Simulating Massive Quantum Circuits Through Probability Filtering (QILCOM 2020 Model)

Authored by River Schieberl & Ian Baker. *Advised by Mingwei Jin*.

1 Introduction

Quantum computers are playing an increasingly important role in modern society and developing algorithms to run on these quantum computers is not trivial. For quantum algorithm creation, it is useful to simulate it running on a classical computer before testing it on a real quantum computer. Unfortunately, the modern method of simulating a quantum computer requires a large amount of time and memory so it is impractical to simulate large quantum circuits.

The output of a quantum algorithm is a probability distribution, the goal of the algorithm is to maximize certain values and minimize all of the others. For example, we can see the output of a 4 qubit quantum circuit here:

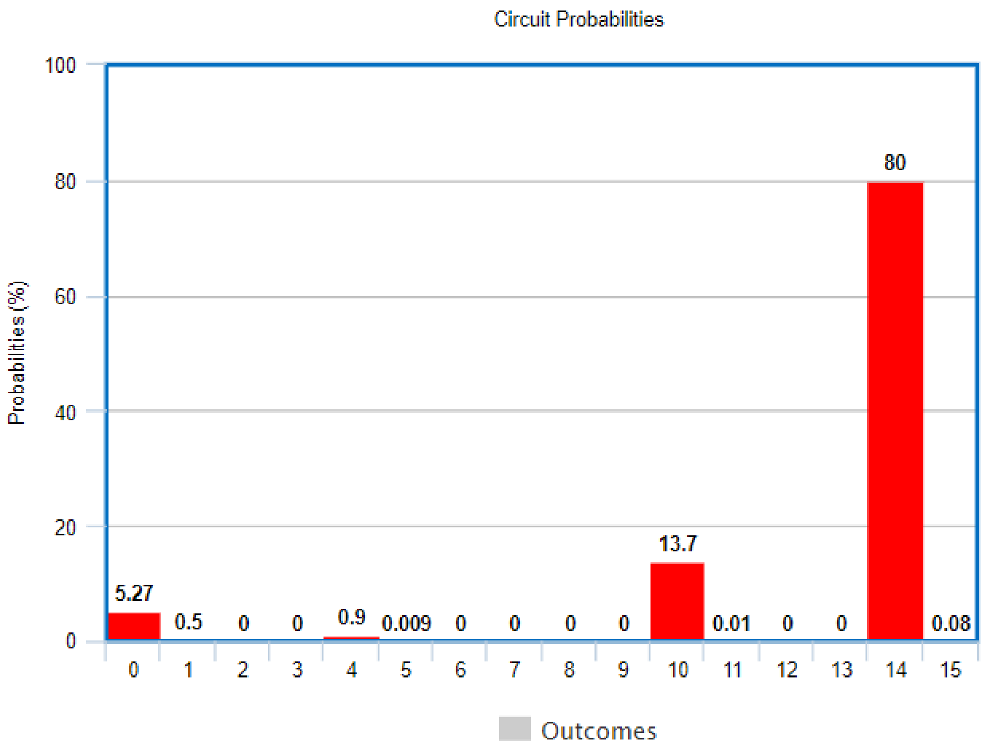


Figure 1: A Probability Distribution

The value 14 is extremely likely with 10 and 0 being somewhat likely outcomes as well. All of the other values are near zero. Nearly every quantum algorithm has an outcome similar to this, a handful of values are very likely and all other values are close to zero.

2 Approach

The modern method of simulating gate-based quantum circuits requires using 2^n complex numbers for n qubits to contain the quantum state. However, as we have seen in *Figure 1*, not all of these numbers are equally valuable. Instead of simulating the entire quantum state, it would be more efficient to just filter the probabilities and reveal the highest ones.

Previously, our team attempted this by a process called *deferred entanglement*, in which a circuit would be partitioned into two sections: rotation gates followed by a series of CNOT gates. After further experimentation, we have found that not all quantum states can be perfectly described by this method. However, we have also found that an *approximate state* can be created which estimates the original state.

