Quantum Circuit Simulation

Object Oriented Programming in C++ Final Project

Ksenija Kovalenka, 10485506

May 20, 2023

Abstract

This report describes the implementation of the quantum circuit simulator using Object Oriented Programming (OOP) in C++. The created program allows one to construct circuits with up to 6 quantum bits (qubits) and apply the selection of single-qubit and two-qubit gates. An example circuit is included, which demonstrates all the basic features of the program. The available gates construct the complete basis and hence allow in principle any quantum computation within the qubit limit. Amongst other attributes, the source code is featuring a custom matrix class for gate representation, utilising smart pointers for clear ownership and safe memory management.

1 Introduction

In recent years, quantum computing has been gaining popularity among researchers and businesses as it claims to shift the paradigm of information processing. It has already been shown to be extremely powerful for certain types of problems and its rapid development continues with the invention of new types of algorithms and hardware [1]. Achieving the advantage of using quantum computation over traditional classical computers comes closer to reality with the Noisy Intermediate Scale Quantum (NISQ) era, prompting large interest not only from the academic but also the industry sector [2]. Although there are certain challenges tied to the fact that coherent quantum states are extremely fragile and struggle to hold memory, quantum computation provides numerous exciting opportunities in the study of many-body complex quantum systems and acceleration of minimisation algorithms. This report explores the mechanics of quantum computation by describing a created circuit simulation with a set of gate components operating on qubit states. An example circuit is discussed, as well as the ability to construct a fully custom quantum computation.

2 Theory

Quantum computers operate on quantum bits (qubits) instead of classical bits. The distinctive feature of qubits is their quantum mechanical behaviour allowing an exhibition of quantum

phenomena of superposition and entanglement. Utilising these features allows for computational speed-up due to the exponential increase in qubit state-space dimensionality with the number of qubits in the system. A single qubit state can be expressed as a superposition of two basis states $|0\rangle$ and $|1\rangle$. The basis is arbitrarily chosen to be the eigenstates of the Pauli Z matrix, which could represent a spin projection of the system onto the z-direction, as an example. Each state in the superposition has a complex amplitude (z_0 and z_1), so the total state $|\psi\rangle$ can be described as follows

$$|\psi\rangle = z_0 |0\rangle + z_1 |1\rangle = z_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + z_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \end{pmatrix}. \tag{1}$$

The result of the computation is given by the measurement of the qubit state. When the qubit is measured, its superposition state collapses into one of the basis states with the corresponding probabilities, given by the squares of their respective amplitudes ($|z|^2 = z \cdot z^*$, where z^* is the complex conjugate of z). The measurement and hence the result of the computation is independent of the overall phase of the qubit state $|\,|\psi\rangle\,|^2 = |e^{i\delta}\,|\psi\rangle\,|^2$. This introduces the constraint of the qubit state, which allows us to fully specify the state using 3 real parameters instead of 4 real (or 2 complex) ones. There is an additional constraint, provided by the conservation of total probability ($|z_0|^2 + |z_1|^2 = 1$). Thus, the most general qubits state can be expressed as

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle\,, \tag{2}$$

where θ and ϕ are new parameters [3]. This way of writing the qubit state allows a famous Bloch sphere qubit representation, shown in Figure 1.

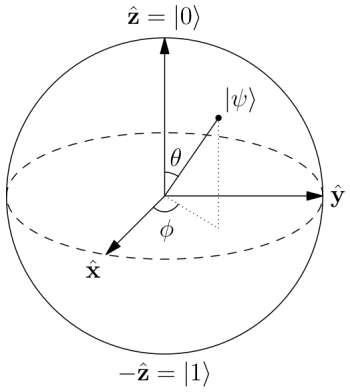

Figure 1: Bloch sphere representation of the normalised qubit state $|\psi\rangle$.

