C++ Quantum Circuits Project

Aebel Shajan

May 2023

Abstract

This project aims to document the implementation of a quantum circuit simulator in C++. The report is divided into five sections: introduction, quantum theory, implementation, results, and conclusion. In the introduction, we provide an overview of quantum computing and its potential advantages over classical computing. In the theoretical background section, we then describe the basic elements of quantum circuits, such as qubits, gates, and measurements. In the implementation section, we go over the code design and the different types of classes used. In the results section, we present the results of our quantum circuit simulations. We demonstrate that our simulator can accurately simulate simple quantum circuits.

[1]

[1] The total word count is below 2500 excluding code. Check by going here: `https://www.overleaf.com/read/dcprbhbftntf` and doing Menu-¿Actions-¿Word count.

Contents

1 Introduction

Quantum computing is a rapidly growing field that has the potential to revolutionize many areas of science and technology. At the heart of quantum computing are quantum systems, which are the basic building blocks of quantum computers. In this report, we will explore quantum systems and their role in quantum circuits.

2 Quantum theory

2.1 Fundamental mathematics

In classical computing, data is represented using bits. Bits are binary and can only either be 1 or 0. Instead of bits we use qubits in quantum computing. A qubit can be 1 or 0 just like as in classical computing. We denote qubit zero by $|0\rangle$ and qubit one by $|1\rangle$. However, unlike classical computing, qubits have the extra ability to be in a superposition of both $|0\rangle$ and $|1\rangle$ at the same time. Mathematically for an arbitrary qubit $|q\rangle$ this can be formulated as,

$$|q\rangle = a\,|0\rangle + b\,|1\rangle\,, \tag{1}$$

where $a \in \mathbb{C}, b \in \mathbb{C}$ are complex numbers such that $a\bar{a} + b\bar{b} = 1$. The reasoning for this condition will become apparent later. The qubit $|q\rangle$ can be represented as 2D vectors in the complex plane $\mathbb{C}^2$. Let $|0\rangle$ and $|1\rangle$ be the basis vectors of $\mathbb{C}^2$:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2}$$

Then we have:

$$|q\rangle = \begin{pmatrix} a \\ b \end{pmatrix}. \tag{3}$$

The vector $|q\rangle$ transposed and then complex conjugated is represented by $\langle q| \in \mathbb{C}^2$. When we measure the qubit $|q\rangle$, it collapses into either state $|0\rangle$ or $|1\rangle$. The complex coefficients present in (1) determine the probability with which it collapses. Given a state $|q\rangle$ the probability of it being in another state $|u\rangle$ is given by:

$$Pr(\text{State measured as } |u\rangle) = |\,\langle u|q\rangle\,|^2. \tag{4}$$

So the probability that $|q\rangle$ collapse into qubit zero $|0\rangle$ is given by:

$$Pr(\text{State measured as } |0\rangle) = |\,\langle 0|q\rangle\,|^2 = a\bar{a}. \tag{5}$$

Similarly for $|1\rangle$ we have:

$$Pr(\text{State measure as } |1\rangle) = |\,\langle 1|q\rangle\,|^2 = b\bar{b}. \tag{6}$$

The total probability of obtaining either basis state is equal to 1. Hence the condition: $a\bar{a} + b\bar{b} = 1$.

2.2 Operations as gates

In classical computing operations can be performed on bits which change the state each bit is in. These operations are performed with logic gates, which can be aggregated together to make a full circuit. Examples of logic gates are shown in Figure 2 with an example circuit in Figure 2.

Operations can also be performed on qubits through quantum gates. These gates can then be aggregated into a quantum circuit. Similar to how we represented qubits by 2d complex vectors, we can represent quanum operators as 2d complex square unitary matrices. Examples of some operators are shown below:

$$\text{Hadamard Gate: } H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}, \qquad \text{Not Gate: } X = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix},$$

$$\text{Pauli Y Gate: } Y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}, \qquad \text{Pauli Z Gate: } Z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, \quad (7)$$

$$\text{Phase Shift Gate: } P(\lambda) = \begin{pmatrix}1 & 0\\ 0 & e^{i\lambda}\end{pmatrix}.$$

2.3 Multi-qubit systems

When we have multiple qubits, the combined state of the system is represented by the tensor product of the individual qubit states. For example, suppose we have two qubits, q1 and q2, which can each be in the states $|0\rangle$ or $|1\rangle$. Then, the combined state of the system can be written as,

$$|q1\rangle \otimes |q2\rangle,$$

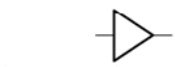
where $\otimes$ represents the tensor product. This gives us a total of four possible states:

$$\begin{aligned}|0\rangle \otimes |0\rangle &= |00\rangle\\ |0\rangle \otimes |1\rangle &= |01\rangle\\ |1\rangle \otimes |0\rangle &= |10\rangle\\ |1\rangle \otimes |1\rangle &= |11\rangle.\end{aligned}$$

Each of these four states is a superposition of the two individual qubit states. Together they form a basis in $\mathbb{C}^4$. The tensor product allows us to represent the combined state of the system as a single mathematical object, which makes it easier to perform computations on the system. Given 2 matrices $A, B \in \mathbb{C}^2$ where,

$$A = \begin{pmatrix}a & b\\ c & d\end{pmatrix} B = \begin{pmatrix}e & f\\ g & h\end{pmatrix}, \quad (8)$$

YES



INPUT	OUTPUT
A	
0	0
1	1

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

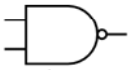
OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

XOR

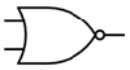
INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

NAND



INPUT		OUTPUT
A	B	
0	0	1
1	0	1
0	1	1
1	1	0

NOR



INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT		OUTPUT
A	B	
0	0	1
1	0	0
0	1	0
1	1	1

Figure 1: Examples of classical gates.

