Lowering qubit requirements using binary codes

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=
```

Introduction

Molecular Hamiltonians are known to have certain symmetries that are not taken into account by mappings like the Jordan-Wigner or Bravyi-Kitaev transform. The most notable of such symmetries is the conservation of the total number of particles in the system. Since those symmetries effectively reduce the degrees of freedom of the system, one is able to reduce the number of qubits required for simulation by utilizing binary codes (arXiv:1712.07067).

We can represent the symmetry-reduced Fermion basis by binary vectors of a set $\mathcal{V} \ni \boldsymbol{\nu}$, with $\boldsymbol{\nu} = (\nu_0, \nu_1, \ldots, \nu_{N-1})$, where every component $\nu_i \in \{0, 1\}$ and N is the total number of Fermion modes. These binary vectors $\boldsymbol{\nu}$ are related to the actual basis states by: $\left[\prod_{i=0}^{N-1} (a_i^\dagger)^{\nu_i}\right] |\mathrm{vac}\rangle$, where $(a_i^\dagger)^0 = 1$. The qubit basis, on the other hand, can be characterized by length-n binary vectors $\boldsymbol{\omega} = (\omega_0, \ldots, \omega_{n-1})$, that represent an n-qubit basis state by:

$$|\omega_0\rangle \otimes |\omega_1\rangle \otimes \cdots \otimes |\omega_{n-1}\rangle \ .$$

Since $\mathcal{V}$ is a mere subset of the N-fold binary space, but the set of the vectors $\boldsymbol{\omega}$ spans the entire n-fold binary space we can assign every vector $\boldsymbol{\nu}$ to a vector $\boldsymbol{\omega}$, such that $n < N$. This reduces the amount of qubits required by $(N - n)$. The mapping can be done by a binary code, a classical object that consists of an encoder function $\boldsymbol{e}$ and a decoder function $\boldsymbol{d}$. These functions relate the binary vectors $\boldsymbol{e}(\boldsymbol{\nu}) = \boldsymbol{\omega}$, $\boldsymbol{d}(\boldsymbol{\omega}) = \boldsymbol{\nu}$, such that $\boldsymbol{d}(\boldsymbol{e}(\boldsymbol{\nu})) = \boldsymbol{\nu}$. In OpenFermion, we, at the moment, allow for non-linear decoders $\boldsymbol{d}$ and linear encoders $\boldsymbol{e}(\boldsymbol{\nu}) = \boldsymbol{A}\boldsymbol{\nu}$, where the matrix multiplication with the $(n \times N)$-binary matrix $\boldsymbol{A}$ is $(\mathbf{mod}\ 2)$ in every component.

Symbolic binary functions

The non-linear binary functions for the components of the decoder are here modeled by the $\mathbf{BinaryPolynomial}$ class in openfermion.ops. For initialization we can conveniently use strings ('w0 w1 + w1 +1' for the binary function $\boldsymbol{\omega} \rightarrow \omega_0\omega_1 + \omega_1 + \mathbf{1}\ \mathbf{mod}\ \mathbf{2}$), the native data structure or symbolic addition and multiplication.

```
from openfermion.ops import BinaryPolynomial

binary_1 = BinaryPolynomial('w0 w1 + w1 + 1')

print("These three expressions are equivalent: \n", binary_1)
print(BinaryPolynomial('w0') * BinaryPolynomial('w1 + 1') + BinaryPolynomial(
print(BinaryPolynomial([(1, 0), (1, ), ('one', )]))

print('The native data type structure can be seen here:')
print(binary_1.terms)
print('We can always evaluate the expression for instance by the vector (w0, \
      binary_1.evaluate('100'))
```

```
These three expressions are equivalent:
 [W0 W1] + [W1] + [1]
[W0 W1] + [W0] + [1]
[W0 W1] + [W1] + [1]
The native data type structure can be seen here:
[(0, 1), (1,), ('one',)]
We can always evaluate the expression for instance by the vector (w0, w1, w2)
```

Binary codes

The **BinaryCode** class bundles a decoder - a list of decoder components, which are instances of **BinaryPolynomial** - and an encoder - the matrix A as sparse numpy array - as a binary code. The constructor however admits (dense) numpy arrays, nested lists or tuples as input for A, and arrays, lists or tuples of **BinaryPolynomial** objects - or valid inputs for **BinaryPolynomial** constructors - as input for d. An instance of the **BinaryCode** class knows about the number of qubits and the number of modes in the mapping.

