THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Software-Level Implementation of Quantum Finite Automata

by

Nicholas Lambourne

School of Information Technology and Electrical Engineering,
University of Queensland

Submitted for the degree of Bachelor of Engineering (Honours)
in the division of Software Engineering

November 2020

Nicholas Lambourne
nicholas.lambourne@uqconnect.edu.au

December 23, 2020

Prof Amin Abbosh
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Queensland 4072

Dear Professor Abbosh,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the School of Information Technology and Electrical Engineering, I present the following thesis entitled

'Software-Level Implementation of Quantum Finite Automata'

This thesis was performed under the supervision of Professor Janet Wiles (ITEE) and Dr Sally Shrapnel (SMP). I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Nicholas Lambourne

To Janet, Sally, and Michael, for making this year possible.

And to Sipser, Say, Yakaryılmaz, Tian, and Freivalds, for laying the groundwork and bringing together two of my favourite fields.

Acknowledgements

Thanks, first and foremost, needs to go to my incredible supervisors: Janet Wiles, Sally Shrapnel, and Michael Kewming. This project came about after I stumbled across a paper on arXiv [1] while a research assistant in Janet's lab at UQ. It was Janet who suggested I reach out to Sally and SMP to explore the prospect of a project that spanned both the School of IT & Electrical Engineering and the School of Mathematics & Physics. With the addition of Michael (then one of Sally's PhD students, now a freshly minted postdoc) - brought in to explain all of the quantum phenomena I routinely struggled with - my set of mentors was complete. Our consistent, weekly Monday and Thursday meetings, held despite all of their busy schedules, kept me on track throughout the year and led to such an expansion of my understanding of both automata theory and quantum computing that I'm still processing it all. Their guidance and support throughout the year has been invaluable and it is to them I owe the most with regards to this work.

Thanks is also due to my friends, coworkers, and classmates: Emily who, having completed her thesis the year before, did an excellent job of commiserating with me over the trials and tribulations, also proved to be an excellent proof-reader, even at the shortest of notice. To Ben, for bringing me on board to help out with his own thesis students this year, which left me hyper-aware of due dates and assessment expectations. To Brae and Ashleigh, for putting up with me asking how their theses were progressing whilst simultaneously being too polite to ask how mine was. And to JT, for understanding what I meant by "orthonormal basis states" when none of my other friends did.

The UQ Computing Society also deserves mention here, primarily due to its ability over the past year to provide convenient distractions I could use to justify any amount of procrastination from my thesis-related obligations. I strongly recommend the club to others coming up through undergraduate degrees in ITEE, for this reason and so many others. To all of my friends in the society, and particularly the new executive team who recently took over (enabling me to finish this document), I am incredibly grateful.

Last but certainly not least, my family, who nodded politely and smiled vacantly, eyes glazing over, when I explained what it was that I was working on and why the lights were still on at 3AM most Monday mornings.

Abstract

This thesis explores the feasibility of implementing quantum finite automata (QFA), the quantum analogue of classical finite state automata (FSA), in software, using commercially available quantum computing platforms. Two theoretical studies proposed by Say & Yakaryılmaz, [2] and one hardware experiment conducted by Tian et al. [3] were successfully implemented across four quantum software platforms: Xanadu Strawberry Fields, IBM Qiskit, Rigetti Forest, and Cirq. The first study addressed the EVENODD problem and showed that QFA can solve certain promise problems using constant ($O(1)$) state when the problem classically required state that grew linearly ($O(n)$) with the problem size. The second study involved the NEQ language recognition problem and showed experimentally that QFA could recognise certain non-regular languages, something FSA are fundamentally unable to do. The final study showed that QFA could classify input based on divisibility using constant ($O(1)$) state. When run on general-purpose quantum computers, each study suffered to some degree from the impacts of noise. This noise effect was most pronounced in the third study, where in certain configurations over 50% of experimental runs resulted in an invalid terminal state. These effects are expected to be mitigated to some degree by future advances in the underlying hardware and quantum error correction, but in the near term the effect makes it appear that multi-qubit QFA are unsuitable for some use cases. Despite this, single-qubit systems were shown to be relatively noise-tolerant and the relative ease with which QFA were implemented suggests that the utility of developing other quantum abstractions based on classical analogues can now also be easily explored experimentally.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The potential of computers built upon quantum phenomena has interested researchers since at least the late nineteen eighties, when Richard Feynman wondered if they might pose the solution to problems faced in simulating highly complex natural systems [4]. Since then, researchers the world over have made incredible progress towards implementing functional, general-purpose quantum computers. What has been lacking is a definitive understanding of how we can harness this power to solve real-world problems. While there have been several notable discoveries in specific fields like search (Grover's algorithm [5]) and prime factorisation (Shor's algorithm [6]), we still lack the fundamental abstractive tools required to use quantum computers as general-purpose problem solving machines, or at the very least direct our efforts towards the kinds of problems they may be best suited to solve.

One avenue of research into harnessing the power of these new devices has been examining whether the abstractions used to solve problems on classical computers have quantum analogues that might prove effective. One particularly powerful abstraction, the finite state automata (FSA), caught the attention of researchers well before quantum hardware was advanced enough to test their hypotheses. Work by Freivalds in the late 1990s [7], [8], later built on by Kondacs and J. Watrous [9], and Say & Yakaryılmaz [2], among others, culminated in a 2014 paper *Quantum finite automata: A modern introduction* in which Say & Yakaryılmaz presented five theoretical examples of ways in which quantum finite state automata (QFA) might outperform their classical equivalents [2]. Alas, ready access to quantum computing hardware was still not available at that time to test these hypotheses experimentally.

With the advent of general purpose quantum computers accessible to the public via internet-enabled application programming interfaces (APIs) and a wealth of quantum software development kits (SDKs) made available by companies like IBM [10], Rigetti [11], Xanadu [12], [13] and others, it is now more than feasible to test theories like these.

This thesis explores the implementation of two of the experiments proposed by Say & Yakaryılmaz: the EVENODD promise problem, purported to demonstrate the power of QFA in reducing the amount of state required to solve some problems; and the NEQ

language recognition problem, which demonstrates their ability to recognise languages that classical finite state automata are fundamentally unable to recognise [2], [14]. In an effort to explore a more sophisticated example, a hardware experiment conducted in 2019 by Tian et al. [3] was also replicated in software. This latter experiment addressed the "prime remainder" problem, where input is assessed and classified based on divisibility.

Each of these three experiments were implemented across four different quantum software platforms. Simulated results were gathered and examined across all platforms and, where possible, the experiments were also run on actual general purpose quantum computers.

Chapter 2

Background

2.1 Finite State Automata

Finite State Automata (FSA) are a specific subclass of Turing machine used to represent a system of possible states where movement between states is determined by input. [15]

As an abstraction on top of classical computing capability, FSA have proven to be incredibly powerful tools. They are able to perform all sorts of tasks, from the simple: determining whether the length of a sequence of characters is even; to the complex: playing a vital role in automatic speech recognition [16].

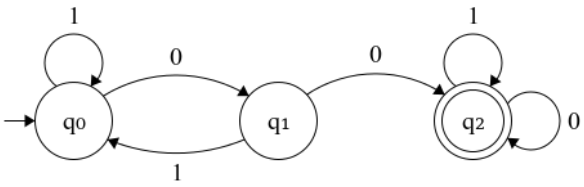

Figure 2.1: Deterministic Finite Automaton for the Detection of '00' in a Binary Sequence [17]

These studies focuses solely on the deterministic and non-deterministic forms of FSA, rather than the more general probabilistic variety that forms a superset of classical non-deterministic automata. Figure 2.1 shows a type of FSA called a Deterministic Finite Automata (DFA) for detecting the sequence '00' in a binary string [17]. Each circle represents a state, and states surrounded by two circles are acceptance states. When an FSA terminates in an acceptance state, the input is considered to have been accepted, termination in any other state indicates rejection. The arrows between states represent transitions achieved by consuming input that matches that arrow's label (here, either zero or one). The unlabeled arrow entering state q_0 represents the start of the process. Taking the string '01001' as our input, we can follow the operation of the DFA:

- The automaton starts its process starts at q_0
- The first character, 0, is consumed. The automaton moves to q_1.
- The second character, 1, is consumed. The automaton moves to q_0.
- The third character, 0, is consumed. The automaton moves to q_1.
- The fourth character, 0, is consumed. The automaton moves to q_2, the accepting state.
- The fifth character, 1, is consumed. The automaton does not move from q_2.
- There is no more input to consume, the automaton completes in an 'accepting' state.

Thus, we can be certain that the string meets the criteria defined by the automaton. This kind of pattern-matching problem is common in computer science (and many other fields), and the use of FSA to solve them is equally prevalent. Most examples, however, are far more complex and rely on additional features like non-determinism (where state transitions can be made at any point without consuming an input), and typically involve many more states than the trivial example above [15]. This potential for massive reliance on state represents both a limitation of classical FSA and an opportunity for the introduction of alternative solutions.

Formally, FSA are defined by a tuple containing five elements. The previous example, now called G, would be represented by the tuple $(Q, \Sigma, \delta, q_0, F)$ [15]. The elements of the tuple, and their meaning with respect to G, are as follows:

- Q: the finite set of all states in the FSA ($\{ q_0, q_1, q_2 \}$).
- Σ: the finite set known as the alphabet of accepted characters (i.e. $\{ 0, 1 \}$).
- δ: the transition functions from each state $Q \times \Sigma \rightarrow Q$ ($\{ q_0 \times 1 \rightarrow q_o, q_0 \times 0 \rightarrow q_1 \ldots \}$).
- q_0: the starting state (q_0)
- F: the set of accepting states ($\{ q_2 \}$). [15]

This notation will be used throughout these studies when formal rigour is required in defining various finite state automata.

2.1.1 Introducing Non-Determinism and the NFA

The automaton described above is an example of a deterministic finite state automaton. There also exist other types of FSA that are non-deterministic. Non-deterministic Finite Automata (NFA) are a superset of their deterministic counterparts, using the same formalism, but introduce new behaviour, notably:

- The ability to "split" and follow two paths on consumption of input. If any of the paths of execution result in completion while in an accepting state, the input is considered accepted.
- The ability to consume the empty string "ϵ" (i.e. move to a new state without consuming the next character of the input). [15]

These new behaviours appear to significantly increase the power of the FSA to solve interesting problems. It has been extensively proven, however, that for every NFA there exists an equivalent deterministic FSA [15]. Despite this, NFA serve a valuable abstractive role when solving problem with automata. Both deterministic and non-deterministic FSA have quantum computing counterparts that provide even greater possibilities [3], [9].

2.2 Quantum Computing and the New State of State

The concept of quantum computing was popularised by Richard Feynman in his 1982 paper *Simulating Physics with Computers* [4] as a solution to the problems posed by trying to simulate the exponential complexity of physical systems using limited classical hardware [18]. Several decades later, we are seeing real progress towards the realisation of Feynman's dream. Universities and private corporations all over the world are demonstrating experimental quantum computers based on various architectures, from ion traps and superconducting circuits, to monolithic diamonds and linear optics [19].

2.2.1 The Qubit

The common theme amongst these various implementations is their reliance on the fundamental unit of quantum computing, the qubit. Qubits, like their classical counterparts, bits, have state. However, unlike classical bits, which can exist exclusively in one of two states (0 or 1), qubits can, in addition to taking the states $|0\rangle$ or $|1\rangle$ (analogous to those two classical states), take additional states comprised of linear combinations of those two classical states [20], in the form:

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.1}$$

Where α and β are complex numbers. This form, known as Dirac notation, allows us to represent any possible state the qubit may take. Another way of representing this new, enhanced form of state is using a Bloch sphere. A Bloch sphere, depicted in Figure 2.2, takes the same information about a qubit's state and represents it as a three-dimensional unit vector [20]. Both formats convey the same information, but one or the other may be better suited to various uses (e.g. conceptualising the application of transformations). In this form, the various states available to the qubit can be envisaged as each position the unit vector in the middle of the sphere can take on the sphere's surface. The direction of

the unit vector in this case can be summarised as the combination of two angles θ and ϕ such that $\alpha = cos(\frac{\theta}{2})$ and $\beta = e^{i\phi}sin(\frac{\theta}{2})$ [20].

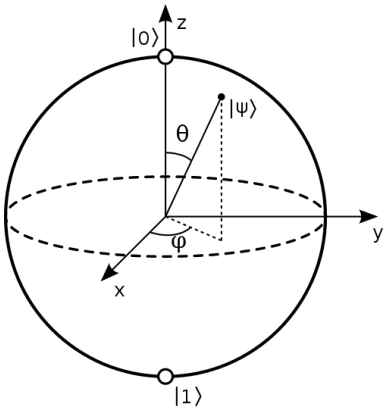

Figure 2.2: The Bloch Sphere Representation of a Qubit [21]

The final way qubit state is commonly described is using vectors [20]. For instance, a qubit in the state $|1\rangle$ can be represented with the following column vector:

$$|1\rangle \equiv 0\,|0\rangle + 1\,|1\rangle \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.2}$$

Or, more generically:

$$|\psi\rangle \equiv \alpha\,|0\rangle + \beta\,|1\rangle \equiv \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.3}$$

2.2.2 Manipulating Qubits

The state of a qubit can be modified by applying physical transformations to its 'spin'. In the abstract, this takes the form of applying 'unitary' matrix operations to the matrix representing the qubit's state. These unitary operations, while changing the direction of the vector, do not change its magnitude from 1. The result of these operations can be envisaged as the unit vector changing its position on the surface of the Bloch sphere [3]. Consider, for example, a qubit in state $|0\rangle$ and the application of the unitary operation $X = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$:

$$X\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.4}$$

The effect of this operation can be seen in Figure 2.3:

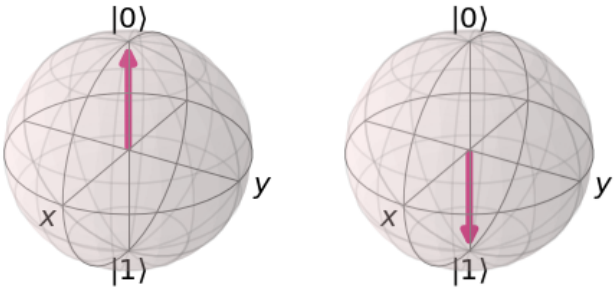

Figure 2.3: $|0\rangle$ Before (Left) and After (Right) the Application of a Pauli-X Transformation [10]

An infinite number of such matrices exist, allowing for the transformation of a qubit's state such that its unit vector could point to any point on the surface of the Bloch sphere. The implied suggestion of practically infinite state is intriguing, but ultimately misleading, as there are various other properties of qubits that limit their potential in this regard. The first and most important of these is the tendency of qubits to 'collapse' upon inspection, whereby upon measurement, only values of zero or one are actually observed, with the respective probabilities $|\alpha|^2$ and $|\beta|^2$ [20]. This being the case, numerous iterations of the same experiment are required to confirm that the experimental findings match the theoretical probabilities; naturally, the more experiments that are conducted, the more accurate the resulting distributions.

2.3 Quantum Finite Automata

There has been significant interest in combining the powerful pattern-recognition capabilities of finite state automata and the potential that quantum computing offers for some time [9]. What has been lacking is adequate hardware (or simulators) to test and develop upon this interest. As such, much of the formalism around what constitutes a quantum finite state automaton has already been established, and is largely similar to its classical counterparts.

2.4 Proposed Advantages of QFA

While it is acknowledged the fact that qubits are not a gateway to costless, essentially unlimited state, they do present opportunities to take advantage of a reduced need for additional state in some specific cases. This reduction in required state, which has less apparent value in more trivial examples, has already provided $O(1)$ space solutions to what formerly would have required $O(n)$ space under classical conditions [3]. This kind of problem, when dealing with a large n, or running thousands or millions of instances of

the same program, could drastically reduce state (memory) requirements, and therefore cost, in the long term. Applications have also been identified in problems where classical deterministic and non-deterministic FSA are unable to produce a result but QFA can, like in the recognition of nonregular languages [2].

2.5 Quantum Software Platforms

In order to provide a concrete implementation of software-level quantum finite state automata, it will be necessary to build off existing abstractions on quantum hardware. Thankfully, various quantum software platforms already exist [19]. This section will outline the various ways in which they abstract on quantum hardware to provide an interface to the user.

2.5.1 Quantum Gates

In a similar fashion to early imperative programs, or the more recent functional paradigm, programs in quantum computing typically take the format of successive applications of various primitive operations, or the application of more complicated functions. Given the somewhat primitive level of abstraction currently employed (compared to classical software), the analogy of electric (or quantum) circuits has been employed to allow researchers to visualise the process qubits undergo as they progress through a circuit. These circuits consist of an input state, various standard or custom "gates", and typically one or more measurement operations (keeping in mind the unique effect of measurement on quantum systems). The example in Figure 2.4 shows a simple quantum circuit consisting of two standard gates, the Pauli-X gate (analogous to a classical NOT gate) and a Hadamard gate (equivalent to a rotation of π about the Z-axis and $\frac{\pi}{2}$ about the Y-axis of the Bloch sphere) [20]. The final element in the circuit is a measurement of the state of the qubit, where, due to the effect of the Hadamard gate, we would expect to see a roughly equal distribution of 0 and 1 results over multiple attempts.

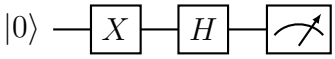

Figure 2.4: A Simple Quantum Circuit Featuring a Pauli-X Gate and a Hadamard Gate

2.5.2 Programming Quantum Computers

While some software platforms like IBM Qiskit [10] provide interactive circuit simulators, it quickly becomes tedious to develop circuits, particularly those involving loops, or mixing classical and quantum behaviour. As such, the default experiment design workflow typically utilises either a dedicated quantum programming language (e.g. Q# [22]) or a quantum computing package built into an existing language like Python (e.g. IBM

Qiskit [10], Rigetti PyQuil [23]). In these languages, gates are normally exposed to the user through an object-based API, which allows for far more complicated (and portable) experiments to be conducted [24]. For example, the previously system-specific circuit simulation depicted in Figure 2.4 is represented in a far more portable format (using IBM's qiskit package for Python 3) in Listing 2.1.

```
# Import required Python Modules
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble

circuit = QuantumCircuit(1)  # Establish a quantum circuit with 1 qubit
backend = Aer.get_backend('statevector_simulator')  # Prepare a simulator
circuit.x(0)  # Add a Pauli-X (NOT) gate to the first qubit's path
circuit.h(0)  # Add a Hadamard gate to the first qubit's path

job = execute(circuit, backend)  # Run the simulation
result = job.result()  # Get the simulation results
outputstate = result.get_statevector(circuit, decimals=3)  # Extract the
    state vector from the result
print(outputstate)
```

Listing 2.1: A Simple Quantum Program in Python

2.5.3 Quantum Programming Platforms & General Tools

Quantum Hardware Implementations

While there are a vast array of paradigms to choose from when it comes to general purpose quantum computing hardware (photonics, quantum dot, ion trap, superconducting qubits... [25]), the platforms utilised for these studies fall into two categories: those that utilise photonics based systems, and those based around the concept of superconducting qubits.

Photonics-based - or linear optical - quantum computing, takes the photon, the elementary light particle, and uses its polarisation as a store of quantum state. Photons are emitted and sent along various optical channels, with various gates (beam splitters, wave plates, phase shifters) designed to impact their polarisation [3], [25]. These photons are then picked up by photon detectors, where their detection (or lack thereof) informs the experimenter about the state of the photon [25].

Superconducting qubits are considered one of the most promising technological implementations of general purpose quantum computers [26]. Their implementations vary substantially between their many variants and hybrid variants but operations are typically effected via microwave irradiation on a pair of electrons known as a Cooper pair [26], [27].

