C++ Quantum Circuit Simulation

Zhiyu Liu, *Department of Physics and Astronomy, University of Manchester*

The goal of this project is to simulate the effect of a quantum circuit on quantum states known as qubits. The project provides an interface that allows users to visualise a quantum circuit with an arbitrary number of qubits and quantum gates. The project can calculate the evolved final state given an initial quantum state. A vector represents the quantum state, and a matrix represents the quantum gate, both of which are imitated by an array. Several advanced C++ features are used in the project, including STL containers, polymorphism, move semantics and smart pointers.

1 Introduction

1.1 *Qubits and Quantum Gates*

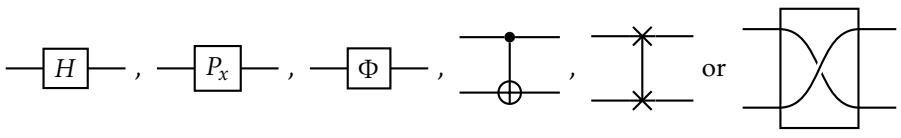
The fundamental unit of information in quantum computing is a qubit, represented by a vector $|a\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. This vector is a superposition of the two basis vectors $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$, contrasting the binary digit 0 or 1 of classical computers. Two-qubit basis vectors are constructed from the tensor product, e.g., $|00\rangle = |0\rangle \otimes |0\rangle = [1\ 0\ 0\ 0]^{\mathrm{T}}$. A general two-qubit state is a normalized vector in the space spanned by $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. A classical n-bit string 01...00 is represented as the vector $|01...00\rangle = |0\rangle \otimes |1\rangle \otimes \cdots \otimes |0\rangle \otimes |0\rangle$, which is a basis vector in the 2^n dimensional space. Quantum gates, which act on n qubits, are unitary operators represented by $2^n \times 2^n$ matrices. The Identity gate leaves the quantum state unchanged. Pauli gates, inspired by electron spins, induce rotations of the qubit around an axis. The Hadamard gate is commonly used to switch from the standard basis to the Bell basis. Phase shift gates perform rotations around the z-axis. The Controlled NOT (CNOT) gate, acting on two qubits, executes an XOR operation $|a, b\rangle \to |a, a \oplus b\rangle$, where a is the control qubit and b the target qubit. The Swap gate switches the positions of the two qubits on which it acts, e.g., $|01\rangle \to |10\rangle$. Matrix representations of these gates and the examples of the effects of the SWAP and CNOT01 (0 being the control qubit and 1 the target qubit) acting on two qubits are provided below:

$$P_x = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix},\ P_y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix},\ P_z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix},\ H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix},\ \Phi = \begin{bmatrix}1 & 0\\ 0 & e^{i\varphi}\end{bmatrix},\ \mathrm{CNOT} = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix},\ \mathrm{SWAP} = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix},$$

$$\mathrm{SWAP}|01\rangle = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}\cdot\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = |10\rangle, \quad \mathrm{CNOT}|11\rangle = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}\cdot\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = |10\rangle$$

1.2 *Quantum Circuit*

A quantum circuit is depicted by several parallel lines, also known as wires, each representing a qubit. Gates, represented by their corresponding symbols, act on these qubits and are drawn on the lines from left to right, following the order of operation. The symbols for the Hadamard, Pauli, Phase, CNOT, and SWAP gates are displayed below:



In quantum computing, sequential gates on the same qubit result in matrix multiplication, while parallel gates on different qubits yield tensor products.

$$0 \text{ —}\boxed{H}\text{—}\boxed{H}\text{—}, \quad H \cdot H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \cdot \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = I \quad \text{(Unitary and Hermitian)}$$

$$\begin{array}{l}0 \text{ —}\boxed{H}\text{—}\\ 1 \text{ —}\boxed{H}\text{—}\end{array}, \quad H \otimes H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix}.$$

2 Code implementation

2.1 *Code Structure*

The simulation is achieved by three primary libraries `Statevector`, `QuantumGate`, `QuantumCircuit` and five sub-libraries, `CNOT`, `Hadamard`, `Pauli`, `Phase`, and `Swap`. The project also makes use of two auxiliary libraries: `Console` and `Format`, which provides an interface and perform formatting respectively.

