

THE UNIVERSITY OF MANCHESTER

QUANTUM CIRCUITS USING C++

Noah Allouane (10735842)

Oxford Rd, Manchester M13 9PL,
England
Email: noah.allouane@student.manchester.ac.uk

Abstract

The following report describes the creation of a quantum circuit using various features of C++. The report aims to outline how the circuit was built and how it can be used to model different configurations using a built-in library of quantum gates. Future work could extend this code creating higher user functionality and a more rigorous error-handling method.

Contents

1	Introduction	2
1.1	Quantum Computing	2
1.2	Quantum Circuits	2
2	Code	4
2.1	File Division	4
2.2	Matrices and Complex numbers	4
2.3	Gates	5
2.4	Circuit	5
3	Results	6
4	Discussion	7
4.1	Problems	7
4.2	Future Developments	7
4.3	Conclusion	8

1 Introduction

1.1 Quantum Computing

Quantum computing has been a rapidly developing field in recent years, computational power for these quantum computers is thought to offer faster and more efficient speeds as compared to their classical counterparts[1].

Quantum computers work in a whole new way of information processing. Conventionally, classical computers operate using a binary system, representing ‘bits’ in either a 0 state: $|0\rangle$ or a 1 state: $|1\rangle$. The state of the bit will then represent a piece of information. Quantum computing on the other hand focuses on applying principles of quantum mechanics, using the rules of superposition. They work on ‘qubits’, these qubits can exist in both of the base states of $|0\rangle$ and $|1\rangle$ at the same time. The state of the qubits can be represented by a combination: $\alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$. $|\alpha|^2$ and $|\beta|^2$ will then respectively represent the probability amplitudes of the system being in the $|0\rangle$ or $|1\rangle$ state. This is normally represented as a vector:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

In a multiqubit system, these states are represented by taking the Kronecker product of the individual qubit states. The Kronecker product for a 2 qubit system: $(\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle)$, where $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \mathbb{C}$. This can be expressed in vector form as:

$$\begin{pmatrix} \alpha_1\alpha_2 \\ \alpha_1\beta_2 \\ \beta_1\alpha_2 \\ \beta_1\beta_2 \end{pmatrix}$$

Now, when we observe (measure) the system, the quantum state, previously mixed in a superposition state, decides to choose one of the base states. This event, known as the collapse of the quantum state, ultimately gives us the specific piece of information we’re looking for.

1.2 Quantum Circuits

In order to use these quantum computers we need a way to represent a set of operations on the state of a qubit system. The set of these operations as well as the order they act in can be categorised as a circuit [3]. Though the word circuit may have connotations of a ‘circular’ picture, the quantum circuit is ‘acyclic’ which means that it does not form a circle. A quantum circuit is represented by horizontal parallel lines, each line representing a qubit in the system. Along these lines, ‘gates’ are placed corresponding to the qubit on which they act. The distance along the horizontal axis represents the order in which the operations are applied, meaning gates are applied to the corresponding qubits from left to right.[3]. An example of this can be shown in Figure 2.

The actions of these quantum gates on qubits can be represented by matrices, which operate on the state of the qubits.














Operator	Gate(s)	Matrix
Pauli-X (X)	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Figure 1: Common Quantum Gates [2]

Some of the standard gates are shown in Figure 1. Again in a multiqubit system, one can take the Kronecker product of gates that act parallel to each other, which means one gate that acts directly under another.

For instance, given two 2x2 matrices A and B :

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

and

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

The Kronecker product of A and B is denoted by $A \otimes B$, is given by:

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{pmatrix}$$

Note that the use of the identity matrix as a way to represent the ‘lack’ of a gate amongst parallel gates is useful when taking a matrix transformation in the context of the entire system[3].

After all the gates are applied by taking their respective operations, you can take the total product of the matrices that each parallel block of gates produces. This forms a singular matrix which represents the entire transformation of the circuit.

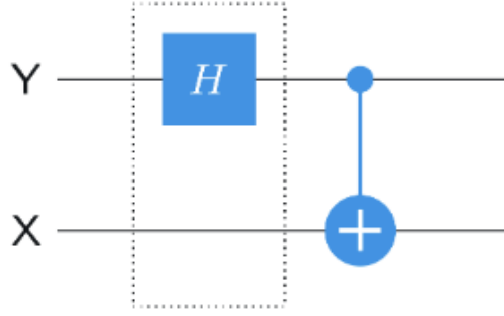


Figure 2: Shows an example of a quantum circuit acting on a 2-qubit system. The ‘H’ represents the Hadamard gates and the other represents a CNOT gate[3]

