School of Physics

Computational Physics Project
Quantum Computer Simulator

Alexander Bunting
Robin Richardson
Anshul Sirur
Andrea Thamm

17 March 2008

Supervisor: Professor A D Kennedy

10 Weeks

Abstract

A quantum computer simulator is presented. The simulator is based on the circuit model of quantum computation [1] in which quantum gates act on quantum registers which comprise a number of quantum bits (qubits). The initialisation is done by specifying the initial states of the qubits that form the quantum register. The output of the simulator is a dynamic graphical display of the quantum register state which shows the probability of measuring each one of the possible quantum register base states. Grover's algorithm for searching an unsorted database is presented using this simulator. It works up to a maximum number of 20 qubits. Additionally, a full quantum adder is implemented which adds numbers up to 60. The quantum computer simulator is easily extensible and offers an excellent opportunity to study quantum computing.

Contents

1 Introduction **2**

2 Theory **4**
2.1 Basic Quantum Computational Concepts 4
2.1.1 Quantum Bits 4
2.1.2 Quantum Registers 4
2.1.3 Quantum Gates 5
2.1.4 Quantum Algorithms 9
2.2 Basic Linear Algebra 14
2.2.1 Linear Spaces 14
2.2.2 Linear Operators 15

3 Implementation **16**
3.1 Code Structure 16
3.2 Linear Algebra 17
3.2.1 Linear Spaces 17
3.2.2 Linear Operators 20
3.3 QuBit, QuRegister, QuGate 22
3.4 QuAlgorithm 24
3.4.1 Grover's Algorithm 24
3.4.2 Quantum Adder 25
3.5 Frontend 27
3.5.1 Organisation 27
3.5.2 Scripting 28

4 Discussion **30**
4.1 Limitations of Grover's Algorithm 30
4.2 Limitations of Quantum Adder 30
4.3 Function versus Matrix representation 31
4.4 Group Management 32
4.5 Prospects 32

5 Conclusion **34**

Chapter 1

Introduction

Classical computing is the dominant computational technology in today's society. The transistor, invented in 1925 and heralded by some as the greatest invention of the 20th century, is the foundation of all computers we know to this day. However, today's silicon based processors are the end of this legacy. The key to making these processors faster and still conform to desktop specifications is to reduce the size of the transistors, however, we have reached a stage where to make them much smaller would cause quantum effects to no longer be negligible which would be detrimental to performance. This means we need a radical new solution to keep computer speeds increasing.

Quantum Computers

In the 1980s, scientists started to apply the ideas of Quantum Mechanics to Computer Science, and in 1982 Richard Feynmann had created an abstract model as to how a quantum system could be used to do computations [2]. However, this was all theoretical for many years as performing operations on a quantum computer is very different to performing them on a classical computer. The benefits of the quantum computer were not realised until 1994 when Peter Shor invented an algorithm for factoring exceptionally large polynomials, a technique useful in the field of cryptography [3]. Then, in 1996, Lov Grover invented an algorithm for searching databases (see section 2.1.4).

The major difference of a quantum computer compared to a conventional computer is its fundamental storage unit: the qubit. Unlike the bit in classical computation, the qubit has a physical implementation with quantum mechanical properties, i.e. it does not exist in definite states but in superposition of these with certain probabilities to collapse into one state. There are various possible physical implementation of a qubit, for instance the polarization direction of a single photon with its two directions horizontal and vertical or the spin of an electron [4].

Although a quantum computer would provide innovative new technology, the reality was that construction of such a computer would be difficult. The big issue with building a quantum computer is decoherence. This means when a quantum entity (such as an atom) interacts with something it will decohere from its quantum superposition,

essentially becoming useless. One solution to this, created by Oxford University [5], is to encase the quantum entity within a buckyball [6] and bombard it from all directions with microwaves. Microwaves do not cause the entity to decohere and also keep it within a certain area of the buckyball.

There is currently a great deal of research money being spent on this problem, and some very promising techniques are emerging that try and predict the effects of decoherence thus providing some means of error checking.

Aims

Quantum computation is studied in various ways. Active research is being done in building a quantum mechanical device. Properties of quantum computers can be studied using a simulator, i.e. a quantum computer being emulated on a classical machine. A simulation is, of course, not capable of actually revealing the advantages and increases in computation-speed a quantum computer offers. It is, however, an ideal way of studying and visualising its properties.

Implementing a quantum computer simulator was the topic of our Computational Physics Project. The aims were

- to learn about and understand the basic concepts of quantum computing.
- to design an abstract and open ended code structure in Java, which may be extended greatly.
- to create a quantum computer simulator.
- to implement Grover's algorithm for searching a database.
- to study some further aspects of quantum computing and code optimisation, if time allows it.

Project Approach

To complete a project of this magnitude it is important to realise where to start and what needs to be achieved along the way. This is a basic outline of what was to be done:

- Learn about quantum computational concepts.
- Create a structure and set some further aims.
- Code a first version and try some different approaches.
- Main program coding.
- Testing and additions.
- Report and presentation.

Chapter 2

Theory

2.1 Basic Quantum Computational Concepts

2.1.1 Quantum Bits

In classical computation a bit can either take on the value 0 or 1. Not so in quantum computing where a qubit can either be in the state $|0\rangle$, $|1\rangle$ or a superposition of both states [7]. The wave function $|\psi\rangle$ of the qubit can be described as:

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

Where α and β are the complex probability amplitudes of the state which have to satisfy the normalisation condition.

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

When taking a measurement the wavefunction collapses into one of the base states according to its probability amplitudes.

A convenient way of representing qubits and their superposition is in vector form [1]. The following notation is used to represent qubits throughout this section.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \text{and} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.3}$$

2.1.2 Quantum Registers

A quantum register is the quantum mechanical analogue of a classical register [8]. Quantum registers are measured on the number of qubits they can hold. The qubits entangle to form a quregister which is why the register can hold a large number of base states simultaneously. Mathematically, this is represented as a tensor product.

$$|\psi\rangle = |\psi\rangle_1 \otimes |\psi\rangle_2 \otimes \ldots \otimes |\psi\rangle_n \tag{2.4}$$

The number of base states a n-qubit register can store simultaneously is 2^n, which means for example that a 250 qubit register can potentially store more numbers simultaneously than there are atoms in the universe [9].

For example, a quantum register representing 3 qubits can hold any integer value from 0 to 7.

$$|0\rangle \otimes |0\rangle \otimes |1\rangle = |001\rangle = |1\rangle \tag{2.5}$$

$$|1\rangle \otimes |0\rangle \otimes |1\rangle = |101\rangle = |5\rangle \tag{2.6}$$

But it can also hold both of them simultaneously by storing a superposition of the two.

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle \otimes |1\rangle = \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle) \tag{2.7}$$

All three qubits can be in superposition which can be written as (ignoring normalisation constants):

$$|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle \tag{2.8}$$

This superposition stores all integer numbers between 0 and 7 simultaneously. Mathematically, quantum registers are represented as one dimensional vectors being created by the tensor product (to be exact, the Kronecker product) of the vector reprensation of the qubits. An example of this is the following.

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{2.9}$$

This representation nicly visualises the state of a quantum register.

2.1.3 Quantum Gates

Quantum Circuit Primer

Quantum circuits are a graphical and conceptual representation of a mathematical process or algorithm that we apply to the system, analogous to conventional electronic circuits. Quantum logic gates are devices that perform a unitary operation on a certain number of qubits in a fixed amount of time. The quantum logic circuit consists of a number of these gates acting on qubits within their quantum registers with their operations being synchronised in time. [1]

Quantum gates are represented mathematically as unitary matrices as they are basically linear operators. Linear algebra allows us to apply the same operations to single-valued qubits as to an entangled system of qubits. This means that unlike classical gates, quantum gates are reversible, where the output bits can be used to reconstruct

the constituent input bits. Analogues to reversible classical gates such as the Toffoli gate, can be used within the quantum logic framework.

Commonly, gates only act on one or two qubits at a time. Gates that act on single qubits, such as the Hadamard operator or the set of phase-shifting gates are represented by a 2×2 matrix while gates that act on two qubits such as the Controlled-NOT (and other controlled gates) are 4×4. Then the operation is applied to the qubit as a matrix is applied to a vector.