2.2 *The `Statevector` library*

The `Statevector` class encapsulates a quantum state vector. It comprises two private attributes: `size_t qubit_n`, representing the number of qubits, and `std::unique_ptr<std::complex<double>[]> array` for storing the elements of the state vector. The array size is 2^n, where n is the number of qubits. The use of a smart pointer ensures automatic deletion of the array when it goes out of scope, preventing memory leaks and eliminating the need for manual memory management in move semantics. An object of the `Statevector` class can be initialized by providing a bit string. For example, `Statevector ket01{0, 1}` initializes the array with the elements `0,1,0,0`. The object can also be constructed by explicitly specifying each element, allowing for vectors of arbitrary length. The three classes, `Statevector`, `QuantumGate`, and `QuantumCircuit`, all include necessary copy/move constructors and overloaded copy/move assignment operators. Arithmetic operators (`+`, `-`, `/`) have been overloaded in line with their mathematical definitions. Elements of the vector can be accessed using the overloaded `[]` operator, and the vector can be visualized either as a column or a row vector. The library also includes functions to generate a list of standard basis vectors given the number of qubits. This list is stored in the STL container `std::map<std::string, Statevector>`, where the string serves as the ket representation. The following example demonstrates a function that prints the information of the basis vectors stored in the `map`.

```
>>Statevector state{0, 0}; // state={1, 0, 0, 0}
>>display_std_basis(4);
|0000> : [ 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ] // Can be initiliased as Statevector state{0,0,0,0}
|0001> : [ 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
....
|1111> : [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 ]
```

Apart from standard basis, the library provides a function that is able to return the Bell basis or the GHZ/W entangled states.

2.3 *The `QuantumGate` library*

The `QuantumGate` class encapsulates quantum gates in a matrix representation with four attributes: `size_t rows, size_t cols, Type type` and a unique pointer to a complex double array. The type of gate is defined by the scoped `enum class QuantumGate::Type`. The gate matrix, though inherently 2D, is represented as a 1D array for

memory efficiency. Elements are accessed via `gate(row, col)= array[(row-1)*cols+(col-1)]` by overloading the `()` operator, mimicking standard 2D access. The `QuantumGate` library, beyond basic matrix operations, supports Kronecker and dyadic products. It also includes utilities for matrix display and pre-defined quantum gates for quick usage. The constructor for the `QuantumGate` object makes use of `initializer_list` and smart pointer and is shown below.

```
QuantumGate::QuantumGate(
Type type_, const size_t size_, std::initializer_list<std::complex<double>> elements) :
rows(size_), cols(size_), type(type_)
{
   array = std::make_unique<std::complex<double>[]>(size_ * size_);
   for (size_t i = 0; i < size_ * size_; i++)
      // initializer_list elements cannot be accessed directly
      array[i] = *(elements.begin() + i);
   this->round();
}
// Using the constructor to define a static class member
QuantumGate QuantumGate::PauliX{Type::PauliX, 2,{0, 1, 1, 0}};
```

2.4 *Derived classes of `QuantumGate`*

In this section, we will detail the construction algorithms for specific quantum gates. These gates include the Hadamard, Pauli, and Phase gates, which are all constructed using predefined matrices in their parent class. If these gates are to be applied to a quantum circuit with more than one qubit, they are expanded by performing a Kronecker product with the identity matrix. For instance, if a Hadamard gate is acting on the first qubit of a three-qubit circuit, the matrix is constructed as $H\otimes I\otimes I$. This is achieved by invoking the command `Hadamard H{3, 0}`. If two Hadamard gates are operating on the first and second qubits, then the gate is constructed as $H\otimes H\otimes I$. This can be achieved by invoking the command `Hadamard H{3,{0,1}}`. The CNOT and Swap gates are constructed using dyadic operations. As an example, when the Swap gate operates on the first and second qubits, the matrix is constructed in the following way:

$$\mathrm{SWAP}^{\text{total qubits}}_{\text{swapped qubits}} = \mathrm{SWAP}^{2}_{01} = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 01| + |11\rangle\langle 11| \tag{1}$$

$$= \begin{bmatrix}1\\0\\0\\0\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} + \begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix} + \begin{bmatrix}0\\0\\1\\0\end{bmatrix}\begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix} + \begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix} \tag{2}$$

$$= \begin{bmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix} + \begin{bmatrix}0&0&0&0\\0&0&1&0\\0&0&0&0\\0&0&0&0\end{bmatrix} + \begin{bmatrix}0&0&0&0\\0&0&0&0\\0&1&0&0\\0&0&0&0\end{bmatrix} + \begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{bmatrix} = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}. \tag{3}$$

If the Swap gate is acting on the first and second qubits in a 3-qubit circuit, then

$$\mathrm{SWAP}^{3}_{01} = |000\rangle\langle 000| + |001\rangle\langle 001| + |010\rangle\langle 100| + |011\rangle\langle 101| + |100\rangle\langle 010| + |101\rangle\langle 011| + |110\rangle\langle 110| + |111\rangle\langle 111|.$$

The CNOT gate is constructed in a similar way. For example the first qubit is the controlled one, then

$$\mathrm{CNOT}_{01}^{2} = |00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 11| + |11\rangle\langle 10| \tag{4}$$

