{O}$:

```
symmetric_ordering(QuadOperator('q0 p0'))
```

```
0.5 [p0 q0] +
0.5 [q0 p0]
```

Consider the symmetric ordering of the square of the bosonic number operator, $\hat{n} = \hat{b}^\dagger \hat{b}$:

```
from openfermion.hamiltonians import number_operator
n2 = number_operator(1, parity=1) * number_operator(1, parity=1)
```

```
n2
```

```
1.0 [0^ 0 0^ 0]
```

```
Sn2 = symmetric_ordering(n2)
Sn2
```

```
0.25 [0 0 0^ 0^] +
0.5 [0 0^ 0^ 0] +
0.25 [0^ 0^ 0 0]
```

We can use `normal_ordered_boson` to simplify this result:

```
Sn2 = normal_ordered(Sn2)
Sn2
```

```
0.5 [] +
2.0 [0^ 0] +
1.0 [0^ 0^ 0 0]
```

Therefore $S(\hat{n}) = \hat{b}^\dagger \hat{b}^\dagger \hat{b} \hat{b} + 2\hat{b}^\dagger \hat{b} + 0.5$. This is equivalent to $\hat{n}^2 + \hat{n} + 0.5$:

```
Sn2 == normal_ordered(n2 + number_operator(1, parity=1) + 0.5*BosonOperator.i
```

```
True
```

Bose-Hubbard Hamiltonian

In addition to the bosonic operators discussed above, we also provide Bosonic Hamiltonians that describe specific models. The Bose-Hubbard Hamiltonian over a discrete lattice or grid described by nodes $V = \{0, 1, \ldots, N-1\}$ is described by:

$$H = - t \sum_{\langle i, j \rangle} b_i^\dagger b_{j + 1}

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

where

- The indices $\langle i, j \rangle$ run over pairs i and j of adjacenct nodes (nodes that are connected) in the grid
- t is the tunneling amplitude
- U is the on-site interaction potential

- μ is the chemical potential
- V is the dipole or nearest-neighbour interaction potential

The Bose-Hubbard Hamiltonian function provided in OpenFermion models a Bose-Hubbard model on a two-dimensional grid, with dimensions given by `[x_dimension, y_dimension]`. It has the form

```
bose_hubbard(x_dimension, y_dimension, tunneling, interaction,
                 chemical_potential=0., dipole=0., periodic=True)
```

where

- `x_dimension` (int): The width of the grid.
- `y_dimension` (int): The height of the grid.
- `tunneling` (float): The tunneling amplitude t.
- `interaction` (float): The attractive local interaction U.
- `chemical_potential` (float, optional): The chemical potential μ at each site. Default value is 0.
- `periodic` (bool, optional): If True, add periodic boundary conditions. Default is True.
- `dipole` (float): The attractive dipole interaction strength V.

Below is an example of a Bose-Hubbard Hamiltonian constructed in OpenFermion.

```
from openfermion.hamiltonians import bose_hubbard, fermi_hubbard
bose_hubbard(2, 2, 1, 1)
```

```
-1.0 [0 1^] +
-1.0 [0 2^] +
-0.5 [0^ 0] +
0.5 [0^ 0 0^ 0] +
-1.0 [0^ 1] +
-1.0 [0^ 2] +
-1.0 [1 3^] +
-0.5 [1^ 1] +
0.5 [1^ 1 1^ 1] +
-1.0 [1^ 3] +
-1.0 [2 3^] +
```

```
-0.5 [2^ 2] +
0.5 [2^ 2 2^ 2] +
-1.0 [2^ 3] +
```

Sparse bosonic operators

Like the fermionic operators, OpenFermion contains the capability to represent bosonic operators as a sparse matrix (`sparse.csc_matrix`). However, as the fermionic operators can be represented as finite matrices, this is not the case of bosonic systems, as they inhabit a infinite-dimensional Fock space. Instead, a integer truncation value N need to be provided - the returned sparse operator will be of size $N^M \times N^M$, where M is the number of modes in the system, and acts on the truncated Fock basis $\{|0\rangle, |1\rangle, \ldots, |N-1\rangle\}$.

```
from openfermion.linalg import boson_operator_sparse
```

The function `boson_operator_sparse` acts on both `BosonOperator`s and `QuadOperator`s:

```
H = boson_operator_sparse(BosonOperator('0^ 0'), 5)
```

```
H.toarray()
```

```
array([[0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 2.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 3.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 4.+0.j]])
```

```
H = boson_operator_sparse(QuadOperator('q0'), 5, hbar=1)
H.toarray()
```

```
array([[0.        +0.j, 0.70710678+0.j, 0.        +0.j, 0.        +0.j,
        0.        +0.j],
```

```
[0.70710678+0.j, 0.        +0.j, 1.        +0.j, 0.        +0.j,
 0.        +0.j],
[0.        +0.j, 1.        +0.j, 0.        +0.j, 1.22474487+0.j,
 0.        +0.j],
[0.        +0.j, 0.        +0.j, 1.22474487+0.j, 0.        +0.j,
 1.41421356+0.j],
[0.        +0.j, 0.        +0.j, 0.        +0.j, 1.41421356+0.j,
 0.        +0.j]])
```


Last updated 2023-01-18 UTC.