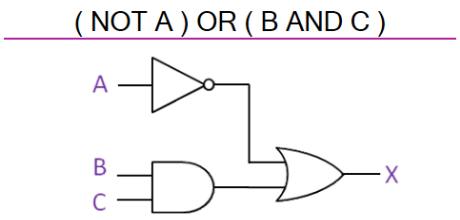


Figure 2: Example of classical circuit.

the tensor product is given by:

$$\begin{aligned} A \otimes B &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} e & f \\ g & h \end{pmatrix} \\ &= \begin{pmatrix} a \begin{pmatrix} e & f \\ g & h \end{pmatrix} & b \begin{pmatrix} e & f \\ g & h \end{pmatrix} \\ c \begin{pmatrix} e & f \\ g & h \end{pmatrix} & d \begin{pmatrix} e & f \\ g & h \end{pmatrix} \end{pmatrix} \\ &= \begin{pmatrix} a \cdot e & a \cdot f & b \cdot e & b \cdot f \\ a \cdot g & a \cdot h & b \cdot g & b \cdot h \\ c \cdot e & c \cdot f & d \cdot e & d \cdot f \\ c \cdot g & c \cdot h & d \cdot g & d \cdot h \end{pmatrix}. \end{aligned} \tag{9}$$

Thie product $A \otimes B$ is a 4d complex square matrix.

Let's consider 2 qubits in a quantum circuit, with both qubits having an inital state $|0\rangle$. The initial state of the 2 qubit system can be represented as a tensor product between two vectors. The state is denoted by:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{10}$$

If we apply the Hadamard gate H from (7) to the second qubit whilst keeping the second qubit the same, we can model the operation on the whole system by:

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & 1 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ 1 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & -1 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \tag{11}$$

where the matrix $I \in \mathbb{C}^2$ is the identity matrix. The state in (10) now becomes:

$$(H \otimes I)\,|00\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{12}$$

.

The resultant state in equation (12) has 50% chance of collapsing either into state $|00\rangle$ or $|10\rangle$. The quantum circuit for this diagram is shown in Figure 3.

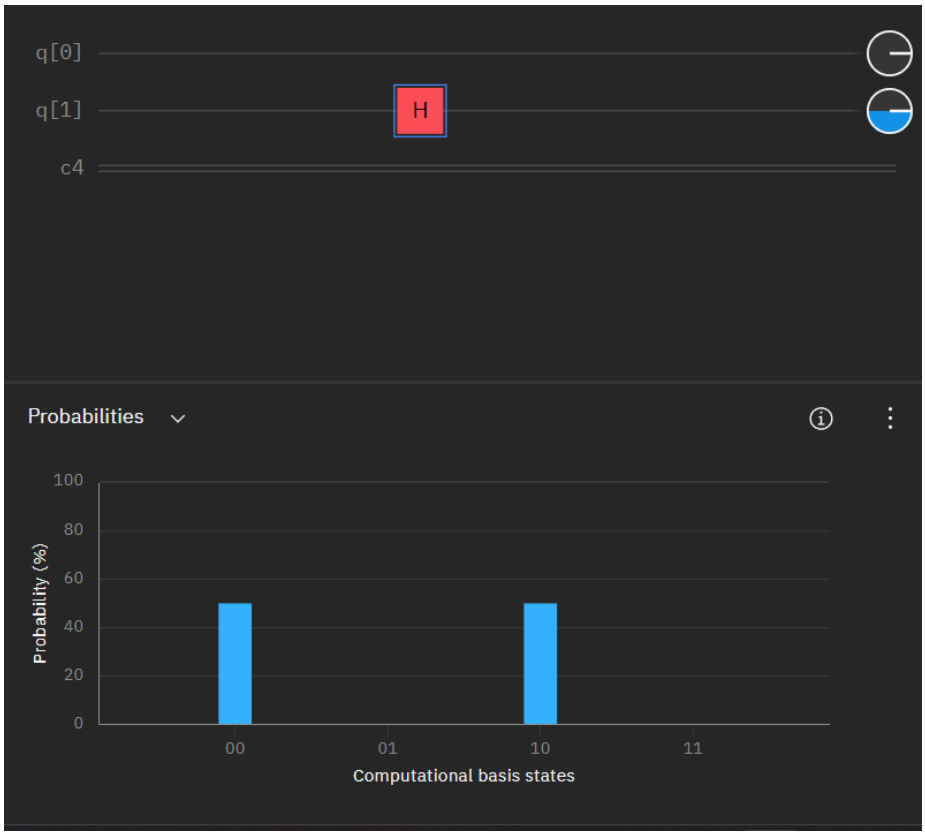


Figure 3: Circuit diagram for operation in (12). The probabilities of each resulting basis state are plotted on the bottom chart. Diagram was obtained from [1]

