QUANTUM CIRCUIT EDITOR FOR VARIOUS ARCHITECTURES OF IBMQ

MR. KITTIPAN CHUENBUNPERM 61070501005
MR. SUTHEP CHANCHUPHOL 61070501054
MR. RUANGVIT SRIUMPHAIVIWAT 61070501066

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF BACHELOR OF ENGINEERING (COMPUTER ENGINEERING)
FACULTY OF ENGINEERING
KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI
2021

Quantum Circuit Editor for Various Architectures of IBMQ

Mr. Kittipan Chuenbunperm 61070501005
Mr. Suthep Chanchuphol 61070501054
Mr. Ruangvit Sriumphaiviwat 61070501066

A Project Submitted in Partial Fulfillment
of the Requirements for
the Degree of Bachelor of Engineering (Computer Engineering)
Faculty of Engineering
King Mongkut's University of Technology Thonburi
2021

Project Committee

..... Project Advisor
(Asst.Prof. Rajchawit Sarochawikasit)

..... Project Co-Advisor
(Prapong Prechaprapranwong, Ph.D.)

..... Committee Member
(Unchalisa Taetragool, Ph.D.)

..... Committee Member
(Jaturon Harnsomburana, Ph.D.)

Project Title	Quantum Circuit Editor for Various Architectures of IBMQ
Credits	3
Member(s)	Mr. Kittipan Chuenbunperm 61070501005
	Mr. Suthep Chanchuphol 61070501054
	Mr. Ruangvit Sriumphaiviwat 61070501066
Project Advisor	Asst.Prof. Rajchawit Sarochawikasit
Co-advisor	Prapong Prechaprapranwong, Ph.D.
Program	Bachelor of Engineering
Field of Study	Computer Engineering
Department	Computer Engineering
Faculty	Engineering
Academic Year	2021

Abstract

Many companies and contributors are now driving the research and development of quantum computers. Among them is IBM. In this research, we aim to create a quantum circuit editing software suitable for the IBMQ quantum computer architectures. In addition, IBM Quantum Composer, the official graphical editor for creating quantum circuits based on the Qiskit library, is missing out a lot in functionality. With this in mind, our team wanted to create a quantum circuit editing software that could solve this problem as well.

Due to the present number of IBMQ machines and their tendency to increase, our quantum circuit editing software will include a recommendation system. It will automatically recommend an IBMQ system suitable for a particular circuit. The main part of this is the transpiler. A transpiler is a component of software that helps translate a quantum circuit before it is used on the IBMQ system. With a different transpiler setting, it will alter the outcomes after running on IBMQ. Since each system has a unique environment, benchmarking is essential to discover the optimized transpiler setting. As a result, the researcher devised an accumulative error measurement to assess the quality of the results, which roughly calculates the error after the circuit is translated. Which we can create a recommendation system based on this value. In addition, we look for similar products and conduct a user usability test to determine the weaknesses and shape our improvement around it.

We have divided the work into 6 phases: Research on related topics, Plan tasks and milestones, Research and user interface design, Implement, Test and improve, and Evaluate. As a result, we have created a quantum circuit editor that helps end users comprehend, analyze, optimize, and select the statistically optimum transpiler for their quantum circuit. Our recommendation system can assist our users in identifying the suited IBMQ architecture for their circuit and displaying the order of the instructions that will be executed on it. Moreover, the user usability test revealed that our editor is the most preferred option among our participants, and we have gotten a great number of favorable comments. And with the feedback from our participants, we are able to identify and shape the next chapter of our product.

Keywords: Quantum Computer Architecture / Quantum Circuit Transpilation / Quantum Computer / IBMQ / Qiskit

หัวข้อปริญญานิพนธ์	ซอฟต์แวร์แก้ไขวงจรควอนตัมสำหรับสถาปัตยกรรมคอมพิวเตอร์ควอนตัมของ IBMQ
หน่วยกิต	3
ผู้เขียน	นายกิตติพันธ์ ชื่นบุญเพิ่ม นายสุเทพ จันทร์ชูผล นายเรืองวิทย์ ศรีอำไพวิวัฒน์
อาจารย์ที่ปรึกษา	ผศ.ราชวิชช์ สโรชวิกสิต ดร.ประพงษ์ ปรีชาประพาฬวงศ์
หลักสูตร	วิศวกรรมศาสตรบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ภาควิชา	วิศวกรรมคอมพิวเตอร์
คณะ	วิศวกรรมศาสตร์
ปีการศึกษา	2564

บทคัดย่อ

ในปัจจุบันนี้ บริษัทและหน่วยงานจำนวนมากกำลังขับเคลื่อนการวิจัยและการพัฒนาคอมพิวเตอร์ควอนตัม ซึ่งหนึ่งในนั้นคือบริษัท IBM การวิจัยครั้งนี้จึงมีวัตถุประสงค์เพื่อสร้างซอฟต์แวร์แก้ไขวงจรควอนตัมที่เหมาะสมกับสถาปัตยกรรมคอมพิวเตอร์ควอนตัมของ IBMQ อีกทั้งซอฟต์แวร์แก้ไขวงจรควอนตัมของ IBM นั่นคือ IBM Quantum Composer นั้น มีการใช้งานที่ค่อนข้างยากและไม่สามารถดึงศักยภาพของ Qiskit ซึ่งเป็นซอฟต์แวร์ตัวกลางออกมาได้ ทางทีมพัฒนาจึงต้องการสร้างซอฟต์แวร์แก้ไขวงจรควอนตัมที่สามารถแก้ไขปัญหานี้ได้ด้วยเช่นกัน

ในปัจจุบันจำนวนเครื่อง IBMQ นั้นมีอยู่เป็นจำนวนมากและมีแนวโน้มว่าจะเพิ่มขึ้นอีกในอนาคต ทีมจึงต้องการพัฒนาซอฟต์แวร์ที่มี Recommendation System ที่ช่วยแนะนำสถาปัตยกรรมคอมพิวเตอร์ควอนตัมของ IBMQ ที่เหมาะสมกับวงจรที่เขียนมาได้ โดยส่วนประกอบที่สำคัญของระบบนี้คือ Transpiler ซึ่งเป็นซอฟต์แวร์หนึ่งในการแปลงวงจรให้เหมาะสมก่อนจะนำไปใช้บนระบบ IBMQ ซึ่งการตั้งค่าที่แตกต่างกันก็จะส่งผลถึงผลลัพธ์ที่ได้หลังจากการทำงานบน IBMQ ด้วย อีกทั้งแต่ละระบบต่างก็มีสภาพแวดล้อมที่แตกต่างกัน การทดสอบประสิทธิภาพในแต่ละการตั้งค่าจึงมีความจำเป็นต่อการพัฒนาระบบนี้โดยตรง โดยทีมพัฒนาได้สร้างหน่วยวัดที่ใช้วัดคุณภาพของผลลัพธ์นั้นชื่อว่า Accumulative Error (ข้อผิดพลาดโดยรวม) ที่จะคำนวณร้อยละของความผิดพลาดโดยคร่าว หลังการแปลงวงจรออกมา แล้วจึงใช้ค่านี้ในการประกอบ Recommendation System นอกจากนี้ เรายังได้ค้นหาซอฟต์แวร์ที่คล้ายคลึงกันและนำมาทำ User Usability Test เพื่อดูว่าเราจะสามารถปรับปรุงข้อเสียหรือนำข้อดีในด้านต่าง ๆ จากซอฟต์แวร์นั้น ๆ มาปรับใช้ได้หรือไม่

เราได้แบ่งการทำงานออกเป็น 6 ช่วง คือ การค้นคว้าหาข้อมูลที่เกี่ยวข้องกับงานวิจัย การวางแผนเป้าหมายและการจัดการ ค้นคว้าเพิ่มเติมและออกแบบหน้าซอฟต์แวร์ สร้างและพัฒนาซอฟต์แวร์ ทดสอบและปรับปรุงซอฟต์แวร์ และนำเสนอซอฟต์แวร์ โดยผลลัพธ์ที่ได้คือซอฟต์แวร์แก้ไขวงจรควอนตัมที่ช่วยให้ผู้ใช้สามารถเข้าใจ วิเคราะห์ พัฒนาประสิทธิภาพ และเลือก Transpiler ที่เหมาะสมที่สุดในเชิงสถิติสำหรับวงจรควอนตัมนั้น ๆ ได้ โดย Recommendation System ของเราสามารถช่วยเหลือผู้ใช้ในการระบุสถาปัตยกรรม IBMQ ที่เหมาะสมสำหรับวงจรของผู้ใช้และแสดงลำดับของคำสั่งที่จะดำเนินการบนระบบนั้นได้ นอกจากนี้ ผลลัพธ์จาก User Usability Test ของผู้ใช้งานแสดงให้เห็นว่าซอฟต์แวร์ของเราเป็นซอฟต์แวร์ที่ผู้เข้าร่วมชื่อชอบมากที่สุด และเรายังได้รับความคิดเห็นที่เป็นประโยชน์มากมาย เนื่องด้วยคำติชมจากผู้เข้าร่วมการทดสอบของเรา ทำให้เราสามารถระบุและกำหนดแผนการพัฒนาต่อไปของผลิตภัณฑ์ของเราได้

คำสำคัญ: สถาปัตยกรรมคอมพิวเตอร์ควอนตัม / การแปลงวงจรควอนตัม / คอมพิวเตอร์ควอนตัม / IBMQ / Qiskit

ACKNOWLEDGMENTS

In this project, we have put out the effort. It would not, however, have been feasible without the kind support and assistance of several individuals and organizations. We'd want to express our heartfelt gratitude to each and every one of them.

To begin, we would like to thank Asst.Prof. Rajchawit Sarochawikasitfor providing us with the wonderful chance to work on this fantastic project titled "Quantum Circuit Editor for Various Architectures of IBMQ". We have been educated, counseled, and helped a great deal by your opportunity to accomplish this project.

Second, we'd want to thank our parents and friends for their assistance in completing this project in such a short amount of time.

CONTENTS

PAGE

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1 INTRODUCTION

1.1 Background

IBM, among other companies, is one of the major developers of quantum computing technology. Despite that, when a developer needs to execute a circuit, there are several architectures of quantum computers to pick from, which may make the developer unsure which architecture to choose. In reality, variations in architecture can have an impact on the output of the same circuit, like a butterfly effect.

When the user selects the architecture on which to run their circuit, the transpiler takes over, compiling and optimizing the original circuit so that it can be run on the target machine. Because of the randomness in the optimizing algorithm, the transpiled result can be drastically different or even worse than the original unoptimized circuit. When the transpiled circuit is performed on the target system, the system's errors can add up and alter the circuit's outcome.

With this knowledge, we can implement a recommendation system that provides helpful insight for a user by combining the data from the transpiler and the architecture.

1.2 Problem Statement, Motivation, and Potential Benefits

As IBM has made a significant impact on quantum computing technology. The company has also developed Qiskit, a sophisticated Python library for designing quantum circuits, which is one of many contributions to the quantum computing field. Despite this, IBM Quantum Composer, the official graphical editor for creating quantum circuits based on the Qiskit library, is missing out a lot in functionality.

We have looked into one feature that could be highly beneficial. The IBMQ system has a variety of architectures that users can pick to perform the circuit. We can take advantage and create a recommendation system. Taking advantage of transpiling is one potentially advantageous capability. We can transpile and show users the transpiled circuit, allowing them to analyze and optimize the program on the fly.

Despite the original editor having a lot of unique features, the command and menu layouts are confusing. The poor user experience stands out when compared to other quantum circuit simulators available online. Without visible user feedback in response to an action, this major problem results in confusing user interaction. This issue should be simply resolved by reordering and carefully planning command placements.

1.3 Proposed Methodology

- Approach
  - Research on related topics about other quantum composers and what technology to use.
  - Conclude and categorize tasks and milestones.
  - Design user interface and interactions between the application and the user.
  - Implement the application.
  - Test and readjust the application.
  - Evaluate the application.
- Objectives

  To develop an online quantum circuit editor that:
  - Allows the end-users to comprehend, analyze, optimize, and choose the best architecture for their quantum circuit.

  - Programs and displays the order of the instructions that will be performed on various IBMQ architectures.
  - Utilizes the advantage of the Qiskit library.

- Scope
  - Fundamental, but fully customizable quantum gates.
  - Group and reuse the set of gates anywhere.
  - Showing a transpile result with adjustable variables.
  - Automatically map your circuit with various IBMQ architectures.
  - Concise explanation with a good user experience.
  - Compile to OpenQASM and Qiskit.

1.4 Original Engineering Content

Major features that distinguish it from IBM Quantum Composer and other online quantum circuit simulators are:

- IBMQ architecture recommendation system: We calculate and give suggestions to users based on provided metrics and data from architecture calibration.
- Circuit Transpilation Result: We show a transpiled circuit that will be executed on real hardware, allowing users to evaluate and optimize it in real-time.

1.5 Task Breakdown, Draft Schedule

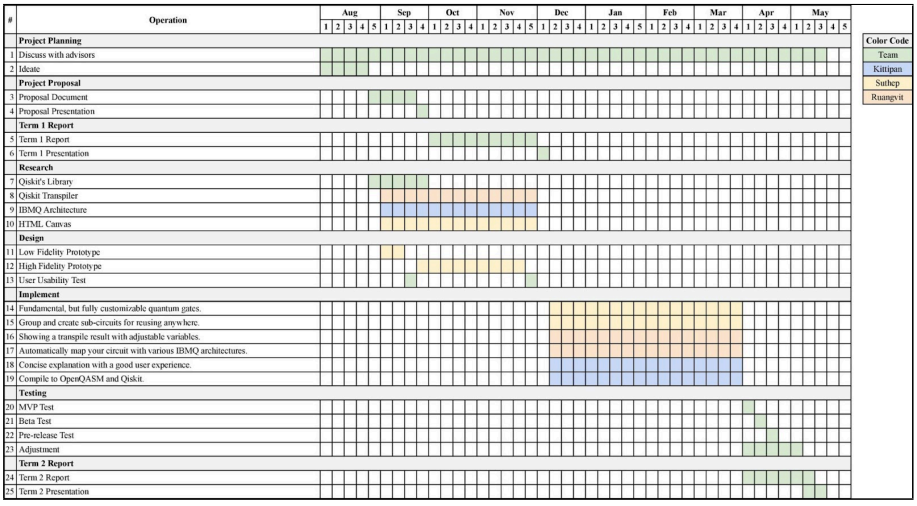

Figure 1.1 Task Breakdown Table Spanning from Term 1 to Term 2

1.6 Deliverable

- Term 1
  - Proposal document
  - Term 1 report
  - Related research paper/references about the topic
  - Application Design
- Term 2
  - Term 2 report
  - Released Application

CHAPTER 2 BACKGROUND THEORY AND RELATED WORK

Quantum Computing is by far the most advanced rocket we have launched into the vast realm of quantum mechanics. Despite the efforts of many individuals, including physicists, to understand the underlying concept, no one can truly comprehend it. In this chapter, we will go over all of the background information for our project, as well as the necessary theory and core concepts to understand before beginning to do the project, tools, related research, and competing solutions that we have gathered for refining and supporting our project to the best of our ability.

2.1 Theory and Core Concepts

2.1.1 Quantum computer

Unlike a classical computer that uses the state of bits in 0s and 1s, the quantum computer computes and calculates using quantum states and superpositions from quantum theory which is called qubits. Qubits use the property of superposition in quantum theory to represent the possibility of the state of 0 and 1 simultaneously. Qubits are most likely to visualize in Q-sphere that one point in Q-sphere surface is one value of Qubits. Many theories are used on a quantum computer to use the theory to solve the problems that consume resources on a classical computer but easy solved on a quantum computer, like superposition or entanglement.

2.1.2 Quantum circuit

A quantum circuit is a model for quantum computation that uses the quantum theory like superposition or entanglement through quantum gates to create the circuit that runs on n-qubit registers. In a quantum circuit, there are 2 types of running, one is simulating that will run the circuit in ideal condition and possibility of superposition based on the quantum theory, and another one is lunch on to the real machine on the provided provider, like IBM, in this running type will use the real situation based on the provider machine.

2.1.3 IBMQ (Quantum architecture)

IBMQ is a collective of IBM quantum computer architectures that uses n-qubit registers constructed with a variety of structures, and each machine has different basic gates so that before running the code from the user, the transpiler will transpile it before running on the machine. The various architectures make the performance of each machine different, in the same code if lunch in 2 machines that are similar to everything but the structure, the result, running time, or the accuracy might be different.

2.1.4 Source-to-source compiler (Transpiler)

A source-to-source compiler converts and optimizes the input programming language into the same programming language as the output. It might be just a simple find and replace text to advance lexical analysis and parsing, create intermediate data structure, optimizing the structure and generate source code from that data structure corresponding to the target language. We can found source-to-source compiler in everyday life such as translating language from one language to another language. Because language is just a tool to carry information from one person to another, so what language translator do is to convert the information he receive and convert it to target language. Another example is on the website that you use everyday. Most website nowadays is written in another language such as TypeScript and SCSS because these language provides expressive and safer way. But the browser that you use supports only 3 language: HTML, JavaScript and CSS so we need to use source-to-source compilation to convert these languages into the language that browser supports,

not only that, even JavaScript itself have many iteration, new standards and features come out every years and browser vendor cannot keep up with that so many developers use source-to-source compiler to compile newer standard of JavaScript into older standard JavaScript so it remains compatible with older browser and wider range of audience.

2.1.5 Simulator

A simulator is a system that imitate behavior the intended device by other means. Simulator is the proof of concept of the work to confirm that in the perfect condition, the core concept exists. Most simulator do not considered all noisy parameter that can be found in the real machine so its result can be considered as an ideal result and it is a good estimation to the result from the real machine. Anyways the simulator can provide a real condition to sneak peek the result before the real launch, because the real launch may cost too much resource to perform a single run, so the real-simulation can be simulated the result to not waste the resource.

2.2 Languages and Technologies (Stack)

2.2.1 Python

Python is a high-level programming language that is utilized in a wide range of industries, from science to embedded devices. Python can run on a variety of platforms, including Linux, Windows, Mac OSX, and even very limited systems like Micro-controllers with only 64 KB of RAM. In contrast to many C-derived languages that use brackets to denote a block of code, the Python language has a fairly straightforward syntax that matches the English language and employs indentation to indicate a block of code. Python supports a variety of programming paradigms, including imperative, object-oriented, and functional programming, thanks to its simple programming model, dynamic typing, and automatic memory management.
We chose Python because it is a straightforward and easy-to-use programming language that is also easy to work with. Python is as quick as many other languages, but it also includes a powerful standard library and a large third-party library that can accomplish everything from quantum circuit simulation to data processing, web framework, and GPIO pin toggling. We also utilize IBM's Qiskit library, which is written in Python; if we use other languages, we will have to build a bridge between them, which will increase the overall complexity of our system.

2.2.2 Qiskit

Qiskit is a quantum circuits library written in Python. It can construct a quantum circuit using both basic and advanced gates like the Pauli X gate and the Controlled-X gate. It also lets you use a matrix or an existing gate to create your gate. Qiskit can also display a circuit, simulate it, and send it to an IBM quantum computer to run. We chose Qiskit because it is the only library capable of transpiling and conducting quantum circuits on IBM's quantum computers. It is also the one that is being used with IBM Quantum Composer, the editor whose usability we are aiming to improve.

2.2.3 OpenQASM

OpenQASM (QASM) is a quantum circuit description and representation programming language. Because they are both used to define a physical connection between fundamental components in any system, the language shares many properties with traditional hardware description languages like Verilog. Primarily, Its function is to communicate with the hardware.

2.2.4 Flask

Flask is a web framework written in the Python programming language. Because it does not include any helper libraries, such as database-assisted frameworks or advanced libraries, the Flask framework is considered a micro-framework. It merely contains the essential functionality for building a website, such as route matching, cookie management, and template rendering. However, third-party extensions can be used to extend it. Flask is being used since it has everything we need to build a back-end for our editor. Other frameworks, such as Django or Pyramid, include too many features that the project did not require, such as database abstraction or database migration. Furthermore, those libraries take far too many steps just to develop a basic hello world website. As a result, the Flask framework meets the project's objectives and expectations the most.

2.2.5 JavaScript

JavaScript is a high-level programming language with a wide range of applications. Its grammar is based on Java, which is based on the C programming language. JavaScript is widely utilized in all major web browsers, including Firefox and Google Chrome. Despite the fact that it is the only programming language supported by modern web browsers, it is nevertheless a widely popular language with a structure that extends beyond web development. Because of its lexical scope and its unique prototype-based object-oriented among modern programming languages, JavaScript has a simple execution and memory model and supports numerous programming paradigms, notably functional and object-oriented.

2.2.6 IBMQ Systems

IBMQ features a large number of systems, each with its own set of characteristics such as qubits and fundamental gates. The stability metrics for each system include ”CNOT Error,” ”Readout Error,” ”Avg. T1,” and ”Avg. T2.” The CNOT Error occurs when the destination qubit does not flip properly to the control bit. A readout error occurs when the system reads the state of the system that is not equal to the real state of the system. The average time it takes for state |1> to decay to state |0> is T1. The average decay period of a pair of qubits' superpositions is T2. Each system also has a quality metric called ”Quantum Volume” which measures the overall ability to process a quantum circuit of its capability.

