

Quantum Computing and Many-Body Physics

Eirik Ovrum



Thesis submitted for the degree of
Candidatus Scientiarum

Department of Physics
University of Oslo

May 2003

Acknowledgments

First I would like to thank Morten Hjort-Jensen and Jon-Magne Leinaas for accepting me as their student, bewildered and uncertain as I was, and for their excellent tutoring that has helped me to a better understanding of physics.

The theory group in Oslo is a great place to be a student, thanks to all the other students, the work on this thesis would have been much harder if not for their forthcoming help. (Especially Morad, who should have had a paid position).

I have to thank my family, I am grateful that they have taught me to be curious and to trust my own judgment.

Finally I thank Thùy Mi for her support and the confidence she has given me.

Contents

1	Introduction	1
1.1	Why quantum computers?	1
1.2	Is it possible to make quantum computers?	3
1.3	Objectives	6
2	Background	9
2.1	Quantum Mechanics	9
2.1.1	The Postulates of Quantum Mechanics	9
2.1.2	Dirac's bra/ket notation	11
2.1.3	Hilbert space	11
2.1.4	Linearity	12
2.1.5	Linear independence and bases spanning a space	12
2.1.6	Dual states and complex conjugates	12
2.1.7	Inner and outer product	13
2.1.8	Orthonormality	13
2.1.9	Completeness	13
2.1.10	Eigenvalues and eigenvectors	14
2.1.11	Observables and operators	14
2.1.12	Collapse of the state	14
2.1.13	Time evolution	15
2.2	Qubits	15
2.2.1	Qubits and matrices, the computational basis	16
2.2.2	Operators as matrices	18
2.2.3	Applying operators to vectors/states	19
2.2.4	Product states and entanglement	20
2.2.5	Example of a 4×4 operation	21
2.3	Quantum gates and quantum circuits	21
2.3.1	A quantum circuit	22
2.3.2	Swapping and conditional operations	23
2.4	Gates	25

3	Quantum computation	27
3.1	The simulation algorithm	27
3.1.1	Simulating a quantum system	27
3.2	The Universality of quantum computers	29
3.2.1	Two-level unitary gates are universal	30
3.2.2	<i>CNOT</i> and single-qubit gate universality	32
3.3	Fourier transform	37
3.3.1	Quantum Fourier transform	37
3.3.2	Implementation of Fourier transform	41
3.3.3	Inverse Fourier transform	43
3.4	Phase estimation	45
3.4.1	The algorithm	46
3.5	Measurements on the system	50
3.5.1	The projection	50
3.5.2	Measuring a single qubit	50
3.5.3	Measurement in phase estimation	51
3.5.4	Simulating measurement	51
3.5.5	Finding E_k	53
3.5.6	Phase estimation and negative ϕ	53
3.5.7	Information required for simulating	54
3.5.8	Analyzing parts of the energy spectrum	54
3.5.9	Initializing qubits	55
3.5.10	The circuit	55
4	The Model	57
4.1	The Heisenberg model	57
4.1.1	Time evolution	58
4.2	Operators for two level unitary matrices	59
4.2.1	σ_z	59
4.2.2	$\sigma_k^i \sigma_{k+1}^i$	60
4.2.3	$\sigma_k^x \sigma_{k+1}^x$	60
4.2.4	$\sigma_k^y \sigma_{k+1}^y$	61
4.2.5	$\sigma_k^z \sigma_{k+1}^z$	62
4.3	Two-level unitary matrices to quantum gates	62
4.3.1	$\exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$	64
4.3.2	$\exp(-i\sigma_k^y \sigma_{k+1}^y \Delta t)$	66
4.3.3	$\exp(-i\sigma_k^z \sigma_{k+1}^z \Delta t)$	66
4.3.4	$\exp(iE_{max} \Delta t)$	67
4.4	The final circuit	68

5	The simulation	69
5.1	Algorithms	69
5.1.1	Operations, gates and matrices	69
5.1.2	Single-qubit gate	70
5.1.3	Controlled operations	70
5.1.4	CU	71
5.1.5	XCU	72
5.1.6	UC	73
5.1.7	XUC	74
5.1.8	Conditional operations on non-neighboring qubits . . .	74
5.2	Eigenvalues by diagonalization	75
5.3	Extracting the eigenvalues	76
5.3.1	Finding the maxima	77
5.4	Minimizing errors	79
5.4.1	The exponential function approximation	80
5.4.2	Work qubits vs. simulation qubits	84
5.4.3	Analysis errors	87
5.4.4	Conclusion on errors	88
5.5	Computational complexity	88
5.5.1	Complexity	88
5.5.2	Classical complexity	89
5.5.3	Quantum computer complexity	90
5.5.4	Polynomial in N	92
5.5.5	Time	92
6	Conclusion	95
6.1	What we have done	95
6.2	A new computational algorithm	95
6.3	The future of quantum computers	96
6.3.1	Research today	97
6.4	My thoughts on the future	98
A	Tensor Products	101
A.1	Tensor product or outer product	101
A.1.1	Matrix tensor products	102
B	The Program	103
B.1	Programming	103
B.1.1	C++ and Blitz++	103
B.2	Bruteforce diagonalization	105
B.2.1	Tensor product code	105

B.2.2	Diagonalization	106
B.3	Simulation	106
B.3.1	single-qubit gate	106
B.4	The measurement	108
B.5	Flow chart of the program	109
B.5.1	Outline of the program	110
C	Program listing	111
C.1	Simulation class	111
C.1.1	Header file	111
C.1.2	CPP file	114
C.2	Gate class	138
C.2.1	Header file	138
C.2.2	CPP file	140
C.3	State class	150
C.3.1	Header file	150
C.3.2	CPP file	151

List of Figures

1.1	ENIAC computer	5
1.2	Articles with Quantum Computer in title	7
2.1	<i>NAND</i> gate with truth table	22
2.2	A quantum circuit	22
2.3	Qubit ordering	23
2.4	<i>CNOT</i> gate	24
3.1	Circuit swapping the states of two qubits	42
3.2	The R_k gate	43
3.3	Fourier transform circuit without swap gates	44
3.4	Inverse Fourier transform circuit without swap gates	44
3.5	The conditional $U^{2^{t-1}}$ gate	48
3.6	Multiple qubit conditional operation	48
3.7	Phase estimation circuit	49
3.8	Simulating measurement	53
3.9	Simulation circuit	56
4.1	V_1 circuit	63
4.2	V_2 circuit	64
4.3	First circuit for “ XX ” operator	65
4.4	<i>XCNOT</i> twice is unity	65
4.5	Final circuit for “ XX ” operator	66
4.6	Circuit for the “ YY ” operator	67
4.7	Circuit for the “ ZZ ” operator	67
4.8	Gate ensuring negative eigenvalues	68
4.9	Final circuit for one term of the Hamiltonian	68
5.1	Control U gate	71
5.2	<i>XCU</i> gate as combination of X and CU	72
5.3	<i>XCU</i> gate	73
5.4	UC gate	74

5.5	XUC gate	74
5.6	The result of the program. Parameters $N = 9$, $t = 6$, $s = 3$. .	77
5.7	The probability spectrum as a function of the energy	79
5.8	Energy spectrum, $\mathcal{O}(\Delta t^2)$ approximation, $n = 1$	81
5.9	Energy spectrum, $\mathcal{O}(\Delta t^2)$ approximation, $n = 100$	82
5.10	Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 1$	82
5.11	Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 30$	83
5.12	Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 10$	83
5.13	Energy spectrum, parameters $N = 4$, $t = 2$, $s = 2$	84
5.14	Logarithm of time vs number of qubits for $s = 2$	90

List of Tables

3.1	Table of two-qubit conditional operations	35
5.1	Exact eigenvalues with corresponding degeneracy	76
5.2	Eigenvectors and eigenvalues for $s = 2$	76
5.3	The results of the analysis	78
5.4	Parameters	79
5.5	Calculated energy levels	80
5.6	Measured eigenvalues for $n = 100$	81
5.7	Energies for $n = 30$	82
5.8	Energies for $n = 10$	83
5.9	Systematic error due to binary number approximation	85
5.10	Energy spectrum, $\phi_k 2^t = 96$ for $E_k = -6$	86
5.11	Averaged over three different $n\Delta t$'s	86
5.12	Varying Δt	87
5.13	Time as function of N and s	90
5.14	Number of gates	92
5.15	Decoherence time and operation time	93

Chapter 1

Introduction

In this first part of the thesis we seek to explain the What? Why? How? and When? of quantum computers.

1.1 Why quantum computers?

To answer this we need to answer two other questions first. What is a computer? and what does quantum mean?

What is a computer?

A computer, as the name suggests, is a machine that performs calculations. A pocket calculator can compute, but only certain things, and is not a universal computer. What we call computers, PCs, macs, solaris machines etc. are universal computers. They can solve all problems that can be solved by calculations, provided their memory is large enough, and the user has the required time and patience.

The basic unit of a computer is the bit, which can be zero or one, true or false. Strings of bits can represent any integer. These integers can be used to represent any quantity and used together with logical operations that change their value in specific ways, can be used to perform calculations. The necessary components of a computer are a few base logical operations, which combined can perform any computation.

Why quantum?

As the size of physical systems become sufficiently small, some physical quantities become discrete, quantized. The energy levels of the hydrogen atom is

the best known example. To describe these systems quantum mechanics was developed, and it has proven to be an excellent description of nature.

The bit can be either zero or one, it can only have two different values. There are many quantum mechanical systems that have only two possible states, the spin of an electron can only be measured to $1/2$ or $-1/2$, a photon can have only two distinct polarizations, and many others. In quantum computers the basic unit is the qubit, the quantum bit. It is a system that has only two states, called zero and one, but unlike the bit it can also be any combination of these two states, a quality that make quantum computers quite different from the ones we are used to.

To perform operations on qubits quantum computers use quantum operations, in analogy with classical computers these operations are called gates. What the operations actually do, depend on what is used as qubits. If, for example, the spin of electrons were used as qubits, an operation could be to rotate the electron in a specific way by applying a magnetic field.

What is the difference between classical and quantum physics?

The introduction of quantum physics early in the twentieth century resulted in a change of paradigms, and therefore the “old” physics is known as “classical” physics. Classical physics, started by Galileo Galilei and given a foundation by among others Isaac Newton, is a system where you can exactly calculate all movement for all time, as long as you know the initial conditions and the system. In other words it is objective deterministic.

The quantum hypothesis was first introduced by Max Planck to explain the black body radiation spectrum, and was pioneered by among others Niels Bohr, Erwin Schrödinger, Werner Heisenberg and Paul Dirac. Niels Bohr especially stressed the new idea that observation affected a system, the idea that scientists could no longer consider themselves apart from the world of experiment, not affecting their outcome. Quantum mechanics is subjective, in that observation depends on the observer and the system itself is changed by observing.

The exact calculations of classical physics are not possible in quantum mechanics, instead the probability distribution of different measurements can be calculated exactly for all time. You cannot know what a given measurement will be, but you know the probability of the different results for all time. Quantum mechanics is a subjective, probabilistic and deterministic theory.

Why would we want quantum computers?

All new realms of science deserve attention and scrutiny, and their study is its own reward. Yet there are far less abstract rewards for creating quantum computers.

The reason quantum computers are interesting is that they are better at solving some types of problems than classical computers. If quantum computers were not an improvement, there would be no point in seeking to make them other than to satisfy our curiosity.