A list of the most commonly-used gates follows with a short description of their usage in quantum logic circuits and the mathematics behind them.

Hadamard Gate

The Hadamard gate is the quantum circuit equivalent of the Hadamard transform, a type of Fourier transform. It is commonly seen as being a quicker alternative to the Fast Fourier Transform which maps a time-dependent function into its frequency-dependent counterpart. [10]

In the quantum circuit, Hadamard gates act on single qubits and are used to place the qubit in a superposition of its two states. Reapplying the gate allows the qubit to collapse to its previous state. Mathematically:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.10}$$

For a qubit in the following state:

$$|a\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \tag{2.11}$$

$$H|a\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} a_0 + a_1 \\ a_0 - a_1 \end{pmatrix} \tag{2.12}$$

Since the gate is being applied to a qubit, the states of that register are placed in a superposition as well.

The Hadamard transform can then be applied to the register itself. The matrix that acts on the register is the end result of a chain of outer products with the Hadamard and Identity matrices, depending on which qubit we wish to apply the transformation to and the total number of qubits in the register.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.13}$$

For instance, if we wish to apply the gate to the 2nd qubit in a register of 3 qubits (8 base states)

$$X = I \otimes H \otimes I = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix} \quad \text{and} \quad |\psi\rangle = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \tag{2.14}$$

$$X|\psi\rangle = \begin{pmatrix} a_0 + a_2 \\ a_1 + a_3 \\ a_0 - a_2 \\ a_1 - a_3 \\ a_4 + a_6 \\ a_5 + a_7 \\ a_4 - a_6 \\ a_5 - a_7 \end{pmatrix} \tag{2.15}$$

Likewise for a Hadamard operation on the 3rd qubit, we would construct

$$X = I \otimes I \otimes H \tag{2.16}$$

This will be covered in greater detail in the operator implementation section (3.2.2).

Phase Shift Gate

$$\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} \tag{2.17}$$

The phase shift gate acts on a single qubit, rotating the $|1\rangle$ component of the qubit by an angle ϕ. This gate is used in conjunction with other gates such as the Hadamard to place the register in various entangled states, where the individual states are not seperable and any operations performed on an entangled state will affect the other states it is in a superposition with.

Controlled-f Gate

This is a specialised gate that reverses the sign of one state coefficient in the entire register. This could be considered as

$$a_i = (-1)^{\delta_{ik}} a_i \text{ where } k \text{ is the state we wish to flip.} \tag{2.18}$$

The gate can be represented by an Identity matrix with one negative element on the diagonal, where for a register of n qubits we have a matrix

$$\{f_k\}_{ij} = \delta_{ij}(-1)^{\delta_{jk}} \qquad i \in \{0, n\} \qquad j \in \{0, n\} \tag{2.19}$$

Controlled-NOT Gate

The Controlled NOT (CNOT) gate is a two-bit gate, acting on a system consisting of a control qubit and a target qubit, which inverts the target qubit depending on the value of the control qubit. In matrix representation

$$U_{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.20}$$

i.e. the result of the gate on the control qubit $|x\rangle$ and the target qubit $|y\rangle$

$$U_{CNOT}[x \otimes y] = |x\rangle \otimes |x \oplus y\rangle$$

where $|x \oplus y\rangle = (x + y) mod 2$

The CNOT gate is a rather important gate in that together with the Hadamard and phase shift gates they form a universal set of gates so that any n-qubit unitary operation can be performed using $4^n n$ of these gates. Note also the CNOT gate is reversible, as all quantum gates must be and this gate has two input and output qubits.

Toffoli Gate

The Toffoli gate is another very important quantum logic component. It is also known as the Controlled-Controlled-NOT gate as it performs a similar operation to the CNOT gate except requiring two control qubits instead of one. The quantum Toffoli gate is actually an analogue to the classical Toffoli gate which is also reversible. In short, the Toffoli gate flips the target bit if the first two bits are set.

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.21}$$

Note the Toffoli gate is not suitable for *universal* quantum computation although it allows a quantum computer to perform all the computations one can perform on a classical computer.

There is a generalised gate based on the Toffoli principle called a Deutsch gate, which is more suitable for quantum computation. The Deutsch gate can be used as a Controlled-Controlled-Switch as well. Essentially, this gate rotates the target bit if the two control bits are set.

2.1.4 Quantum Algorithms

So far only the technical basics of quantum computing have been described. As in classical computation this given hardware structure now has to be worked with. Well defined modifications can be obtained by applying particular gates. In order to perform certain tasks a specific sequence of gates has to be applied successively. This set of rules is defined in an algorithm [1]. With the quantum mechanical framework of quantum bits and their entanglement in mind it becomes obvious that classical algorithms can not be blindly used in a quantum mechanical context. On the contrary, they additionally have to consider and deal with properties like superposition and entanglement. This already shows one of the weak points of quantum computers: at the moment only a few and very specific quantum algorithms exist. Of course, some of them do provide a powerful way of solving problems and offer a significant speedup over their classical counterparts but only a few are fully developed. One of the most important and working quantum algorithms is Shor's algorithm for factoring large polynomials. In 2001, a group at IBM factorised 15 into 3 and 5 with a seven qubit quantum computer using this algorithm [11]. The algorithms implemented in this project are Grover's algorithm and the Quantum Full Adder.

Grover's Algorithm

Grover's algorithm is a quantum algorithm for searching an unsorted database which provides a quadratic speedup over its classical counterpart. A conventional computer would on average find its answer in $\mathrm{O}\left(\frac{1}{2}N\right)$ steps, whereas a quantum computer utilising Grover's algorithm would find the answer in $\mathrm{O}\left(\sqrt{N}\right)$ steps [12]. The algorithm is based on amplitude amplification [13] which means it increases the probability amplitude of the base state that is being searched for. The algorithm is applied to a n-qubit register where each base state has the same probability. Hence, the n-qubit state has to be prepared by a Hadamard gate giving equally sized amplitudes. Grover's algorithm consists of the repeated application of one Grover iterate which defines the following sequence of four transformations.

1. Controlled-f gate for kth base state
2. Hadamard gate
3. Controlled-f gate for 0th base state
4. Hadamard gate

Of course, the entry which is searched for is known and needed to specify the gates. A circuit representation [14] of one Grover iterate is shown in Figure 2.1.

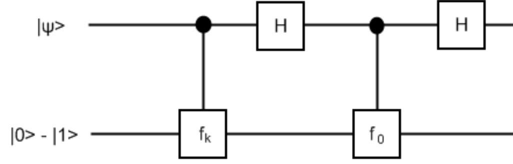

Figure 2.1: Circuit implementation of Grover's algorithm.

The first controlled-f gate already highlights the base state in question: it flips the sign of the kth state and consequently decreases the mean value of the amplitudes slightly. The combination of the H-controlled-f-H can be viewed as an inversion about the mean [12]. This increases the size of the amplitude of the kth state and therefore decreases the amplitudes of all the other base states. The next iterate again flips the sign of the kth state, slightly pushing down the mean value of the amplitudes, and the inversion about the mean increases the amplitude of the kth state. Repeated application of one Grover iterate amplifies the amplitude of the state in question.

For every number of qubits an optimal number of iterations can be calculated. This is done by considering a unit circle in the plane spanned by the wavefunction of the base states except the one that is searched for, $|\phi\rangle$, and the wavefunction of the state that is searched for, $|\omega\rangle$. Any combination of probability amplitudes will be a point on the unit circle. Applying Grover's algorithm is like rotating an arrow incrementally to point at the answer. The angle the arrow makes with the horizontal is defined as

$$\sin\theta = \frac{1}{\sqrt{2^n}} \tag{2.22}$$

where n is the number of qubits. Any state of the quregister can be described in terms of this angle.

$$|\psi\rangle = \sin\theta|\omega\rangle + \cos\theta|\phi\rangle \tag{2.23}$$

An application of i iterations transforms the state to be in position

$$|\psi\rangle = \sin((2i+1)\theta)|\omega\rangle) + \cos((2i+1)\theta)|\phi\rangle \tag{2.24}$$

Now, requiring a rotation close to $|\omega\rangle$ implies

$$\sin((2i+1)\theta) = 1 \tag{2.25}$$

which is satisfied by

$$i = \frac{\pi}{4\theta} - \frac{1}{2} \tag{2.26}$$

This specifies the number of times Grover's algorithm needs to be applied to get the best estimate for the solution.

