Quantum Computer Simulator

Table of Contents:

1. Introduction

Quantum Computers are an emerging new technology that allows computation using quantum mechanical processes. Quantum computers have many key differences to classical computers and it is these differences that can give rise to some advantages. Example of some of these advantages is the ability to efficiently compute the factors of large numbers using Shor's algorithm, to compute NP-Complete problems in a square root of the time it would take on a classical computer using Grover's algorithm and to compute the Fourier transform of a system in exponentially quicker time than on a classical computer. Details about some quantum algorithms can be found in the appendix.

An interesting way to learn about how Quantum computers function on data is to simulate them on classical computers, in a so called quantum computer simulator. Quantum Computer Simulators are quite inefficient as simulating a quantum computer is generally considered an NP-Hard problem, however in this article I have attempted to try to create a more efficient simulator. I have three different models that I've attempted to implement in order to improve the efficiency of the quantum computer, the Independent Qubit Model, the Combinational Model and the Independent-Value model. I also have the Dependent Qubit model which is generally considered the standard implementation for a quantum computer simulator. Though all of these models led to unsuccessful implementations, the Independent-Value Model is still viable as a successful model, however given the time constraints I had I was unable to finish solving the problem of Phase Matching.

In general the biggest problem for Quantum Computers is that you need to take into account 2^n different values, where n is the amount of qubits. This is because each qubit is dependent on other qubits and each value is dependent on other values, thus one needs to store all possible values and qubit operations in order to successfully apply the model. The general concept of all three models I devised is to try to decrease the actual dependency of these values such that all these values don't need to be stored. The Independent qubit model attempts to get rid of all dependencies within the entire quantum system, and attempts to explore different methods to allow that to be feasible. The Combinational Model attempts to get rid of the dependency of separate qubits, but does not get rid of the dependency on separate values. While the Independent-Value model attempts to get rid of the dependency of separate values while retaining the dependency of separate qubits.

Chapters 2-4 attempts to explain the concepts behind a quantum computer, chapter 5 explains the basic standard implementation of a quantum computer while chapters 6-8 represent the models I devised, chapter 9 gets an overall conclusion as well as explorative opportunities for further research. The appendix contains many interesting details including the actual implemented code for the Independent Value Model.

2. What is a qubit?

The most important distinction between a computer and a quantum computer is that instead of classical bits a quantum computer uses quantum-bits or qubits for short. A qubit is the most basic and fundamental packet of information that a quantum computer runs off and, in the same way as a bit, is usually comprised of two states represented "0" and "1". However, quantum mechanics allows for different states to be mixed together in what is known as the wave-function of a system. It is impossible to know what state a quantum system is at until one measures the system, but the quantum system may be in any combination of states before being measured and only decides to be in exactly one of these states after it is measured. In this way, the wave-function represents a probability space of the quantum system being in any combination of states. A qubit is simply a wave-function that is limited to two mixed states centred around 0 or 1 (an example of physical wave-function with these two states is a particles spin, with spin-up being a 0 and spin-down being a 1).

It is sometimes easier to consider a qubit as a vector rather than a function. The qubit is a vector with two dimensions, a dimension for 0 and a dimension for 1. The qubit thus can be represented as

$$|q\rangle = a|0\rangle + b|1\rangle$$

Where $|0\rangle$ is simply the dimension "0", the symbolic notation is known as bra-ket notation but for all purposes this can be just considered a dimension. *a* and *b* represent weightings of each dimension, more so they are complex numbers that represent the probability that a qubit will be in a particular state at a particular time after measuring. Recall this is simply the wave-function of the quantum state, but to actually get information from the state we need to measure it and when we measure it the state can only be 0 or 1. The actual probability that the qubit will exist in a particular state once measured (say 0) is equal to the square of the weighting for that state (so for 0 the probability is a^2). Because all probabilities must add up to 1, this means the following relationship must hold.

$$a^2 + b^2 = 1$$

Multiple qubits can be, of course, stringed together into what's called a Quantum Register. One of the difficulties of simulating a quantum computer is the memory requirements that are created from a quantum register as the amount of dimensions doubles for each qubit (each possible value the quantum register can make is a new dimension). For example, suppose we have a quantum register with two qubits, the two qubits are in the following states.

$$|q_0\rangle = a_0|0\rangle + a_1|1\rangle$$
$$|q_1\rangle = b_0|0\rangle + b_1|1\rangle$$

Then the resulting qubit register would be in the state

$$|q_0\rangle|q_1\rangle = (a_0|0\rangle + a_1|1\rangle)(b_0|0\rangle + b_1|1\rangle)$$
$$= a_0b_0|00\rangle + a_0b_1|01\rangle + a_1b_0|10\rangle + a_1b_1|11\rangle$$

This distinction and doubling of memory becomes important as you will see later when we introduce multi-qubit logic gates.

3. Quantum Logic Gates

A computer is not very useful if all it can do is hold data, we also want the computer to operate on the data as well. In a classical computer we operate on bits using logic gates and, similarly, we do the same with quantum logic gates with a quantum computer. Unfortunately, there is a significant issue with quantum computers that make most classical logic gates impossible to create for a quantum computer. One of the most important theorems in quantum mechanics is the conservation of information, that within a quantum wave function information can never be lost or gained (this is why Black holes are such a puzzler for physicists, as according to current theories quantum information should be lost inside a black hole which violates this principle). Because information cannot be lost inside the wave function, this makes a lot of the logic gates such as AND, NAND, OR and XOR impossible to implement in a quantum world as information becomes lost when applying this logic gate. For example, in a XOR gate, if I give two inputs and output a 0, it is impossible to know if both the inputs were 0 or 1, information has been lost once applying that gate.

Thus all quantum logic gates must be entirely reversible with no information lost. Because Qubits are usually described as vectors, quantum logic gates are usually described as matrices that apply on these vectors. In this way quantum logic gates can be seen more as rotations rather than actual logic gates, each logical gate simply rotates the qubit in its state space. This way quantum logic gates work for a qubit in any arbitrary state.

One of the most important and useful quantum gates is the Hadamand gate, denoted *H*. This gate will take any qubit in a defined state (for example, in the |0> state) and then "quantumizes" it such that it becomes a superposition of the |0> and |1> state. The Hadamand gate is defined by the following matrix

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

To show how the quantum gate works, suppose we have a qubit in the following state (This is known as a pure state)

$$|q\rangle = |0\rangle$$

Writing the same thing but in vector notation makes the qubit

$$|q\rangle = \begin{pmatrix}1\\ 0\end{pmatrix}$$

Applying the matrix to the qubit results in the following

$$H|q\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

This qubit state is known as a Bell state. Thus the qubit has been transformed into an exact superposition between 0 and 1, measuring the qubit will have an equal chance of the qubit being in the 0 state as well as the 1 state.

Other important gates are the Pauli gates (X, Y, Z), particular X which is the quantum equivalent of the NOT gate (it swaps the probabilities of 0 and 1 around). The Phase-shift gate is also an important gate, it is basically the generalized version of the Z gate, in which is rotates the qubit in the complex plane. While this doesn't affect the qubits measuring probabilities, it changes the complex values of the qubit which cause it to behave differently under some particular quantum gates. A list of all the possible quantum gates with description can be found in Appendix A.

4. Controlled Gates and Quantum Parallelization

We have shown how data works in a quantum computer and how data operates on a single qubit, but how does data operate on multiple qubits and what makes a quantum computer more interesting than a normal computer?

