

# BraQit: A PYTHON PACKAGE FOR QUANTUM COMPUTATIONS

Michael Li

July 2020

## Abstract

The present document overviews the main functionalities of **BraQit**, a new python package for quantum computations.

## 1 Introduction

Quantum computers use the alien features of quantum mechanics to perform computational tasks that are beyond the capabilities of classical computers. A salient example of computational superiority is Shor's algorithm which factors very large numbers in exponential speed [1]. This will have a major impact on current cryptosystems. The impact of quantum computing will be overwhelming, affecting almost all industries and fields.

Quantum computing is also very technical field, lying at the intersection of Computer Science, Mathematics and quantum physics, and can be very intimidating for potential users. The goal of the present paper is to give an overview of a new python package dedicated to quantum computations and easy to use. As a matter of fact, we strove to make our package **user friendly** as much as possible, without comprising on **efficiency**:

- **user friendly:** An example of this is illustrated by ... [todo: create a superposition and then apply Hadamard/Cnot]
- **efficient:** All the constructs are written in the highly optimized python package `numpy`, making calculations really fast. In the previous example, the operation `*` is a `numpy.matmul`. Also, states are implemented using sparse structures.

Finally, we have named the new package **BraQit** to pay tribute to the **founders' bra-ket notations and philosophy**, which is a core guideline behind our implementation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The class State and its subclasses</b>	<b>3</b>
2.1	Methods of the State class . . . . .	3
2.1.1	Operations . . . . .	3
2.1.2	tensor method . . . . .	4
2.1.3	to_array method . . . . .	4
2.1.4	to_dict method . . . . .	5
2.1.5	Normalization . . . . .	5
2.1.6	Probabilities . . . . .	5
2.2	Sub-classes of State . . . . .	6
2.2.1	Uniform State . . . . .	6
<b>3</b>	<b>The class Operator and its subclasses</b>	<b>6</b>
3.1	Methods in Operator Class . . . . .	7
3.1.1	Operations . . . . .	7
3.1.2	Complex Conjugate Transpose and isHermitian Method . . . . .	7
3.1.3	Eigenvectors and Eigenvalues . . . . .	8
3.2	Sub-classes of Operator Class . . . . .	8
<b>4</b>	<b>The class Algorithm</b>	<b>10</b>
4.1	Examples: . . . . .	10
4.1.1	Constructing the Greenberger–Horne–Zeilinger State . . . . .	10
4.1.2	Grover’s search . . . . .	12
<b>5</b>	<b>ComplexNumber class</b>	<b>13</b>
5.1	Operations in ComplexNum . . . . .	14
5.2	Complex Conjugate Method . . . . .	14
<b>6</b>	<b>Feedback</b>	<b>14</b>

## 2 The class State and its subclasses

This class, as the name suggests, encapsulates the notion of quantum state (a vector in a *Hilbert* space). Explicitly, the constructor of the state takes as input a python dictionary. In this dictionary, the keys will correlate to the various states of the qubits in a superposition (or the single state if there is only one key inputted), and the values of the keys correlate with the amplitudes (Shown in Figure 1).

```
dict = {'00': 1, '11':1}
State(dict)

|00⟩ + |11⟩
```

Figure 1: Here, 'dict' is the dictionary representation of the 2 qubit state being created. Once we call the constructor on the state class, it will return the state in ket notation

*Sparse implementation:* While Hilbert spaces are exponentially large, states can have only small number, relatively speaking, of nonzero amplitudes. We have thus opted for the use of dict. The class constructor automatically handles these sparse scenarios— Figure 1.

### 2.1 Methods of the State class

#### 2.1.1 Operations

One important aspect of this `State` class is the overridden operations we can perform, allowing for a lot of convenience. This includes addition, multiplication, subtraction, tensor product, and equality. For example, if we wanted to add two states without these methods,  $a$  and  $b$ , we would have to change both states into arrays, add each element, and convert the result back into a state. However, by overriding the `__add__` method, we can simply type  $x + y$  and it will return a result.

We will now dive into detail into these few methods. First, we will look at addition and subtraction. Again, if we wanted to add two states,  $x$  and  $y$ , and say we set  $x = |00\rangle$  and  $y = |11\rangle$ ,  $x + y$  would equal  $|00\rangle + |11\rangle$ . What this allows us is to create a superposition of states using addition and subtraction. This notion of superposition is a quantum system existing at multiple states at once, but only when measured will there be a definitive state of the system.

