Computational Physics Project: a Simulation of a Quantum Computer

Daniel Homer, Stephen Lloyd, James McKellar, Dmitry Tsigelnitskiy

March 2011

Abstract

In this project a quantum computer was simulated in Java[1], to utilize Grover's Algorithm[2]. The simulator was created using a matrix approach. As the efficiency of the quantum computer is a consequence of the fundamental properties of its operations, theoretical efficiency of a quantum computer cannot be achieved using a classical computer. For Grover's algorithm simulated on a classical computer, the classical search would have to be performed. So, in fact, to get the simulator to be as efficient as possible, a classical algorithm would have to be used. As a result, we concentrated on the theory and the use of object oriented design to visualize it. Grover's algorithm was simulated and then was tested for a number of steps the quantum computer would use to get the result.[1]

[1]When writing this report, a lot of emphasis was put on interactivity, so, a reader would benefit from reading an electronic version (PDF) rather than a paper one

Contents

Chapter 1

Introduction and Theory

1.1 Classical Computer

[1] Classical computers, like the ones we were working on, are completely deterministic due to their dependance on a binary system. This property of the computer is so fundamental because it is a result of the very physical components that they are composed of. The classical computer is physically based on transistors which amplify, change or stop the flow of electrons. These transistors are used to construct logic gates: small collections of transistors which give certain outputs for certain inputs. For example, a classical AND gate has two inputs and one output. It will only produce an output of value 1 if both inputs are also of value 1. Input of a 1 and a 0 will result in output of 0. A transistor-based classical computer releases tiny bursts of current in varying patterns, which is essentially what makes up what we would think of as data. This system means that at any given time, a bit of information in the system can either be in a state of 1 or 0 (ie, on or off, depending on whether the current logic gate is outputting current or not). The implication of this is that the computer which is made of such components will be deterministic. There will never be an element of chance involved; the process of calculation, and by extension output is either a definite"yes" or "no". These logic gates are arranged in a network that allows them to be used to solve algorithms. When varying patters of bits are fed into the network, the "answer" will emerge on the other side of the logic gates.

[1]The theory explained was done with help from a "Basic concepts in quantum computation" paper[3] and a book "Quantum Computing Explained"[4]

1.2 Brief History of Quantum Computing

A quantum computer is fundamentally different to this. The idea of using quantum behaviour for computation was first proposed in the 1970s. It was proposed by several theoretical physicists such as Richard Feynmann, Paul Benioff and David Deutsch[5]. However to begin with, the practicality of this was heavily disputed. While most conceded that theoretically a quantum computer would work, the impracticality of harnessing sub atomic systems seemed to greatly outweigh any proposed advantage that could come of it.

In 1994, Peter Shor produced an algorithm for a quantum computer that factorised large numbers tremendously quicker than a classical computer could[6]. This was the first indisputable piece of evidence that quantum computers were potentially advantageous to classical computers. Following this discovery, there was a tremendous increase in research into quantum computing by physicists and computer scientists. Suddenly there was a reason for further research. While many algorithms solved using quantum computation give a definite yes or no answer(for example, see Section 2.1), the quantum computer itself is not built using the same deterministic transistor-based technology that one would find in a classical computer. Quantum computers make use of several properties specific to quantum mechanics, such as entanglement and superposition. This fundamental difference between classical and quantum computers means that a quantum computer can solve certain algorithms exponentially faster than a classical one.

1.3 Tensor product

When working with quantum bits and quantum gates, it is very important to understand tensor products. Tensor products are extremely mathematically intensive to calculate from a computational point of view. Not because there is any particularly complicated or abstract mathematics involved, but rather there are huge numbers of small calculations to make. To make matters more difficult, as the size of the register or gate increases, the number of calculations increases exponentially.

If we want to take the tensor product of two matrices A and B

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{1.1}$$

then we do this by replacing every element in matrix A with the scalar multiple of that element and the whole of matrix B:

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}. \tag{1.2}$$

Registers would classically contain N bits and would be of size N. However in a quantum computer, a register of qubits is calculated by tensoring those qubits together. Similarly, when a gate is needed of a certain dimension, it is tensored with itself until it is of the same dimension as the register that it is working on. This was only used in the first version of the code. In later versions the methods using tensor products were removed from most areas due to their tremendous computational cost.

1.4 Qubits and Quantum registers

1.4.1 Qubits

The qubit is the quantum equivalent of a bit. While like a bit, it has two potential states that it can be in (1/0, on/off), however while a bit is definitely one or the other, a qubit can be 0, 1, or some superposition of the two. This definition leads us to represent a qubit as a linear combination of the two states with each state multiplied by its respective probability:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1.3}$$

Here, α and β are the respective probability amplitudes of the two potential states. They are both complex numbers, that is, they have both real and imaginary components. Physically, qubits are harder to define than their classical cousins. A classical bit is easily defined as either a small current, or a lack thereof. But since qubits are subject to various quantum properties like entanglement and superposition, they cannot be defined so simply.

A very good way to visualize a qubit in a superposition of states would be a "Bloch Sphere" (Figure 1.1)[2]. Vector $|\psi\rangle$ represents a possible state of the qubit, if the vector is in the top hemisphere, it is more likely to collapse to state $|0\rangle$, if it is in the bottom hemisphere, it is more likely to collapse to state $|1\rangle$ during a measurement. One potential qubit would be a single

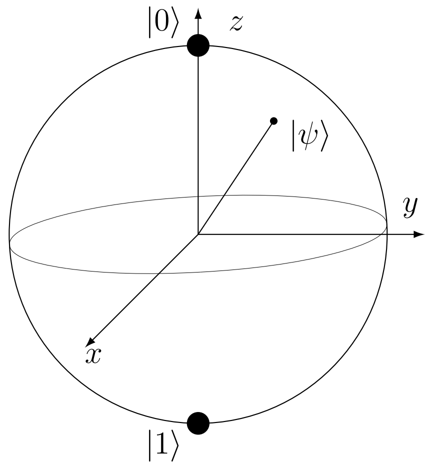


Figure 1.1: Bloch Sphere : vector ψ denotes a possible state of a qubit

electron. The two states are represented by the electron having either an up or down spin. The probability of finding the qubit in one of the states is equal to the square of the probability amplitude of that state. Since there is an absolute certainty that the qubit is in one of the two states, the sum of the squares of the two probability amplitudes is:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{1.4}$$