Controlled Gates are the quantum computers equivalent of a multiple qubit gate, but once again these gates must be reversible. Control gates can be summarized as follows: I have a control bit and a target bit, I also have a single qubit gate. If the control bit is zero I leave the target bit alone, if the control bit is 1 I then apply the quantum gate to the target bit. In this way control gates are basically "conditional gates". There can be infinitely many control bits but only one target bit in any controlled gate operation.

The most important control gate is the C-NOT gate (or controlled-NOT gate). It is as it says, in that it is a controlled gate that, if the control bit is 1, flips the 0 and 1 probabilities around on the target bit. Basically it apply the Pauli-X gate if the control bit is 1. It can be shown that if one can implement all single qubit operations as well as a C-NOT gate, than the quantum computer becomes Turing Complete.

The C-NOT gate may seem relatively simple at first but one has to remember that we live in a quantum world and it is not necessarily known whether the control qubit will be 1 or not. Thus how can the computer know when to apply the gate to the target bit if it doesn't even know the value of the control bit? The conventional way to describe the control gate is to use the above notation for multiple qubits. The control gate turns the following system of two qubits

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_0|10\rangle + a_1 b_1|11\rangle$$

Into the system

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_1|10\rangle + a_1 b_0|11\rangle$$

Thus It flips the probabilities of the 10 and 11 state, but doesn't change the probabilities of the 00 and 01 state. The CNOT gate can be seen as an entangler gate is it allows one qubit to depend on another qubit, effectively allowing the quantum computer to go from "just a pile of qubits" to a working, fully functioning quantum register.

Another important aspect of Quantum Computers is the idea of quantum parallelization, which states that when you're in a mixed state, applying a quantum function to that mixed state applies that function to all possible values. This means any function can be computed on all possible values with only one application, though of course those computations are stored within the quantum state and only one at a time can be measured.

5. The Dependent-Qubit Model

In designing my quantum computer I went through several different models for how to model the quantum computer. The most basic model and the one I considered first was what I called the Dependent-Qubit model, which is generally the go-to method for quantum computer simulators. In this model, all possible values that a quantum register can be in are put into a large array, each value has its own unique probability, which is the probability of being in that value upon a measurement. This array of values can be considered as a value vector, thus applying a quantum gate would be equivalent to applying a large matrix to that value vector.

The amount of possible values that a quantum register can be in is 2^n where *n* is the total amount of qubits. This means that the quantum gate matrix would need to be 2^n x 2^n large, as each quantum gate needs to be applied to every possible value. For example if my quantum system was a 3 qubit system with probability 1 of being in state 000. I can then apply a Hadamard gate to all three qubits, then the system needs to have an equal probability of being in all states at once. One can simply measure a qubit by picking random number between 0 and 1, then starting from the 000 state, deducting the probability from that state and then moving to the next state until the number is less than or equal to zero. Whatever number caused our probability to be less than or equal to zero will be the value we measured.

5.1: Advantages:

The biggest advantage to the dependent qubit model is its simplicity and works well with how a quantum computer is often thought of. Applications are trivial and the implementation is simple. One can reduce the space requirements of the model by applying single qubit operations to each individual values instead of applying them via a matrix. For example in the three qubit system described above, upon application of a hadamard gate one may simply need to just apply a 2 x 2 hadamard gate to every pair of qubits, this avoids storing an 8 x 8 matrix, but requires 3 applications rather than just one. Altogether it lessens the memory requirements and doesn't really affect the time requirements so it seems like a simpler option.

5.2 Disadvantages:

In terms of memory and time complexity the Dependent qubit model is atrocious. It has an exponential memory capacity as all possible values need to be stored. It has an exponential time complexity to apply any quantum gate (controlled gates can be applied with 2^{n-c} comparisons, where *c* is the number of control bits. We can do this because we can consider the control bits as a mask and thus we only apply the operations to the values that apply the mask). Measuring is also exponential time, a measuring algorithm is basically to guess a number between 0 and 1, then go through each value individually and minus the probability of that value occurring from your guessed number. If your number gets to zero or below, you print the value that caused your number to get to zero, this will be your probability number.

So all important operations are exponential time and the memory requirements is exponential, resulting in a generally poor model to implement a quantum computer. Nevertheless I consider it a good starting point and all my other models are simply models to implement a better quantum computer.

6. The Independent-Qubit Model

My first attempt attempt at improving the Dependent qubit model was to consider each individual qubit as an independent system. By doing so we can construct the probability of every possible value based only on the probability of each individual qubit. For example, suppose we have a three qubit system and we want to find the probability of measuring the state 101, we can do this by simply multiplying the probability of the first qubit being in the state 1, with the probability of the second qubit being the state zero and with the probability of the third qubit being in the state 1. Effectively the probabilities would be the same.

Such a system has incredible improvements to both space and time complexity. The space complexity for the independent qubit model is $2n$ where n is the amount of qubits, as oppose to the dependent models 2^n, this is because we only have to store the individual probabilities of each qubit, rather than the probabilities for all possible values. Applying a single qubit quantum gate is also much more efficient as one only needs to apply a 2 x 2 matrix to the target qubit, because all values will use this target qubit applying this matrix to one qubit is effectively the same as applying the matrix to all possible values. This means that applying a single qubit gate goes from being exponential time to constant time. Measuring a qubits state is also linear time as opposed to exponential time, one can guess n numbers for each of the n qubits and choose that qubit to be in the state 0 or 1 based on each guess. The pseudocode would follow as such

```
int measuredValue = 0
FOR (int j = 0; j < n; j++) {
   int value = 1 << j;
   double p = Random(0, 1);
   if (p >= qubits[j].a^2) {
      measuredValue += value;
   }
}
return measuredValue;
```

Clearly this independent model has some impressive advantages, unfortunately the model turns out to be impossible without sacrificing the speed and efficiency of the model.

6.1: Control Gate Problem:

The problem with the Independent qubit model occurs when we start attempting to apply controlled gates. Controlled gates require that one qubit needs to depend on another qubit which inherently contradicts the design of the independent qubit model which states that each qubit must be entirely independent. All control gates can be made from single qubit gates and a CNOT gate, we already know that each single qubit gate works, so now we only need to worry about the CNOT gate. Recall that a CNOT gate transforms the system

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_0|10\rangle + a_1 b_1|11\rangle$$

Into the system

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_1|10\rangle + a_1 b_0|11\rangle$$

When the left most qubit is the control bit and the right most is the target bit. Because in the independent qubit model, each qubit acts as its own isolated system, this means the second

qubit must flip its sign depending on the first qubit. Because each qubit is considered independent and not dependent on values, it would seem that in the above case the right most qubit needs to store TWO different values, a value where the leftmost qubit is one and a value when the leftmost qubit is zero. This could potentially lead to a doubling of information per CNOT gate application which would be problematic. I attempted to solve this problem in a variety of ways, all of which were unfortunately unsuccessful.