There are two main improvements of quantum computers versus classical ones. The first is that some problems can be solved faster, sometimes exponentially faster, on quantum computers, and this has been the most celebrated aspect of quantum computers this far. The two most famous quantum algorithms are Grover's search algorithm [9] and Shor's algorithm for factorizing primes [19]. Using Grover's search algorithm a quantum computer could find a number in a phone book with N entries using averagely \sqrt{N} queries, while a classical computer would use averagely $N/2$ queries. An even more hailed algorithm is Shor's algorithm for finding the prime factors of an integer. The problem of finding the prime factors is a difficult task on classical computers, with an almost exponential time increase as the size of the number to factorize increases. Finding the number once you have the factors is easy, however, it requires only multiplication. A quantum computer using Shor's algorithm would have as output an answer with a probability of being the right one. Checking it does not take long, and averagely Shor's algorithm is much faster than the classical one. There is also hope that they will be able to compute a non-deterministically polynomial complete problem in polynomial time, see section 6.3.1, something that would be a huge breakthrough.

The other reason quantum computers are interesting is that they can simulate quantum systems much more effectively than classical computers. Simulations could be done on relatively small quantum computers (100 qubits) that would require computers with memory capacity thousands of times greater than what is available today, and take decades to perform with the present super-computer speed.

1.2 Is it possible to make quantum computers?

What will the future bring? Will we all someday have quantum laptops?

The theoretical foundation of quantum computers has been laid, remaining is the technical part. Several schemes for constructing a quantum

computer have been proposed, and finding a system that fulfills the requirements of a quantum computer is subject of intense study today. Quantum mechanics in solid states (materials) is promising as one might hope to one day have integrated quantum circuits that can be mass produced. It is a field in evolution and discoveries are made almost daily and no one can see what will be possible tomorrow. It is said that the nineteenth century was the century of the machine, the twentieth of electronics and that the twenty first will be the century of applied quantum mechanics.

As it was at first with Alan Turing's "Turing machine" (the prototype theoretical computer), the problem is implementing the theory. Many people say the problems are insurmountable, and that it is a waste of time. Perhaps it is, we cannot know. What we do know, however, is that we do not know that it is a waste of time.

Experimentalists have already been able to construct a small quantum computer that has successfully factorized the number 15, see reference [20], into five and three. It may not seem as much, and one can wonder if quantum computers large enough to compute anything of importance can be made. There are several obstacles to be overcome, decoherence time versus ability to manipulate for one (see section 5.5.5), but it will do to take a look at history.

Lessons from history

Before we make a guess at what will come, we must study the past. Looking at the history of classical computers might shed some light on the progress of science, and tells us that things sometimes move faster than we think. The inability to forecast the future is seen in these statements on the future of computers.

I think there is a world market for maybe five computers
Thomas Watson, chairman of IBM, 1943

Computers in the future may weigh no more than 1.5 tons.
Popular Mechanics, forecasting the relentless march of science,
1949

In 1946 the ENIAC (Electronic Numerical Integrator And Computer) (see fig. 1.1) the first large-scale general-purpose electronic computer, weighed 30 tons and occupied a space of 150 square meters. This computer could add 5000 numbers per second, corresponding to a couple of thousand floating point operations per second (flops). Today there are computers which can perform trillions of flops.



Figure 1.1: ENIAC computer

In 1947 the first transistor was made, and it revolutionized the computer industry, dispelling the need for behemoths like ENIAC. In 1965 Gordon Moore [15] predicted that computer power would double at constant cost about every two years. This has held true to today with amazing precision, and is believed to hold true for at least ten more years. The exponential growth of computer power is due to the constant miniaturization of transistors, the number of components per area that yields the minimum costs has increased exponentially since before 1965. There are limits however. Within few years two effects will set a lower boundary for the size of transistors, temperature and quantum effects. The high temperature of transistors is today a major problem for computers, the smaller a circuit gets the more effective cooling devices it needs. There will be a boundary where it will be difficult to create smaller transistors that do not instantly burn when used. Transistors are quite small today, and some time in the future quantum mechanical effects will come into play and seriously disrupt the classical algorithms run on the transistors.

The impact of electronic computers on society is just beginning to be felt and we are still in the early stages of the computer age. The rich part of the world twenty years in the future will be a very different place. Computers and computer run technology will change everyday life of all people who can afford them in a more fundamental way than we think today. Computers are part of our lives, but we have all lived without them and without the high-tech gadgets coming more and more into use, and we can see what life must have been like before. Two generations from now many children will grow up never knowing anything but urban technological life. Society will

change fundamentally.

The history of quantum computation started in 1980 when Paul Benioff released a paper, [3], where he outlined a quantum mechanical Turing machine, a quantum computer. Two years later Richard P. Feynman wrote an article [8], where he deliberated on the fact that some quantum mechanical systems might be very difficult or impossible to simulate on classical computers due to time and size limitations, and he suggested that quantum computers could be used to simulate quantum systems effectively. The last idea is what we have almost put into life in this thesis, showing a quantum computational many-body algorithm that grows exponentially with the size of the problem on classical computers, but only quadratically on a quantum computer.

The field of quantum computations and quantum computers has since its humble beginnings exploded. Hundreds of articles have been written and many new advances have been made. Quantum algorithms that outclass classical ones have been found, several models for physical quantum computers have been suggested, experimentalists have performed computations on small systems and qubits have become a fashion word even for non-scientists. Many today await the arrival of quantum computers with great expectations, and science fiction writers today will probably reflect this by including quantum computers in their visions of the future, the way even cars were driven by atomic energy in 1950's science fiction.

To illustrate the amount of research being done in this field (in an unscientific way) we found 136 articles from 2002 at the Los Alamos preprint archive at www.arxiv.org, when we searched all categories for titles containing the words quantum computer. In the years 1991 through 2003 (april 25, 2003) we found 686 articles. The distribution per year is shown in fig. 1.2. This testifies to a field of great activity.

1.3 Objectives

Three main issues are addressed in this thesis.

First, we provide an introduction to quantum computers for physicists, post-docs and graduate students alike.

Second, we present the quantum computer simulation algorithm and use it to solve the many-body problem of the one dimensional Heisenberg model of a chain of spin $1/2$ particles.

Third, we simulate a quantum computer finding the energies of the Heisenberg system on a classical computer.

This thesis is divided into six chapters. After these opening remarks we

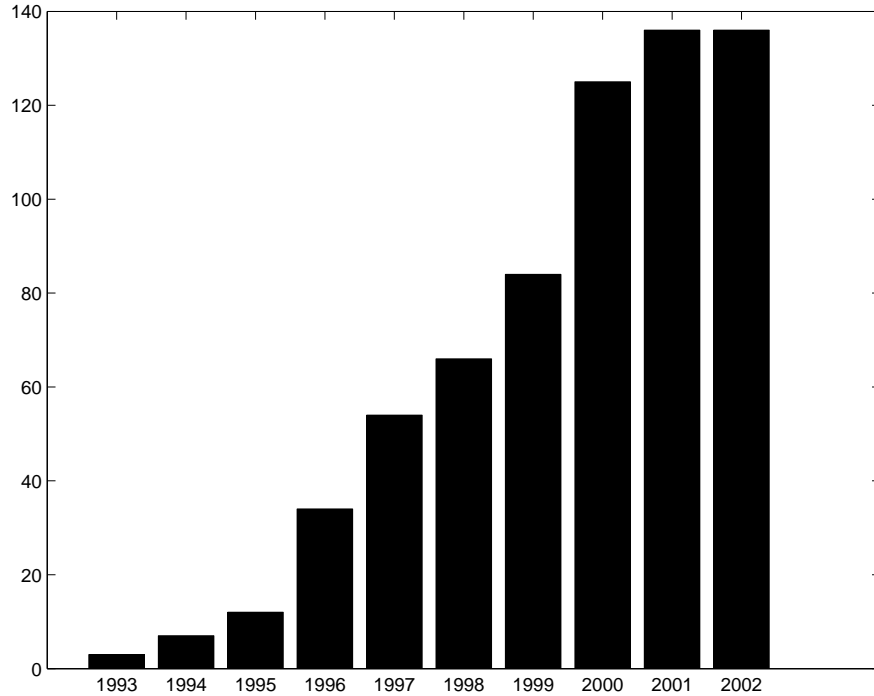


Figure 1.2: Articles with Quantum Computer in title

present in chapter two the physics and nomenclature of quantum computers. We will go through the quantum mechanical principles guiding quantum computers, and explain the computational basis, which we use throughout this thesis.

The third chapter details how quantum computers can work, showing that certain sets of basic operations are universal, that they can simulate all processes that can be done on ordinary computers, and as such are the theoretical equivalents of classical electronic computers. We will go through the recipe for simulating quantum systems on quantum computers and show the most important algorithms and gates used.

In the fourth chapter we show how to simulate the one dimensional Heisenberg model. This is perhaps the most important part of the thesis.

In the fifth chapter we explain the computer simulation done and discuss the results achieved.

In the last chapter we present our conclusions and future perspectives.

Chapter 2

Background

2.1 Quantum Mechanics

In this part we will go through the postulates of quantum mechanics, some necessary linear algebra, the qubit and the computational basis. Leading to the representation of states and operators as abstract bra/ket entities, matrices or lines and boxes in a quantum computer circuit diagram.

2.1.1 The Postulates of Quantum Mechanics

Quantum mechanics evolved from the experimental situation at the start of the twentieth century. The enigmas of the time led to the revolutionary assumptions that lay the foundations of modern physics. Planck hypothesized that light was quantized, not merely waves, to explain black body radiation. Inspired by that idea, de Broglie conjectured that particles should exhibit wave-like behavior. In an experiment electrons were sent through a double slit onto a plate that recorded their position. The electrons produced an interference pattern similar to waves passing through two slits. When the experimentalists measured which slit the individual electrons passed through however, the interference pattern disappeared. This experiment showed the fundamental particle-wave duality of nature and the Heisenberg uncertainty relation.

The postulates of quantum mechanics were formulated in the years that followed, describing in a mathematical language these properties of nature. Quantum mechanics can only tell how the double slit experiment happens, not explain why. It is an intrinsic part of nature that we do not understand, but can calculate using quantum mechanics.

In the extent that the postulates hold true, the rest of the theory holds true as well. First I will list the mathematical postulates, then explain what

they mean for quantum mechanics and in particular for quantum computers.

Postulate 1 *Associated to any physical system is a unit vector in a Hilbert space. This state vector $|\psi, t\rangle$, completely describes the system, and it obeys the time dependent Schrödinger equation*

$$i\hbar \frac{\partial}{\partial t} |\psi, t\rangle = \hat{\mathbf{H}} |\psi, t\rangle \quad (2.1)$$

where $\hat{\mathbf{H}}$ is the Hamiltonian operator. For a single particle in a potential $V(\vec{r})$, the Hamiltonian is $-\nabla^2/2m + V(\vec{r})$ in position space.

Postulate 2 *To any physical observable F there is associated a linear hermitian operator $\hat{\mathbf{F}}$ in the Hilbert space describing the physical system.*

The operator associated with a generalized coordinate and the operator associated with it's corresponding momentum are subject to this commutation relation

$$[\hat{\mathbf{x}}, \hat{\mathbf{p}}] = i\hbar.$$

As an example, the classical quantity $F(x, p)$ becomes an operator depending on the operators $\hat{\mathbf{x}}$ and $\hat{\mathbf{p}}$, $\hat{\mathbf{F}}(\hat{\mathbf{x}}, \hat{\mathbf{p}})$.