Qubits are manipulated with quantum gates, which can rotate the states on the Bloch sphere, introduce the relative phase shift between the basis states, swap two qubits, etc. The manipulations are mathematically represented by the 2x2 gate matrix multiplying the 2x1 state vector of a qubit. Manipulations on the several qubits are represented in terms of the tensor product[1]

[1]Tensor product definition: https://mathworld.wolfram.com/VectorSpaceTensorProduct.html

of qubit states and gate matrices [4]. For example, applying gate A to state $|\psi_1\rangle$ and B to state $|\psi_2\rangle$ is calculated as follows

$$A\,|\psi_1\rangle \otimes B\,|\psi_2\rangle = (A \otimes B)\,|\psi_1\rangle \otimes |\psi_2\rangle\,. \tag{3}$$

Entanglement is provided by the gates E represented by the 4x4 matricies and acting as $E(|\psi_1\rangle \otimes |\psi_2\rangle)$ on the two qubits at the same time.

3 Code Design and Implementation

3.1 Project Aim

This project aims to construct a simulation of a fully arbitrary quantum computation, exploiting OOP in C++. The created program makes use of the main OOP principles: encapsulation, inheritance, polymorphism and abstraction, which are further discussed below (Section 3.3). The available selection of gates represents the universal set i.e. any unitary evolution of the system can be represented with a finite series of gates in the set. Such a set consists of three rotations, a phase shift and a controlled not (CNOT) gate for basic entanglement. Additional widely used gates are included for further convenience. All gate matrices are specified in the `components.cpp` file.

3.2 Program Flow

Quantum computation is performed in several steps as follows:

1. Pick the number of qubits in the circuit.
2. Initialise all qubit states to $|0\rangle$.
3. Perform the unitary evolution by constructing and applying quantum gates.
4. Perform a measurement by calculating the probabilities of each state.

Gate construction happens in 2 steps. A circuit consists of gate layers. All gates within a single layer are applied in parallel, whereas all layers are applied in series to construct a total circuit. All gates in a layer have to be combined using a tensor product as discussed in Section 2. If there is no gate on a qubit in the layer, the identity matrix is inserted into the tensor product, so that all of the combined layer matrices have the same dimensions. When all layers are specified, combined layer matrices are applied to the initialised combined qubit state in succession using Equation 3. An example of layer structure and application is shown in Figure 3 for greater clarity.

3.3 Class Structure

The main classes used in this program are shown in Figure 2. The modularity of the code means that each class is implemented separately with the corresponding header `.h` and `.cpp` files, and are linked together in the `main.cpp` file.