Similarly, the multiplication method in our state class can multiply a state and a number or two states together. For multiplying a state and a number, the values of the amplitudes of the different configurations will change by the factor of the number. On the other hand, the multiplication between two states is simply the dot product of their array representation. This multiplication method

will be very important, especially when we look at the projection operator later on in this paper.

Finally, we also have overridden the equality method in python to make it compatible in our state method. This can be used to compared two states when necessary.

```
A = State({'10':2, '11':4})
B = State({'100':3, '101':-2, '111':4})
A.tensor(B)

6.0|10100> - 4.0|10101> + 8.0|10111> + 12.0|11100> - 8.0|11101> + 16.0|11111>
```

Figure 2: In this example, we have state  $A = 4|10\rangle + 11|11\rangle$  and state  $B = 3|100\rangle - 2|101\rangle + 4|111\rangle$ . When we tensor the states together, we get the 5 qubit state shown as the result.

### 2.1.2 tensor method

The tensor product (also known as the Kronecker's product), denoted by  $\otimes$ , is also another operation we can use. Mathematically speaking, the tensor product of two matrices  $A$  and  $B$  is defined as following:

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{pmatrix}$$

In the case of states, which are In our case, we are dealing with states which are represented by 1 dimensional vectors, the tensor product of two states is another another state (See Figure 2). Physically, tensor product is combining the two quantum systems and creating a larger system with all their qubits.

### 2.1.3 to\_array method

The class `State` also contains two useful methods: `to_array()` and `to_dict()`. The first method i.e., `to_array` method simply converts a state into column matrix. For example, if we have the following 2 qubit state:

$$|00\rangle + |01\rangle - |10\rangle$$

then the array:

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

would be the corresponding vector. The reason why the position 3 has a value of zeros is because 1, in binary, can be represented by 11, and the original state did not contain the state  $|11\rangle$ :

```
st = State({'00':1, '01':1, '10':-1})
st.to_array()
```

Notice again the sparse implementation of the state `st` which is quite convenient.

One important thing to note here is the ordering of our array: In BraQit, the array is defined following the order in

```
[np.binary_repr(i, n) for i in range(2**n)]
```

In the example above, the order is given by

```
['00', '01', '10', '11']
```

which explains the zero at the last coordinate.

#### 2.1.4 to\_dict method

The `to_dict` method is a static method with a parameter that accepts a numpy column matrix. What it does is the opposite of the `to_array` method: it converts the array into a state. To call it, we can simply code:

```
State.to_dict(array)
```

where `array` is the column matrix that would be inputted. One important requirement of this method is that the imputed array must contain  $2^n$  elements where `n` is the number of qubits. Otherwise, it will return an error message.

#### 2.1.5 Normalization

The `normalize` method is an instance method that, as the name indicates, normalizes a specific state. For example, if we consider the state in Figure 1:

$$|00\rangle + |11\rangle,$$

once normalized, the result will be

$$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle.$$

In code, we simply call `State(dict).normalize()` on the `State` object created. The output is the normalized `State` object.

#### 2.1.6 Probabilities

Once normalized, one can also obtain the different probabilities of the state achieving a specific configuration once measured. To run this, there are two instance methods: `get_probabilities()` and `get_prob()`. The latter method, `get_prob`, requires a single parameter which is the configuration. Once run, the method

will return a numerical value corresponding to the probability of the state existing in that configuration when measured. For example, if we take the same state given in Section 2.1.3:

$$|\Psi\rangle = |00\rangle + |01\rangle - |10\rangle,$$

and call

```
phi = State({'00':1, '01':1, '10':-1})
phi.get_prob('00')
```

We get the value of  $\frac{1}{3}$ .

Similarly, `get_probabilities()` does a similar calculation but instead returns a numpy column array of probabilities. The returned array will contain  $2^n$  elements,  $n$  being the number of qubits, where each position in the array will correspond to a specific configuration according to its binary representation. Figure 3 below depicts the process described above.

```
x = State({'00':1, '01':1, '10':1})
x.get_probabilities()

array([[0.33333333],
       [0.33333333],
       [0.33333333],
       [0.         ]])
```

Figure 3: Here, we have a State  $x$  equal to  $|00\rangle + |01\rangle - |10\rangle$ , the original state. Then, when `get_probabilities()` is called on it, it returns the corresponding probability of the state existing at a specific configuration once measured, in this case,  $\frac{1}{3}$  for each configuration.

## 2.2 Sub-classes of State

### 2.2.1 Uniform State

The uniform state is a special case of a state. This is where each possible configuration has an equal probability of existing once the system is measured. For example, the uniform state of a 2 qubit system would be

$$|00\rangle + |01\rangle + |10\rangle + |11\rangle$$

in dirac notation.