2.1 Probability Estimation

The estimated state should have the following property: when the probabilities are sorted, the indexes of the sorted state should match. That is, the most likely outcome of the original state should still be the most likely outcome of the estimate state even if the exact probabilities do not match. To showcase this method, we will be using a randomly generated state as an example.

$$\underset{\text{Original State}}{\begin{bmatrix}0.28874\\0.60118\\0.03467\\0.07541\end{bmatrix}} \approx \underset{\text{Poor Approx.}}{\begin{bmatrix}0.27428\\0.51961\\0.07121\\0.13490\end{bmatrix}}$$

Looking at the above original and estimate states, it is clear this estimate is not amazing, but it does still have all the probabilities in the same order. That is, the largest is the 2nd index, then 1st, 4th, and followed by the 3rd index as the smallest. This estimate state also has the property that it does not use any entanglement, it is produced by a 2 qubit system with a $Y^{0.6}$ and a $Y^{0.3}$ gate.

This estimate state was found by guessing and checking, but having an algorithm to find the best possible estimate state would be helpful. To do that, we need a numerical definition of what the 'best possible' state looks like. There are a number of different ways to describe how close the two states are, but we have used a regression of sorts to accomplish this. This regression computes the sum of the difference squared between each element.

$$\sum \begin{bmatrix}(0.28874-0.27428)^2\\(0.60118-0.51961)^2\\(0.03467-0.07121)^2\\(0.07541-0.13490)^2\end{bmatrix} = 0.01174$$

For notation, we will call this regression value R. This value has similarities with the R^2 coefficient of determination from statistics, but is not exactly the same. For one, our value R is between $0 \leq R \leq 2$ for a 2 qubit system. The highest value of 2 is only possible when comparing different basis states, like $|00\rangle$ and $|01\rangle$. The lowest value of 0 occurs when the estimate is an exact match for the original.

To generalize this, we will use variables to represent the original and estimate states:

$$\underset{\text{Original State}}{\begin{bmatrix}a\\b\\c\\d\end{bmatrix}} \approx \underset{\text{Estimate State}}{\begin{bmatrix}qp\\q(1-p)\\(1-q)p\\(1-q)(1-p)\end{bmatrix}} = \begin{bmatrix}q\\1-q\end{bmatrix} \otimes \begin{bmatrix}p\\1-p\end{bmatrix}$$

The regression value R comes from the minimum of this function:

$$r(p,q) = (qp-a)^2 + (q(1-p)-b)^2 + (p(1-q)-c)^2 + ((1-q)(1-p)-d)^2$$
$$0 \leq p,q,a,b,c,d \leq 1$$
$$a+b+c+d = 1$$

We want to find the p and q which create the smallest output, the global minimum of this function. Luckily, this function behaves similarly to a 3D parabola with 1 local minimum, no maximum, no saddles, and no inflection axes. This simplifies the process of taking the partial derivative and setting it equal to zero.

$$\frac{\partial r}{\partial p} = 8p^2q - 4p^2 - 8pq + 4q + 4p + 2p(-a+b+c-d) + 2(-b+d-1) = 0$$
$$\frac{\partial r}{\partial q} = 8pq^2 - 4q^2 - 8pq + 4q + 4p + 2q(-a+b+c-d) + 2(-c+d-1) = 0$$

Finding the point of intersection between these two functions yields the minimum of r, which is our regression value R.

To visualize in 2D, these functions can be rewritten using $a = 1 - b - c - d$ as follows:

$$q = \frac{-2p^2 + p(1+2b+2c) - b + d - 1}{-4p^2 + 4p - 2}$$
$$p = \frac{-2q^2 + q(1+2b+2c) - c + d - 1}{-4q^2 + 4q - 2}$$

Notice these function are in terms of different variables, but they are nearly the same, one reflected over the $p = q$ line. Here is the graph which plots these two functions when the original state is the one shown above.