6.2: Solution Attempt 1: Probability Theory

One of the more promising methods I tried to apply was applying probability theory, if you have two qubits in the following state

$$|q_0\rangle|q_1\rangle = (a_0|0\rangle + a_1|1\rangle)(b_0|0\rangle + b_1|1\rangle)$$

If we apply a C-NOT gate to this system (with the first qubit as our control), what is the probability that the second qubit will go into the 0 state? Well it is the probability that both qubits will be 0 plus the probability that both qubits are 1, in fact the probability changes to

$$|q_0\rangle|q_1\rangle = (a_0|0\rangle + a_1|1\rangle)\big((a_0b_0 + a_1b_1)|0\rangle + (a_0b_1 + a_1b_0)|1\rangle\big)$$

This is interesting because if we could prove this relation to be true, then we could effectively use controlled gates in O(1) time using 0 extra memory! Although this seems to work for Bell-states (exactly half way between 0 and 1) and pure states (Either only 0 or 1), it doesn't work for arbitrary states and it's easy to see why, we failed the rule of probability! When we square the result of both qubits, we get the following

$$(a_0b_0 + a_1b_1)^2 + (a_0b_1 + a_1b_0)^2 = 1$$

When we expand the brackets we get the following

$$(a_0b_0)^2 + (a_0b_1)^2 + (a_1b_0)^2 + (a_1b_1)^2 + 4a_0b_0a_1b_1 = 1$$

However, we know that in a two qubit system the following identity must be true

$$(a_0b_0)^2 + (a_0b_1)^2 + (a_1b_0)^2 + (a_1b_1)^2 = 1$$

This means that the identity only holds when

$$4a_0b_0a_1b_1 = 0$$

Which will not always be the case (this is only true in pure states, which is one of the reasons why it works for pure states)! Clearly we are missing some kind of normalization factor. If we replace the two qubit with the following system

$$|q_0\rangle|q_1\rangle = (a_0|0\rangle + a_1|1\rangle)\left(\frac{(a_0b_0 + a_1b_1)}{k}|0\rangle + \frac{(a_0b_1 + a_1b_0)}{j}|1\rangle\right)$$

Where *k* and *j* are arbitrary complex numbers. And we assert that the following identity must be true

$$\left(\frac{(a_0b_0 + a_1b_1)}{k}\right)^2 + \left(\frac{(a_0b_1 + a_1b_0)}{j}\right)^2 = 1$$

We then expand the brackets of our system we note that the following four equations must also hold in order for our control qubit system to correctly simulate a real quantum system (Based on the quantum system above)

$$a_0{}^2\left(\frac{(a_0b_0 + a_1b_1)}{k}\right)^2 = a_0{}^2b_0{}^2$$

$$a_1{}^2\left(\frac{(a_0b_0 + a_1b_1)}{k}\right)^2 = a_1{}^2b_1{}^2$$

$$a_0{}^2\left(\frac{(a_0b_1 + a_1b_0)}{j}\right)^2 = a_0{}^2b_1{}^2$$

$$a_1{}^2\left(\frac{(a_0b_1 + a_1b_0)}{j}\right)^2 = a_1{}^2b_0{}^2$$

And we note that the following conditions are also inevitably true by definition

$$a_0{}^2 + a_1{}^2 = 1$$
$$b_0{}^2 + b_1{}^2 = 1$$

It is provable that any two variables *k* and *j* cannot satisfy these equation. If we take the first two equations and divide both sides by the left most value we get the following conditions

$$\left(\frac{(a_0b_0 + a_1b_1)}{k}\right)^2 = b_0{}^2$$
$$\left(\frac{(a_0b_0 + a_1b_1)}{k}\right)^2 = b_1{}^2$$

no constant can satisfy both those equations unless

$$b_0{}^2 = b_1{}^2$$

Which generally won't be the case (only true in Bell-states which is why the solution actually worked in Bell states). Thus it is impossible to use this method to solve the independent qubit problem.

6.3 Solution Attempt B: Post-Measurement Tweaking

The other solution will be to lazily evaluate the problem. In this solution all the C-NOT operations are saved in a list and once the quantum register is measured, we apply these operations post-measurement. We once again suppose we are in the following state

$$|q_0q_1\rangle = a_0b_0|00\rangle + a_0b_1|01\rangle + a_1b_0|10\rangle + a_1b_1|11\rangle$$

After the C-NOT gate has been applied the result should be

$$|q_0q_1\rangle = a_0b_0|00\rangle + a_0b_1|01\rangle + a_1b_1|10\rangle + a_1b_0|11\rangle$$

In the lazy evaluated method, we say that we measure both bits, if the control bit is "1" we flip the target bit retroactively. Thus if the left most bit is evaluated to be 1, we then swap the right most bits with the other qubit in question, which is exactly what we need in order for our C-NOT operation to work, and we haven't been required to use extra memory!

Unfortunately, this simply cannot work. We've had to measure the quantum bit in order to compute our C-NOT instruction, which means we lose the ability to apply single Qubit gates between the C-NOT instructions. We can see this by modifying the phase of the a qubit after applying a C-NOT gate. If we have two qubits, the first qubit is in the 0 while the second qubit is the in the state 1, we have the following system

$$|q_1q_0\rangle = 1|10\rangle$$

If we then apply a C-NOT gate operation where the first qubit is set. Ideally we should now be in the state

$$|q_1q_0\rangle = 1|11\rangle$$

Notice that by lazily evaluating we can actually get into this state by measuring the 2nd qubit first and then flipping the first qubit if it is measured. Let us then apply a PHASE-gate to the first qubit, a PHASE gate is defined with the following matrix

$$P = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$$

Applying this to the first qubit would result in the phase

$$P|q_0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ e^{i\phi} \end{pmatrix}$$

Thus the first qubit has been modified. However, when we look at what happens in our "lazy-evaluation" model we don't actually flip the qubit until the qubit is measured. This means that the qubit remains in the state

$$|q_0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$$

And thus applying the phase gate will result in

$$P|q_0\rangle = \begin{pmatrix}1 & 0\\0 & e^{i\phi}\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix}$$

So thus the phase isn't changed. This is actually quite a subtle difference, but is an important difference, the phase of a quantum bit doesn't change its measured value (you'll notice both methods still result in the same value), but they do change how the qubit is affected in other quantum gates (which can lead to a change in its measured value). For example, the quantum Fourier transform (I bet Aleks's favourite) will not work without being able to modify the phase of the qubit. Thus this method won't work as it doesn't preserve the phase of the qubit.

6.4: Solution Attempt 3: All combination store

My final attempt at solving this problem was to try and store all possible values that the qubit could be once a C-NOT gate has been applied to it, and then simply pick which is the correct solution. This seems like a terrible idea, as the amount of possible values that need to be stored is exponential with respect to the amount of controlled gates that need to be applied. The memory becomes 2^g where g is the amount of controlled gates being applied. Considering g is usually much greater than n (the amount of qubits) this would seem worse. It is impossible to reduce the total amount of values that need to be stored.

7. The Combinational Qubit Model

The Combinational qubit model was an attempt at a compromise between the dependent and independent qubit model. The Combination qubit model was a model that attempted to speed up the dependent model, however it did not attempt to decrease the space requirements for the quantum register. Like the dependent qubit model the combinational qubit model stores every single value into a single array of size 2^n, however each value in the array does contain a complex number, instead each value in the array contains a list of references. The references refer to a specific qubit probability and the entire list multiplies to the total probability of that value being in that state. The concept was that the probabilities of different values are usually quite similar or the same and are once again fully described by the probabilities of different qubits. Thus one only needs to store the probabilities of different qubits and for each value store a reference to specific probabilities for those qubits.