```
from openfermion.ops import BinaryCode

code_1 = BinaryCode([[1, 0, 0], [0, 1, 0]], ['w0', 'w1', 'w0 + w1 + 1' ])

print(code_1)
print('number of qubits: ', code_1.n_qubits, '  number of Fermion modes: ', c
print('encoding matrix: \n', code_1.encoder.toarray())
print('decoder: ', code_1.decoder)
```

```
[[[1, 0, 0], [0, 1, 0]], '[[W0],[W1],[W0] + [W1] + [1]]']
number of qubits:  2   number of Fermion modes:  3
encoding matrix:
 [[1 0 0]
 [0 1 0]]
decoder:  [[W0], [W1], [W0] + [W1] + [1]]
```

The code used in the example above, is in fact the (odd) Checksum code, and is implemented already - along with a few other examples from arxiv:1712.07067. In addition to the **checksum_code** the functions **weight_one_segment_code**, **weight_two_segment_code**, that output a subcode each, as well as **weight_one_binary_addressing_code** can be found under openfermion.transforms._code_transform_functions.

There are two other ways to construct new codes from the ones given - both of them can be done conveniently with symbolic operations between two code objects (e, d) and (e', d') to yield a new code (e'', d''):

Appendage

Input and output vectors of two codes are appended to each other such that:

$$e''(\boldsymbol{\nu} \oplus \boldsymbol{\nu}') = \boldsymbol{e}(\boldsymbol{\nu}) \oplus \boldsymbol{e}'(\boldsymbol{\nu}')\,, \qquad d''(\boldsymbol{\omega} \oplus \boldsymbol{\omega}') = \boldsymbol{d}(\boldsymbol{\omega}) \oplus \boldsymbol{d}'(\boldsymbol{\omega}')\,.$$

This is implemented with symbolic addition of two $\mathbf{BinaryCode}$ objects (using + or +=) or, for appending several instances of the same code at once, multiplication of the $\mathbf{BinaryCode}$ with an integer. Appending codes is useful when we want to obtain a segment code, or a segmented transform.

Concatenation

Two codes can (if the corresponding vectors match in size) be applied consecutively, in the sense that the output of the encoder of the first code is input to the encoder of the second code. This defines an entirely new encoder, and the corresponding decoder is defined to undo this operation.

$$\boldsymbol{e}''(\boldsymbol{\nu}'') = \boldsymbol{e}'\left(\boldsymbol{e}(\boldsymbol{\nu}'')\right)\,, \qquad \boldsymbol{d}''(\boldsymbol{\omega}'') = \boldsymbol{d}\left(\boldsymbol{d}'(\boldsymbol{\omega}'')\right)$$

This is done by symbolic multiplication of two $\mathbf{BinaryCode}$ instances (with * or *=). One can concatenate the codes with each other such that additional qubits can be saved (e.g. checksum code * segment code), or to modify the resulting gates after transform (e.g. checksum code * Bravyi-Kitaev code).

A broad palette of codes is provided to help construct codes symbolically. The $\mathbf{jordan\_wigner\_code}$ can be appended to every code to fill the number of modes, concatenating the $\mathbf{bravyi\_kitaev\_code}$ or $\mathbf{parity\_code}$ will modify the appearance of gates after the transform. The $\mathbf{interleaved\_code}$ is useful to concatenate appended codes with if in Hamiltonians, Fermion operators are ordered by spin indexing even-odd (up-down-up-down-up ...) . This particular instance is used in the demonstration below.

Before we turn to describe the transformation, a word of warning has to be spoken here. Controlled gates that occur in the Hamiltonian by using non-linear codes are decomposed into Pauli strings, e.g. $\mathbf{CPHASE}(1,2) = \frac{1}{2}(1 + Z_1 + Z_2 - Z_1 Z_2)$. In that way the amount of terms in a Hamiltonian might rise exponentially, if one chooses to use strongly non-linear codes.

Operator transform

The actual transform of Fermion operators into qubit operators is done with the routine $\mathbf{binary\_code\_transform}$, that takes a Hamiltonian and a suitable code as inputs, outputting a qubit Hamiltonian.

Let us consider the case of a molecule with 4 modes where, due to the absence of magnetic interactions, the set of valid modes is only $\mathcal{V} = \{(1, 1, 0, 0), (1, 0, 0, 1), (0, 1, 1, 0), (0, 0, 1, 1)\}$. One can either use an (even weight) checksum code to save a single qubit, or use and (odd weight) checksum code on spin-up and -down modes each to save two qubits. Since the ordering is even-odd, however, this requires to concatenate the with the interleaved code, which switches the spin indexing of the qubits from even-odd ordering to up-then-down. Instead of using the interleaved code, we can also use the reorder function to apply up-then-down ordering on the hamiltonian.