This condition becomes key to calculation later on. When this condition is true, we say that the qubit is "normalised". Quantum states behave in a way that is mathematically similar to physical vectors. For this reason, when discussing quantum computation, we say that all calculations take place in what we call a "vector space". Qubits can be represented in vector notation

[2]This figure was drawn using latex picture environment; the ellipse was drawn using bezier curves as described by Dr. Urs Oswald [7]

as

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{1.5}$$

then qubits in states $|0\rangle$ and $|1\rangle$ would look like

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1.6}$$

A superposition of all the states will be represented as

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{1.7}$$

Note that the probabilities α^2 and β^2 have to add up to 1, and should be equal for superposition of all the states

1.4.2 Quantum registers

To represent binary numbers in a quantum computer, a quantum register has to be created. So, to represent binary number 2 in a quantum computer, two qubits have to be tensored together:

$$|1\rangle \otimes |0\rangle = |10\rangle \tag{1.8}$$

A similarity can be seen between quantum and classical representation. To represent 2 as a vector, $|1\rangle$ and $|0\rangle$ vectors are tensored together:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{1.9}$$

The main advantage of a quantum computer is that a whole register can be a superposition of all its states:

$$\frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{1.10}$$

In general, a register cannot be represented by a tensor product, such register would be called "entangled". an example of such register would be

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}. \tag{1.11}$$

1.5 Quantum Gates

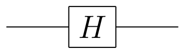
A quantum logic gate is some form of device that performs a specific mathematical operation on a qubit. To make the calculation of the output of a system simpler, we represent quantum logic gates as matrices. Recalling that the qubit is represented in vector form, this makes calculations fairly straightforward from a mathematical perspective: a gate acting on a register can be represented as simple matrix multiplication. In this section gates used in the project are described.

1.5.1 Hadamard Gate

The Hadamard gate is responsible for creating a superposition of all possible states out of a register in a certain state. Hadamard can also create a certain state out of superposition of states. As superposition principle is widely used in a quantum computer, one can imagine that every algorithm uses this gate extensively. The Hadamard gate is defined to be:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} \tag{1.12}$$

In the later sections, quantum networks are used to show schematically the interactions between the gates and qubits. In a quantum circuit, Hadamard Gate would be:



1.5.2 More than one qubit

Most of the times, one is working on a register that consists of more than one qubit. Naturally, the whole register would be passed through the gate. Turns out, representing that mathematically is trivial: a gate is tensored with itself to be of the right dimension. An example of a Hadamard gate for two qubit register:

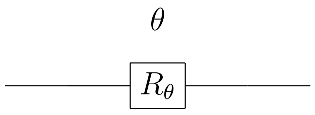
$$H \otimes H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{1.13}$$

1.5.3 Phase Gate

The phase gate is defined to be:

$$R_\theta = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \tag{1.14}$$

This gate gives a phase shift of θ to state $|1\rangle$. This gate is the key element of Grover's algorithm: Section 2.2.2.
Once the qubit has "passed through" a quantum gate it is in some way mathematically changed. In a quantum circuit the gate would be:



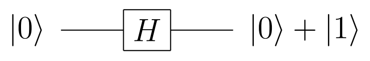
where θ represents the angle of phase shift. A basic phase gate can be extended to shift any qubit in a register, these are refered to as "controlled phase gates":

$$R_\theta = \begin{bmatrix} 1 & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & 0 \\ 0 & 0 & \dots & e^{i\theta} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \tag{1.15}$$

This gate shifts an arbitrary chosen qubit by a phase of θ.

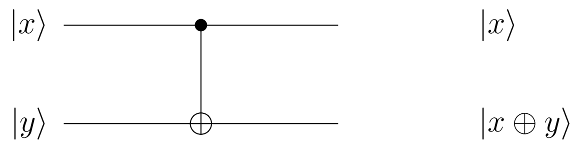
1.6 Quantum Circuits

A quantum circuit is a very useful tool to visualize a quantum algorithm or any kind of interaction between qubits and quantum gates. Suppose, there is a qubit of state $|0\rangle$ and a Hadamard gate acting on it: Such a circuit would be shown as:



Note that $|0\rangle + |1\rangle$ represent normalized superposition of states 1 and 0, i.e. in equation 1.4 α and β are $\frac{1}{\sqrt{2}}$, also, $|0\rangle - |1\rangle$ mean that α is equal to $\frac{1}{\sqrt{2}}$ and β is $-\frac{1}{\sqrt{2}}$ in equation 1.4.

Sometimes, gates act on more than one qubit, the so-called two-qubit gates are very common. An example of such a gate would be a controlled-not gate. In quantum circuit such a gate will look like:



Here, the qubit $|x\rangle$, "control qubit" is unchanged, but determines the nature of the algorithm (here does nothing if x=0 and inverts the second qubit if x=1). The symbol $\oplus$ stands for addition modulo 2. The full study of controlled-not gate is beyond the scope of this report, but the concept of two-qubit gate is useful in the following sections. Note that 3-qubit, 4-qubit, etc. gates are also possible but not that common.

Chapter 2

Algorithms

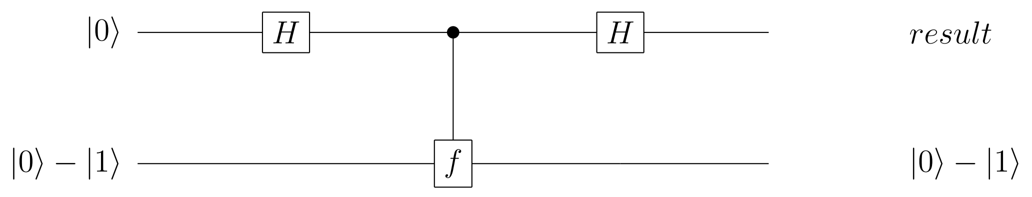
An algorithm can be thought of as a network of gates(classical or quantum) connected together to perform certain tasks. Naturally, a working model of a computer should be able to perform some algorithms.