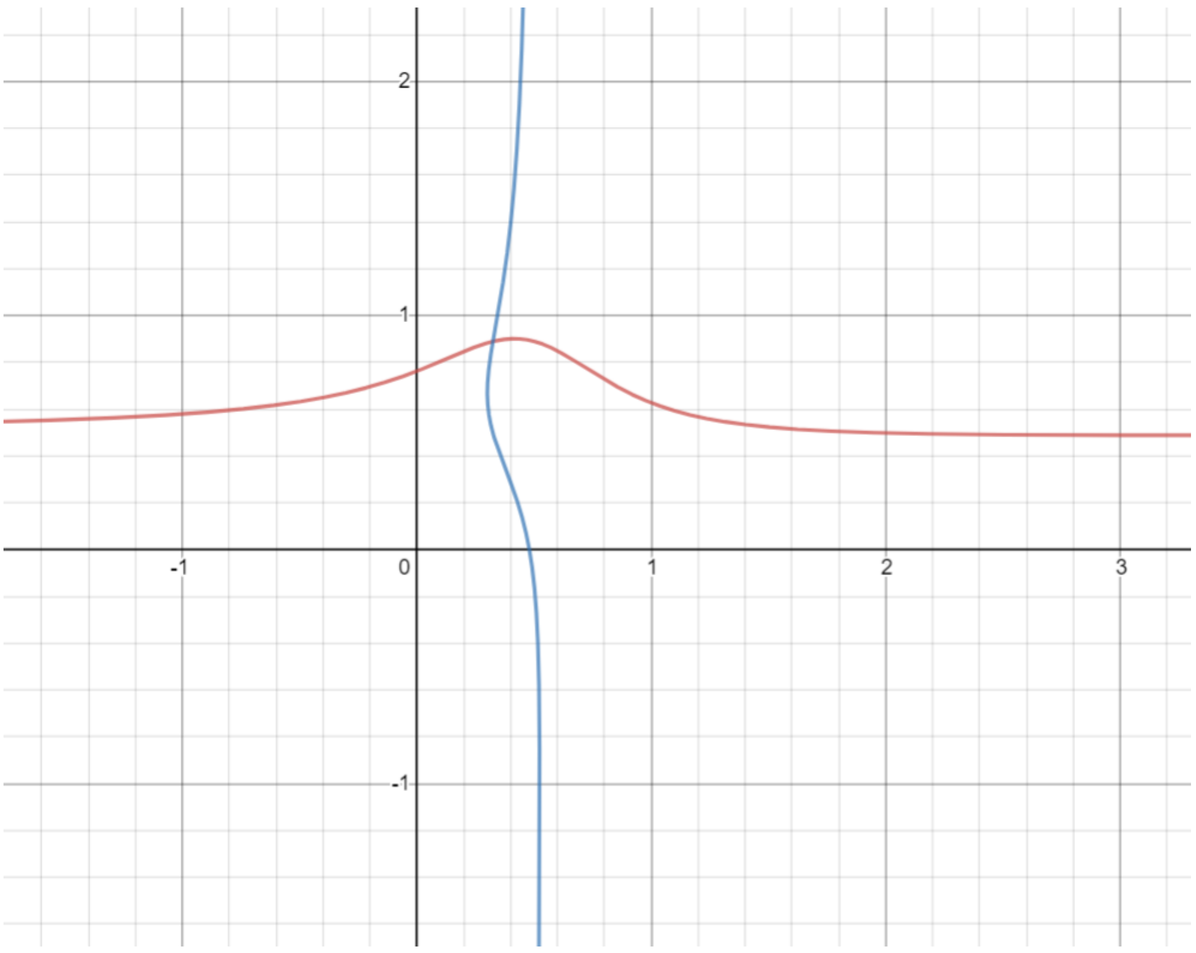


Figure 2: The intersection of the partial derivatives

The horizontal axis is p and the vertical axis is q. Because of the constraints on the function from above, there will always be exactly 1 intersection and it will always fall in the square with vertices $(0, 0)$ and $(1, 1)$. In this example, the intersection is at $(0.3242, 0.8895)$. We can easily find the rotations which create these probabilities:

$$\frac{2}{\pi}\cos^{-1}(\sqrt{0.3242}) = 0.614362 \to Y^{0.614362}$$
$$\frac{2}{\pi}\cos^{-1}(\sqrt{0.8895}) = 0.215727 \to Y^{0.215727}$$

So it appears the guess of $Y^{0.6}$ and $Y^{0.3}$ were not terrible, but there is a better match which creates the closer estimate state:

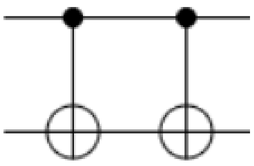
$$\underset{\text{Original State}}{\begin{bmatrix}0.28874\\0.60118\\0.03467\\0.07541\end{bmatrix}} \approx \underset{\text{Better Approx.}}{\begin{bmatrix}0.288376\\0.601124\\0.035824\\0.074676\end{bmatrix}}$$

The R value for this closer estimate is $r(0.3242, 0.8895) = 0.000002$, opposed to 0.01174 from before.