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

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

The snippet for constructing the swap gate is shown below.

```
auto basis = generate_std_basis(qubit_n_); // Get a vector of standard basis mapped by ket rep.
QuantumGate res = Zeros(qubit_n_);
for (auto it = basis.begin(); it != basis.end(); it++)
{
   // comparing the first and second qubit in the ket representation
   if (it->first.at(swap_q1) == it->first.at(swap_q2))
      res = res + dyad(basis[it->first], basis[it->first]);
   else
   {
      std::string temp = it->first;
      temp.at(swap_q1) = it->first.at(swap_q2);
      temp.at(swap_q2) = it->first.at(swap_q1);
      res = res + dyad(basis[it->first], basis[temp]);
   }
}
```

2.5 *The `QuantumCircuit` library*

The `QuantumCircuit` library provides a `QuantumCircuit` class and several functions to display the circuit. The basic component of the object is `gates_targets` which is a vector of `GateWithTarget` pair that is used to store the gates and the qubits that they applied to. The pair is defined as below.

```
using GatesWithTarget = std::pair<std::vector<size_t>, QuantumGate>;
// {{0, 1}, Swap} stands for a Swap gate acting on the first two qubits.
// {{0, 1, 2}, H} stands for 3 Hadamard gates acting on the first three qubits parallelly.
```

The circuit object can only be initialised with the number of qubits and the gates are added later on the corresponding qubits. The user can then initialise a `Statevector` object and get the final state vector after evolved through the circuit. An example is shown below.

```
QuantumCircuit qc{3}; // Initialise a QuantumCircuit object with 3 qubits.
qc.add_Hadamard({0, 1, 2}); // Add three Hadamard gates to the first three qubits.
qc.add_CNOT(1, 2); // Add a CNOT gate, qubit 1,2 is the controlled and target qubit respectively.
qc.display_circuit(); // Display the circuit in ASCII texts.
Statevector initial_state = generate_state(3, "GHZ", ""); // Initial state is the GHZ state.
Statevector final_state = evolve(initial_state, qc);
```

The display method of the class is the most elegant one. The basic component of the ASCII-style circuit is a struct object `circuitLine` composed of three strings. The middle string consists of the wire and gate symbols. The upper and bottom strings are composed of spaces and extra components of the gate symbols. The symbols are pre-defined `const std::string`s in the header file. An example of the print of the circuit is shown below. The process for drawing the circuit is shown in the flow chart.

```
struct circuitLine
{ std::string upper {""}; std::string middle {""}; std::string bottom {""}; };
```

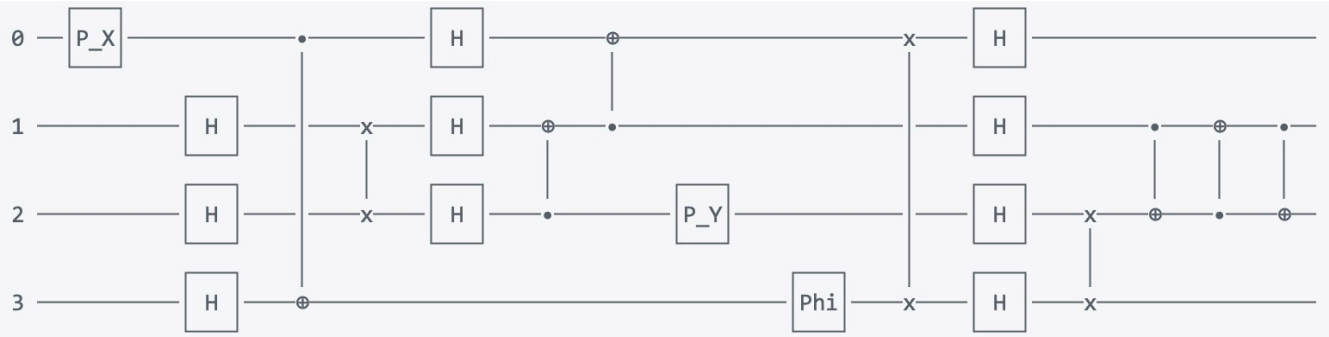


Figure 1: Visualisation of the circuit with arbitrary length and qubits.

