

THE UNIVERSITY OF EDINBURGH
SCHOOL OF PHYSICS AND ASTRONOMY

**Quantum Computing Project:
Report**

by
MICHAEL CHIANG
GENNARO DI PIETRO
WILLIAM McNICHOLS
CHRISTOPH MESSMER

Edinburgh, 24th of March 2015

Contents

1	Introduction	2
1.1	Aims	2
1.2	Background	2
2	Theory	3
2.1	Qubits	3
2.1.1	Generalised bits	3
2.1.2	Measurement	3
2.2	Quantum Register	4
2.3	Quantum Gates	4
2.3.1	Pauli-X Gate	5
2.3.2	Hadamard Gate	5
2.3.3	Phase Gate	5
2.3.4	Control Gates	6
2.3.5	Extension to bigger registers	6
2.3.6	Gate representations	7
2.4	Quantum algorithms	8
2.4.1	Grover's Algorithm	8
2.4.2	Shor's Algorithm	9
3	Implementation	10
3.1	Project organisation	11
3.1.1	Java	11
3.1.2	Git	12
3.2	Development environment	12
3.2.1	Github	12
3.3	Programme structure	12
3.3.1	Overview	12
3.3.2	The Computer	14
3.3.3	Matrix	16
3.3.4	Graphics User Interface	17
3.4	Programme execution	18
4	Results	20
4.1	Grover's Algorithm	20
4.1.1	Rotation of the state vector	20
4.1.2	Computational time	20
4.2	Shor's Algorithm	22
5	Discussion	22
5.1	Matrix or functional representation	22
5.2	Improvements and further steps	22
6	Conclusion	22
7	Appendix	22
7.1	Grover's Algorithm	22
7.2	Details	22
	References	23

1 Introduction

This report is part of the result within the context of the course 'Quantum Computing Project' at the University of Edinburgh and is meant to be comprehensible to 3rd year undergraduate students of physical science (especially informatics and physics).

We start off describing the [aims](#) of the project and are going to provide a [background](#) for the topic of Quantum Computing. That is followed by a chapter about the [theory](#) needed for the course which is, however, purposely not intended to be exhaustive (see [references](#) for further information).

After that we will describe the [implementation](#) of the project including how we organised the programming, which development environment we used, how we structured the program, etc.

We finish with a chapter about the [results](#) followed by a [discussion](#).

1.1 Aims

The aims of the project:

- **Comprehend quantum computing**
One goal is to get familiar with quantum computing as a generalisation of conventional computing.
- **Programming**
The main goal is to simulate a quantum computer on a conventional classical computer. This includes programming basic concepts like [qubits](#), [quantum registers](#) and [quantum gates](#). Finally it should be possible to run [quantum algorithms](#) like [Grover's algorithm](#) (optional: [Shor's algorithm](#)).
- **Presenting results**
This includes not only this report and a verbal presentation but also proper documentation of the programming code to make it comprehensive to other programmers.
- **Teamwork and organisation**
A project like this needs organisation and division of task but also successful communication between all group members. It is a further goal to encourage teamwork and organisation skills.

1.2 Background

The history of computers reaches back to the middle of the 19th century when a design for an Analytical Engine was proposed by Charles Babbage who is considered to be one of the early pioneers of computation. However, for almost 100 years this branch stayed an interesting but rather conceptional one until the invention of the transistor in 1925. The first working computers were built in the 1940s and up to today computers work principally the same way.

Quantum computation on the other hand is a quite recent research field which emerged from the physics of quantum mechanics (1920s). In 1982 Richard Feynman theorised that there seemed to be essential difficulties in simulating quantum mechanical system on classical computers and suggested that a quantum computer would solve these issues.^[5]

Remarkable theoretical breakthroughs in the 1990s followed, when Peter **Shor** demonstrated that essential problems – like factorising integers – could be solved far more efficiently on quantum computers than on conventional, classical computers. Besides Shor's algorithm Lov **Grover** proposed another algorithm in 1995 (only one year later) showing that the problem of conducting a search through some unstructured search space is as well more efficient on quantum computers.

The practical challenges for building a quantum computer are high and therefore the realisation of real quantum computers is still in it's infancy. However, in 2001 the first real quantum computer

was able to factorise 15 into its prime numbers (3 and 5) by using a 7-qubit system.^[7] Since then experimental progress is booming but the state-of-the-art is still a fair way off from practical (and even less daily-life) usage.

2 Theory

In this chapter we introduce the basic concepts of quantum computation, starting off with definitions of **qubits**, **quantum registers** and the presentation of several **quantum gates**. Afterwards we will talk about two **quantum algorithms** that we implemented in our virtual quantum computer. The last chapter will briefly talk about the challenges of **building a real quantum computer**.

2.1 Qubits

2.1.1 Generalised bits

A qubit (from *quantum bit*) is the smallest unit in a quantum computer and therefore the quantum mechanical **generalisation of a classical bit**, as it is used in computers nowadays. A classical bit has one and only one of the two possible states

$$|0\rangle \quad \text{or} \quad |1\rangle \quad (1)$$

at the same time, whereas a qubit is able to be in a state $|\Psi\rangle$ which is a superposition of these two classical states:

$$|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle, \quad \text{where } |\alpha|^2 + |\beta|^2 = 1 \quad (2)$$

One can depict the states via matrices with basis $(|0\rangle, |1\rangle)$:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad |\Psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (3)$$

2.1.2 Measurement

The superposition of states leads to a new understanding of measurement. There are two things to consider:

1. Probabilities P

Given a classical state ($|\Psi\rangle$ either $|0\rangle$ or $|1\rangle$) the result of a measurement is certain (and trivial). That's no longer true for the quantum state: Given the state $|\Psi\rangle$ in Eq. 2, it is solely possible to calculate the **probabilities** P_Ψ of the outcome:

$$P_\Psi(0) = |\langle 0|\Psi\rangle|^2 = \left| \underbrace{\alpha \langle 0|0\rangle}_{=1} + \underbrace{\beta \langle 0|1\rangle}_{=0} \right|^2 = |\alpha|^2 \quad (4)$$

$$P_\Psi(1) = |\langle 1|\Psi\rangle|^2 = \left| \underbrace{\alpha \langle 1|0\rangle}_{=0} + \underbrace{\beta \langle 1|1\rangle}_{=1} \right|^2 = |\beta|^2 \quad (5)$$

2. Collapse of $|\Psi\rangle$

In classical measurements it is fair to say that the measurement itself has no (noticeable) influence on the result. This is no longer true in quantum mechanics: The wave function

$|\Psi\rangle_i$ **collapses after a measurement** to a projection onto the measured eigenstate and therefore becomes a different state $|\Psi\rangle_f$:

$$|\Psi\rangle_i = \alpha|0\rangle + \beta|1\rangle \xrightarrow{\text{Measurement: Value } m} |\Psi\rangle_f = \begin{cases} |0\rangle, & \text{if } m = 0 \\ |1\rangle, & \text{if } m = 1 \end{cases} \quad (6)$$

2.2 Quantum Register

A quantum register of size n is a **collection of n qubits**. Therefore we get $N \equiv 2^n$ basic states:

$$|b_{n-1}\rangle \otimes |b_{n-2}\rangle \otimes \dots \otimes |b_1\rangle \otimes |b_0\rangle \quad (7)$$

where $b_i \in \{0, 1\}$. One can interpret this chain of zeros and ones as binary code, able to store numbers in the range of $[0, 1, \dots, N - 1]$. For example for a 3-qubit system we get:

$$\begin{aligned} |0\rangle \otimes |0\rangle \otimes |0\rangle &\equiv |000\rangle \equiv |0\rangle & |1\rangle \otimes |0\rangle \otimes |0\rangle &\equiv |100\rangle \equiv |4\rangle \\ |0\rangle \otimes |0\rangle \otimes |1\rangle &\equiv |001\rangle \equiv |1\rangle & |1\rangle \otimes |0\rangle \otimes |1\rangle &\equiv |101\rangle \equiv |5\rangle \\ |0\rangle \otimes |1\rangle \otimes |0\rangle &\equiv |010\rangle \equiv |2\rangle & |1\rangle \otimes |1\rangle \otimes |0\rangle &\equiv |110\rangle \equiv |6\rangle \\ |0\rangle \otimes |1\rangle \otimes |1\rangle &\equiv |011\rangle \equiv |3\rangle & |1\rangle \otimes |1\rangle \otimes |1\rangle &\equiv |111\rangle \equiv |7\rangle \end{aligned}$$

We call this collection the **computational basis** of our register.

