Quantum Circuits

Rares-Alexandru Dragomir

Student ID: 10455094
PHYS30762 Object-Oriented Programming in C++
Physics Department, The University of Manchester

May 17, 2022

Abstract

The aim of this project was to build a quantum circuits simulator similar to IBM's Qiskit library using the C++ programming language. The project required basic understanding of quantum computing and of object-oriented programming (OOP) principles. The most fundamental goal in terms of functionality was to build a circuit that uses the Hadamard and CNOT gates to return an entangled Bell state. Once this was achieved, the problem of adding more quantum gates (both single-qubit and multi-qubit) became trivial. As the number of qubits increases, the computing power and memory requirements of simulating quantum circuits on classical computers grow exponentially. Since pen and paper calculations can only be done for circuits with a couple of qubits, the preferred way of checking the results was to run them against IMB's simulator.

1 Brief History of Computers

The history of classical computers starts with Alan Turing's 1936 paper titled "On Computable Numbers, with an Application to the Entscheidungsproblem", in which he proved the existence of a Universal Turing Machine. This

machine was defined as a theoretical apparatus that could simulate any algorithmic process that runs on another Turing machine, say a personal computer. His famous result now serves as the backbone of modern computer science and is known as the "Church-Turing thesis". Three decades later, it was postulated that the Turing model of computation was at least as efficient as any other alternative model, which lead to the strengthened version of the Church-Turing thesis: "Any algorithmic process can be simulated efficiently using a Turing machine." The problems with the new thesis became apparent immediately, when no deterministic efficient algorithm was found to solve the problem of finding the prime factors of an integer, a computational challenge on which present-day cryptography is based upon. Instead of looking for new corrections to Alan Turing's original idea, and inspired from the quantum mechanical nature of physical reality, David Deutsch came up in 1985 with an entirely new model of computation based on his idea of a Universal Quantum Computer. The greatest breakthroughs in quantum computing came only a decade later, with Shor's factorising algorithm, and with Grover's finding algorithm, both of which standing now as proof for the supremacy of quantum computing.[1]

2 Quantum Computing in a Nutshell

We start off by defining the quantum bit (or qubit) as the quantum variant of the classical bit, or more formally, as a superposition of two orthonormal states:

$$|q\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{1}$$

where $|0\rangle$ and $|1\rangle$ are the eigenstates of the Pauli-Z operator, and α and β are the associated probability amplitudes. Since the qubit state must be normalised,

$$1 = \alpha^2 + \beta^2. \tag{2}$$

The $|0\rangle$ and $|1\rangle$ states are represented in their own eigenbasis as:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \tag{3}$$

Note that, unless a different eigenbasis is specified, we will assume all single-qubit operators to be represented in the eigenbasis described by the Pauli-Z

operator:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{4}$$

We call the matrix above the Z gate. Other elementary single-qubit gates are the identity, the X gate, the Y gate, the T gate, and the Hadamard gate:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5}$$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{6}$$

$$Y = \begin{bmatrix} 1 & -i \\ i & 0 \end{bmatrix} \tag{7}$$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \tag{8}$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{9}$$

We will discuss briefly the intuition behind each of the mentioned single-qubit gates. The simplest one, the identity gate, leaves the qubit unchanged. The X gate flips the qubit on its head (e.g. $|0\rangle$ is transformed into $|1\rangle$), or in other words, performs a π rotation with respect to the x-axis. Unsurprisingly, the Y and Z gates also perform rotations by π around the y and z-axis of the Bloch sphere. The T gate is part of a family of phase shifting gates and only adds an innocuous global phase shift of $e^{i\pi/4}$ that is unobservable. But if we control the operation using another qubit, the phase is no longer global but relative, which changes the relative phase in the control qubit. The last gate mentioned above, the Hadamard gate (H gate), is a fundamental quantum gate, and it allows us to move away from the poles of the Bloch sphere and create superpositions of $|0\rangle$ and $|1\rangle$.