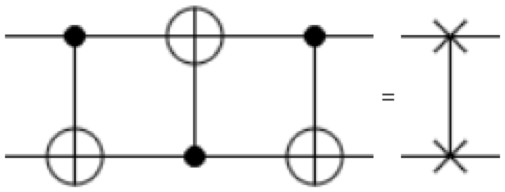
2.2 Adding CNOTs

The probability estimation described above is able to create the approximate rotations, but all of its estimates do not have entanglement. Entanglement is an incredibly important aspect of quantum circuits and cannot be overlooked. The CNOT gate is used to generate entanglement between qubits, so we can have our rotations from before and append a series of CNOT gates at the end of the circuit to generate entanglement.

The next step is to find an algorithm that can tell which CNOT gates must be applied to create the best approximate state. We can use some properties of the CNOT gate to create this algorithm. The first property is sequential CNOT gates of the same orientation form an identity.

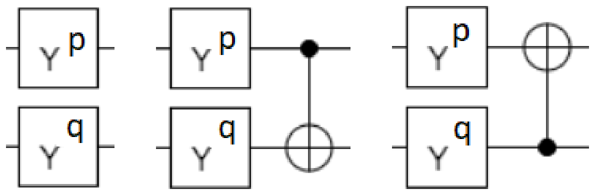


This means any CNOT gates which appear after our rotation gates must be alternating in orientation. Furthermore, 3 alternating CNOT gates forms a SWAP gate.



Since 2 SWAP gates form an identity, a series of alternating CNOT gates is periodic because 6 alternating CNOTs form an identity. Now, instead of an infinite number of CNOT gates in any orientation, we have narrowed the search space down to 0-5 alternating CNOT gates. We can further reduce the possible number of CNOTs by looking at the property of SWAP gates. Since 3 alternating CNOTs form a SWAP, we can remove the first 3 CNOTs by simply swapping the p and q of the rotation gates which essentially acts as a SWAP gate.

By using all of these facts about CNOT gates, there are now three possible scenarios: there are no CNOTs, there is one CNOT, or there are two CNOTs. The last reduction we will perform is to transform the two CNOTs into one upside down CNOT. Now, we know our original quantum state can be best estimated by one of the following three circuits:



This result can also be verified experimentally. We tested all of the different CNOT configurations, and every other CNOT sequence was a duplicate of one of these three.

2.3 Phase Estimation

With the addition of the CNOT gates, our approximation state can come very close to the original probability distribution. However, the probability is not the only part of the state which matters. The phase of the qubits is also a part of the quantum state. To create a true estimate of a quantum state, the phase also must be estimated.

The original state used in section 2.1 is actually the probability distribution, the real quantum state has a polar imaginary part which acts as the phase:

$$\begin{bmatrix}\sqrt{a}\,e^{i\hat{a}}\\ \sqrt{b}\,e^{i\hat{b}}\\ \sqrt{c}\,e^{i\hat{c}}\\ \sqrt{d}\,e^{i\hat{d}}\end{bmatrix} \approx \begin{bmatrix}\sqrt{qp}\,e^{i(\hat{q}_0+\hat{p}_0)}\\ \sqrt{q(1-p)}\,e^{i(\hat{q}_0+\hat{p}_1)}\\ \sqrt{(1-q)p}\,e^{i(\hat{q}_1+\hat{p}_0)}\\ \sqrt{(1-q)(1-p)}\,e^{i(\hat{q}_1+\hat{p}_1)}\end{bmatrix}$$

The $\hat{a}, \hat{b}, \hat{c}, \hat{d}$ are the phases for each coefficient in the original quantum state, we can approximate the phase by using the same regression idea from before:

$$\hat{r}(\hat{q}_0,\hat{q}_1,\hat{p}_0,\hat{p}_1) = (\hat{q}_0+\hat{p}_0-\hat{a})^2 + (\hat{q}_0+\hat{p}_1-\hat{b})^2 + (\hat{q}_1+\hat{p}_0-\hat{c})^2 + (\hat{q}_1+\hat{p}_1-\hat{d})^2$$