However, in contrast to a classical system, a quantum register is able to be in a **state of superposition** which turns out to be the fundamental advantage for quantum computation. If for example the second qubit is set to a superposition $|\Psi_{b_1}\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ the total state of the register will be:

$$|\Psi^{\text{tot}}\rangle = |\Psi_{b_2}\rangle \otimes |\Psi_{b_1}\rangle \otimes |\Psi_{b_0}\rangle \quad (8)$$

$$= |0\rangle \otimes \left[\frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \right] \otimes |1\rangle \quad (9)$$

$$= \frac{1}{\sqrt{2}} [|001\rangle - |011\rangle] \quad (10)$$

$$\equiv \frac{1}{\sqrt{2}} [|1\rangle - |3\rangle] \quad (11)$$

Eq. 11 is always reducible to a (tensor) product of three single states, as Eq. 8 suggests. This is not always the case. Consider the 2-qubit system where

$$|\Psi^{\text{ent}}\rangle = \frac{1}{\sqrt{2}} [|00\rangle + |11\rangle]. \quad (12)$$

There is no way to separate this wave function into a (tensor) product of two states $\{|\Psi_{b_1}\rangle, |\Psi_{b_0}\rangle\}$. Thus, the state $|\Psi^{\text{ent}}\rangle$ is called **entangled**.

2.3 Quantum Gates

After defining the quantum register, we now want to process it through a number of so-called quantum gates. These are (mathematically spoken) **unitary operations** applied to our register in order to change its total state $|\Psi^{\text{tot}}\rangle$. Since we work in the Hilbert space, all our operations are **linear**, therefore we can represent any gate working on a n -qubit register by a $N \times N$ matrix.

In the following subsections we will first introduce the most important gates used in our project, and then speak about generalisations of these gates for bigger registers.

2.3.1 Pauli-X Gate

The first example for a simple 1-qubit gate is the Pauli-X gate. It simply maps the state $|0\rangle \rightarrow |1\rangle$ and vice versa and is therefore equivalent to the logic NOT. The representing matrix in the computational basis $\{|0\rangle, |1\rangle\}$ is:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad (13)$$

$$X |0\rangle = |1\rangle \quad (14)$$

$$X |1\rangle = |0\rangle \quad (15)$$

$$(16)$$

$$|b\rangle \xrightarrow{X} |1-b\rangle$$

2.3.2 Hadamard Gate

A common gate in quantum computation is the Hadamard gate. It performs the Hadamard transformation on a single qubit system in the following way:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad (17)$$

$$H |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad (18)$$

$$H |1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \quad (19)$$

$$|b\rangle \xrightarrow{H} \frac{1}{\sqrt{2}} [(-1)^b |b\rangle + |1-b\rangle]$$

The Hadamard gate is a ‘real’ quantum gate since it is able to set the state to a superposition of basic states.

For a n -qubit system it is necessary to define on which qubit a gate is acting. In the following we will use the subscript to depict this: The gate G_k is acting on the k -th qubit.

Note that a combination of Hadamard gates acting on every single qubit in a n -qubit register with initial state $|\Psi\rangle = |00\dots 0\rangle$ will lead to an **uniform superposition** of all basic states:

$$\left(\prod_{k=0}^{n-1} H_k \right) |\Psi\rangle = H_{n-1} \underbrace{|b_{n-1}\rangle}_{=|0\rangle} \otimes \dots \otimes H_0 \underbrace{|b_0\rangle}_{=|0\rangle} \quad (20)$$

$$= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \dots \otimes \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad (21)$$

$$= 2^{-\frac{n}{2}} \sum_{k=0}^{N-1} |k\rangle \quad (22)$$

2.3.3 Phase Gate

The Hadamard gate already uses special properties of quantum computation, but all operations are part of the real subspace of the Hilbert space. In general the gates and quantum register can operate on a complex vector space. The phase gate is such a gate which is defined for a single qubit system as:

$$R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} \quad (23)$$

$$R_\phi |0\rangle = |0\rangle \quad (24)$$

$$R_\phi |1\rangle = e^{i\phi} |1\rangle \quad (25)$$

$$|b\rangle \xrightarrow{R_\phi} e^{ib\phi} |b\rangle$$

2.3.4 Control Gates

Apart from single qubit gates, there are also control gates. The gate

2.3.5 Extension to bigger registers

As roughly mentioned before, any single qubit gate can be applied to the k th qubit of a n -qubit register. The challenge now is how to construct a matrix representation of a single qubit gate when it is applied to a register with multiple qubits. Mathematically, this involves taking the tensor product of the matrix representing the gate on the desired target qubit with the identity on others. Consider the example of applying a single qubit gate G on the left qubit in a 2-qubit system. The resulting matrix G_{tot} is:

$$G_{\text{tot}} = G_1 \otimes \underbrace{G_0}_{=\mathbb{I}_2} \quad (26)$$

$$= \begin{pmatrix} g_{00} & g_{01} \\ g_{10} & g_{11} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (27)$$

$$= \begin{pmatrix} g_{00} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & g_{01} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ g_{10} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & g_{11} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \quad (28)$$

$$= \begin{pmatrix} g_{00} & 0 & g_{01} & 0 \\ 0 & g_{00} & 0 & g_{01} \\ g_{10} & 0 & g_{11} & 0 \\ 0 & g_{10} & 0 & g_{11} \end{pmatrix} \quad (29)$$

The general expression for n -qubit systems is analogous:

$$G_{\text{tot}} = G_{n-1} \otimes \dots \otimes G_k \otimes \dots \otimes G_0 \quad (30)$$

$$= \mathbb{I}_2 \otimes \dots \otimes \begin{pmatrix} g_{00} & g_{01} \\ g_{10} & g_{11} \end{pmatrix} \otimes \dots \otimes \mathbb{I}_2 \quad (31)$$

Instead of carrying out this tensor product explicitly, which is likely to be computationally expensive and inefficient, we notice that the linear operation serves as an active transformation of the state vector. From linear algebra, we know the columns within the matrix that describes such a transformation are the basis vectors of the transformed basis (or the new basis) expressed in terms of the old basis. Therefore, we can construct the matrix for this transformation easily if we know how each basis state in the new basis is described by the old basis.