## 3 The class Operator and its subclasses

(todo: review of the class methods + examples, such as the use of `*` operator)

The second class in our package we would like to talk about is the Operator class. To construct the operator in our class, all that is needed is the array representation of the operator.

## 3.1 Methods in Operator Class

### 3.1.1 Operations

Similar to the state class, there are a few basic mathematical operations we can perform on Operators. This includes addition, subtraction, multiplication, tensor product, and equality.

In terms of addition, subtraction, and equality, it is relatively straightforward: it is the addition or subtraction of the matrix representation between two operators and then return another operator while the equality function will determine whether or not two operators are the same.

For multiplication, we can multiply an operator with another operator, a state, or a number. For multiplying an operator with another operator, it is the same as multiplying the matrix representation of both and forming a new operator object. As for multiplying a state with an operator, it is the same process of turning one state into another. For example, if we apply the  $\text{Sigma}_X(\sigma_x)$  operator on a state  $|0\rangle$ , we would get  $|1\rangle$  as the resulting state. An important thing to note here is that order does matter. This is because matrix multiplication is not always commutative, and typically, we can only multiply an operator and a state in that order, not the other way around. Similarly, this is a rule that applies to the code. It is also possible to multiply a constant with an operator which will return another operator where each entry is scaled by the constant.

Finally, we have the tensor product. This will be come up again and will be elaborated on when we talk about our Class Algorithm in section 4. However, simply, the tensor product allows us to easily operate an operator on a multi-qubit state.

### 3.1.2 Complex Conjugate Transpose and isHermitian Method

The complex conjugate transpose in matrices is simply the complex conjugate, which is further touched on in section 5.2 in our ComplexNum class methods, of all entries of a matrix and then taking the transpose of it. Mathematically speaking, we can describe this using the following equation:

$$A_{ij}^H = \overline{A_{ji}}$$

Where  $A_{ij}^H$  is the complex conjugate transpose of entry  $(i, j)$  of matrix  $A$  and  $\overline{A_{ji}}$  is the complex conjugate of entry  $(j, i)$  of matrix  $A$ . This holds true for any values for  $i$  and  $j$  as long as the pair does represent an entry of the matrix.

This will then lead us into the `isHermitian()` method which allows to test us if an operator is Hermitian or not. Basically, for an operator to be Hermitian, its matrix representation is equal to its complex conjugate transpose. In other words,

$$A = A^H$$

where  $A$  is the matrix representation of a Hermitian operator.

### 3.1.3 Eigenvectors and Eigenvalues

The final two methods in this operator class are `eigenvalues()` and `eigenvectors()`. For the `eigenvalues` method, when called, it will return the eigenvalues of an operator in the form of a numpy array. Likewise, the `eigenvectors` method would also return a numpy array containing all the eigenvectors when each column is an individual vector.

## 3.2 Sub-classes of Operator Class

Within the Operator class, there are also many sub-classes, all of them being operators that can be implemented as gates in a quantum circuit which we will discuss in section 4. The following table shows the available operators in our class, how to call each one, and the matrix representation:













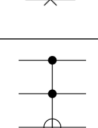

Operator	Matrix	Quantum Gate	BraQit
Sigma-X ( $\sigma_x$ )	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$		Sigma(1)
Sigma-Y ( $\sigma_y$ )	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$		Sigma(2)
Sigma-Z ( $\sigma_z$ )	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$		Sigma(3)
Hadamard	$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$		Hadamard()
Phase	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$		Phase()
$\pi/8$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{\pi i}{4}} \end{pmatrix}$		Pi8()
Controlled-Not (CNOT, CX)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$		CNOT()
Controlled-Y (CY)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix}$		CY()
Controlled-Z (CZ)	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$		CZ()
SWAP	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$		SWAP()
Toffoli (CCNOT)	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$		Toffoli()
Identity (1 qubit)	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$		Identity(1)

Table 1: Here are the available quantum logic gates implemented in our package

In addition to these operators available, our class also contains the projection operator. There are two parameters: a state and a notation. The state is the state we would project on and the notation is any notation which would appear in the representation of the projection. For example, if we wanted to create a projection operator on the state  $|00\rangle$ , we can call

```
Projection(State({'00':1}), '00')
```

When we print this,  $|00\rangle\langle 00|$  would be the corresponding representation dis-

played. The notation here can also be different, in fact any string like ' $\Psi$ ' would be allowed as this parameter has no affect on the matrix of the projection operator, just on the representation.