2.1 Deutsch-Jozsa Algorithm

This is the first algorithm that was consiered, as it shows very clearly in what way a quantum computer can be superior to classical ones. The situation is as follows: There is a "black box" That can take 1 or 0 as input and produce 1 or 0 as output. A box is thought of as a function $f(x)$ where x can be 0 or 1. It is clear, that a box can be in 1 of the 4 states:

- $f(0) = 0$, $f(1) = 0$
- $f(0) = 0$, $f(1) = 1$
- $f(0) = 1$, $f(1) = 0$
- $f(0) = 1$, $f(1) = 1$

The 1st and the last state are refered to as being "constant", the others are refered to as "balanced". The task is to determine whether the "black box" is constant or balanced. Classically, one needs two queries to black box: to know the result for input one and zero. However, with a quantum computer it is possible to find the answer with only one query. Algorithm proposed by Deutsch did solve the question, however, the answer was not deterministic, in fact, the answer was 50% certain[8]. Nevertheless, the algorithm described in this section is deterministic, it is an improved version of the original Deutsch algorithm[9]. This algorithm can be represented as a quantum circuit:



Note that if function f takes (x, y), then the result can be represented as $(x, y \oplus f(x))$. This algorithm's input is $|0\rangle(|0\rangle - |1\rangle)$, after the first Hadamard gate becomes $(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$. Now, for $x \in \{$ 0,1 $\}$ under the effect of the "black box" [10]

$$|x\rangle(|0\rangle - |1\rangle) \longrightarrow |x\rangle(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) \tag{2.1}$$

leading to:

$$(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle) \tag{2.2}$$

expanding the left part gives:

$$((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle)(|0\rangle - |1\rangle) \tag{2.3}$$

using the states of the "black box" described earlier, becomes

- $(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$
- $(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$
- $(-|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$
- $(-|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$

respectively. acting with a Hadamard Gate on the first qubit gives

- $|0\rangle$
- $|1\rangle$
- $-|1\rangle$

- $-|0\rangle$

respectively. $|0\rangle$ is the same as $-|0\rangle$ when measuring, so the algorithm returns $|0\rangle$ for constant function and $|1\rangle$ for balanced function. The algorithm gives a deterministic result by accessing black box only once. Although this algorithm is not particulary useful in practice, it was implemented in the first version of the simulator as it helps understanding quantum computing better and Grover's algorithm in particular.

2.2 Grover's Algorithm

Grover's Algorithm was first put forward by Lov Grover [11] in 1996. It is a method whereby using a quantum computer, the search of an unordered list or database can be done for a fraction of the computational cost of that of a classical computer.

2.2.1 Classical search

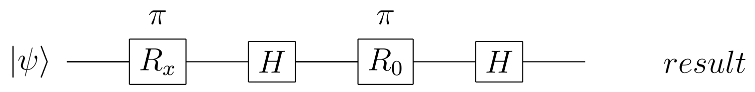
When faced with an unordered list of size N a classical computer must approach a search of this list in only one way. If for example we are looking for the letter "x" in a scrambled list of the alphabet, a Classical computer has no other option than to search each entry in the list systematically and check with a Boolean statement whether it is the letter "x". It may occur that it finds it in the first entry of the list; however it will be equally likely to find it in the last entry. This means that the classical computer cannot recover the search in any less than the order of N steps[12].

2.2.2 Quantum search

In quantum computing the process can be greatly enhanced. Due to the nature of a quantum register of qubits, all possible states can be represented simultaneously; a search can be undertaken on all states at once. If we again take the example of a scrambled alphabet, classically we would map this on to a register of bits and we could represent the first entry in the list as "000...00", the second entry as "100...00" etc. In order to search this list each bit must be switched to "1" or "0" to represent the binary representation of each entry.

Quantum computing registers represent each entry as a probability and to begin with there is equal probability of finding every entry. The Grover's Algorithm uses Phase and Hadamard gates to increase the probability of finding the correct answer. Because it can do this by applying changes to all qubits in the register simultaneously the answer is found in far fewer steps. Each step has the effect of reducing the probability of all the entries that do not correspond to the correct answer and increasing the probability of the correct one.

As stated the algorithm employs only two types of quantum gate, firstly the Phase gate and Secondly the Hadamard gate. The Phase gate firstly acts on the superposition register $|\psi\rangle$ to invert the phase of the entry which contains the correct answer by a phase of π. This has the effect of inverting the sign. The Hadamard is then applied followed by another Phase gate which phase shifts the zero state of the register. The final step is to pass the register through the Hadamard gate to produce a result. This final stage amplifies the probability of the correct answer.The effect this algorithm has is a rotation of the register into a more favourable position:



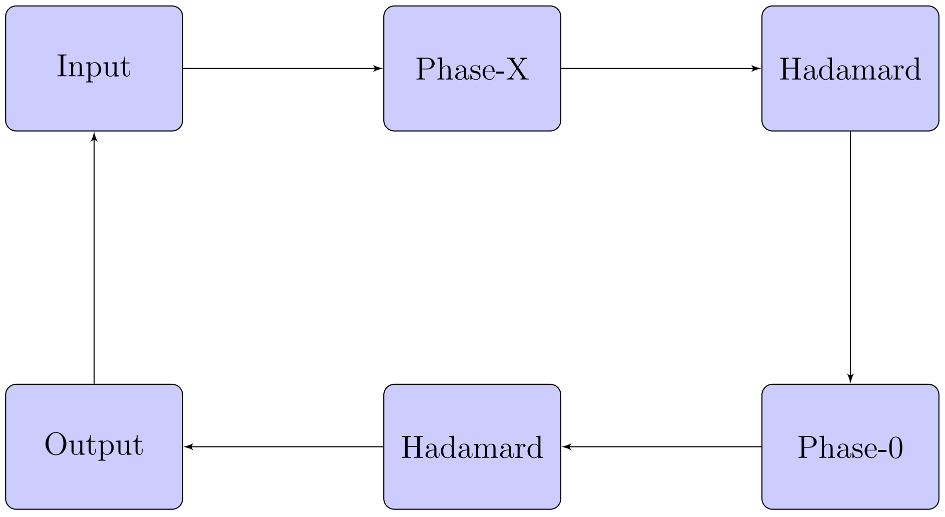
Our simulator shows a quantum qubit as vector, represented by a matrix in Java. A quantum register is a tensor product of these matrices. At the start of the algorithm a zero register is created; this can be done in one of two ways. Either a zero qubit [1,0] is created and tensored together with itself the desired number of times, or an empty matrix the appropriate length is created and filled with

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix}. \tag{2.4}$$

The second method is by far the quickest and most convenient way to create a quantum register in the zero state but obviously does not show the quantum theory of creating a register. Similarly to create a fully entangled register the zero register can either be passed through a Hadamard gate or an empty matrix filled with