This turns out to be a simple task to do on a classical computer using bitwise operations, provided that we know how the gate operates on a single qubit. Consider a n -qubit system and we wish to apply an arbitrary gate G on the k th qubit. Suppose G has the following effect on the states $|0\rangle$ and $|1\rangle$ of a single qubit:

$$G|0\rangle = \alpha_0|0\rangle + \beta_0|1\rangle \quad (32)$$

$$G|1\rangle = \alpha_1|0\rangle + \beta_1|1\rangle \quad (33)$$

We can write this more compactly as

$$G|y\rangle = \alpha(y)|0\rangle + \beta(y)|1\rangle, \quad (34)$$

where we let $\alpha(0) = \alpha_0$, $\alpha(1) = \alpha_1$ and similarly for β . We wish to find the relationship between the new basis $|x'\rangle$ and the old basis $|x\rangle$ such that $|x'\rangle = G|x\rangle$. For this discussion, it would be most convenient to write each basis state in terms of the computational basis, which is $|x\rangle \equiv$

$|x_{n-1}x_{n-2}\cdots x_0\rangle$ where $x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0$. The new basis state $|x\rangle'$ is therefore given by:

$$|x\rangle' = G|x\rangle = G(|x_{n-1}\rangle|x_{n-2}\rangle\cdots|x_k\rangle\cdots|x_0\rangle) \quad (35)$$

$$= |x_{x-1}\rangle \otimes |x_{n-2}\rangle \otimes \cdots \otimes G|x_k\rangle \otimes \cdots \otimes |x_0\rangle \quad (36)$$

where we used the fact that G only operates on the k th qubit. Using equation 34, we can see that every new basis state can be described by two, and only two, basis vectors from the old basis:

$$|x\rangle' = |x_{x-1}\rangle \otimes |x_{n-2}\rangle \otimes \cdots \otimes (\alpha(x_k)|0\rangle + \beta(x_k)|1\rangle) \otimes \cdots \otimes |x_0\rangle \quad (37)$$

$$= \alpha(x_k)|x_{n-1}x_{n-2}\cdots 0\cdots x_0\rangle + \beta(x_k)|x_{n-1}x_{n-2}\cdots 1\cdots x_0\rangle \quad (38)$$

This means that there would only be two elements in each column within the matrix G . Let $G(i, j) \equiv G_{ij}$ and since $G_{ij} = \langle i|G|j\rangle$, the two elements in column x of G are:

$$G(x_{n-1}x_{n-2}\cdots 0\cdots x_0, x_{n-1}x_{n-2}\cdots x_k\cdots x_0) = \alpha(x_k) \quad (39)$$

$$G(x_{n-1}x_{n-2}\cdots 1\cdots x_0, x_{n-1}x_{n-2}\cdots x_k\cdots x_0) = \beta(x_k) \quad (40)$$

Therefore, we can find all elements of the matrix by iterating over the column index x , $x \in [0, N-1]$. We notice the value of the element depends on x_k , or whether the k th bit is 0 or 1. This can be found by performing the bitwise AND operation between x and a *mask* with value 2^{k-1} . If the result is zero, $x_k = 0$; otherwise, $x_k = 1$. We can also toggle the k th bit in x to find the row index of the two elements by performing an XOR operation between x and the mask.

The matrix representation of a control gate can also be found using the same procedure. From the previous section, we know the operation in a control gate is only performed if the control qubits are all 1. Consider the gate G that is controlled by m qubits with values c_1, c_2, \dots, c_m , $c_i \in \{0, 1\}$. The new basis is related to the old basis by:

$$|x\rangle' = G^{c_1c_2\cdots c_m}|x\rangle = \begin{cases} G|x\rangle & \text{if } c_1 = c_2 = \dots = c_m = 1 \\ \mathbb{I}|x\rangle & \text{otherwise} \end{cases} \quad (41)$$

Hence, we can still iterate over the column index x to find the elements, except we need to check whether each of the control bits $\{c_i\}$ in x is 1 using the AND operation as described above. If the bits are all 1, we would obtain the two elements as stated in equation 39 and 40; otherwise, the new basis state would be the same as the old basis state, so the only non-zero element in column x would be $G(x, x) = 1$.

2.3.6 Gate representations

The resulting matrix G_{tot} is in general a $N \times N$ -matrix with two and only two non-zero entries in every row and column. That means that only $2 \cdot 2^n$ of 2^{n^2} entries are non-zero. The ratio is 2^{1-n} which rapidly goes towards zero for big n . Thus, most entries of the gate matrices will be zero.

This leads to the consideration of the following three possible representations:

1. Dense matrix representation

A dense matrix representation is the standard representation, storing every single entry of a matrix. This leads to two major disadvantages in our special case:

(a) Memory

A classical computer reserves a certain amount of memory for every entry of a conventional matrix, no matter if the value is zero or non-zero. Therefore using dense matrices for big quantum registers might cause a serious lack of working memory.

(b) *Calculation*

We get a lot of trivial (and unnecessary) calculations using the standard matrix multiplication rule.

The quantum register, however, has mostly non-zero entries for most of the steps of the usual algorithms.

2. Sparse matrix representation

A sparse matrix representation only stores non-zero elements of a matrix in a list. Therefore every non-zero element requires a positioning index and the number itself to store.

3. Functional representation

Since there are only two non-zero elements in a row (in our special case), a functional representation would be reasonable.

2.4 Quantum algorithms

After defining quantum gates it is now simple to predict the next step: A so-called quantum algorithm defines in what manner a quantum register is applied to a sequence of quantum gates in order to achieve a certain computation result. Two very important algorithms are Grover's algorithm and Shor's algorithm which we are going to introduce next.

2.4.1 Grover's Algorithm

Grover's algorithm is also known as the quantum search algorithm. On a classical computer, searching through a non-ordered list with N entries would require $O(N)$ operations. This can be sped up to $O(\sqrt{N})$ operations using a quantum circuit that implements Grover's algorithm. Different from normal search operations, this algorithm searches through the *index* of the elements (which ranges from 0 to $N - 1$) rather than the elements themselves. The index of an element corresponds to a basis state in the register. By using a 'black box' (also known as an *oracle*) which can *recognise* the solution states without *knowing* them explicitly, the algorithm finds the solutions by repeatedly performing a certain set of transformations on the register to maximise the probability of measuring the solution states.

Suppose we want to search through a list of N elements, and there are M solutions to the search, where $1 \leq M \leq N$. For simplicity, let $N = 2^n$ such that the index of the elements can be exactly represented by n qubits in the register. As mentioned above, the oracle has the ability to recognise the solution states. This can be signalled by changing the state of a qubit $|q\rangle$ in the oracle. Specifically, if we let $|x\rangle$ to be a basis state associated with a particular element and let $f(x) = 1$ if $|x\rangle$ is a solution and $f(x) = 0$ if it is not, the oracle performs the following unitary operation (denoted as O):

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle \quad (42)$$

where \oplus denotes the addition modulus 2 (which is the same as the binary XOR operation in this case). Hence, $|q\rangle$ would only change its state if $f(x) = 1$, which is when the oracle has recognised a solution. For convenience, we can prepare the qubit in the oracle to be $|q\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ such that it does not change in the operation:

$$|x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \xrightarrow{O} (-1)^{f(x)} |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad (43)$$

Therefore, the oracle marks the solution states by applying a phase shift of -1 to them:

$$|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle \quad (44)$$

To maximise the probability of the solution states, the algorithm repeatedly applies the following set of operations on the register, which, as a whole, is known as the Grover's diffusion operation:

1. Apply the oracle O
2. Apply the Hadamard operation on all qubits $H^{\otimes n}$
3. Perform a phase shift of -1 on every state of the computational basis except $|0\rangle$:

$$|x\rangle \longrightarrow -(-1)^{\delta_{x0}} |x\rangle \quad (45)$$

4. Apply the Hadamard operation on all qubits again $H^{\otimes n}$

Step 2 to step 4 is also known as the *inversion about mean* operation. Mathematically, this iteration can be written compactly as:

$$G = (2|\psi\rangle\langle\psi| - \mathbb{I})O. \quad (46)$$

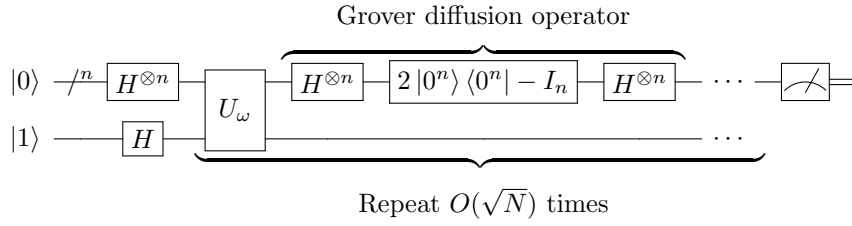


Figure 1: Circuit of Grover's Algorithm^[1]

Fig. 1 shows a schematic circuit diagram for Grover's algorithm.