2.3 Related Research

2.3.1 An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures[1]

In recent years, quantum computers have evolved from academic research into usable computing devices. As seen from IBM's project IBM Q, with a goal of providing access to quantum computers to a broad audience, allowing anyone to do an experiment on their quantum computer up to 5 qubit device. In order to use these devices, we need to map quantum circuits under the physical constraints of the devices so that they can run on the devices. First, we need to convert high-level gates into elementary gates that are supported by the hardware. Second, the qubit of a circuit must be mapped to the underlying constraint of a coupling pair such that all operations can be done. But those maps usually change along the depth of the circuit. Additional gates, such as SWAP gates, are inserted to move qubits around to comply with the constraint and be able to perform the operation properly. But those additional gates affect the reliability of the circuit (they may introduce additional errors due to additional operations and increase the execution time), so their number should be kept at a minimum. But IBM's own solution, that is deployed in their Python SDK, Qiskit, fails in many circumstances since their usage of random search for mapping cannot find suitable mapping for some cases and cannot generate results in an acceptable time.

In this work, they introduce a multi-step approach to creating suitable mapping for the circuit by first deconstructing high-level gates into elementary gates that are supported by the target architecture, and then employing depth-based partitioning and A* as a backing algorithm and the ability to update the initial mapping along the mapping process (as opposed to fixing it at the beginning of the algorithm) to satisfy the CNOT constraint of a target device. This work is integrated very well with IBM's Qiskit. It can be used as a post-processor of Qiskit before using the circuit to further optimize the circuit to reduce its depth and gate count, thus increasing reliability and reducing execution time.

The experimental results in their paper show that the algorithm is highly efficient in many test circuits that fail to map properly with Qiskit. The algorithm can find a suitable mapping for the particular device within seconds when IBM's Qiskit takes more than an hour to find a suitable mapping. The result from the algorithm also has fewer gates and depth compared to the output from Qiskit, which made the circuit much more reliable and took less execution time.

2.3.2 Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Numbers of SWAP and H Operations[2]

A decade ago, quantum theory proved to be a superior way of computing, and several algorithms were developed for it. Nowadays, quantum computing is gaining a lot of traction. The real driver of this movement is the collaboration of big players in this industry, such as IBM, Microsoft, and Google. They aim to apply quantum computing to many difficult tasks nowadays, such as chemistry simulation, protein synthesis, optimization problems, machine learning, etc. But before we can run any circuits on these devices, some constraints have to be resolved. First, these quantum computers only contain elementary gates. It is the minimum gate set that can form a universal quantum gate. If the circuits contain any gates that are not in that set, they must be decomposed into elementary gates first. Second, not every qubit in the quantum computer can interact with other qubits. This constraint is known as the ”coupling map”. It is a map that contains which qubit can interact with which other qubit, which one is a control qubit, and which one is a target qubit. By the way, to overcome this constraint, we have to add SWAP and H gates to ”move” qubits around so they fall into those coupling pairs and can interact with each other. But by adding SWAP and H gates, we increase the circuit size and thus increase the error rate and execution time. Finding a SWAP and H gate combination that can satisfy the constraint while producing the smallest circuit is an NP-complete problem. Several solutions have been proposed to tackle this problem, but none of these methods solves the problem in exactly the same manner; instead, they employ a heuristic. That is a reasonable strategy to solve an NP-complete problem, but it remains unknown how well those methods are compared to minimal solutions.

In this paper, they propose a solution to compute a close-to-minimal SWAP and H gate within a reasonable time by converting all possible combinations of SWAP and H gate in the circuit into a Boolean satisfiable problem and letting the SAT solver resolve this large search space. They also show us how to limit search space while retaining a close-to-minimal solution. Although the method that was proposed in this paper is suitable for small circuits only. This may bring us a result that is not minimal anymore, but a close one. From the experimentation, it shows that the heuristic algorithm that IBM employed in their Python SDK, Qiskit, produces more than twice the minimum solution on average. This shows that there are more room to develop a better heuristic algorithm to map the circuit.

2.3.3 Exploiting Quantum Teleportation in Quantum Circuit Mapping [3]

Certain problems require quantum computers to give us a significant improvement over classical computers. But nowadays, state-of-the-art quantum computers are still considered noist intermediate scale quantum devices (NISQ) that have many constraints that we need to resolve before we can use them. It only allows interaction between specific pairs of qubits due to the inherent difficulty in hardware design.In a small circuit, we can find a suitable qubit mapping that can satisfy the entire circuit. But most of the time, we cannot. We resolve this by using quantum circuit mapping. By moving qubits around by using SWAP and H gates so they can interact with each other. By introducing an additional gate to overcome the couling pair problem, it leads to increasing errors and longer execution time. Also, swapping bits around is a classical way of doing it and does not use the available mechanics that the quantum world gives us.

In this paper, they explored the alternative ways to map a quantum circuit by using quantum teleportation as a transport mechanism as opposed to a SWAP chain. By using a quantum transport channel, we can transfer the state of a qubit over an arbitrary distance. This method allows us to move qubits around the quantum processor to overcome the coupling pair problem. By using quantum teleport, it has an advantage over a SWAP chain in that it has a fixed cost. Once the teleport channel has been established, it can transfer qubit state to anywhere, compared to SWAP chains that require a longer chain of SWAP gates for a longer transfer distance.

From the experimentation on the IBMQ Tokyo machine, it shows that this method can reduce overhead by 20% in conventional SWAP chains and up to 30% in some circuits. The authors of this paper believe that when a larger machine appears, this method will further reduce transfer overhead, overcome coupling pair constraints, and help reduce the error rate of the circuit.

2.4 Related Product

We looked for other products that were relevant to our theme. We discovered 3 intriguing quantum-circuit editors: IBM Quantum Composer, Quantum Programming Studio, and Quirk.

2.4.1 IBM Quantum Composer

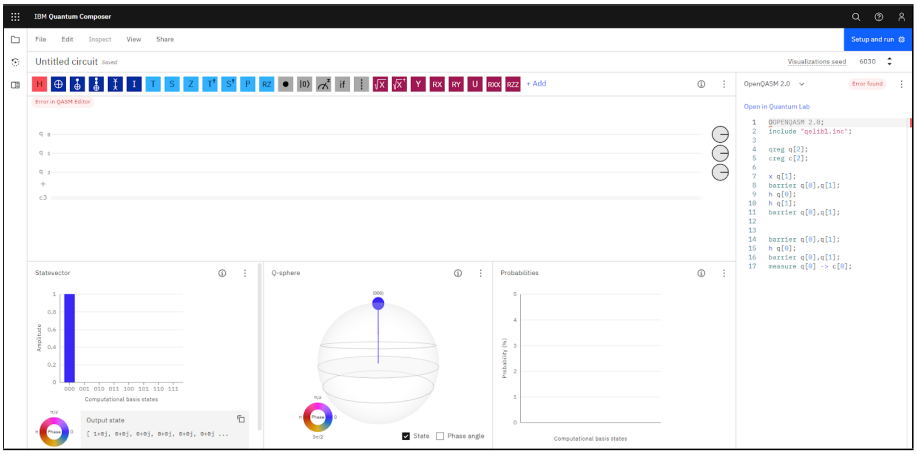

Figure 2.1 IBM Quantum Composer User Interface

IBM Quantum Composer is a quantum circuit editor developed by IBM that you can use online. It has a simple user interface and a number of gates for users to use, as shown in Figure 2.1. To build the circuit, drag the gate from the toolbar into the line of a qubit. It can also be changed from the right panel using the

OpenQASM language. Users can see the circuit's result in real-time thanks to a state vector chart, Q-sphere, and probability chart at the bottom. The simulation is customizable and is based on the seed in the upper right corner.

Although IBM Quantum Composer offers a lot of flexibility, it also has some limitations. There are a few gates with a poor user experience, such as the inability to delete gates by dragging them outside of the editing area. The gate builder has a poor user interface and is unable to construct a gate from a matrix. Users are less likely to use them for more complex projects because of these disadvantages.

2.4.2 Quantum Programming Studio

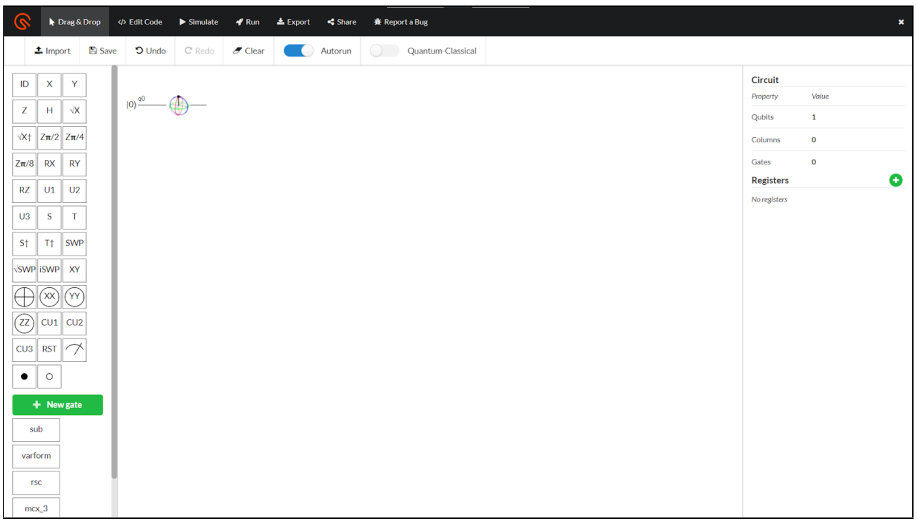

Figure 2.2 Quantum Programming Studio User Interface

Quantum Programming Studio (QPS) is a quantum circuit IDE that was created as a user interface for the `quantum-circuit` package in 2018. Since then, it has evolved into an important part of a company called "Quantastica." It has a large number of quantum gates and a spacious interface, as shown in Figure 2.2. To build a circuit, users can drag the gate from the left panel into the work area. The right panel will display information and options for the currently selected gate. Quantum Programming Studio can not only run on the simulator online, but it can also convert to a variety of other syntaxes and execute them with Rigetti QCS or IBMQ via its client.

Despite having a large number of quantum gates and being cross-compatible with many ecosystems, the editor itself has a poor and confusing user experience. Many beneficial capabilities have been overlooked due to a lack of proper user guides or explicit instructions, making the development process slow and inconvenient.

2.4.3 Quirk

Quirk is an open-source online quantum circuit simulator developed by Craig Gidney, a software engineer on the quantum computing team at Google. It includes a large number of quantum gates, as shown in Figure 2.3, as well as a clear and simple user guide. Users can easily use this editor by simply dragging and dropping it into the desired qubit line. As previously stated, Quirk is only a quantum circuit editor, which is the most significant disadvantage. Even if this software is an excellent tool for learning quantum algorithms and designing circuits before putting them into practice in a real-world setting. This software's circuit cannot be implemented on real-world hardware.

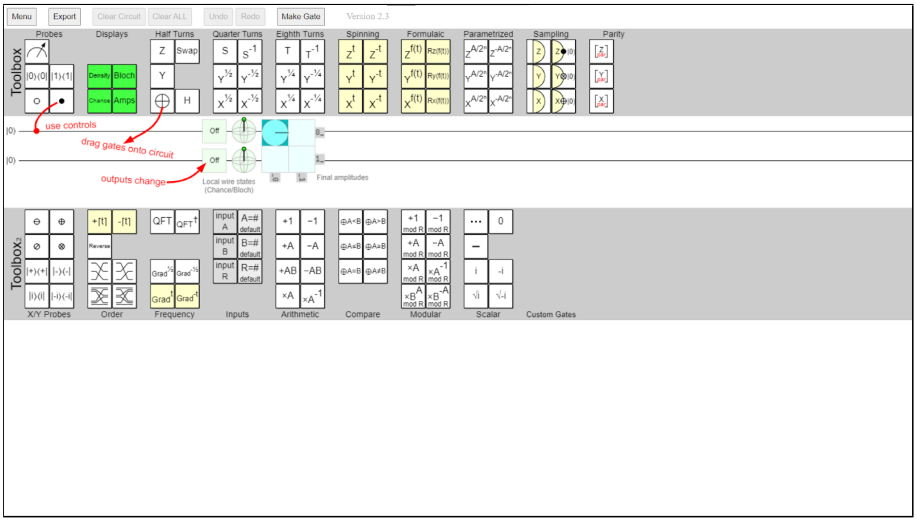

Figure 2.3 Quirk User Interface

2.5 Related Product Discussion

IBM Quantum Composer is the most relevant and closest to our needs since we are planning to develop a quantum circuit editor for the framework Qiskit. But generally, we can identify the pros and cons of each of the three quantum circuit editors after examining them. We have realized how crucial a good user experience is, as well as the importance of a good user guide. However, we cannot be certain that our viewpoints are as valid as they appear to be based only on our perspective and desire to create a better quantum circuit editor. As a result, we intend to perform a user usability test based on a variety of scenarios to understand more about our target consumers and the difficulties that this technology will provide them. After gathering and evaluating the data, we may use the comments as a starting point for designing and developing a better quantum circuit editor for our users.

If you want to find out about our user usability test and the results that we obtained, it is provided in Appendix B.

CHAPTER 3 PROPOSED WORK

To get the most out of a topic, research needs a well-defined framework and technique. In this chapter, we will go over the features we will be working on in this project, as well as the general system architecture and our project's evaluation plans.

3.1 System Specifications and Requirements

- Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation.
- Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit.
- Users can alter the compiler and optimization level as well as see the transpilation output that will be executed on the IBMQ system.
- Users can get IBMQ system recommendations from the application before executing the circuit on the hardware.
- Users can see a brief description of the gate when hovering over it.
- Users will receive instant and relevant feedback from the user interface.
- Users can compile the circuit into OpenQASM or Qiskit.

3.2 Application Architecture

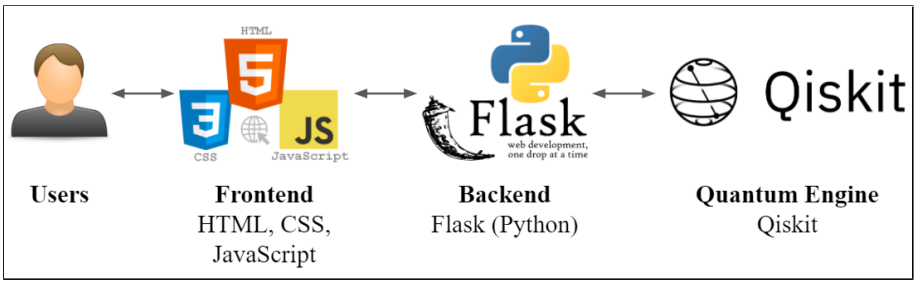

Figure 3.1 Application Architecture

The application will be developed as a web application with the overall system architecture, as shown in Figure 3.1. The frontend will use standard tools and languages like HTML, CSS, and JavaScript. Because of the ease of development with Qiskit, the quantum engine that will accomplish everything from simulation to execution on real quantum hardware, the server will use Flask and Python as a backend. The application's logic and algorithms can be found on both the frontend and the backend. In general, the circuit design logic will be on the frontend, while the logic involving transpilation and IBMQ systems will be on the backend.

3.2.1 Detailed description of each section

- Front-end is a piece of software that user interact with. It has some logic on it such as circuit validation, circuit compaction, etc.
- Back-end is a central piece of our system. It manage all data in this system. It create new Transpiler instance for each user editing session. It accept circuit information from frontend and pass it to managed Qiskit instance to compile and analyze circuit with user updating the circuit. After transpiler and our algorithm analyze circuit complete, It then send analyzed result back to the user.
- Quantum engine is a software that we use to parse, validate and convert the circuit into analyzable form.

3.3 Component Diagram

From Figure 3.2 you will see use case diagram of our quantum composer web application. It consists of 3 parts, user, application and external provider.

- **User** is the user of our system. They interact to our system via front-end of our application and can create circuit, edit circuit, transpile circuit and execute circuit via external provider or in our simulator.
- **Application** This is our application. It consists of following component
  - **Front-end** This is the main interface of our application. User will use our application via it.
  - **Back-end** This is the core of our application. It facilitate job from the user to qiskit or external runtime such as IBMQ. It manage qiskit runtime per one session of user. When user enter our editor, It create a editing session and spawn qiskit to handle circuit for that session. When user edit the circuit, front-end sends update to back-end, that will recompile the circuit and sends recommends system and analyzed result back to front-end. When user executes the circuit, they can select whether to run in our simulator or run on external provider.
  - **Qiskit** This is our Quantum engine. It can transpile the circuit into OpenQASM, decompose circuit into basis gate circuits that we use to analyze the circuit. Also, It can connect to IBMQ and queue the circuit to run on real machines.
- **IBMQ** IBMQ is Quantum computing platform operated by IBM. It consist of multiple Quantum computer that user can access and view their property. Personal account user can access up to 5 qubit machines. Their system have Quantum computer system that have up to 127 qubits but if you want to access you must contact them. They also act as a Platform as a Service that you submit your code to run on their quantum computer and wait for your program to run and wait for your program's result.

3.4 Use case diagram and use case narrative

The use case diagram and use case narrative are the ways to explain the scenarios that can happen when a user is using the application. The diagram and description below will explain to you how we design the way users will interact with the application created from use case and function requirements.

3.4.1 Use case diagram

From Figure 3.3 you will see our use case diagram. This is a diagram that describes how we design the way users interact with our application. First, they will see the edit circuit interface. The open circuit and creating a new blank circuit are just special cases of edit circuit. In that interface, they can edit the circuit by adding

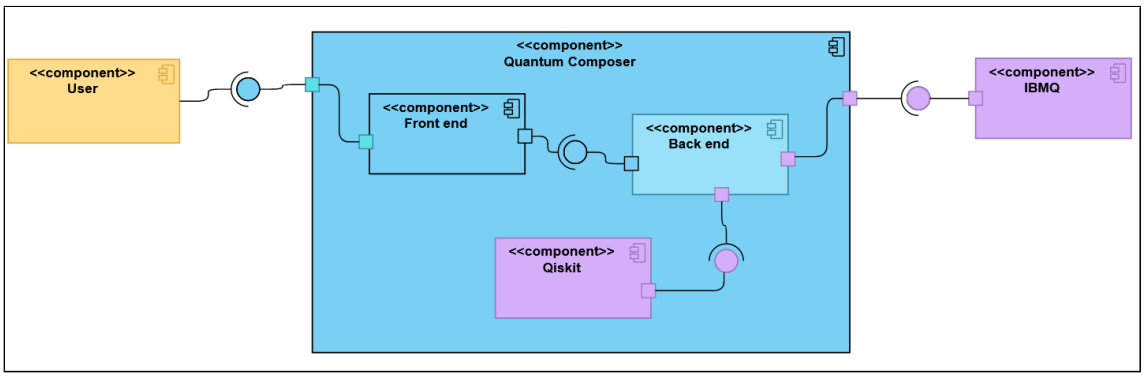

Figure 3.2 Component diagram

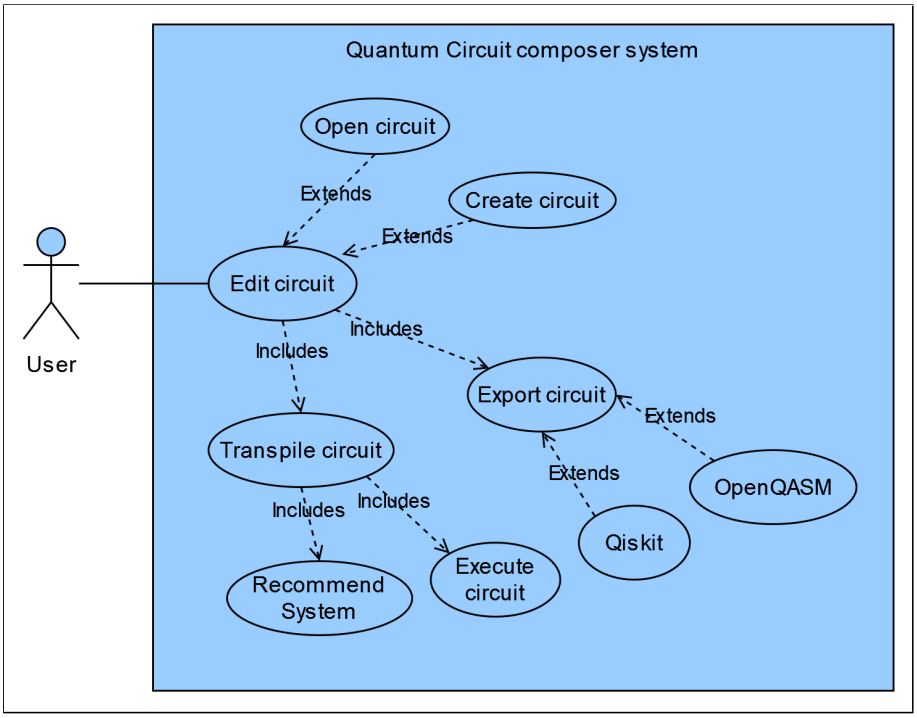

Figure 3.3 Use case diagram

more gates, moving gates around, adjusting gate parameters, etc. After that, they can transpile the circuit. It will show the recommended systems and they can choose which system to execute the circuit. They can also export the circuit into many formats, such as Qiskit's Python code or OpenQASM code.

3.4.2 Use case narrative

Scenario 1: Build circuits

1. **Goal:** To build circuits using fundamental gates
2. **Main Success Scenario:**
   (a) User open our application
   (b) User found blank circuit with 3 qubit registers and 3 classical bit register
   (c) User drag gates from top toolbar and drop into qubit registers

(d) Gate will persist in that qubit register in the position that you dropped into.