The same minimization technique is also used by setting the partial derivatives to zero. However, these $\hat{q}$ and $\hat{p}$ are the global phase; we are looking for the relative phase, so the actual values which matter are $\hat{q}_1 - \hat{q}_0$ and $\hat{p}_1 - \hat{p}_0$. They can be found by:

$$\frac{\partial \hat{r}}{\partial \hat{q}_1} - \frac{\partial \hat{r}}{\partial \hat{q}_0} = 0 \Rightarrow \hat{q}_1 - \hat{q}_0 = -\frac{\hat{a}+\hat{b}-\hat{c}-\hat{d}}{2}$$
$$\frac{\partial \hat{r}}{\partial \hat{p}_1} - \frac{\partial \hat{r}}{\partial \hat{p}_0} = 0 \Rightarrow \hat{p}_1 - \hat{p}_0 = -\frac{\hat{a}-\hat{b}+\hat{c}-\hat{d}}{2}$$

With this, we have created an algorithm which can estimate a quantum state with both probability and phase.

3 Results

In order to test how this algorithm handles actual quantum circuits, we generated 1000 random circuits which each consisted of 100 gates with mixed rotations and CNOTs. This is the distribution of R values:

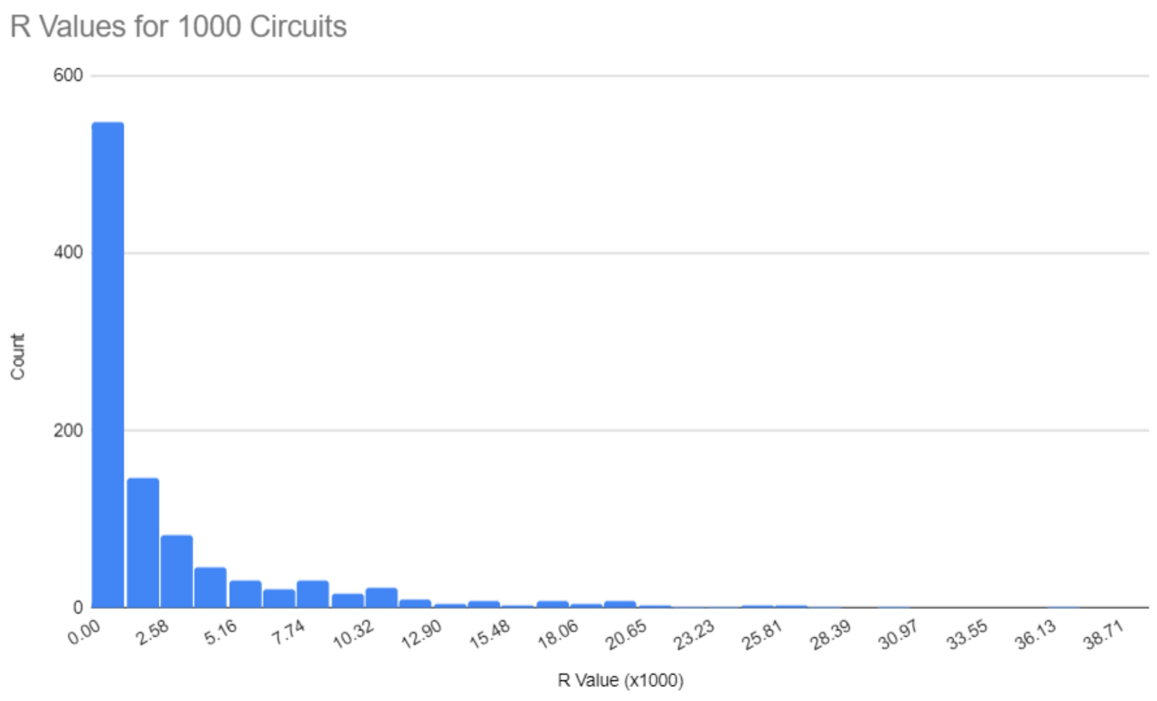


Figure 3: R value distribution across random circuits

The horizontal axis R values have been multiplied by 1000 for clarity. The largest shown is 40 which corresponds to a 0.04 R value. Over half of the random circuits are less than 0.001.