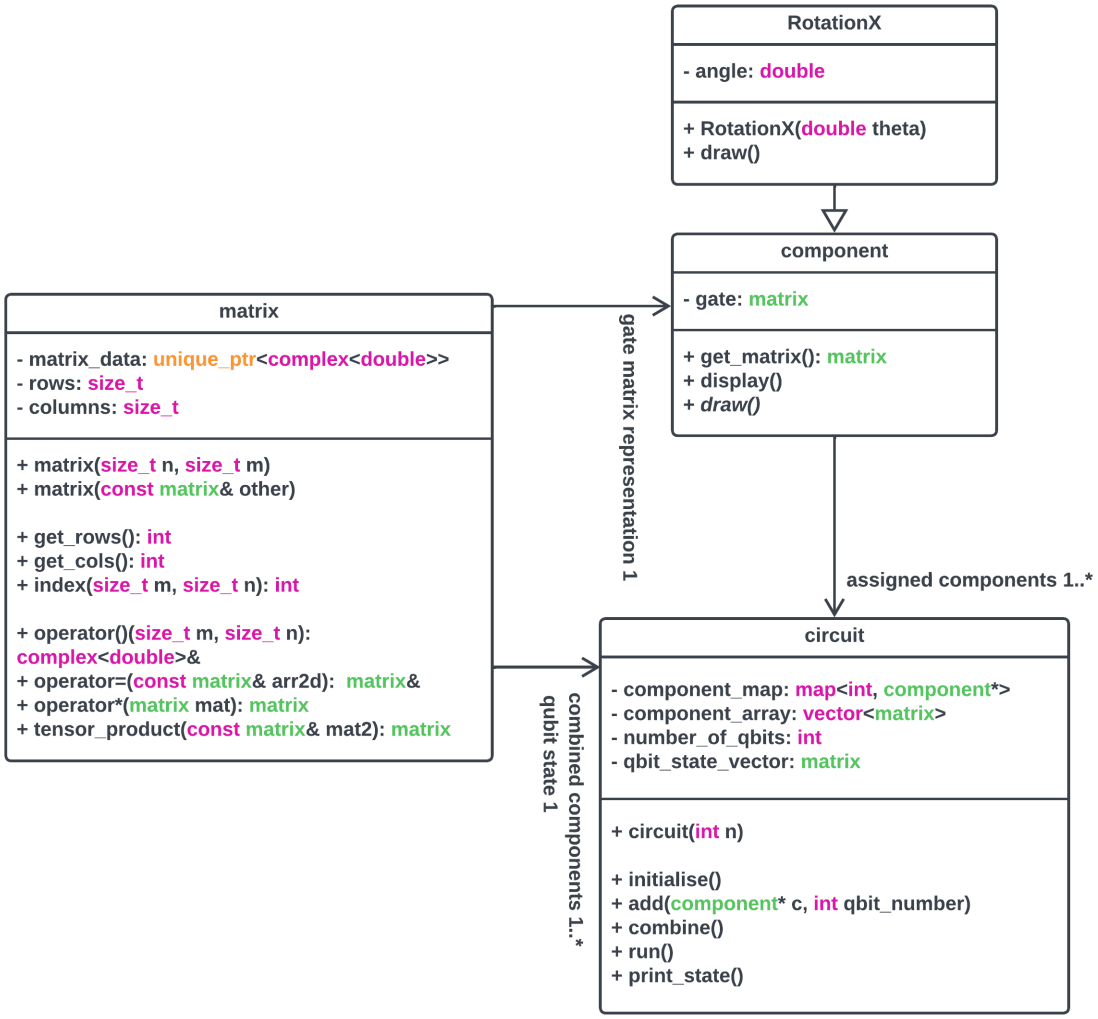

Figure 2: Unified Modeling Language diagram for class structure for the program, featuring main class dependencies (represented with line arrows) and an example of inheritance (represented with empty head arrows) on the component class.

All the data is private or protected (employing encapsulation) and can be accessed only by friend classes or through `get_` member functions. A matrix class lays in the basis of all computations. The component class uses it for gate matrix representation and circuit utilises it in layer and qubit state matrices. All gates in the circuit inherit from the component class to implement polymorphism in layer construction, which makes the program scalable. Note that `RotationX`

class in Figure 2 is only an example of many more gate classes in the program. Additionally, the program has a qubit class in the `component.h` file for additional bookkeeping. This class is not used in the main file of this program, but it can be built upon if one decides, for example, to visualise qubits as in Figure 1.

Matrix Class

Matrix class allows the user to construct arbitrary size matrices with complex entries. It makes use of smart pointers to manage matrix data memory and has a number of overloaded operators. It has custom assignment operators employing deep copying and useful matrix operations like multiplication and matrix product.

Component Class

Component class in an abstract class with a virtual `draw()` member function to display the component in the circuit diagram. This base class implements a matrix representation of the quantum gates, which inherit from it. Some of the derived classes have extra parameters, and they all overwrite a virtual `draw()` function with putting their name in the output. Two-qubit gates overwrite the 2x2 default component matrix with a 4x4.

Circuit Class

Circuit class makes the main computation. It has member functions to implement all the computation steps described in Section 3.2. Circuit initialisation takes in an integer number of qubits. The state of the qubits is then initialised with `.initialise()` to $|0\rangle \otimes ... \otimes |0\rangle = (10...0)^T$, meaning that all the qubits in the combined state are pointing up the z-axis. Components can be added to a layer of gates using `.add()` member function, which takes an argument of the pointer to a specified component and a target qubit. Two qubit gates are applied to the specified qubit and the next one in a row. For example, an entry `.add(&CNOT, 2)` adds a two-qubit gate CNOT, which is applied to qubits 2 and 3. When the layer is ready, it must be combined using `.combine()`, which performs a tensor product of the corresponding matrices and puts the result in the queue called `component_array`. Finally, `.run()` function is used to apply all the queued gate layers one by one and `.print_state()` outputs the resulting state vector (or indeed any other intermediate state of the calculation).

3.4 Program Functionality

The program has 11 available gates. The instances of the static gate components (i.e. ones which do not require a parameter) are created on stack, allowing them to get destroyed with the default destructors when the program goes out of the scope of the main function. The main function consists of an example circuit run (shown in Figure 3) and a user input circuit, each controlled

by the `bool example` and `bool circuit`, respectively. When both are set as `true`, two circuit objects are created.