Scenario 2: Generate new gate using a matrix or rotation

1. **Goal:** To create new gate from rotation or a matrix
2. **Main Success Scenario:**
   (a) User open our application
   (b) User clicked new gate button
   (c) User enter name of this custom gate
   (d) User select types of data they want to input as a gate
      i. Select matrix
         A. Input number of qubit registers to operate on
         B. Matrix input will scale according to number of qubits to operate on
         C. User fill value into the matrix data input
      ii. Select rotation
         A. User enter rotation of each axis
         B. It shows that it will create custom U gate
      iii. Select gate group
         A. It tells user instruct to create custom gate from gate group and bring user back to canvas.
         B. User selected multiple gates, maybe across many qubit registers.
         C. User clicked convert to gate button
         D. It till warn user, ”This will flatten all operation into a matrix. Proceed?”
         E. User press proceed.
   (e) User finish creating by press OK button
   (f) New gate will appear in the custom gate section of toolbox on the left

Scenario 3: Alter transpilation parameters

1. **Goal:** To adjust Qiskit transpilation parameter before transpiling
2. **Main Success Scenario:**
   (a) User clicked at Execute button
   (b) User adjust transpilation parameter

Scenario 4: View transpilation output

1. **Goal:** To view the result of transpilation before running on real hardware or a simulator
2. **Main Success Scenario:**
   (a) User clicked at Execute button
   (b) Transpilation output will showed up

Scenario 5: Group quantum operations together and reuse them elsewhere by copying and pasting them

1. **Goal:** To help user creating a circuit that has multiple duplicating sub units such as grover search circuit.
2. **Main Success Scenario:**
   (a) User select gates from circuit
   (b) User clicked on Group button in the toolbar
   (c) It will ask for group name
   (d) User enter group name and done

Scenario 6: Get IBMQ system recommendation from the application before executing the circuit on the hardware

1. **Goal:** To aid user which system they should use to run their circuit
2. **Main Success Scenario:**
   (a) User clicked execute button
   (b) That dialog will show recommended IBMQ system

Scenario 7: Sees brief description of each gate in the circuit

1. **Goal:** To aid user which system they should use to run their circuit
2. **Main Success Scenario:**
   (a) User hover cursor over some gate in the circuit
   (b) It will show tool tip that has following information
      - Gate name
      - Gate description (what it does)
      - Gate parameters

Scenario 8: Export circuit into OpenQASM or Qiskit code

1. **Goal:** To export a circuit into portable format such as OpenQASM or into Qiskit's python code to use on other platform.
2. **Main Success Scenario:**
   - Export into OpenQASM
     (a) Open code editor plane
     (b) The QASM code will be there
   - Export to Qiskit Python code
     (a) Open code editor plane
     (b) User click export
     (c) Export dialog will show
     (d) Select export type: Qiskit
     (e) Qiskit code will appear

3.4.3 Sequence Diagram

As you can see in Figure 3.4, this is the main sequence of our application. First, when a user enters our application, it sends an initiation session request to our back-end. The back-end then creates a new Qiskit instance and a temporary data structure to store information for that session. After that, it sends the Session ID back to the front-end. Front-ends receive the Session ID and then initiate the User Interface to its initial state. The user then sees the blank circuit, ready to change to their circuit. At the same time, the front-end submits those edits to the back-end to let the back-end analyze the circuit and return the analyzed result back to the front-end to show it to the user. This makes our editor look responsive because it does not wait for the user to submit the circuit to the back-end by themselves. This was handled by our system automatically. They finish editing or executing after the user has finished editing or executing. may close the editor, which sends a destroy session request to the back-end, thus cleaning up the temporary data structure that our back-end holds for that session and terminating the Qiskit instance, freeing up resources in the system.

3.5 User Interface Design

You may discover the results of our user usability test on comparable items in Appendix B, as we showed you in the previous chapter. After we had finalized the study's findings, we began to build our editor to be as simple as possible in order to enhance user usability. The interface has also been created around various situations and use cases, which may be found in Appendix C.

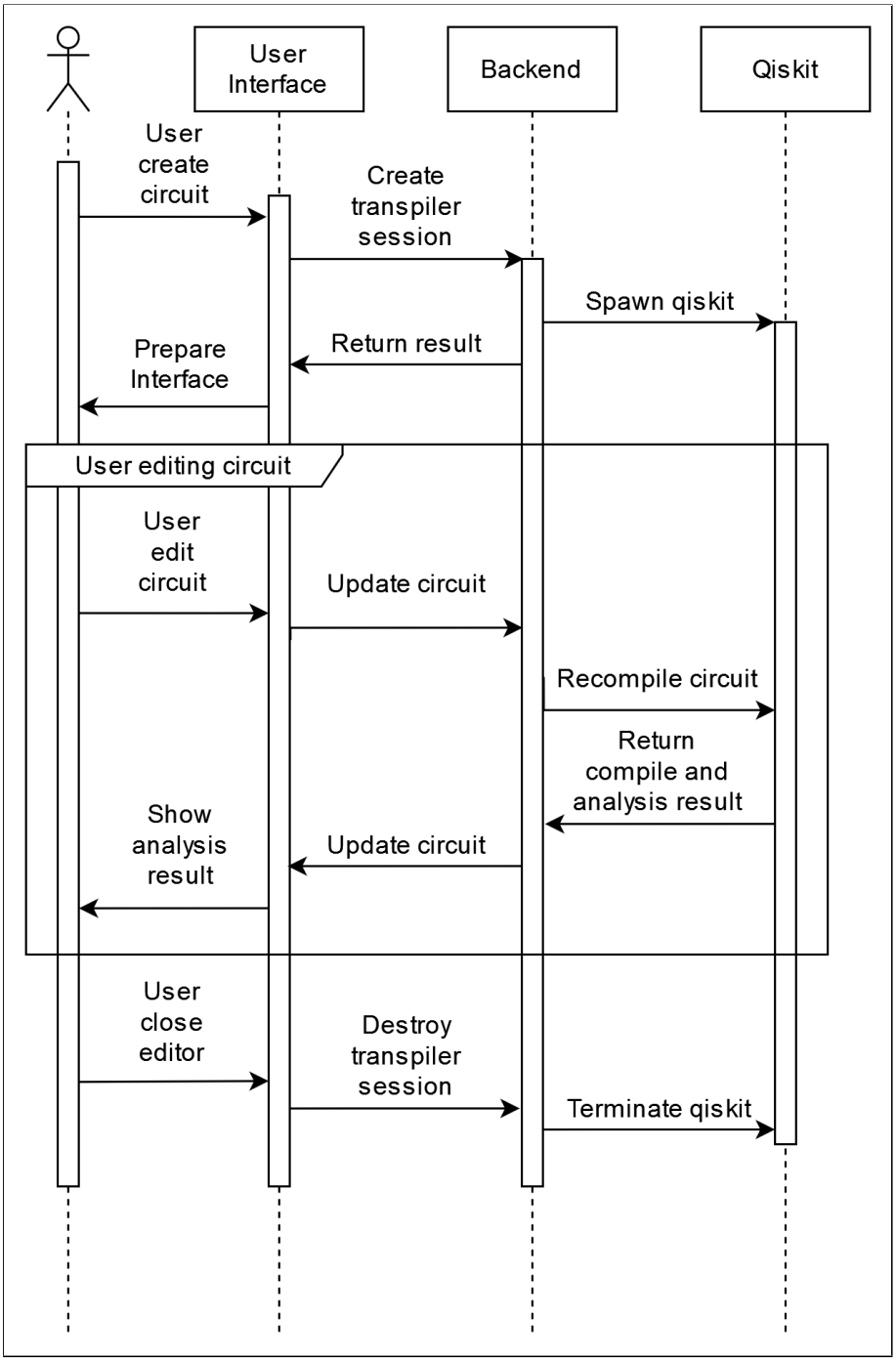

Figure 3.4 Edit circuit sequence diagram

CHAPTER 4 IMPLEMENTATION RESULTS

After establishing a framework and methodology in the previous chapter, we will proceed to our real implementation. In this chapter, we will discuss about the overall implementation in Section 4.1 and changed specifications in Section 4.2. After that, we will show you the development result of the editor in Section 4.3, and then go ahead and explain the recommendation system in Section 4.4. Finally, we will talk about all of the API endpoints which connect the editor to the recommendation system in Section 4.5.

4.1 Overall Implementation

Table 4.1 depicts our application's overall implementation. It comprises a breakdown of the preceding chapter's specifications and requirements, as well as the status of the project and any notes that may be relevant to that part. Briefly, we were able to complete 92.3% of our project (100% progress on 12 of 13 features). And the next section will describe any features that we were unable to complete due to technical difficulties and changed specifications.

Table 4.1: Application development overall progress

#	Requirement	Progress	Note
1. Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation			
1.1	Create circuit using fundamental gates	100%	-
1.2	Create custom gate using matrix	100%	-
1.3	Create custom gate using rotation	100%	-
2. Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit			
2.1	Group circuit together	0%	Changed specification due to technical problems
2.2	Copy and paste a part of circuit into the other part of the circuit	100%	-
3. Users can alter the compiler and optimization level as well as see the transpilation output that will be executed on the IBMQ system			
3.1	Change transpiler options	100%	-
3.2	Change optimization level	100%	-
3.3	See transpilation output	100%	-
4. Users can get IBMQ system recommendations from the application before executing the circuit on the hardware			
4.1	See IBMQ system recommendation before running the circuit	100%	-
5. Users can see a brief description of the gate when hovering over it			

Table 4.1: Application development overall progress

#	Requirement	Progress	Note
5.1	See the gate information	100%	-
6. Users will receive instant and relevant feedback from the user interface			
6.1	Users can use the application effortlessly	100%	Progress validated by conducting a user usability test
7. Users can compile the circuit into OpenQASM or Qiskit			
7.1	Get OpenQASM output	100%	-
7.2	Get Qiskit output	100%	-

4.2 Changed Specifications

From the previous section, most of our application are completed and finished. However, during the development, some unforeseen problems and limitations have appeared into our view. In the Table 4.2, we have summarized the specification have been altered to accommodate these technological difficulties and limitations during development.

Table 4.2: Changed specification detail

Proposed Requirement	Current Implementation	Unchanged Requirement
1: Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation		
When creating a new gate from a rotation, it will show the result as a decomposition of a U3 gate. And it will be able to create a new gate from a part of the circuit.	When creating a new gate from a rotation, it will show the result of the rotation of a vector in a bloch sphere. And it is not able to create a new gate from a part of the circuit.	Create a circuit using fundamental gates. Create custom gate using matrix and rotation.
2: Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit		
The user can create a group consists of multiple quantum gates.	The user cannot create a group consists of multiple quantum gates.	Copy and paste a part of circuit into the other part of the circuit.
4: Users can get IBMQ system recommendations from the application before executing the circuit on the hardware		
After opening an execute dialog, the dialog will automatically show a recommended IBMQ system.	The user must manually click ”Get Recommendation” button to get a recommendation for a statistically optimal configuration and suitable IBMQ system.	-
5: Users can see a brief description of the gate when hovering over it		
The tooltip will show gate’s name, gate’s description (of what it does), and gate’s parameters.	Since some gates can have a complicated parameter, gate’s parameters will be contained in gate’s description in some gates instead of all gates.	The user can see the tooltip by hovering the pointer over the gates.
7: Users can compile the circuit into OpenQASM or Qiskit		

Table 4.2: Changed specification detail

Proposed Requirement	Current Implementation	Unchanged Requirement
The code editor plane will contain QASM code and the user can get Qiskit's Python code by exporting.	The code editor plane will contain Qiskit's Python code and the user can get QASM code by exporting. The user can also find Qiskit's Python code in exporting dialog.	-

4.3 The Editor

4.3.1 User Interface

After experimenting with several technologies and creating a number of prototypes presented in Appendix D. Our development yielded the following results:

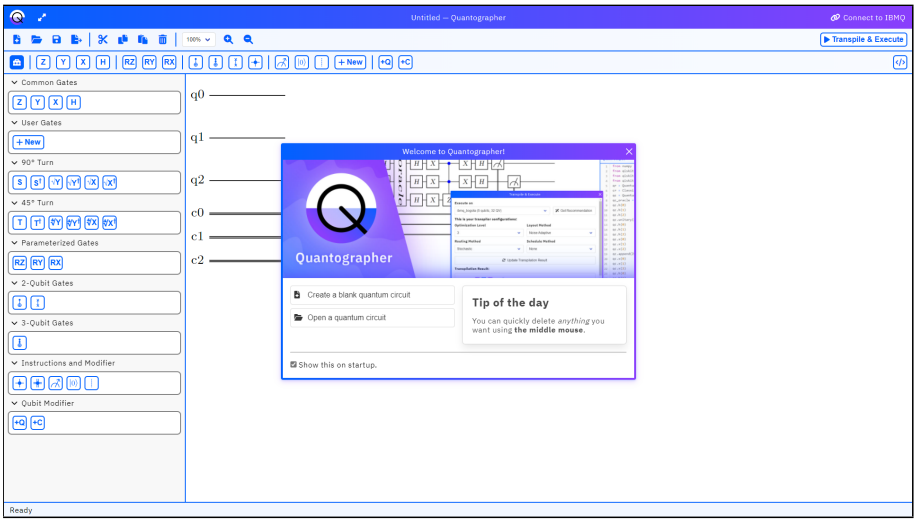

Figure 4.1 Startup interface

As you can see in Figure 4.1, our program was purposefully created with a simple style with toolbars and panels. This allows users to map their past knowledge from other programs to our application, allowing them to accomplish their work more successfully.

The main workspace is in the center. Appending a gate to a circuit is as simple as dragging a quantum gate from the toolbar or the gate palette panel (on the left) onto the editor in the middle.

The user can also easily create their own customized gate by clicking on the "New" button. The customized gate can be created using either a qubit rotation or a matrix (as in Figure 4.3 and Figure 4.4).

There are two primary panels in our application (as in Figure 4.5). The first is the "Gate Palette" panel, which contains every fundamental quantum gate classified by attributes such as the rotation angle or the number of qubits. To accommodate their preferences, users can switch between each category as they like. The second panel is the "Code" panel, which displays the circuit's output as Qiskit's Python code. The user can also easily switch individual panel to broaden their workspace, as in Figure 4.6.

In Figure 4.7, when the user clicks the "Transpile & Execute" button, the application will submit the IBMQ API key through an API request to get all of available systems. After that, the user can adjust the parameters to their liking, or they can press the button "Get Recommendation". By pressing the button, the application will

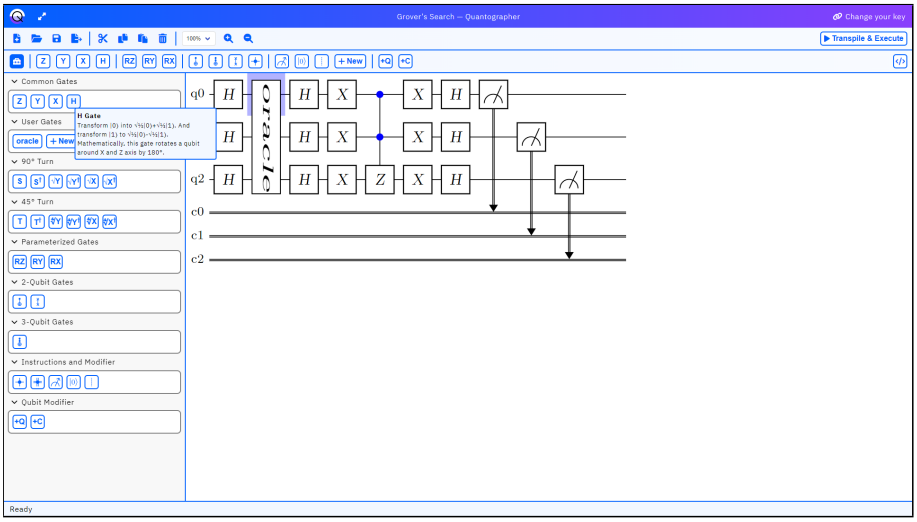

Figure 4.2 Circuit editing and user feedback

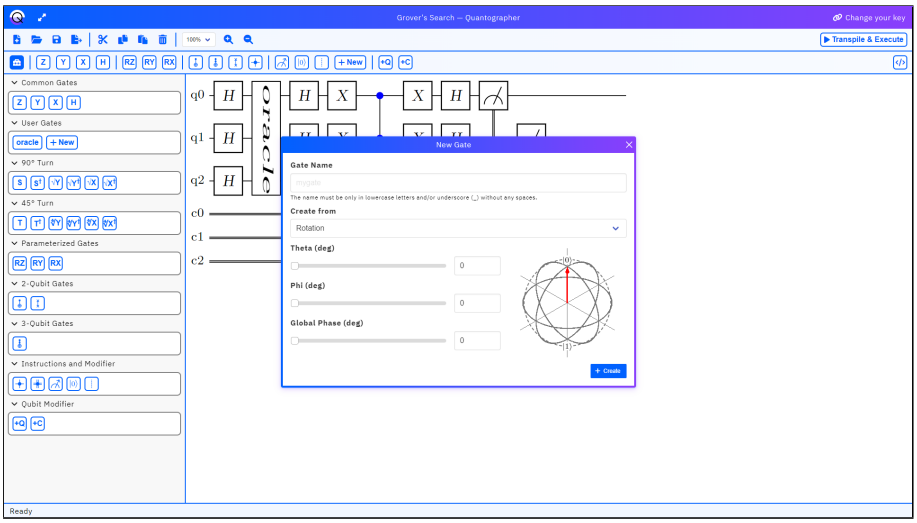

Figure 4.3 Create new gate window using a rotation

submit the circuit data to the backend and find the statistically optimal configuration. The backend will then respond with the configuration of that circuit and inform the user (Figure 4.8). As stated earlier, the user has complete freedom to customize the execution settings to their own requirements before executing the circuit on their selected system.

After the user is satisfied with the transpilation output, they can submit their circuit to be execute on real quantum hardware. When you scroll down the execute window, you will see the ”Execute” button. After you press the button, it will send the circuit data to the backend (Figure 4.9). If your circuit is valid and can be run on IBMQ, your circuit will be queue along other quantum circuits from other users. As you can see in Figure 4.10, the user can see what order of their queue is, and the estimated time of their circuit execution.

After the execution, the application will show the result of an execution. It is able to show either key-value pair data between a result and its count (Figure 4.11), or a histogram of the data (Figure 4.12).

Aside from the Qiskit’s Python generated by the editor, the user can export the circuit into QASM code, as in Figure 4.13.

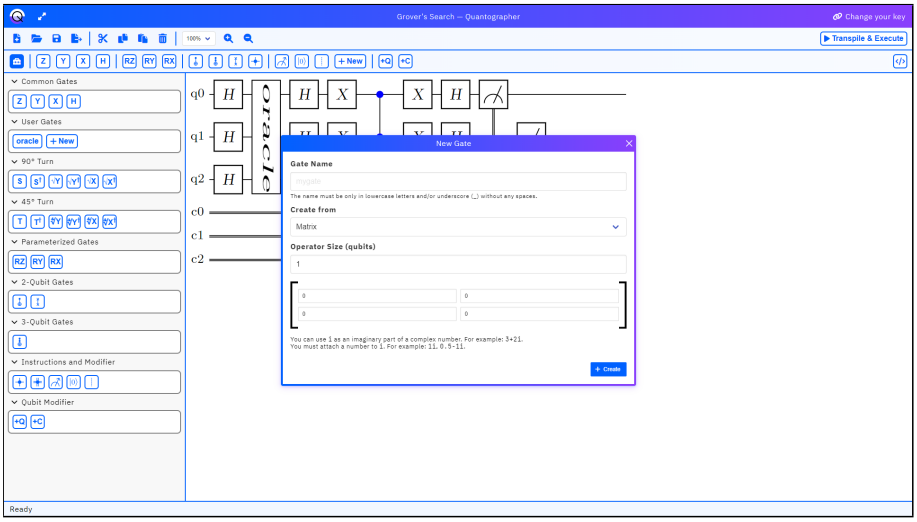

Figure 4.4 Create new gate window using a matrix

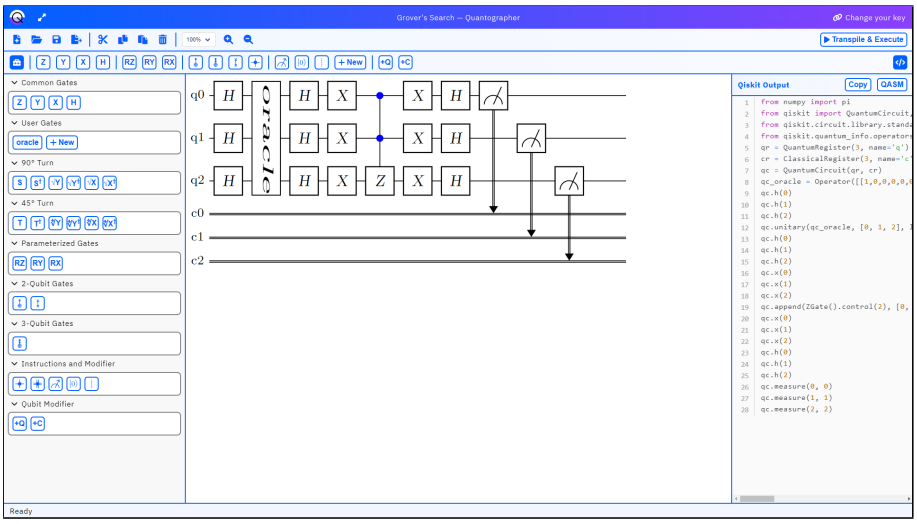

Figure 4.5 Two main panels of the application. On the left, users can toggle the category of the gates to their preferences

4.3.2 User Usability Test

We conducted the same user usability test to the same participants. Yet, in addition to our unique features, we created one new scenario to test them. In summary, many feedback came out positive. However, majority of users complaint about the location of QASM code and how challenging it is to edit a circuit to some degree. At the moment, we have already adjusted some of the problems to improve the usability for the user. Still, some features might have to be developed in the future development.