The only multi-qubit gate implemented in our project was the controlled-NOT gate (CNOT gate), which applies on two qubits, a control qubit and a target qubit. Its operation is trivial: if the control qubit is "true" (i.e. in the $|0\rangle$ state), the target qubit is flipped. This is the quantum equivalent of the classical XOR gate. For a system of only two qubits, where q_0 is the control

qubit and q_1 the target qubit, the CNOT gate is represented by:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{10}$$

Now that we have covered the most basic quantum gates, we are finally in position to build the quantum circuit that will give us the famous fully-entangled Bell state:

$$|\text{Bell state}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \tag{11}$$

which means that there is a 50% probability to measure the $|00\rangle$ state, and a 50% probability to measure the $|11\rangle$ state. More interestingly, if we only measure the first qubit, we then know the state of the second qubit with 100% certainty.[2] This is indeed what Einstein called "spooky action at a distance". And as previously stated in the Abstract, the fundamental goal of this project was to build a quantum circuit that imitates the functionality of the following Qiskit code:

```
[1]: from qiskit import QuantumCircuit

     qc = QuantumCircuit(2)
     qc.h(0)
     qc.cx(0, 1)
     qc.draw()
```

```
[1]:
```

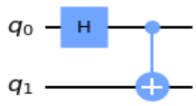

Figure 1: Figure showing Qiskit code run in IBM's Quantum Lab environment. The resulting state vector is the Bell state. Note that qubits are initialized by default in the $|0\rangle$ state.

3 Project Implementation

Since states and operators are represented by complex matrices, we started off the project by building a Matrix class that served as the base class for

almost all the classes that followed. (Note that this was already done as part of Assignment 5 of the same course. The only added functionality whose purpose will become apparent soon was the addition of a Kronecker product function.) As a small tangent, the reason why we chose not to make the base Matrix class purely virtual was because it would have over-complicated the implementation without adding much (if any) functionality to it. As an example, in the current implementation, the overloaded multiplication operator returns another Matrix-type object, but were we to define the Matrix class as purely virtual, we would have had to use unique pointers, and define the operation in the derived classes. This would have made the code more bloated and harder to understand, with the only benefit of having an excuse to use pointers.

Now suppose we have a 3-qubit circuit and we want to apply a Hadamard gate on q_1. What operator do we have to apply on the state vector? The answer is given by the Kronecker product:

$$|\text{final state}\rangle = (I \otimes H \otimes I) \, |q_2 q_1 q_0\rangle \,, \tag{12}$$

where we call the two terms on the RHS the "operator matrix" and the "state vector", respectively. Also, we kept the Qiskit convention of ordering qubits from right to left starting from 0. In our implementation, we defined a derived class for each of the single-qubit gates mentioned above, and the Kronecker product shown above is performed as a method of the QuantumCircuit class every time we apply a single-qubit gate on a qubit.

3.1 The CNOT Gate

The next challenge was to express the following mathematical operation in code:

$$|\text{final state}\rangle = CNOT_{02} \, |q_2 q_1 q_0\rangle \,, \tag{13}$$

where the $CNOT_{02}$ operator is represented by a 2^3 by 2^3 matrix with q_0 as the control qubit and q_2 as the target qubit. (Note that the state vector is represented by a 2^n by 1 vector, and the operator matrix that acts on it by a 2^n by 2^n matrix, where n is the number of qubits in the circuit.) In order to find the elements of the $CNOT_{02}$ matrix, we used the following identity:

$$CNOT_{02} = \begin{bmatrix} \langle 000| \, CNOT_{02} \, |000\rangle & \langle 000| \, CNOT_{02} \, |001\rangle & \dots \\ \langle 001| \, CNOT_{02} \, |000\rangle & \langle 001| \, CNOT_{02} \, |001\rangle & \dots \\ \dots & \dots & \dots \\ \langle 111| \, CNOT_{02} \, |000\rangle & \langle 111| \, CNOT_{02} \, |001\rangle & \dots \end{bmatrix}. \tag{14}$$