The primary goal of this work is to show that quantum finite state automata can be implemented in software, so the underlying hardware is of little concern unless it impacts the functionality (e.g. gates) available at the software level.

Quantum Platforms

Each study has been implemented across all four of the following platforms. Each platform is implemented as a Python library:

- **Strawberry Fields**: the only photonics-based platform used in these studies Strawberry Fields is a quantum computing platform provided by Xanadu [12], [13]. Hardware was reasonably accessible, but was complicated by the type of studies conducted (see Section 3.5).
- **Qiskit**: IBM's Qiskit SDK and "Quantum Experience" platform is based around superconducting qubits and proved to be the most accessible [10]. Their network of general-purpose quantum computers around the globe and the ability to submit jobs to them via API proved invaluable in assessing the impact of noise.
- **Forest**: Rigetti's Forest platform is also based around a superconducting qubit architecture, it was noticeably the only to require a compiler and virtual machine to be running as servers on the local machine [11]. Access to Rigetti hardware was limited (see Section 3.7.1).
- **Cirq**: an unofficial Google project, Cirq is run as an open-source project hosted on GitHub. Based on a superconducting qubit paradigm, Cirq offers a full complement of standard gates and is the only one of the four platforms to natively support qudits (qubits with other than two basis states). Access to compatible Google hardware is restricted and requires the sponsorship of a Google employee [28].

Chapter 3

Study 1: State Reduction

This first study centres around an experiment theorised by Say & Yakaryılmaz [2] which was designed to demonstrate the superiority of a quantum finite automata in terms of the amount of state required to solve a particular variety of promise problem. This experiment, one of five presented in their 2014 paper published in *Computing with New Resources*, purports to show that the EVENODD promise problem is a candidate for expressing this experimentally. The aim of this study is to not only implement the experiment as theorised, for the first, time to my knowledge, but to do so in software and across all four quantum platforms outlined in Section 2.5.3.

3.1 The Importance of State Reduction

One of the biggest limitations of classical computers is their finite resources. Even massive, auto-scaling computer clusters operated by the likes of Google and Amazon have a tangible upper limit on capacity. These limits typically fall into one of two categories: computational constraints (compute) and memory constraints (space). Developments in computer science have led to a variety of optimisations in each of these areas designed to make more efficient use of available resources, but these optimisations are generally limited to particular domains. For example, a computation optimisation in the sorting was the development of the Quicksort algorithm [29], capable of sorting n elements in $O(nlogn)$ time (on average). From a space optimisation perspective, the concept of delayed or 'lazy' evaluation (while also a computation improvement), means that storing the (potentially large number of) computed values is delayed until those values are actually needed for execution. This is typified by the Python programming language's 'generator' objects [30].

Quantum computers have been theorised to have something similar to offer in terms of domain-specific optimisations [2]. Specifically, for certain kinds of promise or pattern matching problems, unique aspects of how quantum computers function can be utilised to solve existing problems in novel ways using less state. The practical impact of this, while not truly demonstrated in the smaller scoped examples provided by Say & Yakaryılmaz,

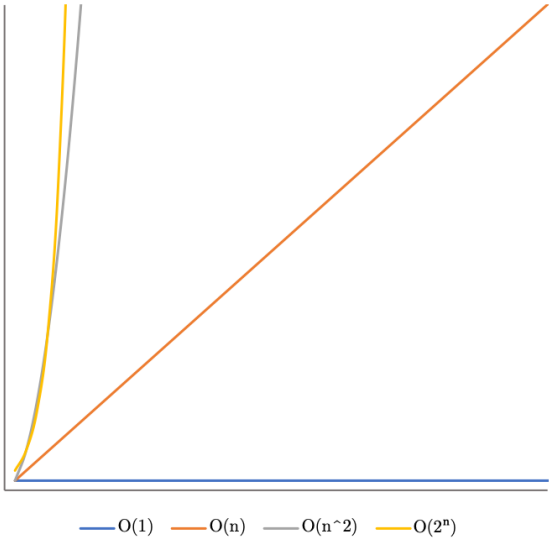

Figure 3.1: State Growth with n

can most clearly be demonstrated when expanded to larger scale problems. The impact of a reduction from requiring $O(2^n)$ space to constant ($O(1)$) space is visualised in Figure 3.1.

3.2 The Compute/Space Trade-off

The optimisations outlined in Section 3.1 do not come without cost. Because of the probabilistic nature of quantum computers, each experiment needs to be run several times (potentially thousands) in order to establish stable and accurate results. These 'runs', or 'shots', involve running the full experiment and determining a projection of the state of the qubits in the system. Each shot's results are aggregated into a frequency distribution showing how likely the system was to collapse into any given state.

This results in a trade-off between use of additional computational resources (required to achieve the desired level of accuracy) in exchange for a decrease in state typically measured in orders of magnitude. This compromise can be thought of as analogous to the reverse situation of 'caching' in classical computing, where additional space is utilised to store commonly accessed values, typically increasing the speed of computation. In using quantum computers to achieve space supremacy, we can theoretically reduce space requirements from $O(n)$ or even $O(2^n)$ to $O(1)$, or constant space (i.e. a constant number of qubits).

3.3 The EVENODD Promise Problem

The EVENODD promise problem was first proposed by Say & Yakaryılmaz in 2014 and later expanded on by Geffert and Yakaryılmaz [31] describes the following situation:

$$EVENODD^k_{accept} = \{a^{j2^k} | j \text{ is a non-negative even integer}\}$$
$$EVENODD^k_{reject} = \{a^{j2^k} | j \text{ is a non-negative odd integer}\}$$

This set of conditions describes two distinct sets of strings for each variation of k, an integer. All strings in both sets consist solely of a characters, varying only in length. The first set ($EVENODD^k_{accept}$) contains strings of a length $j2^k$ where j is even, the second set contains all those strings where j is odd. Strings of all other lengths are ignored. To solve the EVENODD problem with an automata, it must accept all strings in the first set with probability of one and accept all in the second with a probability of zero. [2]

Solving this problem with a classical automata would would require at least 2^{k+1} states, indicating that as the problem-defining variable k increases, it results in an exponential increase in the amount of required state. [2]

3.4 Implementing EVENODD in Quantum Software

To solve the EVENODD problem using a quantum finite automata, Say & Yakaryılmaz [2] propose using a single 2-state qubit system where, upon consumption of each block of 2^k a characters, it makes a rotation of θ about the y-axis of the Bloch sphere (depicted in Figure 3.2).

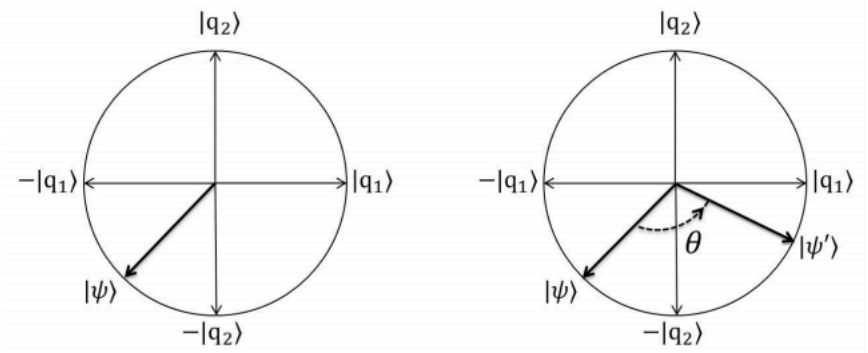

Figure 3.2: θ Bi-Directional Rotation Around the Y-Axis [2]

The value of θ is derived, based on the value of k that defines the particular instance of EVENODD:

$$\theta = \frac{\pi}{2^{k+1}} \quad [2]$$

This choice of rotational angle exploits the mathematical fact that when θ is a rational product of π then the rotational unitary function:

$$U_\theta = \begin{pmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{pmatrix}$$

is periodic and will return to its starting location after a finite number of rotations around the y-axis of the Bloch sphere. [2]. Say & Yakaryılmaz use this mechanic repeatedly to solve a variety of similar problems.

In the QFA solution to the EVENODD problem specifically, on consumption of each a a rotation θ is made. Each set of 2^k rotations sum to a total rotation of $\frac{\pi}{2}$. Because each EVENODD language is only defined for length multiples of 2^k, the rotations must conclude at one of the four positions outlined in Figure 3.2: $|q_2\rangle$, $-|q_1\rangle$, $-|q_2\rangle$, or $-|q_1\rangle$ (rotations of $\frac{\pi}{2}$, π, $\frac{3\pi}{2}$, or 2π, respectively). In fact, by doubling the size of θ we can ensure that rotations cease on only $|q_1\rangle$ or $|q_2\rangle$, which are analogous to our single qubit's $|0\rangle$ and $|1\rangle$ states, respectively, with $|0\rangle$ as our single accepting state. This fulfils the requirements for what Say & Yakaryılmaz call a "succinct exact solution of promise problems," where there is no ambiguity or chance of error from the algorithm (this does not factor in error introduced by other sources, e.g. hardware noise) [2].

To effect this solution on the various software platforms it was necessary to find ways to perform repeated rotations about the y-axis and measure the resulting state of the single-qubit system. An execution harness was developed that was responsible for running each implementation over the same set of inputs, collecting the results and producing the frequency distribution plots. This code, along with a common function for deriving θ was packaged into a utilities module and is available in Appendix A. This harness takes a single parameter: a function to execute that returns a list of normalised frequencies corresponding to the measured (collapsed) basis states $|0\rangle$ and $|1\rangle$. That function, typically passed partially evaluated with the parameter simulation populated takes the following parameters:

- k: the integer value defining the variant of the EVENODD language.
- j: the integer variable, with k, that determines the length of the test string.
- simulation: a Boolean value that determines whether the experiment will be run on quantum hardware (if available), when false, or as a simulation, when true.
- shots: the number of times to repeat the same experiment (typically 4096).

For each platform, an implementation of this function, named run_experiment was required and injected into the execution harness. These implementations would be responsible for setting up the quantum environment and its initial state, applying the θ rotations about the y-axis, as well as measuring and reporting the resulting state of system.

3.5 Implementation 1: Strawberry Fields

The first system on which the EVENODD problem was implemented was Xanadu's Strawberry Fields. This photonics-based system is the most unique amongst the four platforms. Using two modes (or channels) a simple Strawberry Fields program was instantiated:

```
program = sf.Program(2)
engine = sf.Engine("fock",backend_options={"cutoff_dim": 2})
```

It was necessary to construct the input string and derive a value for θ using the provided parameters and the common library provided in Appendix A:

```
input_string = "a" * (j * 2) ** k
theta = get_theta(k)
```

While not strictly necessary, the decision was made to replicate the string and consume it character by character so as to most closely match the behaviour of a QFA under operational constraints. With theta and the input string available, the preparation of the circuit could begin in earnest. Using Strawberry Fields' unique implementation of a Python run-time context the two channels were instantiated, the first with a Fock(1) command (releases a single photon in the channel) and the second with a Vacuum() command (releasing no photons). The use of two channels was required because Strawberry fields, operating on the photonic implementation paradigm, does not natively support the y-axis rotation mechanism required to implement the circuit. Instead, a photonics-native gate, the BSgate, was utilised for the same purpose. The BSgate, or beam splitter gate, places any entering photon in a superposition across two channels. It takes a single parameter θ, which determines the likelihood of the photon, on measurement, being being detected in either channel. This effect is analogous to the operation of simple y-axis rotation gates found on other platforms. Still in the Strawberry fields program context, this BSgate is passed the parameter theta and applied across the two channels one time for every character in the input_string:

```
with program.context as q:
    Fock(1) | q[0]
    Vacuum() | q[1]

    for _ in input_string:
        BSgate(theta) | (q[0], q[1])
```

Finally the completed program is run using the engine defined above and the probabilities are extracted and normalised:

```
result = engine.run(program, shots=shots)
state = result.state
freqs = [round(abs(state.mean_photon(0)[0]), 4),
         round(abs(state.mean_photon(1)[0]), 4)]
```

The full implementation of this solution can be found in Appendix B.

3.5.1 Results

Due to present limitations of the hardware available from Xanadu for use with quantum circuits designed in Strawberry Fields, it was not possible to run this circuit on an actual (non-simulated) quantum computer. As such, only simulated results have been provided. Discussions with Xanadu staff indicated that the hardware was simply not capable of releasing individual photons in the manner required by the experiment, and it was not possible to refactor the experiment to suit the available hardware. It remains to be seen whether future developments in Xanadu hardware will alleviate this impediment as the changes required were described as "not currently on their roadmap."

Simulated Results

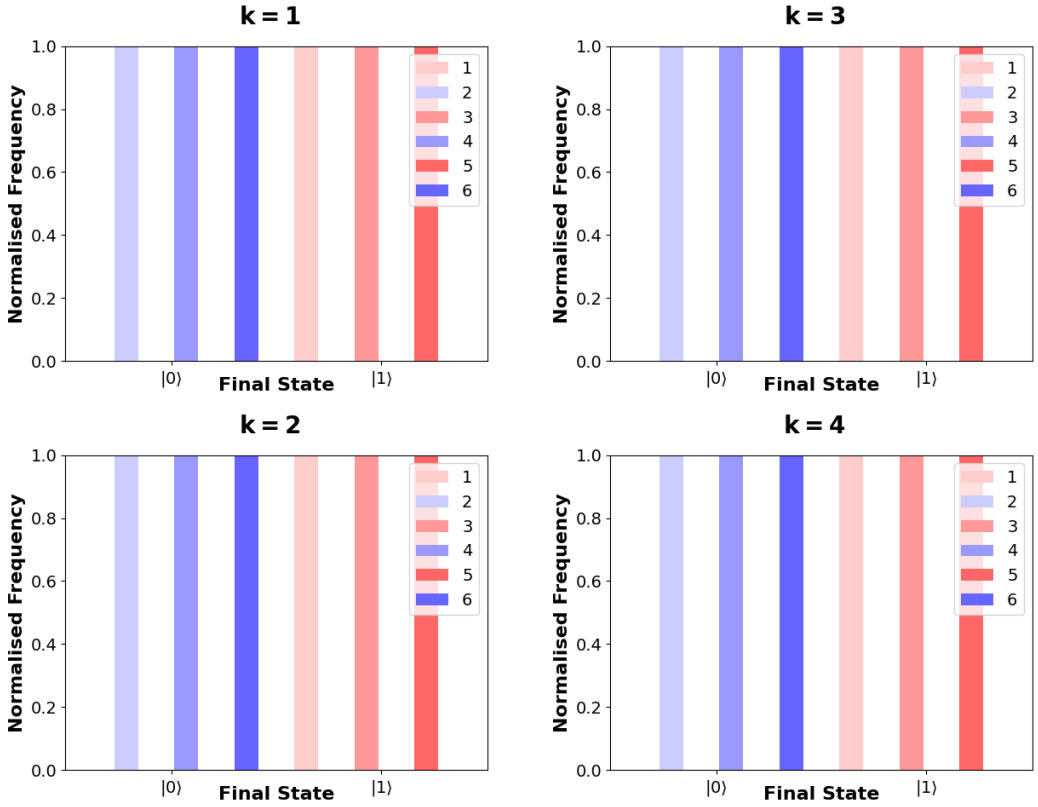

Figure 3.3: Simulated Results of EVENODD in Strawberry Fields

Figure 3.3 shows the results of running the described circuit 4096 times using the Strawberry Fields Fock simulator. It can be observed that even inputs for the parameter j

result in simulations concluding in the $|0\rangle$ accepting state with 100% frequency, while odd j inputs always result in termination in the $|1\rangle$ (rejecting) state. This aligns with the prediction of Say & Yakaryılmaz [2] but, being a simulation, does not take into account noise introduced by actual quantum hardware.

3.6 Implementation 2: IBM Qiskit

The second implementation of the EVENODD problem was created using IBM's Qiskit platform. Based around a superconducting qubit paradigm rather than photonics, this system uses the concept of circuits, which need to be instantiated with arguments defining how many qubits and how many classical registers are required. These classical registers can be used to store the classical value obtained from measuring one of the qubits. In this instance, only one qubit and one classical bit are required:

```
circuit = QuantumCircuit(1, 1)
if simulation:
    backend = Aer.get_backend('qasm_simulator')
else:
    provider = IBMQ.get_provider(group='open', project='main')
    backend = provider.get_backend('ibmq_16_melbourne')
```

Because Qiskit supports access to actual quantum hardware it was necessary to be able to dynamically select the back end based on whether the execution was being conducted as a simulation. Here this resulted in a choice between Qiskit's default QASM simulator, and a sixteen qubit quantum computer based in Melbourne. As in the Strawberry Fields implementation, reconstruction of the input string and the derivation of θ was required:

```
input_string = "a" * (j * 2) ** k
theta = get_theta(k)
```

Qiskit provides a native y-axis rotation gate, so the repeated application of the rotations to the qubit was far simpler. Note that because of the particular implementation of the ry gate in Qiskit it was necessary to explicitly double the value of θ to achieve the desired behaviour:

```
for _ in input_string:
    circuit.ry(theta * 2, 0)
```

Finally, the experiment is run on the circuit and repeated in line with the value of the parameter shots, each time the measured value of the qubit being stored in the available classical register. These results are then collated and normalised:

```
circuit.measure(0, 0)
job = execute(circuit, backend, shots=shots)
result = job.result()
counts = result.get_counts(circuit)
states = ['0', '1']
freqs = [counts.get(state, 0) / shots for state in states]
```

The full implementation of this solution can be found in Appendix C.

3.6.1 Results

IBM Qiskit was the only identified quantum computing platform that provided ready access to actual quantum hardware. Through their IBM Quantum experience (and locally run simulators) it was possible to gather experimental data that would give some insight into the effects that the noisiness of quantum hardware would have on the utility of QFA.

Simulated Results

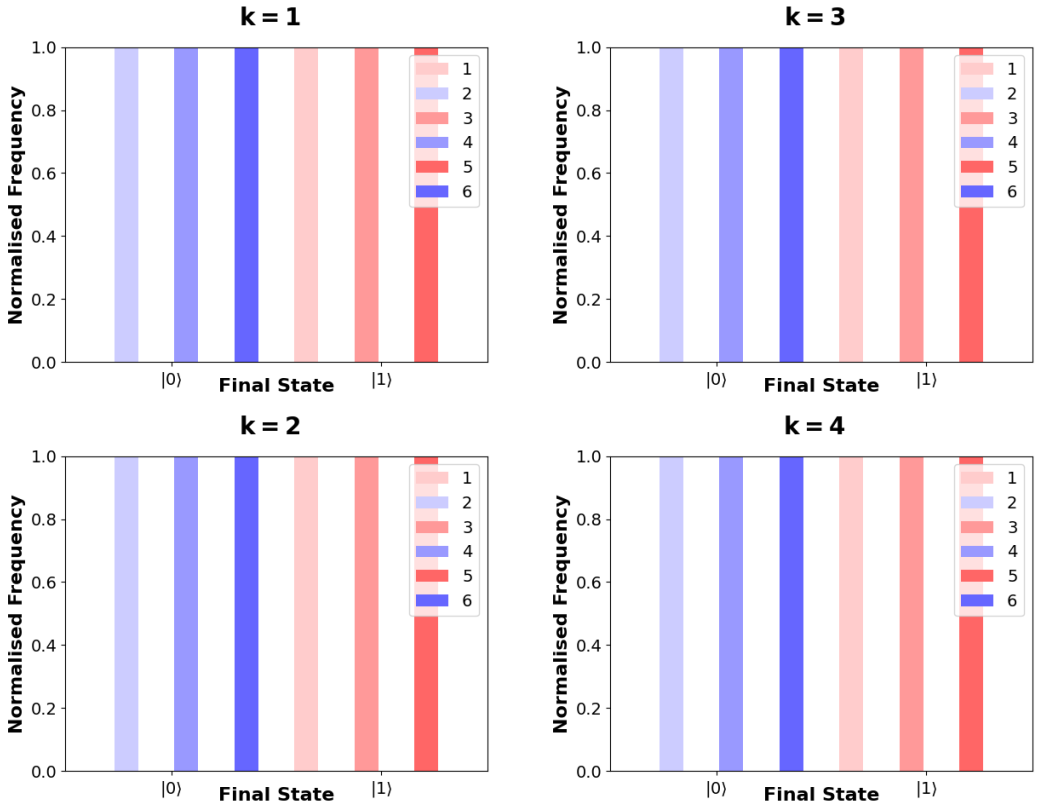

Figure 3.4: Simulated Results of EVENODD in IBM Qiskit

Simulations were run locally using Qiskit's QASM simulator over 4096 shots. It is clear from Figure 3.4 that the behaviour is identical to that of the Strawberry Fields implementation. At the conclusion of each shot, when j is even, the qubit is consistently in the $|0\rangle$ state, and when j is odd, it always resolves to the $|1\rangle$ state.