Moreover, after asking participants to order their preferred quantum circuit editors. We then grade the most preferred for 4 points, to the least as 1 point (since we have 4 products to be scored). Our application apparently is the most preferred choice with 28 points, follow by IBM Quantum Composer with 24 points, Quirk for 19 points, and Quantum Programming Studio, which appears to be the least popular one, with 10 points.

For the detailed data, see Appendix B.

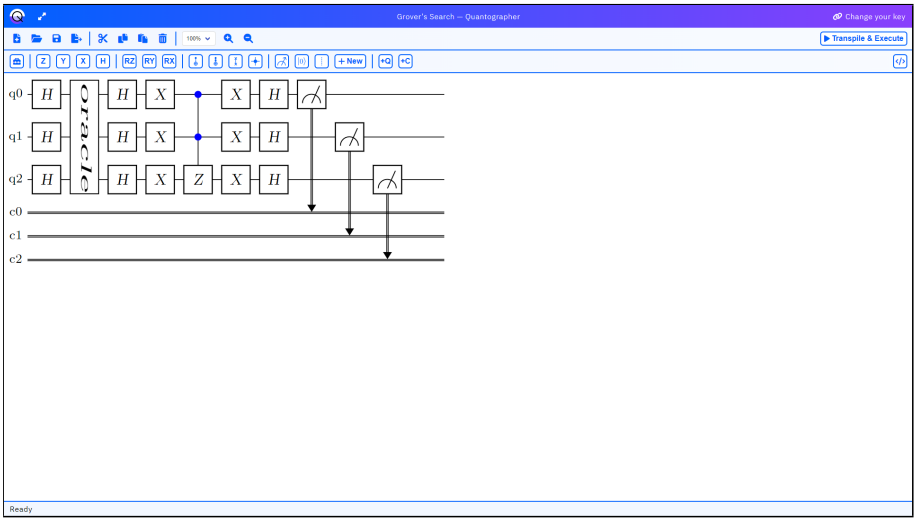

Figure 4.6 The editor when both panels are closed

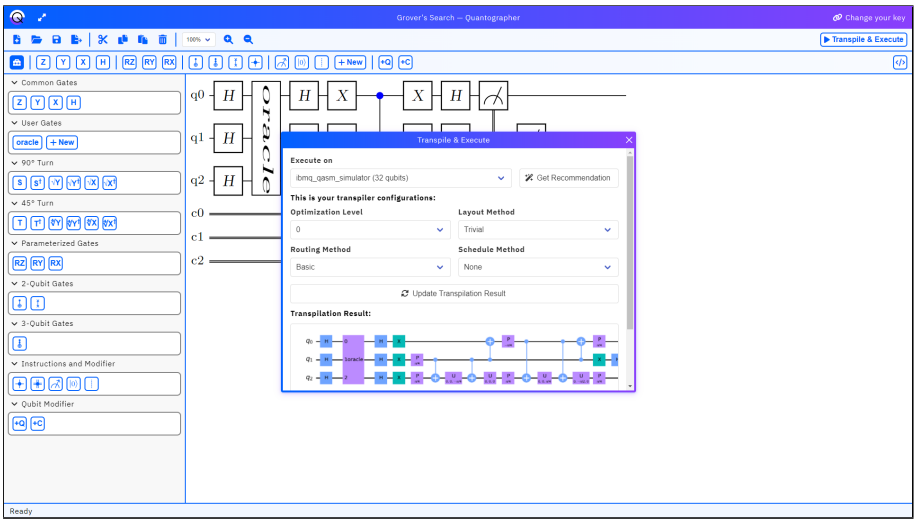

Figure 4.7 Transpile & Execute circuit window

4.4 Recommendation System

In our recommendation system, users must supply an IBMQ API key into our software, allowing our software to interact with the IBMQ API on behalf of the user. Our software fetches the list of IBMQ machines that users are allowed to use. After that, it uses information about those machines and uses information that we collected from Table A.1, Table A.2, and Table A.3 to choose the optimal machine. We did not brute force through all the parameter combinations because it takes a lot of time to compute. Instead, we just pick these parameters (layout method, routing method) which are (noise_adaptive, stochastic), (noise_adaptive, basic) and (trivial, basic) and with an optimize level from 0 to 3. Because from our experiment, it gave a circuit with the least error. We did not set a scheduling method because it does not affect the result in a significant way. Our software gets the statistical information from a real machine. One such piece of information is the error of each basis gate on the machine. It uses this information combined with transpiled circuits that are also transpiled by using this information, fed into Equation A.1 to get an estimated error of a circuit. This equation works by combining gate error and the amount of that gate in a circuit together. A deeper explanation can be found in Appendix A. Our software collected estimated errors for all available machines and all parameters and sent

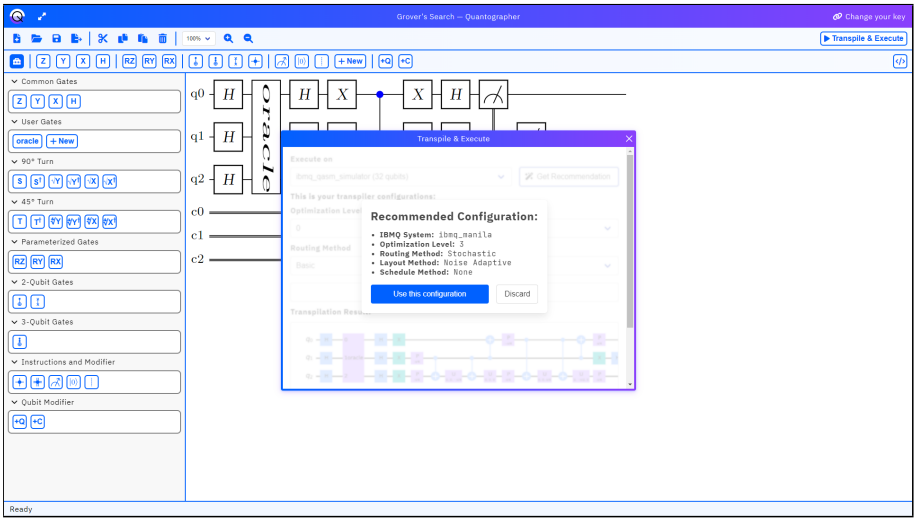

Figure 4.8 Recommended configuration received from the backend

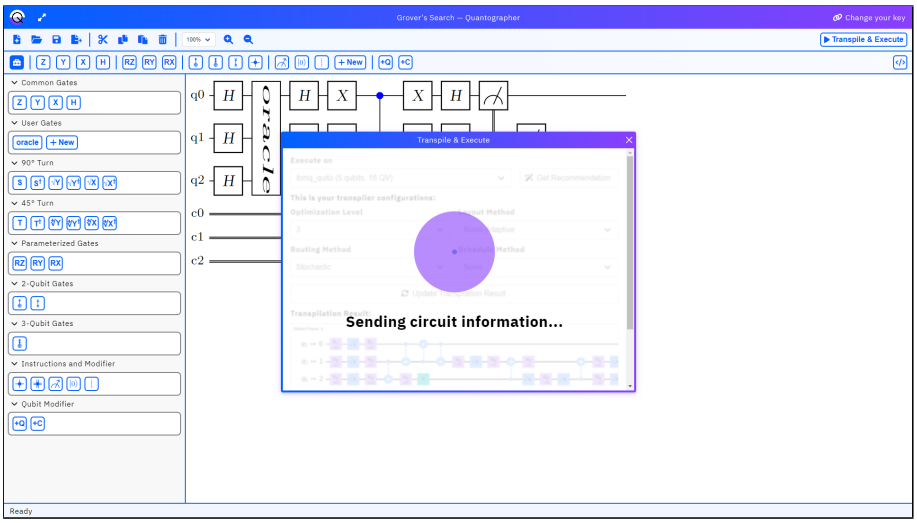

Figure 4.9 The application is submitting your circuit data to backend to be executed

them back to the user interface, as seen in Figure 4.8. The user interface will sort the result from the lease estimate error to the greatest and suggest the user with the least error. If the user accepts that configuration, it will set the runtime transpile parameter to it. The overall process of the recommendation system can also be found in the Figure 4.14.

4.5 API Endpoints

These API Endpoints were created to connect to the User Interface, handle the job that cannot or difficult to be done at User Interface. The objective of endpoint, the name of url, and the requested body is listed at Table 4.3. Note that all of the API endpoints use POST method.

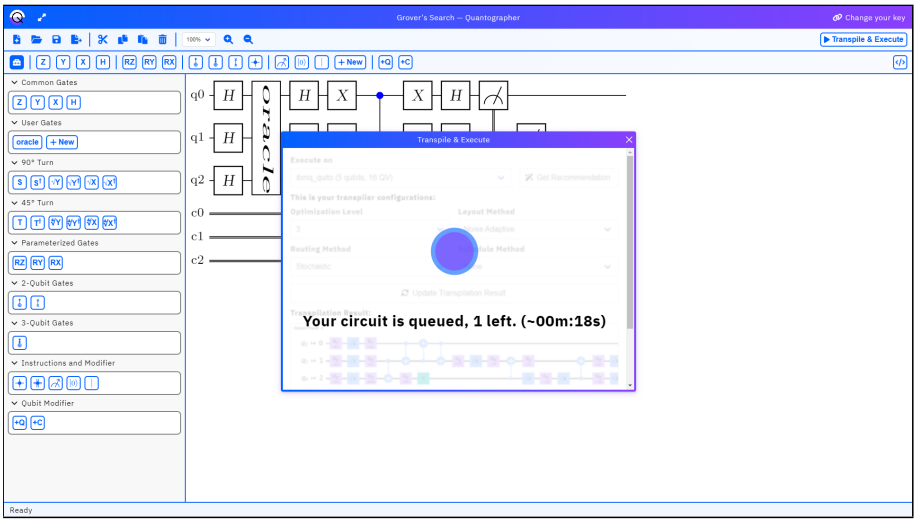

Figure 4.10 The information while your circuit is waiting for an execution

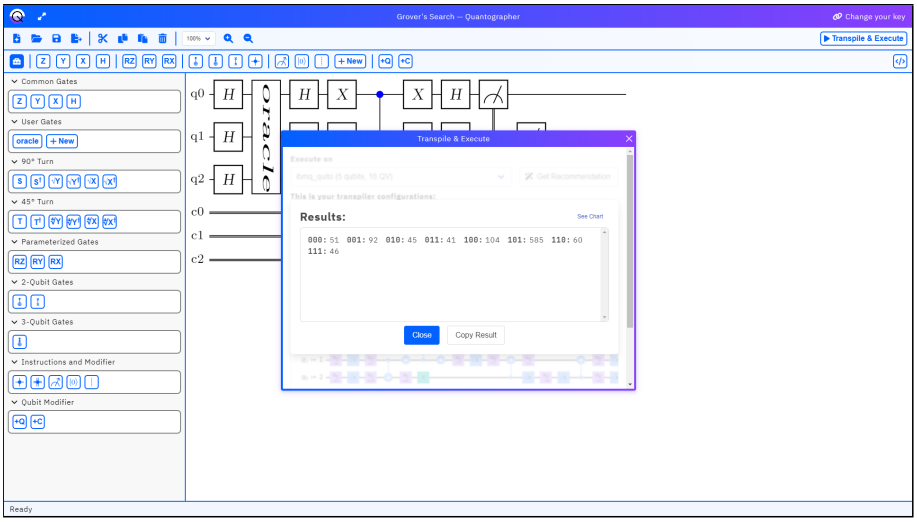

Figure 4.11 The result of the execution shown as a key-value pair between a binary result and its count

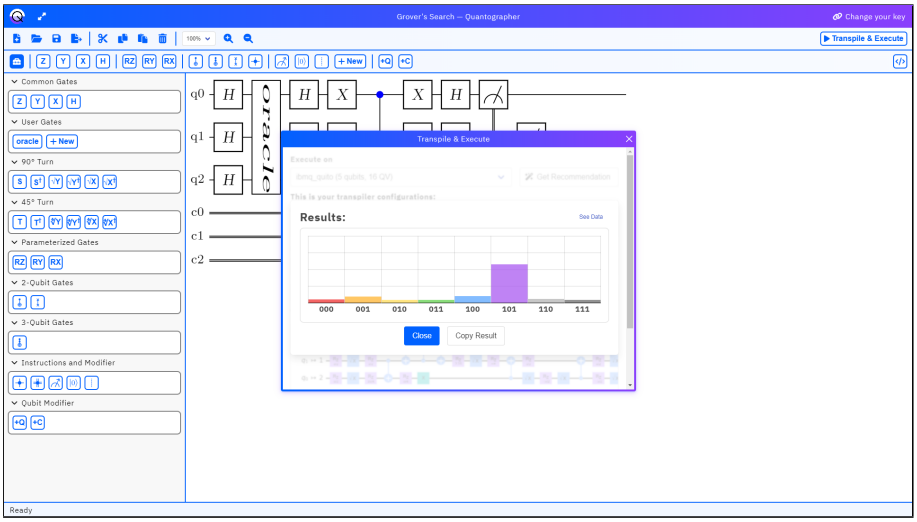

Figure 4.12 The result of the execution shown as a histogram between a binary result and its count.

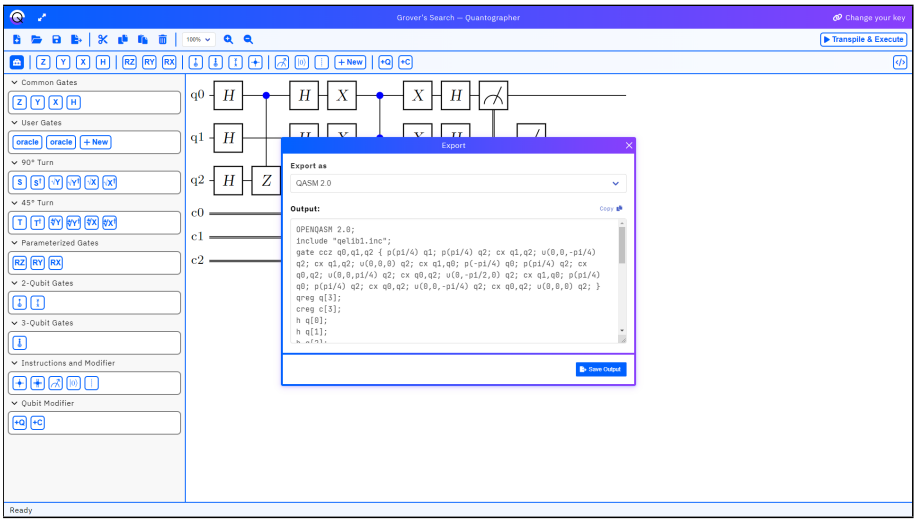

Figure 4.13 Export window

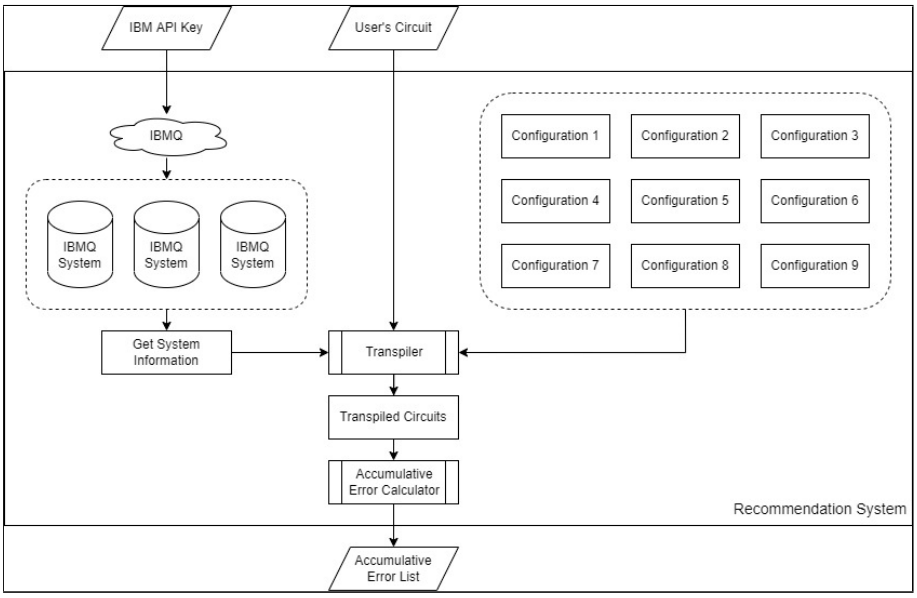

Figure 4.14 Process flow diagram of the recommendation system

Table 4.3 API Endpoints List

URL	Body (JSON format)	Objective
/convert_qasm	code	Receive JSON containing user's circuit then return qasm converted circuit back as a string.
/convert_image	code	Receive JSON containing user's circuit then return base64 formatted string of circuit picture.
/recommend	key, code	Receive JSON containing user's circuit and then loop through optimized configuration list to find the least accumulative error from the transpiled circuit.
/transpile	code, key, system, layout, routing, scheduling, optlvl	Receive JSON containing user's circuit, user's qiskit api key, and configuration of transpiler then transpile the given circuit and return a base64 formatted string of transpiled circuit picture.
/available_backend	key	Receive user's qiskit api key then query available systems along with number of qubits and quantum volume.
/run (Websocket)	key, system, code, layout, routing, scheduling, level, shots	Websocket that receive JSON containing user's circuit, user's qiskit api key, and configuration of transpiler then execute circuit to a real IBM quantum computer and keep returning status data.

CHAPTER 5 CONCLUSIONS

5.1 Discussion

We have created a quantum circuit editor in this project to help end users comprehend, analyze, optimize, and select the statistically optimum transpiler for their quantum circuit. We performed transpiler benchmarking using the advantages of the Qiskit library and incorporated the results to assist us prioritize the settings in our recommendation system. As a result, our recommendation system can assist our users in identifying the IBMQ architecture that is suited to their circuit and displaying the order of the instructions that will be executed on it. Furthermore, we conducted a user usability test comparing our solution to similar products. The test revealed that our editor is the most preferred option among our participants, and we have gotten a great number of favorable comments. However, there will always be issues and concerns that we would like to discuss and perhaps shape the next chapter of our product.

5.2 Problems and Solutions

- **Technical Debt:** From the beginning, our team came together with the common goal of creating a better quantum circuit editor. But we did not have a clear picture of our program, nor how our program should be. So, we did some research on existing software. Some of it has been used by our team before, and we collected the pain points of those programs. We also conduct user usability tests, where we invite other people who have been using quantum circuit editor to test an existing product and collect feedback from them. We gradually shape our software from this information, including feedback from our advisor and from discussion in our team. But as time went on, we could not wait until we got the entire specification of the program, so we began to construct the program anyway. When we build a program with a partial specification, we need to make up something, and sometimes that assumption is incorrect. Some decisions that we made a long time ago in the foundation of our software ended up preventing us from making many improvements. Such as, we use an internal mutable array as a core data structure. This simplifies many things in our software, but makes the undo and redo features a very challenging feature to add. This makes our software codebase look confusing because we have to overcome and avoid those technical problems. So, for the future work. We can use the insight that we have collected for this software as a base for the next software and the next implementation will not suffer the same fate as us.

- **Mental Health:** We cannot deny that mental wellness plays a significant role in our development. To build a software in a certain time limit, while also worrying about one's career after graduation, takes a huge toll on one's mental health. Cooperating with the COVID-19 situation in Thailand, with two of our team members suffering from the illness, our work was put on hold several times. This makes our desire of creating software a burden. As a result, many of the features must be removed from the final product. We would like to emphasize that we take our lives very seriously, as young adults learning how to deal with their lives, is a demanding duty that is far from a bed of sugar-coated roses. And we would want to apologize for any mistakes we have made.

5.3 Future Works

The following is a list of features and suggestions for future improvements to our product:

- Refactor the whole codebase.
- Add U1 U2 and U3 gates.
- Able to flip a part of circuit horizontally, Duplicate to the left and right
- Able to undo and redo.
- Able to select some part of a circuit and create a gate from it.
- Add a delete area for gates.
- Able to group parts a circuit.
- Able to move selected gates.
- Able to Insert a time step between gates.
- Easier qubit manipulation.
- Include a manual.
- Improve user experience.
- Able to store data to API server.
- Improve connection time to API server.
- Add jobs management. (cancel queue, continue queue, history result)
- Able to check the status of IBMQ systems.
- Able to use specific seed for transpilation.

REFERENCES

1. Alwin Zulehner, Alexandru Paler, and Robert Wille, 2019, "An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures," **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, vol. 38, no. 7, pp. 1226–1236, July 2019.

2. Robert Wille, Lukas Burgholzer, and Alwin Zulehner, 2019, "Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations," in **Proceedings of the 56th Annual Design Automation Conference 2019**, Las Vegas NV USA, June 2019, pp. 1–6, ACM.