Example of Grover's algorithm for a 2 QuBit device. For instance, search for the 2nd entry in a database.
Step 1: Initialisation. Applying the Hadamard gate to prepare the quregister in the required state of equal probability of all its base states (omitting normalisation factors).

$$|\psi\rangle = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{2.27}$$

Step 2: Applying a Controlled-f gate to flip the 2nd base state.

$$|\psi\rangle = \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \tag{2.28}$$

Step 3: Applying the series of Hadamard, flipping the 0th state and Hadamard.

$$|\psi\rangle = \begin{pmatrix} 2 \\ 2 \\ -2 \\ 2 \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} -2 \\ 2 \\ -2 \\ 2 \end{pmatrix} \quad \Longrightarrow \quad \begin{pmatrix} 0 \\ -8 \\ 0 \\ 0 \end{pmatrix} \tag{2.29}$$

Hence, in the case of a two qubit register the base state searched for has a probability of 100% to be measured. Although it is a unique case, this example illustrates the method of operation of Grover's algorithm. The above sequence of gates is equivalent to one Grover iterate.

Quantum Full Adder

One of the most basic forms of computer is a simple calculator performing basic arithmetic operations. In this project we have dealt with a quantum full adder.

In classical computers, binary addition is done by half and full adders. Of course, they can be optimised in many ways. The algorithm used in conventional machines can not simply be transferred to quantum computers. Although very similar, the quantum algorithm is different.

Classical Ripple Carry Adder. Ripple Carry Adders consist of a series of full adders preceded by a half adder. Classical full adders take three bits as input and produce two output bits. The three input bits are for example bit a, bit b, which are the two numbers to be added and finally bit c, called the carry bit. The output consists of the *sum* bit and a new *carry* bit [15].

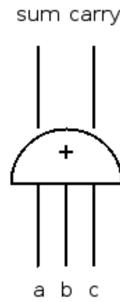

Figure 2.2: Classical Full Adder

A single full adder can therefore only add up to three bits. In order to add more bits, additional full adders are required. Each adder takes in two bits a and b, and also the *carry* bit of the previous adder. This will produce a sum which can be read off the sum outputs. It can also be considered as a conversion from unitary to binary.

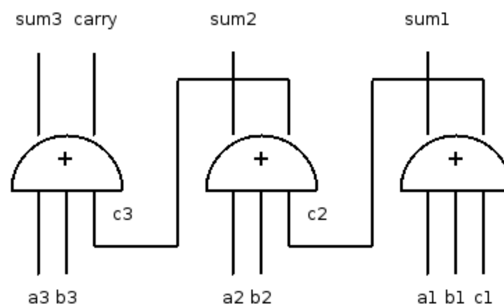

Figure 2.3: Classical Ripple Carry Adder

The weak point of the ripple carry adder is its low speed. Each individual adder has to wait for its predecessor to finish the calculation in order to start itself. For a long chain of adders this adds up do a remarkable time delay and is often optimized by slightly different structures, using magnifiers for example.

Quantum Ripple Carry Adder. The quantum mechanical analogue is very similar to its classical counterpart. A quantum algorithm, however, has to be reversible which can only be the case with an equal number of input and output qubits. Hence, the quantum full adder is a 4-input, 4-output device. The input still consists of qubit a, qubit b, the carrier and additionally an ancillary qubit. a and b are untouched in the output, the carrier turns into the sum while the ancillary qubit is the new carrier. This can be implemented with CNot and Toffoli gates as follows [15]:

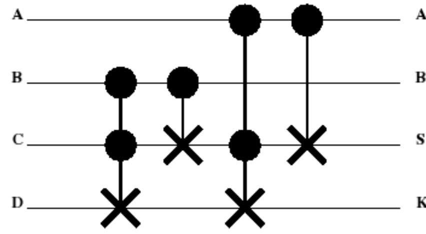

Figure 2.4: Quantum Full Adder

Just as the classical analogue the quantum full adders can be chained to form a quantum ripple carry adder. The carry output of the previous adder is input to the next.

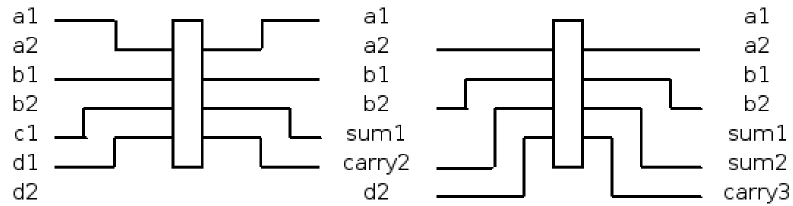

Figure 2.5: Quantum Ripple Carry Adder

The three qubits $carry_3$, sum_2, sum_1 encode the result in binary. An example of this would be to compute the sum of 3 and 2. Since $3 = 11$ and $2 = 10$ in binary this gives:

Table 2.1: Quantum full adder

	a1	a2	b1	b2	c1	d1	d2
initial state	1	1	0	1	0	0	0
first quantum full adder	1 1 1 1		0 0 0 0		0 0 0 1	0 0 0 0	
second quantum full adder		1 1 1 1		1 1 1 1		0 1 1 0	0 0 1 1
final state	1	1	0	1	1	0	1
	a1	a2	b1	b2	s1	s2	c2

The last three bits in reverse order give the required result:

$$3 + 2 = 11 + 10 = 101 = 5 \tag{2.30}$$

This works very well for qubits being in definite states 0 or 1, however they can also be in a superposition of these states. Although this is one of the greatest strengths of quantum computers (and in this specific case means that more than one sum can be calculated simultaneously) it also implies that the result will be a superposition of states from which it will be impossible to tell which result goes with which sum.

2.2 Basic Linear Algebra

Linear algebra is so-called because it deals with linear operators and their application to linear spaces. It is a powerful branch of mathematics and an integral part of the Quantum Mechanical model. It is because of this role in Quantum Mechanics that it is very important to the development of a quantum computer. A very brief and superficial description of the fundementals of linear algebra is given below. The specific way these concepts were implemented in the project are given in detail in the Implementation section, 3.2.

2.2.1 Linear Spaces

A linear space, also called a vector space, defines the operations and axioms for the set of vectors it is constructed from. Specifically, it is the addition and scalar multiplication of vectors that a linear space defines.

The axioms of a linear space dictate that the addition is

- Commutative: $a + b = b + c$
- Associative: $a + (b + c) = (a + b) + c$

and that the multiplication of a vector by a scalar is

- Distributive: $a(b + c) = ab + ac$
- Associative: $a(bc) = (ab)c$

Further axioms also define a zero and an identity vector, and an inverse, effectively defining subtraction. As such the operations defined by a linear space are addition, multiplication by a scalar and negation (and therefore subtraction) [16].

2.2.2 Linear Operators

Put concisely, a linear operator on a vector space is a linear map from the space to itself [16]. Given a linear operator $A : L \rightarrow L$ (where L is a linear space) then

- $A(x + y) = Ax + Ay$ for any x, y belonging to L
- $A(\lambda x) = \lambda(Ax)$ for any x belonging to L and where λ is a scalar [16]

It is possible for the actions of certain operators to be represented by matrices [16].
As mentioned above, this is a very short description of linear operators and spaces. The actual structure employed is given in 3.2.

Chapter 3

Implementation

3.1 Code Structure

The aim of the project was to write a simulator for a quantum computer in Java and to implement a simple quantum algorithm using it. However, we did not want the simulator to just be able to deal with one specific algorithm but rather to implement a powerful tool which allows for various internal representations and easy extension. This goal called for a certain level of abstraction.