2.4 Controlled gates

A controlled gate is a quantum gate that acts on two or more qubits, where the operation of the gate depends on the state of a control qubit. The controlled-NOT (CNOT) gate is a simple example of a controlled gate. The CNOT gate uses two qubits, one for control and one for the target. The target qubit remains unchanged if the control qubit is in the state $|0\rangle$. If the control qubit is in the state $|1\rangle$, then the target qubit is flipped, resulting in the target state $|0\rangle$ becoming $|1\rangle$ and vice versa. The CNOT gate's operation can be represented

by the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{13}$$

More generally for any unitary operator $U \in \mathbb{C}^2$ the matrix is given by

$$\begin{aligned} Controlled - U = & I \otimes ... \otimes |0\rangle \langle 0| \otimes ... \otimes I \otimes ... \otimes I + \\ & I \otimes ... \otimes |1\rangle \langle 1| \otimes ... \otimes U \otimes ... \otimes I, \end{aligned} \tag{14}$$

where $|0\rangle \langle 0|$ and $|1\rangle \langle 1|$ act on the controlled qubit whilst the U acts on the target qubit. The matrices $|0\rangle \langle 0|$ and $|1\rangle \langle 1|$ are given by:

$$\begin{aligned} |0\rangle \langle 0| &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \\ |1\rangle \langle 1| &= \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}. \end{aligned} \tag{15}$$

3 Code design and implementation

The project was implemented in the C++ language using Visual Studio Code. The code can be found at [2]. The project consists of 3 different base classes `Matrix`, `QuantumComponent` and `QuantumCircuit`. The file structure of the code is shown in Figure 4. There is a header file and a cpp file for each class as well as `main.cpp` containing the code to be executed.

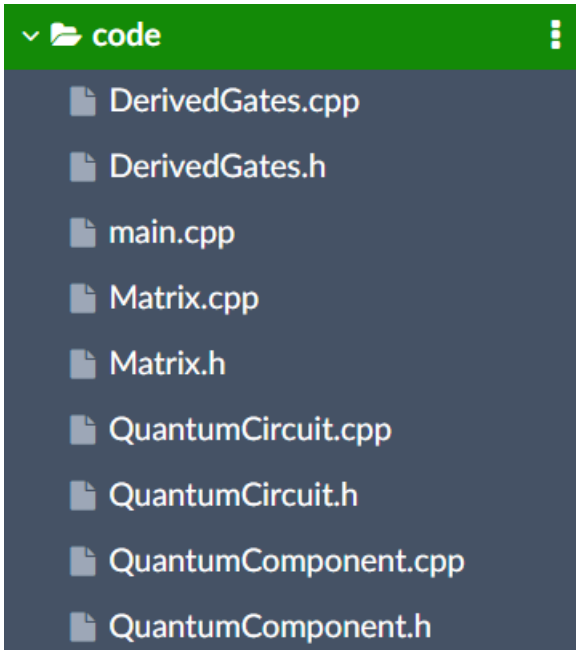


Figure 4: File structure for the project.

3.1 Matrix class

The Matrix class provides a way to represent matrices which contain complex doubles. The header file `Matrix.h` contains the class's function declarations and is shown in Listing 1. The class constructor takes as input a 2d vector array of complex doubles. There are member functions to overload the basic arithmetic operators to suit matrix operations. The `tensor_product()` member function is crucial when it comes to calculating operators on multi-qubit systems.

```
class Matrix
{
    // friend functions
    friend std::ostream& operator<<(std::ostream& os, const Matrix& matrix);
    friend std::istream& operator>>(std::istream& is, Matrix& matrix);
private:
    size_t rows;
```

```
    size_t cols;
    std::vector<std::vector<std::complex<double>>> data;

public:
    // Constructors and destructors
    Matrix();
    Matrix(size_t rows, size_t cols);
    Matrix(std::vector<std::vector<std::complex<double>>> data);
    ~Matrix() {}

    // Accessors
    size_t get_rows() const { return rows; }
    size_t get_cols() const { return cols; }
    const std::complex<double> operator()(size_t, size_t) const;

    // Mutators
    Matrix& operator=(const Matrix&);
    Matrix operator+(const Matrix&);
    Matrix operator-(const Matrix&);
    Matrix operator*(const Matrix&);
    bool operator==(const Matrix);
    std::complex<double>& operator()(size_t, size_t);
    Matrix tensor_product(const Matrix&);
    Matrix transpose();
    Matrix conjugate();
    Matrix adjoint();
};

// Non-member functions
Matrix identity_matrix(size_t n);
Matrix perform_tensor_product(const std::vector<Matrix> matrices);
#endif
```

Listing 1: Matrix.h

3.2 QuantumComponent class and its derived classes

The `QuantumComponent` class is the base abstract class from which all quantum gates are derived. Both `SingleGate` and `MultiGate` inherit from the `QuantumComponent` class. The inheritance tree of all the classes that are derived from `QuantumComponent` is shown in Figure 5.