3. Stefan Hillmich, Alwin Zulehner, and Robert Wille, 2021, "Exploiting Quantum Teleportation in Quantum Circuit Mapping," in **Proceedings of the 26th Asia and South Pacific Design Automation Conference**, Tokyo Japan, Jan. 2021, pp. 792–797, ACM.

APPENDIX A

QISKIT TRANSPILER BENCHMARKING

Qiskit Transpiler Benchmarking

A.1 Introduction

Because we are working with IBMQ, which has multiple architectures and each with its environment, even if we input the same circuit, the output may not be the same. In addition, many circuits cannot be operated directly on the machine; they must first be transpiled to convert the gates to those that are compatible with that machine.

The transpiler can be configured in a variety of ways, such as routing technique and optimization level, and each setting impacts the output in different ways, resulting in distinct types of circuit output that can run on different machines. So, with each configuration of a transpiler, we want to know which configuration is better for transpile a circuit.

A.2 Related Research

Today's IBMQ architecture includes a plethora of systems to be chosen, including Brooklyn, Montreal, Armonk, and others. Qubits, QV, CLOPS, processor type, and basis gates are all different aspects of each design. QV stands for "Quantum Volume," and in a nutshell, it is the metric that represents the machine's quality. QV is calculated by running random basic circuits in every possible qubit with a depth equal to the circuit's depth and calculating the probability of heavy circuit, then choosing the most qubit with a probability of heavy circuit above the threshold. CLOPS stands for "Circuit layer operations per second," and it is a statistic that represents the machine's speed. In other words, CLOPS is the number of circuit layers of a QV circuit that operate in one second.

The CNOT Error, Readout Error, T1, and T2 error probabilities for each machine in the IBMQ system refer to CNOT Error, Readout Error, T1, and T2. When the control is $|1\rangle$, the CNOT gate does not perform x gate, resulting in a CNOT Error. T1 is the relaxation period, which is how long $|1\rangle$ will relax to $|0\rangle$ before recording $|1\rangle$. T2 is the dephasing time, which is how long it will take for $|+\rangle$ to dephase to $|-\rangle$ and record $|+\rangle$.

A.3 Approach

We use the knowledge that each machine has gates' errors by introduce an accumulated error metric. The metric calculates a possibility of the output being wrong using average error in each of the base circuit's operations. We calculate accumulated mistakes for each circuit by multiplying non-error probabilities together and subtracting from 1 to yield a rough error as shown in Equation A.1. By using this equation to represent the quality of the transpile output, we will be able to assess the rough accumulated error from each circuit.

$$\mathbb{P}\left(\text{Error}_{\text{accumulate}}\right) \approx 1 - \prod_{gate} \left(1 - \mathbb{P}\left(\text{Error}_{\text{gate}}\right)\right)^{n_{\text{gate}}} ; gate \in \{\text{CX}, \text{ID}, \text{RZ}, \text{SX}, \text{X}\} \tag{A.1}$$

In the benchmarking, we brute-force the circuits from the research "An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures[1]" through every configuration of a transpiler. We picked `fake_guadalupe`, `fake_sydney`, and `fake_montreal` as our fake providers. We chose `fake_guadalupe` since the research circuits are in 16-qubit format, which it can handle. We also want to test providers with more than 16 qubits, so we choose `fake_sydney` and `fake_montreal`, both of which have 27 qubits. Following that, we keep note of each accumulated error to see which configuration has the lowest error. There may be many perfect configurations for each circuit, which will be counted as well.

A.4 Result

After we have executed the circuits on three fake providers (`fake_guadalupe`, `fake_montreal`, and `fake_sydney`), as shown in Table A.1, Table A.2, and Table A.3, we can clearly see that the dominated configuration is "`noise_adaptive`" for the layout method and "`stochastic`" for the routing method with any optimization level or scheduling method.

Table A.1: Benchmarking Result on `fake_guadalupe`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
2	noise_adaptive	stochastic	as_late_as_possible	53
0	noise_adaptive	stochastic	None	51
2	noise_adaptive	stochastic	as_soon_as_possible	48
3	noise_adaptive	stochastic	as_soon_as_possible	48
1	noise_adaptive	stochastic	as_late_as_possible	48
3	noise_adaptive	stochastic	as_late_as_possible	45
1	noise_adaptive	stochastic	None	45
0	noise_adaptive	stochastic	as_late_as_possible	44
2	noise_adaptive	stochastic	None	44
1	noise_adaptive	stochastic	as_soon_as_possible	43
2	noise_adaptive	basic	as_soon_as_possible	42
1	noise_adaptive	basic	None	42
1	noise_adaptive	basic	as_late_as_possible	42
1	noise_adaptive	basic	as_soon_as_possible	42
0	noise_adaptive	stochastic	as_soon_as_possible	42
2	noise_adaptive	basic	as_late_as_possible	42
2	noise_adaptive	basic	None	42
0	noise_adaptive	basic	as_soon_as_possible	42
0	noise_adaptive	basic	as_late_as_possible	42
3	noise_adaptive	basic	None	42
3	noise_adaptive	basic	as_late_as_possible	42
3	noise_adaptive	basic	as_soon_as_possible	42
0	noise_adaptive	basic	None	42
3	noise_adaptive	stochastic	None	41
1	trivial	stochastic	as_late_as_possible	28
3	trivial	stochastic	as_late_as_possible	24
1	trivial	basic	as_soon_as_possible	23
1	trivial	basic	as_late_as_possible	23
2	trivial	basic	as_soon_as_possible	23
1	trivial	basic	None	23
2	trivial	basic	None	23
2	trivial	basic	as_late_as_possible	23
0	trivial	basic	as_soon_as_possible	23
0	trivial	basic	as_late_as_possible	23
0	trivial	basic	None	23
3	trivial	basic	None	23
3	trivial	basic	as_late_as_possible	23
3	trivial	basic	as_soon_as_possible	23
0	trivial	stochastic	as_late_as_possible	21
2	trivial	stochastic	as_late_as_possible	21
1	trivial	stochastic	None	21

Table A.1: Benchmarking Result on `fake_guadalupe`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
1	trivial	stochastic	as_soon_as_possible	21
3	trivial	stochastic	as_soon_as_possible	20
3	trivial	stochastic	None	20
2	trivial	stochastic	as_soon_as_possible	20
2	trivial	stochastic	None	19
0	trivial	stochastic	None	19
0	trivial	stochastic	as_soon_as_possible	18
2	dense	stochastic	as_late_as_possible	17
0	dense	basic	as_late_as_possible	16
0	dense	basic	as_soon_as_possible	16
0	dense	stochastic	None	16
2	dense	basic	as_late_as_possible	16
2	dense	basic	None	16
1	dense	basic	None	16
1	dense	stochastic	as_late_as_possible	16
2	dense	basic	as_soon_as_possible	16
1	dense	stochastic	as_soon_as_possible	16
0	dense	basic	None	16
1	dense	basic	as_soon_as_possible	16
1	dense	basic	as_late_as_possible	16
3	dense	basic	None	15
3	dense	basic	as_late_as_possible	15
3	dense	basic	as_soon_as_possible	15
3	dense	stochastic	None	15
2	dense	stochastic	as_soon_as_possible	15
1	dense	stochastic	None	15
0	dense	stochastic	as_soon_as_possible	15
0	dense	stochastic	as_late_as_possible	15
2	dense	stochastic	None	15
3	dense	stochastic	as_soon_as_possible	14
3	dense	stochastic	as_late_as_possible	14

Table A.2: Benchmarking Result on `fake_sydney`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
0	noise_adaptive	stochastic	as_late_as_possible	50
2	trivial	basic	as_soon_as_possible	49
0	trivial	basic	None	49
1	trivial	basic	None	49
3	trivial	basic	as_soon_as_possible	49
3	trivial	basic	as_late_as_possible	49
3	trivial	basic	None	49
1	trivial	basic	as_late_as_possible	49
1	trivial	basic	as_soon_as_possible	49
0	trivial	basic	as_soon_as_possible	49
2	trivial	basic	None	49

Table A.2: Benchmarking Result on `fake_sydney`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
2	trivial	basic	as_late_as_possible	49
2	trivial	stochastic	None	49
0	trivial	basic	as_late_as_possible	49
2	noise_adaptive	stochastic	as_late_as_possible	47
1	noise_adaptive	stochastic	as_soon_as_possible	47
0	noise_adaptive	stochastic	as_soon_as_possible	47
3	trivial	stochastic	None	47
1	trivial	stochastic	as_late_as_possible	46
1	noise_adaptive	stochastic	None	46
2	noise_adaptive	stochastic	None	45
3	noise_adaptive	stochastic	None	45
3	noise_adaptive	stochastic	as_late_as_possible	44
0	trivial	stochastic	as_soon_as_possible	44
0	noise_adaptive	stochastic	None	44
0	trivial	stochastic	None	44
0	trivial	stochastic	as_late_as_possible	44
1	noise_adaptive	stochastic	as_late_as_possible	44
3	trivial	stochastic	as_soon_as_possible	44
3	trivial	stochastic	as_late_as_possible	43
3	noise_adaptive	stochastic	as_soon_as_possible	43
2	trivial	stochastic	as_soon_as_possible	43
2	noise_adaptive	stochastic	as_soon_as_possible	42
3	dense	stochastic	as_soon_as_possible	42
1	trivial	stochastic	None	42
2	trivial	stochastic	as_late_as_possible	42
0	noise_adaptive	basic	None	41
0	noise_adaptive	basic	as_late_as_possible	41
0	noise_adaptive	basic	as_soon_as_possible	41
1	dense	stochastic	None	41
1	noise_adaptive	basic	None	41
1	noise_adaptive	basic	as_late_as_possible	41
1	noise_adaptive	basic	as_soon_as_possible	41
1	trivial	stochastic	as_soon_as_possible	41
2	noise_adaptive	basic	None	41
3	noise_adaptive	basic	as_late_as_possible	41
3	noise_adaptive	basic	as_soon_as_possible	41
2	noise_adaptive	basic	as_soon_as_possible	41
2	noise_adaptive	basic	as_late_as_possible	41
3	noise_adaptive	basic	None	41
2	dense	stochastic	None	40
2	dense	stochastic	as_late_as_possible	40
0	dense	stochastic	None	40
2	dense	stochastic	as_soon_as_possible	40
0	dense	stochastic	as_late_as_possible	39
3	dense	stochastic	None	39
1	dense	stochastic	as_late_as_possible	39
1	dense	stochastic	as_soon_as_possible	39

Table A.2: Benchmarking Result on `fake_sydney`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
0	dense	basic	as_soon_as_possible	38
0	dense	stochastic	as_soon_as_possible	38
0	dense	basic	as_late_as_possible	38
2	dense	basic	None	38
3	dense	basic	None	38
3	dense	stochastic	as_late_as_possible	38
3	dense	basic	as_soon_as_possible	38
3	dense	basic	as_late_as_possible	38
2	dense	basic	as_late_as_possible	38
1	dense	basic	None	38
1	dense	basic	as_late_as_possible	38
1	dense	basic	as_soon_as_possible	38
2	dense	basic	as_soon_as_possible	38
0	dense	basic	None	38

Table A.3: Benchmarking Result on `fake_montreal`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
3	noise_adaptive	stochastic	None	59
0	noise_adaptive	stochastic	as_late_as_possible	56
2	noise_adaptive	stochastic	as_soon_as_possible	56
0	noise_adaptive	stochastic	None	54
0	noise_adaptive	stochastic	as_soon_as_possible	53
3	noise_adaptive	stochastic	as_soon_as_possible	52
2	noise_adaptive	stochastic	None	52
1	noise_adaptive	stochastic	as_soon_as_possible	52
1	noise_adaptive	stochastic	None	52
2	noise_adaptive	stochastic	as_late_as_possible	50
3	noise_adaptive	stochastic	as_late_as_possible	49
1	noise_adaptive	stochastic	as_late_as_possible	48
0	noise_adaptive	basic	as_late_as_possible	46
2	noise_adaptive	basic	None	46
2	noise_adaptive	basic	as_late_as_possible	46
2	noise_adaptive	basic	as_soon_as_possible	46
0	noise_adaptive	basic	as_soon_as_possible	46
0	noise_adaptive	basic	None	46
3	noise_adaptive	basic	None	46
1	noise_adaptive	basic	None	46
3	noise_adaptive	basic	as_soon_as_possible	46
3	noise_adaptive	basic	as_late_as_possible	46
1	noise_adaptive	basic	as_late_as_possible	46
1	noise_adaptive	basic	as_soon_as_possible	46
2	trivial	basic	as_late_as_possible	41
2	trivial	basic	None	41
2	trivial	basic	as_soon_as_possible	41
1	trivial	basic	None	41

Table A.3: Benchmarking Result on `fake_montreal`

Optimization Level	Layout Method	Routing Method	Scheduling Method	Count
1	trivial	basic	as_soon_as_possible	41
3	trivial	basic	None	41
3	trivial	basic	as_late_as_possible	41
3	trivial	basic	as_soon_as_possible	41
1	trivial	basic	as_late_as_possible	41
0	trivial	basic	None	41
0	trivial	basic	as_late_as_possible	41
0	trivial	basic	as_soon_as_possible	41
3	trivial	stochastic	as_soon_as_possible	39
1	trivial	stochastic	as_late_as_possible	37
1	trivial	stochastic	None	37
1	dense	stochastic	None	37
2	trivial	stochastic	None	37
3	trivial	stochastic	None	37
1	dense	stochastic	as_soon_as_possible	37
0	trivial	stochastic	None	36
2	dense	stochastic	None	36
3	dense	stochastic	None	36
0	dense	stochastic	None	36
1	dense	stochastic	as_late_as_possible	36
0	dense	stochastic	as_soon_as_possible	35
0	dense	stochastic	as_late_as_possible	35
1	trivial	stochastic	as_soon_as_possible	35
2	dense	stochastic	as_soon_as_possible	35
2	trivial	stochastic	as_soon_as_possible	35
3	dense	stochastic	as_soon_as_possible	35
3	trivial	stochastic	as_late_as_possible	35
3	dense	stochastic	as_late_as_possible	35
2	dense	stochastic	as_late_as_possible	35
2	trivial	stochastic	as_late_as_possible	35
0	dense	basic	as_late_as_possible	34
0	dense	basic	as_soon_as_possible	34
3	dense	basic	as_soon_as_possible	34
3	dense	basic	as_late_as_possible	34
3	dense	basic	None	34
1	dense	basic	as_soon_as_possible	34
1	dense	basic	as_late_as_possible	34
2	dense	basic	as_soon_as_possible	34
2	dense	basic	as_late_as_possible	34
2	dense	basic	None	34
0	trivial	stochastic	as_late_as_possible	34
0	trivial	stochastic	as_soon_as_possible	34
1	dense	basic	None	34
0	dense	basic	None	34

A.5 Discussion

We can clearly see why the ”`noise_adaptive`” for the layout method and ”`stochastic`” for the routing method is the best: it tries to apply the operations on the lowest error qubits available, resulting in the lowest accumulated error, which makes the circuit more dependable. We will use this conclusion to our recommendation system for recommending the most optimal circuit to the user, and use it on the target provider.

APPENDIX B

USER USABILITY TEST

User Usability Test

B.1 Testing Metrics

- M1: Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation.
- M2: Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit.
- M3: Users can alter the compiler and optimization level as well as see the transpilation output that will be executed on the IBMQ system.
- M4: Users can get IBMQ system recommendations from the application before executing the circuit on the hardware.
- M5: Users can see a brief description of the gate when hovering over it.
- M6: Users will receive instant and relevant feedback from the user interface.
- M7: Users can compile the circuit into OpenQASM or Qiskit.

B.2 Test Cases Matrix for IBM Quantum Composer and Quantum Programming Studio

Table B.4 Test Cases Matrix for IBM Quantum Composer and Quantum Programming Studio

Metric ID	Metric	IBM Quantum Composer	Quantum Programming Studio
M1: Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation			
M1.1	Create circuit using fundamental gates	Create a circuit from gates	Create a circuit from gates
M1.2	Create custom gate using matrix	N/A	N/A
M1.3	Create custom gate using rotation	Use a rotation gate to manipulate probability	Use a rotation gate to manipulate probability
M2: Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit			
M2.1	Group circuit together	Select gates, right click, and click “Group”	N/A
M2.2	Copy and paste a part of circuit into the other part of the circuit	Copy and paste the group	N/A
M3: Users can alter the compiler and optimization level as well as see the transpilation output that will be executed on the IBMQ system			
M3.1	Change transpiler options	N/A	N/A
M3.2	Change optimization level	N/A	N/A
M3.3	See transpilation output	N/A	N/A
M4: Users can get IBMQ system recommendations from the application before executing the circuit on the hardware			
M4.1	See IBMQ system recommendation before running the circuit	N/A	N/A
M5: Users can see a brief description of the gate when hovering over it			
M5.1	See the gate information	N/A	N/A
M7: Users can compile the circuit into OpenQASM or Qiskit			
M7.1	Get OpenQASM output	Locate OpenQASM output in code editor	Locate OpenQASM output in edit code tab
M7.2	Get Qiskit output	Locate Qiskit output in code editor	Locate Qiskit output in export tab

B.3 Test Cases Matrix for Quirk and Quantographer

Table B.5 Test Cases Matrix for Quirk and Quantographer

Metric ID	Metric	Quirk	Quantographer
M1: Users can build circuits using fundamental gates, as well as generate new gates using a matrix or rotation			
M1.1	Create circuit using fundamental gates	Create a circuit from gates	Create a circuit from gates
M1.2	Create custom gate using matrix	Create new gate from a matrix	Create new gate from a matrix
M1.3	Create custom gate using rotation	Use a rotation gate to manipulate probability	Use a rotation gate to manipulate probability
M2: Users can group quantum gates together and reuse them by copying and pasting them into other areas of the circuit			
M2.1	Group circuit together	Using create gate feature to create new gate from current circuit	N/A
M2.2	Copy and paste a part of circuit into the other part of the circuit	N/A	Copy and paste part of circuit
M3: Users can alter the compiler and optimization level as well as see the transpilation output that will be executed on the IBMQ system			
M3.1	Change transpiler options	N/A	Change in execute dialog
M3.2	Change optimization level	N/A	Change in execute dialog
M3.3	See transpilation output	N/A	Locate transpilation output in execute dialog
M4: Users can get IBMQ system recommendations from the application before executing the circuit on the hardware			
M4.1	See IBMQ system recommendation before running the circuit	N/A	Click ”Get Recommendation” button in execute dialog
M5: Users can see a brief description of the gate when hovering over it			
M5.1	See the gate information	Hover over the gates	Hover over the gates
M7: Users can compile the circuit into OpenQASM or Qiskit			
M7.1	Get OpenQASM output	N/A	Locate OpenQASM output in export dialog
M7.2	Get Qiskit output	N/A	Locate Qiskit output in code editor

Note: “M6: Users will receive instant and relevant feedback from the user interface” will be measured by questioning after the test has been conducted.

B.4 Moderator Code of Conduct

- Before the test is conducted, the moderator must ask these following questions:
  - How did you know about quantum circuit? (From class, self-learning, etc.)
  - How long have you been studying or working with quantum circuit? (Since when they have learned)
  - How often do you have to interact with quantum circuit? (Any measurements, days, weeks, etc.)
  - Are you currently using a quantum circuit editor? If yes, what product?
  - Please describe your experience with your tool.
  - On a scale of 1 to 5 (1 = Not at all confident, 5 = Very confident), how would you rate your level of confidence in using a quantum circuit editor?
- Before each scenario, the moderator must do the following:
  - State the name of the scenario
  - State the instructions
  - Ask if participant is ready
- While a scenario is performed, the moderator must:
  - Note the interaction while the participant is performing a task.
  - Remain silent observing the participant unless notified or the scenario is finished.
  - The moderator must think about actions happening while observing:
    * Why does the participant go there? What are they thinking as they do that?
    * What is the cause that make the participant do the action?
    * Why is this task taking a long time?
- After each scenario, the moderator must ask these questions:
  - How do you feel right now after the scenario?
  - What is the easiest thing to do in this scenario?
  - What is the hardest thing to do in this scenario?
  - What, if anything, surprised you in this scenario?
  - What, if anything, caused you frustration?
  - I saw you do ___/go to ___. What were you thinking at that point?
  - If you were looking for ___, where would you expect to find it?
- After the test is conducted, the moderator must ask these questions:
  - What was your overall impression of these editors?
  - Which of the editors do you favor the most? Please rank from most desired to least liked.
  - What are the things you will not get from these editors?

B.5 Scenarios

B.5.1 Scenario A: Qubit rotation circuit

- Metrics: M1.1 M1.3 M5.1 M7.1 M7.2
- Tested product: IBM Quantum Composer, Quantum Programming Studio, Quirk, Quantographer
- Instructions
  1. Create the circuit as the diagram in Figure B.1.
  2. Locate the location of QASM and Qiskit code.
  3. If you have any questions, please notify the moderator.

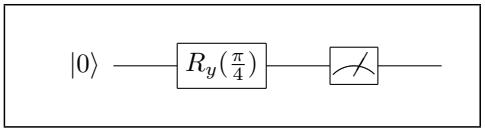

Figure B.1 Qubit rotation circuit

B.5.2 Scenario B: 3-qubit Grover search circuit

- Metrics: M1.1 M2.1 M2.2 M5.1
- Tested product: IBM Quantum Composer, Quirk, Quantographer
- Instructions
  1. Create the circuit as the diagram in Figure B.2. You must create the second amplifier from your first amplifier by these methods:
     - If you are using IBM Quantum Composer, you must use ”Group” feature to create the second amplifier.
     - If you are using Quirk, you must use ”Make Gate” feature to create the second amplifier.
     - If you are using Quantographer, you must copy the first amplifier and paste it as the second one.
  2. If you have any questions, please notify the moderator.