Experimental Results

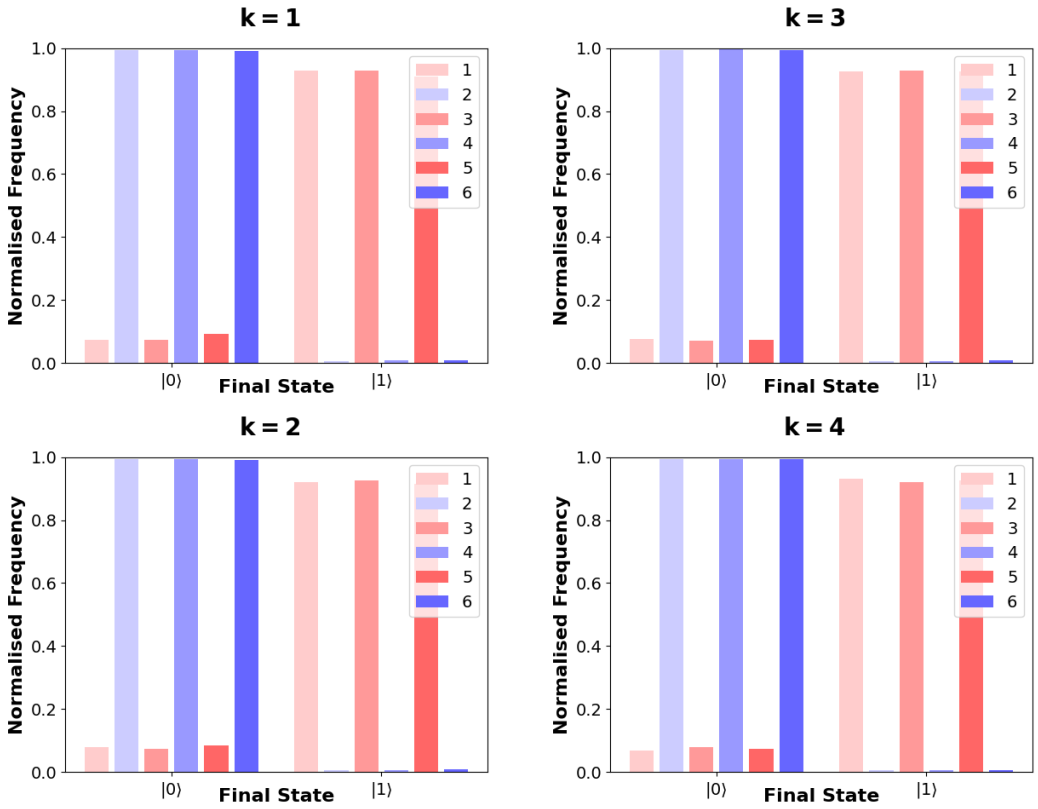

Figure 3.5: Experimental Results of EVENODD in IBM Qiskit

The experimental data visible in Figure 3.5 shows a deviation from the exact output expected by Say & Yakaryılmaz. While even j inputs are typically correctly recognised, roughly six per cent of experiments on odd j values result in an incorrect determination of acceptance (i.e. on measurement they collapsed to $|0\rangle$ instead of $|1\rangle$). Table 3.1 shows the normalised frequencies of running the experiment 4096 times on IBM's Melbourne-based, 16-qubit quantum computer.

Table 3.1: Normalised Frequencies from Running EVENODD on IBM Qiskit's Melbourne Node ($n = 4096$)

k	**j**	**Length**	**Final State (Frequency)**	
			$\|0\rangle$	$\|1\rangle$
1	1	2	0.053	0.947
1	2	4	0.993	0.007
1	3	6	0.059	0.941
1	4	8	0.994	0.006
1	5	10	0.054	0.946
1	6	12	0.992	0.008
2	1	4	0.055	0.945
2	2	16	0.992	0.008
2	3	36	0.057	0.943
2	4	64	0.99	0.01
2	5	100	0.056	0.944
2	6	144	0.995	0.005
3	1	8	0.056	0.944
3	2	64	0.993	0.007
3	3	216	0.052	0.948
3	4	512	0.996	0.004
3	5	1000	0.054	0.946
3	6	1728	0.996	0.004
4	1	16	0.058	0.942
4	2	256	0.993	0.007
4	3	1296	0.067	0.933
4	4	4096	0.994	0.006
4	5	10000	0.063	0.937
4	6	28736	0.994	0.006

3.7 Implementation 3: Rigetti Forest

The third implementation was created on Rigetti's Forest platform, also based on a superconducting qubit paradigm. This platform is quite similar to Qiskit, but applies quantum gates to a program by overloading the addition operator to append them. Forest also differs in that it requires both quantum computer (configured here for single qubit operation) and program objects for normal operation. Like Qiskit it also relies on a defined set of quantum registers (here named output), of which only one was required for this experiment:

```
qc = get_qc('1q-qvm')
```

```
program = Program()
output = program.declare('ro', 'BIT', 1)
```

As usual, we derive θ and the input string:

```
theta = get_theta(k)
input_string = "a" * j * (2 ** k)
```

Repeated application of the y-axis rotation are achieved by utilising the built in gate RY. As in Qiskit, theta has to be explicitly doubled in this implementation. A measurement gate to determine the state of the qubit after the rotations have been applied has also been added. It measures the first qubit (0 and stores the result in the first classical register):

```
for _ in input_string:
    program += RY(theta * 2, 0)
program += MEASURE(0, output[0])
```

Forest contains a utility function, wrap_in_numshots_loop, used to replicate the experiment multiple times and aggregate the results. Forest also requires an explicit compilation step, applied after the replication. The compiled executable can then be run on the predefined quantum computer. The results are then extracted and normalised before being returned.

```
program.wrap_in_numshots_loop(shots)
executable = qc.compile(program)
result = qc.run(executable)
totals = result.flatten().tolist()
freqs = [totals.count(0) / shots, totals.count(1) / shots]
```

The full implementation of this solution can be found in Appendix D.

3.7.1 Results

Hardware compatible with the Rigetti Forest platform was initially inaccessible, with only local simulators available to run circuits on. Access was finally granted through Amazon's Braket service late in 2020, but the overhead involved in rewriting the experiment using the Braket Python library made it infeasible to achieve experimental results for this implementation.

Simulations were run using the same execution harness over 4096 sample shots. As with the other simulated results, these simulations ($n = 4096$) map precisely to the distributions predicted by Say & Yakaryılmaz [2] visible in Figures 3.3 and 3.4. All samples for even j values were correctly measured in the $|0\rangle$ accepting state and all odd j were collapsed to the $|1\rangle$ state.

3.8 Implementation 4: Google Cirq

The final implementation of EVENODD was written using Google's unofficial Cirq library. This required the instantiation of the single qubit required for the experiment, a circuit, and a simulator in which to run the experiment:

```
q0 = NamedQubit('source')
simulator = cirq.Simulator()
circuit = cirq.Circuit()
```

Theta (θ) and the input string also need to be derived, as in the other implementations:

```
input_string = "a" * j * (2 ** k)
theta = get_theta(k)
```

Each rotation is added as one of the built in `ry` gates, targeted at the predefined `q0`. As in Qiskit and Forest, θ has been explicitly doubled:

```
for _ in input_string:
    circuit.append([ry(theta * 2).on(q0)])

circuit.append(measure_each(q0))
```

With the circuit complete, it can be run in the simulator, the results captured, normalised, and returned.

```
results = simulator.run(circuit, repetitions=shots)
results = sum(results.measurements.values()).tolist()
freqs = [results.count([0]) / shots, results.count([1]) / shots]
```

The full implementation of this solution can be found in Appendix E.

3.8.1 Results

Access to Google hardware on which to run Cirq experiments is heavily restricted and it was not possible to obtain experimental results for the platform, however results from locally run simulations were obtained.

These results, simulated using Cirq over 4096 shots, exactly match those of the other three implementations and the expectations of Say & Yakaryılmaz [2] which can be seen in Figures 3.3 and 3.4. All samples on even `j` values collapse to $|0\rangle$ and all odd `j` values resolve to $|1\rangle$.

Chapter 4

Study 2: Non-Regular Languages

Another theorised benefit of quantum finite automata put forward by Say & Yakaryılmaz [2] builds on their earlier work in determining which languages can be recognised using both QFA and their non-deterministic analogue, NQFA [14]. Their work has identified a number of interesting classes of languages recognisable, an example of which was put forward in their 2014 paper. This study aims to implement an example of an NQFA capable of recognising the NEQ language, again across all four quantum software platforms.

4.1 Non-deterministic Quantum Finite Automata

Like classical, deterministic finite state automata, quantum finite automata have a non-deterministic counterpart: the non-deterministic quantum finite automata (NQFA). In classical automata theory non-deterministic automata (and their generalisation, probabilistic finite automata) are termed $2fa$ because (in addition to being able to have probabilistically weighted transitions) they have more options when consuming input: they can consume input normally, or they can read input without consuming it, they can even skip over input without acting on it [15][14]. The terminology comes from their freedom to move in both 'directions' (or "two ways") along in the input (in the context of an analogy of a reading 'head' moving along a read-only tape like that on a classical Turing Machine) [9]. In non-deterministic quantum finite automata the characteristic non-determinism comes from a different mechanism. Like their classical probabilistic counterparts NQFA are defined by an input variable known as a "cut point" or an observed, normalised frequency (often incorrectly defined as a probability) $\eta \in [0, 1)$ of concluding in a given state cite [32]. If the measured frequency (over some number of experimental repetitions) is over the defined cut point η for a given input string then the string is defined as being part of the language. Where NQFA differ is that their cut point is always set at 0 [14]. What this means is that if, for example, $|1\rangle$ is our accepting state, any meaningful frequency of results appearing in state $|1\rangle$ (when noise has been accounted for) will result in the input tested being accepted. These NQFA, or $2qfa$, are defined by the following formalism:

$$M = (Q, \Sigma, \delta, q_0, Q_{acc}, Q_{rej}) \tag{4.1}$$

Where:

- Q: is a finite set of states.
- Σ: is the finite input alphabet.
- δ: is the transition function.
- $q_0 \in Q$: is the initial state of the system.
- $Q_{acc} \subset Q$: are the halting (accepting) states of the system.
- $Q_{rej} \subset Q$: are the non-halting (rejecting) states of the system, $Q_{acc} \cap Q_{rej} = \emptyset$. [9]

In the context of this study, the transitions δ will consist of rotations around the y-axis of the Bloch sphere, as in Study 1.

4.2 The Importance of Non-Regular Languages

Classical non-deterministic finite state automata, while useful as an abstraction for certain problems, are computationally no more powerful (or capable) for language recognition than their deterministic counterparts [2]. In fact, all classical, non-deterministic finite automata can be rewritten in a deterministic form [15]. The implication of this equivalency is that both DFA and NFA (including probabilistic generalisations) are strictly limited to the recognition of regular languages [2].

Quantum finite automata (specifically the non-deterministic variety) on the other hand, possess the ability to recognise the set of all regular languages as well as a subset of non-regular languages. This superset is known as the set of exclusive stochastic languages (or $S^{\neq}$). Figure 4.1 shows the relationships between $S^{\neq}$, and regular and non-regular languages.

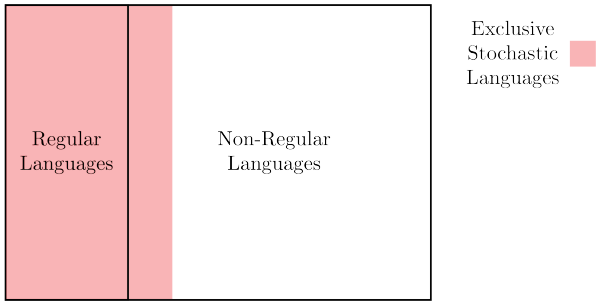

Figure 4.1: Relation of Exclusive Stochastic Languages to Regular and Non-Regular Languages [14]

This is ability to recognise non-regular languages is important because regular languages compose only a tiny fraction of the types of extant languages. Work by Chomsky and others over the past several decades have identified various classes of languages, of which regular languages are the smallest. Figure 4.2 demonstrates the relationship between these languages, while Table X maps each class of language to the type of automata required for recognising that class. Note that as the class grows in size, the sophistication of the required automata also grows.

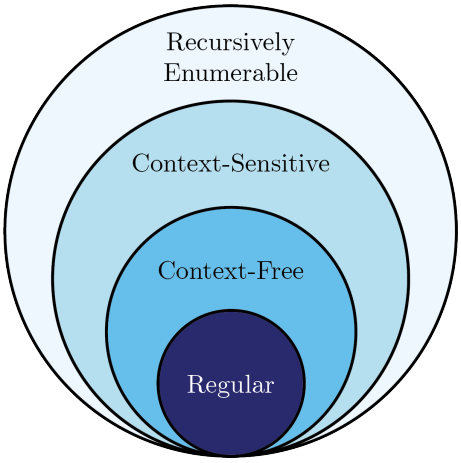

Figure 4.2: Chomsky's Hierarchy of Languages [15]

Table 4.1: Chomsky's Language Hierarchy and Respective Recognising Automata [15]

Grammar	**Language Class**	**Automata Required**
Type-0	Recursively Enumerable	Turing Machine
Type-1	Context-Sensitive	Linear-Bounded Non-Deterministic Turing Machine
Type-2	Context-Free	Non-Deterministic Pushdown Automata
Type-3	Regular	Finite State Automaton

The concept of non-regular languages is difficult to communicate to non-computer-scientists because not only they are defined in the negative, they are defined in the negative of something that is also circularly defined. Regular languages are defined as those recognisable by an FSA, and non-regular languages comprise all other languages as defined in Table 4.1 and Figure 4.2, above. To make this clearer, consider the following non-regular language:

$$L = \{a^n b^n | n \in \mathbb{N}\} \tag{4.2}$$

Each word in the language is defined as some number (n) of a characters, followed by the same number of b characters. Any word where the number of a and b characters do not match, or containing characters other than a or b is considered not to be a part of the

language. This language is not recognisable by a classical FSA because the potential size of n is unbounded (as would the number of states required in an FSA designed to recognise the language, a contradiction of the 'finite' requirement of all FSA), whereas by definition the number of states in any FSA must be finite, or bounded. Classical solutions to the problem of recognising this language require the use of a more sophisticated automata, a pushdown automata, as indicated in Table 4.1.

Non-regular languages are not limited to trivial examples like that described above, in fact many vitally important features of our every day lives, though we may not realise. One real-world example is the Hyptertext Markup Language, or HTML. HTML uses series of matching tags (e.g. and) to define the content of web pages, each tag (with a few rare exceptions) must be matched or the element it defines (or even the whole web page) will not render. This critically important language matches the format of the previous example (though multiple tag types mean it is far more sophisticated in actuality):

$$\{\text{<div>}^n\text{</div>}^n | n \in \mathbb{N}\} \tag{4.3}$$

While only an example, this illustrates the potential value and applicability of new ways of recognising non-regular languages.

4.3 The NEQ Language

To demonstrate this power of non-deterministic QFA Say & Yakaryılmaz [2] proposed building an NQFA to recognise the exclusive stochastic (non-regular) NEQ language, which is defined as:

$$\text{NEQ} = \{.w | w|_a \neq |w|_b\} \tag{4.4}$$

This results in a language comprised of all strings where the number of a characters differs to that of the number of b characters. For example the string aab $\in$ NEQ, whereas aabb $\notin$ NEQ [2].

Because of the limitations of classical finite automata outlined in Section 4.2, this language cannot be recognised without introducing more sophisticated automata (e.g. a pushdown automata). This study aims to show that NQFA do not suffer from this limitation.

4.4 Implementing NEQ in Quantum Software

As in the EVENODD problem addressed in Study 1, Say & Yakaryılmaz [2] propose a single qubit system with unitary rotations conducted around the y-axis (see Figure 3.2). Where it differs is that the angle of the rotation is no longer dependent on the language

parameters, but depends on the input consumed. On reading an a character, the rotation occurs in the clockwise direction:

$$\theta = \sqrt{2}\pi \tag{4.5}$$

While when the b character is observed the rotation is made counterclockwise:

$$\theta = -\sqrt{2}\pi \tag{4.6}$$

These rotations use the same unitary rotational operator as Study 1, which rotates θ radians about the y-axis:

$$U_\theta = \begin{pmatrix} cos\,\theta & -sin\,\theta \\ sin\,\theta & cos\,\theta \end{pmatrix} \tag{4.7}$$

By utilising the irrational $\sqrt{2}$ multiple from equations 4.5 and 4.6 the periodic function (or rotation about the y-axis) is altered and two things critical to this study are assured:

- Any b-rotation will exactly cancel out the effect of any a-rotation and vice versa.
- No amount of a-rotations or b-rotations will ever result in the automata returning to it's exact starting point. ($|0\rangle$). [2]

This study differs from that evaluating the EVENODD promise problem in that it neither requires, nor expects, a high proportion of shots to conclude in the accepting state $|1\rangle$. In fact, *any* recognisable proportion (allowing for systematic noise) of results measured as $|1\rangle$ will constitute grounds for acceptance of the input string as being part of the language. This full reliance on the probabilities (as a result of the cut point being 0), or rather frequencies, is what defines this QFA as non-deterministic [2].

As in Study 1, to reduce duplication of code across the various implementations, an execution harness was developed to control the operation of each each implementation: gathering the results of repeated executions, collating them and presenting them in graphical form. The full code of the NEQ execution harness is provided in Appendix F. This harness takes a single parameter in the form of the implementation function. Each implementation function, named run_experiment, takes the following parameters:

- a: the number of a characters in the input string.
- b: the number of b characters in the input string.
- simulation: whether to run the experiment as a simulation, or on real quantum hardware (where available). This parameter is typically pre-populated using Python's partial evaluation utilities so it can be controlled at the same execution site

- shots: the number of time to run the experiment in the configuration defined by the other parameters.

Each implementation consists of three components, setup of the quantum circuits and qubits, an operational phase where rotations (whose magnitude is fixed, but direction is determined by the input character consumed) around the y-axis are added to the circuit, and a final stage where the experiment is run (repeatedly, in accordance with the magnitude of the shots parameter), each time having the state of the qubit examined, the results collated, and those collated results returned to the harness for further processing.