The example above shows the difficulty of implementing multi-qubit gates on circuits with an arbitrary number of qubits. However, it is worth saying that without multi-qubit gates, quantum computers would have no purpose. Below we show our C++ implementation of the construction of the CNOT gate of arbitrary size:

```
ControlNot::ControlNot(const int& nqubits,
    const int& controlq, const int& targetq) :
    Matrix{power(2, nqubits), power(2, nqubits)}
{
    std::bitset<MAXQUBITS> ket;
    std::bitset<MAXQUBITS> bra;
    for (size_t i {0}; i < rows; i++) {
        for (size_t j {0}; j < cols; j++) {
            ket = std::bitset<MAXQUBITS>(j);
            bra = std::bitset<MAXQUBITS>(i);
            if (ket[controlq] == 1)
                ket.flip(targetq);
            if (ket == bra)
                mtrx_data[index(i, j)] = 1;
            else
                mtrx_data[index(i, j)] = 0;
        }
    }
}
```

Note that the std::bitset conversion function requires to know the maximum number of bits at compile time. This motivated us to define the MAXQUBITS variable as a preprocessor directive, and due to memory and computing power constraints, we set this value at 8 (i.e. our circuit will accept a maximum of 8 qubits). The motivation behind the conversion to binary came from our realisation that the indices of the bras and kets in Equation (14) are the matrix indices converted to binary:

$$CNOT(i,j) = \langle \text{std::bitset(i)} | \, CNOT \, | \text{std::bitset(j)} \rangle \, . \tag{15}$$

3.2 The QuantumCircuit Class

In order to allow the user to initialize the qubits, we created a SingleQubit class. Its main functionality was to normalize the probability amplitudes

specified by the user, according to Equation (2). Now with the derived classes out of the way, the only thing left at this stage was to tie everything together in a QuantumCircuit class that *has* all the classes mentioned above. Before applying a quantum gate on any of our qubits, the single-qubit states were saved individually in an array of SingleQubit objects, to allow the user to initialize each qubit separately. By default, the circuit initializes all qubits to the $|0\rangle$ state. Note that once a gate is applied, the state vector is built according to:

$$|\text{state vector}\rangle = |q_{n-1}\rangle \otimes |q_{n_2}\rangle \otimes \cdots \otimes |q_0\rangle \,, \tag{16}$$

and after the creation of the state vector, the qubits cannot be initialized separately anymore (a safety measure we took against the possibility of the user attempting to initialize entangled qubits). Furthermore, another important method of the QuantumCircuit class was the .measure_all() function which prints out all the measurement probabilities, and also collapses the state vector onto the $|00\ldots0\rangle$ state. Note that we tried to avoid causing confusion by keeping Qiskit's naming conventions of public methods. Lastly, it should be mentioned that we built a power function equivalent to std :: pow that returns size_t types to compute matrix dimensions. We chose not to override the pow function to avoid using namespaces for only one instance.

4 Results and Project Outcomes

Overall, this project served as a great introduction to both quantum computing and to OOP principles. The main aim of building a Bell state was achieved, and the implementation was tested successfully against IBM's Qiskit simulator, as can be seen in the Examples/bell_state.cpp file. Another more complicated example and its Qiskit simulation can be found in Examples/main.cpp and in Examples/main.py, respectively. In terms of code features, the project made use of header files, error handling, default function arguments, and standard libraries such as <bitset> and <iomanip>.

```
(base) rares@nitro:~/Uni/OOP in C++/Quantum_circuits$ ./main.exe
Measurement probabilities:
Probability of |00> is: 0.5
Probability of |11> is: 0.5
```

Figure 2: Figure showing the successful output of measuring the Bell state.

5 Discussion

The room for improvement for the current version of the project is vast, ranging from adding noise and individual-qubit measurement gates, to implementing complicated algorithms such as Shor's or Grover's that can prove quantum supremacy. Parallelization can also be added to boost the performance of the code. As a final note, we would like to encourage the user to build their own circuits, and to code their own gates on top of the current infrastructure.

References

[1] Nielsen MA, Chuang IL. Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press; 2011.

[2] Abbas A, Andersson S, Asfaw A, Corcoles A, Bello L, Ben-Haim Y, et al.. Learn Quantum Computation Using Qiskit; 2020. Available from: http://community.qiskit.org/textbook.