Example

An example circuit shown in Figure 3 has all the basic features of the circuit construction. It has several layers, with gates applied to different qubits. However, qubit 2 still has two consecutive operations in layers 2 and 3. Layer 2 features a 2-qubit CNOT gate and X gate applied in parallel. Layer 3 shows the parameterised $R_y(\pi)$ gate action.

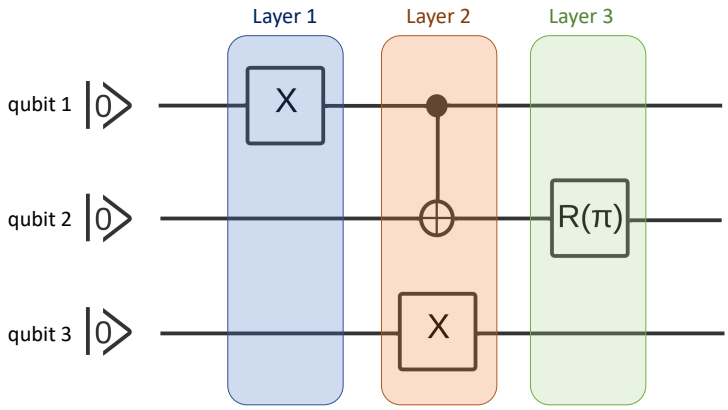

Figure 3: Example circuit, showcasing main circuit construction features.

User Interface

The user is first asked to give the number of qubits for the circuit. Then, the component library gets printed out and they are asked to complete the first layer by entering the gate key and which qubit they wish to apply the gate to. After this, the program finds the corresponding gate in the library in the component library map by the corresponding key and adds it to the layer map inside the circuit. When the layer is completed, the user is asked whether they would like to add another one, and the process repeats. A flow diagram for the circuit building by the user is shown in Figure 4 for further clarity. The procedure is slightly more complicated with the parameterised gates. If the user enters the key which corresponds to one of such gates, they are prompted to give a parameter. There is also an option to go with the default version of the gate. However, if the custom parameter is chosen, an instance of a `new` parameterised gate is created. The memory for it must be dynamically allocated on heap as the component objects must live outside the layer construction loop for the execution of the `.combine()` command. The custom component is given a new key to distinguish it from the default components. The memory is freed using the `delete` command after the `.combine()` is executed.

```cpp
// allocation
if (key == "rx") {
    component_library.insert(std::make_pair("rxc", new RotationX(gate_parameter)
    ));
    key = "rxc";
}
// other code...
// deallocation
for (const auto& key_value : component_library) {
    if (key_value.first == "rxc" || key_value.first == "ryc" || key_value.first
    == "rzc" || key_value.first == "pc") {
        delete key_value.second;
    }
}
```

Listing 1: An example of dynamic memory allocation on heap.

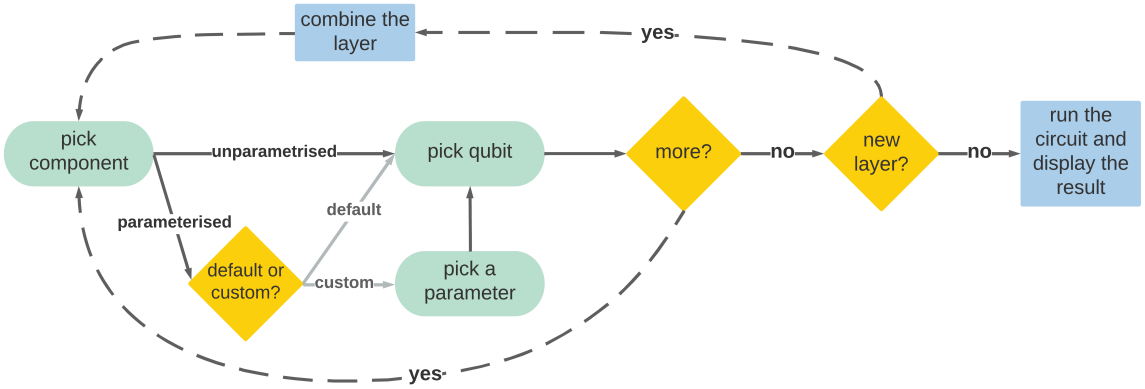

Figure 4: Flowchart of the user circuit construction process.