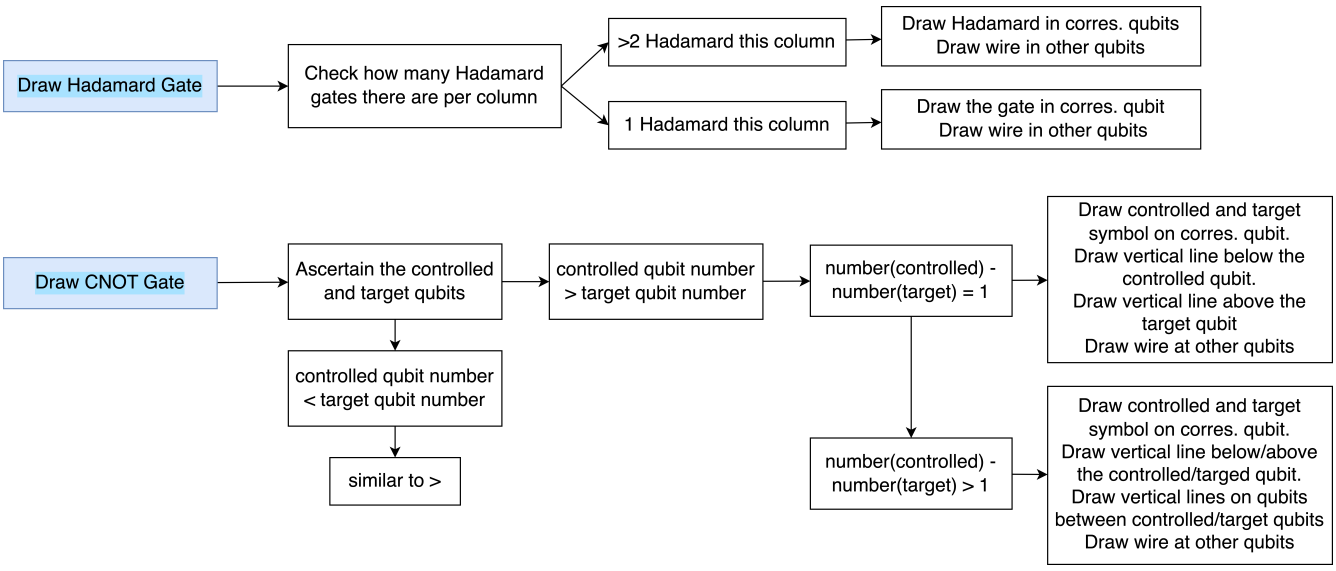


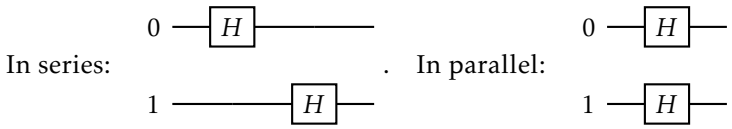
Figure 2: Process of drawing the circuit

2.6 *The `Console` library*

The `Console` library employs polymorphism to establish a flexible user interface with various parallel menus and sub-menus. The base class, `QuantumCircuitConsole` has four derived classes serve as parallel consoles: `QuantumCircuitMainConsole`, `CircuitOperationConsole`, `VectorOperationConsole`, and `DisplayInfoConsole`. Using the STL container `std::map`, each console is associated with a unique key. The program displays the appropriate menu based on the `current_console` key and processes user input, thus enabling navigation across different menus. Sub-menus are handled by the static class member `menu_level`.

```
std::map<std::string, std::unique_ptr<QuantumCircuitConsole>> consoles;
consoles["Main"] = std::make_unique<QauntumCircuitMainConsole>();
consoles["CircuitOperation"] = std::make_unique<CircuitOperationConsole>();
consoles["GateInfo"] = std::make_unique<DisplayInfoConsole>();
consoles["VectorOperation"] = std::make_unique<VectorOperationConsole>();
std::string current_console = "Main";
while (true) {
   consoles[current_console]->display_menu();
   // The most important line that handles the navigation of different consoles.
   current_console =
       consoles[current_console]->process_option(consoles[current_console]->read_option());
}
```

In the console interface, quantum gates can only be added in series, not in parallel. If a user wishes to add quantum gates in parallel, they would need to manually modify this in the `main.cpp` file. They can construct the `QuantumCircuit` and add Hadamard gates in parallel using the `add_Hadamard` method provided earlier. Indeed, the ordering of parallel and serial quantum gates doesn't alter their overall effect. For instance, the following two circuits produce the same result: applying a Hadamard gate to the first qubit and then to the second qubit, which can be expressed as $(H \otimes I) \cdot (I \otimes H) = H \otimes H$ mathematically.



3 Conclusion and outlook

The library effectively demonstrates basic operations of a quantum circuit, providing both computational simulation and visualisation. Currently, the program only supports parallel Hadamard gates. An area of potential enhancement is the inclusion of support for additional parallel gates, such as the Toffoli gate. The simulation of the circuit is currently limited to certain basis vectors for the initial state. This could be improved by supporting arbitrary vectors, with the implementation of a normalization method to ensure the state vector maintains its requisite property of being a unit vector.