Postulate 3 *The only possible results of a measurement of F are the eigenvalues f_n of the operator $\hat{\mathbf{F}}$,*

$$\hat{\mathbf{F}} |\phi_n\rangle = f_n |\phi_n\rangle.$$

Postulate 4 *The expectation value of an observable F in a system in the state $|\psi\rangle$ is*

$$\langle F \rangle = \langle \psi | \hat{\mathbf{F}} | \psi \rangle. \quad (2.2)$$

Since the Schrödinger equation is linear a superposition of solutions is also a solution, and therefore all states can be represented by a superposition of eigenstates of $\hat{\mathbf{F}}$,

$$|\psi\rangle = \sum_i |\phi_i\rangle = |\phi_1\rangle + |\phi_2\rangle + \dots$$

If $|\psi\rangle = |\phi_n\rangle$ then the expectation value is the eigenvalue $\langle \phi_n | \hat{\mathbf{F}} | \phi_n \rangle = f_n$.

2.1.2 Dirac's bra/ket notation

Dirac introduced the bra/ket notation as a way to distinguish between vectors and their dual vectors, $|a\rangle$ is a vector and $\langle a|$ is its dual vector. The first is a ket state and the second is a bra, as in bracket: $\langle \rangle$. In matrix terms a bra state is the hermitian conjugate of its corresponding ket state. The Hermitian conjugate of a matrix is its transposed with every element complex conjugated. We represent a ket state as a $1 \times N$ matrix, and therefore its hermitian conjugated bra state is represented by an $N \times 1$ matrix.

2.1.3 Hilbert space

A Hilbert space is a complex linear vector space with a defined inner product and all vectors normalized to unity.

A linear vector space is a set of vectors, $|x\rangle$, which obeys these rules

- Commutativity: $|a\rangle + |b\rangle = |b\rangle + |a\rangle$,
- associativity: $|a\rangle + (|b\rangle + |c\rangle) = (|a\rangle + |b\rangle) + |c\rangle$,
- there exists a null vector: $|b\rangle + |a\rangle = 0 + |a\rangle = |a\rangle$,
- there exists an inverse of all vectors: $|a\rangle + |b\rangle = |a\rangle + (-|a\rangle) = 0$,
- the vectors can be multiplied by scalars and added: $\alpha|a\rangle + \beta|b\rangle$.

A vector space under the above definitions has an infinity of members, and we can describe a vector as a set of numbers under a given mapping or coordinate system. When we use complex numbers to describe the coordinates of the vectors, we have a complex vector space. Normalized to unity means that the sum of the absolute values of the coordinates is one.

The first postulate states that every quantum mechanical system can be described fully by a vector in the Hilbert space of that system. It also defines the Schrödinger equation, the most important equation in physics. Its importance is both of a calculational and a philosophical character; it states how the system will evolve for all time. As we briefly discussed in the introduction, section 1.1, Newtonian mechanics is objective deterministic. The observer does not partake in the system and if we know all initial conditions we can calculate all positions and momenta of a system for all time. Quantum mechanics, however, is subjective probabilistic deterministic. Subjective because measurements affect the system. Probabilistic because we cannot always know what we will measure, but we know the probability of a given measurement. Deterministic because knowing all the initial conditions

and the Hamiltonian of a system enables us to calculate all probabilities for all time.

2.1.4 Linearity

All vectors can be written as a linear combination of basis vectors in a given coordinate basis,

$$|a\rangle = a_1|1\rangle + a_2|2\rangle + a_3|3\rangle \cdots \quad (2.3)$$

2.1.5 Linear independence and bases spanning a space

To properly define a basis for a vector space we first need a few definitions and theorems.

A set of vectors $\{|a\rangle, |b\rangle, \dots\}$ spans a vector space if every vector in that space can be written as a linear combination of vectors from that set.

A set of vectors is linearly independent if and only if no vector in the set is expressible as a linear combination of the other vectors in the set.

If S is a set of vectors in the space V , then S is a basis for V if

- S is linearly independent.
- S spans V .

If this is the case then any vector in V can always be written as a linear combination of the vectors in S .

2.1.6 Dual states and complex conjugates

The dual vector of $|a\rangle$ is $\langle a|$. The dual vector of $a_1|a\rangle$ is $a_1^*\langle a|$. This defines the Hermitian conjugate symbolized by the dagger,

$$(a_1|a\rangle)^\dagger = \langle a|a_1^*, \quad (2.4)$$

where a_1 is a complex scalar.

The Hermitian conjugate of an operator is defined by the conjugate of the state it operates on. If $\hat{\mathbf{F}}$ is an operator, the hermitian conjugate of $\hat{\mathbf{F}}|a\rangle$ is $\langle a|\hat{\mathbf{F}}^\dagger$. The Hermitian conjugate of a matrix is then the transposed matrix with all components complex conjugated, viz.,

$$\begin{aligned} \hat{\mathbf{F}} = |\psi\rangle\langle\phi| &\Rightarrow (\hat{\mathbf{F}}|a\rangle)^\dagger = (|\psi\rangle\langle\phi|a\rangle)^\dagger = \langle a|\phi\rangle\langle\psi| = \langle a|\hat{\mathbf{F}}^\dagger \\ &\Rightarrow \hat{\mathbf{F}}^\dagger = |\phi\rangle\langle\psi|. \end{aligned} \quad (2.5)$$

2.1.7 Inner and outer product

Since these state vectors are in a Hilbert space they must also have a defined inner product, which we write as $\langle a|b\rangle$. This is a function that takes two vectors and gives us a real number.

A vector is said to be normalized if the inner product of the vector with itself is one, $\langle x|x\rangle = 1$.

We also have an outer product, which takes two vectors and gives us an operator, $|a\rangle\langle b|$. In linear algebra the inner product corresponds to a row vector multiplied with a column matrix resulting in a number, and the outer product corresponds to a column matrix multiplied with a row matrix giving us an $N \times N$ matrix, where N is the dimension of the vectors.

The inner product also gives us the probability aspect of quantum mechanics. If we have a state $|\psi\rangle$ and a state $|\phi\rangle$, then the probability that $|\psi\rangle$ will be in the state $|\phi\rangle$ is $|\langle\phi|\psi\rangle|^2$, which is a real number, and less than or equal to one.

2.1.8 Orthonormality

As we stated in the definition of the Hilbert space, all state vectors must be normalized to one ensuring that the total probability of all outcomes is one.

Two vectors are orthogonal if their inner product is zero. That is, $\langle a|b\rangle = 0$ means $|a\rangle$ and $|b\rangle$ are orthogonal.

A set of normalized orthogonal vectors is called an orthonormal set.

2.1.9 Completeness

For a complete set of basis vectors, $\{|k\rangle\}$, any vector in that space can be described by a superposition of these vectors

$$|\psi\rangle = \sum_k c_k |k\rangle. \quad (2.6)$$

For an orthogonal basis, $\langle m|k\rangle = \delta_{m,k}$, the coefficients c_k are found by multiplying from the left with $\langle k|$, yielding

$$|\psi\rangle = \sum_k \langle k|\psi\rangle |k\rangle = \sum_k |k\rangle \langle k|\psi\rangle. \quad (2.7)$$

All complete basis sets must obey the completeness relation

$$\sum_k |k\rangle \langle k| = \hat{1}. \quad (2.8)$$

2.1.10 Eigenvalues and eigenvectors

The eigenvalues and eigenvectors of an operator are defined by the eigenvalue equation

$$\hat{\mathbf{A}}|a\rangle = \lambda_a|a\rangle, \quad (2.9)$$

where λ_a is a complex number called the eigenvalue, and $|a\rangle$ is the eigenvector.

2.1.11 Observables and operators

By an observable we mean a physical quantity. Something we can measure in a system, for example temperature in a box, the electrical charge or position of a molecule and so on.

The second postulate says that every such observable has a linear Hermitian operator associated with it. A quantum mechanical operator is a mathematical operator that changes a state into another state, in the bra/ket notation it is represented as an outer product of two states,

$$\hat{\mathbf{F}} = |a\rangle\langle b| \Rightarrow \hat{\mathbf{F}}|c\rangle = (|a\rangle\langle b|)|c\rangle = |a\rangle\langle b|c\rangle \equiv k|a\rangle, \quad (2.10)$$

where $\langle b|c\rangle \equiv k$ is a complex number, showing that $\hat{\mathbf{F}}$ has changed $|c\rangle$ into $|a\rangle$ times a complex scalar. When $|c\rangle$ is an eigenvector of $\hat{\mathbf{F}}$, then $|a\rangle = |c\rangle$. An operator can also operate on a bra state, then

$$\langle c|\hat{\mathbf{F}} = \langle c|a\rangle\langle b|. \quad (2.11)$$

Hermitian eigenvalues are real

Combining this with the third and fourth postulate, we see that the eigenvalues of hermitian operators (which are the only things we can measure) must be real, $\langle\psi|(\hat{\mathbf{F}}|\psi\rangle) = f\langle\psi|\psi\rangle = (\langle\psi|\hat{\mathbf{F}})|\psi\rangle = f^*\langle\psi|\psi\rangle \Rightarrow f = f^*$, see eq. (2.9) for the definition of eigenvalues.

2.1.12 Collapse of the state

The electron double slit experiment, see section 2.1.1, showed that measuring which slit the electron passed through affected the outcome of the experiment. It meant that when the interference pattern was produced, it was impossible to tell which slit the electron had used. It also meant that the state of the electron was changed by the measurement, it was forced to choose between the two slits and could no longer remain in a superposition of both possibilities. This destruction of superposition by measurement is called collapse of the state.

A system that can be in two different states, $|a\rangle$ and $|b\rangle$, can also be in a superposition of these two states, $|\phi\rangle = k_1|a\rangle + k_2|b\rangle$. A measurement can yield two different results with probabilities $|k_1|^2$ and $|k_2|^2$, but if another identical measurement is performed on the system, it can only yield the same result as the first, because the state of the system has changed into one of the eigenstates of the operator corresponding to the observable. If a measurement gets a result corresponding to $|a\rangle$ in the above system, then the state of the system has changed from $k_1|a\rangle + k_2|b\rangle$ to $k'|a\rangle$, where k' is a real normalization constant.

In some cases it is possible to gather information on a system without measuring directly, using the collapse of the state. In a thought experiment Elitzer and Vaidman [7] used a photon in a superposition of two states to determine whether a bomb was armed or not. By observing a result that could only come from a photon that was no longer in a superposition they could say that the bomb had forced the state of the photon to collapse, and therefore it was armed.

2.1.13 Time evolution

The first postulate gives us the Schrödinger equation and thereby the system's time evolution as a first order differential equation. Time evolution can also be seen as the work of an operator, which we define by

$$|\psi(t)\rangle = \hat{U}(t - t_0)|\psi(t_0)\rangle. \quad (2.12)$$

The time evolution operator must be reversible, $\hat{U}^{-1}\hat{U} = \hat{1}$, because there can be only one state when we move backwards in time. Together with normalization we see that \hat{U} must be unitary $\langle\psi(t)|\psi(t)\rangle = \langle\psi(t_0)|\hat{U}^\dagger\hat{U}|\psi(t_0)\rangle = \langle\psi(t_0)|\hat{U}^{-1}\hat{U}|\psi(t_0)\rangle = \langle\psi(t_0)|\psi(t_0)\rangle = 1$.

2.2 Qubits

The simplest, least complex, quantum mechanical system there is, is a two-level system, a system with only two basis states. When creating a quantum computer we can in principle choose any system, as long as we can device circuits to perform the calculation. In order to have a as general as possible computer we choose the basic components to be two-level systems, which we call qubits.