To determine the number of times the diffusion operation should be applied, we need Geometrically, the iteration performs a rotation of the state vector $|\Psi\rangle$ towards the vector representing a uniform superposition of the solution states on the plane spanned by the

$$|\alpha\rangle \equiv \frac{1}{\sqrt{N-M}} \sum_{\text{solutions}} |x\rangle \quad (47)$$

$$|\beta\rangle \equiv \frac{1}{\sqrt{M}} \sum_{\text{non solutions}} |x\rangle \quad (48)$$

Substituting these to the state vector gives:

$$|\Psi\rangle = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle \quad (49)$$

Limitations of Grover's Algorithm

2.4.2 Shor's Algorithm

Shor's algorithm was published 1994 by Peter Shor and proposes an procedure for factorising integers. It is able to find non-trivial divisors in polynomial time (that means, that there exists polynomial function which is the upper time limit for the calculation), whereas classical algorithms are significantly above polynomial time (though sub-exponential).

This has vast impact on today's cryptography: Many encryption systems (like RSA)^[8] rely on the fact, that it is impossible to factorise integers in a reasonable time. Shor's algorithm showed that it is (theoretically) possible for quantum computers, however the practical difficulties of building a real quantum computer are still preventing Shor's algorithm to undermine modern cryptography.

Shor's algorithm belongs to the Monte-Carlo algorithms which means that it is based on probabilistic calculations. Therefore in some cases it might lead to undesired results.

Fig. 2 shows a schematic circuit diagram for Shor's algorithm.

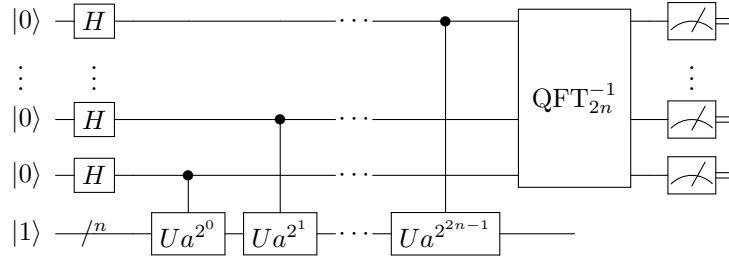


Figure 2: Circuit of Shor's Algorithm^[3]

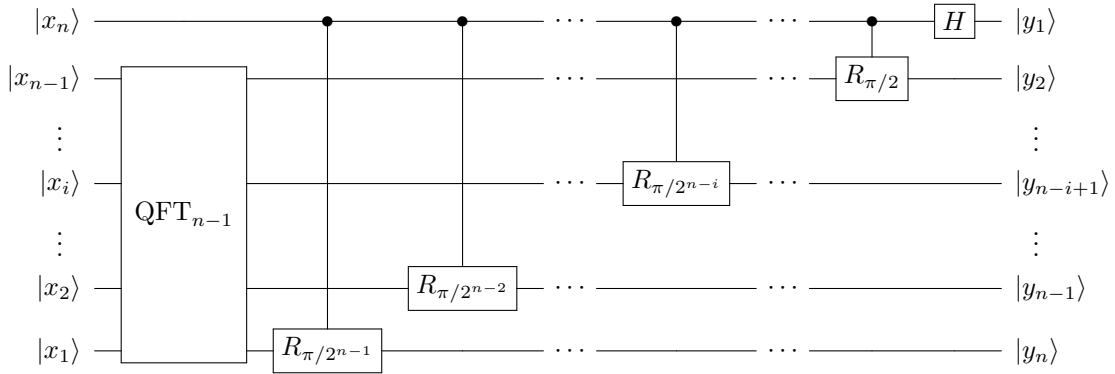


Figure 3: Circuit of Quantum Fourier Transforms^[2]

3 Implementation

In this section we go over the approach used in this programme to tackle the task of simulating the aforementioned concepts on a classical computer. The problem of representing many different concepts, which both built on and used one another, naturally led to a large object-oriented design. Aside from modelling the quantum concepts that ultimately formed a simulation of a basic quantum computer this project also required the creation of the matrix libraries used in the underlying mathematics and a user interface to test and interact with the simulated computer. Thus the implementation of the program can be understood by understanding each of these three major components and how they interact. This section will begin by a discussion of what software development tools were leveraged in order to organise the project between the group members and create the simulator. Following that is a more in depth discussion of how each component of

the program is designed. (For even more detailed documentation of our project see the Javadoc documentation provided with the submission.)

3.1 Project organisation

Prior to implementing the program an overall architecture was formed in order to define the interfaces by which each of the various parts of the program would communicate with one another. A simplified view of this architecture is provided below. The full design includes the interfaces used to communicate between these components, including the method names expected inputs and expected outputs.

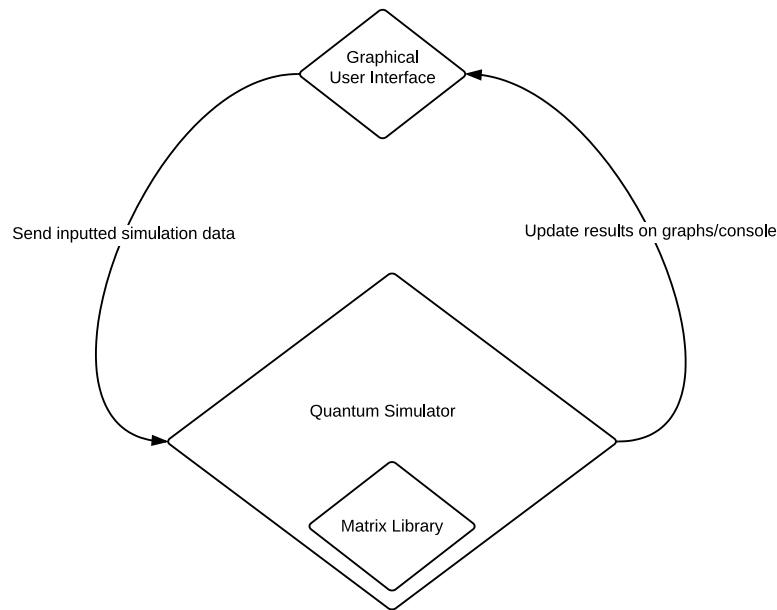


Figure 4

3.1.1 Java

There were many viable programming languages that could be used to implement this simulator. The primary considerations in choosing the appropriate language were the following.

1. The language should readily support large object oriented design implementations
2. The program should be able to efficiently simulate as large of a quantum register as possible
3. The resulting program should run on as many platforms as possible
4. All members of the group should be comfortable with the language

The two primary languages that were ultimately considered were Java and C++. Java was chosen for two primary reasons: it provided an operating system agnostic, well-documented library for Graphical User Interface creation and more of our group members were familiar with the language. It is worth noting that a ‘lower level’ programming language such as C++ would have allowed for more efficient memory management and optimisations but at the cost of increased complexity. However, since we only had limited time to complete our task we decided that we would forgo the speed benefits of C++ for the robustness and ease of Java.

3.1.2 Git

Once an architecture and programming language were decided, the next task was to implement the program as a team. The primary challenge involved in this is allowing each group member to work on the project simultaneously without interfering with each another's progress. The primary tool used in this project to solve this problem is Git source control. Git allows many contributors to simultaneously add changes to a working code base remotely by storing the iterations of the project on both a central server and also on each contributor's local device. In addition it provides many tools to assist in simultaneous development such as previous version recovery, conflict resolution when multiple contributors change the same lines of code in a file, and branch creation (which allows a developer to work in their own version of a project without seeing changes made by other contributors).

3.2 Development environment

Two primary software solutions were used in order to implement our project. The first was Github, which was used to host the Git repository and also functioned as a communication tool. The second was Eclipse and Integrated Development Environment (IDE), which was the primary text editor and compiler used to create and test the program. Eclipse also contained a built in unit-testing environment called JUnit. This was we used in data collection because it allowed testing that bypassed the GUI to speed up how quickly the results were obtained.

3.2.1 Github

As mentioned earlier Github was not only a means of hosting the git repository of our project but also one of the primary means of communication between the group members. The primary feature of the software that was leveraged for this was the issue tracker, which was used as a forum, to-do list, and bug reporting platform.