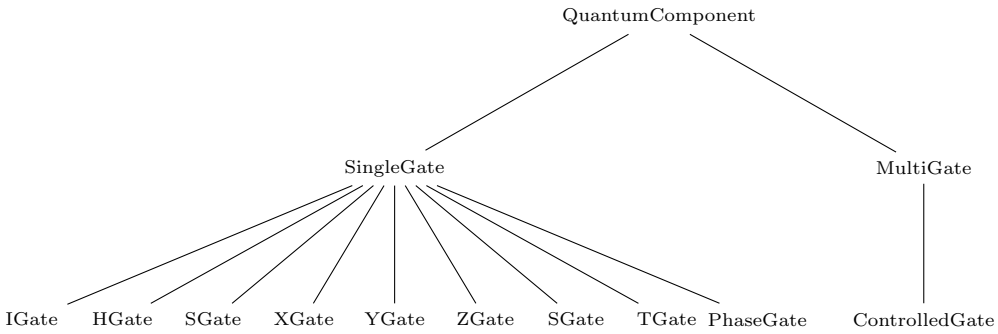


Figure 5: Inheritance of classes.

Listing 2 shows the members of the `QuantumComponent` class. The class

contains 3 private data members: `symbol`, `qubit_index` and `matrix`. The `qubit_index` variable refers to the register of the circuit the component will be in. The `matrix` variable stores the 2d complex square matrix representation of the component. Each component has a string symbol associated with it which is printed to the console when a circuit is drawn. The class has member getter functions to access the private variables.

```
class QuantumComponent
{
protected:
    std::string symbol="?";
    size_t qubit_index=0;
    Matrix matrix;

    // Constructors
    QuantumComponent();
    QuantumComponent(size_t qubit_index, std::string symbol, Matrix matrix);
public:
    // Destructor
    virtual ~QuantumComponent() {}

    // Accessors
    std::string get_symbol() const;
    size_t get_index() const;
    virtual Matrix get_matrix() const=0;
    virtual Matrix get_matrix(size_t register_size) const=0;
    virtual std::string get_gate_type() const=0;
    virtual bool can_gate_fit(size_t register_size) const=0;
    virtual std::string get_terminal_output(size_t terminal_line,
        size_t register_index) const=0;
    virtual std::string get_line(std::string type) const;
    int get_line_length() const;
};
```

Listing 2: QuantumComponent class function declarations.

3.2.1 SingleGate

The constructor for `SingleGate` takes in `qubit_index`, `matrix` and `symbol`. The default constructor returns an identity gate which is in the first register and has the symbol "I". The overloaded function for `get_matrix()` of `SingleGate` is shown in Listing 3.

```
Matrix SingleGate::get_matrix(size_t register_size) const
{
    if (!can_gate_fit(register_size))
    {
        throw std::invalid_argument("Gate␣cannot␣fit␣in␣register␣for␣SingleGate::
            get_matrix()");
    }
    std::vector<Matrix> matrices(register_size, identity_matrix(2));
    matrices[get_index()]=matrix;
    return perform_tensor_product(matrices);
}
```

Listing 3: Overloading of `get_matrix()` to return a matrix that has the same dimensions as a matrix representation of a single step in a circuit.

3.2.2 MultiGate

The `MultiGate` constructor takes in `qubit_index`, symbol, matrix and gate_size. The matrix of the multigate must be a sqaure matrix with dimensions equal to $2^{\texttt{gate\_size}}$). For example, if we had a multi-gate which takes up 3 registers then the matrix should be a $2^3 = 4$ dimensional square matrix. Listing 4 contains the constructor for `MultiGate`.

```
MultiGate::MultiGate() : MultiGate(0, "I", identity_matrix(2), 1) {};
MultiGate::MultiGate(size_t n, std::string symbol_in, Matrix matrix_in, size_t
    gate_size_in) : QuantumComponent(n, symbol_in, matrix_in)
{
    if (1<<gate_size_in!=matrix_in.get_rows()||1<<gate_size_in!=matrix_in.get_cols())
    {
        throw std::invalid_argument("Matrix size does not match gate size for MultiGate
            constructor");
    }
    gate_size=gate_size_in;
};
```

Listing 4: MultiGate constructor.

3.2.3 ControlledGate

The controlled gate inherits the `MultiGate` class as it takes up at least 2 registers; one register is needed for the target gate and with other used as the control register. The constructor for `ControlledGate` takes in a shared pointer to a `SingleGate` and the control index. The code for the constructor is shown in Listing 5.

```
ControlledGate::ControlledGate(std::shared_ptr<SingleGate> gate, size_t control_index_in
    )
{
    if (control_index_in==gate->get_index())
    {
        throw std::invalid_argument("Controlled gate cannot be at the same index as the
            index it is controlled by.");
    }
    control_index=control_index_in;
    target_index=gate->get_index();
    // Set the qubit index to index that "appears first".
    qubit_index=std::min(control_index, target_index);
    symbol=gate->get_symbol();
    gate_size=abs(control_index-target_index)+1;
    matrix=get_controlled_matrix(gate->get_matrix());
}
```

Listing 5: ControlledGate constructor.