All basis states, all Bell states, and equal superposition have an R value of 0 since they can be represented perfectly, but the state which has the highest value is when three values are in equal superposition with one value at zero:

$$\begin{bmatrix} {}^{1}\!/_{3} \\ 0 \\ {}^{1}\!/_{3} \\ {}^{1}\!/_{3} \end{bmatrix}$$

This state has an R value of 0.043, but even still the order is preserved so that the zero value still has the smallest probability.

Accuracy is the most important factor to examine when looking at an estimation algorithm, but the time it takes to execute can not be overlooked either. We have taken the average of 10 executions for each circuit depth between 10,000 and 100,000 and graphed them.

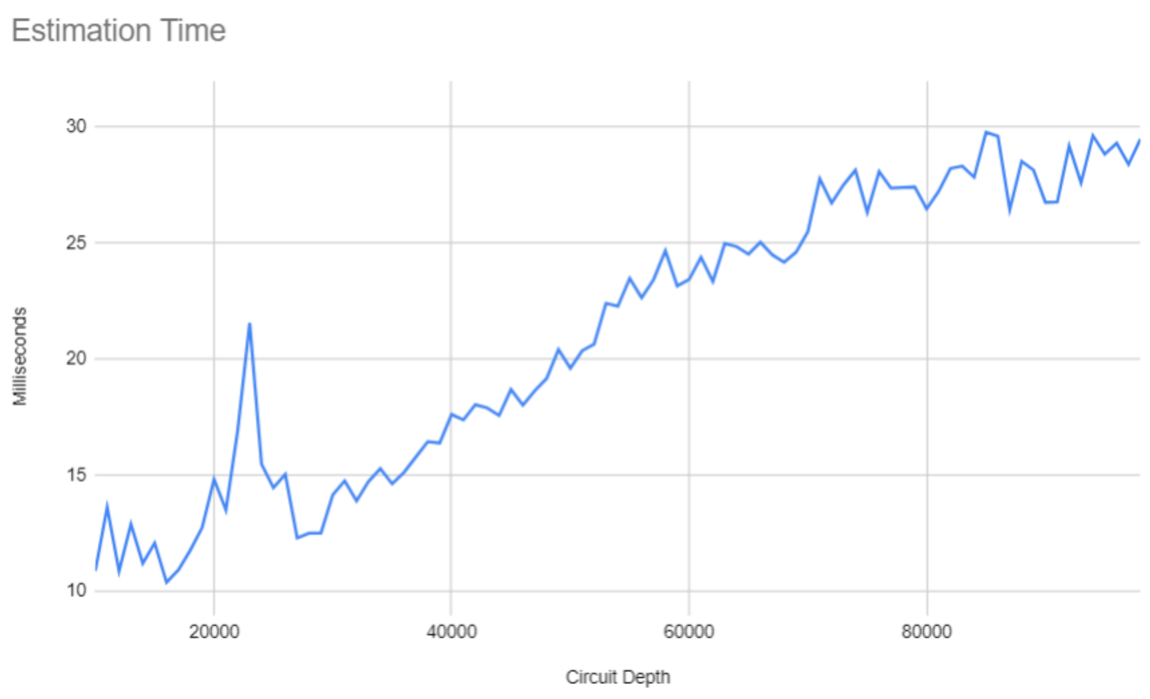


Figure 4: Graph of algorithm runtime

The runtime of the algorithm is roughly linear as a function of depth of the circuit, making it appear as $\Theta(n)$. These times are measured to include the generation, evaluation, and estimation of a random circuit. A circuit of depth 100,000 gates takes less than 30 milliseconds to execute. The space complexity is constant.

4 Next Steps

This model has shown that the idea of simulating quantum circuits by filtering the probabilities and finding the most likely outcomes can be accomplished by estimating the quantum state. The math discussed for deriving the algorithm and the code to execute it are based off of 2 qubit circuits. This is done for the model to show the idea works at a basic level before extending it to an arbitrary number of qubits.

In the next steps of the project, we would like to experiment with our algorithm and use it to find patterns between rotation and CNOT gates. We believe that we can improve the algorithm so that it is able to find the optimal approximate state by evaluating the quantum circuit sequentially instead of as a batch.