As explained before a quantum computer consists of quantum bits, quantum registers and quantum gates. Thus, a basic working cycle of a quantum computer looks like the following:

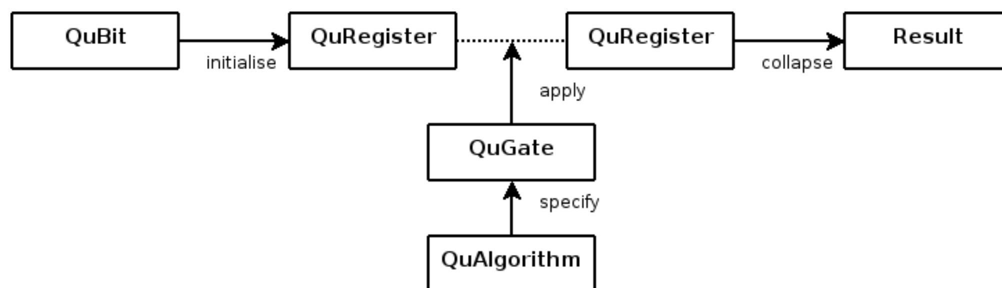

Figure 3.1: Diagram demonstrating a basic working cycle of a quantum computer

The quantum computer is initialised with a certain number of qubits which may be in randomly superposed states. The qubits entangle to form a quregister which is a representation of the wave function. The quregister is one of the most fundamental entities of a quantum computer since this is what is acted on by the quantum gates. A quantum algorithm merely defines a specific sequence of qugates which are consecutively applied to the quregister. The final quregister can not be split into qubits anymore since it still is an entanglement of all of them. However, now a measurement can be taken which corresponds to collapsing the wave function. The final result will be one specific base state collapsed into according to its probability amplitude. All additional

information contained in the wave function is going to be lost. This basic working cycle suggests abstract classes for the `QuBit`, `QuRegister`, `QuGate` and the `QuAlgorithm` which describes them as fundamental entities of a quantum computer independent of their precise implementation.

This leads to the second important point in designing a code structure: the chosen representation. A seemingly convenient representation describes qubits and quregisters in vector form and quantum gates as matrices.

However, it was expected this implementation would cause problems due to the memory requirements of a dense matrix representation. By defining a strong linear algebra framework and abstracting these ideas of linear space and linear operators the basic code structure would be independent of the final representation. Therefore in cases where a faster, functional representation was necessary, it could be incorporated without the need for designing a separate framework.

Code abstraction is always a tradeoff between generality and possible power of the implementation and the goal of actually getting the code to work. We decided for an abstraction of the basic ideas. Hence, we could implement two different representations of qugates, namely in matrix and functional form, as well as two algorithms being Grover's algorithm and a full quantum adder.

3.2 Linear Algebra

In order to produce the most general implementation of linear algebra, the final design concerns itself only with two types of objects: Linear Spaces and Linear Operators. The structure is designed to be totally self-contained. The linear algebra 'engine' is completely independent of any of the concepts introduced by the quantum computer classes (such as those discussed in the Code Structure Section 3.1), and as such runs alone without the need for any external classes.

3.2.1 Linear Spaces

From the linear algebra theory section (see 2.2), we know that a linear space defines the form of certain operations, such as addition and multiplication by scalar. The behaviour of scalars, vectors and matrices can be well defined using the operation axioms, and for this reason are considered as objects of type `LinearSpace`. This is not to say that, for example, a scalar is a linear space, but rather that it implements the operations defined by one. See figure 3.2 for a diagram of the class hierarchy.

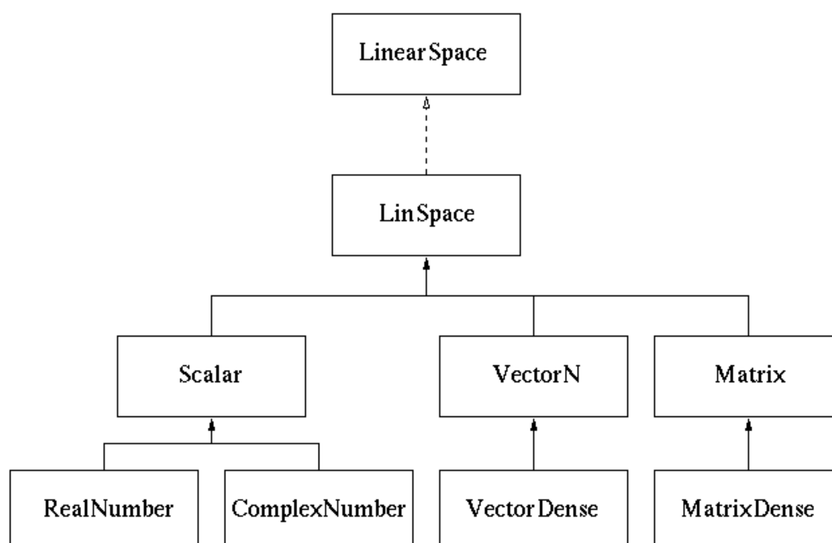

Figure 3.2: Diagram of underlying linear algebra structure

As can be seen in the diagram, at the top level we have the interface `LinearSpace`. This defines general rules such as addition and multiplication in a linear space. Notably,

- The sum of two `LinearSpace`s is a `LinearSpace`
- A `LinearSpace` multiplied by a scalar is a `LinearSpace`
- The inner product of two `LinearSpace`s is a `LinearSpace`
- The outer product of two `LinearSpace`s is a `LinearSpace`

The methods declared in the interface ensure that all implementing classes follow the above definitions. The first 'layer' of implementation is provided by the abstract base class `LinSpace`. This mainly provides type checking, ensuring that the objects being passed are of the correct type. `Scalar`, `VectorN` and `Matrix` extend this base class, providing abstract definitions of a scalar, an N component vector, and a simple m x n matrix.

Finally, in the third 'layer' of implementation, some concrete representations of these objects are given:

Scalars

A `Scalar` is represented by, in general, a `ComplexNumber` - that is, it is stored as two double precision variables holding the real and imaginary parts of that number. In order to minimise memory usage, scalars that consist of only real parts are represented by `RealNumber`s. These use only one double precision variable as storage.

Vectors

The only implemented subclass of `VectorN` was the `VectorDense` class. This was essentially an `ArrayList` of `LinearSpace`s that stored every term, regardless of its value. This meant that vectors consisting mostly of zero values (for example) would still use as much memory as a vector containing differing non-zero elements.

The structure of this linear algebra 'engine' was intended to facilitate extensions and the addition of more classes. In this case, the `VectorDense` class was created with a view to creating a `VectorSparse` class. Such a class would lower the memory required to store a vector by, as hinted in the paragraph above, storing only non-zero elements (for example).

Matrices

Just as `VectorN`s are concretely represented by `VectorDense`s, the implementation of a `Matrix` is handled by a `MatrixDense`. This represents the matrix by a two dimensional array of `LinearSpace`s, with no attempt to reduce the memory used for its storage. It was intended that a `MatrixSparse` class would be created that used a more intelligent and efficient way of storing elements, in the same way a `VectorSparse` would.

Exceptions

Finally, in order to police the calculations between different objects, a lot of type checking is necessary. This is to prevent linear operations between objects that are, for whatever reason, incompatible. Figure 3.3 shows the exceptions used for problems when dealing with `LinearSpace`s.

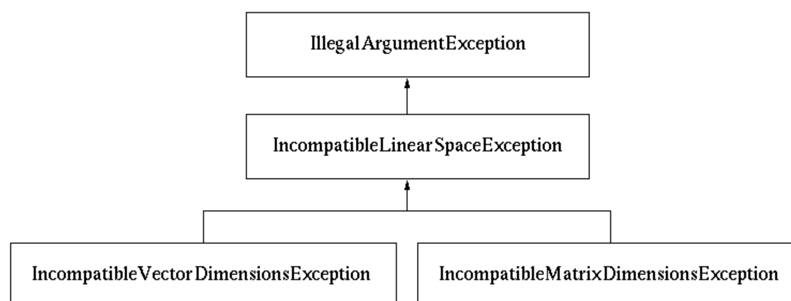

Figure 3.3: Class tree of exceptions relating to objects of type `LinearSpace`

`IncompatibleLinearSpaceExceptions` is the general exception thrown when the types of `LinearSpace` are not compatible. The `IncompatibleVectorDimensionsException` and `IncompatibleMatrixDimensionsException` provide a refination for vectors and matrices. The names of the exceptions are self-explanatory. All exceptions used are subclasses of `IllegalArgumentException` from the java API.

3.2.2 Linear Operators

The other important part of the linear algebra engine is the structure and implementation of linear operators. These act on the `VectorN`s discussed in 3.2.1.

For the purposes of the program, a linear operator is taken to have the following behaviour:

- An `Operator` applied to a `VectorN` produces a `VectorN`
- An `Operator` applied to an `Operator` produces an `Operator`

As before, an interface is used to define the methods ensuring this basic functionality, and an abstract base class `Op` gives the first 'layer' of implementation.