For a qubit we define an orthonormal basis $\{|0\rangle, |1\rangle\}$,

$$\langle 0|0\rangle = 1, \langle 1|1\rangle = 1, \langle 0|1\rangle = 0, \langle 1|0\rangle = 0. \quad (2.13)$$

The Hilbert space is complex so we can use complex numbers a and b to describe any state of the qubit

$$|\psi\rangle = a|0\rangle + b|1\rangle. \quad (2.14)$$

Classical computers use boolean algebra, with all variables being either false or true, zero or one. Binary numbers are made up by binary digits, also called bits, and strings of these can represent integers. A classical computer is made of circuits which perform logical operations on bits.

The qubit is the quantum mechanical equivalent of the classical bit, although it is no digit the name is used because it is the basic unit of computing for quantum computers. Any two level quantum system can be used as a physical qubit, for example an electron in a quantum dot has a spin projection $m_s = +1/2$ or $m_s = -1/2$. We can for example say that $m_s = +1/2$ means the value of the bit is zero, and $m_s = -1/2$ means the bit is one. Quantum mechanics has a natural binary representation that can be used in computations.

2.2.1 Qubits and matrices, the computational basis

When we have chosen a basis in the Hilbert space of the qubit, we can use a column matrix to represent a vector in that space. The first element is the coefficient of the $|0\rangle$ basis state, and the second element the coefficient of the $|1\rangle$ basis state,

$$|0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \quad (2.15)$$

For a general two-level state the matrix representation becomes

$$|\psi\rangle = a|0\rangle + b|1\rangle \rightarrow \begin{pmatrix} a \\ b \end{pmatrix} = a \begin{pmatrix} 1 \\ 0 \end{pmatrix} + b \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad (2.16)$$

Multiple qubits

To describe a quantum system of multiple parts, where each part is a quantum system with a Hilbert space, we introduce the tensor product, see appendix A.1 for more details. The state of a multiple part system is a vector in the sum of the Hilbert spaces of each subsystem, and the basis vectors are described as tensor products of the basis states of each subsystem.

A system consisting of two subsystems in the states $|a\rangle$ and $|b\rangle$ respectively, is in the total state $|a\rangle \otimes |b\rangle$. This is called a tensor product.

For a two-qubit system the four basis states are

$$|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle. \quad (2.17)$$

A two qubit system has four different tensor products of basis states, this means we need a column matrix with four elements to represent it, and we define the representation as

$$|0\rangle \otimes |0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, |0\rangle \otimes |1\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, |1\rangle \otimes |0\rangle \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, |1\rangle \otimes |1\rangle \rightarrow \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \quad (2.18)$$

We say that the qubit to the left in the tensor product is the first, and continue to enumerate from left to right. The generalization of this is called the computational basis, where any 2^N dimensional vector can be described by N two-dimensional vectors.

Any two qubit state can be written as a column matrix using this representation. Here we find the product state of two arbitrary one qubit states,

$$\begin{aligned} |\xi\rangle \otimes |\eta\rangle &= (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \\ &= ac|0\rangle \otimes |0\rangle + ad|0\rangle \otimes |1\rangle + bc|1\rangle \otimes |0\rangle + bd|1\rangle \otimes |1\rangle \\ &\rightarrow \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix} \end{aligned} \quad (2.19)$$

The rules for tensor products from appendix A.1, show us that the basis column matrices for higher level systems can be derived from the two-level ones, as an example a basis state in a four dimensional Hilbert space,

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \quad (2.20)$$

In this way we can describe multi-dimensional systems using two level building blocks, and when building a quantum computer we can simulate a complex system using several basic two-level systems, qubits. If we have n two-dimensional vectors, their tensor products can describe a 2^n -dimensional system.

Some texts use shorter ways of writing the states of multi-qubit systems, instead of $|0\rangle \otimes |1\rangle \otimes |0\rangle$ some write $|0\rangle|1\rangle|0\rangle$, skipping the \otimes 's, or even shorter $|010\rangle$. In this thesis we will use four different notations for a multiple qubit state. The two first are

$$\begin{aligned} &|0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |1\rangle \\ &= |0\rangle|1\rangle|1\rangle|1\rangle, \end{aligned} \quad (2.21)$$

we have skipped the \otimes signs in the second notation. We will also use two other notations, especially in chapter 3 discussing the Fourier transform and the phase-estimation algorithm. An n qubit system has 2^n basis states, if we enumerate them from 0 to $2^n - 1$, with $|0 \dots 00\rangle$ as number zero, $|0 \dots 01\rangle$ as number one, $|0 \dots 11\rangle$ as number two and $|1 \dots 11\rangle$ as number $2^n - 1$. We see that the string of ones and zeros gives the enumeration as a binary number, $0 \dots 00 = 0$, $0 \dots 01 = 1$, $0 \dots 10 = 2$, and $1 \dots 11 = 2^n - 1$. In this third notation we use binary enumeration

$$|0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |1\rangle = |0111\rangle. \quad (2.22)$$

We will also at some points use integer enumeration. For example the binary number 0111 equals the integer 7, and we write

$$|0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |1\rangle = |7\rangle. \quad (2.23)$$

For a comprehensive listing of the multi-qubit basis states see eq. (3.16).

These four different notations can be confusing and we will do our best to inform the reader when we change from one to another.

2.2.2 Operators as matrices

In the same way as vectors are represented as column matrices, we can represent operators as square matrices in the computational basis. A vector is represented by a column matrix, a dual vector is then represented by a row matrix. In this way an inner product of a dual vector and a vector is represented by the matrix multiplication of a row matrix and a column matrix.

$$\langle a|b\rangle \rightarrow (a_0 \ a_1) \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = a_0^* b_0 + a_1^* b_1.$$

Quantum mechanical operators are linear and can be written as superpositions of other operators. A vector basis can be used to describe any operator in that vector space, in the one qubit case a general operator in the computational basis is given by

$$\hat{\mathbf{A}} = a|0\rangle\langle 0| + b|0\rangle\langle 1| + c|1\rangle\langle 0| + d|1\rangle\langle 1|. \quad (2.24)$$

Using the rules of outer-products, matrix multiplication of a column matrix and a row matrix, we define a matrix to represent this operator as follows,

$$|0\rangle\langle 0| \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} (1 \ 0) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

$$|0\rangle\langle 1| \rightarrow \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},$$

$$|1\rangle\langle 0| \rightarrow \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix},$$

$$|1\rangle\langle 1| \rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

This gives us a matrix representation $\hat{\mathbf{A}}$,

$$\hat{\mathbf{A}} \rightarrow \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

The rules of tensor products provide us with a means to calculate matrices representing operators for systems of several qubits from operators on single qubits or other subsystems.

2.2.3 Applying operators to vectors/states

So far we have two different ways of describing an operation on a state, the bra/ket notation of Dirac and matrix multiplication from linear algebra. Here we calculate the result of a general one qubit operation using both the bra/ket notation and matrices, $\hat{\mathbf{A}}$ on the general state $|\psi\rangle = \xi|0\rangle + \eta|1\rangle$. First in the bra/ket way

$$\begin{aligned} \hat{\mathbf{A}}|\psi\rangle &= (a|0\rangle\langle 0| + b|0\rangle\langle 1| + c|1\rangle\langle 0| + d|1\rangle\langle 1|)(\xi|0\rangle + \eta|1\rangle) = \\ &\xi(a|0\rangle\langle 0|0\rangle + b|0\rangle\langle 1|0\rangle + c|1\rangle\langle 0|0\rangle + d|1\rangle\langle 1|0\rangle) + \\ &\eta(a|0\rangle\langle 0|1\rangle + b|0\rangle\langle 1|1\rangle + c|1\rangle\langle 0|1\rangle + d|1\rangle\langle 1|1\rangle) \\ &= \xi(a|0\rangle + c|1\rangle) + \eta(b|0\rangle + d|1\rangle) \\ &= (\xi a + \eta b)|0\rangle + (\xi c + \eta d)|1\rangle. \end{aligned}$$

Here is the same calculation in the matrix representation

$$\hat{\mathbf{A}}|\psi\rangle \rightarrow \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \xi \\ \eta \end{pmatrix} = \begin{pmatrix} a\xi + b\eta \\ c\xi + d\eta \end{pmatrix}.$$

2.2.4 Product states and entanglement

When we have a system consisting of two or more subsystems each with a Hilbert space, the Hilbert space of the total system is a product space.

If a system has two subsystems in the states $|\alpha\rangle$ and $|\beta\rangle$, then the total state is $|\alpha\rangle \otimes |\beta\rangle$, which is a state in the product space of the system. This is a tensor product.

A product state is a state that can be written as a tensor product of single qubit states, see appendix A.1. A state that cannot be written as a product state is entangled, and must be represented as a superposition of product states.

In a two qubit system the basis states for each qubit are $|0\rangle$ and $|1\rangle$. The state $|0\rangle \otimes |0\rangle$ is a product state, while $|0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle$ is entangled. The state $|0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle$ is not entangled however, as it can be written as $(|0\rangle + |1\rangle) \otimes |0\rangle$.

Entanglement means that the state of a sub-system cannot be known separately from the rest of the system. In the above entangled state,

$$\frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle), \quad (2.25)$$

now normalized, if we measure a $|0\rangle$ for the first qubit we know that the second qubit must be in the $|1\rangle$ state, and vice versa. We could take two particles in this state and measure both in such a short time interval, and at such a distance, that no signal moving at the speed of light could be sent between them. When we measure one particle both results are equally likely, but then the measurement of the other is given. Theoretically we could measure one particle at one end of the universe and know immediately what the other measurement would be, but if the state of the other particle was determined first, the result of our measurement would be given. Does the particles affect each other faster than light? or do we change the world by observing?

This is what made Einstein, Podolsky and Rosen write an article in 1935, [6]. They believed that neither of the above explanations could be true and therefore quantum mechanics could not be a complete theory of nature. Bell showed in 1965, [2], through his famous inequalities, that there was a fundamental difference between quantum mechanics and the views of Einstein, Rosen and Podolsky. In 1982, Aspect et al. [1] demonstrated through an experiment, that, within experimental uncertainties, Bell's inequalities were broken, lending support to the Copenhagen interpretation of quantum mechanics.

Entanglement is not merely interesting for philosophical reasons, it is of fundamental importance to quantum computation. It lies at the heart of such phenomena as quantum teleportation and maximum parallelism, allowing us to do the same computation on a vast number of states at the same time. In fact, Guifré Vidal showed in [22] that a quantum algorithm that is only slightly entangled can be simulated efficiently on a classical computer, showing us that we need entanglement in quantum computation to really exploit the power of quantum computers.

2.2.5 Example of a 4×4 operation

In this example we have a two qubit system and wish to perform two different operations on each of these qubits, what is the result?

The first qubit is in the $|0\rangle$ state, the second in the $1/\sqrt{2}(|0\rangle + |1\rangle)$ state. In matrix terms the total state is given by the tensor product

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}. \quad (2.26)$$

We wish to perform the Pauli x operation on the first qubit, and the Pauli z operation on the second, see section 2.4. The total operation is given by the tensor product of the two matrices

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \quad (2.27)$$

Performing the operation on the state yields the final state

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \end{pmatrix}. \quad (2.28)$$

In Dirac notation the result is $1/\sqrt{2}|1\rangle \otimes (|0\rangle - |1\rangle)$, we see that it is indeed what we would expect from the operations on each of the qubits.

2.3 Quantum gates and quantum circuits

Here we will give a third and equivalent description of quantum states and operators.

Quantum gates are the equivalent of classical gates on classical computers. They perform operations on the quantum equivalent to the bit, the qubit. Series of quantum gates form quantum circuits, which enable us to perform computations on qubits.