4.5 Implementation 1: Strawberry Fields

The Strawberry Fields implementation of NEQ, the sole photonics-based platform, was the first to be implemented. Two nodes (or channels) and a simulator based on Strawberry Fields' Fock engine were instantiated:

```
program = sf.Program(2)
engine = sf.Engine("fock", backend_options={"cutoff_dim": 2})
```

Theta (θ) and the input string are both derived from functions defined in the NEQ common utilities: θ being the fixed value $\sqrt{2}\pi$ and input_string being generated in a pseudo-random order from a a characters and b b characters to more readily represent typical input configurations.

```
theta = get_theta()
input_string, length = get_input(a, b)
```

Using Strawberry Fields' custom program context, the two previously initiated channels are instantiated. As in Section 3.5, the first channel is initiated so as to release a single photon at the start of the experiment, while the second channel is initialised as empty (a vacuum). Then the channels are populated with a series of gates based on the contents of the input_string. As discussed in Section 3.5, the Strawberry Fields API does not include a native y-axis rotational gate, as such we use beam splitter gates (BSgate) with an angle equivalent to θ or $-\theta$ dependent on whether the consumed input character was a or b. This will result in a superposition equivalent to that achieved by a simple y-axis rotational gate:

```
with program.context as q:
    Fock(1) | q[0]
    Vacuum() | q[1]

    for char in input_string:
        if char == "a":
            BSgate(theta) | (q[0], q[1])
        elif char == "b":
            BSgate(-theta) | (q[0], q[1])
```

```
        else:
            raise NotImplementedError
```

Finally, the circuit is run repeatedly ($n = 4096$) using the Fock engine. A photon detected in the first channel corresponds to a recognised result of $|0\rangle$ while a photon detected in the second channel is interpreted as $|1\rangle$.

```
result = engine.run(program, shots=shots)
state = result.state
freqs = [round(abs(state.mean_photon(0)[0]), 4),
         round(abs(state.mean_photon(1)[0]), 4)]
```

4.5.1 Results

This implementation suffered from similar limitations to Study 1 in that results-gathering was limited to Strawberry Fields' Fock simulator due to the current inability of the Strawberry Fields hardware to release single photons. Interpreting the simulated results that could be gathered differs quite substantially from Study 1. Say & Yakaryılmaz [2] do not provide expected frequencies to verify against, instead the non-deterministic QFA used in this study operates on the principle that *any* non-zero frequency in $|1\rangle$ () when $|w|_a \neq |w|_b$. From this we can infer that when $|w|_a = |w|_b$ we should expect all shots to resolve to $|1\rangle$.

Simulated Results

The results of running the NEQ circuit on Strawberry fields showed exact alignment with the inferences made above based on the predictions of Say & Yakaryılmaz [2]. The actual magnitudes of the frequencies are determined by the final state of the single qubit system when the measurement operation occurs, the only requirement is that they be non-zero. Figure 4.3 shows the results of the simulations running over all permutations of the input parameters $a \in \{1, 2, 3, 4\}$ and $b \in \{1, 2, 3, 4, 5, 6\}$.

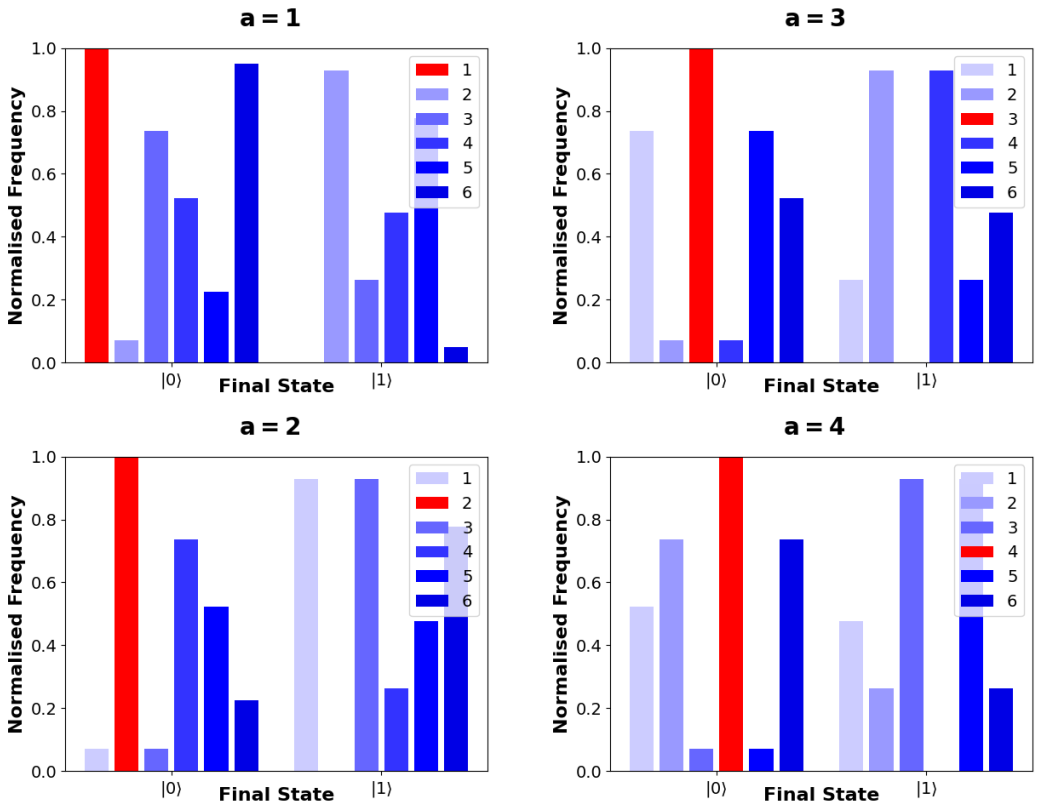

Figure 4.3: Simulated Results of NEQ in Strawberry Fields

4.6 Implementation 2: IBM Qiskit

The second implementation of the NEQ experiment was developed using IBM Qiskit so as to allow for a platform capable of providing both simulations and experimental results. This study also involved a single qubit system and required a single classical register in which the results of the measurement operation could be stored (QuantumCircuit(1, 1)). The Qiskit backend was again configured to be dynamically allocated based on whether simulated or experimental results were sought:

```
circuit = QuantumCircuit(1, 1)
if simulation:
    backend = Aer.get_backend('qasm_simulator')
else:
    provider = IBMQ.get_provider(group='open',
                                 project='main')
    backend = provider.get_backend('ibmq_16_melbourne')
```

The values of θ and the input string were gathered using utility functions as described in Section 4.5. The get_input() utility function has been reformulated in this study to also return the length of the string, which is used for logging purposes:

```
theta = get_theta()
input_string, length = get_input(a, b)
```

With the input string computed, it can be iterated over and examined. On each iteration of the loop, the character (char) at that position is evaluated. If char is a, a Qiskit ry, or y-axis rotation, gate is applied with a positive value of θ. When char is b, the same gate is applied but with a negative θ value. Any other values are deemed to be incompatible and will case the program to stop. Note again that due to differences in implementation, θ values are doubled when provided to this particular Qiskit gate. The final gate to be added is the measurement operation, assessing the qubit (causing its collapse) and storing the value in the first (and only) classical register.

```
for char in input_string:
    if char == "a":
        circuit.ry(theta * 2, 0)
    elif char == "b":
        circuit.ry(-theta * 2, 0)
    else:
        raise NotImplementedError

circuit.measure(0, 0)
```

To see the results of this circuit a job is constructed to execute the circuit on the predetermined backend and repeated $n = shots$ times. The results of these repetitions are extracted from the completed job, collated, and returned.

```
job = execute(circuit, backend, shots=shots)
result = job.result()
counts = result.get_counts(circuit)
states = ['0', '1']
freqs = [counts.get(state, 0) / shots for state in states]
```

4.6.1 Results

As with Study 1, the relative accessibility of IBM's Quantum Experience platform meant that both simulated and experimental (hardware) results were available. This provided a basis for comparison that proved vital in assessing whether the noise present in current generation general-purpose quantum computers would impact the efficacy of software-defined QFAs in the near term.

Simulated Results

As can be observed in Figure 4.4, the results of running NEQ on IBMs classical (QASM) simulator showed results identical to that of the Strawberry Fields implementation described earlier in Section 4.5. Observe that once again when the number of as and bs align, all shots are observed as being rejected (in the $|0\rangle$ state) and that no unequal pairing has zero frequency in the $|1\rangle$ state.

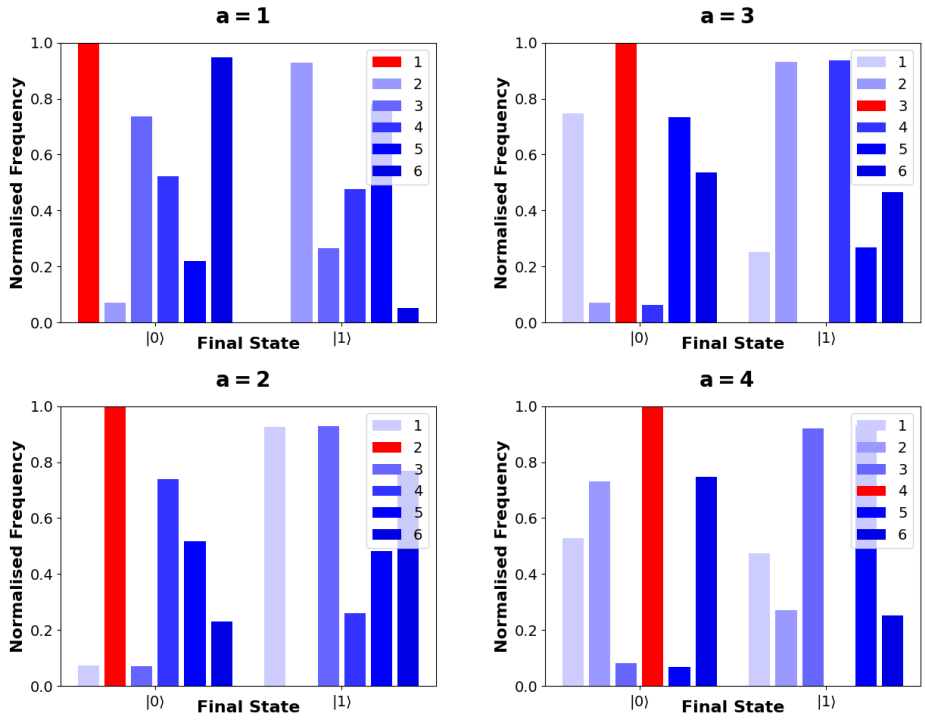

Figure 4.4: Simulated Results of NEQ in IBM Qiskit

Experimental Results

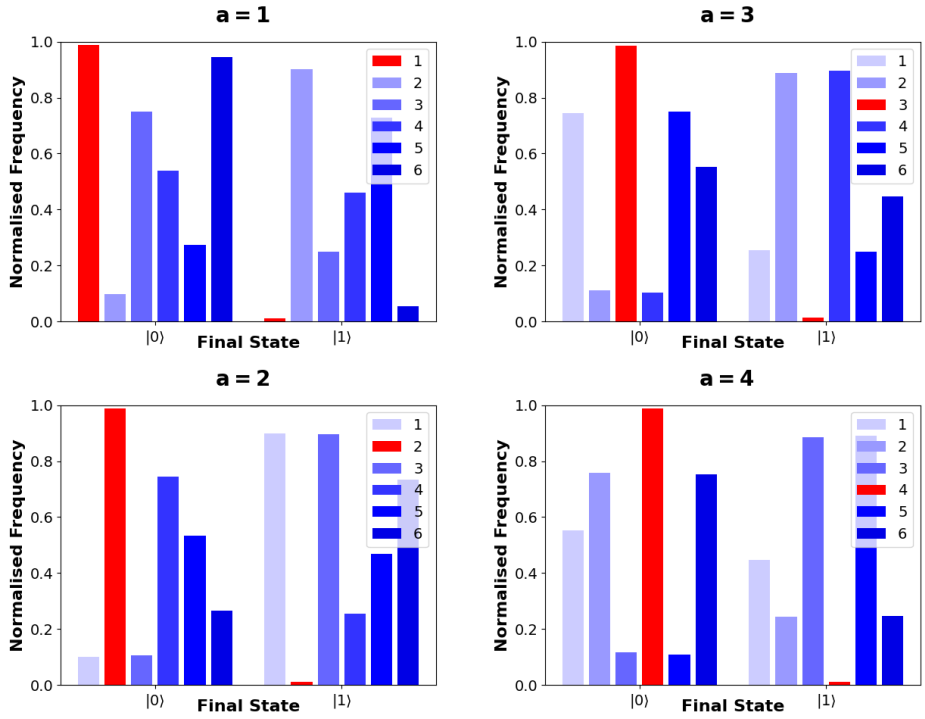

Figure 4.5: Experimental Results of NEQ in IBM Qiskit

In addition to the simulated results explained above, experimental results were also gathered over the same combinations of input parameters ($a \in \{1, 2, 3, 4\}$ and $b \in \{1, 2, 3, 4, 5, 6\}$) as the Strawberry Fields implementation. The experiment was repeated 4096 times in each configuration and results have been provided in Table 4.2 and are represented graphically in Figure 4.5.

Table 4.2: Normalised Frequencies from Running NEQ on IBM Qiskit's Melbourne Node ($n = 4096$)

a	**b**	**Length**	**Final State (Frequency)**	
			$\|0\rangle$	$\|1\rangle$
1	1	2	0.991	0.009
1	2	3	0.137	0.863
1	3	4	0.748	0.252
1	4	5	0.553	0.447
1	5	6	0.280	0.720
1	6	7	0.952	0.048
2	1	3	0.141	0.859
2	2	4	0.992	0.008
2	3	5	0.133	0.867
2	4	6	0.735	0.265
2	5	7	0.550	0.450
2	6	8	0.271	0.729
3	1	4	0.749	0.251
3	2	5	0.134	0.866
3	3	6	0.993	0.007
3	4	7	0.128	0.872
3	5	8	0.735	0.265
3	6	9	0.551	0.449
4	1	5	0.564	0.436
4	2	6	0.760	0.240
4	3	7	0.137	0.863
4	4	8	0.994	0.006
4	5	9	0.141	0.859
4	6	10	0.744	0.256

In interpreting these results there are a number of interesting features to observe that have been introduced as a result of noise present in the experiment that was entirely lacking in the simulations. The first and most impactful issue is that in configurations where $|w|_a = |w|_b$, there now exist shots collapsing to a $|1\rangle$ state, albeit less than 1% of cases. Despite these anomalies it is still clear from visual inspection which cases in each

configuration should not be included in NEQ. The other, potentially less serious issue introduced by this noise is the disparity between the frequencies reported by the simulation for cases when $|w|_a \neq |w|_b$ and those derived experimentally. The simulated results express the frequencies under perfect conditions, so any deviation can be interpreted as the result of noise from some source. While these deviations would not change the result of determinations made under these circumstances, it does raise the prospect of it contributing to incorrect determinations made under alternative configurations or for applications to different problems.

4.7 Implementation 3: Rigetti Forest

The third implementation of the NEQ problem was constructed using Rigetti Forest. To initialise the experiment, a single-qubit quantum computer (qc) and an empty quantum program are created along with a single classical register (output):

```
qc = get_qc('1q-qvm')
program = Program()
output = program.declare('ro', 'BIT', 1)
```

The value for θ ($\sqrt{2}\pi$) is obtained from the NEQ common utilities and the pseudo-random permutation of the input string is generated:

```
theta = get_theta()
input_string, length = get_input(a, b)
```

The input string is iterated over, each character triggering a y-axis rotation of either $\sqrt{2}\pi$ or $-\sqrt{2}\pi$ for a or b, respectively. This is accomplished using the Foresk SDK's natively supported y-rotation gate. The last gate to be added represents a single measurement operation placing the measured state of the system's qubit into the classical register:

```
for char in input_string:
    if char == "a":
        program += RY(theta * 2, 0)
    elif char == "b":
        program += RY(-theta * 2, 0)
    else:
        raise NotImplementedError

program += MEASURE(0, output[0])
```

With the program complete it can be configured to execute repeatedly in line with the value of shots. The program is then executed, the resulting values unpacked, normalised and returned:

```
program.wrap_in_numshots_loop(shots)
executable = qc.compile(program)
result = qc.run(executable)
```

```
totals = result.flatten().tolist()
freqs = [totals.count(0) / shots, totals.count(1) / shots]
```

4.7.1 Results

As explained in Section 3.7.1, delays in gaining access to Rigetti hardware made it infeasible to conduct hardware-based experiments. Simulations were conducted, using the execution harness, over all possible permutations of the input variables $a \in \{1, 2, 3, 4\}$ and $b \in \{1, 2, 3, 4, 5, 6\}$. Experiments in each configuration were conducted 4096 times. The results exactly correspond to those found in the NEQ simulations conducted on Strawberry Fields and Qiskit, visible in Figures 4.3 and 4.4, respectively.

4.8 Implementation 4: Google Cirq

The last NEQ implementation was created using Google Cirq. Setting up the single qubit system suggested by Say & Yakaryılmaz [2] involved initialising a qubit q0, preparing a simulator, and creating an empty circuit to populate with gates:

```
q0 = NamedQubit('source')
simulator = cirq.Simulator()
circuit = cirq.Circuit()
```

As in the other implementations, θ derived from the predifined function from the problem's common utilities. The input string is constructed pseudo-randomly in line with the a and b parameters:

```
theta = get_theta()
input_string, length = get_input(a, b)
```

For each character in the input_string, add a y-rotation gate operating on the qubit q0, with either θ or $-\theta$ depending on whether the character in question is a or b. After all of the rotations have been added to the circuit, a measurement operation, storing the result of measuring q0, is added as the final gate:

```
for char in input_string:
    if char == "a":
        circuit.append([ry(theta * 2).on(q0)])
    elif char == "b":
        circuit.append([ry(-theta * 2).on(q0)])
    else:
        raise NotImplementedError

circuit.append(measure_each(q0))
```

Finally, the simulation is run repeatedly, shots times, before the measured values are unpacked, collated, normalised and returned.

```
measurements = simulator.run(circuit, repetitions=shots)
results = sum(measurements.measurements.values()).tolist()
freqs = [results.count([0]) / shots, results.count([1]) / shots]
```

4.8.1 Results

Results for this implementation were limited to simulations for the reasons outlined in Section 3.8.1. These simulations were conducted on Cirq's classically implemented quantum simulator over the same input combinations as the other implementations ($a \in \{1, 2, 3, 4\}$ and $b \in \{1, 2, 3, 4, 5, 6\}$) and results were found to exactly correspond to those of the simulations produced by the other three implementations over the same 4096 repetitions. All samples for configurations corresponding to words not in the language ($|w|_a \neq |w|_b$) were observed to fall in state $|0\rangle$, whilst all words in the language observed non-zero frequency in the $|1\rangle$ state. These results can be observed in Figures 4.3 and 4.4.

Chapter 5

Study 3: Higher Dimensional State Reduction

Quantum computing is typically conducted with a state composed of some number of qubits (states consisting of some combination of two basis states). Both prior studies have conformed to this dynamic, however it is possible to create quantum circuits based on higher dimensions (combinations of more than two basis states are typically known as "qudits", where the dimensionality is indicated by the variable d). This study shows that the same state reduction possible in qubit-based state systems is achievable in higher dimensional systems as well.

5.1 Tian et al., 2019

This study centres around a paper published in *Nature Quantum Information* by Tain et al. in 2019 [3]. The paper described a demonstration which used a linear optical system to prove experimentally that their quantum solution could solve the prime remainder problem using only three orthonormal basis states in a single qutrit (a three-dimensional state-space, compared to the two-dimensional spaces described above for qubits), compared to the P states required by the classical solution.