For example, suppose one has a two qubit system and they want to find the probability of the state 10. The probability would be a list of two references, one reference leading to the probability of the first qubit being in 1 and one reference leading to the second qubit leading to 0. The advantage of this model over the dependent qubit model is that single qubit gates can be done in constant time rather than exponential, as one only needs to apply a 2x2 matrix to one value and all references for that value will automatically be updated, thus allowing a single application of a qubit to be constant time as opposed to exponential for the dependent qubit model.

In this model control qubits can also be applied, one just simply needs to swap the references around. For example suppose you have the following system

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_0|10\rangle + a_1 b_1|11\rangle$$

If you're applying a CNOT gate to the system and using the left most bit as a control, one can just swap the references of the values around to get the following result.

$$|q_0 q_1\rangle = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_1|10\rangle + a_1 b_0|11\rangle$$

This is exactly the same result as if you were actually applying the CNOT gate, so CNOT gates can be successfully applied in this model, though they are done in 2^{n-c} time, with *c* being the amount of control bits in the control gate. Unfortunately the model does suffer from slightly worse memory than the dependent qubit model, with memory akin to $n2^n$ due to the list of references per value.

7.1: Phase Conservation Problem

Although it may seem like a CNOT gate seems to work in this model, it unfortunately does not, as the phase is not conserved when a CNOT gate is applied for the same reasons as the phase was not conserved in Solution B: Post-Measurement Tweaking of the Independent Qubit model (see that section for a full overview). Basically, when a CNOT gate is applied the probability of the value changes but the qubit remains unchanged, this can come to a situation in which a phase gate cannot be applied. An example is the following code.

The

```
  static void Main(string[] args)
        {
            QReg reg = new QReg(2);
            reg.ApplyGate(Gates.X, 1);
            reg.ApplyCNOTGate(0, 1);
            reg.ApplyGate(Gates.PhaseGate(1), 0);
            reg.ApplyCNOTGate(0, 1);
            reg.ApplyGate(Gates.X, 1);
            reg.PrintQuantumState();
            reg.MeasureState();
        }
```

Upon conclusion of the code the only value should be the 00 value, but it should have a modified phase. This code is the equivalent of a controlled phase gate. One can observe while this would occur. We go through the entire matrix solution for our first qubit, we have

$$\binom{1}{0} CPC = \binom{0}{1} PC = \binom{0}{e^i} C = \binom{e^i}{0}$$

In the Combination Qubit Model however, none of the control qubits actually change the state of the qubit, only the references for the values, which leads to the following result

$$\binom{1}{0} CPC = \binom{1}{0} PC = \binom{1}{0} C = \binom{1}{0}$$

Because the references change, the actual value of the qubit is the same at all points of the programs, but the phase is not conserved due to the qubit actually not changing. This can cause a significant problem for some qubit operations, thus this problem is considered fatal for the combination qubit model.

8: Independent Value Model

This model gets rid of the exponential memory requirements and instead replaces them with a linear memory requirement, with hopefully not too much loss in time complexity. The model begins like this. When we apply a quantum gate, we don't actually apply the quantum gate, instead we save it and say that this quantum gate is going to be applied later. When we measure the qubit, we apply all the quantum gates at once and measure just a single value of the resulting probability space. Can we measure the value without happening to store the entire probability space? If we can do this, we can then just pick a random number between 0 and 1, then go through random values and find the probability for that value being the case, we then continually add up these probabilities until we get to our random number, we then print that as the value we want to measure. The Pseudocode would work as follows

```
P = random(0, 1);
K = 0;
While (P > 0) {
  P = P – CHECK_PROBABILITY_OF_VALUE(K);
  K++;
}
PRINT K;
```

So how can we calculate the probability of a single value without copying the entire values vector? Well it turns out to do so is difficult, but may not impossible. We first start off using the same concept as the independent qubit model, that is we consider our entire array of qubits and each qubit has a probability of being in the 0 state and being in the 1 state. We then apply our quantum gates on these qubits. We can split the single qubit gates into three categories, Phase gates, Swap gates and Hadamard gates, we'll cover the third one later in this section. Phase gates are single qubit gates that merely modify the phase of a qubit without modifying the probability distribution of that qubit, two examples of this are the Z gate and the Phase gate, these are trivially applied directly to the target qubit. Then there are swap qubits, these qubits replace the 0 probability with the 1 probability and vice versa such as the X and Y gate. The problem with swap gates is that they require that I know the probability of both values, for example suppose I am in the state 01 and I want to swap the left qubit, the probability of 01 needs to be replaced with the probability of 11, which violates the independent value model, the values are no longer independent and we've had to calculate more than one value! We can do something very clever however to solve these problems very simply.

Suppose we want to measure some value *v*, we don't actually care if we measure the value of *v*, we just want to measure the probability of our qubit being in some state. If we apply a swap gate, instead of swapping our qubit and dealing with a bunch of trouble, the easiest method is to actually swap the value we are measuring. For example, suppose the value we want to measure is the 01 state, and we've applied a Pauli-X gate on the left most qubit, instead of swapping the probabilities of the state we can simply say we are now measuring the 11 state, and then swap our probabilities around to reflect that. By changing the value we are measuring, rather than just the probabilities, we allow for a very interesting application to be applied, we can do CNOT gates and we can do them very well! To understand this, suppose we are measuring the state 01 and we apply a CNOT gate where the left bit is the

target bit and the right bit is the control bit. We know what value we are checking so we KNOW the control bit is a "1" which means we know we can swap the left qubit. We are actually applying the correct gate to the CNOT gate so we don't get into problems with phase conservation and we apply the CNOT gate at the appropriate time. By swapping the value rather than the probabilities we're able to calculate the probability of a single value without requiring to calculate the probability of other values

It doesn't matter that the value is not preserved, if we input "0" as the value we want to measure, and we get the probability for the value "156" instead, it doesn't matter because we're not interested in the actual value that's being measured, we're interested in the probability only. Suppose we have our function for calculating the probability, even though the value we're measuring is different than the value we initially started with it is still useful because the probabilities are still preserved and the probabilities are still injective if and only if the initial probability distribution is also injective. This means that, if we calculated the entire probability distribution of all the qubits and found that each qubit value has a unique probability, then using this method you would also find that each qubit value would have a unique probability because a C-NOT gate only swaps probabilities, it does not duplicate them! In fact duplicating probabilities never happens, this is known as the No-cloning theorem, so this little quirk of quantum mechanics actually allows this method to be possible!

We now have the ability to find the probability of any one single value without happening to save the entire probability space, the question now becomes is this method more efficient and can this method be more efficient? If we used the method naively, measuring would be less efficient than that of the dependent model because not only do we have to find the probability of every value, but we have to also compute the probability of every value (which means we have to run the entire quantum logical circuit 2^n times!). However, it should be noted that actually applying the gates on the quantum register is done in constant time, as opposed to exponential time, so overall if you are only measuring once the time complexity is the same, if you're measuring multiple times it may be slightly less. This may seem bad but actually we can make this quite efficient. Instead of applying the quantum gates one after the other, we can realize that the quantum gates themselves are all 2 x 2 matrixes. We can just multiple all the matrixes together and thus we can have a simple matrix application per value check, which is a relatively small overhead!