2.3.1 A quantum circuit

To perform a computation we might need several different operators working on different qubits. A set of operations on a set of qubits is called a quantum circuit.

The operation performed by a quantum gate must be a non-measurement physical operation, a time evolution guided by the Schrödinger equation, and all quantum gate operations must therefore be unitary and reversible, see section 2.1.13.

Like circuits for classical computers, we use diagrams for visualization. In fig. 2.1 we have the classical *NAND* gate, with its corresponding truth table. Its output is determined by the two input values, and if the output is 1 we cannot know what the input was and cannot traverse the circuit backwards in time, making this an irreversible gate and a gate that we cannot have in quantum computing.

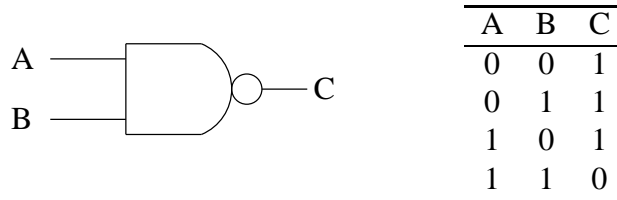


Figure 2.1: *NAND* gate with truth table

In a circuit diagram every line represents a qubit, and every box represents a gate working on the qubits whose lines goes through it. Below in fig. 2.2 we have an example of a quantum circuit.

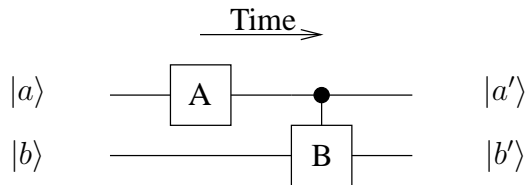


Figure 2.2: A quantum circuit

The state of the first qubit before the operator is applied is $|a\rangle$, and $|a'\rangle$ after, showing the left to right time development in all quantum circuits.

However, since all quantum gates are reversible we could traverse the gate the other way, but then we would have to invert the gates to find their effect backwards in time, $\hat{U}(-\Delta t) = \hat{U}^{-1}(\Delta t)$, using the hermitian conjugate of operators we found in section 2.1.6. In addition we see that the uppermost line represents the first qubit in the tensor-product representation of the total state. We have a state $|a\rangle \otimes |b\rangle$, and $|a\rangle$ is represented by the upper line, and $|b\rangle$ the next and so on.

The qubits in a circuit are ordered from top down when compared to tensor products ordered from left to right.

$$\begin{array}{ll} \text{Qubit 1:} & |0\rangle \quad \text{—————} \\ \text{Qubit 2:} & \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad \text{—————} \end{array}$$

Figure 2.3: Qubit ordering

Using the bra/ket notation the two qubit state in fig. 2.3 is written as

$$|0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle - |0\rangle \otimes |1\rangle). \quad (2.29)$$

Using matrices the same state is written as

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix}. \quad (2.30)$$

2.3.2 Swapping and conditional operations

Swapping and conditional operations are important and we use them extensively throughout the thesis.

Swapping is an operation that interchanges the coefficients of a state. The X gate, see section 2.4 for a description, swaps the coefficients of a single qubit

$$X(a|0\rangle + b|1\rangle) = b|0\rangle + a|1\rangle. \quad (2.31)$$

A conditional operator is a two-qubit operator that performs an operation on the target qubit if the control is in a given state, and is therefore comparable to a classical if-test.

If U is a single-qubit operator, then the operator that performs U on the target qubit if the control qubit is in the $|1\rangle$ state, is called CU . The C is a

prefix implying it is a conditional operation. To see how this works we show the effect of the CU operator on a two-qubit state with the first qubit as the control and the second qubit as the target,

$$\begin{aligned} CU[(a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle)] \\ = a|0\rangle \otimes (c|0\rangle + d|1\rangle) + b|1\rangle \otimes U(c|0\rangle + d|1\rangle), \end{aligned} \quad (2.32)$$

we see that only the basis two-qubit states with the first qubit in the $|1\rangle$ state are affected. Using matrices the same operation is

$$CU \left[\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} \right] = \begin{pmatrix} a \\ b \end{pmatrix} \otimes U \begin{pmatrix} c \\ d \end{pmatrix}. \quad (2.33)$$

In section 5.1.4 we derive the matrix representation of the CU operation,

$$CU = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_a & U_b \\ 0 & 0 & U_c & U_d \end{pmatrix}. \quad (2.34)$$

The $CNOT$ gate is a very important gate, it is an abbreviation of controlled not, and is an example of a conditional operation. It is a CU operation where U is the X gate, which means it swaps the state of the second qubit if the first qubit is in the $|1\rangle$ state. In fig. 2.4 we show the quantum gate representing the $CNOT$ operation and a truth table.

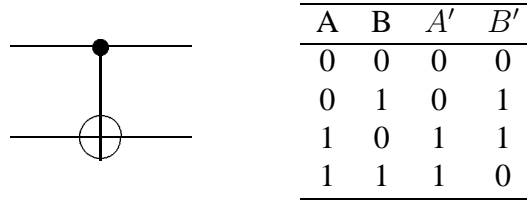


Figure 2.4: $CNOT$ gate

The $CNOT$ operation on basis states in the bra/ket notation,

$$\begin{aligned} |0\rangle \otimes |0\rangle &\rightarrow |0\rangle \otimes |0\rangle, \\ |0\rangle \otimes |1\rangle &\rightarrow |0\rangle \otimes |1\rangle, \\ |1\rangle \otimes |0\rangle &\rightarrow |1\rangle \otimes |1\rangle, \\ |1\rangle \otimes |1\rangle &\rightarrow |1\rangle \otimes |0\rangle. \end{aligned}$$

Finally we show the corresponding matrix representation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \quad (2.35)$$

In fig. 3.1 we see an example of how *CNOT* gates can be used to swap the states of two qubits. In section 3.2.2 we use other swap operations to swap basis state coefficients of multiple qubit systems.

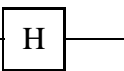
In section 3.2 we show that single-qubit gates together with the two-qubit *CNOT* gate can be combined to perform any unitary transformation, and therefore we say they are universal. An actual quantum computer will have to be able to perform a certain set of universal operations, it is not certain that the best physical realization has a *CNOT* operation as a single physical operation. Maybe it is a controlled *Z* operation, or some other kind of controlled operation, but in our thesis the *CNOT* is the most important building block of our theoretical quantum computer. In the discrete quantum Fourier transform, the phase estimation algorithm and our implementation of the quantum simulation algorithm, we use several conditional operations. In [16], page 177, it is shown how one can construct any two qubit conditional operator, *CU*, by using two *CNOT* gates and four single qubit gates. This means a total of six operations are needed to perform a general two-qubit conditional operation.

In section 2.4 we give a listing of some the most commonly used gates and their symbols.

2.4 Gates

In this section we list the basic gates used in this thesis. First we show the pictorial representation used in various circuit diagrams, and then we show the matrix representation and it's effect on an arbitrary target state.

The $\pi/8$ and phase gates are not used in this thesis, but are included since we mention them when discussing universal sets of elementary operations in section 3.2.

Hadamard 

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} a+b \\ a-b \end{pmatrix}$$

$$X \quad \text{---} \boxed{\text{X}} \text{---} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a \end{pmatrix}$$

$$Y \quad \text{---} \boxed{\text{Y}} \text{---} \quad \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} -bi \\ ai \end{pmatrix}$$

$$Z \quad \text{---} \boxed{\text{Z}} \text{---} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ -b \end{pmatrix}$$

$$CNOT \quad \begin{array}{c} \bullet \\ | \\ \oplus \end{array} = \begin{array}{c} \bullet \\ | \\ \boxed{\text{X}} \end{array} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ b \\ d \\ c \end{pmatrix}$$

$$XCNOT \quad \begin{array}{c} \circ \\ | \\ \oplus \end{array} \equiv \begin{array}{c} \boxed{\text{X}} \bullet \boxed{\text{X}} \\ | \\ \oplus \end{array} \quad \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} b \\ a \\ c \\ d \end{pmatrix}$$

$$\frac{\pi}{8} \quad \text{---} \boxed{\text{T}} \text{---} \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} a+b \\ a-b \end{pmatrix}$$

$$\text{phase} \quad \text{---} \boxed{\text{S}} \text{---} \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} a+b \\ a-b \end{pmatrix}$$

Chapter 3

Quantum computation

In this chapter we introduce a quantum computer simulation algorithm, the foundation upon which all our work is based. To implement it in the language of quantum gates we need to prove that quantum computers are universal, and find a way to describe any operation as a set of gates. Then we introduce the discrete quantum fast Fourier transform and the phase-estimation algorithm, which we use to extract information from the simulation algorithm.

In this chapter we have drawn extensively from the book Quantum Computation and Quantum Information by Nielsen and Chuang, [16], which is an excellent introduction, and more, to quantum computers.

3.1 The simulation algorithm

The quantum-mechanical time evolution is governed by the Schrödinger equation, see eq. (2.1). Our goal is to calculate the state of the system for all time, to do that we need the Hamiltonian and the initial state $|\psi(t_0)\rangle$. We want the time evolution as an operator seen in eq. (2.12), and to do that we use the Schrödinger picture where $\hat{\mathbf{H}}$ is time independent and the state itself is a function of time. We set $\hbar = 1$ and get

$$U = e^{-i\hat{\mathbf{H}}(t-t_0)} \equiv e^{-i\hat{\mathbf{H}}(\Delta t)}, \quad (3.1)$$

where the exponential of an operator is defined through the series expansion of the exponential function.

3.1.1 Simulating a quantum system

One of the great uses of quantum computers will be in simulating quantum systems, most of which are too complex to simulate on a classical computer due to limitations on speed and memory.

This simulation algorithm is a recipe for designing a quantum computer circuit that will perform operations on qubits approximating the effect of a time evolution operator on a quantum mechanical system.

From Hamiltonian to gates

In section 3.2.2, we show that any unitary operation on a finite set of qubits, a finite system, can be represented by single-qubit gates and *CNOT* gates. This means that if we have a Hamiltonian which consists of a sum of operations where each part only operates on a finite part of the system, $\hat{\mathbf{H}} = A + B$, then $e^{-iA\Delta t}$ can be simulated by a quantum circuit. However, due to the fact that operators do not generally commute, $AB \neq BA$, we cannot straightforwardly factorize $\hat{\mathbf{U}}$, $e^{-i(A+B)\Delta t} \neq e^{-iA\Delta t}e^{-iB\Delta t}$.

The Trotter formula, see [16] page 207 for more details, gives us an approximation we can use,

$$e^{-i(A+B)\Delta t} = e^{-iA\Delta t}e^{-iB\Delta t} + \mathcal{O}(\Delta t^2). \quad (3.2)$$

to order $\mathcal{O}(\Delta t^3)$ we have

$$e^{-i(A+B)\Delta t} = e^{-iA\Delta t/2}e^{-iB\Delta t}e^{-iA\Delta t/2} + \mathcal{O}(\Delta t^3). \quad (3.3)$$

A Hamiltonian that is a sum of operations on parts of the system, $\hat{\mathbf{H}} = \sum_i \hat{\mathbf{H}}_i$, can be used with these approximations to find the time evolution operator as a product of operators on parts of the system.