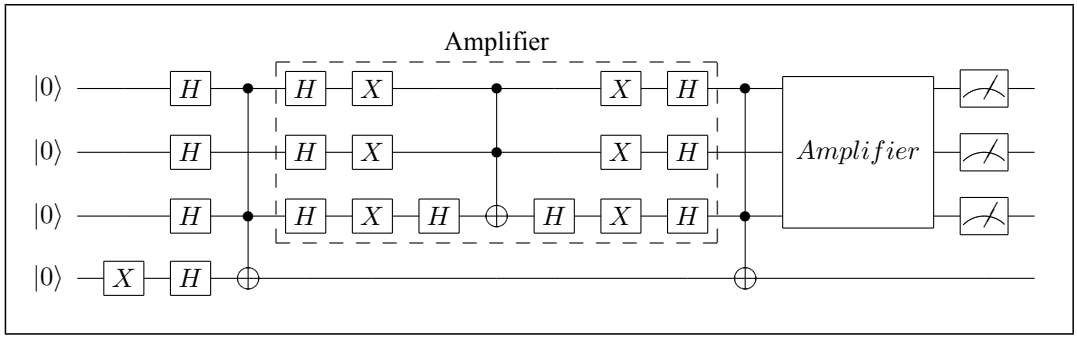

Figure B.2 3-qubit Grover search circuit

B.5.3 Scenario C: Qubit operation from a matrix

- Metrics: M1.1 M1.2 M5.1
- Tested product: Quirk, Quantographer
- Instructions
  1. Create the circuit as the diagram in Figure B.3.
  2. If you have any questions, please notify the moderator.

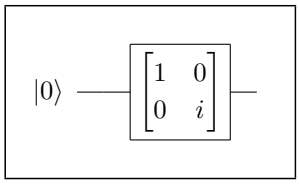

Figure B.3 Qubit operation from a matrix

B.5.4 Scenario D: Quantum teleportation circuit

- Metrics: M1.1 M5.1
- Tested product: IBM Quantum Composer, Quantum Programming Studio, Quirk, Quantographer
- Instructions
  1. Create the circuit as the diagram in Figure B.4.
  2. If you have any questions, please notify the moderator.

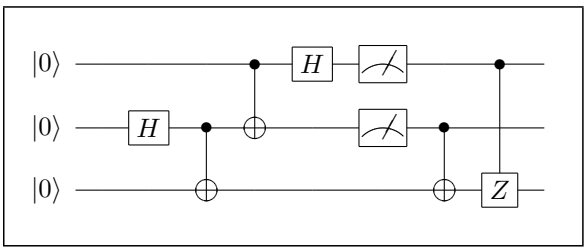

Figure B.4 Quantum teleportation circuit

B.5.5 Scenario E: Transpile a quantum circuit

- Metrics: M1.1 M3.1 M3.2 M3.3 M4.1 M5.1
- Tested product: Quantographer
- Instructions
  1. Create the circuit as the diagram in Figure B.5.
  2. Connect to IBMQ by using the key provided by the moderator.
  3. Locate a transpilation output.

4. Get a system recommendation and use the recommended configuration.
5. Re-transpile the configuration.
6. Change the configuration to:
   - System: `ibmq_bogota`
   - Optimization Level: `3`
   - Layout Method: `Dense`
   - Routing Method: `Lookahead`
   - Schedule Method: `As late as possible`
7. Re-transpile the configuration.
8. If you have any questions, please notify the moderator.

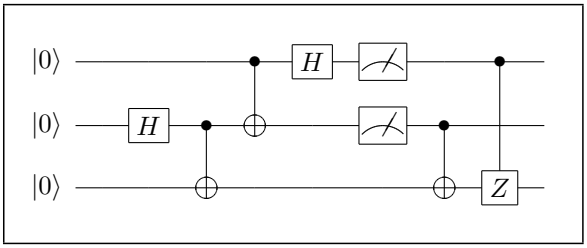

Figure B.5 Quantum teleportation circuit

B.6 Result

B.6.1 Participant Demographics

A total of 8 people took part in our study. The majority of them have studied quantum circuits as a relevant subject, as presented in Figure B.6. Over half of them have 1–2 years of experience. However, as shown in Figures B.7 and B.8, the majority of them seldom interact with a quantum circuit. Because of its simplicity of use, half of the participants opted to utilize a quantum circuit editor like IBM Quantum Composer or Quirk (see Figure B.9). Another half, on the other hand, favored Python because of its convenience in development. When questioned about their confidence in utilizing a quantum circuit editor, the participants gave an average score of 2.5 out of 5, as shown in Figure B.10.

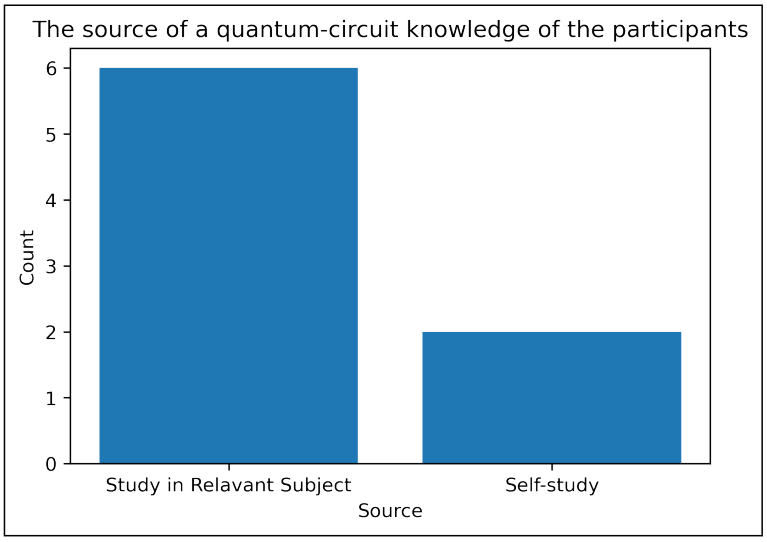

Figure B.6 The source of a quantum-circuit knowledge of the participants

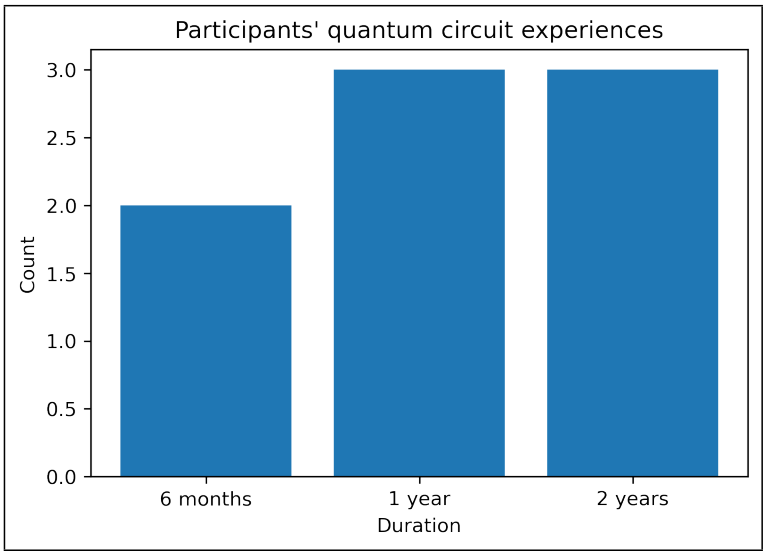

Figure B.7 Participants' quantum circuit experiences

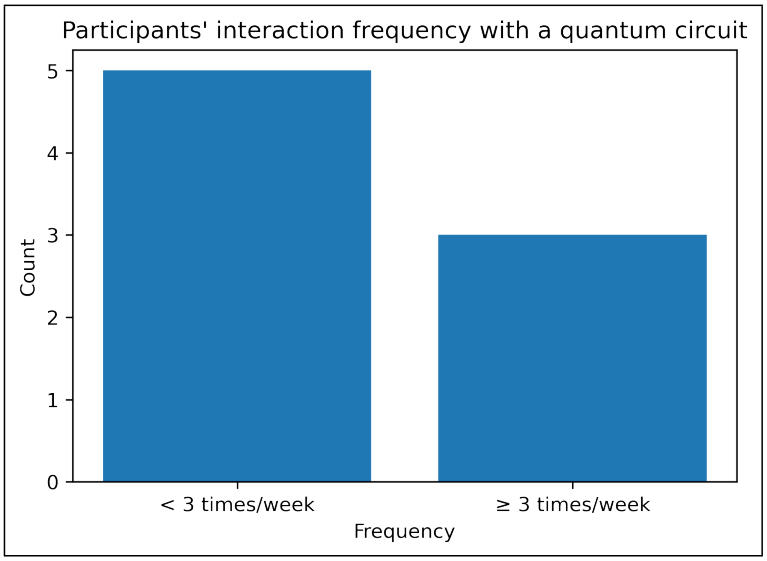

Figure B.8 Participants' interaction frequency with a quantum circuit

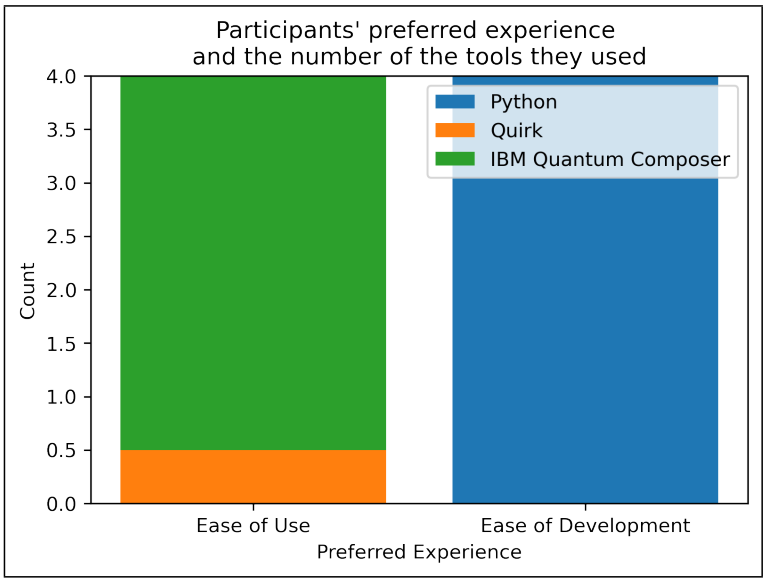

Figure B.9 Participants' preferred experience and the number of the tools they used. Please note that one participant used both the IBM Quantum Composer and Quirk. As a result, both preferences will be at half points.

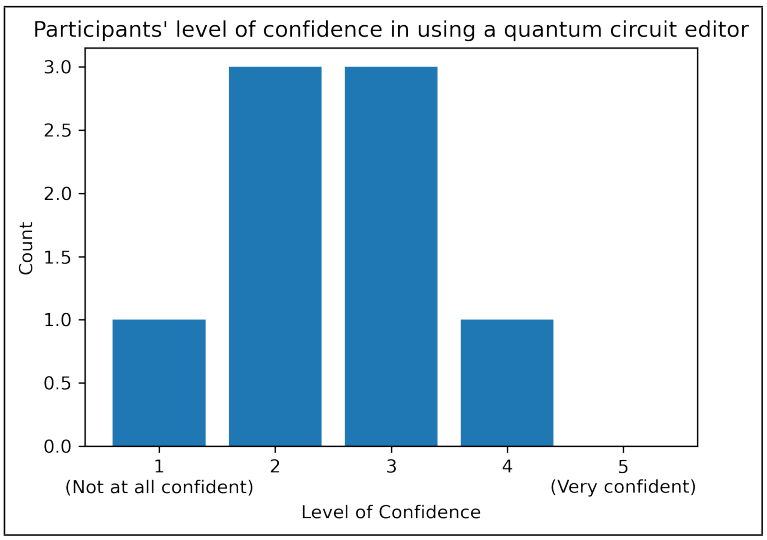

Figure B.10 Participants' level of confidence in using a quantum circuit editor

B.6.2 Participant Feedback

Table B.6 shows that when participants were asked for input on IBM Quantum Composer, the majority of them claimed that novice users could have problems figuring out how to adjust the angle of the RY gates (which can be done by double-clicking on the gate or right click on the gate and select "Edit Gate" option). Furthermore, more than half of the participants expressed their confusion about how a control gate modifier works, dissatisfaction with how the gates are oriented to the left, and dissatisfaction with how difficult it is to manage a barrier gate, which should be easy for this basic function.

According to Table B.7, which is participants' feedback on Quantum Programming Studio, most people do not understand that while altering the angle of the RY gate, the theta in the right plane (which presents the information of the presently chosen gate) is modifiable. This is due to the fact that the editor displays it as a regular text rather than an input. Another significant observation is that the participants are unaware that $\oplus$ is a controlled-X gate. It is because the $\oplus$ sign is commonly used to represent a Pauli X gate.

In the Table B.8, which contains comments from participants on Quirk — a quantum circuit simulator. The majority of them appreciate how the editor arranges gates in a grid layout, how gates are organized into relevant categories, and how tooltips give them a vast amount of information. However, a lot of them also complain about the number of gates, claiming that there are just too many of them, making it difficult to locate the gate they desire. Making a gate from a circuit is also an unpleasant and puzzling process, therefore, they prefer not to use it as a substitute for grouping circuits together.

Finally, Table B.9 comprises all of the participant feedback on Quantographer, our product. Despite receiving a lot of favorable reviews, such as "The application is really smooth." or "The layout of the application is very well designed.", the program nevertheless has some severe flaws. The most prevalent issue is that it is difficult to locate the QASM code, which is tucked away in the export dialog. Another aspect that participants desire is the ability to undo and redo actions. Due to half-baked features (due to time constraints), editing the circuit might be challenging at times. It will be easier to complete tasks if the user is able to undo and redo the editing.

Participants were asked to order the editors from the most preferred to the least one. We then score each of them in points, with the most preferred one being 4 points, to the least one for 1 point. After that, we add up the points to see the overall order. As indicated in Table B.10, the participants preferred Quantographer the

most with the score of 28 points. Followed by IBM Quantum Composer with 24 points, Quirk for 19 points, and Quantum Programming Studio, which appears to be the least popular one, with 10 points.

Table B.6: Participant Feedback on IBM Quantum Composer

Participant feedback	**Count**
New users might not know about double-clicking	6
I'm confused about how a control gate modifier works	5
I don't like when the circuit get aligned to the left	5
A barrier should be able to drag to expand across qubits	5
There should be something informing you that you can append a control gate modifier to the gate	3
It's a bit confusing when making CZ gate	3
I don't know that grouping is possible	3
I like how grouping work	3
I like how easy the drag-and-drop system is	3
It's laggy	3
New users might have problems with the gate alignment	2
I'm confused about how custom gate naming work	2
The bounding box of the editing area is too small for selecting gates	2
It should be possible to drag the control gate modifier to the line and choose the target gate	1
Long custom gates names get squashed	1
Shift-clicking should be able to select a range of gates	1
I don't know that you can select gates by dragging	1
New users might get overwhelmed at first	1
I don't like dragging a gate for a long distance	1
I want more information about the gates	1
I like how gates are getting swapped	1
I want a user tutorial	1
It should be able to reverse/flip a group of gate	1
There should be a trash can to drag to delete a gate	1
I want more control with classical bits	1
The coding plane is useful	1

Table B.7: Participant Feedback on Quantum Programming Studio

Participant feedback	**Count**
I don't know that "theta" in the right plane is editable	7
I don't know that $\oplus$ is a controlled-X gate	4
It's easy to use a control gate modifier	3
I want some gate tooltips	3
The Bloch sphere is placed directly on the line which the line is continued, so I thought that I have to place the gates after it.	3
I don't know how to add more qubits	3
I like how the editor align gates into the grid	3
I don't know where the Qiskit code is located	3
I don't know how to input π	2
I thought that $\oplus$ is X gate	2
The gate should be categorized into groups	2
There should be some color in the editors	2

There should be some hint about how to add more qubits	2
There should be a plus button below the last qubit to add more qubits	2
It's hard to find things when the GUI is plain white	2
I want shortcut keys	2
I like that you don't have to double click on the RY gate to change its value	1
Gate's parameter should be able to be edited in the editing area	1
I'm confused about how to use a control gate modifier	1
I prefer dragging a control gate modifier into a gate then this	1
Gate information plane is not useful	1
I like how easy the drag-and-drop system is	1
I don't like a Bloch sphere after the circuit	1
I don't like how gates move when get swapped	1
It's fast	1
I want more visualization	1
the GUI is not appealing	1
The Qiskit code should be available in the code editor directly	1
I like how the code and the GUI editor is separated	1
I like how you can export into different languages	1
I wish you can edit more than QASM	1

Table B.8: Participant Feedback on Quirk

Participant feedback	**Count**
I like how the editor align gates into the grid	5
I don't know how to make a gate from a circuit	4
The number of gates made it hard to find specific gate	4
I like how gates are categorized into groups	4
I like how the tooltips provide detailed information	4
I don't understand how to create a custom gate from a matrix.	3
When making a gate from a circuit, it should be able to drag from a preview to select a range of circuits you want	3
Create gate from a circuit should be separated from "Make gate"	2
There should be pre-made controlled-gate sets	2
I don't know how to add more qubits	2
I don't like how much information is provided	2
It should be able to select gates by dragging over the gates	2
It's hard to see a custom gate in the toolboxes	1
I don't know that I can fill the matrix	1
If I never saw "From Matrix" tabs in "Make Gate", I might not know that I can make a gate from a matrix.	1
They should add "Double click to edit gate" so I don't have to delete the gate and make the gate again.	1
The matrix placeholder should be a little bit pale.	1
I like how many ways you can create a custom gate	1
I don't know where will my custom gate will be	1
New users might get confused if they didn't know about the make gate feature	1
Make gate button should be inside the custom gate category	1
There should be a separated box for each matrix members	1
There should be a universal parameterized gate to edit several variables	1

I’m confused about a control gate modifier	1
There should be a plus button below the last qubit to add more qubits	1
This editor looks more suitable for a user with math knowledge	1
I like the animation	1
The toolbars should scroll with the circuit	1
There should be a barrier	1
I wish the gate is moved on swapping without stacking on each other	1
It should be able to measure on another basis	1
I don’t like the font Arial	1
I like how much output it provides	1
The dialog for editing RY gate should be better	1
I want shortcut keys	1
There should be a trash can to drag to delete a gate	1
I don’t like that you have to drag the gate out of the editor to delete	1
The ”Change” button in the RY gate is obvious for users	1

Table B.9: Participant Feedback on Quantographer

Participant Feedback	**Count**
I cannot find where the QASM code is.	7
There should be an undo and redo function.	5
The application is very smooth.	5
Adding and deleting qubit is confusing.	3
Multiple qubit gate should auto-align when placed at bottom qubits.	3
The gates’ hitbox is poorly designed.	3
The gate name should be horizontal since long custom gate name might be difficult to read.	3
I don’t know how to use a control qubit modifier.	3
The application’s layout is very well designed.	3
Barrier is hard to use.	2
Deleting gate is confusing.	2
There should be able to drag gate out of the workspace to delete it.	2
I don’t know that I can drag to select nultiple gates at once.	2
Drag to select is hard to use.	2
I like the execute dialog.	2
There should be an option for exporting Qiskit too.	1
It should be able to see the QASM code from code panel.	1
Copying and pasting is easy.	1
Pasting copied selection is not appending to the circuit.	1
Changing parameter is easy.	1
I don’t know if it’s already copied since the contextmenu didn’t disappear.	1
It should tell me if I place in a wrong spot.	1
There should be a shortcut key for copying and pasting.	1
Multiple qubit gate should add qubits when placing on the last qubit.	1
Control qubit should change its color when moving.	1
It should be able to delete a qubit by a context menu.	1
I don’t know to put ”`pi`” or ”`pi.`”	1
I don’t like to delete gate with middle mouse button.	1
Copying and pasting performs poorly.	1

Table B.9: Participant Feedback on Quantographer

Participant Feedback	**Count**
It should append a gate when you drag one over another one.	1
It should be able to delete a gate by double-clicking	1
There should be an auto arrange function.	1
I like how it's organized in grid.	1
It should be able to insert a time step.	1
It should show me silhouette when I drag the gate over the circuit.	1
I like that I can drag to select multiple gates at once.	1
It should be able to see what a custom gate made from and edit it.	1
I like how a matrix for a custom gate is already filled with 0.	1
It should be able to import large matrix.	1
It should be able to import/export user's gate.	1
Adjusting a control qubit modifier is hard.	1
It should be able to delete a control qubit modifier from the context menu.	1
There should be some information about each settings.	1
I don't know that you can transpile a circuit in execute dialog.	1
The "Update Transpilation Result" button should be in other color.	1
Other users might get confused between "Execute" and "Update Transpilation"	1
The amount of information in tooltips is sufficient.	1
The tooltip should show a matrix for a particular gate.	1
The tooltip should show how to use a difficult gate/instruction like a control qubit modifier or a barrier.	1
"Tip of the day" should be in a workspace.	1
It should ask user to save current file when creating a new circuit.	1
The saved file name should be the same as project name.	1
There should be some basic result showing while editing.	1
The interface looks old.	1
Protip is not distinctive enough.	1
Control qubit/bit modifier icon might be hard to read.	1

Table B.10: Participant Quantum Circuit Editor Preferences

Software	**Participant**								**Total**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
IBM Quantum Composer	3	4	4	2	4	2	4	1	24
Quantum Programming Studio	1	2	1	1	1	1	1	2	10
Quirk	2	1	2	3	3	3	2	3	19
Quantographer	4	3	3	4	3	4	3	4	28

APPENDIX C

GRAPHICAL USER INTERFACE DESIGN

Graphical User Interface Design

As we discussed in Chapter 3, our user interface was created using the results of the user usability test described in Appendix 2. We attempted to make our editor as basic as possible in order to improve user friendliness. This part also contains the interface depending on various conditions and use cases.