Though technically this is not actually true, once again because of our C-NOT gate, we are unsure of what the C-NOT matrix will be until we know what value we're trying to measure, so we can't multiply C-NOT gates together, we have to keep them. Luckily, a C-NOT gate is a relatively small simple overhead, it is a simple swap operation which is constant time, thus a very large series of C-NOT operations would be required to cause a significant overhead. Thus the overall worst case time complexity for measuring naively becomes $O(g2^n)$ where g is the amount of C-NOT operations. Is this extra g factor worth the exponential decrease in memory? Before we do that, we must ask can we actually do this method?

8.1: Proof of Correct Probability Distribution

A question that must be asked is will the probabilities be the same? If I measure the quantum register and get a value, is the probability of me getting that value correct based on how we measure? To answer this we need to use probability theory. We suppose that we pick a random number *p* that is between 0 and 1. We have a function *F(v)* that gives the

probability of our value *v* where *n* is the number of possible values. The probability of getting of any value should be equal to

$$P(v) = F(v)$$

Proving this is difficult as the equations are complicated, but we can prove it! We first ask ourselves what is the probability of having *v* as our first guess, we'll call this function $G_0(v)$. The probability of getting *v* in the first guess is the probability of guessing *v* multiplied by the probability that *p* is less than or equal to *F*(*v*). That is

$$G_0(v) = \frac{F(v)}{n}$$

So what is the probability of getting a guess in our second try? Well the probability is when we guess some value that has its probability less than *p* (otherwise that value would be chosen) and then guess the correct value. We also need to assert that the probability of this value should be greater than $p - F(v)$ otherwise when we guess the correct value on the second try, we will move onto a third try and won't guess our value. The range of possible *p* that this inhabits is simply $F(v)$! There are $n - 1$ possible combinations of values to choose from so the total probability of guessing *v* on the second try is

$$G_1(v) = \frac{(n-1)F(v)}{n(n-1)} = \frac{F(v)}{n}$$

Finding the value *v* on an arbitrary *k*th guess is simply the chance of picking *k* times randomly and getting to value *v*, multiplied by the chance of getting a *p* such that *k* tries does not pick *v* but one more try will pick *v* multiplied by the amount of possible subsets of length *k* in the remaining $n - 1$ elements. The permutation gives the amount of subsets of length *k* are in our set of length n - 1, $F(v)$ gives the probability of us getting the correct *p* (using the same logic as before) and $n(n-1)(n-2)\ldots(n-k)$ is the probability of picking *k* values with *v* as the last value. Thus we have our final result as

$$G_k(v) = \frac{(n-1)!}{(n-k-1)!}\frac{1}{\prod_{j=0}^{k}(n-k)}F(v)$$

We note the clear relationship

$$\prod_{j=0}^{k}(n-k) = \frac{n!}{(n-k-1)!}$$

Subbing that in we get

$$G_k(v) = \frac{(n-1)!}{(n-k-1)!}\frac{(n-k-1)!}{n!}F(v)$$

Which simply equals

$$G_k(v) = \frac{F(v)}{n}$$

Now that we've figured it out for an arbitrary case, the remaining problem is relatively simple. Clearly the probability of getting to a certain value *v* is the addition of all possibilities for getting to that value under all possible steps. That is

$$P(v) = \sum_{k=0}^{n-1} G_k(v)$$

Expanding out we get

$$P(v) = \sum_{k=0}^{n-1} \frac{F(v)}{n}$$

Which simply gives

$$P(v) = F(v)$$

Which is exactly what we wanted. Thus if we pick our values randomly, we can get the exact same probability.

Will the probability given always be the same probability? It is clear that by applying all the single-qubit gates we can easily get the same probability but it is the controlled gates we are interested in. Previously in the article I suggested that if we apply a C-NOT gate, we simply see if the control bit is one and if it is, then we switch our value (rather than our probability), but this won't exactly works as we're missing a simple step. Recall the problem with the Combinational Qubit model in that when applying a C-NOT gate, the actual qubit values don't change and because of that future qubit gate operations will not give the same value as they would in a real quantum computer. The same problem would occur if we simply switched the value and didn't change the probabilities when a C-NOT gate is applied, but we can circumnavigate that problem by doing one simple thing, just apply the Pauli X gate to the target bit as well.

When we apply a C-NOT gate we switch the value so that we don't have to recalculate a brand new value, the new value we're calculating is one where the target bit has been flipped. However, in doing so we actually have to switch the probabilities to actually display that flip, in the sense that suppose we switch the target bit from a 0 to a 1, we actually have to replace the probability of getting a zero with the probability of getting a one as well. This means that the probabilities are correctly and perfectly managed and the actual measured probability will be the same, so this can actually be used to apply any quantum gate operation. This solution is perfect!

How can we decrease the amount of values that we need to check? A simple way to do this is to check our worst case scenarios. Our worst case scenario is if our probability that we randomly pick is "1", using the above pseudocode algorithm, we would need to check all values and pick the last value on our list. However we can get some information based on our values that we measure. For example, suppose we measure a value and we find that every single qubit in that value has a probability of "0" to be in that state, that would imply that the only possible value would be the all those qubits flipped. It's important to note that this isn't NECESSARILY the case, but by analysing the probabilities of each value, we can make an educated guess on what type of probability distribution we are getting to, and thus we can make educated guesses on what is the most probable value and thus the most likely to get our p to 0 (thus minimizing the guesses we make). It is provable that any arbitrary path of guesses will lead to the same probability distribution when guessing a random number between 0 and 1, though it is not necessarily true that "educated" guesses will lead to the same probability distribution, this would be done for further research.

Another advantage to the Independent Value Model is that measuring a value can be done concurrently. This is important because the Dependent Qubit Model couldn't be done concurrently as all the data had to be stored in a massive array that is not easily split up into regions due to dependencies, thus different threads would need to communicate with each other very consistently. In our measuring process, we've shown that it is irrelevant the path one chooses to measure the values, so we could have different threads checking different regions and all simply adding or subtracting from a shared probability. This requires much less communication between processes and thus is much more easy to create a concurrent solution.

8.2 The Hadamard Problem:

We talked about how the Independent Value model works with Phase gates and Swap Gates, but a problem occurs when we try to apply the third option, Hadamard gates. Hadamard gates themselves are complicated, instead of swapping the gate one needs to know both probabilities of both values in order to solve the solution. This generally comes down to two problems, qubit entanglement and qubit phase matching.

8.3: Solution A: Entanglement Matrix:

Entanglement is a relatively complicated niche of Hadamard gates that causes trouble for our independent value model. The simplest explanation of entanglement is in the following process, suppose we have a two qubit system and in our system we apply a Hadamard gate to the 0^{th} qubit, then a control gate where the 0^{th} qubit is the control qubit, and then apply another Hadamard gate to the 0^{th} qubit. The system follows this system measurement (recall the dimensions of each vector are 00, 01, 10 and 11)

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}H_0C_{01}H_0 = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\0\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & -1\end{pmatrix}C_{01}H_0 = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\\0\\0\end{pmatrix}C_{01}H_0$$

$$= \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\\0\\0\end{pmatrix}\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\end{pmatrix}H_0 = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}H_0 = \frac{1}{2}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & -1\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1\\1\\1\\-1\end{pmatrix}$$

Thus the application of these three gates should lead to the system

$$\frac{1}{2}\begin{pmatrix}1\\1\\1\\-1\end{pmatrix}$$

However under our dependent qubit model, we simply apply the Hadamard gate to that specified qubit, applying the CNOT gate doesn't change the value of the hadamard gate, we then apply the Hadamard gate again but this simply applied the Hadamard gate to that single qubit again, without any form of entanglement of the values this just reverts the hadamard gate back to its original state which gives us our original value of