To achieve a good approximation we need to use small time steps, Δt , and apply $\hat{\mathbf{U}}$ several times. We split the time interval into several smaller intervals, $\Delta t' = \Delta t/n$, and apply $\hat{\mathbf{U}}(\Delta t')$ consecutively n times. If we perform $\hat{\mathbf{U}}(\Delta t/2)$ twice the result is

$$\begin{aligned} [U(\Delta t/2) + \mathcal{O}((\Delta t/2)^2)][U(\Delta t/2) + \mathcal{O}((\Delta t/2)^2)] \\ = U^2(\Delta t/2) + \mathcal{O}((\Delta t/2)^2) \\ = U(\Delta t) + \mathcal{O}((\Delta t/2)^2), \end{aligned} \quad (3.4)$$

the error is now of the order of $\Delta t^2/4$, which means the error is reduced by a quarter when Δt is halved. If we choose $\Delta t' = \Delta t/n$ to be sufficiently small, the error $\mathcal{O}(\Delta t'^2)$ will be negligible.

Algorithm

Any quantum mechanical system where we know the Hamiltonian can be simulated using this algorithm, provided it can be described as a sum of

operations on finite subsystems. Below we show the steps needed to perform the simulation.

- Take the Hamiltonian, $\hat{\mathbf{H}} = \sum_i \hat{\mathbf{H}}_i$.
- Use eq. 3.3 to factorize the time evolution operator $\hat{\mathbf{U}}$ into operations on parts of the system.
- Choose a time interval Δt , divide it into n parts to minimize the error.
- Design a quantum circuit for each factor in $\hat{\mathbf{U}}(\Delta t)$, see section 3.2.2 for an example.
- Repeat the circuit n times.

In chapter 4 we show the implementation of this algorithm using the one-dimensional Heisenberg Hamiltonian.

3.2 The Universality of quantum computers

Quantum computers would have limited usefulness if we had to build a new one for every problem we wanted to solve. The aim is to find quantum computers that can solve all problems.

Quantum computers are indeed universal, as we will see in this section there exists sets of universal operations that can approximate any unitary operator. That unitary operators can be used to solve any computational problem was proven by Benioff in [3].

In the case of classical computers there are gates which can be combined to do any logical operation, an example is the *AND*, *OR* and *NOT* gates. All quantum mechanical operations have to be reversible however, so we cannot find quantum equivalents of these gates to provide universality.

In this part we will show two things. First that all unitary matrices can be factorized into two-level unitary matrices. Then we will show that all two-level unitary matrices can be implemented as operations by the *CNOT* (see fig. 2.4) gate and single-qubit gates. Together these results show that any quantum gate can be made by using only *CNOT* gates and single-qubit gates.

There are also finite (small) sets of single-qubit operations that can approximate any single-qubit operation with arbitrary precision, and therefore are universal in conjunction with the *CNOT* gate. We will not prove this in this thesis, but refer to Nielsen and Chuang [16] page 194. It is generally

hard to approximate a gate using these sets, meaning there is no simple algorithm. An example of a universal set of gates is the *CNOT*, Hadamard, phase and $\pi/8$ gates. These gates are described in section 2.4.

3.2.1 Two-level unitary gates are universal

A two-level unitary matrix is a matrix which acts non-trivially only on two or fewer vector components,

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a & \cdots & b & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & c & \cdots & d & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_\xi \\ \vdots \\ \alpha_\eta \\ \vdots \\ \alpha_{t-1} \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \vdots \\ a\alpha_\xi + b\alpha_\eta \\ \vdots \\ c\alpha_\xi + d\alpha_\eta \\ \vdots \\ \alpha_{t-1} \end{pmatrix}, \quad (3.5)$$

with all elements on the diagonal being 1, except where a, b, c or d are on the diagonal. It is an identity matrix plus a matrix with only 4 components different from zero.

3×3 matrix

To prove that all unitary matrices can be factorized into two-level unitary matrices, we first show it for the 3×3 case

$$U \equiv \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & j \end{pmatrix}. \quad (3.6)$$

For a 3×3 matrix we need three two-level unitary matrices, U_1, U_2 and U_3 , that fulfill

$$U_3 U_2 U_1 U = I. \quad (3.7)$$

If they do, then

$$\begin{aligned} U_1^\dagger U_2^\dagger U_3^\dagger U_3 U_2 U_1 U &= U_1^\dagger U_2^\dagger U_3^\dagger I \\ &= U_1^\dagger U_2^\dagger U_3^\dagger \\ &= U. \end{aligned} \quad (3.8)$$

Because the inverse of a two-level unitary matrix is also a two-level unitary matrix, we will have proved this universality for 3×3 matrices if we find U_1 ,

U_2 and U_3 . When we have done that, we use the same procedure to show it for the general case of an $N \times N$ matrix.

First we choose U_1 to be

$$\begin{pmatrix} \frac{a^*}{\sqrt{|a|^2+|b|^2}} & \frac{b^*}{\sqrt{|a|^2+|b|^2}} & 0 \\ \frac{b}{\sqrt{|a|^2+|b|^2}} & \frac{-a}{\sqrt{|a|^2+|b|^2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.9)$$

Then we multiply U_1 with U and get

$$U_1 U = \begin{pmatrix} a' & d' & g' \\ 0 & e' & h' \\ c' & f' & j' \end{pmatrix}. \quad (3.10)$$

Here we have introduced new variables a' and so forth, their values are not interesting. What we need is that the first component of the second row is zero, as we can see from

$$\frac{b}{\sqrt{|a|^2+|b|^2}}a + \frac{-a}{\sqrt{|a|^2+|b|^2}}b + 0 = 0.$$

Applying a matrix U_2 of this form

$$U_2 \equiv \begin{pmatrix} \frac{a'^*}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{c'^*}{\sqrt{|a'|^2+|c'|^2}} \\ 0 & 1 & 0 \\ \frac{c'}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{-a'}{\sqrt{|a'|^2+|c'|^2}} \end{pmatrix} \quad (3.11)$$

to $U_1 U$, we find

$$U_2 U_1 U = \begin{pmatrix} 1 & d'' & g'' \\ 0 & e'' & h'' \\ 0 & f'' & j'' \end{pmatrix}. \quad (3.12)$$

$U_2 U_1 U$ is unitary since each of the factors are. This means that the Hermitian conjugate is also the inverse and therefore the renamed factors d'' and g'' must be zero. If we now choose U_3 to be the Hermitian conjugate of $U_2 U_1 U$,

$$U_3 \equiv (U_2 U_1 U)^\dagger = U^\dagger U_1^\dagger U_2^\dagger = \begin{pmatrix} 1 & 0 & 0 \\ 0 & e''^* & f''^* \\ 0 & h''^* & j''^* \end{pmatrix}, \quad (3.13)$$

then $U_3 U_2 U_1 U$ will be the identity matrix.

This proves it for the 3×3 case, for larger matrices we follow the same procedure to find lower level unitary matrices. If we have an $N \times N$ unitary

matrix U , we can find a two-level unitary matrix, U_1 , so that the second element in the first column of U_1U is zero. Using the technique above we then find U_2 so that the third element of the first row of U_2U_1U is zero, and so on. When we have all zeros below the diagonal in the first column, we have used at most $N - 1$ matrices. Now we have the case of eq. (3.13) where U is the product of $N - 1$ two-level unitary matrices and one $(N - 1)$ -level unitary matrix, A ,

$$U' = \left[\prod_i U_i U \right] A \equiv \left[\prod_i U_i U \right] \begin{pmatrix} 1 & 0 \\ 0 & A^{(N-1) \times (N-1)} \end{pmatrix}^{N \times N}.$$

Using the same procedure we find $N - 2$ two-level unitary matrices that multiplied with U' gives us an $(N - 2)$ -level unitary matrix. In the end we have U as a product of at most

$$k \leq \sum_{i=1}^N (N - i) = (N - 1) + (N - 2) + (N - 3) + \cdots + 1 = N(N - 1)/2 \quad (3.14)$$

two-level unitary matrices.

3.2.2 *CNOT* and single-qubit gate universality

What remains now is to prove that a two-level unitary matrix in the computational basis can be represented as a series of *CNOT* and single-qubit gates.

From eq. (3.5) we see that the effect of the operator U is the unity operator on all state vector elements except α_ξ and α_η . We define the matrix \tilde{U} to be a 2×2 matrix of the non-trivial components of U from eq. (3.5),

$$\tilde{U} \equiv \begin{pmatrix} a & b \\ c & d \end{pmatrix}. \quad (3.15)$$

We will show that a series of operations where we swap coefficients of basis states, *CNOT*s, and a conditional \tilde{U} operation will be the equivalent of the U operator. See section 2.3.2 for more information on swapping and conditional operations.

Quantum circuits of unitary two-level matrices

An N -qubit system has the dimension 2^N . In the computational basis, basis vectors of this system are the tensor products of $|0\rangle$ and $|1\rangle$ states for each qubit. The first is $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle$ where all qubits are in the $|0\rangle$ state,

and then we have one qubit in the $|1\rangle$ state, $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |1\rangle$ and so on. We enumerate these states as $|0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle$ being the zero state, or in binary numbers $00 \dots 0$, and all other states as the binary number you get from the sequence of numbers representing the single-qubit states. See section 2.2.1 on the computational basis. Here we use eqs. (2.22) and (2.23) to represent the same multi-qubit basis states,

$$\begin{aligned}
 |0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle \otimes |0\rangle &= |0 \dots 00\rangle = |0\rangle \\
 |0\rangle \otimes |0\rangle \otimes \cdots \otimes |0\rangle \otimes |1\rangle &= |0 \dots 01\rangle = |1\rangle \\
 |0\rangle \otimes |0\rangle \otimes \cdots \otimes |1\rangle \otimes |0\rangle &= |00 \dots 10\rangle = |2\rangle \\
 &\vdots \\
 |1\rangle \otimes |1\rangle \otimes \cdots \otimes |1\rangle \otimes |1\rangle &= |1 \dots 11\rangle = |2^N - 1\rangle.
 \end{aligned} \tag{3.16}$$

This way we can describe any given state in this basis as

$$|\alpha\rangle = \sum_{i=0}^{2^N-1} \alpha_i |i\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_\xi |\xi\rangle + \cdots + \alpha_\eta |\eta\rangle + \cdots + \alpha_{2^N-1} |2^N - 1\rangle, \tag{3.17}$$

where we have used i to enumerate the basis states, see eq. (2.23). These coefficients are the ones used in the column matrix in eq. (3.5).

The effect of the two-level unitary operator U , from eq. (3.5), on $|\alpha\rangle$ is

$$\begin{aligned}
 U|\alpha\rangle &= U \sum_{i=0}^{2^N-1} \alpha_i |i\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + (a\alpha_\xi + b\alpha_\eta) |\xi\rangle + \cdots \\
 &\quad + (c\alpha_\xi + d\alpha_\eta) |\eta\rangle + \cdots + \alpha_{2^N-1} |2^N - 1\rangle.
 \end{aligned} \tag{3.18}$$

How can an operation that only affects certain computational basis states be represented by gates? If a state affects only the $|j\rangle = |j_0\rangle |j_1\rangle \dots |j_N\rangle$ basis state (where we use the second and third types of tensor products from eqs. (2.22) and (2.23)), where j_i is zero or one, then we would have to create a gate conditional upon all qubits being in the states j_1 through j_N . A conditional gate is basically an if-test, see fig. 2.4 for an example of a conditional gate.

We want an operation that only applies to two basis states that differ in the k^{th} qubit,

$$|j_0\rangle |j_1\rangle \dots |j_k = 0\rangle \dots |j_N\rangle, \tag{3.19}$$

$$|j_0\rangle |j_1\rangle \dots |j_k = 1\rangle \dots |j_N\rangle. \tag{3.20}$$

We do that by creating a circuit that is conditional on all qubits, except the k^{th} , being in the $|j_0\rangle$ through $|j_N\rangle$ states. In [16] page 184 it is shown how multiple *CNOT*, Hadamard, *phase* and $\pi/8$ gates can be used to create a gate that is conditional on any number of qubits.