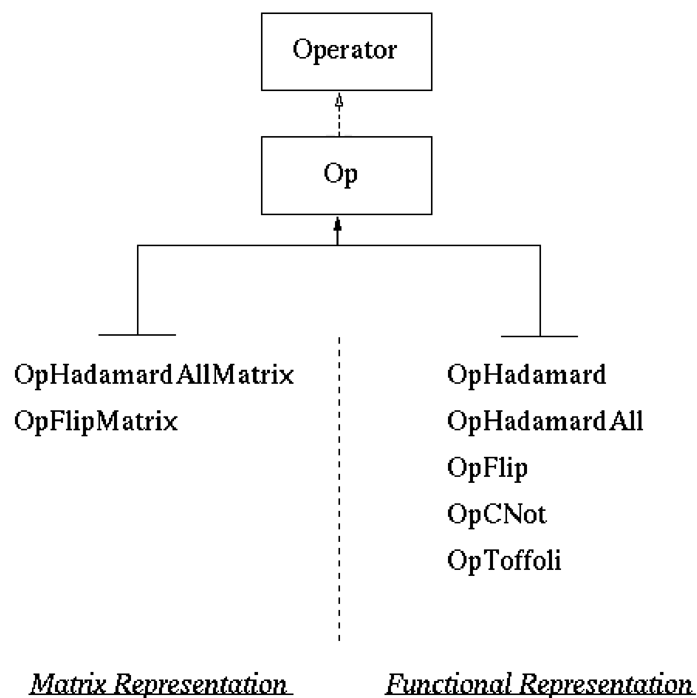

Figure 3.4: General structure of linear operator interfaces and classes

Figure 3.4 shows this structure. As for the linear spaces, exceptions were thrown in the event of misuse of an `Operator`. An `OperatorException` (not shown in diagram) was used for this purpose.

The `Operator` interface merely imposes the rules for applying operators to vectors and other operators. It does not specify a particular representation. As such, both matrix and functional forms of the operators could be written.

Matrix representation

Both `OpHadamardAllMatrix` and `OpFlipMatrix` act through the application of `MatrixDense`s to `VectorN`s. The `outerProduct` method of the matrix is used to produce Hadamard matrices of the correct size to act on a `VectorN` of a given size. The Hadamard and Flip matrices used are discussed in the Quantum Gates section 2.1.3. These were the only two operators to be coded in this way as it was apparent that the matrix form of the operators was a lot more computationally expensive, both in memory and in time taken. See 'Function Vs Matrix representation' (4.3) in the Discussion section for a more complete explanation of the decision to use functional over matrix representation.

Functional representation

Finding functional representations for all the quantum gates basically consisted of analysing the mathematical representations of the gates and applying them to registers of various sizes. Most mathematical representations only ever demonstrate the effect of the gate on its input qubits and while this would be most useful in a real world application for an emulator it was more useful to see the effect of a gate on the entire register.

OpHadamard

The functional version of the Hadamard gate was derived purely through application of the Hadamard matrix to different qubits in registers of various sizes and analysing how the state coefficients were altered. Keeping in mind that the operator matrix needs to be constructed by performing an outer product with Identity matrices in order to apply the gate to the correct qubit. By careful analysis a pattern for the state coefficient changes was found.

The Hadamard gate was applied to the 2nd qubit in a 3-qubit register above in equations 2.14 and 2.15.
We can see

$a'_0 = a_0 + a_2 \qquad a'_1 = a_1 + a_3 \qquad a'_2 = a_0 - a_2 \qquad a'_3 = a_1 - a_3, \ldots$

By analysis we can see that if we are applying the gate to the nth qubit, where n starts at 0 then we let $d = 2^n$. Then we cycle through each state in $|\psi'\rangle$; we begin by

$$a'_i = a_i + a_{i+d}$$

But when i is a multiple of d we switch from adding to subtracting and so

$$a'_i = a_{i-d} - a_i$$

So we have constructed $|\psi'\rangle$.

OpFlip

The Flip gate was simple to implement functionally, it was just a case of negating the necessary element in the vector representing the states of the register.

OpCNot

The CNot gate has a simple functionality and would be very easy to code would the simulator work with non entangled qubits. However, it becomes more complicated as it acts on a quregister and even more so because we wanted to implement it in a functional form. This requires a consideration of the base states of the quregister and to work out a pattern to tell which base states corresponds to what configuration of qubits. An example of a three qubit register is

$$|\psi\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad \Longrightarrow \qquad \begin{matrix} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{matrix} \tag{3.1}$$

In this example the register is in the $|101\rangle$ base state. If the first qubit is the controlling bit it would have to be set to $|111\rangle$ if the second bit is controlled or similarly to $|100\rangle$ if the third bit is the controlled qubit. This was achieved by spotting a regular pattern. Depending on the position of the qubit (where numbering starts with the leftmost qubit to be one) it changes its value from zero to one in regular intervals. With the help of these intervals (specific to each qubit position) it can be determined whether the controlling qubit is equal to one. If it is then it should be decided what value the target qubit has and according to this the value can be set either one interval further (if zero) or to one interval before (if one), where the interval depends on the target qubit. This procedure correctly implements the functionality of a CNot gate in functional form.

OpToffoli

The Toffoli gate is implemented very similarly to the CNot gate with the only difference of one additional loop because we need to consider two qubits rather than one. Only, if the set quregister base state proves to be in a state where both controlling qubits are one is the value of the target qubit swapped.

3.3 QuBit, QuRegister, QuGate

Equipped with this algebraic framework it now has to be applied in a quantum computational context.

As described previously, the fundamental entities of a quantum computer are qubits, quregisters and qugates. In our implementation all three of those are abstract classes. `QuBit` and `QuRegister` both hold a member variable of type `LinearSpace` since they will always be an element in linear space independent of their concrete representation.

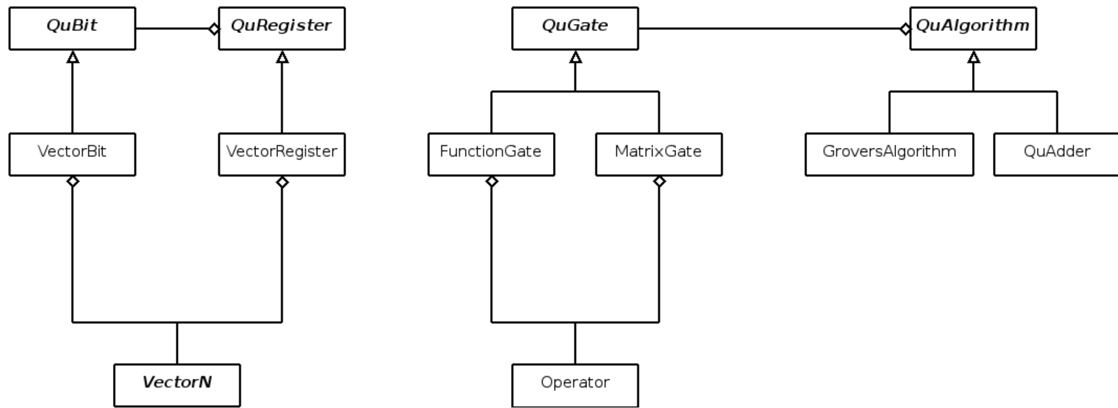

Figure 3.5: UML Diagram for QuBit, QuRegister and QuGate

Also common to both is a `normalise()` method which normalises the qubit or quregister to one. A specification of what the linear space is actually going to be as well as an implementation of `getState()` is left to be dealt with in the subclasses. Specific to the `QuRegister` is its `initialise()` method which builds the register by entangling (tensoring) the qubits. Another core feature of the `QuRegister` is its `collapse()` method which collapses the wave function into one specific state according to the probability amplitudes of its base states. The class `QuGate` establishes the connection of every qugate to be a linear operator. Hence, it holds a member variable of type `Operator`. According to this basic structure a rigid implementation is needed. This is done in the classes `VectorBit` and `VectorRegister` which specify the linear space to be of type `VectorDense`. In case of a change of representation these classes would have to be rewritten. The class `FunctionGate`, however, is still completely general and therefore maybe not even absolutely necessary. It nicely fits into the structure though and does enable a very easy extension.