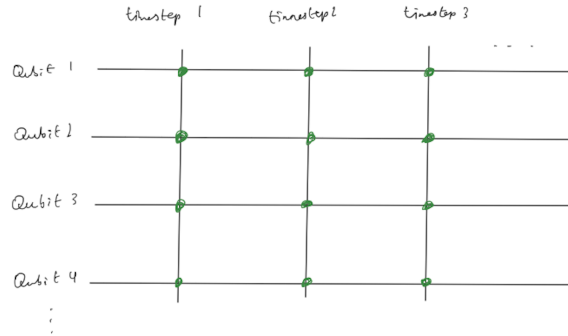


Figure 3: Shows the thought process behind the building of the circuit, the green dots represent the quantum gates.

2 Code

2.1 File Division

In the development of this code, the structure was split up into three distinct sections:

1. Configuration of the **Matrix** class including the utilization of complex numbers, coming from the **Complex** class.
2. Creation of derived classes for gates, which was accomplished through the creation of the **QuantumComponent** abstract base class.
3. Construction of the quantum circuit.

The partitioning of the code into separate header files, namely **Complex.h**, **Matrix.h**, **Gates.h**, and **Circuit.h**, enhanced the organization and navigability of the codebase. This strategy facilitated prompt and efficient access during troubleshooting, thereby streamlining the debugging process.

2.2 Matrices and Complex numbers

Matrix functionality was added by using the ‘**Matrix**’ class from a prior assignment, to allow for the operation of the quantum gates on the qubits. Modifications were made

such as adding a “Kronecker product” method which was used to calculate the ‘unitary operation’[3] at each timestep.

Complex numbers were needed for the matrices of gates such as ‘Pauli-Y’, using the ‘Complex’ class, again from a prior assignment, allowed for complex entries. Changes to the Matrix class were made to achieve functionality with complex numbers, for example, one change was to change the type for the matrix data:

```
double* matrix_data ---> Complex* matrix_data;
```

2.3 Gates

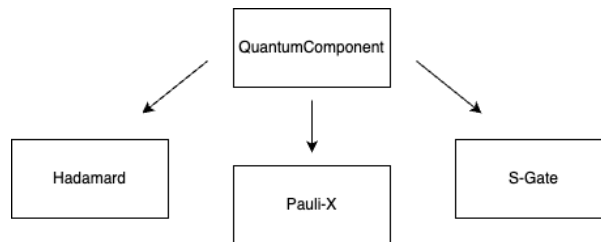


Figure 4: Example of some of the derived classes from QuantumComponent, the diagram shows the class hierarchy.

The implementation of components in the circuit comes from creating an ‘abstract base class’ called ‘QuantumComponent’ this allowed for the creation of derived classes to represent gates, for this to work virtual functions for ‘getMatrix’ and ‘getName’ were created. This class hierarchy is shown in Figure 4. The use of a virtual ‘clone’ function was also added, this allowed for the creation of a copy constructor in the next sections Circuit class. All the derived classes that were created such as Pauli gates, Hadamard, etc. overrode these with their respective matrix representation.

```

class QuantumComponent {
public:
    virtual ~QuantumComponent() = default;
    virtual Matrix getMatrix() const = 0;
    virtual std::string getName() const = 0;
    virtual std::shared_ptr<QuantumComponent> clone() const = 0;
    //clone method to allow copy of circuits
}
  
```

The code also made use of the QuantumComponentFactory class, this was added as a way for the user to create a library of gates from a selection in the ‘Gates.h’ file. This will be used later in the ‘Results’ section when mentioning the configureCircuit method.

2.4 Circuit

The circuit itself was built by conceptualizing it as a 2x2 grid as shown in Figure 3. All circuits that are built are instances of a class ‘Circuit’ which include methods that build it up step by step. To create this grid, two vectors were used for 2x2 entries:

```
std::vector<std::vector<std::shared_ptr<QuantumComponent>>>
    Qcircuit
```

The entries of the outer vector represented each parallel block, with the position in this vector representing the order in which these blocks occur, which in the code is referred to as the ‘timestep’. The inner vector then has the entries of gates, the positions in the vector represent the qubit the gate is acting on. the gates are added to using a `shared_ptr` which allows for dynamic memory allocation by deleting the data after it goes out of scope.

In creating a constructor for the circuit, the user initially chooses the number of qubits for their circuit from the start, in doing this the constructor is set up to create a singular ‘inner vector’, so at each singular time step, each entry of the vector made to be the identity gate.