The algorithm also has many places to be improved in terms of efficiency. The algorithm's runtime appears fast, however this is because there is only 2 qubits. As we extend the algorithm for larger number of qubits, we expect the runtime to increase dramatically. Multithreading or even using a graphics processing unit may be necessary to decrease the runtime for larger circuits.

By analyzing the time breakdown of which aspects of the algorithm take the longest to run, we have found that locating the intersection of the partial derivatives in *Figure 2* takes the most amount of time. This is because we were unable to solve the intersection problem algebraically and found the point by performing a quarter-search on the domain. Finding a better method of locating the intersection would improve the runtime.

Once the algorithm is extended to higher number of qubits and we augment it to perform the estimation sequentially, the best method to test it is to compare it against the standard method of simulation. We will choose a quantum algorithm and run both methods on a variety of inputs to see which is faster and uses less memory.

5 Conclusion

Developing quantum algorithms is soon to be an incredibly large field of study and having a place to test these preliminary algorithms is critical to being an efficient developer. It is infeasible to simulate large quantum circuits using the modern method of simulation, so a new method is required. Nearly every algorithm maximizes a handful of values and has the rest near zero. Instead of finding the entire quantum state, it is more practical to locate just the most likely values.

The exact probabilities which come from a quantum circuit aren't needed, just the order of the outcomes when sorted by their probability. By estimating the original quantum state using an approximate state that has the same order, we hope to find the important values more efficiently. This model has shown that an algorithm to estimate the quantum state is feasible, fast, and accurate for the most basic 2 qubit randomly generated circuits.

By continuing to work on the project over the summer, we hope to extend our model to cover higher numbers of qubits, refine the algorithm, and have it run against other methods of simulation.

Code Appendix

The code for this project was written in Java. It contains 23 files and 2731 lines of code. The beginning of execution is the `main()` function in the Main.java file. The code files fall into one of five types:

- Tests. These files are used to perform various tests on the algorithms.
- Algorithms. These files contain the actual code which implements the math described in this document.
- Data Structures. These files contain the containers of data which the algorithms manipulate.
- Utilities. These files contain helper code which makes the rest of the project run smoothly.
- Deprecated. These files do not contain active code, but are left in to showcase the development process.

Here is a file breakdown by type:

- `Main` - **Test**. Contains high level tests for various experiments.
- `UnitTests` - **Test**. Contains low level tests for small components.
- `Const` - **Utility**. Contains constants used in the program.
- `circuit/*` - **Data Structures**. All of the files in the circuit folder are containers for circuits and gates.
- `exceptions/*` - **Utilities**. All of the files in the exceptions folder are utilities to notify the user if something failed.
- `misc/*` - **Utilities**. All of the files in the misc folder are utilities which help the code to become more readable.
- `simulators/Complex` - **Data Structure**. Container for complex numbers.
- `simulators/Simulator` - **Utility**. Parent class for several simulators to standardize code formatting.
- `simulators/fastsim/QuantumEstimator` - **Algorithm**. The algorithm responsible for implementing all of the math in this document.
- `simulators/fastsim/*` - **Deprecated**. The rest of the files in this folder are previous attempts which did not work.
- `simulators/matrixsim/MatrixSim` - **Algorithm**. The modern method of simulation.
- `simulators/qubit/Qubit` - **Deprecated**. An attempted method of simulated based on the Bloch Sphere.

Interactions

To run the project, we have pre-compiled the code into an executable JAR file. Ensure you have Java installed, then navigate to folder and run the following command:

`java -jar QuantumSimTest.jar`

Currently, the project is set to run the runtime experiment shown in *Figure 4*. However, it is easy to run other experiments by simply uncommenting and commenting lines of code inside the `main()` function.

For example, to input your own probability distribution in the form of 4 numbers, uncomment the `estimatorDirect()` function and change the float array on line 37 to the desired numbers. Currently it is set to the example used in this document. The output will be a link to https://algassert.com/quirk with the estimated circuit.

Another experiment which can be run is to estimate a circuit of your choice by uncommenting `inputEstimator()`. First, produce your own circuit on algassert.com and then copy the link and run the project. When it prompts you, paste it into the input field. The program will create an estimate and output a different link.

Of course, to run these other experiments you'll need to re-compile the project either by a Java IDE or on the Java command line interface.