A convenient way of tracking the changes in the quregister is a graphical output of the probability amplitudes of each base state in the register. The interface `StateListener` can be implemented by any class that wants to be informed of any change in the state of the register. This is done for instance in the `QuRegisterPlotter`.

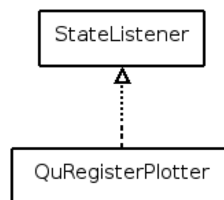

Figure 3.6: UML Diagram for QuRegisterPlotter

The output looks like the following.

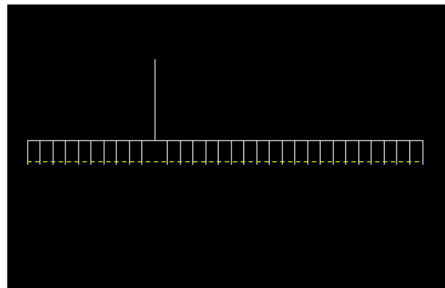

Figure 3.7: Example of QuRegisterPlotter

3.4 QuAlgorithm

To allow for an easy implementation of various algorithms the abstract class `QuAlgorithm` was created . This already proved to be very useful while coding the quantum adder which could easily be fit into this given structure. `QuAlgorithm` holds two member variables necessary to any algorithm, namely the `QuRegister` that the algorithm is applied to and an array of `QuGates`, which stores the sequence of qugates which is applied successively to the quregister. The core method is called `iterate()`. This is the place where the qugates are actually applied to the quregister. If the algorithm has to be iterated anyway, this method can be called repeatedly, if not, this method is only called once.

3.4.1 Grover's Algorithm

The class `GroversAlgorithm` extends `QuAlgorithm` and defines a single Grover iterate. Initially, the quregister is prepared by applying the Hadamard gate to the entire quregister.

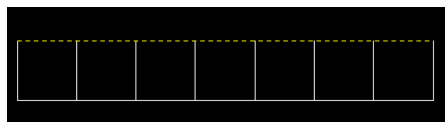

Figure 3.8: Uniform superposition of all possible states

Now, the sequence of qugates is defined consisting of two Controlled-f gates and two Hadamard gates. The first gate flips the sign of the state in question.

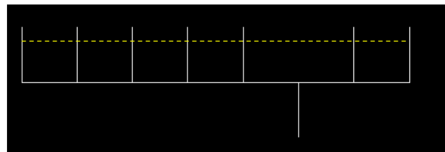

Figure 3.9: State after a single application of the Controlled-f gate

The next three gates, H-Controlled-f-H, invert each probability amplitude around the mean providing a considerable amplitude amplification of the searched bease state.

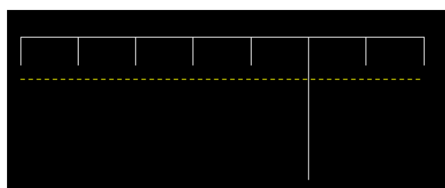

Figure 3.10: Inversion about the mean

Repeating this process increases the amplitude up to a certain maximum.

3.4.2 Quantum Adder

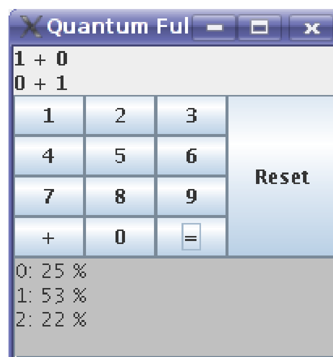

Figure 3.11: Quantum Adder GUI

The implementation of a full quantum adder fitted nicely into our structure and we found it very interesting to see how conventional tasks like addition can be dealt with by a quantum computer simulator.

Just as `GroversAlgorithm` the class `QuAdder` extends `QuAlgorithm` and defines the gate sequence as a series of two Toffoli and two CNot gates. Here, the difference is that the gates are not always applied to the same qubits, i.e. the gates applied to the

quregister look different. Therefore the four qubits considered in each adder have to be specified individually which is done by the `specifyQuBits()` method.
The tricky part of the quantum adder is not the correct application of the qugates but the in- and output which is dealt with by the class `AdderIO`. The user can input numbers in a range from 0 to 9. These are converted to binary and stored as representations of equal length by adding zeros to the left if necessary.

$$3 + 4 = 11 + 100 = 011 + 100 \tag{3.2}$$

Now, the qubits are initialised. They are either 0, 1, or in case of simultaneous addition, a superposition of both. Referring to figure 2.5, the qubits for the previous example are:

$$a_1 = 1 \qquad b_1 = 0$$
$$a_2 = 1 \qquad b_2 = 0$$
$$a_3 = 0 \qquad b_3 = 1$$

Simultaneous addition requires the qubits to be in superposition. The probability amplitudes of $|0\rangle$ and $|1\rangle$ are set according to their occurrence in the respective addend. With all the c and d qubits set to zero the quregister can now be initialised.

The `QuAdder` is iterated through as many iterations as there are qubits for one addend, i.e. three iterations in this example. The result can be extracted by considering the binary representation of the collapsed state. Important are the sum and the carry bits. In this example

$$sum_1 = 1$$
$$sum_2 = 1$$
$$sum_3 = 1$$
$$carry_4 = 0$$

which finally gives the required result:

$$3 + 4 = 0111 = 7 \tag{3.3}$$

The possible simultaneous addition is not very easy to work with since there can be many results and it is impossible to tell which result belongs to what calculation. The graphical user interface only allows to calculate two sums simultaneously. However, the code is not restricted to two only. A very nice demonstration of simultaneous addition gives the most simple example of adding 1+0 and 0+1 at the same time, which gives the results 0, 1 and 2, yet, with different probabilities (see Figure 3.11).

3.5 Frontend

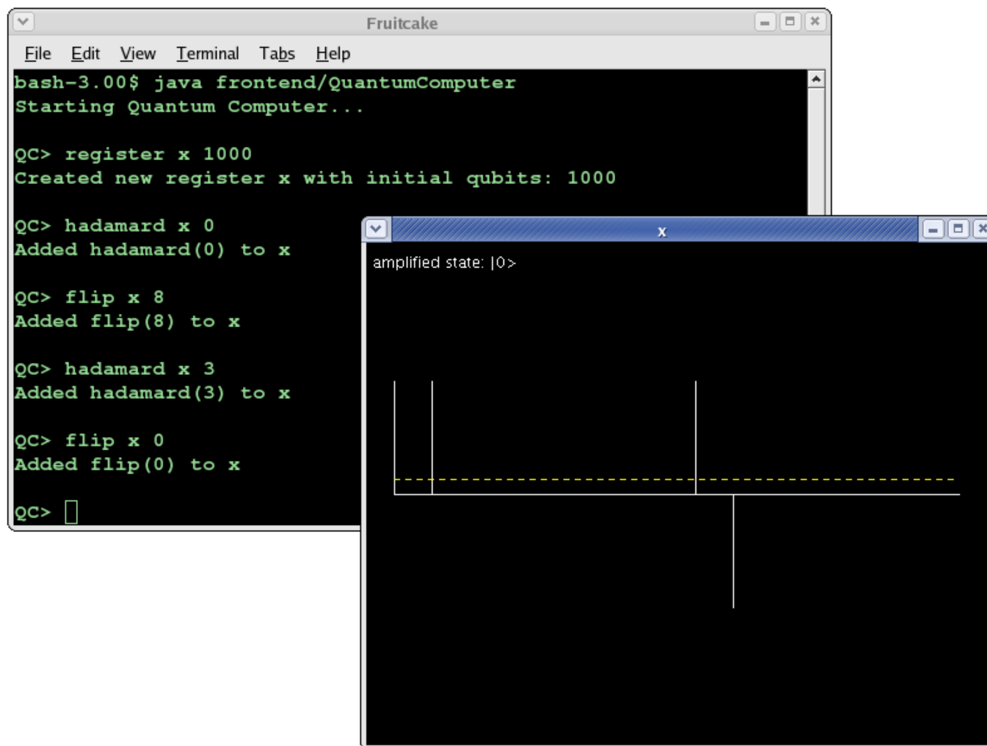

Figure 3.12: Frontend Screenshot

The quantum computer emulator was designed to be fully extensible, such that it is possible to create hardcoded Java classes that run algorithms using the linear operators available. However it was deemed useful to design a quick way of testing the application of operators to registers without the need for creating new test classes.

