Quantum circuits in Python using nothing but Numpy

Joris Kattemölle

There are fancy packages that can (classically) run small quantum circuits in Python, such as Cirq, Qiskit, ProjectQ and even QuTiP. It is, of course, possible to do this without using any of these packages. I think this is much more enlightening. Here, I show how it can be done in a minimalistic, but high-performance way. Though simple, the code below is actually competitive with aforementioned packages in terms of speed.

In the following, I'll assume you have a good understanding of quantum mechanics, and a basic working knowledge of Python and Numpy. I'll use expressions like a = `a` , where a is in standard mathematical notation, and `a` is a representation of that same thing in Python. The following is going to involve a lot of explanation, but in the end only a few lines of code, and overview of which can be found at the end of this page. This page can be downloaded as a Jupyter notebook or as a pdf.

Contents:

States

For simplicity, let us consider $n = 4$ spin-1/2 particles, or qubits. Any state $|\psi\rangle$ can be expanded as

$$|\psi\rangle = \sum_{ijkl} \psi_{ijkl} |i\rangle|j\rangle|k\rangle|l\rangle.$$

To store this state in Python we just store the values $\{\psi_{ijkl}\}$ in a Numpy array `psi` in such a way that ψ_{ijkl} = `psi[i,j,k,l]` .

Note that the shape of `psi` is `(2,2,2,2)`. Or in other words, it has four indices, each of which can take two values.

Example 1

Let $|\psi\rangle = (|0000\rangle + |1111\rangle)/\sqrt{2}$, or in component notation: $\psi_{0000} = \psi_{1111} = 1/\sqrt{2}$ with all other components vanishing. This state is stored as

```
import numpy as np
psi=np.zeros((2,2,2,2)) # Create an array of zeros with the right shape.
psi[0,0,0,0]=psi[1,1,1,1]=1/np.sqrt(2) # Set the right entries to 1/sqrt
```

Single qubit operators

The Hadamard operator is given by $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

```
H_matrix=1/np.sqrt(2)*np.array([[1, 1],
                                [1,-1]])
```

Say we want to act with this operator on the 0th qubit, $|\psi'\rangle = X \otimes I \otimes I \otimes I \; |\psi\rangle$. (We count qubits from left to right, starting from 0. The same holds for indices.) In component notation, this means we want to create an array with components

`psi_prime[i,j,k,l]` $= \psi'_{ijkl} = \sum_{i'} X_{ii'} \psi_{i'jkl}$.

The can be achieved in a high-performance way by using np.tensordot,

```
psi_prime=np.tensordot(H_matrix,psi,(1,0)) # Contract the 1st index of H
```

> Remark 1. Of course, we could also represent states as long, one-dimensional arrays, in the way you were probably thought in your quantum mechanics 101 course. In this case, the operator $X \otimes I \otimes I \otimes I \; |\psi\rangle$ would be constructed by taking four Kronecker products. However, this approach is not very efficient, nor very elegant, for systems with $n \gg 4$ qubits. This is because merely constructing the operator $X \otimes I \otimes \ldots \otimes I$ requires you to create an (albeit sparse) $2^n \times 2^n$ matrix.

There is an important caveat if we want to apply H to the 1st qubit (or 2nd or 3rd). Namely, by construction, `np.tensordot(H_matrix,psi,(1,1))`, which contracts the `1` st index of `H_matrix` with the `1` st index of `psi`, produces an array with entries `psi_prime[j,i,k,l]` $= \sum_{j'} H_{jj'} \psi_{ij'kl}$. Note the order of the indices:

`np.tensordot` contracts indices, but leaves the free indices in the order reading from left to right. This is not what we're used to in quantum mechanics. We don't want an operator to change the order of the qubits. To fix this, we have to move the indices (or axes) back in the right places with np.moveaxis. So, the correct way to construct `psi_prime` is by

```
psi_prime=np.tensordot(H_matrix,psi,(1,1)) # Contract the 1st index of H
psi_prime=np.moveaxis(psi_prime,0,1) # Put axes in the right place by pu
```