Following is a two qubit example where A , with the matrix of non-trivial components \tilde{A} given by eq. (3.15), operates on the $|0\rangle \otimes |1\rangle$ and $|1\rangle \otimes |1\rangle$ states. The matrix representation of A and it's effect on a general state is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ a\alpha_2 + b\alpha_3 \\ c\alpha_2 + d\alpha_3 \end{pmatrix}. \quad (3.21)$$

When compared to eq. (2.34) we see that A is indeed CU , and the U from eq. (2.34) is in this case the matrix of non-trivial components of A given in eq. (3.15). What this means is that A can be simulated by a controlled \tilde{A} operation. In the same way all 4×4 two-level unitary matrices can be implemented by controlled operations. The operations seen in table 3.1 are all the 4×4 two-level unitary matrices that do not require swap gates to be implemented by conditional operations. All conditional operations can be implemented by *CNOT* gates and single qubit gates. In total six operations are needed to implement a general CU operation, see [16] page 177 for proof.

Consider the following example

$$A = \begin{pmatrix} a & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ c & 0 & 0 & d \end{pmatrix}, \quad (3.22)$$

where A operates non-trivially on both the $|00\rangle$ and $|11\rangle$ states, A cannot be represented as a single conditional operation, because the two states have different values for both qubits. What we do is swap the coefficients of A so that the coefficients appear on basis states with only one qubit value differing. That way we can use one of the conditional operations in table 3.1, and then swap back. For A in eq. (3.22) we see that if we swap the coefficients of the $|1\rangle|1\rangle$ and the $|0\rangle|1\rangle$ states,

$$|1\rangle \otimes |1\rangle \rightarrow |0\rangle \otimes |1\rangle, \quad (3.23)$$

and then perform the XCU operation on table 3.1, and swap back, we will have implemented A .

Using matrices we see how the swapping of coefficients can be used together with the conditional operation to obtain A . The effect of A on a

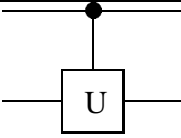
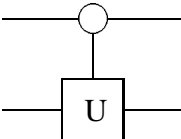
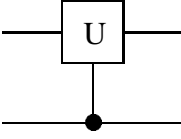
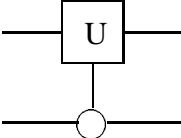
States	operated on	matrix	name	gate
$ 1\rangle \otimes 0\rangle$	$ 1\rangle \otimes 1\rangle$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{pmatrix}$	CU	
$ 0\rangle \otimes 0\rangle$	$ 0\rangle \otimes 1\rangle$	$\begin{pmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	XCU	
$ 0\rangle \otimes 1\rangle$	$ 1\rangle \otimes 1\rangle$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & c & 0 & d \end{pmatrix}$	UC	
$ 0\rangle \otimes 0\rangle$	$ 1\rangle \otimes 0\rangle$	$\begin{pmatrix} a & 0 & b & 0 \\ 0 & 1 & 0 & 0 \\ c & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	XUC	

Table 3.1: Table of two-qubit conditional operations

general state is

$$\begin{pmatrix} a & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ c & 0 & 0 & d \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} a\alpha_0 + b\alpha_3 \\ \alpha_1 \\ \alpha_2 \\ c\alpha_0 + d\alpha_3 \end{pmatrix}. \quad (3.24)$$

The swapping of coefficients between $|0\rangle|1\rangle$ and $|1\rangle|1\rangle$ is performed by an operator we call $\hat{\mathbf{S}}$,

$$\hat{\mathbf{S}} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_3 \\ \alpha_2 \\ \alpha_1 \end{pmatrix}, \quad (3.25)$$

then we apply XCU from table 3.1

$$\begin{pmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_3 \\ \alpha_2 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} a\alpha_0 + b\alpha_3 \\ c\alpha_0 + d\alpha_3 \\ \alpha_2 \\ \alpha_1 \end{pmatrix}, \quad (3.26)$$

and finally we apply $\hat{\mathbf{S}}$ once more

$$\hat{\mathbf{S}} \begin{pmatrix} a\alpha_0 + b\alpha_3 \\ c\alpha_0 + d\alpha_3 \\ \alpha_2 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} a\alpha_0 + b\alpha_3 \\ \alpha_1 \\ \alpha_2 \\ c\alpha_0 + d\alpha_3 \end{pmatrix}, \quad (3.27)$$

and we have the desired state.

The swap operation between the $|0\rangle|1\rangle$ and $|1\rangle|1\rangle$ states can be implemented by a $CNOT$ gate with the first qubit as the target. Using abstract operators the implementation of A is

$$\hat{\mathbf{S}} XCU \hat{\mathbf{S}}. \quad (3.28)$$

Swapping and applying the conditional \tilde{V}

Using the above method we can implement any two-level unitary matrix with swaps and conditional operations, but when we have a higher-dimensional unitary matrix we must perform more swaps before we can apply the conditional operation.

Here we generalize to the case where there are more than two qubits differing between the states, by showing a general procedure for creating a quantum circuit of a two-level unitary matrix.

For two states, $|a\rangle$ and $|b\rangle$, differing only in the i^{th} qubit, we define the swap operation $\hat{\mathbf{S}}(a,b)$. $\hat{\mathbf{S}}$ is a flip (Pauli X -matrix, see section 2.4) operation on the i^{th} qubit, conditional on all the other qubits being in the same states as for $|a\rangle$ and $|b\rangle$. That is, $\hat{\mathbf{S}}(a,b)|a\rangle = |b\rangle$ and $\hat{\mathbf{S}}(b,a)|b\rangle = |a\rangle$. If $|a\rangle = |0\rangle|1\rangle \dots |0\rangle \dots |0\rangle|1\rangle$, and $|b\rangle = |0\rangle|1\rangle \dots |1\rangle \dots |0\rangle|1\rangle$, the conditional flip on the total state will flip the i^{th} qubit only on states $|a\rangle$ and $|b\rangle$,

$$\begin{aligned} & \hat{\mathbf{S}}[\alpha_1|1\rangle + \alpha_2|2\rangle + \dots + \alpha_a|a\rangle + \dots + \alpha_b|b\rangle + \dots + \alpha_t|t\rangle] \\ & \equiv \alpha_1|1\rangle + \alpha_2|2\rangle + \dots + \alpha_b|a\rangle + \dots + \alpha_a|b\rangle + \dots + \alpha_t|t\rangle. \end{aligned}$$

The general procedure for finding which swaps to perform is to set up the two states to be swapped, and a path of states between them where each

neighboring state only differ in the value of one qubit. The reason why we have to swap through all neighboring states is that by using the conditional X gate on two states, we need all the other qubits to have the same value. See the above example in eq. 3.23.

In the general case when we wish to swap the coefficients of the two states $|g_1\rangle$ and $|g_m\rangle$, we do the swap operation on the states $|g_1\rangle$ and $|g_2\rangle$ through $|g_{m-1}\rangle$. Then we can apply the conditional single-qubit operation \tilde{U} on the $|g_{m-1}\rangle$ and $|g_m\rangle$ states. Finally we swap the coefficients back. The total procedure is this, where $C\tilde{U}$ is the conditional \tilde{U} operator,

$$U = \hat{\mathbf{S}}(g_2, g_1) \hat{\mathbf{S}}(g_3, g_2) \cdots \hat{\mathbf{S}}(g_{m-1}, g_{m-2}) \quad (3.29)$$

$$\times C\tilde{U} \quad (3.30)$$

$$\times \hat{\mathbf{S}}(g_{m-2}, g_{m-1}) \cdots \hat{\mathbf{S}}(g_2, g_3) \hat{\mathbf{S}}(g_1, g_2). \quad (3.31)$$

Making a gate that is conditional on more than one qubit can be done by combining *CNOT* gates and single-qubit operations, see [16] page 182. So now we have come to the point where we can construct all unitary operators as quantum circuits consisting of *CNOT* gates and single-qubit gates.

An example of this will be shown in section 4.3, where we go through the construction of a quantum circuit in detail.

3.3 Fourier transform

One of the most useful mathematical procedures in physics is the Fourier transform. There are a number of problems which can be solved by Fourier transforming equations and fields. In this thesis we use the inverse Fourier transform as part of an algorithm to calculate the eigenvalues of the time evolution operator.

3.3.1 Quantum Fourier transform

The discrete Fourier transform takes a set of complex numbers x_0, \dots, x_{N-1} to another set of complex numbers y_0, \dots, y_{N-1} , defined by

$$y_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{(i2\pi/N)jk}. \quad (3.32)$$

In the quantum case, we wish to Fourier transform states, and so define the Fourier transform on an orthonormal basis set $|0\rangle, \dots, |N-1\rangle$ to be a linear operator yielding

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{(i2\pi/N)jk} |k\rangle. \quad (3.33)$$

Here the state $|j\rangle$ and the number j are given in terms of the computational basis explained in section 2.2.1, using the enumeration tensor-product notation found in eq. (2.23).

The connection between the quantum and the non-quantum transforms is that the complex amplitudes of the state after transformation are the Fourier transformed sequence of numbers, x_j , of the number j , using the definition in eq. (3.32). If we have t qubits, we have 2^t different states and $N = 2^t$. On a superposition of states the Fourier transform may be written as

$$\begin{aligned} \sum_{j=0}^{N-1} x_j |j\rangle &\rightarrow \sum_{j=0}^{N-1} x_j \left[\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{(i2\pi/N)jk} |k\rangle \right] \\ &= \sum_{k=0}^{N-1} \left[\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{(i2\pi/N)jk} x_j |k\rangle \right] \\ &= \sum_{k=0}^{N-1} y_k |k\rangle, \end{aligned} \quad (3.34)$$

where x_j and y_k are the same as in eq. (3.32).

Binary numbers

Before moving on, let's examine how we represent an integer using binary expansion, where all digits are either zero or one,

$$\phi = \phi_1 2^{N-1} + \phi_2 2^{N-2} + \cdots + \phi_N 2^{N-N} = \sum_{i=1}^N \phi_i 2^{N-i}, \quad (3.35)$$

in this case $\phi_i = 0$ or 1 .

Now we can write any integer ϕ as $\phi_1 \phi_2 \cdots \phi_N$, with sufficiently large N . The abbreviation $\phi_1 \phi_2 \cdots \phi_N$ is the binary version of the abbreviation 42, which really is $4 \cdot 10^1 + 2 \cdot 10^0$.