The frontend follows a simple command-line input format and users can see the actions of various operators on their system in real time. In the interest of making it simpler to perform repetitive tasks, the command parser was given the ability to accept external files as rudimentary 'scripts'. See the Scripting section on page 28 below.

3.5.1 Organisation

Essentially, the frontend consists of two modules, the `RegisterList` which is analogous to the computer's memory and the `ScriptingEngine` which accepts and parses user input from screen or file (see Figure 3.13).

Unfortunately there are a few shortcomings with this system, mostly due to necessitating the alteration of `ScriptingEngine`, where a list of all commands are stored, before a new command or `Operator` is useable. In future versions it would be far better to have some sort of wrapper for `Operators` that handles the command processing for

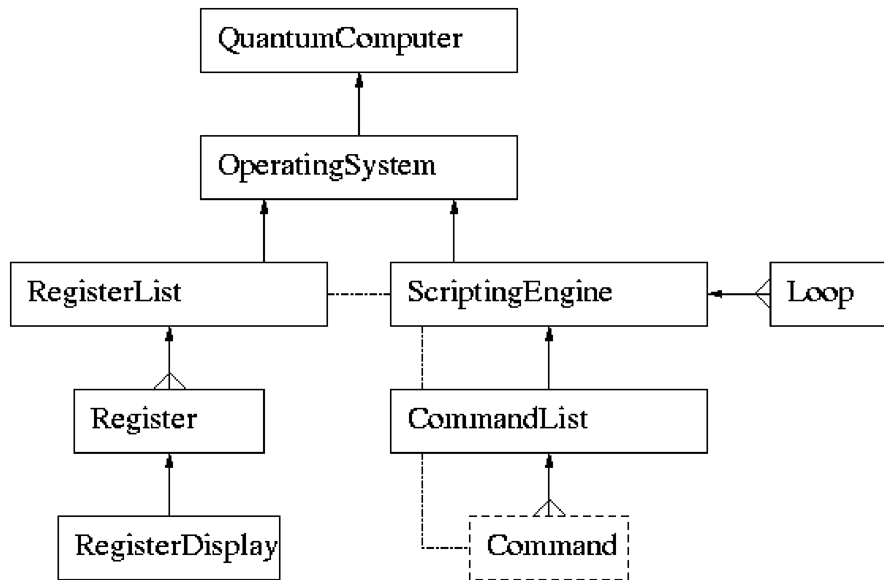

Figure 3.13: Frontend Code Structure

that operator. However the current system does ensure that anyone altering the source code knows exactly what a command is doing, and adding a new command is fairly simple thanks to the `CommandList` class and `Command` interface.

The `RegisterList` needs to be able to find a register by name and so a wrapper class was created to wrap `VectorRegister`. This also facilitates the initialisation of a register's states via a string of the initial classical settings of each qubit.

The `CommandList` essentially works in the same way, searching for a command that matches the user input before checking the arguments and executing the command. However this system requires that a pointer to the `RegisterList` is carried throughout the tree down to the command that is being executed.

3.5.2 Scripting

The scripting system was initially an extension of the screen-based user input, both consisting of the exact same syntax and commands. Later on, loops were implemented that execute commands within the loop for a certain number of iterations. Scripts are simply stored in any text file. The syntax chosen was as follows:

- Multiple commands are separated by a semicolon `;` or one command per line. The parser divides commands by both newlines and the semicolon character.
- Each command consists of the command name followed by the arguments it requires separated by a space. This means that register names should not have spaces in them (they cannot be created with spaces).
- Comments are preceded by `#`.

- Loops are defined by opening the loop with { `n` where n is the number of iterations and closing with }. Again these can be on the same line as the command as long as they are separated by semicolons.

A table of available commands can be found in the appendix. An example script performing 10 iterations of Grover's Algorithm, searching for state 3:

```
 # ------------- GROVER'S ALGORITHM --------------
register g 000
hadamardall g
{ 10
# the state you want to flip
flip g 3
hadamardall g
flip g 0
hadamardall g
}
```

Chapter 4

Discussion

4.1 Limitations of Grover's Algorithm

Although Grover's algorithm provides a considerable speedup in searching a database it does have a dramatic limitation: to this day it is unknown how to finally extract the useful information. The simplest form of a database can be thought of as a table where each value is allocated a unique number, the key. In conventional computing the problem is solved by finding the key since each value is connected uniquely with its value. This simple concept breaks down in quantum computing due to the entanglement of the qubits. What Grover's algorithm does is to search and find a given key correctly, however no information can be extracted. The qubits representing the keys are entangled as well as the qubits representing the values and, moreover, the keys entangle with the values. Basically, no unique allocation is possible anymore. Each key exists with every value in the database. Although the key may be found it is so far impossible to find its corresponding value. Grover's algorithm can not cope with this kind of problem, yet.

4.2 Limitations of Quantum Adder

The quantum full adder is another nice example of the power of a quantum computer on the one hand but the impossibility of actually using these advantages on the other hand.

The superposed state of a qubit offers the great possibility of calculating more than one sum at once. Naively, this may seem like a dramatic improvement and speed up. However, superposition in the initial state implies superposition of the final results with no way of linking calculation and result. This becomes obvious when considering every qubit in an equal superposition of its two states. Adding these qubits corresponds to calculating every possible sum within a certain range of integers. Yet, there is no information about what sum results from what calculation. In practise, qubits will always be in superposition. Thus, in the common sense of a calculator, a quantum adder is useless. It can, however, be used for a few, very specific tasks where the information that can be gained is sufficient. For example, for some algorithms it may

be enough to know whether the result is even or odd, which is an important result for Shor's algorithm. This information, of course, can be obtained.

Thus, as was said before, a quantum computer can be a particularly powerful device. In practice, however, it is probably only usable in a limited number of cases and for very specific tasks.

4.3 Function versus Matrix representation

Due to the linear algebra framework the quantum computer emulator was built on, we had a choice between using a dense matrix representation for all our gates or a functional implementation. Initially, a functional implementation was assumed to be the better option since it would not involve the generation of large matrices for registers of many qubits. We decided to check whether this was indeed the case by performing a series of measurements on the time taken to perform ten iterations of Grover's algorithm using functional Hadamard and Controlled-f gates against dense matrix representations of the same gates, increasing the number of qubits.

The results are displayed below:

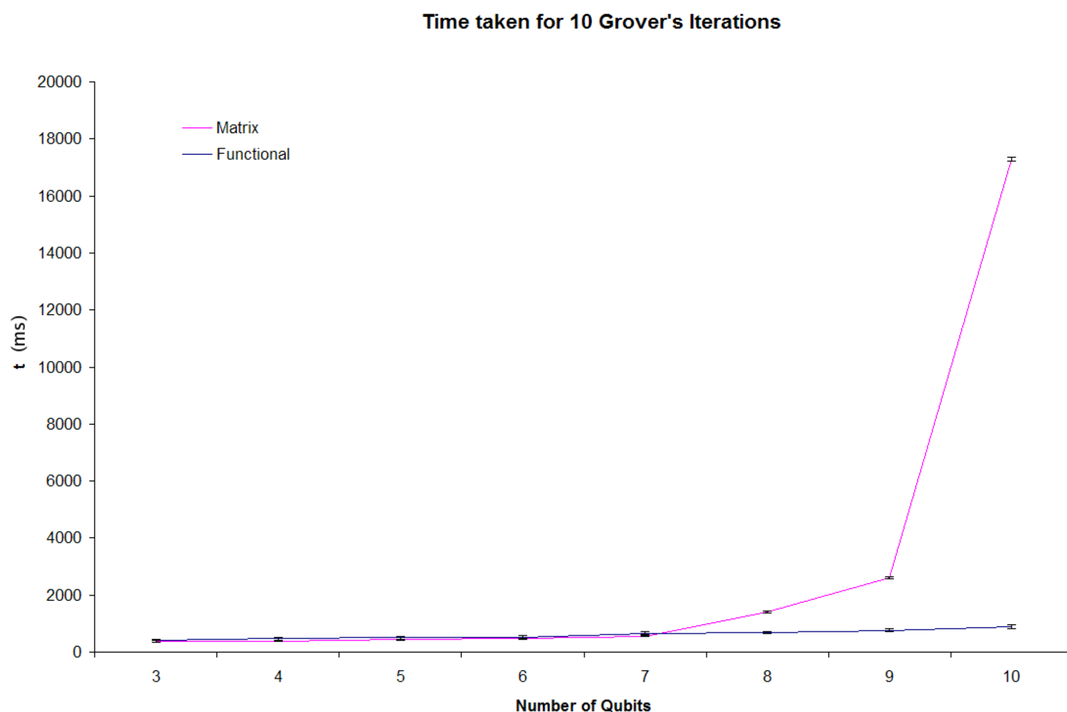

Figure 4.1: A comparison of the time taken to perform 10 iterations of Grover's algorithm in functional and matrix representations.