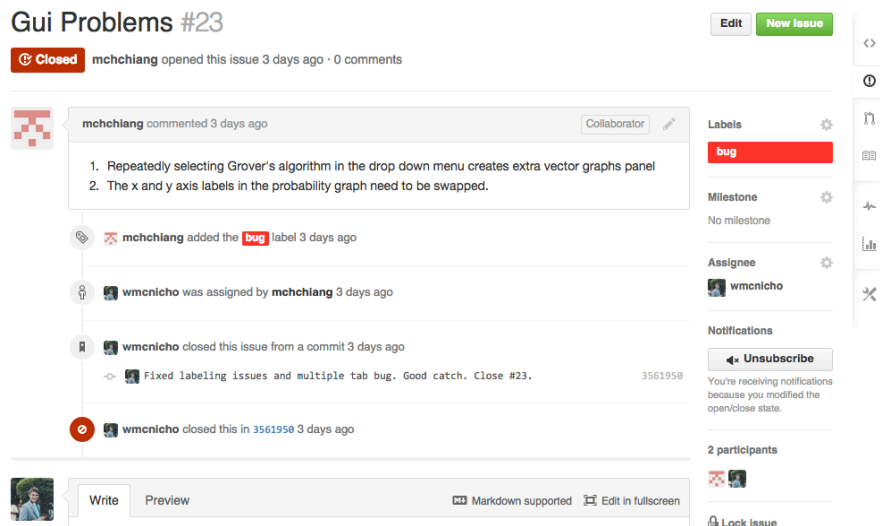


Figure 5

3.3 Programme structure

3.3.1 Overview

As mentioned previously, the architecture for this simulator contains three major components.

1. The Quantum computer
2. The underlying matrix library
3. The Graphical User Interface for testing the computer

This section will present some overall design considerations and then will go in-depth in describing the design and implementations of each of the components.

The overall design of the quantum computer simulator requires careful consideration, as it is the backbone for running various quantum algorithms. An effective, well-structured design allows one to implement any algorithm at ease and compare the performance of the programme under different representations. Our group has identified some key specifications for the design of the computer:

1. It should be consistent with a circuit-model quantum computer.
2. It should allow different representations for each type of object within the computer to be changed easily (i.e. the circuit, gate and the register).
3. It should be extendible and compatible. One should be able to implement another representation for the objects within the computer that would work with the current code.
4. It should maximise code reusability.
5. It should be stable and user friendly.

The first point is fundamental as the project brief specifically asks us to create a simulator of a circuit-model quantum computer. We therefore design the programme based on the physical objects that exist in such computer. This includes the register, which is an array storing n qubits, the quantum gates, and the circuits, which consist of a series of gates that perform more complex operations on the register, such as the quantum search algorithm. These form the three main groups of classes for the computer part of the programme.

To achieve the remaining specifications, we created an interface for each type of object in the program and adhered to common object oriented design patterns. Each interface describes the fundamental behaviours of the corresponding object that are independent of the representation. This allows objects to interact without having to be concerned about each other's representation. It thus separates the task of representing an object and using it to build circuits for implementing the algorithms, which is key for maximising code reusability and making the programme extendible. The use of commonly used design architectures (such as the Factory Pattern and MVVM) allow for other programs to quickly understand the design and usage of the parts of the program.

3.3.2 The Computer

The computer is at the heart of our programme for simulating the quantum algorithms. Figure XX shows the overall structure of this part of the programme. Here we discuss the design choices we made for each of the three components of the computer

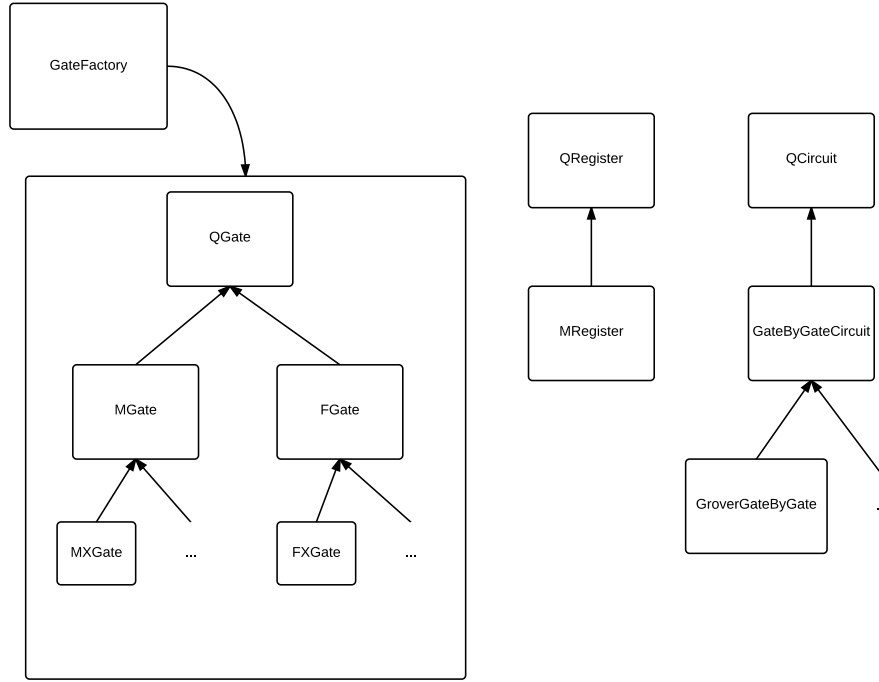


Figure 6

1. Register

The register consists of an array of n qubits that would be manipulated by the gates. From an object-oriented perspective, one might be tempted to create a qubit object with the register storing an array of these objects. This, however, is unphysical as qubits can be entangled and may not be separable as shown in equation XX. Hence, we decided to treat the n qubits as a single system, and the register should store an array of the complex amplitudes of the basis states formed by the qubits (i.e. from $|0\rangle$ to $|N - 1\rangle$).

We developed the `QRegister` interface with the above concept in mind. It requires the classes representing the register to have accessor methods for setting and getting the amplitudes of the basis states. It also requires these classes to provide a method for measuring the register, as such operation is required to obtain physical results after applying a circuit to the register. We created a concrete class called `MRegister` that complies with the interface. It stores the amplitudes of the basis state as a column vector. Since the amplitudes of most of the states would generally not be non-zero, we decided to use a regular dense matrix representation for the vector.

2. Gates

All quantum gates within the circuit implements the `QGate` interface. This interface requires any gate classes to implement the method called `applyGate(QRegister reg)`, as each gate must perform an operation on the register. It also requires the class to provide info about the type of gate it is representing, which would be useful for debugging the programme.

As discussed in section 2, there are many types of single qubit gates, and their associated linear operations on the register can be represented as a matrix or a function. Furthermore, we derived a strategy for creating the representation of a single qubit gate applied to a multiple qubit register that is general for any types of gate, requiring only the knowledge of how the gate operates on a single qubit; therefore, it seems natural to create an abstract

class for the two representations of the gate, which are `MGate` for matrix representation and `FGate` for functional representation.

The `MGate` has a method called `initGate()` which creates the matrix representation of any single qubit gate. It requires . It implements the `applyGate()` method by performing the matrix multiplication of the register's state vector with the gate matrix. The subclasses of `MGate` include: `MXGate`, `MYGate`, `MZGate`, `MPhaseGate`, `MSGate`, and `MTGate`.

Gate	Matrix Rep. Class	Functional Rep. Class
G^H	<code>MHGate</code>	<code>FHGate</code>
G^{mot}	<code>MNOTGate</code>	<code>FNOTGate</code>
G^X	<code>MXGate</code>	<code>FXGate</code>
G^Y	<code>MYGate</code>	<code>FYGate</code>
G^Z	<code>MZGate</code>	<code>FZGate</code>
G^S	<code>MSGate</code>	<code>FSGate</code>
G^T	<code>MTGate</code>	<code>FTGate</code>
G^ϕ	<code>MPhaseGate</code>	<code>FPhaseGate</code>

Table 1: A complete list of gates implemented in the programme.