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$$

The fix for this however is actually surprisingly simple, we can pretend the qubits become entangled by storing an entanglement matrix. An entanglement matrix is simply just a matrix with *n* columns and *c* rows where each row corresponds to a CNOT gate application. Each cell is a mapping from qubit to qubit, that when a Hadamard gate is applied it is applied to whatever qubit has that mapping. By default the mapping of the entanglement matrix is just mapping every qubit to itself, however when one applies a CNOT gate to a qubit that is in a mixed state where that qubit is the control bit, we then say that the control qubit and target qubit of that state are entangled. We then map the control qubit to the target qubit and say that if we are to apply a Hadamard gate to the control qubit while it is entangled, we instead apply it to the target qubit via the entanglement matrix. Doing so actually results in the exact same output given that same operations for that above matrix. And using empirical testing with the Quantum Computer Playground, it seems that this entanglement matrix replicates all Hadamard gate entanglement issues perfectly, though I have not proven this. The

entanglement breaks when another CNOT gate is applied or the qubit goes out of a mixed state.

8.4: Solution B: Phase Matching:

The other, perhaps more difficult problem, is that of phase matching. If I have one value that has some phase *p* and I have another value who has some phase *q*, then Hadamarding these two values should result in these phases being added. This is generally not a problem for normal Hadamard gates because when we're applying the phase to the same qubit we can assume that the phase we apply will be the same for all values for that qubit. However, when applying a CNOT gate this may not necessarily be true, the phase may change (by simulating a C-PHASE gate) based on the value. Because the latent checking model considers each value independent, when hadamarding two values together the phase of both values may not be considered and thus the final phase may not be the two phases added together!

This problem is particularly difficult because it relies on that fact that we should know the phase of a different value, and calculating a phase for that value may mean we have to run the entire circuit on that value which violates the independent value model! It can be easily shown that if one value depends on the value of another value, then you can always get into a situation in which one must calculate every single value in order to calculate a single value, which would make independent value model no better, and perhaps even worse, than the dependent qubit model.

One method I attempted to try and implement was to say that whenever a phase gate was applied, I would store what phase it was applying but I wouldn't actually apply the phase unless it would normally be applied. In this way every qubit has a list of all the possible phases that it could in, it is stored as a list of phases plus a list of Booleans of whether or not that phase would be applied. The hope was that, by knowing all the gates that were being applied, one could deduce the exact phase that this value would need to be from the list of possible phases, without happening to calculate any other values. Unfortunately I did not have enough time to explore this fully and the problem is quite a niche and difficult one. I did prove it was possible to deduce the exact phase, but the method I was using required a potential exponential amount of applications, so whether one can do the deduction in a reasonable amount of time is still a question that hasn't been answered.

9. Conclusions:

Making quantum computers more efficient is a difficult problem! The sheer memory requirements that are needed to store all those values is perhaps the biggest problem facing quantum computers at the moment. In the dependent-qubit model I could only simulate 24 qubits with 16 GB of RAM, also the overhead for initializing that much memory in a massive array is also intense with it taking up to 5-6 minutes to initialize that memory array, and several more minutes to apply the quantum logic gates. If we, however, assume that the Phase matching problem of the independent value model can be fixed with a non-exponential amount of memory, then the memory for our independent-value model could be much better. Without phase matching, I was able to simulate 26 qubits in 3 minutes using only 16 MB of RAM for a relatively complex quantum circuit. Given a day, a quantum computer given the independent value model could simulate 34 qubits, the record for the total amount of qubits simulated on a classical computer is held by QSPACE, being able to simulate 42 qubits (on extremely advanced super computers). Keeping this in mind, if the phase matching problem can be solved, this makes the independent value model a very lucrative model as my desktop computer can simulate about a 256^{th} of a million dollar parallelized super computer. The advantage of concurrency and multiple cores for this problem is also worth mentioning.

However this is all assuming the phase-matching problem can be solved and that there aren't any other lingering problems with the Independent Value model that I simply haven't discovered. Thus further research would need to be put into the phase-matching problem in order to attempt to solve whether or not such a problem is a fatal one for the independent-value model or not.

Even if I was unsuccessful at developing a quantum computer simulator that was more efficient than the standard implementation, I did come up with several interesting ideas, which I hope could encourage more abstract thought about how we can think about quantum computers. The thing is people mainly think of quantum computers as a translation from mathematics directly onto computers, what I was attempting to do was to instead "cheat", and attempt to simulate the quantum computer using more efficient methods that, although don't run the same way as the mathematics, should be exactly equivalent. Basically, I'm hoping this could encourage people to think of quantum computer simulators in a different way, to try and create quantum computer models that do not follow the standard form of solving the Hamiltonian or applying linear algebra.

Overall I learned a lot about how quantum computers operate and overall found the experience to be extremely enlightening, though at times frustrating.

Appendix A: List of all Quantum Gates

Here is a quick description of every single important quantum gate that is required for a quantum computer to be universal.

Hadamard Gate

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

Turns pure states into bell states. Puts a qubit in a perfect superposition between its zero state and its 1 state. Could also be considered putting the qubit into and out of the Hadamard basis defined as

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}; |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

The Pauli-X gate

$$X = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

Generalized version of the classical NOT gate, will flip the 0 probability and the 1 probability of a qubit.

The Pauli-Y gate

$$Y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$$

Similar to a Pauli-X gate, however flips the phase of the qubit as well.

The Pauli-Z gate

$$Z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

Rotates the phase by of a qubit by a complex angle of Pi/2.

The Phase Gate

$$Phase = \begin{pmatrix}1 & 0\\ 0 & e^{i\phi}\end{pmatrix}$$

Generalized version of the Pauli-Z gate, in that it rotates the phase of the qubit, but does so with an arbitrary angle.

The C-NOT gate

$$CNOT = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$$

Entangler gate, applies the Pauli-X operation on a qubit given that a certain control bit is 1.

Appendix B: Implementation of Independent-Value Model Quantum Register

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

namespace QBlue
{
    public class QReg
    {
        private Qubit[] qubits; // All the qubits in the quantum register
        private int size; // The number of qubits in this quantum register
        private int controlGateNum = 0; // The number of control gates being applied
to this quantum register
        private List<QGateStore> gates = new List<QGateStore>(); // All the gates, in
order, being applied to this quantum register.
        private List<int[]> entanglement = new List<int[]>(); // The Entanglement
Matrix

        public QReg(int size = 1)
        {
            if (size <= 0) throw new ArgumentOutOfRangeException("Argument must be a
positive integer above 0");
            this.qubits = new Qubit[size];
            this.size = size;
            for (int i = 0; i < size; i++)
            {
                qubits[i] = new Qubit();
            }
        }

        /// <summary>
        /// Apply a single qubit quantum gate to all qubits of our quantum register.
        /// </summary>
        /// <param name="gate">Gate to apply</param>
        public void ApplyGate(QGate gate)
        {
            for (int i = 0; i < size; i++)
            {
                // We use the QGateStore to store information about our quantum gate
                QGateStore store = new QGateStore();
                store.gateToApply = gate; // The actual matrix that this quantum gate
represents
                store.isControlGate = false; // Whether or not the gate is a control
gate.
                store.controlBits = null; // If there are control bits in the gate,
what are they?
                store.targetBit = i; // What qubit are we applying this gate to.
                store.type = QGateToQType(gate); // What type of gate is this quantum
gate
                gates.Add(store); // Ad the gate to our list of gates.
            }