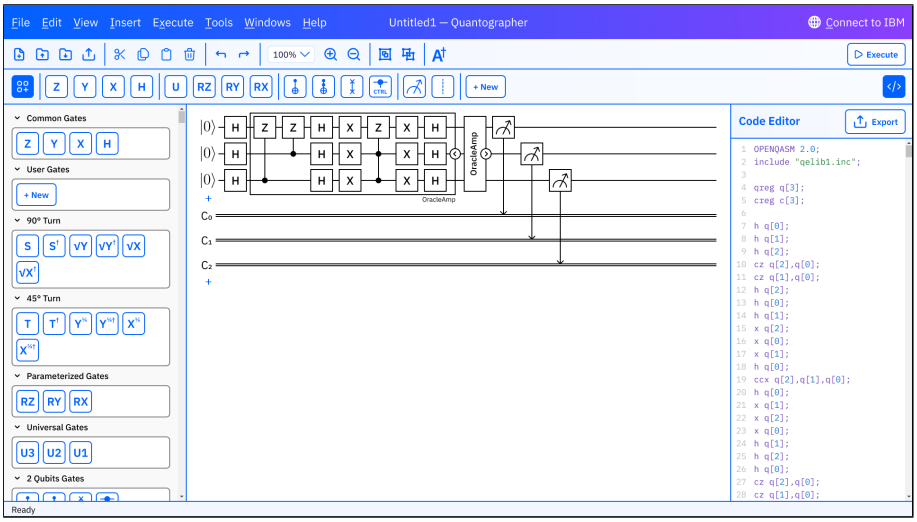

Figure C.1 Main workspace

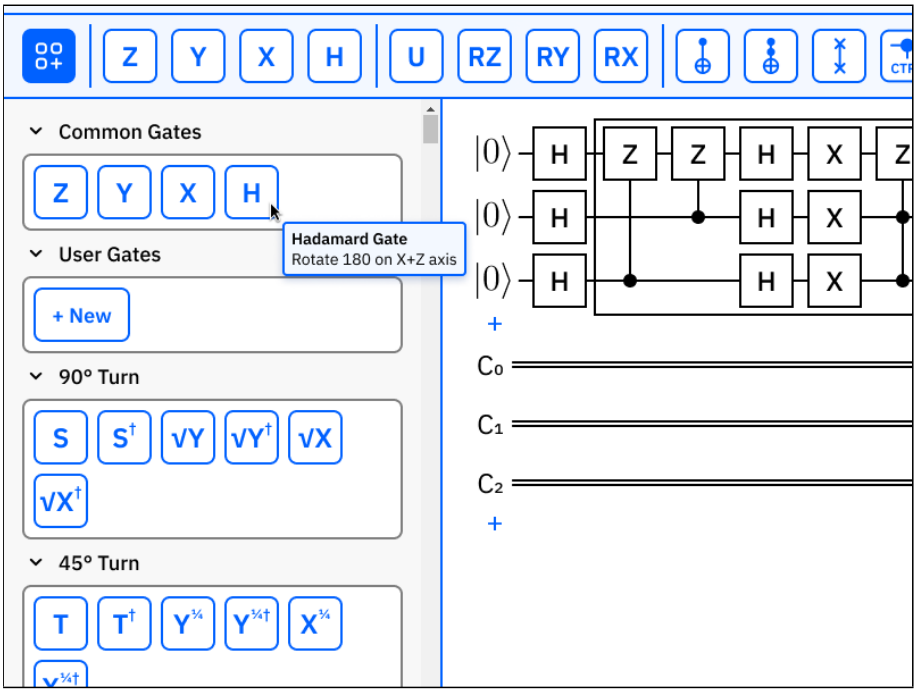

Figure C.2 Gate information on hover

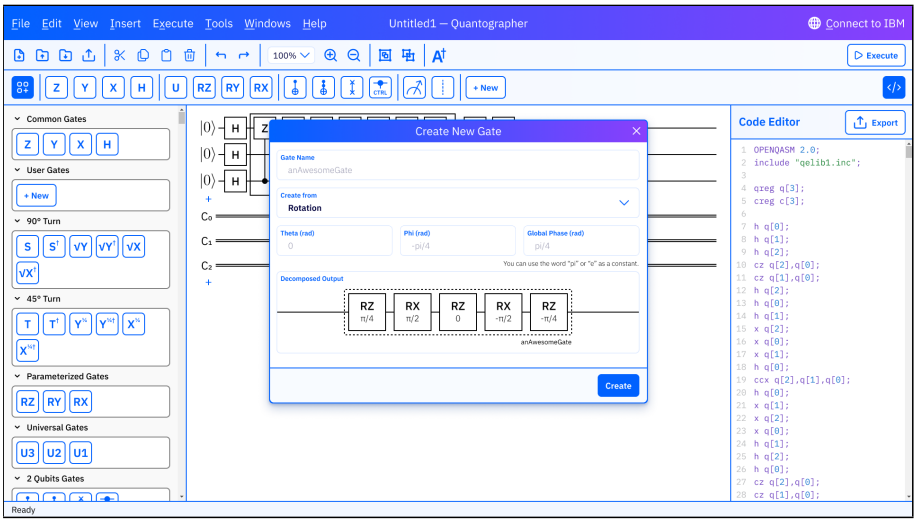

Figure C.3 Create new gate from rotation

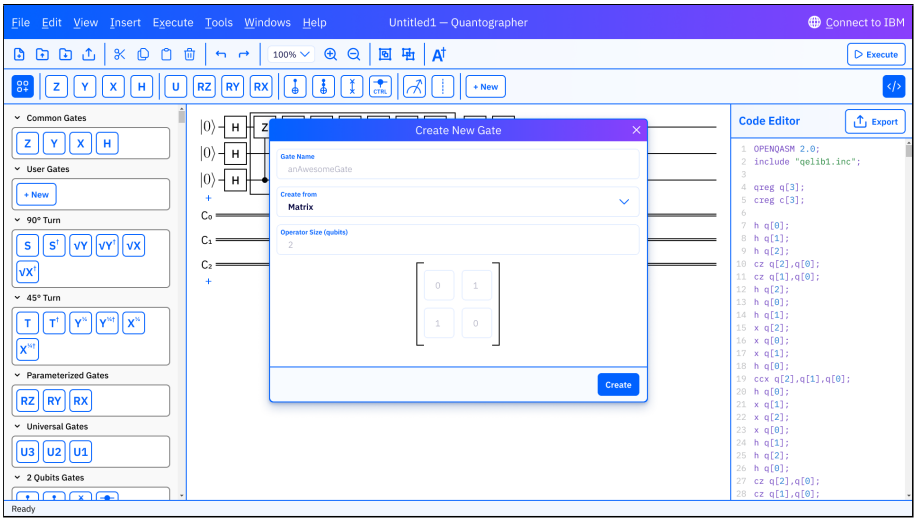

Figure C.4 Create new gate from matrix

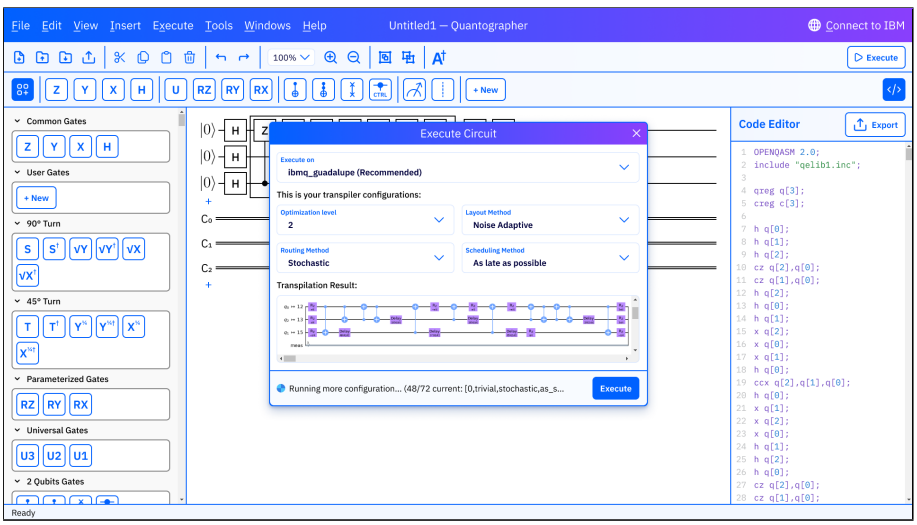

Figure C.5 Execute circuit on IBMQ system

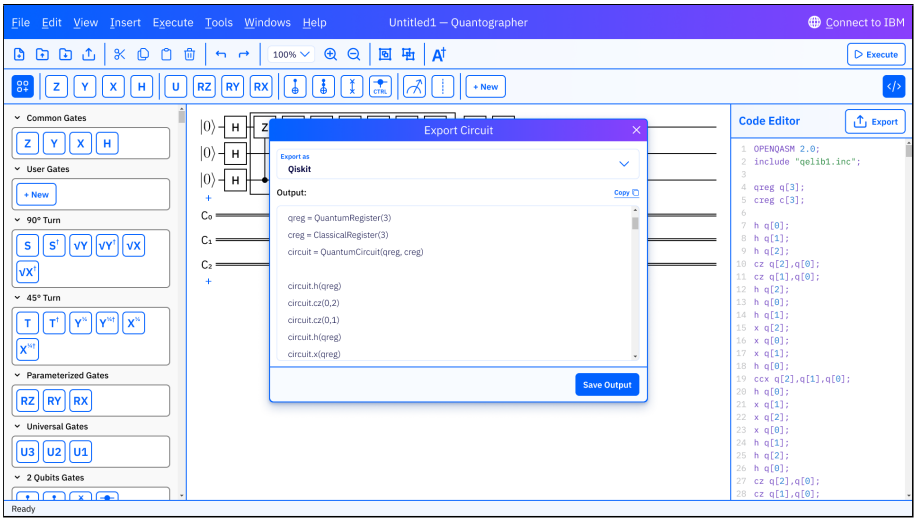

Figure C.6 Export circuit into other format

APPENDIX D

PROTOTYPES

Prototypes

As we discussed in Chapter 3, our user interface was created using the results of the user usability test described in Appendix 2. We attempted to make our editor as basic as possible in order to improve user friendliness. This part also contains the interface depending on various conditions and use cases.
After my friend ended designing our software user interface, I started working on a crucial and central part of the user interface, the Editor. I did some research on the topic of how to create such a highly interactive user interface on a webpage and got 2 answers.
One is to use canvas, and the other one is to use SVG. I tried canvas first because I have some experience with it and have been using it on some of my personal projects. But this time it is different because, unlike other projects that I have used canvas for, I use it for drawing only. This work required drawing that one could interact with. It is a tough task because the canvas can only handle a drawing. I have to handle interaction by myself. Canvas will notify you with the coordinates of the cursor when the user presses mouse buttons, moves the cursor, and releases mouse buttons in the canvas area, and you have to figure out what actions the user intended to do and what elements they are interacting with on that canvas by yourself. I spent two months experimenting with canvas, trying to create an editor that is fully extensible and can add components with ease. I ended up with four prototypes that I cannot continue to work with because the code is such a giant mess that it is too difficult to understand and maintain. I tried to simplify it many times (by creating it from the ground up, keeping only the concept). I re-implemented it and dubbed it ”another prototype,” but I still could not come up with anything better.The difficult part is to create an interactive system that can be easily extensible.

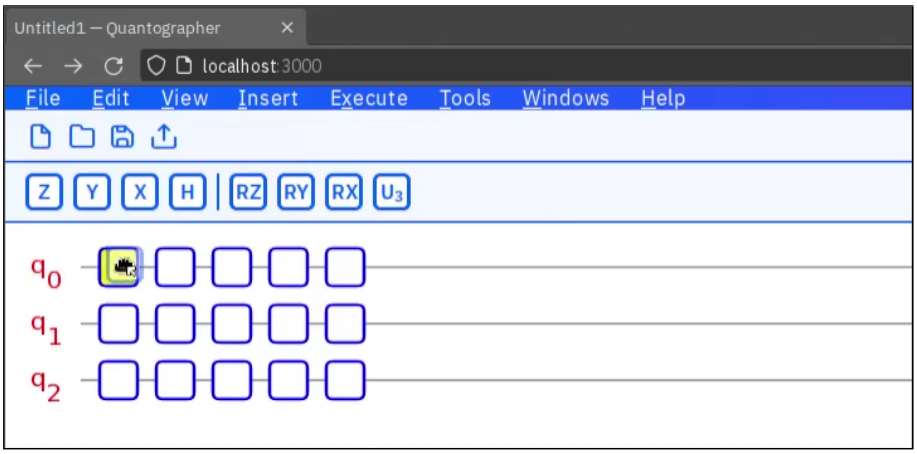

Figure D.1 First prototype

In the Figure D.1, this is my first prototype. It is just a proof of concept to test that I can draw on a canvas and use mouse event to update it real time.
In the Figure D.2, D.3, and D.4. it is a first prototype to achieve a realistic circuit rendering. Moreover, I am experimenting with how control qubits relation lines should look like. Because at that time, I did not like how it is done in Qiskit and I want to achieve more compact representation.
After test with many ways that control qubits and qubit that it control an be aligned. I found that two lines is not totally aligned together as in Figure D.4. So, I create some quadratic equation to calculate a curve's control point to make multiple curve line together. The result is in Figure D.5. And in Figure D.6, I modified it a little so that the curve tilt towards the control qubit.

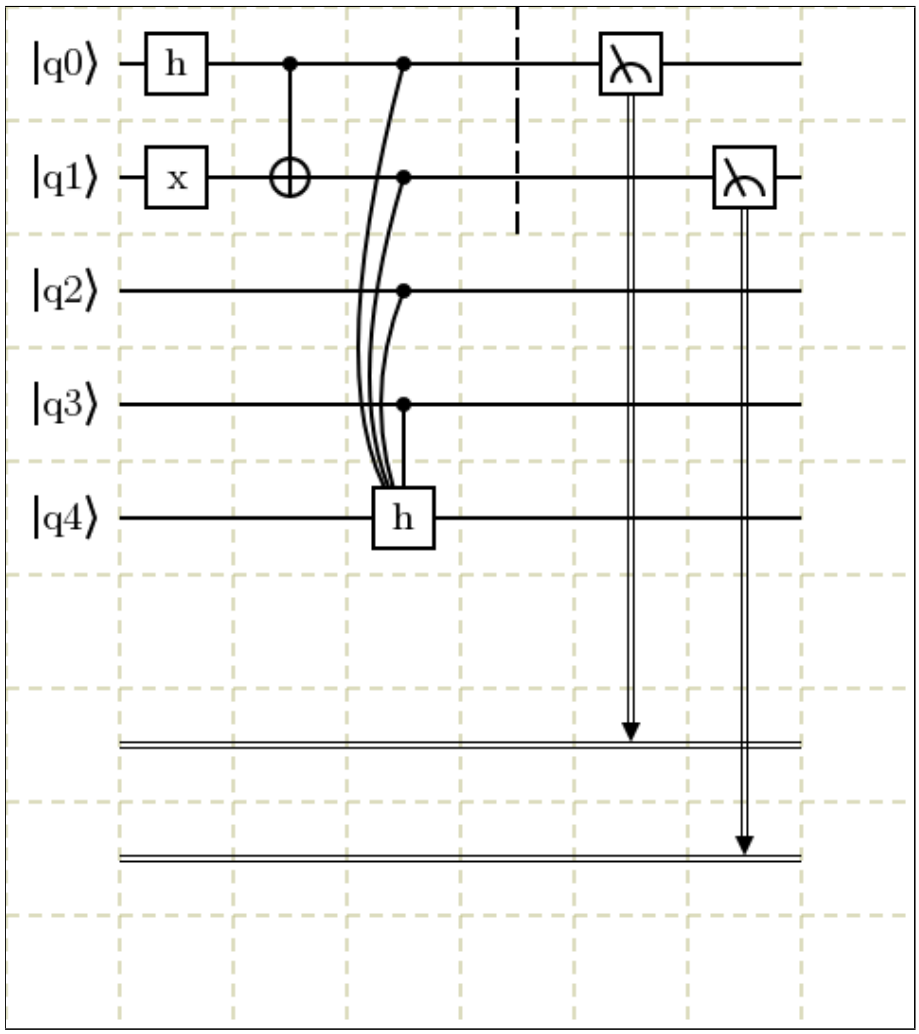

Figure D.2 First curve control lines

So in my opinion, if I made those line curve, I can put something under those curve. It looks like it is space efficient but turns out it made circuit looks more confusing. So I ditch that idea entirely.

In Figure D.7, I am experimenting with dynamic-width cell configuration. My friend told me that in order to support parameterized gate, we need more space than a square to display those gate's parameter. A pink line show how I segment the area. A narrow vertical space is a insertion space. If user want to insert some operation between another operations, they can drop it there.

But after some initial test, Dynamic-width operation is very slow because it has to do search and look-up every time user move mouse over it. This is very expensive calculation and make it unusable. So I come back to fixed-width operation style.

This time I refactor the code to use more look-up table and make the operation an extension of the other operation so that we can merge some common code path into single point and use it everywhere. By this method, adding more gate is just adding some additional code over the base gate as in Figure D.8. It looks promising but after adding more gate to the code base. It turns out to be a mess. Figure D.9.

After many failed attempts at canvas technology, I switched to SVG. SVG is a technology to create a vector graphic using the same structure as an HTML document. Instead of a paragraph of text and embedded images, we can draw circles, squares, lines, and many complex shapes. Because every graphic on the screen has a corresponding element associated with it, I can easily work with it. I can draw some lines and change their

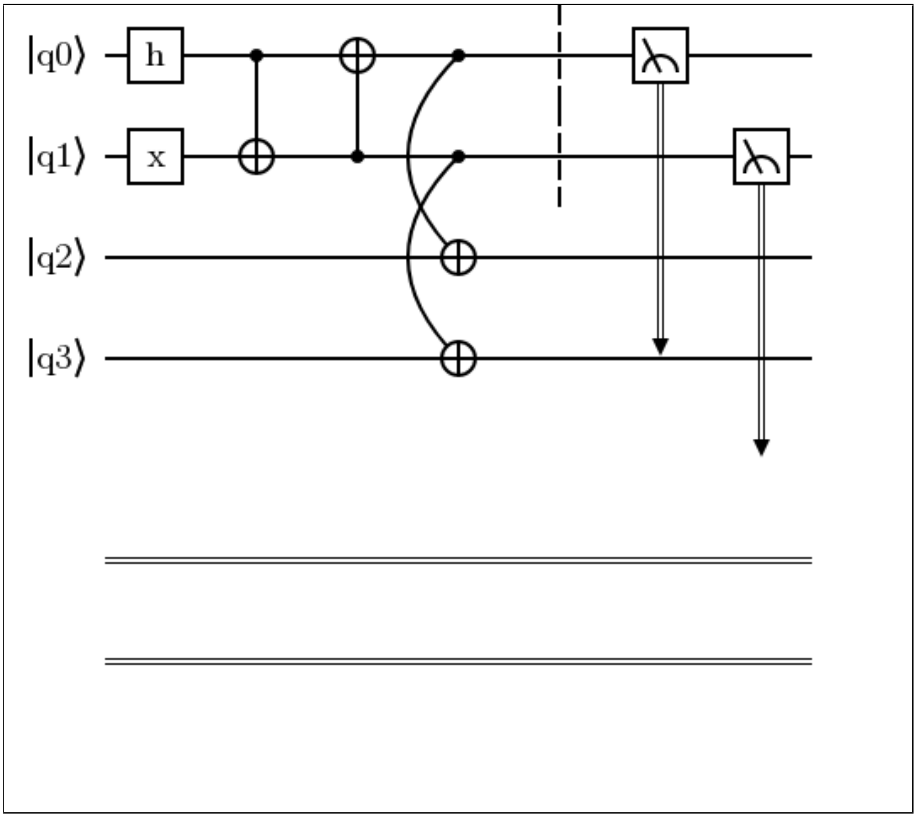

Figure D.3 Make it more curves

length and position at a later time by updating the attributes of that line's element. Also, I can associate an event listener with each graphic element. So, when a user interacts with any graphic, it will invoke our code to handle that interaction. This makes my job a lot easier because I do not have to implement such complex interaction systems by myself anymore. I just use the interaction system that exists in Document Object Model of HTML and SVG instead. Now, when I want to create some object in the editor, I break it down into a bunch of graphic elements, decide which parts the user can interact with, add those parts into SVG, and attach an event handler to that part accordingly. But with that advantage, it still comes with some drawbacks. In the canvas system, canvas is just a medium on which we can draw. It did not have any object models or any internal states. Its state consists only of a bitmap of things we draw on it. By this way, I have to store an interaction state by myself and erase and draw canvas again if something changes. So, in canvas, we have a single data source. It is our data structure. But, in SVG, SVG itself has an object that we need to populate with graphic objects in order to display something. I use my data structure to populate it. After that I need to maintain parity between those data structures: the SVG object model and my data structure. But this is still far easier than using canvas and inventing a interaction system by myself a lot. See Figure D.10 for how SVG renderer looks like.