5.1.1 The Prime Remainder Problem

The prime remainder problem is a classification problem that differentiates between whether the length of the provided input is a specified multiple, k, of a prime, P, or some multiple of P plus a specific remainder, R. Figure 5.1 shows the classical finite automata capable of recognising both classes of the prime remainder problem, with $P = 5, R = 1$ in five states, over the alphabet $\{0, 1\}$. The set of inputs resulting in the automata concluding in the q_0 accepting state correspond to the $k \cdot P$ input lengths $\{0, 5, 10, 15...\}$ while those concluding in the q_1 accepting state correspond to $k \cdot P + R$ include lengths $\{1, 6, 11, 16...\}$. This representation is slightly different from that required of the prime

remainder problem as FSA, by definition, can only report acceptance or rejection, not the the specific final state. The prime remainder problem's requirement to classify two related languages independently would require two FSA, identical to the one demonstrated in 5.1 but that each would only have a single accepting state q_0 or $q1$, respectively. Input strings would be run through both FSA and any not accepted by (only) one automaton would be discarded, leaving the lengths of all remaining inputs classified as either $L = k \cdot P$ or $L = k \cdot P + R$.

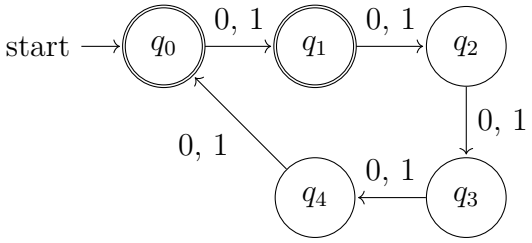

Figure 5.1: Classical FSA for the Prime Remainder Problem ($P = 5, R = 1$)

5.1.2 Achieving Advantage

Tian et al. [3] demonstrated the superiority of their quantum solution using four examples:

- A QFSA to detect input of length $3k$ ($P \cdot k, P = 5$).
- A QFSA to detect input of length $3k + 1$ ($P \cdot k + R, P = 3, R = 1$).
- A QFSA to detect input of length $5k$ ($P \cdot k, P = 5$).
- A QFSA to detect input of length $5k + 2$ ($P \cdot k + R, P = 5, R = 2$).

Classical finite state automata would have required a minimum of three and five states respectively for each configuration, with required states growing linearly with n. By contrast, their quantum solution for the same problem requited only three basis states (a single third order unit of state) and used a much simpler architecture, where the same three-dimensional unitary operation U was applied to the qubit for each input, rotating the unit vector around the Bloch sphere. If $k \cdot P + R$ rotations are effected, the vector will return to the starting $|0\rangle$ state on application of measurement and collapse, otherwise the qubit will emerge in either $|1\rangle$ or $|2\rangle$ with respective probabilities dependent on R and P. This setup (though expressed physically) maintains the $O(n)$ running time of the FSA solution, however reduces the required state from $O(n)$ to $O(1)$ [3].

5.1.3 Existing Hardware Implementation

Tian et al. [3] demonstrated their quantum advantage through a physical circuit. This circuit, observable in Figure 5.2 uses photons released from a photon emitter and travelling along three optical channels

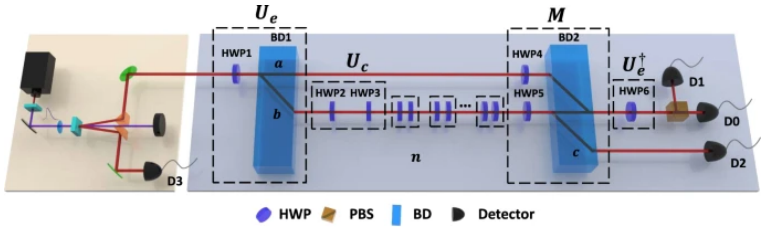

Figure 5.2: The Physical Circuit Used by Tian et al., 2019 [3]

This implementation consists of four main hardware components that would need to be replicated in software:

- A Photon Emitter: releases individual photons along a specified optical channel.
- Half-Wave Plates: alters the polarisation (state) of the photons passing through it.
- Beam Splitters (Beam Dividers): a photon enters on one channel and is placed in a superposition across two exit channels. The probability of ending up in either channel is determined by its state on entering the splitter.
- Photon Detectors: sit at the end of optical channels and determine whether or not a photon was present on the channel.

5.1.4 Reusable Building Blocks for a Software Implementation

In order to re-implement the experiment described above at the software level, it was first necessary to genericise the process of determining the appropriate angle settings of the U_e and U_c unitary operations for each configuration of prime and remainder. Angles were provided only for the prime remainder settings demonstrated by the authors, but algorithms were provided which purportedly produce angles for any prime remainder combination. These angles, arrived at via the identification of intermediate variables Q and t, were determined through the implementation of a series of solvers. The value of Q is bound by the following constraints as identified by Tian et al. [3]:

- $Q \in \mathbb{Z}^+$ and $Q < P$.
- $cos(\frac{2\pi Q}{P}R) < 0$.
- $$Q = \begin{cases} \lceil \frac{P}{4R} \rceil & R \leq \frac{P}{4} \\ 1 & \frac{P}{4} \leq R \leq \frac{3P}{4} \\ \lfloor \frac{P}{R}(j+\frac{1}{4}) \rfloor + 1, \frac{1}{4} < j\frac{P-R}{R} = \lfloor j\frac{P-R}{R} \rfloor < \frac{2}{3} & R > \frac{3P}{4} \end{cases}$$

An equation solver was implemented to derive Q from P and R in line with these constraints and can be found in Appendix K. This solver allowed for the computation of

the angles utilised in the U_e and U_c gates, the functions for deriving these angles from Q are included in Appendix L. The execution harness, as described in previous studies, has also been provided in Appendix M.

5.2 Implementation 1: Strawberry Fields

5.2.1 A Photonics Implementation in Software

With the inputs to each unitary operation readily available for any permutation of prime and remainder, the only thing left to ascertain was how to recreate those unitary operations, formerly expressed in hardware, in software. This proved somewhat easier using Strawberry Fields than in the other platforms as Strawberry Fields is built around the same photonics-based optical technology used by the Tian et al. hardware experiment [3]. This meant native support for gates like beam splitters, which were used extensively.

Before the software-defined gates could be initialised, an empty program had to be created, along with a Fock simulator in which to run it. Note here that the circuit is created with three subsystems, or channels, to match the three optical channels observed in Figure 5.2.

```
n_subsystems = 3
program = sf.Program(n_subsystems)
engine = sf.Engine("fock", backend_options={"cutoff_dim": 2})
```

The rotational angles required for the experiment and described in Section 5.1.4 were obtained from the utility functions based on the configuration parameters:

```
theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)
```

The systems channels were set up such that the second and third were initialised as a vacuum while the first is configured such that a single photon will be released at the start of the experiment:

```
with program.context as q:
    Fock(1)  | q[0]  # Setup channel 0
    Vacuum() | q[1]  # Empty channel 1
    Vacuum() | q[2]  # Empty channel 2
```

Creating the block of operations labeled by Tian et al. as U_e in software was made reasonably straightforward by two factors, first that the half wave plate identified as $HWP1$ was deemed unnecessary based on the previous set up steps. This left only BD1, which could be replicated with a single beam splitter gate (BSgate). Note that the BSgate is configured first with the angles theta_e and phi_e derived earlier before being applied to the first and second channels. All of this takes place within the program.context shown in the previous block:

```
    BSe = BSgate(theta_e, phi_e)
    BSe | (q[0], q[1])
```

The next next stage of the Tian et al. circuit is the repeated application of U_c blocks. Due to the lack of a native half-wave plate gate operation an equivalent was constructed from a single BSgate across the second and third channels using one of the derived angles, theta_c. These operations also take place in the program.context. Note that the BSgate is only defined once, but applied repeatedly in line with the length of the input:

```
    BSc = BSgate(theta_c, 0)
    for _ in range(len(input_string)):  # n applications of Uc
        BSc | (q[1], q[2])
```

The final step involved measuring and collating the results. Due to the nature of the experiment and the Strawberry Fields platform, there was no need to implement the M block of the circuit as Strawberry Fields' measures the results as part of normal operation. This removed the need to implement $HWP4$, $HWP5$ and $BD2$.

```
result = engine.run(program)
state = result.state
freqs = [state.mean_photon(i)[0] for i in range(n_subsystems)]
```

The full implementation is provided in Appendix N.

5.2.2 Results

Due to present limitations of the Strawberry Fields hardware platform it was not possible to test this implementation on actual quantum hardware and obtain experimental results. As was explained by a representative of the Xanadu directly, their focus has been on continuous variable quantum computing, and not on the development of individual, controllable qubits. In practice, their quantum chips can only initialise channels in vacuum or squeeze states, whereas this experiment requires initialisation of the first channel in a Fock state whereupon a single photon is released into the channel. While it was not possible to reformulate the experiment to make use of squeeze states, simulated results were obtained and are included below.

Simulated Results

Simulations were carried out using the Strawberry Fields Fock simulator on the same sets of configuration permutations used by Tian et al. [3]. Primes ($P \in \{3, 5\}$) and remainders ($R \in 1, 2$) were tested against inputs strictly designed to fall into either $P \cdot k$ or $P \cdot k + R$.

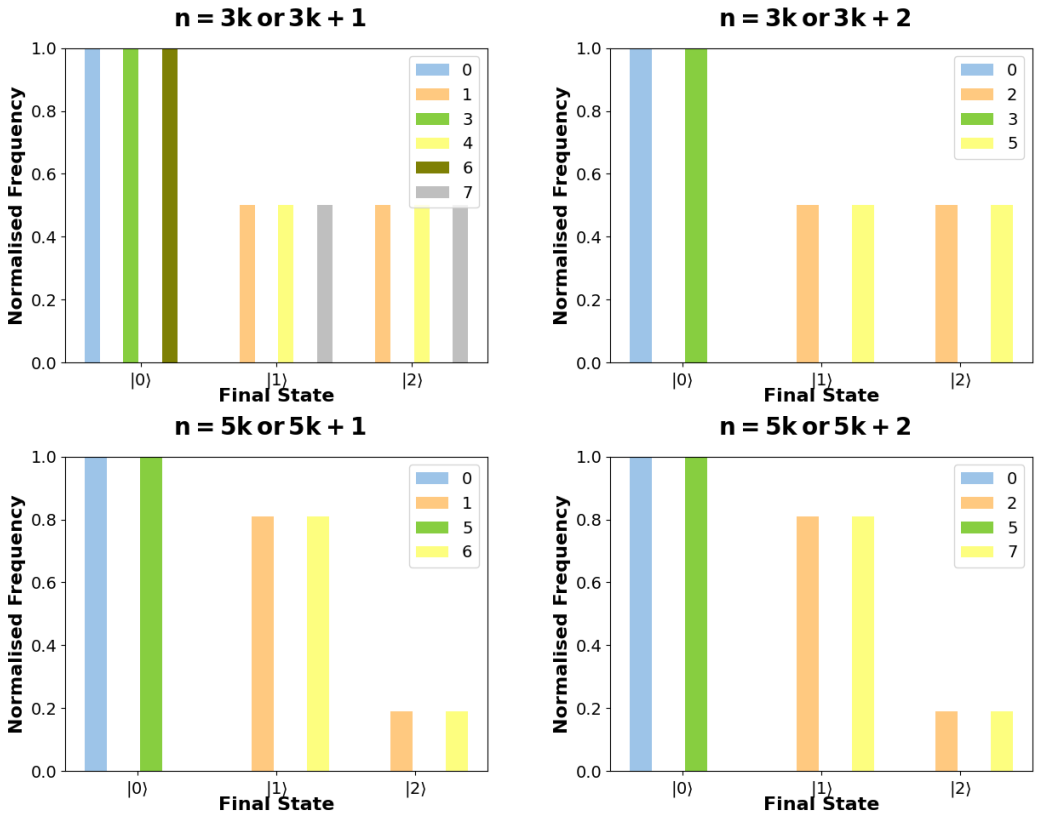

Figure 5.3: Results for Tian et al. (2019) running on a simulated photonic quantum computer

The results of these simulations, displayed in Figure 5.3, show an exact correspondence to the expectations of Tian et al., albeit without the small amount of noise present in their experimental results. Note that $P \cdot k$ length inputs (those with no remainder) appear with 100% frequency in the $|0\rangle$ state, while configurations involving $P \cdot k + R$ length inputs all resolve to either $|1\rangle$ or $|2\rangle$.

5.3 Implementation 2: IBM Qiskit

The Tian et al. experiment was also implemented in software using IBM's Qiskit SDK. Because of the relative availability of IBM hardware, this provided the unique ability to examine the differences in performance between the dedicated Tian et al. circuit and the same circuit as implemented in software on a general purpose quantum computer.

This implementation posed certain challenges as due to the change in underlying hardware paradigm, gates like beam splitters were no longer natively provided and needed to be constructed from other, more primitive gates. Another challenge was implementing a qutrit-based system on hardware that only natively supports qubit-based operation. This latter challenge was overcome by using three qubits to mimic the behaviour of a single qutrit, which retains the constant advantage of constant space requirements. Apart from the increase in qubits (and corresponding classical registers) the creation of the circuit

and back end remains unchanged from Studies 1 and 2.

```
circuit = QuantumCircuit(3, 3)

if simulation:
    backend = Aer.get_backend('qasm_simulator')
else:
    provider = IBMQ.get_provider(group='open',
                                 project='main')
    backend = provider.get_backend('ibmq_16_melbourne')
```

The angles for the half wave plates and beam splitters (or their equivalents are obtained from the study utilities, as in the Strawberry Fields implementation:

```
theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)
```

In order to initialise the first qubit in the $|1\rangle$ state (analogous to releasing a single photon in the first channel) a y-rotation of π radians is made:

```
circuit.ry(pi, 0)
```

As mentioned the lack of native beam splitters necessitated the creation of an equivalent compatible with the superconducting qubit paradigm. This was achieved by chaining three more primitive gates together. The first, a "controlled y" gate, rotates the state vector of the second qubit θ radians around the y-axis only if the first is $|1\rangle$ when evaluated. The same is done for the z-axis with the angle ϕ. Then a controlled Pauli X gate (cx) rotates the first qubit's state vector π radians around the x-axis, but only if the second qubit is $|1\rangle$ when evaluated. The net result of this is to set up a quantum superposition across the first and second channels in the same way as a beam splitter, with probabilities dependent on the values of θ and ϕ. This is utilised for creating both U_e and U_c. U_e, operating on the first and second qubits is demonstrated here:

```
def bs(circuit, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q1 is incoming whereas q2 is outgoing
    circuit.cry(theta, q0, q1)
    circuit.crz(phi, q0, q1)
    circuit.cx(q1, q0)

bs(circuit, 0, 1, theta=2 * theta_e, phi=phi_e) # Ue
```

The beam splitter equivalent gate is also used in effecting U_c, here applied repeatedly when processing each input character. Here it is applied over the second and third qubits, with different values for θ and ϕ than U_e:

```
for _ in range(len(input_string)):
    bs(circuit, 1, 2, 2 * theta_c, phi=phi_c) # Uc
```

The implementation of M and $U_e^\dagger$ operates slightly differently in that the constituent operations are applied in reverse order to that of U_e. The reason for this is evident on

observing the formation of these elements in Figure 5.2, where the constituent components similarly occur in reverse.

```
def bsd(circuit, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q0 is incoming whereas q1 is outgoing
    circuit.cx(q1, q0)
    circuit.crz(phi, q0, q1)
    circuit.cry(theta, q0, q1)

bsd(circuit, 1, 0, theta=2 * theta_e, phi=phi_e) # Ued
```

As a side effect of the $U_e^\dagger$ operations already conducted on the circuit, a portion of the results that should be attributable to state $|0\rangle$ (001) has ended up in the unlabelled state 011. The application of a controlled NOT gate acting on the second qubit (as controlled by the first) corrects this error without affecting results truly meant for the state where the second qubit should be active $|1\rangle$ (010).

```
circuit.cx(0, 1)
```

Finally the state of the three qubits is measured, the results extracted from the completed job object and normalised. Observe here that three "states" are extracted from the results. Because three qubits have been used, each with two orthonormal basis states to collapse to, there are actually nine states the system could resolve to. However only the three identified here were actually used and are relevant, though the presence of the others becomes important in interpreting experimental results.

```
circuit.measure(range(3), range(3))
shots = 4096
job = execute(circuit, backend, shots=shots)
counts = job.result().get_counts(circuit)
states = ['001', '010', '100']
freqs = [counts.get(state, 0) / shots for state in states]
```

The full implementation of Tian et al. in Qiskit is provided in Appendix O.

5.3.1 Results

Simulated Results

Simulations were run using the same sample of primes and remainders as the original Tian et al. study and the Strawberry Fields implementation. The results were also found to be consistent with these previous examples. As can be observed in Figure 5.3, the qubit-based system simulated by the Qiskit SDK has also found that all samples involving $P \cdot k$ words resolved to $|0\rangle$ while $P \cdot k + R$ samples are distributed between $|1\rangle$ and $|2\rangle$.

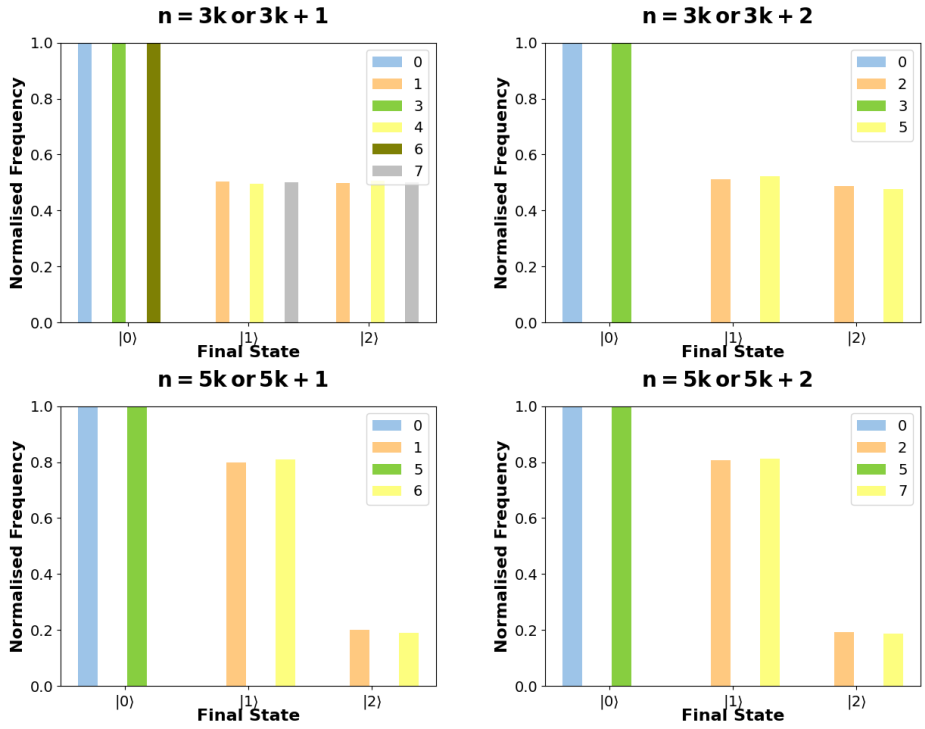

Figure 5.4: Results of Running Tian et al. (2019) on an IBM Qiskit Simulator

Experimental Results

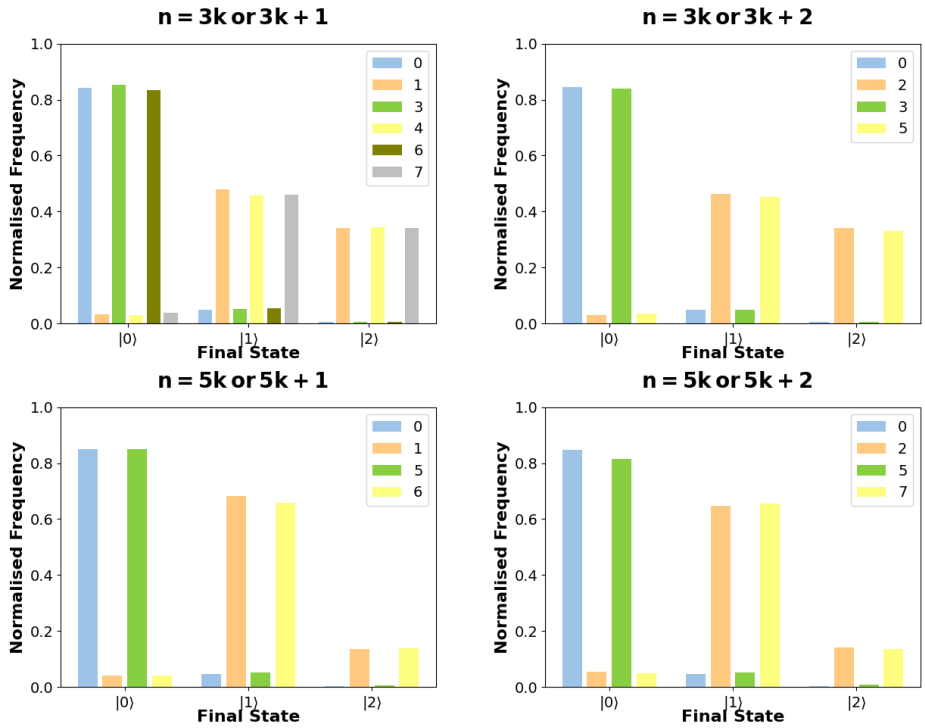

Figure 5.5: Experimental Results of Running Tian et al. (2019) on IBM Qiskit Hardware

In addition to the simulations described above, the Qiskit implementation of Tian at al. was also run on IBM's 16-qubit, Melbourne-based quantum computer with the same set of primes, remainders and input string configurations. Figure 5.5 shows the results of these experiments. Figure 5.5 shows a number of interesting elements of these results. The first and arguably most important factor is the apparent loss of samples across all configurations and inputs. In the simulated results for both Qiskit and Strawberry fields, all of the various configurations 4096 samples were distributed across the three possible states. Here though, samples appear to disappear, with the visible normalised frequencies for each configuration summing to less than 1.0. These "missing" samples are being resolved to combinations of states other than those demarcated as $|0\rangle$ (001), $|1\rangle$ (010), $|2\rangle$ (100), e.g. 111. The existence of these other possible states is a side effect of the way the qutrit has been constructed from three separate qubits. Table 5.1 shows the normalised frequencies of the experiment for each configuration, broken down by final state (including invalid states - labelled here as "Other"). Observe that the impact of these lost samples is quite severe, in the worst case more than 50% of samples resolve in invalid final states. Also present, though somewhat obscured by the missing sample issue, is the same hardware-introduced noise present in the hardware implementation of Studies 1 and 2.