## 4 The class Algorithm

This is the final and one of the main classes in the package. What this class allows us to do is in fact create a virtual quantum circuit. To create one, we first need to create a list. Each element of the list would be the tensor product of all the operators we would perform on a specified state. For example, if we take a 2 qubit state,  $|00\rangle$ :

```
st = State({'0':1})
```

and call Hadamard on the 2 qubits, we would create a list as the following:

```
l = [Hadamard().tensor(Hadamard())]
```

The reason why we have the tensor product here is that it actually allows us to perform the operators, in order, on each individual qubit (or multiple qubits if the operator does so) in the state. Finally, we can call

```
Algorithm(l)*st
```

And it would return a final state. An important this to be aware about is the list. When quantum circuits are larger, the list would contain multiple elements. As a result, we have to ensure that each element is compatible with the state. This is because we can't call the list in the previous example on a 1 qubit state since it is only compatible with 2 qubits.

### 4.1 Examples:

#### 4.1.1 Constructing the Greenberger–Horne–Zeilinger State

The Greenberger–Horne–Zeilinger state (GHZ state for short) is given by

$$|\Psi\rangle = |000\rangle + |111\rangle \quad (1)$$

We illustrate below how GHZ state is constructed using BraQit:

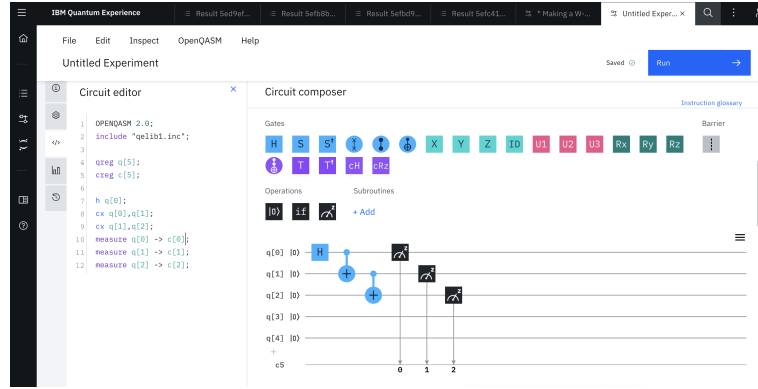


Figure 4: The construction of GHZ on IBMQ. Left: the OPENQASM code.

In BraQit, the above circuit is implemented as follows:

```

L = [Hadamard().tensor(Identity(2)), CNOT().tensor(Identity(1)), \
      Identity(1).tensor(CNOT())]
st = State({'000':1})
Algorithm(L)*st

```

```

L = [Hadamard().tensor(Identity(2)), CNOT().tensor(Identity(1)), \
      Identity(1).tensor(CNOT())]
st = State({'000':1})
Algorithm(L)*st

0.7071067811865475|000> + 0.7071067811865475|111>

```

Figure 5: The construction of normalized GHZ state using BraQit.

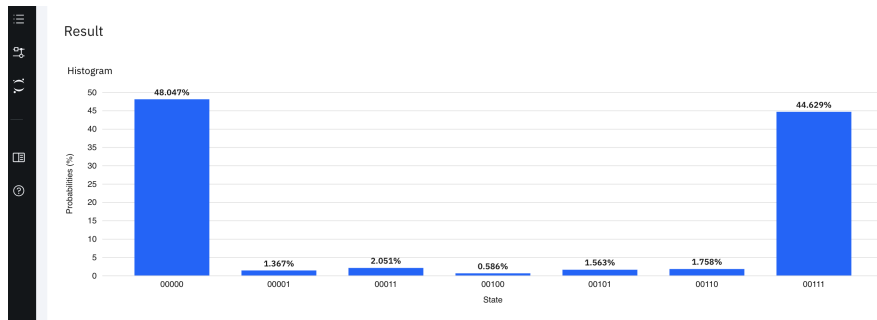


Figure 6: The construction of GHZ on IBMQ (backend = ibmq\_ourense). Final result.

#### 4.1.2 Grover's search

In this example, we show how to write Grover's search algorithm in BraQit. Grover's algorithm, in short, is an algorithm that can help us find an element in a  $N$ -sized, unsorted database in  $O(\sqrt{N})$  steps, quicker than classical computers that can take up to  $O(N)$  steps. The difference in run-time is more and more apparent as the search space increases, making this a much more efficient search algorithm.