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix} \tag{2.5}$$

and sufficiently normalised. The method that creates these from the basic elements was prefered as it shows how to create any size register and appropriate gate from the base elements, taking as an argument the number of qubits to be used. The problematic section of the Grover's Algorithm is creating the phase gate. In quantum computing the gate would act on the whole register at once and invert the phase of the desired result. In a classical simulator in order to do this we must first carry out the equivalent of a classical search to know which element of our register vector we want to invert. The Phase gate is created by first creating an Identity matrix and setting the diagonal element which will be responsible for the phase shift to "-1". When the register is acted upon by the phase gate the Identity matrix returns the same result as the input. Having one element set to "-1" instead of "1" changes the sign of that element in the register. This is a simplistic approach to the Phase gate which is usually represented as having diagonal elements $e^{i\phi}$ but when $\phi = 0$ the element reduces to "1" and when $\phi = \pi$ the element reduces to "-1" when dealing with real numbers. In this way the problem has been somewhat simplified by the simulator but this is to be expected. The simulator uses only three gates, The Hadamard, the Phase-x and the Phase-0. These act in succession on the register as seen below:



After each cycle the magnitudes of each element in the array which represent the probability of each entry in our unordered list have been altered. The magnitudes of all entries that do not correspond to the correct answer have been lessened and the entry that corresponds to the correct answer has been amplified. By repeating this cycle we end up with one entry in the register corresponding to the desired entry in the list having nearly 100% probability while all others fall off to practically zero. Figure 2.1 shows the start and

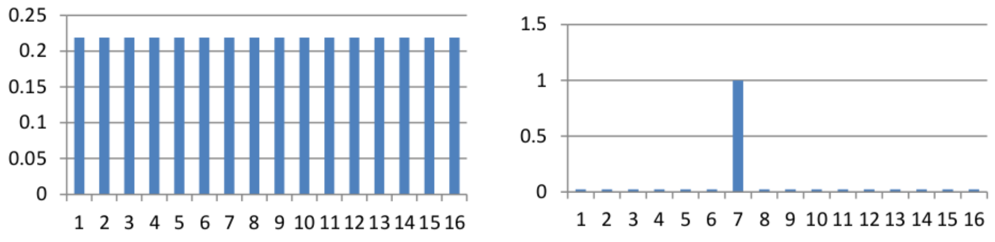


Figure 2.1: Initial Magnitude and Final Magnitude

end results of the algorithm. By mapping each entry in our unordered list to each element in the register then we have found the correct entry we are looking for using only the order of $\sqrt{N}$ calculations. Obviously within our simulation this is not entirely accurate as each time we start the algorithm we must perform a classical search on the list in order to initialise the Phase-x gate but if we consider each cycle of the algorithm as one operation of a quantum computer then we see that the number of operations compared to that of a classical search is greatly reduced.

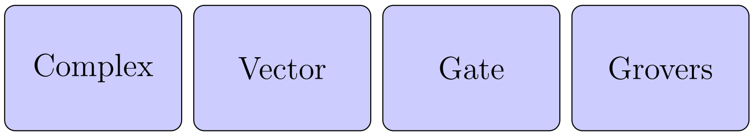
Chapter 3

Code Design

3.1 CRC Analysis

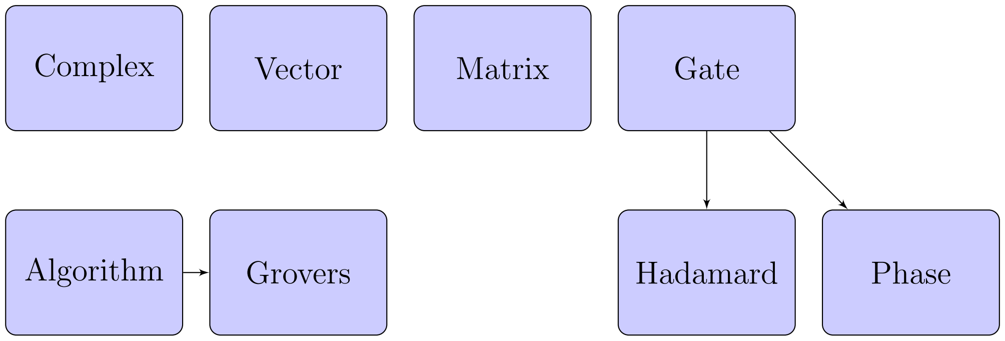
By writing our quantum simulator in java, it allowed us to take advantage of good object orientated practise to create something that is modular and logical in structure. This meant that we were able to come back to the project and improve it without changing large amounts of code at a time; resulting in a number of revised versions of the simulator, each being an improvement on the last.

3.2 Version 1



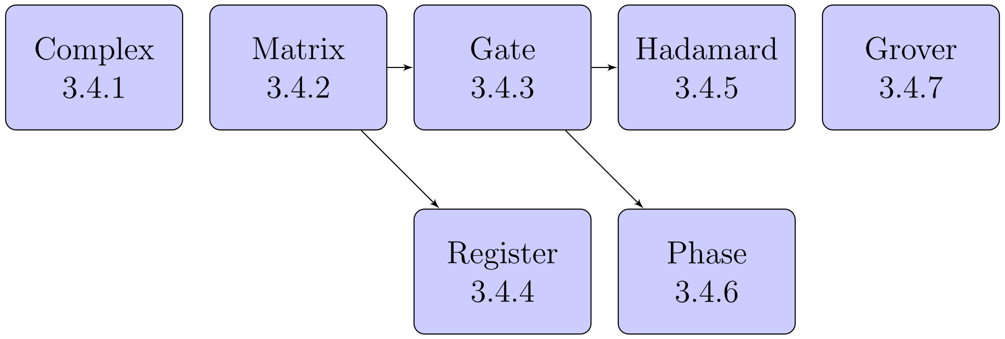
Initial versions of the simulator were focussed on first achieving results for our Grover's algorithm. This was a valuable learning exercise in not only understanding the significance of the algorithm itself, but also on how its implementation should shape the rest of the simulator. This algorithm-centred design focus motivated several changes to our initial class structure; such as abstracting different gates in to their own respective classes, each extending a base class called Gate. This gave the code a logical class structure that made algorithm implementation easier for both the coder and the reader. An algorithm-centred design structure also naturally motivated computational optimisation.

3.3 Version 2



Taking advantage of class abstraction resulted in other similar changes in subsequent simulator revisions. Noticing that both registers and gates required the same methods such as tensor products and matrix multiplications. We were able to write a base super class called Matrix which was extended by both Register and Gate. This superclass contained single methods that could be used by both. This abstraction resulted in less overall code, that was somewhat more analogous to the mathematical formulation of gates and registers as matrices.