        }

        /// <summary>
        /// Apply a gate to a series of quantum gates.
        /// </summary>
        /// <param name="gate">Quantum gate to apply</param>
```

```
        /// <param name="qubitsToApply">All the qubits we want to apply our quantum
gate to</param>
        public void ApplyGate(QGate gate, params int[] qubitsToApply)
        {
            foreach (int qubit in qubitsToApply)
            {
                QGateStore store = new QGateStore();
                store.gateToApply = gate;
                store.isControlGate = false;
                store.controlBits = null;
                store.targetBit = qubit;
                store.type = QGateToQType(gate);
                gates.Add(store);
            }
        }

        /// <summary>
        /// Apply our quantum gate to a single qubit
        /// </summary>
        /// <param name="gate">Gate to apply to our quantum register</param>
        /// <param name="qubitToApply">qubit to apply to our quantum gate</param>
        public void ApplyGate(QGate gate, int qubitToApply)
        {
            QGateStore store = new QGateStore();
            store.gateToApply = gate;
            store.isControlGate = false;
            store.controlBits = null;
            store.targetBit = qubitToApply;
            store.type = QGateToQType(gate);
            gates.Add(store);
        }

        /// <summary>
        /// Apply a Control-NOT gate to our quantum register
        /// </summary>
        /// <param name="targetBit">Our target bit</param>
        /// <param name="controlBits">All our control bits</param>
        public void ApplyCNOTGate(int targetBit, params int[] controlBits)
        {
            QGateStore store = new QGateStore();
            store.gateToApply = Gates.X; // A CNOT gate is just a controlled-X gate so
we're applying X specifically
            store.isControlGate = true;
            store.targetBit = targetBit;
            store.controlBits = controlBits;
            store.type = QType.CNOT;
            gates.Add(store);
        }

        /// <summary>
        /// Measures the state of our quantum register
        /// </summary>
        public void MeasureState()
        {
            // Create the entanglement Matrix for our quantum register
            CreateEntanglementMatrix();
            // Guess a number between 0 and one
            double p = QBlueManager.random.NextDouble();
            int value = 0;
            // Wait until our probability becomes less than or equal to zero
            for (int i = 0; p > 0; i++)
            {
```

```
                // Get the probability of measuring this one value
                ValueStore result = MeasureOneValue(i);
                // deduct the probability squared (technically probability multiplied
by its complex conjugate).
                p -= result.probability.Real*result.probability.Real +
result.probability.Imaginary*result.probability.Imaginary;
                // As the value we are measuring may of changed we need to adjust what
value we're actually measuring
                value = result.value;
            }
            // Print the value
            System.Diagnostics.Debug.WriteLine("Value measured: " + value);
            // Print the quantum state
            System.Diagnostics.Debug.WriteLine("State measured: " +
ConvertValueToString(value));
        }

        /// <summary>
        /// Wrapper for the original "PrintQuantumState"
        /// </summary>
        public void PrintQuantumState()
        {
            // Create entanglementMatrix, we only need to do this once per
measurement.
            CreateEntanglementMatrix();
            for (int i = 0; i < Math.Pow(2, size); i++)
            {
                // Compute the probability of every possible value
                ValueStore result = MeasureOneValue(i);
                System.Diagnostics.Debug.Write(result.probability.ToString() +
ConvertValueToString(result.value) + " +");
            }
            System.Diagnostics.Debug.WriteLine("");
        }

        /// <summary>
        /// Calculates the probability of a single value
        /// </summary>
        /// <param name="value">The value that we want to find the probability
for</param>
        /// <returns>Value store, a probability + value pair, as the value may of
changed.</returns>
        private ValueStore MeasureOneValue(int value)
        {
            ValueStore result = new ValueStore();
            // Initiate our probability
            Complex probability = 1;
            // Because of Hadamard entanglement we need to know what control gate
we're up to for the entanglement matrix.
            controlGateNum = -1;
            // Clear the state of the quantum register
            foreach (Qubit qubit in qubits)
            {
                qubit.Reset();
            }

            // Apply each quantum gate one by one in order
            foreach (QGateStore gate in gates)
            {
                // If we are dealing with a control gate
                if (gate.isControlGate)
                {
```

```
                    // Increment the control gate counter
                    controlGateNum++;
                    // If all the control bits are one
                    if (IsAllOne(value, gate.controlBits))
                    {
                        // We swap the value corresponding to the target qubit from 0
to 1 or 1 to 0
                        value ^= 1 << gate.targetBit;
                        // We then apply the gate associated with the CNOT gate, which
is a pauli-X.
                        qubits[gate.targetBit].ApplyGate(gate.gateToApply);
                    }
                }
                // If the gate is a hadamard gate
                else if (gate.type == QType.Hadamard)
                {
                    if (controlGateNum >= 0)
                    {
                        // Use the entanglement matrix to get our target qubit

qubits[entanglement[controlGateNum][gate.targetBit]].ApplyGate(gate.gateToApply);
                    }
                    else
                    {
                        // We do not require our entanglement matrix.
                        qubits[gate.targetBit].ApplyGate(gate.gateToApply);
                    }
                }
                else {
                    // X and Y are in the format <<0|a>,<b|0>> thus they require the
value to flip for both these gates
                    if (gate.type == QType.X || gate.type == QType.Y)
                    {
                        value ^= 1 << gate.targetBit;
                    }
                    // If it is just a phase gate or Z gate then we don't need to flip
the value
                    qubits[gate.targetBit].ApplyGate(gate.gateToApply);
                }
            }
            result.value = value;
            // Multiply all our single qubits together to get the resulting
probability.
            for (int j = 0; j < size; j++)
            {
                probability *= qubits[j].Get((value & 1 << j) == 0);
            }
            result.probability = probability;
            return result;
        }

        /// <summary>
        /// Determines if a series of control bits is entirely one.
        /// </summary>
        /// <param name="value">The value we are checking</param>
        /// <param name="bits">The bits that must be one</param>
        /// <returns></returns>
        private bool IsAllOne(int value, params int[] bits)
        {
            foreach (int bit in bits)
            {
                if ((value & 1 << bit) == 0) return false;
```

```
            }
            return true;
        }

        /// <summary>
        /// Creates the entanglement Matrix that is used to calculate Hadamard
operations
        /// </summary>
        private void CreateEntanglementMatrix()
        {
            // Reset current Entanglement Matrix
            entanglement.Clear();
            // Count number of Control gate operations we are applying
            int cGateNum = -1;
            // Records a list of all our values that are in Bell states.
            bool[] hadamard = new bool[size];
            foreach (QGateStore gate in gates)
            {
                // If our gate is in a bell state we add it to the Hadamard array
                if (gate.type == QType.Hadamard)
                {
                    hadamard[gate.targetBit] = !hadamard[gate.targetBit];
                } else if (gate.type == QType.CNOT)
                {
                    // We are applying a CNOT operation so we need to create a new
entry to our entanglement Matrix
                    entanglement.Add(new int[size]);
                    cGateNum++;
                    // Set the values of our new entanglement row
                    if (cGateNum == 0)
                    {
                        // By default the row just maps the ith qubit to the ith
qubit.
                        for (int i = 0; i < size; i++)
                        {
                            entanglement[cGateNum][i] = i;
                        }
                    }
                    else
                    {
                        // If there is a previous row, copy that row
                        for (int i = 0; i < size; i++)
                        {
                            entanglement[cGateNum][i] = entanglement[cGateNum - 1][i];
                        }
                    }
                    // Only works for one control bit at the moment
                    bool applyEntanglement = hadamard[gate.controlBits[0]];
                    // If we are to apply an entanglement
                    if (applyEntanglement)
                    {
                        // These bits are already entangled, so we instead remove the
entanglement
                        if (cGateNum != 0 &&
entanglement[cGateNum][gate.controlBits[0]] == entanglement[cGateNum][gate.targetBit])
                        {
                            entanglement[cGateNum][gate.controlBits[0]] =
gate.controlBits[0];
                        } else
                        {
                            // Otherwise we map our controlbit to our targetbit,
entangling those bits
```