To calculate the matrix for controlled gates we use the equation for controlled matrices outlined in (14). Listing 6 contains the code for calculating matrices of controlled gates. Two vector arrays containing matrices are created, with each array corresponding to the two terms in (14). In one array we set the element at the control index to $|0\rangle\langle 0|$. In the other, we set the element at the control index to $|1\rangle\langle 1|$ and the element at the target index to the target gate's matrix, A tensor product is performed between all the matrices in both the vector arrays

and then the two resulting matrices are added together to form the controlled gates matrix representation.

```
Matrix ControlledGate::get_controlled_matrix(Matrix gate_matrix) const
{
    // Controlled gates have matrices of the form:
    // I x ... x|0><0| x ... x I x ... x I +
    // I x ... x|1><1| x ... x U x ... x I
    // where U is the gate that is being controlled
    Matrix zero(2, 2); // |0><0|
    zero(0, 0)=1;
    Matrix one(2, 2); // |1><1|
    one(1, 1)=1;
    Matrix I=identity_matrix(2);
    std::vector<Matrix> matrices_0(gate_size, identity_matrix(2));
    std::vector<Matrix> matrices_1(gate_size, identity_matrix(2));
    int relative_control=control_index-qubit_index;
    int relative_target=target_index-qubit_index;
    matrices_0[relative_control]=zero;
    matrices_1[relative_control]=one;
    matrices_1[relative_target]=gate_matrix;
    Matrix controlled_matrix=perform_tensor_product(matrices_0)+perform_tensor_product(
        matrices_1);
    return controlled_matrix;
}
```

Listing 6: ControlledGate::get_controlled_matrix()

3.3 QuantumCircuit class

The `QuantumCircuit` class is used to create a circuit of quantum components. It has 4 data members. The `input_register` variable is a vector of integers containing either 1s or 0s depending on whether the input qubit at each qubit is $|0\rangle$ or $|1\rangle$. The size of this register is stored in `register_size`. The `components` variable is a 2d vector array containing `QuantumComponents` that comprise the quantum circuit. The number of total steps to cycle through the circuit is stored in `total_steps`.

```
class QuantumCircuit
{
private:
    std::vector<std::vector<std::shared_ptr<QuantumComponent>>> components;
    size_t register_size;
    size_t total_steps=-1;
    std::vector<int> input_register;

public:
    // Constructor and destructor
    QuantumCircuit(size_t register_size);
    ~QuantumCircuit();

    // Accessors
    Matrix get_initial_state() const;
    Matrix get_final_state() const;
    Matrix get_state_after_step(size_t step_index) const;
    size_t get_register_size() const;
    size_t get_total_steps() const;
    Matrix get_matrix_at_step(size_t step_index) const;
    Matrix get_matrix() const;
    bool step_contains_multigate(size_t step_index) const;
```

```
    std::shared_ptr<QuantumComponent> get_multigate_at_step(size_t step_index)
        const;
    bool is_step_empty(size_t step_index) const;
    bool is_gate_in_circuit(std::shared_ptr<QuantumComponent>) const;

    // Functions to draw output to console
    void draw_circuit() const;
    void draw_probability_distribution() const;

    // Mutators
    void set_input_register(std::vector<int> input_register);
    void add_component(std::shared_ptr<QuantumComponent> gate);
    void replace_component(std::shared_ptr<QuantumComponent> gate,
        size_t register_index,
        size_t step_index);
    void evolve();
    void evolve(size_t step_number);
    void ask_for_input();
    void test_circuit();

};

// Non Member functions
std::string get_binary_representation(int number, int register_size);
Matrix calculate_matrix_for_register(std::vector<int> register_values);
bool is_power_of_two(int number);
void draw_state(Matrix state);
#endif
```

Listing 7: QuantumCircuit class.

Components are added to the circuit using the public member function `add_component()`. The function takes in a shared pointer to a `QuantumComponent`. Shared pointers are used as gates can have multiple owners; one possible owner when the gate is initialised and another being the circuit. Checks are implemented in the function to ensure that the gate is not already in the circuit and that the gate can fit within the register of the circuit. Multiple single gates can fit into a single step of a circuit. However, multi-gates should take up a whole step and should be handled differently. The algorithm for adding components is shown in Listing 8.

```
void QuantumCircuit::add_component(std::shared_ptr<QuantumComponent> gate)
{
    // Check input
    if (is_gate_in_circuit(gate))
    {
        throw std::invalid_argument("Gate␣already␣in␣circuit!");
        return;
    }
    if (!gate->can_gate_fit(register_size))
    {
        throw std::invalid_argument("Gate␣is␣not␣within␣circuit's␣register␣size!");
        return;
    }
    int target_index=gate->get_index();
    std::string gate_type=gate->get_gate_type();
    // Add component by replacing the Identity gate at the current step.
    if (gate_type=="SingleGate")
    {
        if (components[target_index][total_steps]->get_symbol()!="I")
        {
            evolve();
```

```
        }
        replace_component(gate, target_index, total_steps);
    }
    else if (gate_type=="MultiGate")
    {
        if (!is_step_empty(total_steps))
        {
            evolve();
        }
        replace_component(gate, target_index, total_steps);
        evolve();
    }
    else
    {
        throw std::invalid_argument("Gate is not a valid gate type!");
    }
}
```

Listing 8: QuantumCircuit::add_component()

The initial quantum register of the circuit can be set using `set_input_register()`. The matrix representation of the circuit is found by first calculating the matrix representation of a combination of gates at each step using `get_matrix_at_step()`. Then the matrices at each step are multiplied by each other to get the full matrix describing the circuit as a whole. This is returned using `get_matrix()`. Both these functions are shown in Listing 9.

```
Matrix QuantumCircuit::get_matrix_at_step(size_t step_index) const
{
    // Create an identity matrix that has the same size as n gates tensor producted
    //  together.
    Matrix resultant_matrix=identity_matrix(1<<register_size);
    for (size_t i=0; i<register_size; i++)
    {
        // Multiply by the gates matrix (which is changed to account for the circuits
            size).
        resultant_matrix=components[i][step_index]->get_matrix(register_size)*
            resultant_matrix;
    }
    return resultant_matrix;
}

Matrix QuantumCircuit::get_matrix() const
{
    Matrix circuit_matrix=get_matrix_at_step(0);
    for (size_t i=1; i<total_steps+1; i++)
    {
        circuit_matrix=get_matrix_at_step(i)*circuit_matrix;
    }
    return circuit_matrix;
}
```

Listing 9: Matrix calculation functions of the circuit.