We know the representation of the gate would depend on the qubit that it is applied to in the register (i.e. the target qubit) and whether there are control qubits associated with gate. This leads to two options for designing the gates:

- Construct a general form of the gate that would require the specification of the target and control qubits when it is applied.
- Construct a specific form of the gate that would only work for a specific target and set of control qubits.

Although the first option keeps the gate class generic and more flexible, it requires the representation to be recalculated every time the gate is used. This may lead to a significant increase in computation time, hindering the programme performance. The latter option does not have such problem, as the gates are instantiated for a specific target qubit and set of control qubits. However, this has the side effect of consuming more memory, as we need to store multiple versions of the same gate for different targets and controls. We decided to design our gates based on the second option, as we believed such usage of memory is tolerable, and we would like to obtain results within a reasonable amount of time. In addition, it simplifies the task for building a quantum circuit. As each gate is pre-programmed to operate on a specific target with a specific set of controls, we would only need to store a list of gates to be applied, sparing the need for a complicated set of data about the target and control qubits for the gates. Furthermore, it highlights the problem of memory-computation time trade-off, which is reoccurring theme in this project.

3. Circuits

A quantum circuit comprises of a series of gates that perform more complicated operations on the register. All circuits classes in the programme implements the `QCircuit` interface. They are required to implement the `applyCircuit(QRegister reg)` method, as they would perform some operations on the register. To allow one to build a general circuit with gates of different representations, we created the `GateByGateCircuit` class.

4. Implementation of the Quantum Algorithms

3.3.3 Matrix

After deciding that we wanted to implement a variety of gate implementations. We decided that for our code to be as user friendly as possible we should be able to switch between our two different matrix representations by only changing a single argument that is passed to the Matrix package. The package should hide the representation from the user and deal with all of the messiness internally.

To achieve this we created an abstract class `Matrix` which is then implemented by all of our concrete matrix representations. The user only ever has to deal with this abstract class. This presented some challenges in implementing this design. The first challenge we encountered was how to call the constructors for the different implementations from a single method call, where one of the arguments decided the type of the Matrix. To solve this problem we employed a simple factory design pattern. Our `MatrixFactory` contains a static method, `MatrixFactory.create(int i, int j, String type)` which returns a matrix of the type specified in the by the string passed in the method.

The `Matrix` class contains methods for all of the matrix manipulations we require in our project. This is mainly involves getting and setting elements and multiplication of matrices. Each of the concrete implementations of `Matrix` override the get and set methods in `Matrix` as they need to be written specifically for the data structure which each implementation used.

The two main types of matrices we implemented were dense and sparse matrices. The dense matrix itself comes in two different types: real and complex. These two share some common features such as the fact that each element is explicitly stored even if it is a zero. This contrasts with the sparse matrix as only non-zero elements and their indices are stored.

We made several choices in our data structures which are related to the performance of the dense matrices. Firstly we decided that the overhead created by 2 dimensional arrays is too much as it results in more memory lookups and more cache misses. This can be reduced by using a single array to store all the matrix entries. It then requires some trivial arithmetic to retrieve the array indices (element read and writes are still $O(1)$ as with multi-dimensional arrays). The cost of calculating these indices is lower than the time wasted on cache misses. The next optimisation we implemented was not storing complex numbers in a complex object as it would again introduce a non-negligible amount of overhead, both in terms of object allocation and a general memory overhead associated with every Java object.

Our sparse matrix implementation contains both a general sparse matrix class as well as a specially optimised class designed for gates. In section 666.666 we saw that each gate can be represented by a matrix containing two elements in each column. This allowed us to design a data structure specifically tailored towards this problem.

The `SparseGateMatrix` class only stores the non-zero entries in the matrix. As we know each column only contains 2 entries, it is enough to store the value and the row index of the element. As long as the entries for each column are stored in order, the true index can be easily recovered with trivial arithmetic in $O(1)$ operations.

The matrix multiplication is dealt with in a separate helper class called `MatrixMultiply`. It contains a very general and naïve implementation of a matrix multiplication algorithm. It then contains further methods which are optimised towards each of our implementations, the most important of these being the multiplication of a complex matrix by a sparse matrix representing a gate. This method is where almost of the computational resources are spent when applying our circuits to the register. It turns out that it is possible to write a more efficient sparse-dense multiplication algorithm if we consider $|\psi\rangle^*$ operator rather than operator $\langle\psi|$. This is because in this case we know that there are only 2 elements in each row of the sparse matrix which

are very quick to access as they will be located next to each other in the system memory. It is then only required to find the 2 corresponding elements in the register to be able to calculate the corresponding element in the answer. While this approach incurs a small overhead due to having to convert from $\langle\psi|$ to $|\psi\rangle$ and back again at the end, this can be minimised very easily and the final code runs faster than if this approach was not used.

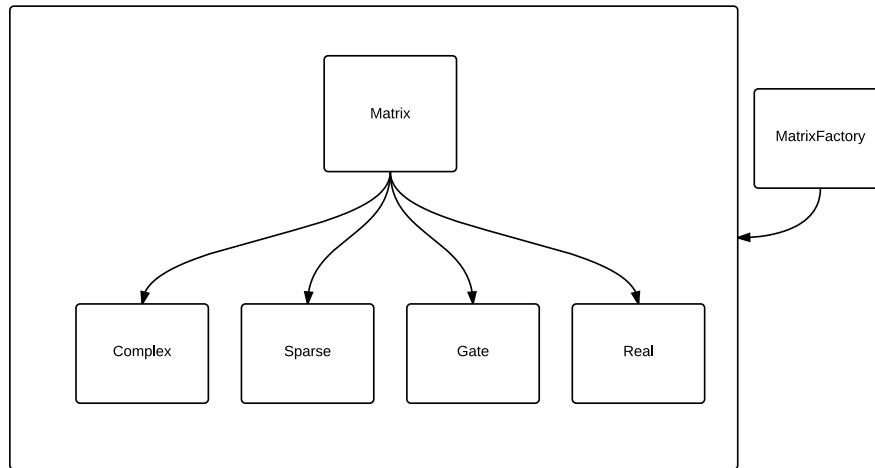


Figure 7

3.3.4 Graphics User Interface

The Graphics User Interface (GUI) for this program is constructed through the use of the Java swing library and one external, open source library for graphing called JFreeChart. In designing the GUI there were a couple of goals in mind.

1. Make it easy to use
2. Have it display as much of the potential of the simulator as possible
3. Ensure it's usage does not significantly slow down the simulation process

To accomplish the first goal the entire interface is limited to a small selection of buttons and drop down menus that requires the user to do very little to start a simulation. The GUI also dynamically changes itself in an attempt to instruct the end-user on how to use the program. The program also has a built in help window with instructions on how to perform the various simulations.

To accomplish the second and third goals, a variation of the Model-View-ViewModel interface was implemented. In this architecture, the Model (i.e. the quantum simulator) would communicate relevant data changes to the ViewModel (QViewModel.java) through static methods. The ViewModel would then perform any necessary conversion or reinterpretation of the data and then would instruct the view (i.e. the animations and console window) to update what it is displaying. This accomplished the goal of not slowing down the program because the simulation code does not need to wait for the GUI to render to continue it's simulation. It achieves the second goal because it provides a simple interface for the people working on the model code to use in order to make changes to the GUI and facilitates communication between the View and Model programmers.