```
                            entanglement[cGateNum][gate.controlBits[0]] =
gate.targetBit;
                        }
                    } else if (hadamard[gate.targetBit] ||
(entanglement[cGateNum][gate.controlBits[0]] ==
entanglement[cGateNum][gate.targetBit]))
                    {
                        // We are applying a CNOT gate but out entangler bit is the
target bit, this removes any entanglement.
                        entanglement[cGateNum][gate.controlBits[0]] =
gate.controlBits[0];
                    }
                }
            }
        }

        /// <summary>
        /// Converts an integer value into a string of binary digits
        /// </summary>
        /// <param name="value">Value to convert</param>
        /// <returns>string of binary digits</returns>
        private string ConvertValueToString(int value)
        {
            string s = "|";
            for (int i = size - 1; i >= 0; i--)
            {
                if ((value & 1 << i) == 0)
                {
                    s += "0";
                } else
                {
                    s += "1";
                }
            }
            return s + ">";
        }

        /// <summary>
        /// Returns the type of the quantum gate we are applying
        /// </summary>
        /// <param name="gate">Gate to discover type of</param>
        /// <returns>The type of our quantum gate</returns>
        private QType QGateToQType(QGate gate)
        {
            if (gate.Equals(Gates.X)) return QType.X;
            else if (gate.Equals(Gates.Y)) return QType.Y;
            else if (gate.Equals(Gates.Z)) return QType.Z;
            else if (gate.Equals(Gates.Hadamand)) return QType.Hadamard;
            else if (gate.phaseGate) return QType.Phase;
            else return QType.Other;
        }
    }
}
```

Appendix C: Quantum Fourier Transform

I just decided to write a quick summary of the Quantum Fourier Transform as I feel Aleks will find this interesting. The Quantum Fourier Transform is an algorithm for computing the Discrete Fourier transform of 2^n different values in only n^2 quantum gates. Of course any arbitrary number of values can be computed, we'll assume that the amount of values we need to compute N is 2^n for simplicity. We once again just briefly cover that the Discrete Fourier transform, given a vector X of N values, transforms those values into a vector Y where each element of Y is given as

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega^{jk}$$

If we take the probability of a quantum state being a certain value, we can compute the discrete Fourier Transform of that quantum state using Quantum parallelization. We recall that the Fourier transform can also be applied in matrix form as

$$FX = \left(\begin{bmatrix} 1 & \cdots & \omega^{N-1} \\ \vdots & \ddots & \vdots \\ \omega^{N-1} & \cdots & \omega^{(N-1)^2} \end{bmatrix} \right) X$$

Where *X* is our value vector. Thus we only require to construct a series of quantum gates that create such a matrix. It turns out that such a series does exist and the algorithm to develop it is pretty simple. The algorithm follows the following pseudocode

```
QFT(Q, N) {
  IF (N == 0) {
    Hadamard(Q[0]);
    RETURN;
  } ELSE {
    QFT(Q, N – 1);
    FOR (int k = 0; k < N – 1; k++) {
      CONTROLLED-PHASE(control = N; target = k; Phase = π/2^(N−k−1));
    }
    Hadamard(Q[N]);
  }
}
```

In this pseudocode algorithm we've assumed that that *Q* is a global array of qubits. We can test to see if this algorithm works on a quantum system. Suppose we had four arbitrary values for our quantum state. Thus our quantum state is in the following system

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

We then apply the algorithm, the algorithm will result in the following gate sequence

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} H_0 CPHASE_{10} \frac{\pi}{2} H_1$$

Applying the gates in order gives

$$\frac{1}{\sqrt{2}}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\1 & -1 & 0 & 0\\0 & 0 & 1 & 1\\0 & 0 & 1 & -1\end{pmatrix}CPHASE_{10\frac{\pi}{2}}H_1$$

$$=\frac{1}{\sqrt{2}}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\1 & -1 & 0 & 0\\0 & 0 & 1 & 1\\0 & 0 & 1 & -1\end{pmatrix}\begin{pmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & e^{i\frac{\pi}{2}}\end{pmatrix}H_1=\frac{1}{\sqrt{2}}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\1 & -1 & 0 & 0\\0 & 0 & 1 & 1\\0 & 0 & 1 & -e^{i\frac{\pi}{2}}\end{pmatrix}H_1$$

$$=\frac{1}{2}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\\1 & -1 & 0 & 0\\0 & 0 & 1 & 1\\0 & 0 & 1 & -e^{i\frac{\pi}{2}}\end{pmatrix}\begin{pmatrix}1 & 0 & 1 & 0\\0 & 1 & 0 & 1\\1 & 0 & -1 & 0\\0 & 1 & 0 & -1\end{pmatrix}=\left(\frac{1}{2}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix}\begin{pmatrix}1 & 1 & 1 & 1\\1 & -1 & 1 & -1\\1 & i & -1 & -i\\1 & -i & -1 & i\end{pmatrix}\right)$$

Thus our final vector is in the form of

$$\frac{1}{2}\frac{1}{2}\begin{pmatrix}x_0+x_1+x_2+x_3\\x_0+ix_1-x_2-ix_3\\x_0+ix_1-x_2-ix_3\\x_0-ix_1-x_2+ix_3\end{pmatrix}$$

If we apply the Fourier transform on the same values, as specified above, noting that

$$\omega^{jk}=e^{\frac{2\pi ijk}{N}}$$

Then we have our array being (with *N* being 4)

$$\frac{1}{2}\begin{pmatrix}e^0x_0+e^0x_1+e^0x_2+e^0x_3\\e^0x_0+e^{\frac{i\pi}{2}}x_1+e^{i\pi}x_2+e^{\frac{i3\pi}{2}}x_3\\e^0x_0+e^{i\pi}x_1+e^{2i\pi}x_2+e^{3i\pi}x_3\\e^0x_0+e^{\frac{i3\pi}{2}}x_1+e^{i3\pi}x_2+e^{\frac{i9\pi}{2}}x_3\end{pmatrix}$$

Which just becomes

$$\frac{1}{2}\begin{pmatrix}x_0+x_1+x_2+x_3\\x_0+ix_1-x_2-ix_3\\x_0-x_1+x_2-x_3\\x_0-ix_1-x_2+ix_3\end{pmatrix}$$

Clearly the second and third row are swapped, but that may be because of a slightly improper implementation of the quantum Fourier transform, nevertheless one can simply swap these two rows without too much trouble. Thus we have successfully implemented the Quantum Fourier transform on 4 values using only 3 gates and 2 qubits.