A visual representation of the circuit can be drawn to the console using `draw_circuit()`. This function uses ASCII characters to draw each single-gate or multi-gate register by register. The `draw_probability_distribution()` function draws a histogram of probabilities of each basis present in the final state of the circuit. The function is shown in Listing 10.

```
void QuantumCircuit::draw_probability_distribution() const
{
    // Calculate initial state in vector form and draw as ket.
    std::cout<<"Intial state:"<<std::endl;
    draw_state(get_initial_state());
    // Draw basis states in ket form and calculate probabilities of each basis
    //  state.
    std::cout<<std::endl
        <<"Probabilities of final states:"<<std::endl;
    Matrix final_state=get_final_state();
    for (int i=0; i<1<<register_size; i++)
    {
        Matrix basis_state=Matrix(1<<register_size, 1);
        basis_state(i, 0)=1;
        draw_state(basis_state);
        // Draw probability distribution as a histogram.
        double probability=(final_state(i, 0)*std::conj(final_state(i, 0))).real();
        std::cout<<"||"<<std::to_string(probability).substr(0, 5)<<" ||";
        int filled_blocks=probability*50;
        for (int j=0; j<filled_blocks; j++)
        {
            std::cout<<"#";
        }
        std::cout<<std::endl;
    }
}
```

Listing 10: QuantumCircuit::draw_probability_distribution()

To function `ask_for_input()` asks the user for an initial state from the terminal and returns the corresponding final state. The function `test_circuit()` prints all the possible input-output combinations to the console.

3.4 DerivedGates file

The `DerivedGates.h` file contains various functions which make creating gates easier and less tedious. The functions were created as an alternative to typing `std::make_shared<GateType>(n)` each time. The functions have concise names which makes creating new gates less tedious.

```
std::shared_ptr<SingleGate> h(size_t qubit_index);
std::shared_ptr<SingleGate> x(size_t qubit_index);
std::shared_ptr<SingleGate> y(size_t qubit_index);
std::shared_ptr<SingleGate> z(size_t qubit_index);
std::shared_ptr<SingleGate> s(size_t qubit_index);
std::shared_ptr<SingleGate> t(size_t qubit_index);
std::shared_ptr<SingleGate> p(size_t qubit_index, double phase);
std::shared_ptr<SingleGate> adjoint(std::shared_ptr<SingleGate> gate);
std::shared_ptr<MultiGate> gate_from_circuit(QuantumCircuit circuit,
    size_t qubit_index,
    std::string symbol);
std::shared_ptr<MultiGate> controlled(std::shared_ptr<SingleGate> target_gate,
    size_t control_index);
std::shared_ptr<MultiGate> swap(size_t index_1, size_t index_2);
std::shared_ptr<MultiGate> toffoli(size_t target_index, size_t control_1,
    size_t control_2);
```

Listing 11: Function declarations in DerivedGates.h

There is also an added functionality to create gates from circuits. This is done using `gate_from_circuit()`. Listing 12 shows the implementation of this. The `gate_from_circuit()`. function allows for more complicated multi-gates to be created by the user. Examples of gates include the swap gate whose implementation can be found in Listing 13.

```
std::shared_ptr<MultiGate> gate_from_circuit(QuantumCircuit qc, size_t n, std::string
    symbol)
{
    std::shared_ptr<MultiGate> gate=std::make_shared<MultiGate>(n, symbol, qc.get_matrix
        (), qc.get_register_size());
    return gate;
}
```

Listing 12: Implementation of gate_from_circuit().

```
std::shared_ptr<MultiGate> swap(size_t index_1, size_t index_2)
{
    size_t first_index=std::min(index_1, index_2);
    size_t last_index=std::max(index_1, index_2);
    size_t gate_size=last_index-first_index+1;
    QuantumCircuit swap_circuit(gate_size);
    swap_circuit.add_component(controlled(x(0), gate_size-1));
    swap_circuit.add_component(controlled(x(gate_size-1), 0));
    swap_circuit.add_component(controlled(x(0), gate_size-1));
    std::shared_ptr<MultiGate> gate=gate_from_circuit(swap_circuit, first_index,
        std::to_string(first_index)+"␣<->␣"+std::to_string(last_index));
    return gate;
}
```

Listing 13: Implementation of swap().