3.4 Version 3



3.4.1 Class: Complex

Responsibilities:

- Defines a complex object in terms of real and imaginary class variables.
- Contains methods for complex addition, multiplication and 2-norm (magnitude) finding.

Collaborations:

- Used throughout the simulator, as all possible numerical values are assumed complex. The tradeoff here between the additional memory cost of storing real values as Complex objects (imaginary part being zero still having to be allocated as 0 in memory) and the ease of code reading and writing by doing so is discussed later in the report.
- Matrix
- Register
- Hadamard
- Phase

3.4.2 Class: Matrix

Responsibilities:

- Defines a mathematically general $m * n$ matrix.
- Contains general methods for matrix multiplication and tensor products as well as methods for normalisation and output formatting.

Collaborations:

- Used as the chosen formulation of how gates should act on registers and also how gates and registers are built up in the multi-qubit case.
- All subclasses of Matrix and subsequent sub-subclasses

3.4.3 Class: Gate

Responsibilities:

- Defines a gate as a matrix
- Contains no methods other than the constructor which defines it to be a type of matrix. Used logically to define all gates, such as Phase and Hadamard as matrices so that they all inherit the same matrix methods required.

Collaborations:
- Only responsible for its subclasses
- Only responsible for its subclassesHadamard
- Phase

3.4.4 Class: Register

Responsibilities:
- Defines a n-qubit register object
- Contains methods for checking if two registers are equal and how to normalise them.

Collaborations:
- Used by algorithm classes as the fundamental objects being acted upon
- Grover

3.4.5 Class: Hadamard

Responsibilities:
- Defines a n-qubit Hadamard gate object
- Contains constructor to create a n-qubit gate using only its functional representation rather than tensoring a single Hadamard gate n times. This will improve computational cost dramatically and will be discussed later.

Collaborations:
- Used in algorithms in forming a network of gates acting on registers
- Grover

3.4.6 Class: Phase

Responsibilities:
- Defines n-qubit Phase gate object in terms of a double class variable which determines the magnitude of the phase shift.
- Contains constructor to create a n-qubit gate again using only functional representation with the same benefits as for Hadamard

Collaborations:
- Again used by algorithms in forming a network of gates acting on registers
- Grover

3.4.7 Class: Grover

Responsibilities: – Contains a method for running the Grover's quantum search algorithm as described in the previous section

3.5 Code Optimisation and Efficiency

One obvious and important motivation behind our simulator design was efficiency. As well as writing code that took advantage of good object orientated practice, it is important to achieve results as efficiently as possible. With subsequent simulator revisions; several code optimisations were put in place as well as design changes which increased efficiency.

One particular example of a good code optimisation was the realisation that our code was explicitly calculating powers of 2. Our code regularly called upon the Math.pow() method to calculate 2^n, for example, when determining the dimensionality of registers and gates. This method uses logarithms to calculate powers of any base [13]. Integers are already stored as binary numbers in memory. It seemed more natural, and efficient, to simply use a bit-shift operation. By using the signed left bit-shift method, ¡¡, the binary representation of the number 2 (10 in binary) is shifted to the left by a certain amount. This has the effect of appending a 0 to the right hand side of the bit pattern.

$$10 \rightarrow << \rightarrow 100 \rightarrow << \rightarrow 1000 \tag{3.1}$$

Hence powers of 2 could be calculated by writing:

$$2 << (n-1) \rightarrow 2^n \tag{3.2}$$

An example of an overall design change was motivated by the large computational cost of calculating tensor products. Initial revisions of the simulator involved tensoring gates and registers with themselves N times, where N is the number of qubits used in the calculation. For large N, this required a lot of floating point operations. Tensor products are key in understanding how our objects interact, but we avoided using them as much as possible in later simulator revisions. Writing down N-qubit registers and N-qubit phase gates without using tensor products was trivial, but more complicated when considering an N-qubit Hadamard gate. To avoid tensoring a single qubit Hadmard gate together N times, we decided to seek a functional formulation of the Hadamard transform to construct the matrix. It was found that the rows of a N-qubit Hadmard gate are defined by the Walsh functions[14]. This gives us a method for generating elements of the matrix:

$$(H_n)_{i,j} = \frac{1}{2^{\frac{n}{2}}}(-1)^{i \cdot j} \tag{3.3}$$

where $i \cdot j$ is the bitwise dot product of the binary representations of i and j. for example, for i=4, j=5:

$$4 \cdot 5 = 100 \cdot 101 = (1 \times 1) + (0 \times 0) + (0 \times 1) = 1 \tag{3.4}$$

Although in practice this required a lot of casting, calculating and then re-casting in java; it was found to be a lot more efficient than calculating tensor products for large values of N.

If given more time on this task, further optimisation measures could be taken. One example of this would be to avoid matrix representation as much as possible or to at least use a sparse representation when storing them in memory. For large numbers of qubits, roughly 12 or more, it was found that the memory allocation of the large resultant gates was exceeding the Java heap size. Rather than simply increasing the heap space allocated to the JVM, measures could be taken to avoid the problem of large memory cost all together. Sparse representation would avoid storing repeated elements of the same value in a matrix which would help reduce memory cost. A better way would have been to use functional representation rather than a matrix one. For example, a Hadamard matrix can be treated as a functional Hadamard transform. This way it does not require the creation of additional objects to be stored in memory. A matrix representation seemed like a natural approach and this made it easier to see how everything is built up using basic objects.

Chapter 4

Results and Analysis

4.1 Results

The results from 4 Qubits shown in Figure 4.1 show that it took the simulator 8 cycles to return a result with a probability of greater the 99%. This is as many steps as for 7 Qubits(Figure 4.2). This can be attributed to the tolerance of the program. If the simulator is told to stop once the probability reaches 98% the simulator would have terminated after 4 cycles. This would give the expected result of $\sqrt{N}$ cycles for 16 entries in the list which is 4 cycles.

In subsequent trials for 5 Qubits the simulator terminated after 4 cycles(Figure 4.3). For 32 entries available the answer should be found in the order of 5-6 steps and this is in a good agreement with the simulators actions.

In all trials for 6 Qubits and above the simulator returns the result in less steps than the expected value. Although the result was found quicker than expected it was still within the order of magnitude expected.