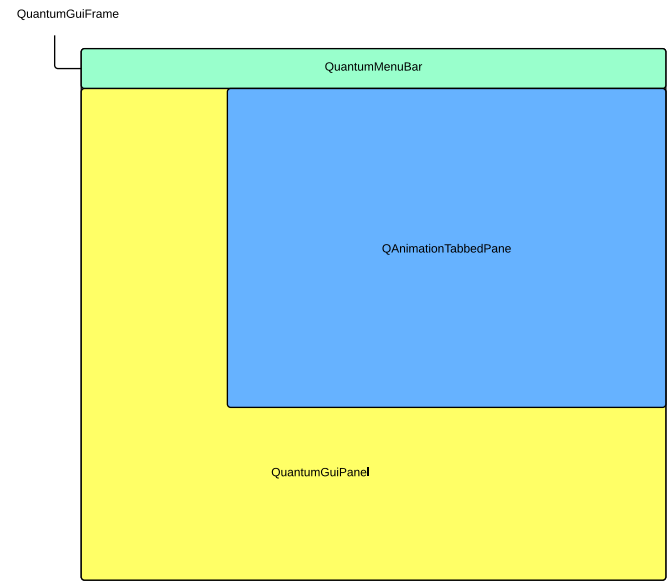


Figure 8

The class `QuantumGuiFrame` contains all of the other created classes that ultimately form the overall GUI. These include `QuantumMenuBar` and `QuantumGuiPanel` (which contains a `QAnimationTabbedPane`). This structure is diagrammed below.

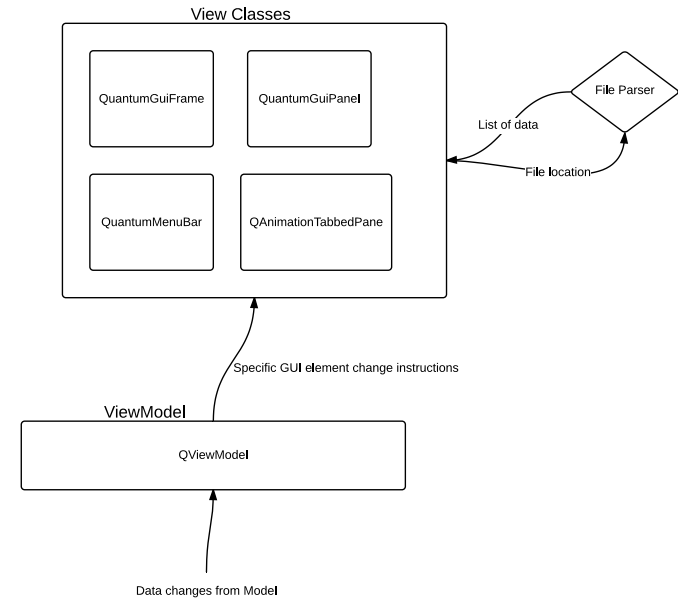


Figure 9

3.4 Programme execution

The execution flow of the program is a two threaded system but could be expanded to many more threads in the event this program is extended to have multi-core or GPU support. When the

program is started the main function in `QuantumGuiFrame` creates the GUI and waits for user input. Once the user has selected the type of simulation and the associated data they will then press the start button which will create a `QProcess` object. This `QProcess` class will launch a new thread in its constructor that will create a quantum circuit based off of the specifications that the user provided and run that circuit on the target data. The GUI thread and the simulation thread will then run in parallel until the simulation is complete. As the simulation runs it updates the GUI informing the user of the progress through various animation elements and a console window.

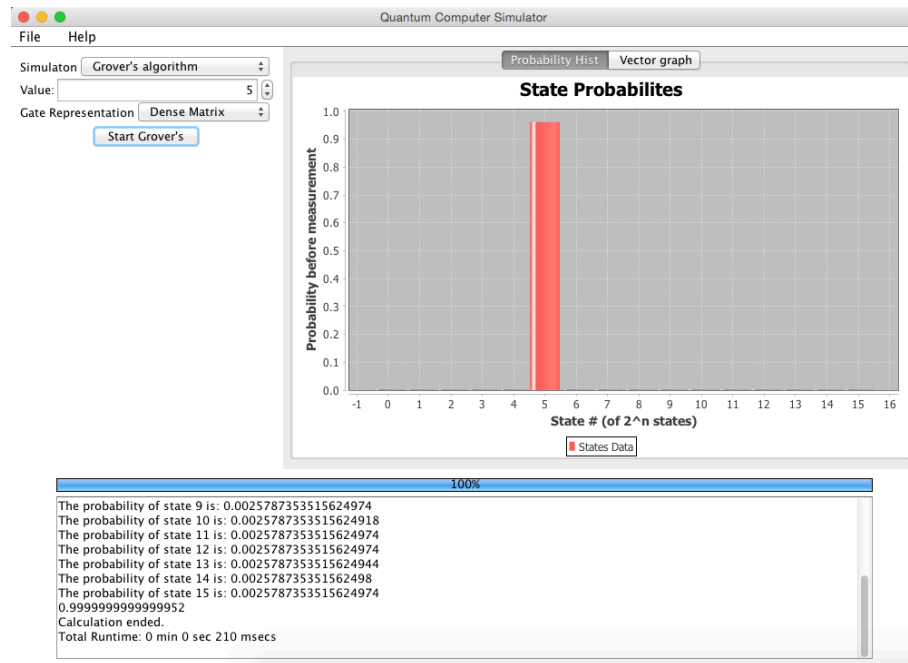


Figure 10

4 Results

4.1 Grover's Algorithm

4.1.1 Rotation of the state vector

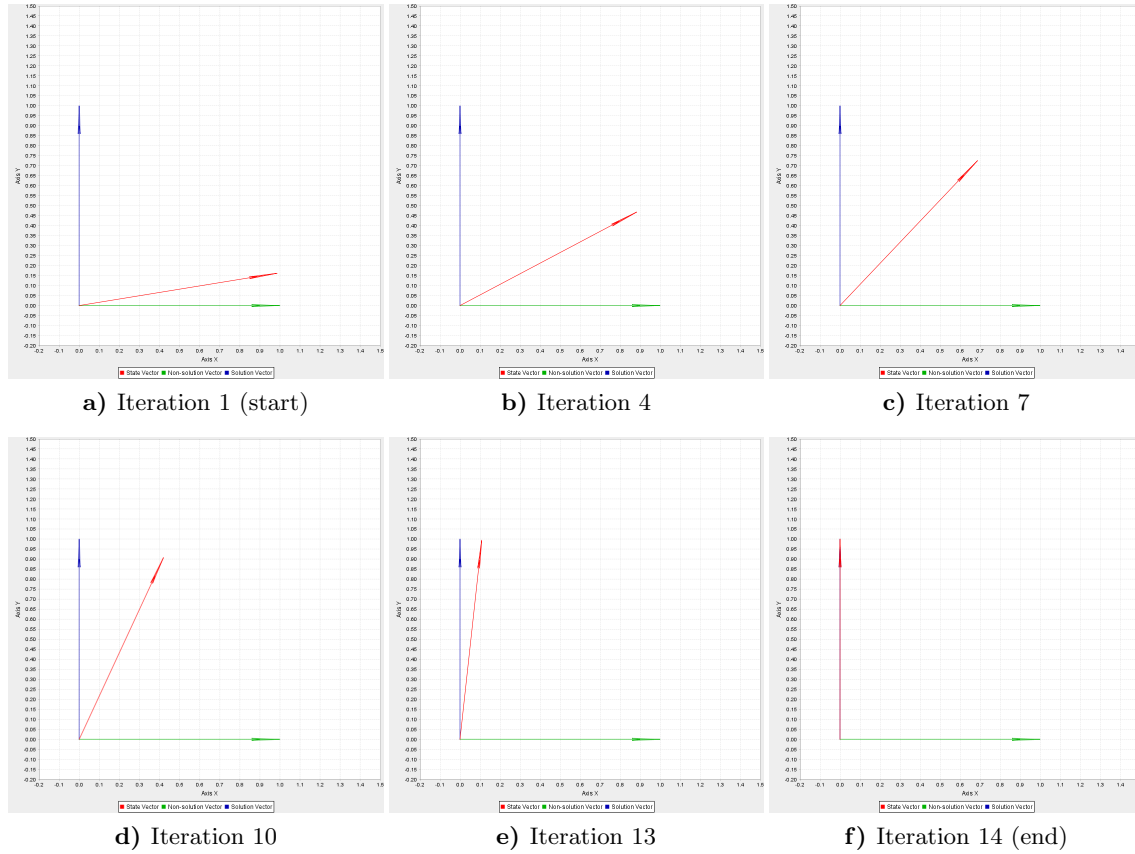


Figure 11: Depiction of Grover's Algorithm: The green vector depicts the non-solution vector, the blue vector depicts the vector of the solution. The red vector is the projection of the current state. With every consecutive iteration the state vector rotates towards the solution state.