4 Results

To utilize this project, in a new file (e.g. `main.cpp`) include the headers `QuantumCircuit.h` and `DerivedGates.h`. To create a circuit, initialise QuantumCircuit passing the register size as an argument. Then initialise a shared pointer to a gate using the gates constructor or by using one of the functions in `DerivedGates.h`. To add the gate to the circuit, call `add_component()`, using the shared pointer as an argument. Listing 14 shows an example circuit. The output from this circuit is shown in Listing 15. Applying `draw_circuit()` draws the circuit to the console and `draw_probability_distribution()` draws the probability distribution of basis states which comprise the final to the console.

```cpp
#include <iostream> // to output things to terminal
#include <memory> // for use of shared pointers
#include "QuantumCircuit.h"
#include "DerivedGates.h"

int main() {
    QuantumCircuit circuit(3);
    // initialising using constructor
    std::shared_ptr<HGate>h0 = std::make_shared<HGate>(0);
    // initialising using DerivedGates.h function
    std::shared_ptr<QuantumComponent>h1(h(1));
    circuit.add_component(h0);
    circuit.add_component(h1);
    circuit.add_component(h(2)); // easier
    circuit.draw_circuit();
    circuit.draw_probability_distribution();
    circuit.test_circuit();
    return 0;
}
```

Listing 14: Example code to create a quantum circuit

```
QuantumCircuit : 0x3e367ff620
        +---+
q_0 : ==| H |==
        +---+
        +---+
q_1 : ==| H |==
        +---+
        +---+
q_2 : ==| H |==
        +---+
Intial state:
|000>
Probabilities of final states:
|000> ||0.125 ||######
|001> ||0.125 ||######
|010> ||0.125 ||######
|011> ||0.125 ||######
|100> ||0.125 ||######
|101> ||0.125 ||######
|110> ||0.125 ||######
|111> ||0.125 ||######
```

Listing 15: Output for the example circuit from Listing 14.

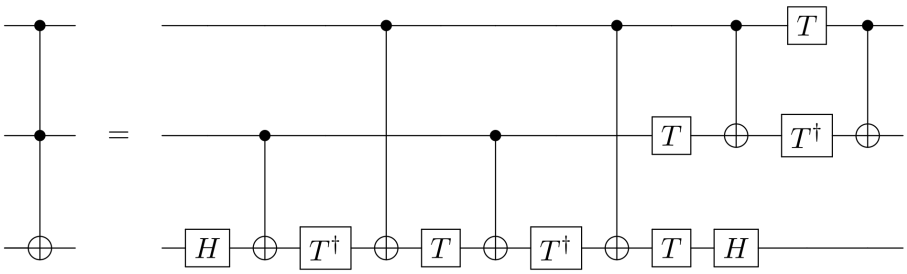


Figure 6: Diagram of the Toffoli gate and its construction from other quantum gates. Picture taken from [3].

4.1 The Toffoli gate

The Toffoli gate is a three-qubit gate that performs a not operation on a target qubit register if and only if the 2 control qubits are in the state $|1\rangle$. To build the Toffoli gate we can add a combination of Hadamard gates, CNOT gates, and phase gates to a 3-qubit quantum circuit. A diagram of the Toffoli gate and its construction is shown in Figure 6. The code for the construction of the Toffoli gate is shown in Listing 16 and the corresponding output is in Listing 17.

```
#include <iostream> // to output things to terminal
#include <memory> // for use of shared pointers
#include "QuantumCircuit.h"
#include "DerivedGates.h"


int main() {
    int control_1=0;
    int control_2=1;
    int target=2;
    int first_qubit=std::min(std::min(control_1, control_2), target);
    int last_qubit=std::max(std::max(control_1, control_2), target);
    int r_t=target-first_qubit;
    int r_c1=control_1-first_qubit;
    int r_c2=control_2-first_qubit;

    QuantumCircuit qc(last_qubit-first_qubit+1);
    qc.add_component(h(r_t));
    qc.add_component(controlled(x(r_t), r_c2));
    qc.add_component(adjoint(t(r_t)));
    qc.add_component(controlled(x(r_t), r_c1));
    qc.add_component(t(r_t));
    qc.add_component(controlled(x(r_t), r_c2));
    qc.add_component(adjoint(t(r_t)));
    qc.add_component(controlled(x(r_t), r_c1));
    qc.add_component(t(r_c2));
    qc.add_component(t(r_t));
    qc.add_component(h(r_t));
    qc.add_component(controlled(x(r_c2), r_c1));
    qc.add_component(t(r_c1));
    qc.add_component(adjoint(t(r_c2)));
    qc.add_component(controlled(x(r_c2), r_c1));

    qc.draw_circuit();
    qc.test_circuit();
    return 0;
```

```
}
```

Listing 16: Example code to create a quantum circuit

```
QuantumCircuit : 0x73129ff520
                                                                                  +---+
q_0 : ==========================0===========================0====================0====| T |=====0===
                                |                           |                    |    +---+     |
                                |                           |    +---+         ++|++  +----+  ++|++
q_1 : ============0==============|=============0=============|====| T |==========| X |==| T* |==| X |=
                  |              |             |             |    +---+          +---+  +----+  +---+
          +---+  ++|++  +----+  ++|++  +---+  ++|++  +----+  ++|++  +---+  +---+
q_2 : ==| H |==| X |==| T* |==| X |==| T |==| X |==| T* |==| X |==| T |==| H |======================
          +---+  +---+  +----+  +---+  +---+  +---+  +----+  +---+  +---+  +---+
input states -> output states
|000> ->|000>
|001> ->|001>
|010> ->|010>
|011> ->|111>
|100> ->|100>
|101> ->|101>
|110> ->|110>
|111> ->|011>
```

Listing 17: Output from Listing 16

We can check that the code runs as expected by comparing the output from Listing 17 to the circuit diagram in Figure 6. Also by using the `test_circuit()` function, we can evaluate every possible input-output combination.