```
from openfermion.transforms import *
from openfermion.chem import MolecularData
from openfermion.transforms import binary_code_transform
from openfermion.transforms import get_fermion_operator
from openfermion.linalg import eigenspectrum
from openfermion.transforms import normal_ordered, reorder
from openfermion.utils import up_then_down

def LiH_hamiltonian():
    geometry = [('Li', (0., 0., 0.)), ('H', (0., 0., 1.45))]
    molecule = MolecularData(geometry, 'sto-3g', 1,
                             description="1.45")
    molecule.load()
    molecular_hamiltonian = molecule.get_molecular_hamiltonian(occupied_indice
    hamiltonian = normal_ordered(get_fermion_operator(molecular_hamiltonian))
    return hamiltonian

hamiltonian = LiH_hamiltonian()
print('Fermionic Hamiltonian')
print (hamiltonian)
print("The eigenspectrum")
print(eigenspectrum(hamiltonian))

print('\n-----\n')
jw = binary_code_transform(hamiltonian, jordan_wigner_code(4))
print('Jordan-Wigner transformed Hamiltonian')
print(jw)
print("the eigenspectrum of the transformed hamiltonian")
print(eigenspectrum(jw))

print('\n-----\n')
cksm_save_one = binary_code_transform(hamiltonian, checksum_code(4,0))
print('Even-weight checksum code')
print(cksm_save_one)
print("the eigenspectrum of the transformed hamiltonian")
print(eigenspectrum(cksm_save_one))
```

```
print('\n-----\n')
up_down_save_two = binary_code_transform(hamiltonian, interleaved_code(4)*(2*
print('Double odd-weight checksum codes')
print(up_down_save_two )
print("the eigenspectrum of the transformed hamiltonian")
print(eigenspectrum(up_down_save_two ))

print('\n-----\n')
print('Instead of interleaving, we can apply up-then-down ordering using the
up_down_save_two = binary_code_transform(reorder(hamiltonian,up_then_down), 2
print(up_down_save_two)
print("the eigenspectrum of the transformed hamiltonian")
print(eigenspectrum(up_down_save_two))
```

```
Fermionic Hamiltonian
-6.7698132180879735 [] +
-0.7952726864779313 [0^ 0] +
0.04614563473199314 [0^ 2] +
-0.4977908053255035 [1^ 0^ 1 0] +
-0.046145652803099894 [1^ 0^ 2 1] +
0.046145652803099894 [1^ 0^ 3 0] +
-0.011731985763800887 [1^ 0^ 3 2] +
-0.7952726864779313 [1^ 1] +
0.04614563473199314 [1^ 3] +
0.04614563473199324 [2^ 0] +
-0.21652178317319534 [2^ 0^ 2 0] +
-0.04614565280309991 [2^ 1^ 1 0] +
```


Last updated 2023-01-18 UTC.