The register class

To make the process a bit smoother, let's define the register as a class, and define the Hadamard gate as a function that acts on this class. The class only contains the state of the register, `psi`, and the number of qubits, `n`. The state of the register is initialized to the n-qubit state $|0,0,\ldots,0\rangle$.

```
class Reg:
    def __init__(self,n):
        self.n=n
        self.psi=np.zeros((2,)*n) # make array of zeros with right shape
        self.psi[(0,)*n]=1 # put psi[0,0,...,0] to 1
```

We can now act with the Hadamard operator on the ith qubit by invoking the function

```
def H(i,reg): # Alter the array psi into the array where H has acted on
    reg.psi=np.tensordot(H_matrix,reg.psi,(1,i))
    reg.psi=np.moveaxis(reg.psi,0,i)
```

Examples

```
reg=Reg(4)
print(reg.psi.flatten())
```

```
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

```
reg=Reg(4)
H(0,reg)
print(reg.psi.flatten())
```

```
[0.70710678 0.         0.         0.         0.         0.
 0.         0.         0.70710678 0.         0.         0.
 0.         0.         0.         0.        ]
```

Two-qubit operators

We now extend the previous to two-qubit operations. As an example, take the CNOT operator, defined by

$$\mathrm{CNOT}|00\rangle = |00\rangle, \quad \mathrm{CNOT}|01\rangle = |01\rangle, \quad \mathrm{CNOT}|10\rangle = |11\rangle, \quad \mathrm{CNOT}|11\rangle = |10\rangle.$$

In component notation,

$\mathrm{CNOT}_{0000} = \mathrm{CNOT}_{0101} = \mathrm{CNOT}_{1011} = \mathrm{CNOT}_{1110} = 1$, with other components vanishing.

Here the 0th qubit is called the control qubit and the 1st qubit the target qubit. Say we want to implement $\mathrm{CNOT} \otimes I \otimes I|\psi\rangle$, or, in component notation, construct the array with components `psi_prime[i,j,k,l]` $= \psi'_{ijkl} = \sum_{k'l'} \mathrm{CNOT}_{ijk'l'}\psi_{k'l'kl}$.

Now, before we proceed, I want to note that more often than not, the CNOT operator is depicted as a 2×2 array,

$$\mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

```
CNOT_matrix=np.array([[1,0,0,0],
                      [0,1,0,0],
                      [0,0,0,1],
                      [0,0,1,0]])
```

This is the form you obtain following the method of Remark 1. However, for our purposes, this tensor does not have the right shape. It has only two axes, or indices, whereas we need four in the component formulas above. Fortunately, we can easily get it in the right shape by using np.reshape.

```
CNOT_tensor=np.reshape(CNOT_matrix, (2,2,2,2))
```

Again taking into account the right order of the indices, we can implement this operator by

applying the function

```
def CNOT(control, target, reg):
    # Contract 2nd index of CNOT_tensor with control index, and 3rd inde
    reg.psi=np.tensordot(CNOT_tensor, reg.psi, ((2,3),(control, target))
    # Put axes back in the right place
    reg.psi=np.moveaxis(reg.psi,(0,1),(control,target))
```

Example 2

When acting on `psi` $= |00\ldots\rangle$, the following function constructs the (generalization of) the state from Example 1.

```
def generate_GHZ(reg):
    H(0,reg)
    for i in range(reg.n-1):
        CNOT(i,i+1,reg)

# Usage
reg=Reg(4)
generate_GHZ(reg)
print(reg.psi.flatten())
```

```
[0.70710678 0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.70710678]
```

Measurement

A measurement of the ith qubit in the computational basis changes the state $|\psi\rangle$ into $|\psi'\rangle = \frac{|j\rangle_i\langle j|_i|\psi\rangle}{\||j\rangle_i\langle j|_i|\psi\rangle\|}$ with probability $p(j) = \| |j\rangle_i\langle j|_i|\psi\rangle\|^2$. (Here e.g. $\langle j|_0 = \langle j| \otimes I \otimes I \otimes I$).

First, let us implement $|j\rangle_i\langle j|_i$ for the two different vales of j. Given a value for j, the operator $|j\rangle_i\langle j|_i$ can be written as a matrix, and we may implement this matrix in the same way we've implemented `H_matrix`.