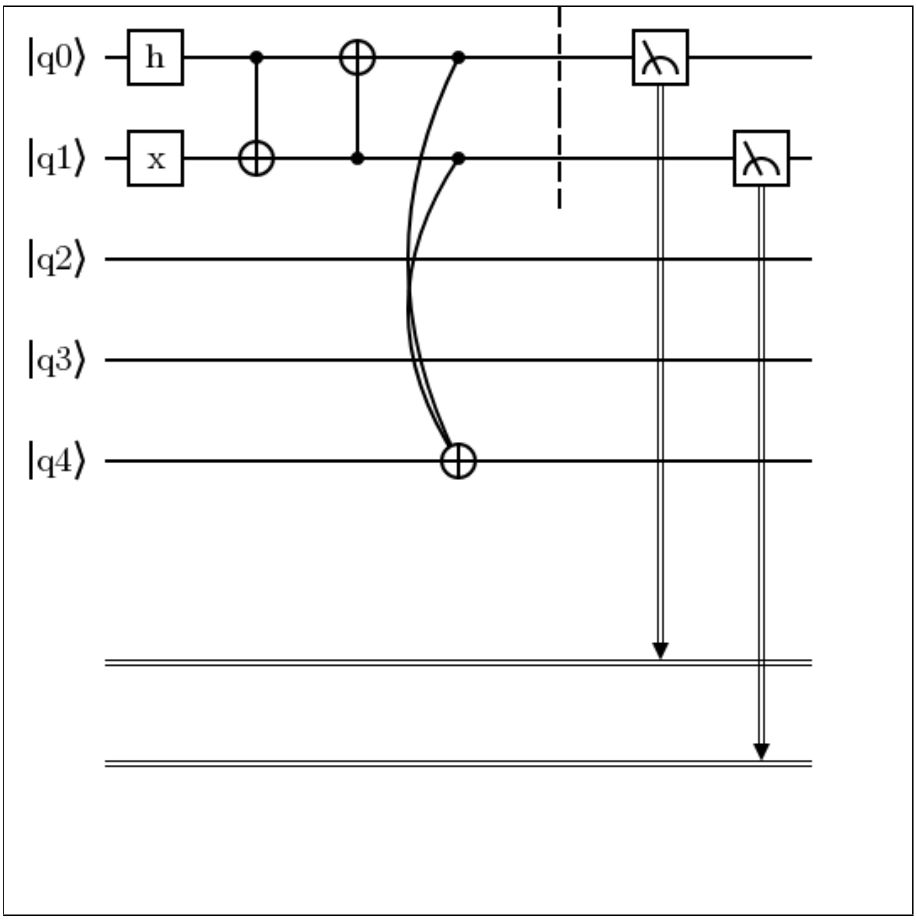

Figure D.4 Lines is not aligned

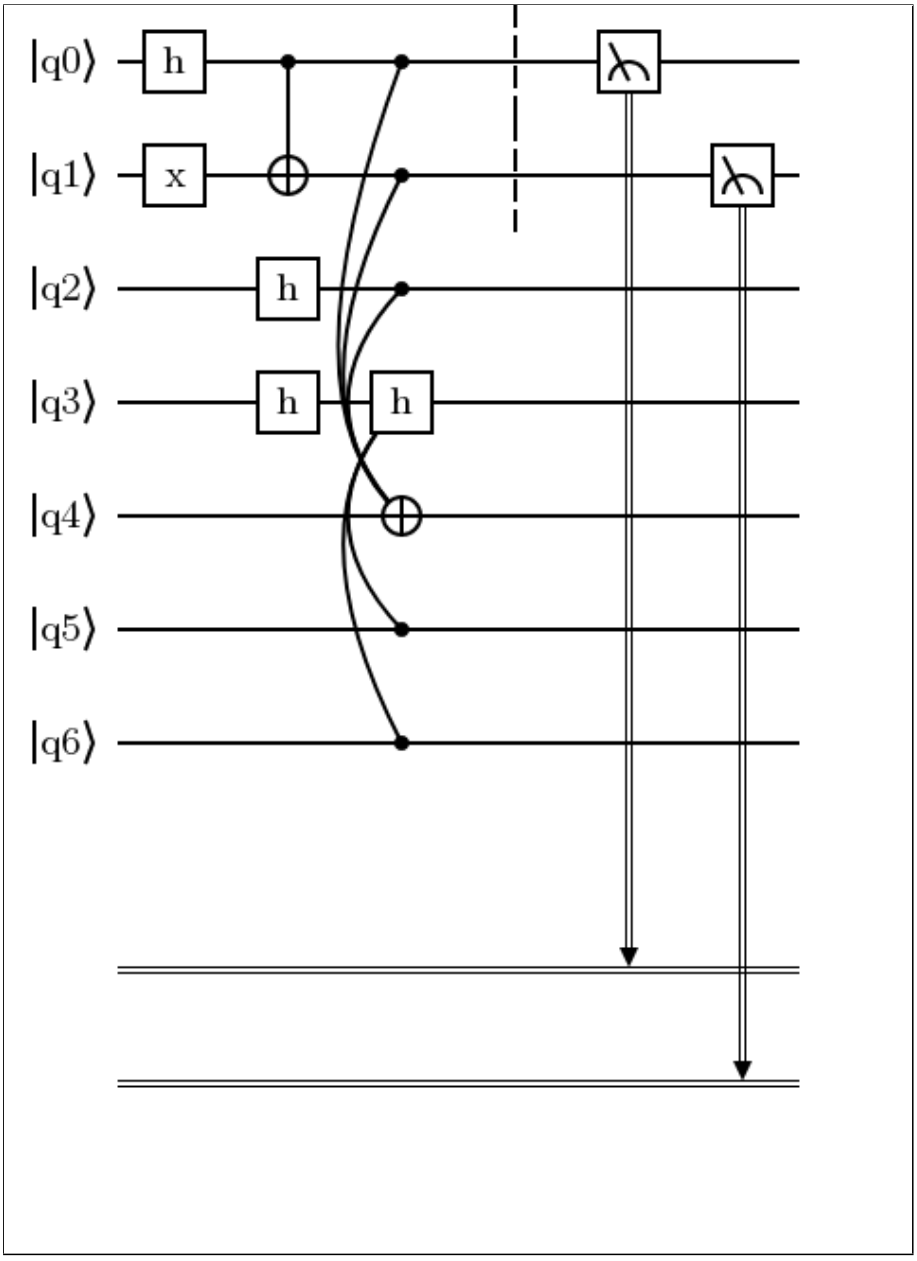

Figure D.5 Aligned curved line

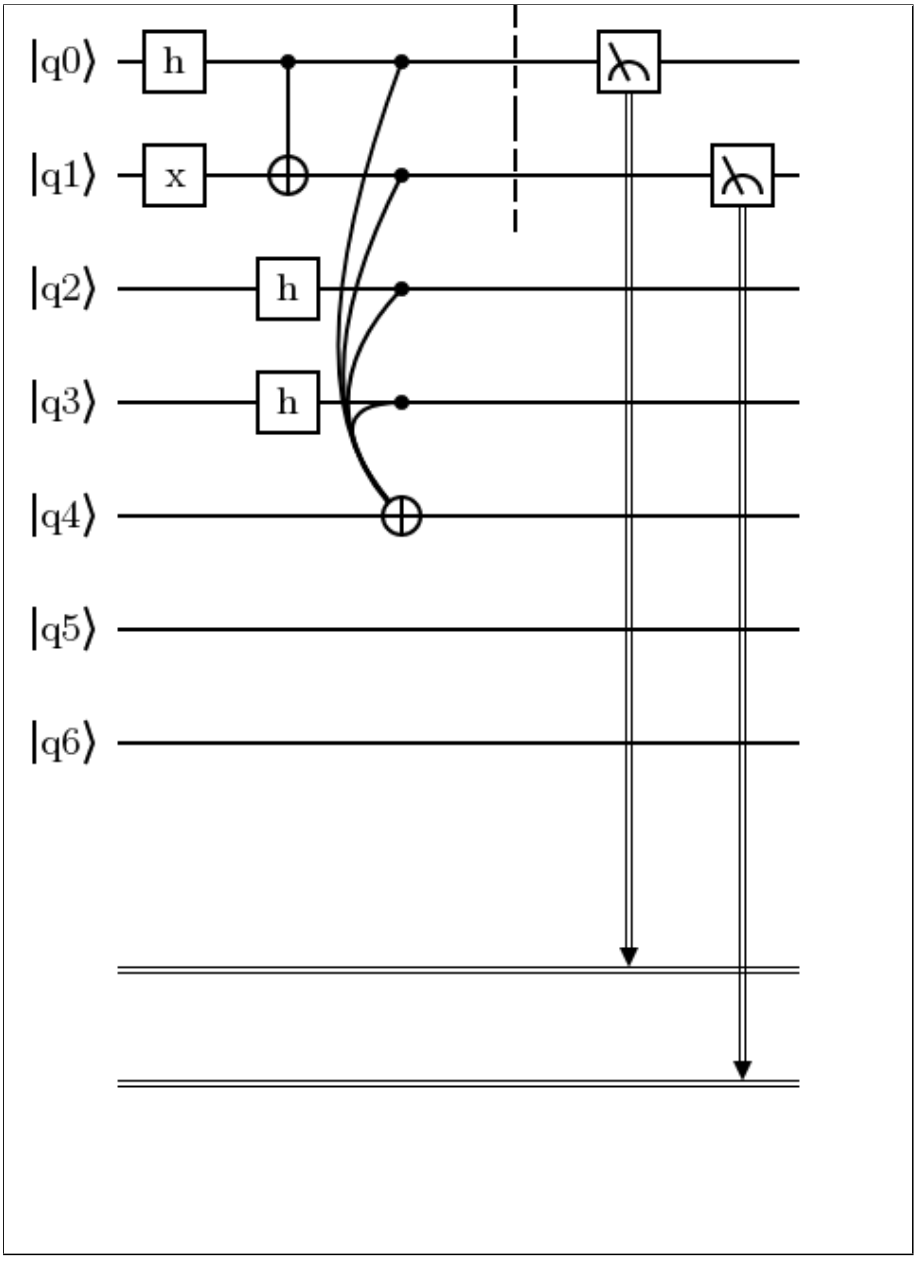

Figure D.6 Inclined curved line

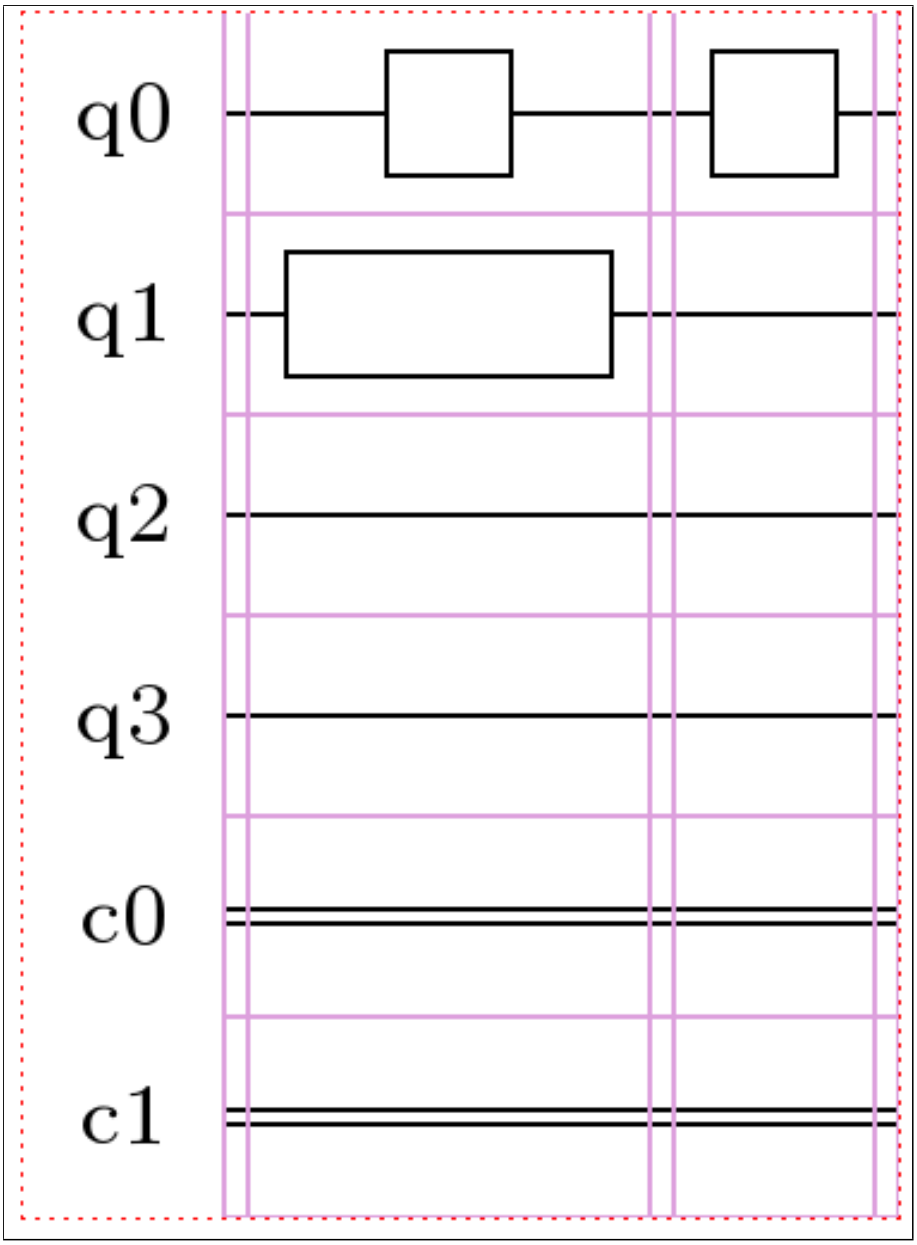

Figure D.7 Dynamic-width operation style prototype

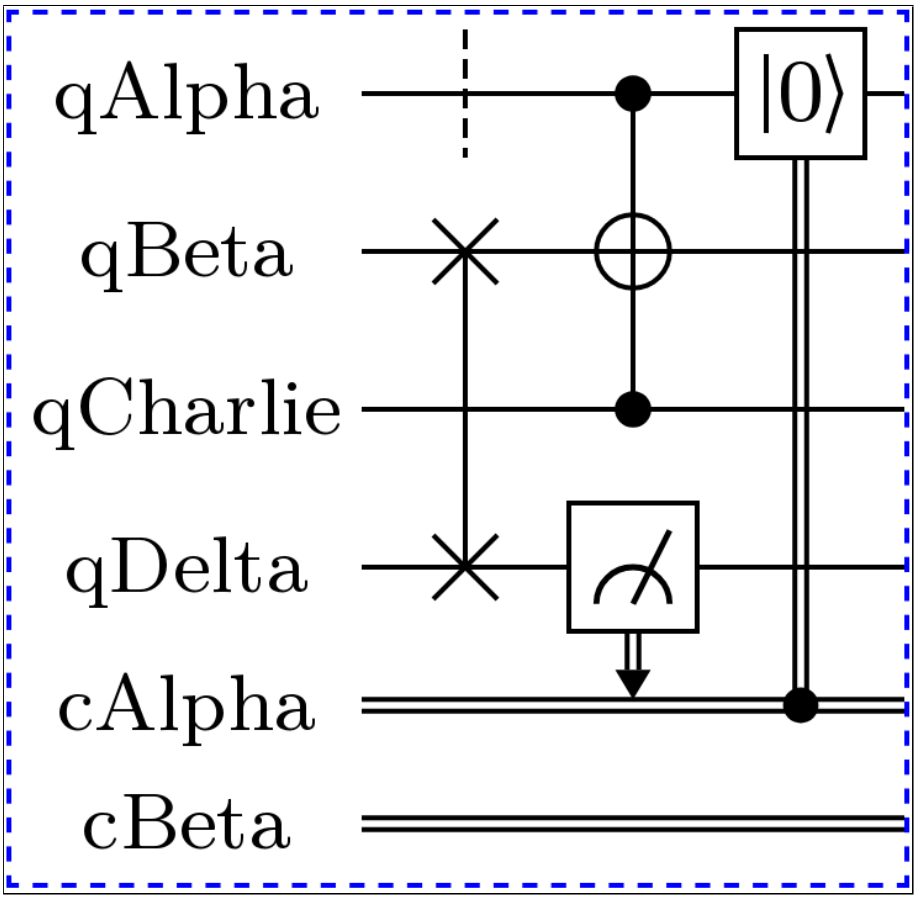

Figure D.8 Extensible operation prototype

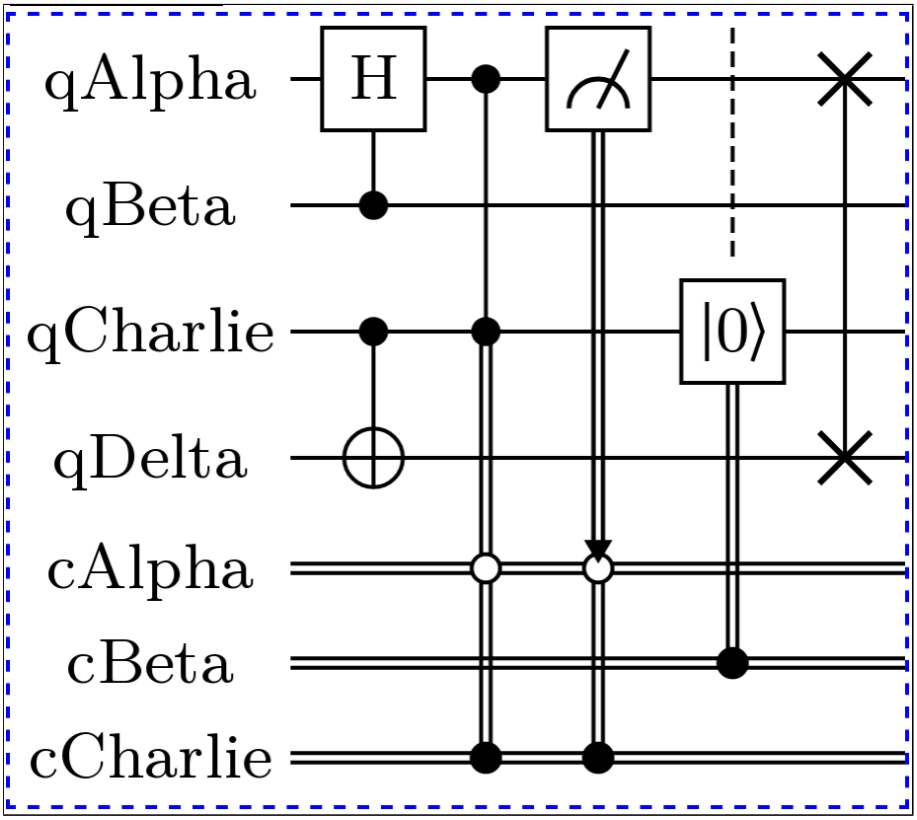

Figure D.9 Final extensible operation prototype

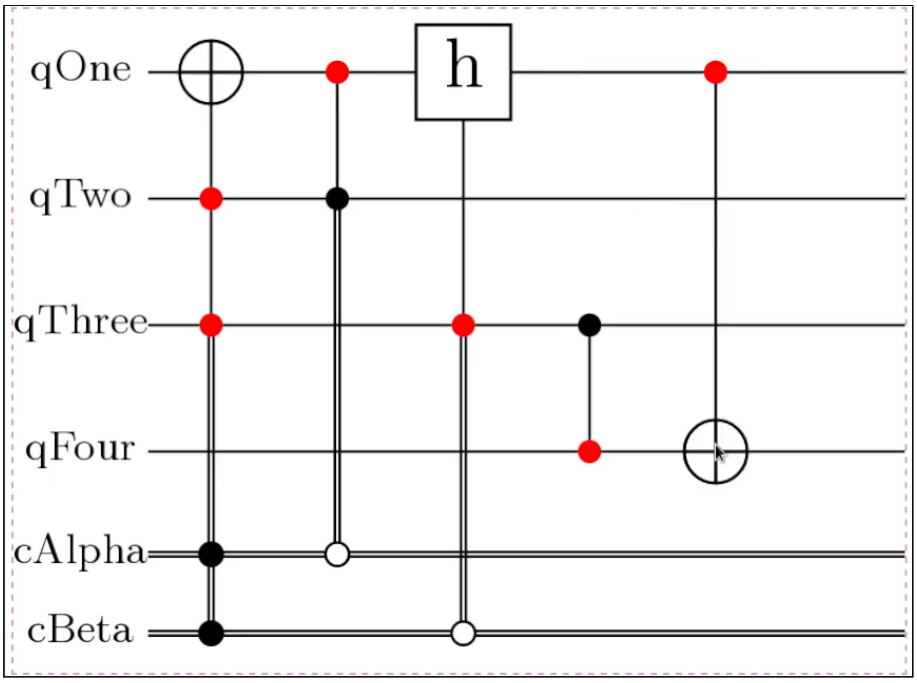

Figure D.10 SVG renderer