Validation

The program utilises various types of validation. Firstly, integer and double inputs in the particular range are validated by the extra template function, defined outside the main routine. The `template` is needed to account for both types of data. Secondly, the user is not allowed to overwrite the gate components, by putting a second gate on the qubit which already has a gate in the same layer. One qubit can have multiple gates applied to it, but it must be done in separate layers. To account for that, there is a `vector` which keeps track of the occupied qubits.

```cpp
template<typename T>
T get_value(T min, T max) {
    T value;
    // function implementation ...
```

Listing 2: Template function implementation in user input validation.

Two qubit gates require lots of additional validation. Firstly, the program must check that two qubits are still available when the user enters the key for the two-qubit gate. Secondly, applying a two-qubit gate on the last qubit does not make sense with the definition of their action, hence the available qubit range is reduced when the user inputs the target qubit. And lastly, an application of a two-qubit gate must mark two qubits as unavailable, instead of one.

3.5 Advanced Features

The program makes use of several advanced C++ features. Standard Template Library (STL) containers can be seen throughout the code. Vectors are used whenever dynamic arrays are required, such as the occupied qubit record. Maps are used twice: for labelling gates by the corresponding keys in the main routine and for keeping track of which gates are applied to which qubits in the circuit layer combination routine.

```
std::map<int, component*> component_map;
// other code ...
component_map.insert(std::make_pair(qbit_number, c));
// other code ...
for (size_t i{start}; i<=number_of_qbits; i++) {
    if (component_map.count(i)) {
        matrix single_component{component_map[i]->get_matrix()};
        combined_component= combined_component.tensor_product(single_component);
        // check if the component is applied to two gates
        if (single_component.get_cols()==4) {i++;}
    } else {
        // assume identity if nothing specified
        combined_component= combined_component.tensor_product(I.get_matrix());
    }
```

Listing 3: Part on the code, demonstrating the use of maps. In the for loop, one can loop over the qubits.

The standard library complex number template class is also used as a reliable and fast complex number implementation option. On the other hand, the matrix class was created from scratch, utilising unique smart pointers for safe memory management. A template function was created for the validation of two different data types of numbers, to avoid repetition. Lambda statement was used in validating the qubit availability, as the implementation can be done concisely inside the loop running through the occupied qubit vector.

```
bool unavailable = std::any_of(occupied_qubits.begin(), occupied_qubits.end(),
[which_qubit](int value) {
    return value == which_qubit;
});
```

Listing 4: Lambda function in the qubit availability validation loop.

4 Results

The example circuit initialises the combined 3 qubit state, which corresponds to a vector of length $2^3 = 8$. Initialised state looks like $|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$. Then, the X gate on the first qubit flips the state to $|100\rangle$ in the first layer. In the second layer, the CNOT flips the state of the second qubit as the first (control) qubit is in the state 1. In the same layer, another X gate flips the state of qubit 3, resulting the combined state of $|111\rangle$. Finally, in the third layer $R_y(\pi)$ gate flips back the state of qubit 2 and introduces a phase of $e^{i\pi} = -1$, so the final state is $|101\rangle$ with a corresponding phase. The same calculation can be done in terms of state vectors as follows,

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{4}$$

which is exactly what the code outputs as shown in Figure 5. The same circuit can be constructed manually with the user input and it produces the same output as shown in the same figure. Validation successfully rejects any invalid string values and values out of the range for integers and doubles. Layer and 2-qubit gate validation also works as expected. A Makefile was also made to automate the build process of a project on the Unix-like operating systems, which is shown in Figure 6.

5 Discussion and Conclusion

In summary, the circuit simulator program achieves the aim of constructing an arbitrary quantum circuit of 6 qubits and outputs the correct computation result for an example circuit, both predefined and user constructed. It satisfies the minimal class design and functionality requirements. The program has an abstract base class and derived classes for components with matrix representation. The user is able to pick any gate from the library and apply it to any qubit in series or parallel within the rules of circuit construction. Extra functionality includes the 2-qubit gates and validation for them, as well as dynamically created parametrised gates which provide a full computational basis and freedom. The constructed circuit is printed out at the end of the computation, which makes the program output much more clear. The code features both the main principles of OOP and advanced C++ features.