5.4 Implementation 3: Rigetti Forest

The Tian et al. circuit was also realised on the Rigetti Forest platform using similar techniques to those used in the Qiskit implementation.

In contrast to the previous two studies, implementing Tian et al. required a three qubit quantum computer, along with three classical registers in which to store measurements of each qubit:

```
qc = get_qc('3q-qvm')
program = Program()
output = program.declare('ro', 'BIT', 3)
```

The angles for the various components were gathered from the common utilities as in previous implementations.

```
theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)
```

The first qubit was rotated such that it began the circuit in the $|1\rangle$ state.

```
program += RY(pi, 0)
```

Forest's syntax for implementing controlled gates was somewhat different to that of Qiskit, so utility functions were set up to apply controlled y-rotation and z-rotation gates more easily. These functions were utilised in the implementation of a beam splitter equivalent that follows the same logic as the Qiskit element created for the same purpose:

```
def cry(program, control, target, theta) -> None:
```

Table 5.1: Normalised Frequencies from Running Tian et al. (2019) on IBM Qiskit's Melbourne Node ($n = 4096$)

P	**R**	**Length**	**Final State (Frequency)**			
			$\|0\rangle$	$\|1\rangle$	$\|2\rangle$	**Other**
3	1	0	0.688	0.086	0.004	0.222
3	1	1	0.196	0.284	0.048	0.472
3	1	3	0.702	0.079	0.004	0.215
3	1	4	0.205	0.277	0.051	0.467
3	1	6	0.746	0.082	0.005	0.168
3	1	7	0.203	0.270	0.043	0.484
3	2	0	0.687	0.082	0.006	0.224
3	2	2	0.112	0.266	0.126	0.497
3	2	3	0.62	0.081	0.017	0.282
3	2	5	0.177	0.258	0.06	0.506
5	1	0	0.656	0.083	0.009	0.252
5	1	1	0.16	0.357	0.023	0.459
5	1	5	0.646	0.079	0.011	0.264
5	1	6	0.082	0.548	0.162	0.208
5	2	0	0.688	0.083	0.005	0.223
5	2	2	0.172	0.370	0.033	0.425
5	2	5	0.736	0.089	0.004	0.171
5	2	7	0.153	0.348	0.037	0.462

```
    program += RY(theta, target).controlled(control)

def crz(program, control, target, phi) -> None:
    program += RZ(phi, target).controlled(control)

def bs(program, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q1 is incoming whereas q2 is outgoing
    cry(program, q0, q1, theta)  # circuit.cry(theta, q0, q1)
    crz(program, q0, q1, phi)  # circuit.crz(phi, q0, q1)
    program += CNOT(q1, q0)  # circuit.cx(q1, q0)

bs(program, 0, 1, theta=2 * theta_e, phi=phi_e)
```

To effect the repeated U_c blocks present in the Tian circuit, it was possible to utilise a simple y-axis rotation gate, controlled by the presence of a measured $|1\rangle$ state in the third qubit, in combination with a CNOT gate acting on the second channel based on the state of the third.

```
for _ in range(length):
```

```
    cry(program, 1, 2, theta_c * 2)
program += CNOT(2, 1)
```

$BS\dagger$ was again implemented by reversing the order of operations of the BS gate:

```
def bsd(program, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q1 is incoming whereas q2 is outgoing
    program += CNOT(q1, q0)
    crz(program, q0, q1, phi)
    cry(program, q0, q1, theta)

bsd(program, 1, 0, theta=2 * theta_e, phi=phi_e)
```

The reversion of the incorrectly flipped bit, explained in Section 5.3 is again corrected with a simple CNOT gate controlled on the first qubit and effected on the second.

```
program += CNOT(0, 1)
```

Each of the three qubits are measured, with the results stored in the the classical registers:

```
program += MEASURE(0, output[0])
program += MEASURE(1, output[1])
program += MEASURE(2, output[2])
```

The Forest utility function wrap_in_numshots_loop is again called to ensure repetitions in line with the shots parameter passed to the function. With the repetitions in place, the program is compiled, run and the results are extracted, normlised, and returned.

```
program.wrap_in_numshots_loop(shots)
executable = qc.compile(program)
result = qc.run(executable)
totals = result.sum(axis=0).tolist()
total_shots = result.sum()
freqs = totals / total_shots
```

The full Forest implementation is included in Appendix P.

5.4.1 Results

Simulations were run ($n = 4096$) using the local Rigetti quantum virtual machine's (QVM) server-based simulator on the same inputs, consisting of primes ($P \in \{3, 5\}$) and remainders ($R \in \{1, 2\}$), as configured in previous implementations. The results of these simulations conformed to the behaviour of those simulations performed on the Strawberry Fields and Qiskit implementations which can be observed in Figures 5.3 and 5.4, respectively.

5.5 Implementation 4: Google Cirq

The final implementation of the Tian et al. circuit was written using Google's unofficial Cirq project. Based on a similar superconducting qubit hardware paradigm, this implementation shares many similarities with the Qiskit and Forest implementations, but was implemented using a class-based approach.

A three qubit system was initialised using a Cirq LineQubit object, as well an empty circuit and a simulator to run it on:

```
q0, q1, q2 = cirq.LineQubit.range(3)
circuit = cirq.Circuit()
simulator = cirq.Simulator()
```

The requisite angles were computed based on the prime and remainder parameters:

```
theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)
```

The first qubit was initialised in the $|1\rangle$ state by rotating its state vector π radians about the y-axis:

```
circuit.append([ry(pi).on(q0)])
```

A set of gates equivalent to a beam splitter gate (in photonics) was created with a utility function bs. The logic behind this combination is explained in detail in Section 5.3

```
def bs(circuit: cirq.Circuit,
       q0: cirq.LineQubit,
       q1: cirq.LineQubit,
       theta: float = np.pi,
       phi: float = np.pi) -> None:
    circuit.append([ry(theta).on(q1).controlled_by(q0),
                    rz(phi).on(q1).controlled_by(q0),
                    CX(q1, q0)])
bs(circuit, q0, q1, theta=2 * theta_e, phi=phi_e)
```

A second bs combination was used to simulate the behaviour of the U_c block described by Tian et al. and utilised once per input character:

```
for _ in range(len(input_string)):
    bs(circuit, q1, q2, theta=2 * theta_c, phi=0)
```

The reversed bsd combination was then used to replicate the behaviour of the $U_e^\dagger$ block:

```
def bsd(circuit: cirq.Circuit,
        q0: cirq.LineQubit,
        q1: cirq.LineQubit,
        theta: float = np.pi,
        phi: float = np.pi) -> None:
    circuit.append([CX(q1, q0),
                    rz(phi).on(q1).controlled_by(q0),
                    ry(theta).on(q1).controlled_by(q0)])
bsd(circuit, q1, q0, theta=2 * theta_e, phi=phi_e)
```

The state flip ambiguity described in Section 5.3 was rectified by the application of a CNOT gate controlled by the state of the first qubit and acting on the second:

```
circuit.append([CX(q0, q1)])
```

A measurement operation affecting all three qubits in the system was the last gate to be added to the circuit.

```
circuit.append(measure_each(q0, q1, q2))
```

With the circuit complete, experiments can be run and the frequencies of the shots falling in each state can be gathered and returned:

```
results = simulator.run(circuit, repetitions=shots)
freqs = {state: sum(values.flatten().tolist()) / shots
         for state, values in results.measurements.items()}
return [freqs[key] for key in sorted(freqs.keys())]
```

The full Cirq implementation of Tian et al. has been made available in Appendix Q.

5.5.1 Results

The results of ($n = 4096$) simulations run on this implementation of Tian et al. using the same configuration settings and test inputs produced results identical to those obtained by simulations on the Strawberry Fields, Qiskit, and Forest implementations. These results are observable in Figures 5.3 and 5.4. As previously explained it was not possible to gather experimental hardware results due to a lack of access to the Cirq-compatible hardware platform maintained by Google.

Chapter 6

Discussion & Conclusions

These twelve implementations across three studies provide a number of insights into the feasibility of implementing quantum finite state automata on general-purpose quantum computers and their utility as abstractions for problem solving. The results of simulations conducted on the circuits produced by Study 1 and Study 2 affirm the previously only theoretical advantages of QFA proposed by Say & Yakaryılmaz [2] in terms of state reduction and the ability to recognise exclusive stochastic languages. These advantages were similarly borne out by the experiments run on IBM's 16-qubit general-purpose quantum computer in Melbourne, albeit not without some caveats.

The effect of noise in the system had a varying level of effect on each of the three studies. Study 1 found that roughly 5% of odd j samples were misidentified as being even, but the redundancy built into the design of the EVENODD problem made it quite evident (based on the preponderance of samples) that these inputs should be identified as odd. Similarly, Study 2 also suffered from the effects of noise, but the probabilistic nature of the interpretation of results made the level of noise observed acceptable. In both cases, a noise threshold of 5% - 10% would ensure that the correct results are accepted without an undue level of risk of false positives.

This issue of noise was compounded with the application of multiple qubits in the attempt to replicate the qutrit-based system used in the Tian et al. hardware implementation [3]. The effect of "missing samples" caused by the combination of noise and the hidden states resulting from the qutrit implementation meant that an unacceptable level of noise was observed, some configurations "losing" up to 50% of samples to these hidden states. This diverged starkly from both the results reported by Tian et al. (roughly 1% error) and the simulations conducted across the four platforms, which aligned with Tian et al. [3]. These issues are not addressable by a simple noise threshold. Some alternatives for combating this issue might include the further reduction of hardware noise through, for example, advances in quantum error correcting codes, or exploration of different alternatives for implementation of higher-dimensional quantum state objects.

The current generation of quantum computers, appropriately named "Noisy Intermediate-Scale Quantum Computers," or NISQ devices, are defined by the presence of this noise

and some researchers have found ways to harness their power despite it [33]. Studies 1 and 2 add to this work in providing models that function regardless of the presence of this level of noise. Study 3 and the implementation of the multi-qubit prime-remainder circuit may be forced to await the further development of less noisy hardware. Analysis has shown that, as in classical computing [34], there exist techniques for the development of systems with arbitrarily low noise from noisy constituent components [20], [35]. These advances may prove the key to the feasibility of multi-qubit quantum finite automata.

In addition to the noise inherent in NISQ-based systems, another form of potential error was identified as a result of these studies, which has been termed "rotational atrophy." This variety of error results from the rotation about the Bloch sphere's various axes and is the result of two systematic issues. The first is the imprecision inherent in effecting these rotations given the available hardware, even for narrowly defined θ or ϕ values. This form of error is largely indistinguishable from the general noise of the system described above, but its effects can be compounded by the types of repeated applications required of QFA. The second component is simple rounding error, which builds up in the same way and could feasibly, with enough rotations, lead to an incorrect determination. An example of the implications of this is shown when the irrational multiple of π utilised in Study 2, when applied enough times, with enough imprecision, could potentially fall on the $|0\rangle$ state, indicating that $|w|_a = |w|_b$ when in fact the input word consisted only of a single character. An exploratory study into the impacts of this kind of noise and its effects on the operation of QFA (e.g. at for large problem sizes) is warranted and necessary if they are to act as components of a general-purpose quantum computing toolkit.

6.1 Future Research Directions

While a number of platforms were utilised in these studies only one provided access to actual quantum hardware in a time frame that allowed for experiments to be conducted. Late stage access to the Amazon Braket platform came too late to be incorporated into these studies given the overhead involved in rewriting the circuits using the Braket Python library. However, future studies could utilise this platform to access Rigetti's superconducting quantum computers as well as other platforms based on other hardware paradigms like D-Wave (quantum annealer-based superconducting qubits) or IonQ (trapped ions) [36].

One of the most frustrating parts of the implementation of these studies across the four platforms was how much repetition was involved and how minute differences in nomenclature between providers could result in days spent finding bugs. In order to facilitate comparative studies like that described above, an overarching, Python-based software library with a single, consistent syntax capable of instantiating circuits and running simulations and experiments on each platform is suggested. This library, while potentially difficult to maintain over frequently changing APIs from each of the vendors,

would facilitate far easier comparative testing, potentially even identifying which platforms outperform others for particular applications. While some platforms like Cirq and Forest support compilation to intermediate assembly languages, and others like Braket support multiple platforms and even different hardware paradigms, these are still limited in their coverage and typically closely coupled with with a proprietary (and paid) system (Braket) or operate on a narrowly defined domain (Xanadu's Penny Lane is targeted at Quantum Machine Learning). Because of the universality of the physical phenomena underlying each implementation of quantum hardware there should be a way to map the functionality (state) and operations (gates) between all platforms.

A system like that described above would also facilitate further expansion of the type of research into porting classical automata to quantum systems. Efforts have already been made at the theoretical level to support more sophisticated computing abstractions (e.g. [1]) and the ability to test the efficacy of these on the various hardware implementations would serve to shorten the feedback loop in the efforts to develop an intuitive model of general-purpose problem solving with quantum computers.

This lack of intuition is currently one of the biggest shortcomings in understanding how the potentially substantial power of quantum computers can be harnessed. Examples to date tend to rely on particularly clever but narrowly applicable phenomena, like the remarkable use of irrational periodic functions in Study 2. What is required is a collection of generalised, actionable tools that require less specific intuitive understanding of the underlying physical phenomena. This fundamentally necessary process of continued abstraction is analogous to the development of classical computing over the past seventy years.

Only as this process of abstraction evolves will it be possible to tackle general-purpose, or real-world, problems. All of experiments demonstrated here were based on reasonably trivial problems that already have classical solutions, but this study has shown that quantum computers, under the right circumstances, have the potential to improve markedly any available classical solutions.

Chapter 7

Bibliography

[1] R. Freivalds and A. Winter, "Quantum finite state transducers," in *SOFSEM 2001: Theory and Practice of Informatics*, L. Pacholski and P. Ružička, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 233–242, ISBN: 978-3-540-45627-8.

[2] A. C. Say and A. Yakaryılmaz, "Quantum finite automata: A modern introduction," in *Computing with New Resources: Essays Dedicated to Jozef Gruska on the Occasion of His 80th Birthday*, C. S. Calude, R. Freivalds, and I. Kazuo, Eds. Cham: Springer International Publishing, 2014, pp. 208–222, ISBN: 978-3-319-13350-8. DOI: 10.1007/978-3-319-13350-8_16. [Online]. Available: https://doi.org/10.1007/978-3-319-13350-8_16.

[3] Y. Tian, T. Feng, M. Luo, S. Zheng, and X. Zhou, "Experimental demonstration of quantum finite automaton," *npj Quantum Information*, vol. 5, no. 1, p. 56, 2019. DOI: 10.1038/s41534-019-0163-x. [Online]. Available: https://doi.org/10.1038/s41534-019-0163-x.

[4] R. Feynman, "Simulating physics with computers," eng, *International Journal of Theoretical Physics*, vol. 21, no. 6-7, pp. 467–488, 1982, ISSN: 0020-7748.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[6] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[7] A. Ambainis and R. Freivalds, "1-way quantum finite automata: Strengths, weaknesses and generalizations," in *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*, 1998, pp. 332–341. DOI: 10.1109/SFCS.1998.743469.

[8] A. Ambainis, R. Bonner, R. Freivalds, and A. Ķikusts, "Probabilities to accept languages by quantum finite automata," in *Computing and Combinatorics*, T. Asano, H. Imai, D. T. Lee, S.-i. Nakano, and T. Tokuyama, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 174–183, ISBN: 978-3-540-48686-2.

[9] A. Kondacs and J. Watrous, "On the power of quantum finite state automata," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, ser. FOCS '97, USA: IEEE Computer Society, 1997, p. 66, ISBN: 0818681977.

[10] H. Abraham, I. Y. Akhalwaya, G. Aleksandrowicz, *et al.*, *Qiskit: An open-source framework for quantum computing*, 2019. DOI: 10.5281/zenodo.2562110.

[11] P. J. Karalekas, N. A. Tezak, E. C. Peterson, *et al.*, "A quantum-classical cloud platform optimized for variational hybrid algorithms," *Quantum Science and Technology*, vol. 5, no. 2, p. 024 003, Apr. 2020. DOI: 10.1088/2058-9565/ab7559. [Online]. Available: https://doi.org/10.1088%2F2058-9565%2Fab7559.

[12] T. R. Bromley, J. M. Arrazola, S. Jahangiri, *et al.*, "Applications of near-term photonic quantum computers: Software and algorithms," *Quantum Science and Technology*, vol. 5, no. 3, p. 034 010, May 2020, ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab8504. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/ab8504.

[13] N. Killoran, J. Izaac, N. Quesada, *et al.*, "Strawberry Fields: A Software Platform for Photonic Quantum Computing," *Quantum*, vol. 3, p. 129, Mar. 2019, ISSN: 2521-327X. DOI: 10.22331/q-2019-03-11-129. [Online]. Available: https://doi.org/10.22331/q-2019-03-11-129.