```
projectors=[np.array([[1,0],[0,0]]), np.array([[0,0],[0,1]]) ] # list co

def project(i,j,reg): # RETURN state with ith qubit of reg projected ont
    projected=np.tensordot(projectors[j],reg.psi,(1,i))
    return np.moveaxis(projected,0,i)
```

Now to emulate measurement, we need to act with the right projector with the right probability:

```
from scipy.linalg import norm

def measure(i,reg): # Change reg.psi into post-measurement state w/ corr
    projected=project(i,0,reg)
    norm_projected=norm(projected.flatten())
    if np.random.random()<norm_projected**2: # Sample according to proba
        reg.psi=projected/norm_projected
        return 0
    else:
        projected=project(i,1,reg)
        reg.psi=projected/norm(projected)
        return 1
```

Example 3

After applying a Hadamard to the 0th qubit of the all zero state, we have a 50% change of getting the outcome 0.

```
for i in range(100):
    reg=Reg(4)
    H(0,reg)
    print(measure(0,reg), end='')
```

```
11110101010011101110111100110101010000110000100000100001000101110000110 1
```

If we get the outcome j, but do not re-initialize the state after every measurement, the probability of getting the same outcome is 1.

```
reg=Reg(4)
H(0,reg)
for i in range(100):
    print(measure(0,reg), end='')
```

```
11111111111111111111111111111111111111111111111111111111111111111111111111111111
```

When we generate a Bell state (which equals the 2-qubit GHZ state) and measure the first qubit, a sequential measurement of the second qubit will *always* yield the same result.

```
for i in range(100):
    reg=Reg(4)
    generate_GHZ(reg)
    print(measure(0,reg),end='')
    print(measure(1,reg),end=' ')
```

```
00 00 00 00 00 00 11 00 00 00 11 00 00 00 00 11 00 11 00 11 11 00 00 11
```

Conclusion

In this article, we've seen how to implement basic quantum primitives such as state initialization, the application of small unitary operators such as H and CNOT, and measurement in the computational basis. Given a good understanding of these primitives, it is easy to define your own primitives, and with a few lines of extra code, you can already implement algorithms such as a variational quantum eigensolver. You could go on and write a proper quantum library that defines gates and circuits as classes, which is outside the scope of the current article. Then you could define a circuit without actually running it, so that you can draw the circuit graphically, or export the gate sequence as OpenQASM.

Code

As an overview, the code below presents all the code above, leaving out the examples and code that has been superseded.

```
import numpy as np
from scipy.linalg import norm

H_matrix=1/np.sqrt(2)*np.array([[1, 1],
                                [1,-1]])

CNOT_matrix=np.array([[1,0,0,0],
                      [0,1,0,0],
                      [0,0,0,1],
                      [0,0,1,0]])

CNOT_tensor=np.reshape(CNOT_matrix, (2,2,2,2))
```

```
class Reg:
    def __init__(self,n):
        self.n=n
        self.psi=np.zeros((2,)*n)
        self.psi[(0,)*n]=1

def H(i,reg):
    reg.psi=np.tensordot(H_matrix,reg.psi,(1,i))
    reg.psi=np.moveaxis(reg.psi,0,i)

def CNOT(control, target, reg):
    reg.psi=np.tensordot(CNOT_tensor, reg.psi, ((2,3),(control, target))
    reg.psi=np.moveaxis(reg.psi,(0,1),(control,target))

def measure(i,reg):
    projectors=[ np.array([[1,0],[0,0]]), np.array([[0,0],[0,1]]) ]

    def project(i,j,reg):
        projected=np.tensordot(projectors[j],reg.psi,(1,i))
        return np.moveaxis(projected,0,i)

    projected=project(i,0,reg)
    norm_projected=norm(projected.flatten())
    if np.random.random()<norm_projected**2:
        reg.psi=projected/norm_projected
        return 0
    else:
        projected=project(i,1,reg)
        reg.psi=projected/norm(projected)
        return 1

# Example of final usage: create uniform superposition
reg=Reg(4)
for i in range(reg.n):
    H(i,reg)

print(reg.psi.flatten())
```

```
[0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
 0.25 0.25]
```