Quick note: The `addComponentToLibrary` method is used to create a library of gates for the user, this allowed gates to be added to:

```
std::vector<std::shared_ptr<QuantumComponent>> componentLibrary
```

In order to add gates to the circuit, the constructor-placed identity gates are overridden using the ‘`addGate`’ method. The method takes in the gate the user wants to place and the qubit they want to place it on, along with the timestep, if the timestep specified is further down the line, parallel block vectors will be created till it reaches that timestep, allowing for an infinitely long circuit.

```
addGate(std::shared_ptr<QuantumComponent> gate, int qubit, int
    timestep)
```

Upon completion of gate additions, the user can compile their quantum circuit using the `applyCircuit` function. This function calls the methods `calculateTimestepMatrix` and the `calculateTotalMatrix`. These functions find the matrix transformations represented at each timestep, done by taking the Kronecker product of all the parallel blocks in the circuit and then multiplying these matrices in reverse order of their timestep. Finally, the `stateVector`, which is initialized to all the qubits being in the $|0\rangle$ state, is then multiplied by this matrix as an example of showing the transformation of the basis state.

Storing the data in a vector format was appreciated when it came to printing a schematic of the circuit to the terminal. This is all represented in the `printCircuit` function, using ASCII art it really added to the project as it allowed for direct visualisation of the code.

3 Results

The produced code successfully outputs a functioning quantum circuit. The creation of ‘`configureCircuit`’ is designed to allow for a user-friendly experience. The user is prompted on what gates they want to add to their library, stored in:

```
std::vector<std::shared_ptr<QuantumComponent>> componentLibrary
```

After creating your library, you are prompted to add gates to your circuit, asking for the timestep and qubit it occupies. The following code which is shown in Figure 5, displays

the output of adding a ‘Hadamard’ gate to the first qubit at the first timestep, giving inputs of (0,0), and adding a CNOT gate at the second timestep occupying the first and second qubit. Important note: The user must add the `CNOTcontrol` and the `CNOTtarget` into your circuit. This will be the same with the Toffoli gate, requiring two `CNOTcontrol` and a `Toffolitarget` gate. Therefore, the `CNOTcontrol` gate was added at (0,1) and the `CNOTtarget` was added at (1,1).

```
State |0>: Amplitude = 0.707, Probability = 0.5
State |1>: Amplitude = 0.707, Probability = 0.5
Quantum Circuit:
Number of qubits: 2
Number of timesteps: 2

Qubit 1: |Hadamard   |-----|CNOTcontrol|-----
Qubit 2: |.          |-----|CNOTtarget |-----
```

Figure 5: Shows the output achieved by adding a Hadamard and CNOT gate in series.

4 Discussion

4.1 Problems

As mentioned in the previous section, a notable challenge encountered was with multi-qubit gates. For example, CNOT spans 2-gates. In the code this was resolved by splitting the gates into a control and a target; the latter being the 1x1 identity matrix. Consequently, when the Kronecker product is taken, the desired gate is achieved. However, there is no error handling of this gate meaning you can add the individual control and target gates wherever you want, which can break the code. Given more time, this could be handled by verifying what gates are allowed at certain grid points.

4.2 Future Developments

The code is meant to showcase a representation of what can be done using C++ code, meaning significant potential exists to scale this code.

To facilitate potential future projects and code reuse, a copy constructor and a copy assignment operator were introduced. These allow for the replication of one circuit onto another, proving particularly useful when duplicating complex circuits.

```
Circuit(const Circuit& other);
Circuit& operator=(const Circuit& other);
```

Functionality to push back gates if there is space for them in the previous time step can be added allowing for a cleaner output. Of course, adding more gates will allow for bigger projects and more complicated circuits.

Measurements could be added to see the passing of information, though at this level showing the probabilities of outputs made more sense.

4.3 Conclusion

This project was selected due to the ever-changing space of quantum computing. Last year, IBM announced their 433 qubit processor called ‘Osprey’ [4]. This project seemed to be an interesting way to understand how these quantum computers work.

To take this quantum circuit simulator seriously and make it comparable to industry-standard simulators such as IBM will generally require a lot of work; more gates, extensive error handling and a better user interface would be a start. However, this code shows a good representation of what can be achieved using principles of object-oriented programming applied to the world of quantum circuits.

References

- [1] Qiskit. Why quantum computing?, 2023. Accessed: 2023-05-15.
- [2] Wikipedia contributors. Quantum logic gate — Wikipedia, the free encyclopedia, 2023. [Online; accessed 12-May-2023].
- [3] Qiskit. Quantum circuits, 2023. Accessed: 2023-05-10.
- [4] IBM. Ibm unveils 400-qubit plus quantum processor and next-generation ibm quantum system two, November 2022.