For the algorithm to work, we need to have an oracle. An oracle is a "black box" that can tell us whether a result of something is the answer to a specific problem. In the case of the Grover search, the oracle is defined by the following action:

$$|x\rangle \xrightarrow{O} (-1)^{f(x)}|x\rangle$$

where  $f(x)$  is 1 if  $x$  is a solution to the problem and 0 if it is not. One important note is that the oracle function is not part of the algorithm. Rather, it's something which we call and it will perform an operation accordingly. After knowing about the oracle, we can now construct the Grover search which starts off with  $n$  qubits, where  $n = \log_2(N)$ , all set to the state of 0. Then, we will call Hadamard on all of them to put the qubits in an equal superposition of all states. Afterwards, we will call the Grover iterator which can be represented by

$$O(H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n})$$

Where  $O$  is when we call the oracle on the state,  $H^{\otimes n}$  is the Hadamard operator applied to all qubits, and  $I$  is the identity matrix. This expression can be simplified to:

$$O(2|\psi\rangle\langle\psi| - I)$$

Where  $\psi$  is the uniform superposition of all states. However, we will showcase how our code can create the quantum circuit described in the first equation of the Grover iterator.

After knowing the Grover Iterator, we then call it  $\sqrt{N}$  times. What should come as a result is a relatively high probability of getting one state when measured. This can all be implemented using our package:

Suppose for instance the database is  $\{|z\rangle\}_{z \in \{0,1\}^4}$ , i.e., the unordered set of all binary strings of size 4, and we are looking for the item  $|0101\rangle$ . In order to illustrate the working of our implementation of Grover search, let us first fake an oracle<sup>1</sup> specific for this example:

```
Oracle: lambda s: s.clist['0101'] == - s.clist['0101']
```

---

<sup>1</sup>Recall that a "real" oracle is a black box (e.g., a physical device which its inner working is hidden to us), to which we can only pass yes/no queries and get answers.

Now, we can create the second part of the Grover iterator:

```
Hn = Hadamard().tensor(Hadamard().tensor(Hadamard().tensor(Hadamard()))
G = [Hn, 2*Projection(State({'0000':1}), '0000') - Identity(4), Hn]
```

Here, `Hn` and `G` are specified for this example as this code wouldn't be compatible with a search space that doesn't include exactly 4 qubits. However, this code can easily be generalized. Afterwards, we can compile the code and create the Grover search:

First we have the state  $|0000\rangle$  and we call the Hadamard gate on all qubits::

```
st = State({'0000': 1})
st = Hn*st
```

Then, we can perform the Grover iterator  $\sqrt{N}$  times, which is, in our case, 4 times:

```
for i in range(4):
    oracle(st)
    st = Algorithm(G)*st
```

And after we return the state, we are done. Using only a few lines of code, we have managed to code the Grover search. To verify if the algorithm actually worked, we can call:

```
st.get_probabilities()
```

And we should receive a high probability of getting the state  $|0101\rangle$  relatively to the possible states. This shows how user friendly and efficient the package is.

## 5 ComplexNumber class

We intended to make `BraQit` self-contained so we have added the `ComplexNumber` class for complex numbers.

In our complex number class called `ComplexNum`, there are two instance variables, `re` and `im`, representing the real and imaginary part of complex numbers respectively. In addition, when we call the constructor of the `ComplexNum` class, the first parameter corresponds to a real part and the second parameter corresponds to the imaginary part. For example, if we wanted to create a complex number, say  $1 + 2i$  we can call

```
ComplexNum(1,2)
```

to create it using the `ComplexNum` class.

## 5.1 Operations in ComplexNum

ComplexNum has several overridden methods, similar to the other classes. This includes the addition, multiplication, subtraction, and equality method. As it may be expected, these methods allow us to perform operations between objects of our ComplexNum class. It is also compatible with the standard numerical value. For example, if we wanted to get  $2(1 + 2i)$ , to code it, it is simply

```
2*ComplexNum(1,2)
```

This is one advantage of the ComplexNum class because it allows us to compute, easily and conveniently, complex numbers of which pop up in quantum computation very often.

## 5.2 Complex Conjugate Method

This complex conjugate method, denoted as `conj()`, is another method in this ComplexNum class. In mathematics, the complex conjugate of a complex number is another complex number with the opposite sign on the imaginary part. In other words, if we take the general representation of a complex number,  $a + bi$ ,  $a$  being the real part and  $b$  being the imaginary part, that the complex conjugate of  $a + bi$  is  $a - bi$ .

## 6 Feedback

Feedback and comments are welcome, and can be sent to [michaelli2005li@gmail.com](mailto:michaelli2005li@gmail.com).

## References

- [1] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.