It is apparent at small register sizes, the functional representations are in fact slower than the dense matrix representations but then we see an exponential rise in the time taken for matrix-based gates to perform their operations on the register vector. This is mainly due to the matrices having to be constructed beforehand. Since the register grows as 2^n where n is the number of qubits then it stands to reason that we have 2^{2n} elements in a matrix being applied to the register vector. Therefore for a register of many qubits the dense matrix representation is far from ideal.

4.4 Group Management

Group management is an important part of the project. Our group met once a week with our supervisor where we dicussed new ideas we had and could ask questions. Most weeks we met a second time to dicuss next steps. Good communication is vital to an efficient working and a successful outcome. A forum is a great means of communicating with other group members. Free discussion forums can be acquired easily online. We've used Vanilla1 to organise group meetings or ask questions in the group.

Object orientation

Java offers the possibility to design an easily extensible structure, allowing us to split the code into logical parts. These being the implementation of the algebraic framework and the quantum computer components. The use of interfaces and abstract classes ensured that objects coded by different team members could be integrated effectively into the code. The code was far more understandable when divided into packages and it was of great help to all the members who were in charge of maintaining their own packages. For this project, the use of an object-oriented language was very successful.

For the coding we used a mixture of IDEs, Netbeans, Eclipse and EMACS, depending upon personal preferances.

The more code was being written the more urgent was the pressure for tidy code organisation. We decided to use subversion [17] which needed a while to get used to but was increadibly useful. Subversion stores old versions of the code and ensures, there is always a working version to fall back upon. It also ment that the latest changes could be easily accessed by the other group members so we only had one current version of the code. The repository was hosted online on a free subversion hosting page [18].

4.5 Prospects

Quantum computing is still in its infancy and only relatively recently did experimental and theoretical research in this area gain some attention. The practical realisation of quantum computers is still explored, especially for larger numbers of qubits. What already becomes obvious in our simulation is the need for more theoretical research. Only a limited number of algorithms for very specific tasks exists so far. It can be distinguished between those that provide speedups over conventional computers and

those that accomplish certain tasks but without significant speedup. As was discussed before, even existing algorithms often need extensions and further development to be fully operational.

What we have learned.

One of the most important points we have learned during our work on this project is that the common notion of quantum computers solving all existing problems has to be considered critically. In practice, quantum computers are very far from solving highly complicated mathematical problems [19]. In fact, it remains disputed that they ever will! Undoubtedly they provide interesting new ideas and may be valuable in certain application areas. However, it is questionable whether quantum computers will ever reach the efficiency of classical computers or ever replace them.

Apart from a very interesting insight into quantum computing we definitely learned to appreciate object orientated programming if we did not do so before. It was a new experience for all of us to start a slightly bigger project from scratch, to plan the structure ourselves and discover and use the possibilities Java offers. We do see it as a success to have a working implementation of Grover's algorithm and that we even had some time to consider code optimisation and another algorithm.

We also learned a lot about working and organising a team. With many ideas in a group it can be hard to make an absolute decision, especially since no member had ever attempted to code such an application before. Time management and splitting the work into manageable pieces were the biggest challenges faced.

What we would have liked to do.

Given more time we would have liked to consider some further aspects. More algorithms, especially Shor's algorithm for factorising integers, would have been of interest. Another idea was to implement a classical calculator on our simulator with all four basic arithmetic operations and not just addition. Code optimisation is also a very important area we would have liked to explore further.

All in all, we consider this project to be a success. First and foremost, it gave us many new ideas and insights.

Chapter 5

Conclusion

A quantum computer simulator based on the circuit model of quantum computation was presented. The quantum computational concepts are implemented in an abstract and easily extensible way. Grover's algorithm for searching an unsorted database runs successfully using our simulator. Furthermore, a quantum full adder was implemented showing the capability of a quantum computer simulator to cope with conventional tasks like addition.

Finally, the limitations of Grover's algorithm and the full quantum adder were explored which gave a greater understanding into the possibilities and usefulness of quantum computers.

Bibliography

[1] A. Ekert, P. Hayden, and H. Inamori. Basic concepts in quantum computation. *lectures given at les Houches Summer School on "Coherent Matter Waves", arXiv:quant-ph/0011013v1*, April 2001.

[2] A brief introduction of quantum computers [online]. Available from: http://www.cs.caltech.edu/ westside/quantum-intro.html [cited 17 February 2008].

[3] G. P. Berman, G. D. Doolen, R. Mainieri, and V. I. Tsifrinovich. *Introduction to Quantum Computers*. World Scientific Publishing, first edition, 1998.

[4] Wikipedia: Qubit [online]. Available from: http://en.wikipedia.org/wiki/Qubit [cited 15 March 2008].

[5] Bang bang method [online]. Available from: http://www.primidi.com/2006/01/07.html [cited 01 March 2008].

[6] Bucky balls [online]. Available from: http://en.wikipedia.org/wiki/Fullerene [cited 01 March 2008].

[7] R. T. Perry. Temple of quantum computing [online]. April 2006. Available from: http://www.toqc.com/.

[8] Quantum registers [online]. Available from: http://www.quantiki.org/wiki/index.php/Quantum_register [cited 01 March 2008].

[9] Isis innovation spring 2000 newsletter [online]. Available from: http://www.isis-innovation.com/news/newsletter/spring2000.pdf [cited 14 March 2008].

[10] Walsh-hadamard transforms: A literature survey [online]. Available from: http://www.ciphersbyritter.com/RES/WALHAD.HTM [cited 14 March 2008].

[11] Shor's algorithm [online]. Available from: http://en.wikipedia.org/wiki/Shors_algorithm [cited 2nd March 2008].

[12] L. K. Grover. A fast quantum mechanical algorithm for database search. *arXiv:quant-ph/9605043v3*, November 1996.

[13] P. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, first edition, 2007.

[14] I. G. Karafyllidis. Quantum computer simulator based on the circuit model of quantum computation. *IEEE Transactions on Circuits and Systems - I: Regular papers*, August 2005.

[15] P. Gossett. Quantum carry-save arithmetic. *Silicon Graphics*, August 1998.

[16] E. B. Vinberg. *A Course in Algebra*. American Mathematical Society, 2003.

[17] Suversion [online]. Available from: http://subversion.tigris.org/ [cited 01 March 2008].

[18] Beanstalkapp [online]. Available from: http://www.beanstalkapp.com/ [cited 14 March 2008].

[19] S. Aaronson. The limits of quantum computers. *Scientific American, Volume 298 Number 3*, March 2008.

APPENDIX

Table 1: available frontend commands

Command	**Arguments**	**Action**
help		Outputs a list of all available commands.
exit		Exits the frontend.
echo	*text*	Outputs *text* to screen.
file	*file1 file2* ...	Runs the scripts in *file1, file2,* ....
starttimer		Begins a stopwatch.
endtimer		Ends the stopwatch and outputs the time in ms.
register	*name bitstring*	Creates a register called *name* and initialises it based on the string of 0s and 1s in *bitstring*.
init	*register bitstring*	Reinitialises *register* with *bitstring*.
kill	*register*	Deletes *register*.
collapse	*register*	Makes a measurement and collapses *register*.
hadamard	*register qubit*	Applies the Hadamard gate to qubit *qubit* in *register*, where qubit numbering begins with 0.
hadamardall	*register*	Applies Hadamard gates to all the qubits in *register*.
flip	*register state*	Flips *state* in *register*.
cnot	*register control target*	Applies a CNOT gate with a *control*-qubit and *target*-qubit in *register*.
toffoli	*register control1 control2 target*	As above but with a second *control2*-qubit.