For 8 Qubits (not graphed due to size of data), the result is expected to be found in the order of 16 steps. The simulator found a 99% probability in only 11 steps but we can see this is very close to the expected number of steps. Taking into account the results obtained from 4 Qubits it is unclear whether making the tolerance of the simulator more sensitive would return results in the correct number of steps. In the case of 4 Qubits when the result was not found after 4 steps due to the tolerance of the simulator, the magnitude of the correct answer actually decreased before once again increasing to a magnitude of over 99%. Bearing this in mind if we make the simulator more sensitive and program the result to be returned once the magnitude has reached 99.5% or higher the cycle may never reach this and instead would decrease again.

4.2 Analysis

Analysis of the results in comparison to the expected results is plotted in figure 4.5. This shows the aberration arising for 3 and 4 qubits which requires the simulator to perform 100% more cycles of the algorithm than is expected. This can avoided by lowering the tolerance of the algorithm to terminate once the magnitude of an entry is greater than 98%. In practice this fixes the aberration, however it forces a larger deviation from the expected result in higher number of qubits. With a tolerance of 99% all test simulations for higher numbers of qubits terminate before the expected $\sqrt{N}$ cycles. All are within an average 30% deviation from the expected result. By increasing the sensitivity to 99.99% the simulation still terminates before the expected result however, has a smaller deviation. In this case, for lower number of qubits the simulator is never able to reach the required magnitude or runs many more times than is expected.

4.3 Errors

By exploring the theory behind quantum computation; we see that its calculations are inherently probabilistic. This is reaffirmed by the results of the Grover's quantum search, where the process is not entirely deterministic. We find that initially the algorithm converges to the correct answer in the unsorted list with increasing probability of success at each iteration. However, we also found that if the program is not manually terminated at a given probability tolerance. The algorithm will then diverge away from the correct answer before converging again. This process repeats and is best represented by looking at how the wave function defined by the unsorted list changes at each iteration.

This is best done by treating the wave function as a vector in a vector space spanned by a basis as defined by the qubits used in the computation. This is analogous to the 1 qubit case of the Bloch sphere (Figure 1.1) where the wave function is free to rotate, this time in n-dimensional space, where n is the number of qubits. The particular direction of the vector determines its state. In this case, each element of the list would be defined by a particular point, defined by a phase, in the n-dimensional space.

Each iteration of the algorithm involves changing the state of the wave function. i.e. the wave function is rotated by a certain phase at each step which could be represented on a phase diagram. The vector will converge to the correct answer with sufficient probability after $\sqrt{N}$ steps. It will then rotate out of phase with the correct answer unless the algorithm is terminated at this point.

There is no definite correct answer when considering when to terminate the program. Ideally, the program should terminate before the algorithm diverges away from the correct answer. However, in a true quantum computer, there would be no way to observe the intermediate steps of the algorithm process. This is due to the inherent nature of the laws governing the quanta used to physically represent our qubits. That requires knowing the answer. The only way to determine when to terminate the program will be when the observer is satisfied with the probability of answer returned being correct.

The choice of tolerance is completely dependant on the use of the quantum search. If a more accurate answer is required; then a higher probability threshold can be used. If the problem requires faster computation, for example if the list is very large, and the accuracy of the correct answer is not as important; then a lower probability threshold can be used. There is clearly a tradeoff between computational cost and the resultant accuracy. The error here is inherent to quantum mechanics and has nothing to do with the computer itself. Any errors introduced by the possible physical limitations of qubit representation cannot be meaningfully explored in this report; with only access to classical computers. The errors introduced by classical floating point computation in our simulation are negligible and can be ignored.