As the program is modular, new gates can be easily added to expand the available computations. Qubit class can be utilised for bookkeeping and Bloch sphere representation of the qubits. For

the latter, an external C++ graphical user interface library is suggested. Implementing a measurement by calculating state probabilities and drawing a random number for the distribution should be added to fully represent the probabilistic nature of quantum computation. The qubit limit can be easily lifted if the result is used elsewhere and not in the terminal output.

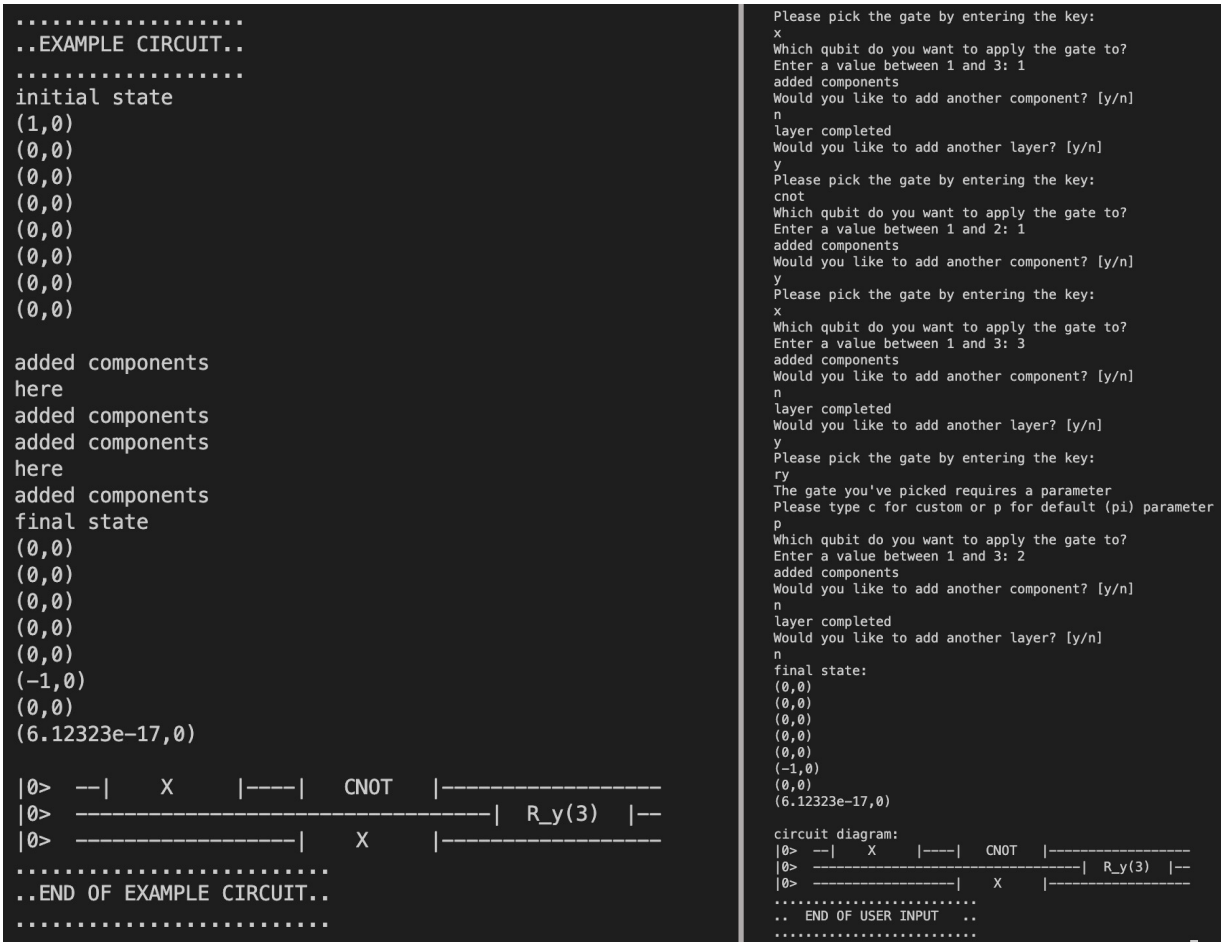

Figure 5: Command-line interface output of the program. The same circuit is constructed twice, as a preprogrammed example and a user input.