4.2 Full adder circuit

The possible binary additions between three bits is:

$$\begin{aligned} 0+0+0&=0 \\ 0+0+1&=1 \\ 1+1+0&=0 \text{ carry } 1 \\ 1+1+1&=1 \text{ carry } 1. \end{aligned} \tag{16}$$

The full adder circuit is a 4-qubit circuit which calculates the binary addition between 2 input qubits and 1 carry qubit. The last qubit must be prepared as $|0\rangle$. Let the 2 input qubits be denoted by $|A\rangle$ and $|B\rangle$. The carry qubit is denoted by $|C_{in}\rangle$. The full adder circuit calculates the sum between these 3 qubits and outputs a resultant sum qubit denoted by $|S\rangle$ as well as an output carry qubit denoted by $|C_{out}\rangle$. A diagram of the circuit is shown in Figure 7.

The full adder circuit is constructed from controlled X and normal X gates. Listing 18 shows the implementation of this construction. We can verify that the code functions correctly by comparing the circuit diagram and input-output combinations in Listing 19 to the circuit in Figure 7 and the truth table determined by equation (16).

```
#include <iostream> // to output things to terminal
#include <memory> // for use of shared pointers
#include "QuantumCircuit.h"
#include "DerivedGates.h"

int main()
```

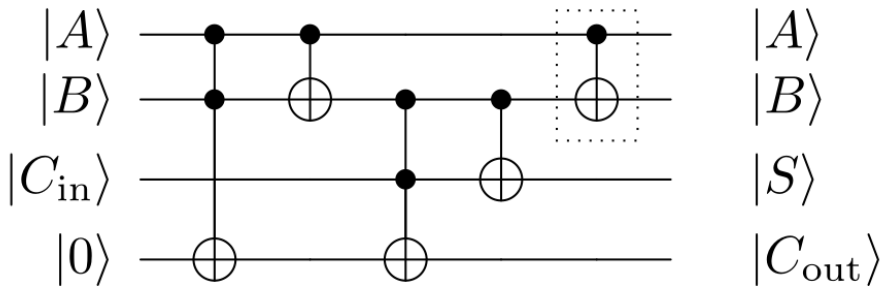


Figure 7: Full adder quantum circuit. Image is taken from

```
{
    QuantumCircuit full_adder(4);
    full_adder.add_component(toffoli(3, 0, 1));
    full_adder.add_component(controlled(x(1), 0));
    full_adder.add_component(toffoli(3, 1, 2));
    full_adder.add_component(controlled(x(2), 1));
    full_adder.add_component(controlled(x(1), 0));
    full_adder.draw_circuit();
    full_adder.test_circuit();
    return 0;
}
```

Listing 18: Example code to create a full adder circuit

```
QuantumCircuit : 0x7c6e3ff9d0
        +-----------+
q_0 : ==|           |====0============================0==========
        |           |    |                            |
        |           |  ++|++  +-----------+         ++|++
q_1 : ==|           |==| X |==|           |====0====| X |=========
        |           |  +---+  |           |    |    +---+
        |           |         |           |  ++|++
q_2 : ==| q0q1(+)q3 |=========| q1q2(+)q3 |==| X |================
        |           |         |           |  +---+
        |           |         |           |
q_3 : ==|           |=========|           |======================
        +-----------+         +-----------+
input states -> output states
|0000> ->|0000>
|0001> ->|0101>
|0010> ->|0110>
|0011> ->|1011>
|0100> ->|0100>
|0101> ->|1001>
|0110> ->|1010>
|0111> ->|1111>
|1000> ->|1000>
|1001> ->|1101>
|1010> ->|1110>
|1011> ->|0011>
|1100> ->|1100>
|1101> ->|0001>
|1110> ->|0010>
|1111> ->|0111>
```

Listing 19: Output from Listing 18

5 Discussion and conclusion

In summary, our implementation of simulating quantum circuits using C++ has proven successful, with our simulator functioning as intended when being applied to various j different circuits. However, there are several areas in which improvements could be made. Examples of improvements include, using namespaces to prevent naming collisions, properly naming constant arguments in functions, and commenting functions in the Doxygen format. These improvements would help contribute to the code's readability and maintainability.

Additionally, further functionality could be added to our simulator. These include the ability to add measurement gates at specific steps, the inclusion of a classical register, and improving the visual representation of the circuit. Also, to further optimize the `QuantumCircuit` class, an alternative approach would be to replace the current 3D component array (which includes identity gates to fill empty spaces between components) with a 1D array that stores depth information in the quantum component itself. This modification would significantly reduce the memory required to store each quantum circuit.

References

[1] IBM Quantum. `https://quantum-computing.ibm.com/`, 2021.

[2] Aebel Shajan. Quantum circuits source code `https://github.com/Aebel-Shajan/Quantum-Circuits`, 2023.

[3] Wikipedia. Toffoli gate `https://en.wikipedia.org/wiki/Toffoli_gate`, 2023.