[14] A. Yakaryilmaz and A. C. C. Say, "Languages recognized by nondeterministic quantum finite automata," *Quantum Info. Comput.*, vol. 10, no. 9, pp. 747–770, Sep. 2010, ISSN: 1533-7146.

[15] M. Sipser, *Introduction to the theory of computation*. Boston, MA: Cengage Learning, 2013, ISBN: 978-1-133-18779-0.

[16] D. Povey, A. Ghoshal, G. Boulianne, *et al.*, "The kaldi speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, IEEE Catalog No.: CFP11SRW-USB, Hilton Waikoloa Village, Big Island, Hawaii, US: IEEE Signal Processing Society, Dec. 2011.

[17] J. D. U. John E. Hopcroft Rajeev Motwani, *Introduction to automata theory, languages, and computation*, 2nd ed. Addison-Wesley, 2001, ISBN: 9780201441246. [Online]. Available: https://bit.ly/327PZrh.

[18] M. Hirvensalo, *Quantum computing / Mika Hirvensalo.* (Natural computing series), eng, Second Edition. Berlin, Germany: Springer, 2004, ISBN: 3-662-09636-6.

[19] L. Gyongyosi and S. Imre, "A survey on quantum computing technology," *Computer Science Review*, vol. 31, pp. 51–71, Feb. 2019. DOI: 10.1016/j.cosrev.2018.11.002. [Online]. Available: https://doi.org/10.1016/j.cosrev.2018.11.002.

[20] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th. USA: Cambridge University Press, 2011, ISBN: 1107002176.

[21] W. Commons. "Bloch sphere; a geometrical representation of a two-level quantum system." (2012), [Online]. Available: https://commons.wikimedia.org/wiki/File:Bloch_Sphere.svg.

[22] M. Q. Team, *Announcing the microsoft quantum development kit*, Dec. 2017. [Online]. Available: https://cloudblogs.microsoft.com/quantum/2017/12/11/announcing-microsoft-quantum-development-kit/.

[23] R. S. Smith, M. J. Curtis, and W. J. Zeng, *A practical quantum instruction set architecture*, 2016. arXiv: 1608.03355 [quant-ph].

[24] R. LaRose, "Overview and comparison of gate level quantum software platforms," *Quantum*, vol. 3, p. 130, Mar. 2019. DOI: 10.22331/q-2019-03-25-130. [Online]. Available: https://doi.org/10.22331/q-2019-03-25-130.

[25] P. Kok, W. J. Munro, K. Nemoto, *et al.*, "Linear optical quantum computing with photonic qubits," *Reviews of Modern Physics*, vol. 79, no. 1, pp. 135–174, Jan. 2007, ISSN: 1539-0756. DOI: 10.1103/revmodphys.79.135. [Online]. Available: http://dx.doi.org/10.1103/RevModPhys.79.135.

[26] M. Kjaergaard, M. E. Schwartz, J. Braumüller, *et al.*, "Superconducting qubits: Current state of play," *Annual Review of Condensed Matter Physics*, vol. 11, no. 1, pp. 369–395, Mar. 2020, ISSN: 1947-5462. DOI: 10.1146/annurev-conmatphys-031119-050605. [Online]. Available: http://dx.doi.org/10.1146/annurev-conmatphys-031119-050605.

[27] J. Hidary, *Quantum Computing: An Applied Approach*. Springer, 2019, ISBN: 9783030239237. [Online]. Available: https://books.google.com.au/books?id=49-5zQEACAAJ.

[28] *Google devices — cirq 0.9.1 documentation*, https://cirq.readthedocs.io/en/stable/docs/google/devices.html, (Accessed on 11/09/2020).

[29] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1962, ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10. eprint: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/5.1.10.

[30] D. Farago, *Generators - python wiki*, https://wiki.python.org/moin/Generators, (Accessed on 10/29/2020).

[31] V. Geffert and A. Yakaryilmaz, "Classical Automata on Promise Problems," *Discrete Mathematics and Theoretical Computer Science*, vol. Vol. 17 no.2, no. 2, pp. 157–180, Sep. 2015. [Online]. Available: https://hal.inria.fr/hal-01349053.

[32] A. Salomaa, I. Sneddon, M. Stark, and J. Kahane, *Theory of Automata* (ISSN). Elsevier Science, 2014, ISBN: 9781483154398. [Online]. Available: https://books.google.com.au/books?id=WnjiBQAAQBAJ.

[33] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018, ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. [Online]. Available: http://dx.doi.org/10.22331/q-2018-08-06-79.

[34] C. E. Shannon and J. McCarthy, *Automata Studies. (AM-34), Volume 34*. Princeton: Princeton University Press, Mar. 2016, ISBN: 978-1-4008-8261-8. DOI: https://doi.org/10.1515/9781400882618. [Online]. Available: https://www.degruyter.com/princetonup/view/title/521931.

[35] E. Knill, R. Laflamme, and W. H. Zurek, "Resilient quantum computation," *Science*, vol. 279, no. 5349, pp. 342–345, 1998, ISSN: 0036-8075. DOI: 10.1126/science.279.5349.342. eprint: https://science.sciencemag.org/content/279/5349/342.full.pdf. [Online]. Available: https://science.sciencemag.org/content/279/5349/342.

[36] Amazon, *Amazon braket hardware providers - amazon web services*, https://aws.amazon.com/braket/hardware-providers/, (Accessed on 11/09/2020).

Appendices

Appendix A

Common Utilities for EVENODD Implementations

```
import os
from typing import Callable, List

from PIL import Image
from matplotlib import pyplot as plt
from numpy import pi


def get_theta(k: int) -> float:
    return pi / (2 ** (k + 1))


def produce_graph(name: str,
                  k: int,
                  js: List[int],
                  run_experiment: Callable,
                  bar_width: int = 1) -> None:
    results = [run_experiment(k=k, j=j)
               for j in js]

    # X-Axis Values
    count = len(js) + 1
    start = [0, count]
    x_axis_values = [start, ]
    for _ in js[1:]:
        x_axis_values.append([x + bar_width
                              for x in x_axis_values[-1]])

    # Plot groups by input length
    colors = ["#FFCCCC", "#CCCCFF",
              "#FF9999", "#9999FF",
              "#FF6666", "#6666FF"]
```

```
    for index, x_axis_value in enumerate(x_axis_values):
        plt.bar(x_axis_value,
                height=results[index],
                color=colors[index],
                label=f"{js[index]}")

    # Add flourishes
    # plt.suptitle(rf"$\mathbf{{k = {k}}}\$")
    plt.suptitle(rf"$\mathbf{{k = {k}}}$", fontsize=20)
    plt.xlabel('Final State', fontweight='bold', fontsize=16, labelpad=-10)
    tick_space = 2.5 if count == 7 else 1.5
    plt.xticks([tick_space + count * r for r in range(len(results[0]))],
               [r'$|0\rangle$', r'$|1\rangle$'], fontsize=14)
    plt.ylim([0, 1])
    plt.tick_params(labelsize=14)
    plt.ylabel('Normalised Frequency', fontweight='bold', fontsize=16)
    plt.legend(fontsize=14)

    # Save and clear
    # plt.show()
    plt.savefig(f"{name}.png")
    plt.clf()


def run_evenodd(run_experiment: Callable):
    produce_graph(name="1",
                  run_experiment=run_experiment,
                  k=1,
                  js=[1, 2, 3, 4, 5, 6])
    produce_graph(name="2",
                  run_experiment=run_experiment,
                  k=2,
                  js=[1, 2, 3, 4, 5, 6])
    produce_graph(name="3",
                  run_experiment=run_experiment,
                  k=3,
                  js=[1, 2, 3, 4, 5, 6])
    produce_graph(name="4",
                  run_experiment=run_experiment,
                  k=4,
                  js=[1, 2, 3, 4, 5, 6])

    # Combine graphs

    files = [
        '1.png',
        '2.png',
        '3.png',
```

```
        '4.png']

    result = Image.new("RGB", (1280, 960))

    for index, file in enumerate(files):
        path = os.path.expanduser(file)
        img = Image.open(path)
        img.thumbnail((640, 480), Image.ANTIALIAS)
        x = index // 2 * 640
        y = index % 2 * 480
        w, h = img.size
        result.paste(img, (x, y, x + w, y + h))

    result.save(os.path.expanduser('combined.png'))
```

Listing A.1: Common Utilities for EVENODD Implementations

Appendix B

Strawberry Fields Implementation of EVENODD

```
from strawberryfields.ops import *
import strawberryfields as sf
from typing import List
from pathlib import Path
from functools import partial

from src.evenodd.utils.common import get_theta, run_evenodd


def run_experiment(k: int,
                   j: int,
                   simulation: bool = True,
                   shots=4096) -> List[float]:


    program = sf.Program(2)
    engine = sf.Engine("fock", backend_options={"cutoff_dim": 2})

    input_string = "a" * j * (2 ** k)
    theta = get_theta(k)
    length = len(input_string)

    with program.context as q:
        Fock(1) | q[0]
        Vacuum() | q[1]

        for _ in input_string:
            BSgate(theta) | (q[0], q[1])

    result = engine.run(program, shots=shots)
    state = result.state
    probs = [round(abs(state.mean_photon(0)[0]), 4),
```

```
33                 round(abs(state.mean_photon(1)[0]), 4)]
34
35     formatted_results = f"L: {length}, k:{k}, j:{j}, Probs: {probs}, " \
36                         f"Shots: {shots}, Expected: {bool(j % 2 == 0)}, " \
37                         f"Got: {bool(probs[0] == 1.0)} \n"
38
39     print(formatted_results, end="")
40
41     with open("results/simulation/evenodd-sf.log", "a+") as log:
42         log.write(formatted_results)
43
44     return probs
45
46
47 if __name__ == "__main__":
48     Path("results/simulation/evenodd-sf.log").unlink(missing_ok=True)
49     run_evenodd(partial(run_experiment, simulation=True))
```

Listing B.1: Strawberry Fields Implementation of EVENODD

Appendix C

IBM Qiskit Implementation of EVENODD

```
from functools import partial
from os import environ
from pathlib import Path
from typing import List

from qiskit import QuantumCircuit, Aer, execute, IBMQ

from src.evenodd.utils.common import get_theta, run_evenodd


def run_experiment(k: int,
                   j: int,
                   simulation: bool = True,
                   shots=4096) -> List[float]:

    circuit = QuantumCircuit(1, 1)

    input_string = "a" * j * (2 ** k)
    theta = get_theta(k)
    length = len(input_string)

    if simulation:
        backend = Aer.get_backend('qasm_simulator')
    else:
        provider = IBMQ.get_provider(group='open', project='main')
        backend = provider.get_backend('ibmq_16_melbourne')

    for _ in input_string:
        circuit.ry(theta * 2, 0)

    circuit.measure(0, 0)
    job = execute(circuit, backend, shots=shots)
```

```
33      result = job.result()
34      counts = result.get_counts(circuit)
35      states = ['0', '1']
36      probs = [counts.get(state, 0) / shots for state in states]
37      formatted_results = f"L: {length}, k:{k}, j:{j}, Probs: {probs}, " \
38                          f"Shots: {shots}, Expected: {bool(j % 2 == 0)}, " \
39                          f"Got: {bool(probs[0] == 1.0)} \n"
40
41      print(formatted_results, end="")
42
43      with open("results/2/evenodd-qiskit.log", "a+") as log:
44          log.write(formatted_results)
45
46      return probs
47
48
49  if __name__ == "__main__":
50      IBMQ.enable_account(token=environ.get('QISKIT_API_KEY'))
51      Path("results/2/evenodd-qiskit.log").unlink(missing_ok=True)
52      run_evenodd(partial(run_experiment, simulation=False))
```

Listing C.1: IBM Implementation of EVENODD

Appendix D

Rigetti Forest Implementation of EVENODD

```
from functools import partial
from pathlib import Path
from typing import List

from pyquil import Program, get_qc
from pyquil.gates import *

from src.evenodd.utils.common import get_theta, run_evenodd


def run_experiment(k: int,
                   j: int,
                   simulation: bool = True,
                   shots=4096) -> List[float]:
    qc = get_qc('1q-qvm')
    program = Program()
    output = program.declare('ro', 'BIT', 1)

    theta = get_theta(k)
    input_string = "a" * j * (2 ** k)
    length = len(input_string)

    for _ in input_string:
        program += RY(theta * 2, 0)

    program += MEASURE(0, output[0])
    program.wrap_in_numshots_loop(shots)
    executable = qc.compile(program)
    result = qc.run(executable)
    totals = result.flatten().tolist()
    probs = [totals.count(0) / shots, totals.count(1) / shots]

```

```
33      formatted_results = f"L: {length}, k:{k}, j:{j}, Probs: {probs}, " \
34                          f"Shots: {shots}, Expected: {bool(j % 2 == 0)}, " \
35                          f"Got: {bool(probs[0] == 1.0)} \n"
36
37      print(formatted_results, end="")
38
39      with open("results/simulation/evenodd-forest.log", "a+") as log:
40          log.write(formatted_results)
41
42      return probs
43
44
45  if __name__ == "__main__":
46      Path("results/simulation/evenodd-forest.log").unlink(missing_ok=True)
47      run_evenodd(partial(run_experiment, simulation=True))
```

Listing D.1: Forest Implementation of EVENODD

Appendix E

Cirq Implementation of EVENODD

```python
from functools import partial
from typing import List

import cirq
from cirq.ops import ry, measure_each, NamedQubit

from src.evenodd.utils.common import get_theta, run_evenodd


def run_experiment(k: int,
                   j: int,
                   simulation: bool = True,
                   shots=4096) -> List[float]:

    q0 = NamedQubit('source')
    circuit = cirq.Circuit()
    simulator = cirq.Simulator()

    input_string = "a" * j * (2 ** k)
    theta = get_theta(k)
    length = len(input_string)

    for _ in input_string:
        circuit.append([ry(theta * 2).on(q0)])

    circuit.append(measure_each(q0))

    results = simulator.run(circuit, repetitions=shots)
    results = sum(results.measurements.values()).tolist()
    probs = [results.count([0]) / shots, results.count([1]) / shots]

    formatted_results = f"L: {length}, k:{k}, j:{j}, Probs: {probs}, " \
                        f"Shots: {shots}, Expected: {bool(j % 2 == 0)}, " \
                        f"Got: {bool(probs[0] == 1.0)} \n"
```

```
35
36      print(formatted_results, end="")
37
38      with open("results/simulation/evenodd-cirq.log", "a+") as log:
39          log.write(formatted_results)
40
41      return probs
42
43
44  if __name__ == "__main__":
45      run_evenodd(partial(run_experiment, simulation=True))
```

Listing E.1: Cirq Implementation of EVENODD

Appendix F

Common Utilities & Execution Harness for NEQ Implementations

```python
import os
from random import shuffle
from typing import Callable, List, Tuple

import numpy as np
from PIL import Image
from numpy import pi
from matplotlib import pyplot as plt


def get_theta() -> float:
    return np.sqrt(2) * pi


def get_input(a: int, b: int) -> Tuple[str, int]:
    input_chars = list("a" * a + "b" * b)
    shuffle(input_chars)
    input_string = "".join(input_chars)
    length = len(input_string)
    return input_string, length


def produce_graph(name: str,
                  a: int,
                  bs: List[int],
                  run_experiment: Callable,
                  bar_width: int = 1) -> None:
    results = [run_experiment(a=a, b=b) for b in bs]

    # X-Axis Values
    count = len(bs) + 1
    start = [0, count]
```

```
    x_axis_values = [start, ]
    for _ in bs[1:]:
        x_axis_values.append([x + bar_width
                              for x in x_axis_values[-1]])

    # Plot groups by input length
    colors = ["#CCCCFF", "#9999FF",
              "#6666FF", "#3333FF",
              "#0000FF", "#0000E5"]

    for index, x_axis_value in enumerate(x_axis_values):
        plt.bar(x_axis_value,
                height=results[index],
                color=colors[index] if results[index][0] < 0.96 else "#
    FF0000",
                label=f"{bs[index]}")

    # Add flourishes
    # plt.suptitle(rf"$\mathbf{{k = {k}}}\$")
    plt.suptitle(rf"$\mathbf{{a = {a}}}$", fontsize=20)
    plt.xlabel('Final State', fontweight='bold', fontsize=16, labelpad=-10)
    tick_space = 2.5 if count == 7 else 1.5
    plt.xticks([tick_space + count * r for r in range(len(results[0]))],
               [r'$|0\rangle$', r'$|1\rangle$'])
    plt.ylim([0, 1])
    plt.tick_params(labelsize=14)
    plt.ylabel('Normalised Frequency', fontweight='bold', fontsize=16)
    plt.legend(fontsize=14)

    # Save and clear
    # plt.show()
    plt.savefig(f"{name}.png")
    plt.clf()


def run_neq(run_experiment: Callable):
    produce_graph(name="1",
                  run_experiment=run_experiment,
                  a=1,
                  bs=[1, 2, 3, 4, 5, 6])
    produce_graph(name="2",
                  run_experiment=run_experiment,
                  a=2,
                  bs=[1, 2, 3, 4, 5, 6])
    produce_graph(name="3",
                  run_experiment=run_experiment,
                  a=3,
                  bs=[1, 2, 3, 4, 5, 6])
```

```
    produce_graph(name="4",
                  run_experiment=run_experiment,
                  a=4,
                  bs=[1, 2, 3, 4, 5, 6])

    # Combine graphs

    files = [
        '1.png',
        '2.png',
        '3.png',
        '4.png']

    result = Image.new("RGB", (1280, 960))

    for index, file in enumerate(files):
        path = os.path.expanduser(file)
        img = Image.open(path)
        img.thumbnail((640, 480), Image.ANTIALIAS)
        x = index // 2 * 640
        y = index % 2 * 480
        w, h = img.size
        result.paste(img, (x, y, x + w, y + h))

    result.save(os.path.expanduser('combined.png'))
```

Listing F.1: Common Utilities for NEQ Implementations

Appendix G

Strawberry Fields Implementation of NEQ

```
from functools import partial
from pathlib import Path
from typing import List
from random import shuffle

from strawberryfields.ops import *
import strawberryfields as sf

from src.neq.utils.common import run_neq, get_theta, get_input


def run_experiment(a: int,
                   b: int,
                   simulation=True,
                   shots=4096) -> List[float]:

    program = sf.Program(2)
    engine = sf.Engine("fock", backend_options={"cutoff_dim": 2})

    theta = get_theta()
    input_string, length = get_input(a, b)

    with program.context as q:
        Fock(1) | q[0]
        Vacuum() | q[1]

        for char in input_string:
            if char == "a":
                BSgate(theta) | (q[0], q[1])
            elif char == "b":
                BSgate(-theta) | (q[0], q[1])
            else:
```

```
                raise NotImplementedError

    result = engine.run(program, shots=shots)
    state = result.state
    probs = [round(abs(state.mean_photon(0)[0]), 4),
             round(abs(state.mean_photon(1)[0]), 4)]

    formatted_results = f"L: {length}, a:{a}, b:{b}, Probs: {probs}, " \
                        f"Shots: {shots}, Expected: {bool(a != b)}, " \
                        f"Got: {bool(probs[0] == 1.0)} \n"

    print(formatted_results, end="")

    with open("results/simulation/neq-sf.log", "a+") as log:
        log.write(formatted_results)

    return probs


if __name__ == "__main__":
    Path("results/simulation/neq-sf.log").unlink(missing_ok=True)
    run_neq(partial(run_experiment, simulation=True))
```