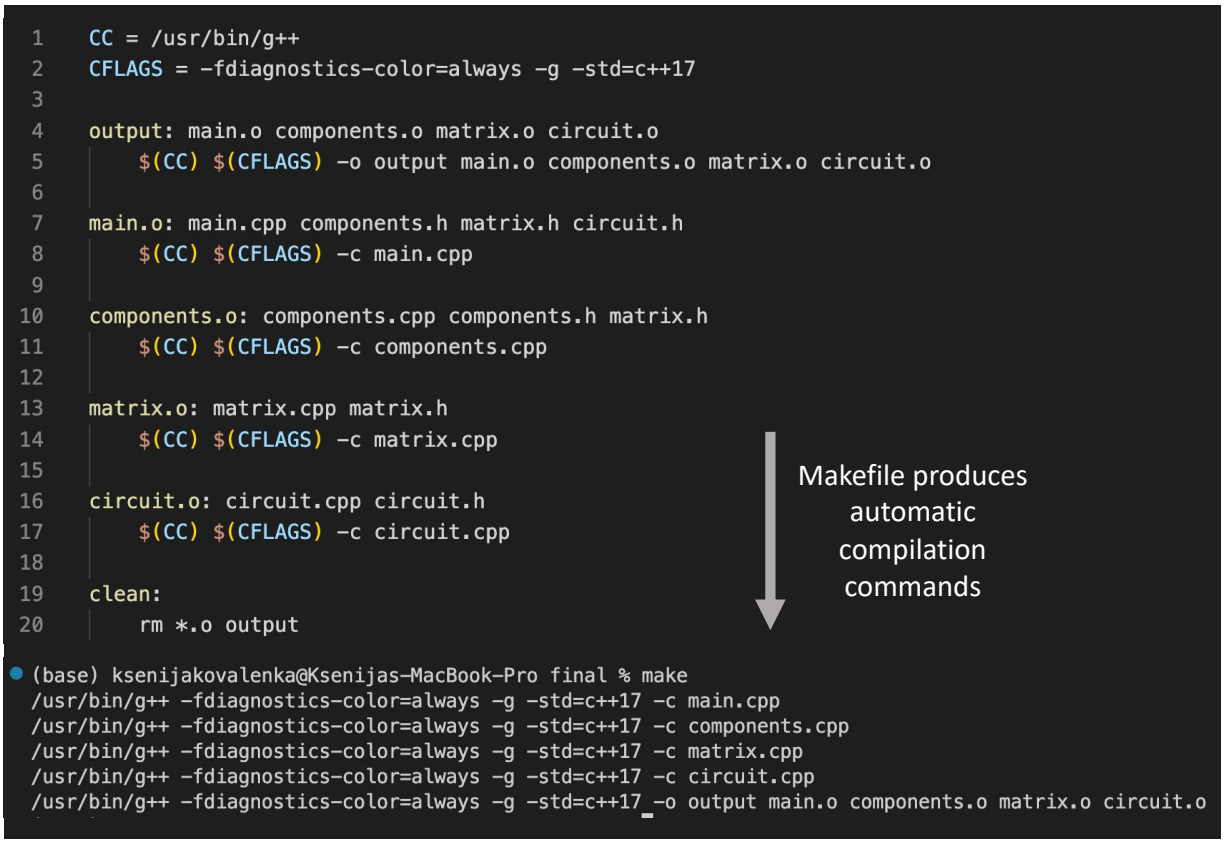

Figure 6: Makefile optimising and automating the compilation procedure.

References

[1] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien, "Quantum computers," *Nature*, vol. 464, no. 7285, pp. 45–53, 2010.

[2] J. W. Z. Lau, K. H. Lim, H. Shrotriya, and L. C. Kwek, "Nisq computing: where are we and where do we go?" *AAPPS Bulletin*, vol. 32, no. 1, p. 27, 2022. [Online]. Available: https://doi.org/10.1007/s43673-022-00058-z

[3] IBM Qiskit Development Team, "Summary of quantum operations," May 2023, accessed 15 April 2023. [Online]. Available: https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html

[4] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press, 2010.