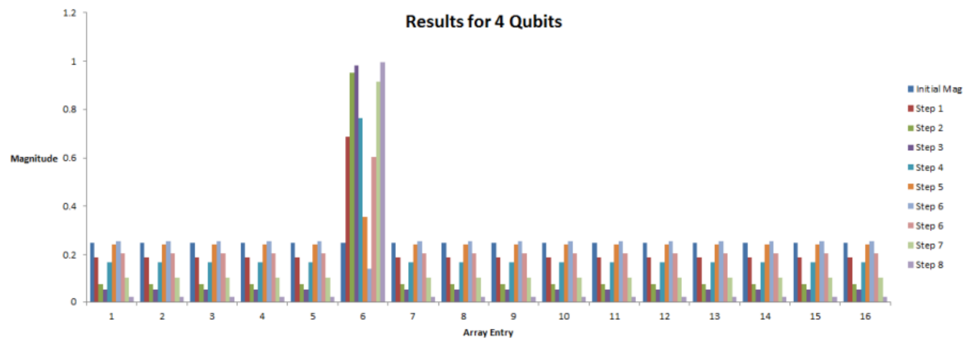


Figure 4.1: 4 Qubits Simulation

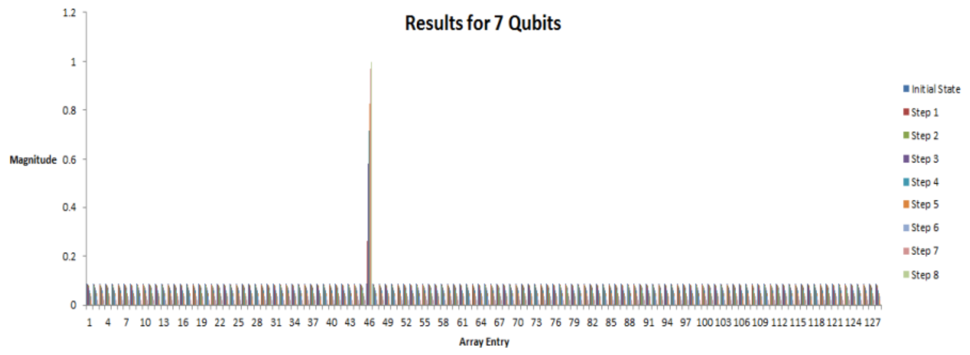


Figure 4.2: 7 Qubits Simulation

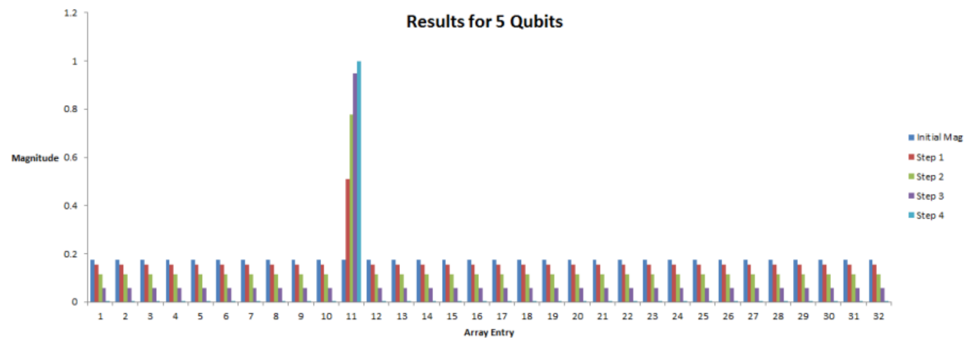


Figure 4.3: 5 Qubits Simulation

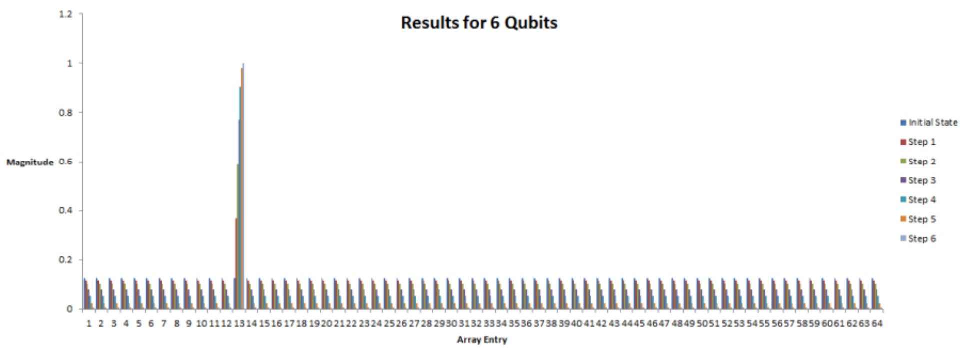


Figure 4.4: 6 Qubits Simulation

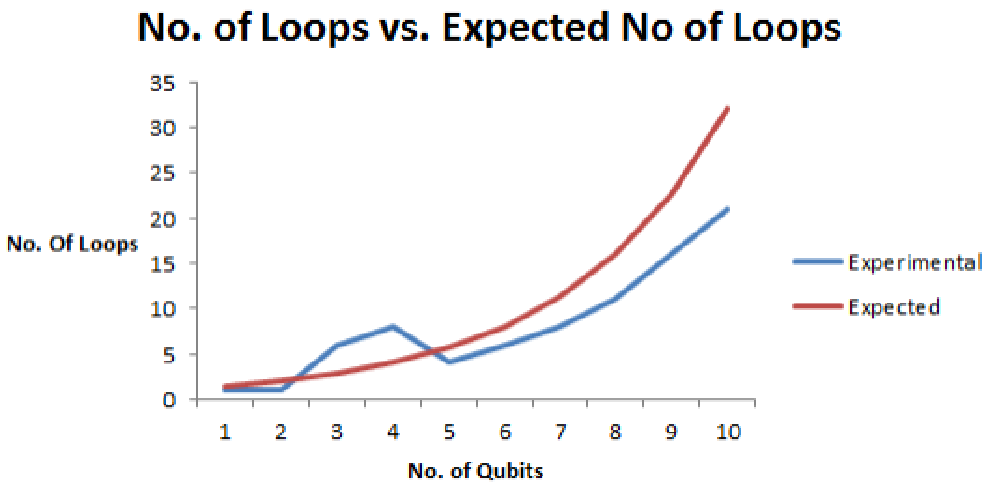


Figure 4.5: Analysis at 99 % tolerance

Chapter 5

Conclusions

We can see from the results obtained that a quantum computer would be massively faster in finding a result when faced with an exponentially large number of entries, compared to a classical computer. Given a list of 256 entries a classical computer would take 256 computations to find the correct result. The expected results for a quantum computer would be the square root of the number of entries, 16. Our results from the simulator are in agreement with this taking into account an error term of 5 cycles. From this we can see that the comparison between classical and quantum grows with each qubit introduced. For each qubit introduced the number of possible entries in the list doubles. Classically the number of computations required will double but for the quantum case the number grows far less.

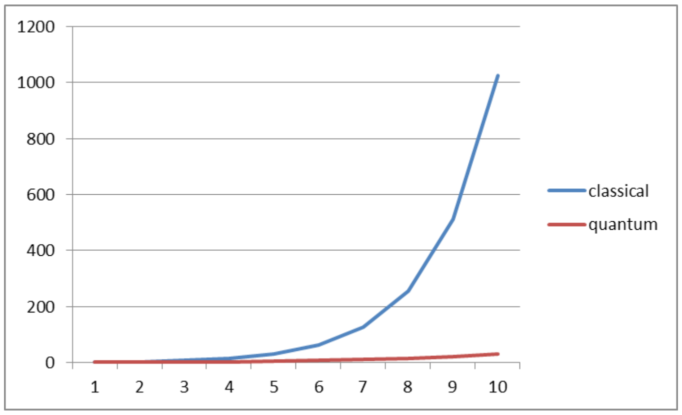


Figure 5.1: The expected results for up to 10 qubits or 1024 entries.