Listing G.1: Strawberry Fields Implementation of NEQ

Appendix H

IBM Qiskit Implementation of NEQ

```
from functools import partial
from os import environ
from pathlib import Path
from typing import List
from random import shuffle

from qiskit import QuantumCircuit, Aer, execute, IBMQ

from src.neq.utils.common import run_neq, get_theta, get_input


def run_experiment(a: int,
                   b: int,
                   simulation=True,
                   shots=4096) -> List[float]:
    circuit = QuantumCircuit(1, 1)
    if simulation:
        backend = Aer.get_backend('qasm_simulator')
    else:
        provider = IBMQ.get_provider(group='open',
                                     project='main')
        backend = provider.get_backend('ibmq_ourense')

    theta = get_theta()
    input_string, length = get_input(a, b)

    # rotations = length // (2 ** k)
    for char in input_string:
        if char == "a":
            circuit.ry(theta * 2, 0)
        elif char == "b":
            circuit.ry(-theta * 2, 0)
        else:
            raise NotImplementedError
```

```
35
36      circuit.measure(0, 0)
37
38      job = execute(circuit, backend, shots=shots)
39      result = job.result()
40      counts = result.get_counts(circuit)
41      states = ['0', '1']
42      probs = [counts.get(state, 0) / shots for state in states]
43      formatted_results = f"L: {length}, a:{a}, b:{b}, Probs: {probs}, " \
44                          f"Shots: {shots}, Expected: {bool(a != b)}, " \
45                          f"Got: {bool(probs[0] == 1.0)} \n"
46
47      print(formatted_results, end="")
48
49      with open("results/2/neq-qiskit.log", "a+") as log:
50          log.write(formatted_results)
51
52      return probs
53
54
55  if __name__ == "__main__":
56      IBMQ.enable_account(token=environ.get('QISKIT_API_KEY'))
57      Path("evenodd-qiskit.log").unlink(missing_ok=True)
58      run_neq(partial(run_experiment, simulation=False))
```

Listing H.1: IBM Qiskit Implementation of NEQ

Appendix I

Rigetti Forest Implementation of NEQ

```
from functools import partial
from pathlib import Path
from random import shuffle
from typing import List

from pyquil import Program, get_qc
from pyquil.gates import *

from src.neq.utils.common import get_theta, run_neq, get_input


def run_experiment(a: int,
                   b: int,
                   simulation: bool = True,
                   shots=4096) -> List[float]:
    circuit = get_qc('1q-qvm')
    program = Program()
    output = program.declare('ro', 'BIT', 1)

    theta = get_theta()
    input_string, length = get_input(a, b)

    for char in input_string:
        if char == "a":
            program += RY(theta * 2, 0)
        elif char == "b":
            program += RY(-theta * 2, 0)
        else:
            raise NotImplementedError

    program += MEASURE(0, output[0])

```

```
    program.wrap_in_numshots_loop(shots)
    executable = circuit.compile(program)
    result = circuit.run(executable)
    totals = result.flatten().tolist()
    probs = [totals.count(0) / shots, totals.count(1) / shots]

    formatted_results = f"L: {length}, a:{a}, b:{b}, Probs: {probs}, " \
                        f"Shots: {shots}, Expected: {bool(a != b)}, " \
                        f"Got: {bool(probs[0] == 1.0)} \n"

    print(formatted_results, end="")

    with open("results/simulation/neq-forest.log", "a+") as log:
        log.write(formatted_results)

    return probs


if __name__ == "__main__":
    Path("results/simulation/neq-forest.log").unlink(missing_ok=True)
    run_neq(partial(run_experiment))
```

Listing I.1: Rigetti Forest Implementation of NEQ

Appendix J

Cirq Implementation of NEQ

```
from functools import partial
from random import shuffle
from typing import List

import cirq
from cirq.ops import ry, measure_each, NamedQubit

from src.neq.utils.common import run_neq, get_theta, get_input


def run_experiment(a: int,
                   b: int,
                   simulation=True,
                   shots=4096):
    q0 = NamedQubit('source')
    simulator = cirq.Simulator()
    circuit = cirq.Circuit()

    theta = get_theta()
    input_string, length = get_input(a, b)

    for char in input_string:
        if char == "a":
            circuit.append([ry(theta * 2).on(q0)])
        elif char == "b":
            circuit.append([ry(-theta * 2).on(q0)])
        else:
            raise NotImplementedError

    circuit.append(measure_each(q0))

    measurements = simulator.run(circuit, repetitions=shots)
    results = sum(measurements.measurements.values()).tolist()
    probs = [results.count([0]) / shots, results.count([1]) / shots]
```

```
35
36      formatted_results = f"L: {length}, a:{a}, b:{b}, Probs: {probs}, " \
37                          f"Shots: {shots}, Expected: {bool(a != b)}, " \
38                          f"Got: {bool(probs[0] == 1.0)} \n"
39
40      print(formatted_results, end="")
41
42      with open("results/simulation/neq-cirq.log", "a+") as log:
43          log.write(formatted_results)
44
45      return probs
46
47
48  if __name__ == "__main__":
49      run_neq(partial(run_experiment, simulation=True))
```

Listing J.1: Cirq Implementation of NEQ

Appendix K

Solver for Q

```python
import numpy as np


def get_q(prime: int, remainder: int) -> int:
    if prime / 4 <= remainder <= 3 * prime / 4:
        q = 1
    elif prime / 4 > remainder:
        q = np.ceil(prime / (4 * remainder))
    else:
        prime = (prime - remainder) / remainder
        j = 1
        temp_j = j * prime - np.floor(j * prime)
        while j < 10:
            if 1 / 4 < temp_j or temp_j < 2 / 3:
                break
            j += 1
        q = np.floor(prime * (j + 1 / 4) / remainder) + 1

    return q
```

Listing K.1: Python Solver for Q in Tian et al. (2019) [3]

Appendix L

Solver for angles Ue and Uc

```python
import numpy as np


def get_angle_ue(prime: int, remainder: int) -> float:
    q = get_q(prime, remainder)

    return 2 * np.pi * q * remainder / prime


def get_angle_uc(prime, remainder) -> float:
    q = get_q(prime, remainder)

    return 2 * np.pi * q / prime
```

Listing L.1: Python Solver for Q in Tian et al. (2019) [3]

Appendix M

Common Utilities for Tian et al. (2019)

```python
import os
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable, Tuple, Any


def get_q(prime: int, remainder: int) -> int:
    # Determine Q
    if prime / 4 <= remainder <= 3 * prime / 4:
        q = 1
    elif prime / 4 > remainder:
        q = np.ceil(prime / (4 * remainder))
    else:
        prime = (prime - remainder) / remainder
        j = 1
        temp_j = j * prime - np.floor(j * prime)
        while j < 10:
            if 1 / 4 < temp_j or temp_j < 2 / 3:
                break
            j += 1
        q = np.floor(prime * (j + 1 / 4) / remainder) + 1

    return q


def get_angle_ue(prime: int, remainder: int) -> float:
    q = get_q(prime, remainder)

    return 2 * np.pi * q * remainder / prime


```

```
def get_angle_uc(prime, remainder) -> float:
    q = get_q(prime, remainder)

    return 2 * np.pi * q / prime


def get_angles(prime: int, remainder: int) -> Tuple[Any, Any, Any, Any]:
    # Need to compute correct angles for Ue
    theta = get_angle_ue(prime, remainder)
    t = np.cos(theta)

    # add perturbation for asymptotes
    t = 1e-10 if theta == 0 else t
    t = t + 1e-10 if theta == 1 else t

    # define initial angles
    x1 = 1j * np.sqrt(t / (1 - t) + 0j)
    x2 = np.sqrt(1 / (1 - t) + 0j)

    # For the angle in Ue we need to define it slightly differently
    theta_e = np.arctan(np.abs(x2) / np.abs(x1))
    phi_e = np.angle(x2 / x1)

    # Get angle for Uc
    theta_c = get_angle_uc(prime, remainder)

    return theta_c, theta_e, 0, phi_e


def produce_graph(name: str,
                  run_experiment: Callable,
                  bar_width: int = 1,
                  prime: int = 3,
                  remainder: int = 1,
                  lengths: tuple = None) -> None:
    results = [run_experiment(prime, remainder, "0" * length)
               for length in lengths]

    # X-Axis Values
    count = len(lengths) + 1
    start = [0, count, count * 2]
    x_axis_values = [start, ]
    for _ in lengths[1:]:
        x_axis_values.append([x + bar_width
                              for x in x_axis_values[-1]])

    # Plot groups by input length
    colors = ["#9DC4E8", "#FFC980", "#87CE40",
```

```
                "#FDFE7F", "#7E8004", "#BFBFBF"]
    for index, x_axis_value in enumerate(x_axis_values):
        plt.bar(x_axis_value,
                height=results[index],
                color=colors[index],
                label=f"{lengths[index]}")

    # Add flourishes
    plt.suptitle(rf"$\mathbf{{n = {prime}k\, or\, "
                 rf"{prime}k + {remainder}}}$", fontsize=20)
    plt.xlabel('Final State', fontweight='bold', fontsize=16, labelpad=-2)
    tick_space = 2.5 if count == 7 else 1.5
    plt.xticks([tick_space + count * r for r in range(len(results[0]))],
               [r'$|0\rangle$', r'$|1\rangle$', r'$|2\rangle$'])
    plt.ylim([0, 1])
    plt.tick_params(labelsize=14)
    plt.ylabel('Normalised Frequency', fontweight='bold', fontsize=16)
    plt.legend(fontsize=14)

    # Save and clear
    # plt.show()
    plt.savefig(f"{name}.png")
    plt.clf()


def run_tian(run_experiment: Callable):
    produce_graph(name="a",
                  run_experiment=run_experiment,
                  prime=3,
                  remainder=1,
                  lengths=(0, 1, 3, 4, 6, 7))
    produce_graph(name="b",
                  run_experiment=run_experiment,
                  prime=3,
                  remainder=2,
                  lengths=(0, 2, 3, 5))
    produce_graph(name="c",
                  run_experiment=run_experiment,
                  prime=5,
                  remainder=1,
                  lengths=(0, 1, 5, 6))
    produce_graph(name="d",
                  run_experiment=run_experiment,
                  prime=5,
                  remainder=2,
                  lengths=(0, 2, 5, 7))

    # Combine graphs
```

```
129
130     files = [
131         'a.png',
132         'c.png',
133         'b.png',
134         'd.png']
135
136     result = Image.new("RGB", (1280, 960))
137
138     for index, file in enumerate(files):
139         path = os.path.expanduser(file)
140         img = Image.open(path)
141         img.thumbnail((640, 480), Image.ANTIALIAS)
142         x = index // 2 * 640
143         y = index % 2 * 480
144         w, h = img.size
145         # print('pos {0},{1} size {2},{3}'.format(x, y, w, h))
146         result.paste(img, (x, y, x + w, y + h))
147
148     result.save(os.path.expanduser('combined.png'))
```

Listing M.1: Common Utilities for Tian et al. (2019) Implementations

Appendix N

Strawberry Fields Implementation of Tian et al.

```python
from strawberryfields.ops import *
import strawberryfields as sf
from typing import List
from src.tian.utils.common import run_tian, get_angles


def run_experiment(prime: int,
                   remainder: int,
                   input_string: str) -> List[float]:

    n_subsystems = 3
    program = sf.Program(n_subsystems)
    engine = sf.Engine("fock", backend_options={"cutoff_dim": 2})

    # Compute angles
    theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)

    with program.context as q:
        Fock(1) | q[0]  # Setup channel 0
        Vacuum() | q[1]  # Empty channel 1
        Vacuum() | q[2]  # Empty channel 2

        # Ue
        BSe = BSgate(theta_e, phi_e)
        BSe | (q[0], q[1])

        # Uc
        BSc = BSgate(theta_c, 0)
        for _ in range(len(input_string)):  # n applications of Uc
            BSc | (q[1], q[2])

        # Ued
```

```
        BSed = BSgate(theta_e, -phi_e)
        BSed | (q[1], q[0])

    result = engine.run(program)
    state = result.state
    probs = [state.mean_photon(i)[0] for i in range(n_subsystems)]

    print(len(input_string), probs)

    return probs


if __name__ == "__main__":
    run_tian(run_experiment)
```

Listing N.1: Strawberry Fields Implementation of Tian et al. (2019)

Appendix O

IBM Qiskit Implementation of Tian et al. (2019)

```
import numpy as np
from os import environ
from functools import partial
from numpy import pi
from qiskit import QuantumCircuit, Aer, execute, IBMQ
from typing import List
from src.tian.utils.common import get_angles, run_tian


def bs(circuit, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q0 is incoming whereas q1 is outgoing
    circuit.cry(theta, q0, q1)
    circuit.crz(phi, q0, q1)
    circuit.cx(q1, q0)


def bsd(circuit, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q0 is incoming whereas q1 is outgoing
    circuit.cx(q1, q0)
    circuit.crz(phi, q0, q1)
    circuit.cry(theta, q0, q1)


def run_experiment(prime: int,
                   remainder: int,
                   input_string: str,
                   simulation: bool = True) -> List[float]:
    circuit = QuantumCircuit(3, 3)

    if simulation:
```

```
33          backend = Aer.get_backend('qasm_simulator')
34      else:
35          provider = IBMQ.get_provider(group='open',
36                                       project='main')
37          backend = provider.get_backend('ibmq_ourense')
38
39      # Define FSA params
40      length = len(input_string)
41
42      theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)
43
44      # Initialise state of first channel to 1
45      circuit.ry(pi, 0)
46
47      # Ue
48      bs(circuit, 0, 1, theta=2 * theta_e, phi=phi_e)
49
50      # # Uc ^ n
51      bs(circuit, 1, 2, 2 * theta_c * length, phi=phi_c)
52
53      # Ued
54      bsd(circuit, 1, 0, theta=2 * theta_e, phi=phi_e)
55
56      # Reverts
57      circuit.cx(0, 1)
58
59      circuit.measure(range(3), range(3))
60      shots = 4096
61      job = execute(circuit, backend, shots=shots)
62      counts = job.result().get_counts(circuit)
63      states = ['001', '010', '100']
64      probs = [counts.get(state, 0) / shots for state in states]
65      all_probs = {key: value/shots for key, value in counts.items()}
66
67      formatted_results = f"P: {prime}, R: {remainder}, L: {length}, Probs: {
    probs} All: {all_probs}\n"
68
69      print(formatted_results, end="")
70
71      with open("results/6/tian-qiskit.log", "a+") as log:
72          log.write(formatted_results)
73
74      # circuit.draw(output='mpl', filename="circuit.png")
75
76      return probs
77
78
79  if __name__ == "__main__":
```

```
80      IBMQ.enable_account(token=environ.get('QISKIT_API_KEY'))
81      run_tian(partial(run_experiment, simulation=False))
82      # run_experiment(3, 1, input_string="000000", simulation=True)
```

Listing O.1: IBM Qiskit Implementation of Tian et al. (2019)

Appendix P

Rigetti Forest Implementation of Tian et al. (2019)

```python
from pyquil import Program, get_qc
from pyquil.gates import *
import numpy as np
from numpy import pi
from src.tian.utils.common import get_angle_ue, get_angle_uc, run_tian,
    get_angles


def cry(program, control, target, theta) -> None:
    program += RY(theta, target).controlled(control)


def crz(program, control, target, phi) -> None:
    program += RZ(phi, target).controlled(control)


def bs(program, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q1 is incoming whereas q2 is outgoing
    cry(program, q0, q1, theta)  # circuit.cry(theta, q0, q1)
    crz(program, q0, q1, phi)  # circuit.crz(phi, q0, q1)
    program += CNOT(q1, q0)  # circuit.cx(q1, q0)


def bsd(program, q0, q1, theta=np.pi / 2, phi=0):
    # Create BS using controlled rotation and cnot
    # q1 is incoming whereas q2 is outgoing
    program += CNOT(q1, q0)
    crz(program, q0, q1, phi)
    cry(program, q0, q1, theta)


```

```
def run_experiment(prime, remainder, input_string, shots=4096):
    qc = get_qc('3q-qvm')
    program = Program()
    output = program.declare('ro', 'BIT', 3)

    length = len(input_string)

    theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)

    # Initialise state of first channel to 1
    program += RY(pi, 0)

    # Ue
    bs(program, 0, 1, theta=2 * theta_e, phi=phi_e)

    # Uc ^ n
    for _ in range(length):
        cry(program, 1, 2, theta_c * 2)
    program += CNOT(2, 1)

    # Ued
    bsd(program, 1, 0, theta=2 * theta_e, phi=phi_e)

    # Revert
    program += CNOT(0, 1)

    program += MEASURE(0, output[0])
    program += MEASURE(1, output[1])
    program += MEASURE(2, output[2])

    program.wrap_in_numshots_loop(shots)

    executable = qc.compile(program)
    result = qc.run(executable)
    totals = result.sum(axis=0).tolist()
    total_shots = result.sum()
    probs = totals / total_shots

    formatted_results = f"P: {prime}, R: {remainder}, L: {length}, Probs: {probs}\n"

    print(formatted_results, end="")

    with open("tian-qiskit.log", "a+") as log:
        log.write(formatted_results)

    return probs

```

```
79
80 if __name__ == "__main__":
81     run_tian(run_experiment)
```

Listing P.1: IBM Qiskit Implementation of Tian et al. (2019)

Appendix Q

Cirq Implementation of Tian et al. (2019)

```
from typing import List

import cirq
import numpy as np
from cirq.ops import ry, rz, CX, measure_each
from numpy import pi

from src.tian.utils.common import run_tian, get_angles


def bs(circuit: cirq.Circuit,
       q0: cirq.LineQubit,
       q1: cirq.LineQubit,
       theta: float = np.pi,
       phi: float = np.pi) -> None:
    circuit.append([ry(theta).on(q1).controlled_by(q0),
                    rz(phi).on(q1).controlled_by(q0),
                    CX(q1, q0)])


def bsd(circuit: cirq.Circuit,
        q0: cirq.LineQubit,
        q1: cirq.LineQubit,
        theta: float = np.pi,
        phi: float = np.pi) -> None:
    circuit.append([CX(q1, q0),
                    rz(phi).on(q1).controlled_by(q0),
                    ry(theta).on(q1).controlled_by(q0)])


class TianCircuit:
    def __init__(self, prime: int,
```

```
                 remainder: int,
                 input_string: str):
        self.q0, self.q1, self.q2 = cirq.LineQubit.range(3)
        self.prime, self.remainder = prime, remainder
        self.circuit = self.prepare_circuit(prime, remainder, input_string)
        self.simulator = cirq.Simulator()

    def prepare_circuit(self,
                        prime: int,
                        remainder: int,
                        input_string: str) -> cirq.Circuit:
        circuit = cirq.Circuit()
        length = len(input_string)

        # Need to compute correct angles for Ue
        theta_c, theta_e, phi_c, phi_e = get_angles(prime, remainder)

        # Initialise state of first channel to 1
        circuit.append([ry(pi).on(self.q0)])

        # Ue
        bs(circuit, self.q0, self.q1, theta=2 * theta_e, phi=phi_e)

        # # Uc ^ n
        bs(circuit, self.q1, self.q2, theta=2 * length * theta_c, phi=0)

        # Ued
        bsd(circuit, self.q1, self.q0, theta=2 * theta_e, phi=phi_e)

        # Revert
        circuit.append([CX(self.q0, self.q1)])

        circuit.append(measure_each(self.q0, self.q1, self.q2))

        return circuit

    def run(self, shots):
        return self.simulator.run(self.circuit, repetitions=shots)


def run_experiment(prime, remainder, input_string, shots=4096) -> List[
    float]:
    circuit = TianCircuit(prime, remainder, input_string)
    results = circuit.run(shots)
    probs = {state: sum(values.flatten().tolist()) / shots
             for state, values in results.measurements.items()}
    return [probs[key] for key in sorted(probs.keys())]

```

```
80
81 if __name__ == "__main__":
82     run_tian(run_experiment)
```

Listing Q.1: Cirq Implementation of Tian et al. (2019)