4.1.2 Computational time

Fig. 12 shows the computational time of Grover's algorithm. All the measurements were executed on the same computer by varying the number of qubits n for the three different representations.

We note the following observations:

- **Dense representation at a disadvantage**

One can see that the dense representation is the slowest one since for already $n \geq 10$ first noticeable computation times occur whereas the algorithm using functional and spare representation is still immediately executed. However, when they start to show noticeable computational times, the dense representation already takes a unbearable amount of time.

- **Spare vs. functional representation**

Considering computational time the sparse representation is superior to the functional rep-

resentation. It appears to be roughly a factor 2, but since we don't have data for bigger registers, this quantitative observation is quite vague.

- **Exponential behaviour**

All three representations show an exponential behaviour. To illustrate the point we plotted the same data in a log-plot (Fig. 13) and got a linear relation.

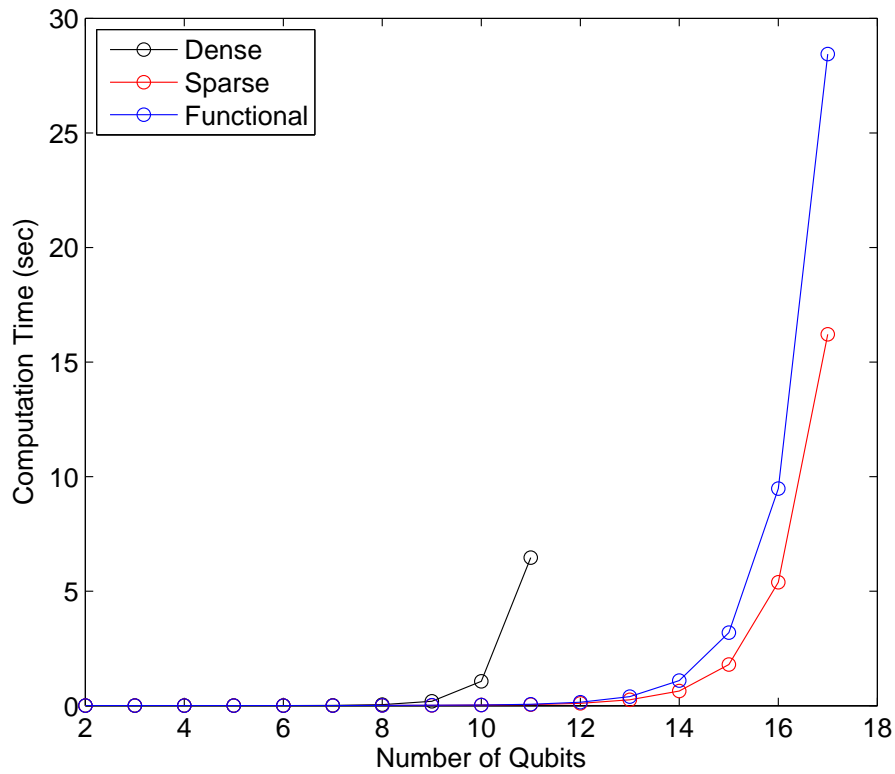


Figure 12: Computational time of Grover's Algorithm in nanoseconds as a function of the quantum register size for the three different forms of representation (dense, sparse, functional).

4.2 Shor's Algorithm

5 Discussion

5.1 Matrix or functional representation

5.2 Improvements and further steps

6 Conclusion

7 Appendix

7.1 Grover's Algorithm

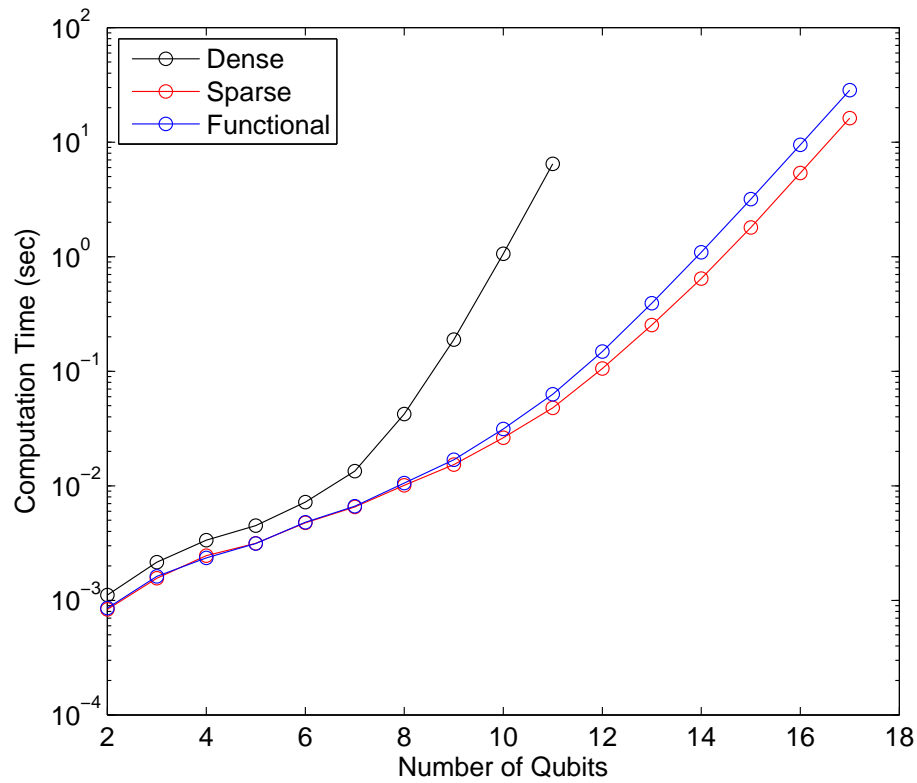


Figure 13: Computational time (log-scale) of Grover's Algorithm in nanoseconds as a function of the quantum register size for the three different forms of representation (dense, sparse, functional). Note that only the behaviour for bigger registers is meaningful (linear).

7.2 Details

The data for the computational time (Fig. 12 and Fig. 13) was collected on a:

Intel i5 3570@3.4 GHz 2x4 GB 1600 Mhz Ram Windows 7

References

- [1] *Grover's algorithm*. http://commons.wikimedia.org/wiki/File%3AGrovers_algorithm.svg, . – [Online; last accessed on 06.03.2015]
- [2] *Quantum Fourier transform*. http://commons.wikimedia.org/wiki/File:Quantum_Fourier_transform_on_n_qubits.svg, . – [Online; last accessed on 19.03.2015]
- [3] *Shor's algorithm*. http://commons.wikimedia.org/wiki/File:Shor%27s_algorithm.svg, . – [Online; last accessed on 06.03.2015]
- [4] EKERT, Artur ; HAYDEN, Patrick ; INAMORI, Hitoshi: *Basic concepts in quantum computation*, 2008
- [5] NIELSEN, Michael A. ; CHUANG, Isaac L.: *Quantum Computation and Quantum Information - 10th Anniversary Edition*. Cambridge : Cambridge University Press, 2010. – ISBN 978–1–139–49548–6
- [6] PERRY, Riley T.: *Quantum Computing from the Ground Up*. Singapur : World Scientific, 2012. – ISBN 978–9–814–41211–7
- [7] VANDERSYPEN, Lieven M. K.: *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*. – [Online; last accessed on 13.03.2015]
- [8] WIKIPEDIA: *RSA cryptosystem*. http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29#Attacks_against_plain_RSA, . – [Online; last accessed on 17.03.2015]