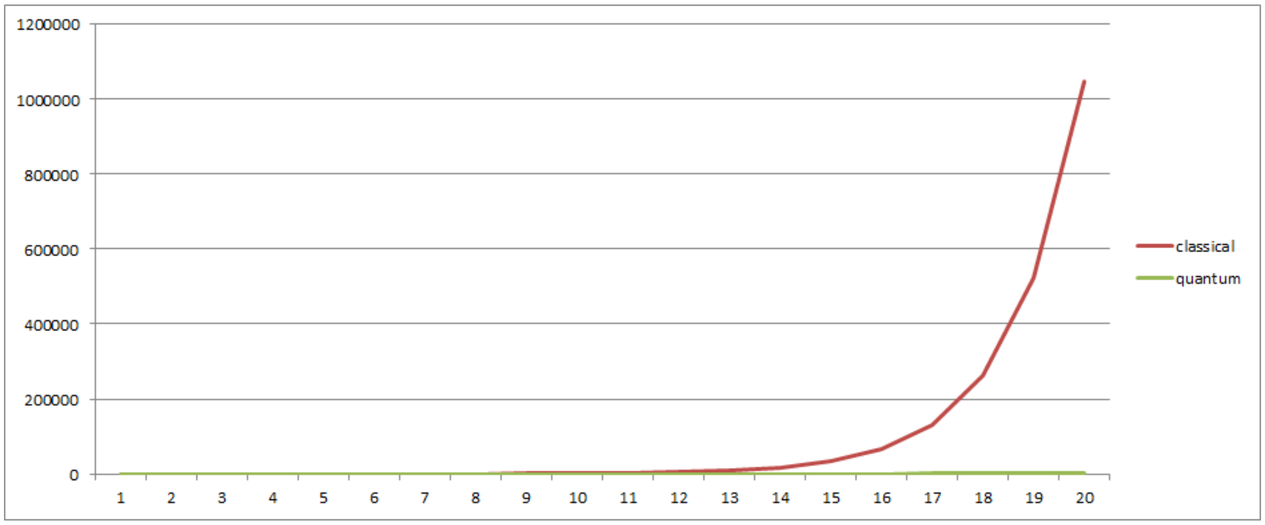


Figure 5.2: The comparison for up to 20 qubits or 1048576 entries.

These results show that the ratio of quantum to classical is $\sqrt{N}$. This means that for a register containing n qubits the number of entries $N = 2^n$. The classical computer will return the result in 2^n computations compared to the quantum computer which would only take $2^{\frac{n}{2}}$ computations. This will be of the order $2^{\frac{n}{2}}$ times less computations.

Figure 5.1 clearly shows that the quantum search is nearly 30 times faster than the classical search for 10 qubits. Figure 5.2 shows the massive difference between classical and quantum for 20 qubits. The quantum expectation is in the order of 1000 times faster than the classical expectation.

Chapter 6

Group Evaluation

For all of us, this was our first experience of programming as part of a group. While this in itself was potentially problematic, we were also left to organise the project by ourselves. We were essentially thrown in at the deep end, and this forced us to take organisation very seriously. Our first task was to fully understand the area of quantum computing. We had at this point previously studied quantum mechanics formally as part of our degrees, but we had no formal education in the area of quantum computing. For this reason we decided that before we attempted to code anything, we would spend about 3 weeks just reading up on the topic of quantum computing and the specific algorithms to ensure that we properly understood it. Once we had done this, it is fair to say that we each had a rough idea of how we were going to go about solving this problem. Luckily, none of our ideas were particularly radically different, so from the start we more or less agreed on how we were going to go about making the program. At this point, we organised a meeting where the four of us sat down and firmly decided which parts from whose ideas we were going to use and how it was all going to come together. We eventually settled on a system where entities such as complex numbers, matrices and gates were all handled by separate classes that are called upon as and when they are needed.

Once this was decided on, we delegated classes out for each of us to write. This was probably the easiest part of the actual coding, since all we had done was to create various classes that described various components in a quantum network. The hardest part would be combing all the classes into an actual algorithm solving system. We expected this to take many weeks.

Much to everyone elses surprise, after he had worked out how it worked on paper, Danny managed to make a working model of Grover's algorithm by himself in a matter of days. Once this was done, and our program for all intents and purposes worked, Stephen thought it would be best if we tidied up the code. From the beginning we really just concentrated on getting the simulator working with little attention being paid to good programming practice. So we all got together and using a synched folder that contained all the files and classes we set about trying to partially rewrite the code to make it more organised and easier to read. This proved very difficult, as we did not properly organise how to do it. All four of us were changing bits of code at once with very little communication about what we were doing. This led to the code "breaking", and we were presented with many compiling errors that we just could not seem to solve. The phrase "too many cooks spoil the broth" was practically written for our situation. After we realised that this method was not working, it was decided that Stephen and Danny would focus on fixing the code, and James and Dmitri would begin the report. Sure enough, the code was eventually fixed and result taking could begin.

Despite a few problems along the way, both technical and political, overall we managed to work quite well as a group. If we were to do a similar project again, though, there are a few things we would do differently. First of all, before writing a line of code we would all agree on set syntax, so that when we each went off and wrote separate sections of the program, there would be far fewer confusing compilation errors since we were all using the same terms. Another area we could have improved upon was communication; at times we needed to make it clear to each other who was doing what. Given that this was our first experience working as a group, and that we were left to organise everything ourselves, its fair to say that we were successful in tackling the project and dealing with any issues that arose along the way.

Bibliography

[1] Java source code for the simulator https://docs.google.com/leaf?id= 0B-VyIbDI63PIOTdmZTk4NWItMDMzMC00NDBmLTg1OTQtN WI1NmJhZmI5ZTIy&hl=en

[2] ”From Schrdinger’s equation to quantum search algorithm”, Grover L.K., Am. J. Phys., 69(7): 769-777 (2001)

[3] ”Basic concepts in quantum computation”, Artur Ekert, Patrick Hayden, Hitoshi Inamori, Les Houches, Volume 72/2001, 661-701 (2001)

[4] ”Quantum Computing Explained”, David McMahon ISBN: 0470096993

[5] ”Simulating physics with computers”, R.Feynman, International Journal of Theoretical Physics, Vol 21, Nos. 6/7, pages 467-488 (1982)

[6] ”Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”, Shor Peter W., SIAM J. Comput. 26 (5): 14841509 (1997)

[7] A paper on graphics in $\LaTeX\,2_\varepsilon$ by Dr. Urs Oswald http://www.ursoswald.ch/LaTeXGraphics/overview/latexgraphics.pdf accessed on 08/03/2011

[8] ”Rapid solutions of problems by quantum computation”, David Deutsch and Richard Jozsa, Proceedings of the Royal Society of London A 439: 553. (1992)

[9] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca (1998). "Quantum algorithms revisited" (PDF). Proceedings of the Royal Society of London A 454: 339354.

[10] "Quantum algorithms revisited", R. Cleve, A. Ekert, C. Macchiavello, M. Mosca, Proc. R. Soc. Lond. A 454, 339-354 (1998)

[11] A page about Lov. K. Grover http://www.bell-labs.com/user/lkgrover/ accessed 05/03/2011

[12] "The Art of Computer Programming" Volume 3: Sorting and Searching, Donald Knuth, ISBN 0-201-89685-0.

[13] IBM Developers page http://www.ibm.com/developerworks/java/library/j-math2.html accessed on 22/03/2011

[14] "Hadamard Transform Image Coding", William K. Pratt, Julius Kane, Harry C. Andrews, Proceedings of the IEEE 57 (1), 58-66 (1969)