The binary fraction is defined as

$$j_l/2 + j_{l+1}/4 + \cdots + j_m/2^{m-l+1} \equiv 0.j_l j_{l+1} \cdots j_m. \quad (3.36)$$

If ϕ is less than 2^N , we can write

$$\begin{aligned}
 \phi/2^N &= \phi_1 2^{(N-N)-1} + \phi_2 2^{(N-N)-2} + \dots + \phi_N 2^{(N-N)-N} \\
 &= \phi_1 2^{-1} + \phi_2 2^{-2} + \dots + \phi_{N-1} 2^{-(N-1)} + \phi_N 2^{-N} \\
 &= 0.\phi_1 \phi_2 \dots \phi_N \\
 &= \sum_{i=1}^N \phi_i 2^{-i}.
 \end{aligned} \tag{3.37}$$

Phase and integers

If $\phi = n + \theta$, where n is a positive integer and $\theta < 1$, we have

$$\begin{aligned}
 e^{2\pi i \phi} &= e^{2\pi i n} e^{2\pi i \theta} = (\cos 2\pi n + i \sin 2\pi n)(\cos 2\pi \theta + i \sin 2\pi \theta) \\
 &= \cos 2\pi \theta + i \sin 2\pi \theta \\
 &= e^{2\pi i \theta}.
 \end{aligned} \tag{3.38}$$

A phase of $\phi/2^t$ gives a factor of $\exp(2\pi i \phi/2^t) = \exp(2\pi i 0.\phi_1 \phi_1 \dots \phi_N)$, if we now have a phase of $2^k \phi/2^t = \phi_1 \dots \phi_k \cdot \phi_{k+1} \dots \phi_N$ instead, the factor is

$$e^{2\pi i 2^k 0.\phi_1 \phi_1 \dots \phi_N} = e^{2\pi i \phi_1 \dots \phi_k \cdot \phi_{k+1} \dots \phi_N} = e^{2\pi i 0.\phi_{k+1} \dots \phi_N}, \tag{3.39}$$

here $\phi_1 \dots \phi_k \cdot \phi_{k+1} \dots \phi_N > 0$ and the lower dot between ϕ_k and ϕ_{k+1} denotes where the fraction elements begin.

If we have a state with $\phi = n + \theta$, and we use the phase estimation algorithm (which determines the phase), we can only extract the θ part. We will make use of this in constructing a quantum Fourier transform.

Fourier transform as tensor products

We have a system consisting of t qubits, with 2^t basis states $|j\rangle$, j can be represented as a binary number, $j = j_1 j_2 \dots j_t$. We use both the enumeration of basis states described in eq. (2.23), and the tensor product notation of eq. (2.21). The Fourier transform of $|j\rangle$ can be written as

$$\begin{aligned}
 \frac{1}{2^{t/2}} \sum_{k=0}^{2^t-1} e^{(i2\pi/2^t)jk} |k\rangle &= \\
 \frac{(|0\rangle + e^{2\pi i 0.j_t} |1\rangle)(|0\rangle + e^{2\pi i 0.j_{t-1} j_t} |1\rangle) \dots (|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_{t-1} j_t} |1\rangle)}{2^{t/2}}.
 \end{aligned} \tag{3.40}$$

The proof follows from the definition

$$|j\rangle \rightarrow \frac{1}{\sqrt{2^t}} \sum_{k=0}^{2^t-1} e^{(i2\pi/2^t)jk} |k\rangle$$

$$= \frac{1}{\sqrt{2^t}} \sum_{k_1=0}^1 \cdots \sum_{k_t=0}^1 e^{(i2\pi)j(\sum_{l=1}^t k_l 2^{-l})} |k_1 \cdots k_t\rangle \quad (3.41)$$

$$= \frac{1}{\sqrt{2^t}} \sum_{k_1=0}^1 \cdots \sum_{k_t=0}^1 \bigotimes_{l=1}^t e^{(i2\pi)jk_l 2^{-l}} |k_l\rangle \quad (3.42)$$

$$= \frac{1}{\sqrt{2^t}} \bigotimes_{l=1}^t \left[\sum_{k_l=0}^1 e^{(i2\pi)jk_l 2^{-l}} |k_l\rangle \right] \quad (3.43)$$

$$= \frac{1}{\sqrt{2^t}} \bigotimes_{l=1}^t \left[|0\rangle + e^{(i2\pi)j2^{-l}} |1\rangle \right] \quad (3.44)$$

$$= \frac{(|0\rangle + e^{2\pi i 0 \cdot j_t} |1\rangle)(|0\rangle + e^{2\pi i 0 \cdot j_{t-1}} |1\rangle) \cdots (|0\rangle + e^{2\pi i 0 \cdot j_1} |1\rangle)}{2^{t/2}}. \quad (3.45)$$

This expression enables us to find a circuit implementing the quantum Fourier transform, and is what we later need to show that the phase estimation algorithm gives us the Fourier transform of the phase.

Example of Fourier transform

This is an example of a Fourier transform of a two qubit basis state using both methods shown above.

We will transform the state $|0\rangle|1\rangle = |j\rangle$, here $j = j_1 j_2$, and $j = 1$ or in binary numbers $j_1 = 0$ and $j_2 = 1$. We use two representations of tensor products from eqs. (2.22) and (2.23). For this state $t = 2$, and $2^t = 4$.

First we use the definition, eq. (3.33),

$$\begin{aligned} |0\rangle|1\rangle &\rightarrow \frac{1}{\sqrt{2}} (e^{(i2\pi/4)0j} |0\rangle + e^{(i2\pi/4)1j} |1\rangle + e^{(i2\pi/4)2j} |2\rangle + e^{(i2\pi/4)3j} |3\rangle) \quad (3.46) \\ &= \frac{1}{\sqrt{2}} (|0\rangle + e^{i\pi/2} |1\rangle + e^{i\pi} |2\rangle + e^{i3\pi/2} |3\rangle) \\ &= \frac{1}{\sqrt{2}} (|0\rangle + i|1\rangle - |2\rangle - i|3\rangle) \\ &= \frac{1}{\sqrt{2}} (|0\rangle|0\rangle + i|0\rangle|1\rangle - |1\rangle|0\rangle - i|1\rangle|1\rangle). \end{aligned}$$

Next we use the product representation,

$$\begin{aligned}
|0\rangle|1\rangle &\rightarrow \frac{(|0\rangle + e^{2\pi i 0.j_2}|1\rangle)(|0\rangle + e^{2\pi i 0.j_1 j_2}|1\rangle)}{\sqrt{2}} \\
&= \frac{(|0\rangle + e^{2\pi i 0.1}|1\rangle)(|0\rangle + e^{2\pi i 0.01}|1\rangle)}{\sqrt{2}} \\
&= \frac{(|0\rangle + e^{2\pi i/2}|1\rangle)(|0\rangle + e^{2\pi i/4}|1\rangle)}{\sqrt{2}} \\
&= \frac{1}{\sqrt{2}} (|0\rangle|0\rangle + e^{2\pi i/4}|0\rangle|1\rangle + e^{2\pi i/2}|1\rangle|0\rangle + e^{2\pi i/2}e^{2\pi i/4}|1\rangle|1\rangle) \\
&= \frac{1}{\sqrt{2}} (|0\rangle|0\rangle + i|0\rangle|1\rangle - |1\rangle|0\rangle - i|1\rangle|1\rangle).
\end{aligned} \tag{3.47}$$

3.3.2 Implementation of Fourier transform

The key to the quantum Fourier transform is the tensor-product representation of the transform seen in eq. (3.40). Using Hadamard transformations on the qubits along with a specific conditional gate, called R_k , we can transform a basis state into it's reversed order Fourier transform as eq. (3.40). We will show this step by step.

Swapping the qubit order

As we will show, the result of the Hadamard and conditional R_k gates on a basis state $|j\rangle$ is this

$$|j\rangle \rightarrow \frac{(|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_{t-1} j_t}|1\rangle) \dots (|0\rangle + e^{2\pi i 0.j_t-1 j_t}|1\rangle)(|0\rangle + e^{2\pi i 0.j_t}|1\rangle)}{2^{t/2}}, \tag{3.48}$$

which is the Fourier transform of $|j\rangle$ with reversed qubit order. We could either reverse the qubit order before or after applying the Hadamard and conditional R_k gates, but choose to do it before applying the rest of the Fourier transform circuit.

Now we are ready to implement the quantum Fourier transform as a circuit of quantum gates using one-qubit gates and $CNOT$ gates. Our input state is $|j\rangle = |j_1 j_2 \dots j_{t-1} j_t\rangle$, where $j_i = 0$ or 1 using the binary enumeration from eq. (2.22), but we swap the order of the qubits to $|j'\rangle = |j_t j_{t-1} \dots j_2 j_1\rangle$ for the Fourier transform circuit, using swap gates that are not included in the diagram of the circuit, fig. 3.3. The swap circuit between two qubits is shown in fig. 3.1.

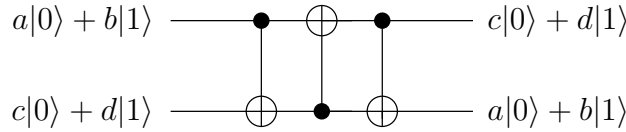


Figure 3.1: Circuit swapping the states of two qubits

Hadamard gates

After the swap gates we apply the Hadamard gate to the last qubit which is now in the state $|j_1\rangle$,

$$|j_1\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle), \quad (3.49)$$

with plus for $j_1 = 0$, and minus for $j_1 = 1$. See the description of the Hadamard gate in section 2.4. This is the same as

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0 \cdot j_1} |1\rangle). \quad (3.50)$$

Because for $j_1 = 1$, we get $0 \cdot j_1$ is $1/2 \rightarrow e^{2\pi i 0 \cdot j_1} = e^{\pi i} = -1$, and for $j_1 = 0 \rightarrow e^{2\pi i 0 \cdot j_1} = e^0 = 1$.

R_k gates

We define the R_k gate below, given as a matrix in the computational basis,

$$R_k \equiv \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i / 2^k} \end{pmatrix} \quad (3.51)$$

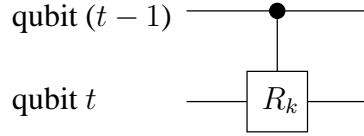
We apply an R_2 gate to qubit t conditional on qubit $t - 1$ being in the $|1\rangle$ state, i.e. a conditional R_2 gate with qubit t as the target and qubit $t - 1$ as the control qubit, see section 2.3.2 on conditional operations. The conditional R_k gate is shown in fig. 3.2 (as we show in fig. 2.3, qubit t is now the qubit to the right in the tensor product).

If $j_2 = 0$, R_2 is not applied, and qubit t is in the state

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0 \cdot j_1} |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0 \cdot j_1 j_2} |1\rangle), \quad (3.52)$$

since then

$$e^{2\pi i 0 \cdot j_1 j_2} = e^{2\pi i 0 \cdot j_1 0} = e^{2\pi i 0 \cdot j_1}. \quad (3.53)$$

Figure 3.2: The R_k gate

If, however, j_2 is 1 we apply R_2 , yielding the prefactor $e^{2\pi i/2^2} = e^{2\pi i \cdot 0 \cdot j_2}$ to $|1\rangle$,

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1} e^{2\pi i \cdot 0 \cdot j_2} |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1 \cdot j_2} |1\rangle). \quad (3.54)$$

We see the results match and we have a general expression for the effect of the conditional R_k gate

$$|j_2\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1} |1\rangle) \rightarrow |j_2\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1 \cdot j_2} |1\rangle). \quad (3.55)$$

If we apply the conditional R_k gate on qubit t with the $(t - k + 1)^{th}$ qubit as control qubit, we see that instead of $j_2/2^2$, we add $j_k/2^k$ in the exponent before the $|1\rangle$ state in qubit t . Doing this for all qubits from $t - 1$ to 1, we obtain

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1} |1\rangle) \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_1 \cdot j_2 \cdots j_t} |1\rangle). \quad (3.56)$$

Now we repeat the process with qubit $t - 1$ instead, obtaining the state $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot j_2 \cdot j_3 \cdots j_t} |1\rangle)$ by applying the Hadamard gate and the R_k gates conditional on the preceding qubits. We do this for all the qubits except qubit one, on which we apply just the Hadamard gate. The result is what is shown in equation 3.40, and is the Fourier transform of $|j\rangle$.

In fig. 3.3 we have the quantum circuit representation of the quantum Fourier transform.

3.3.3 Inverse Fourier transform

The quantum Fourier transform is a unitary transformation, that we have proven by creating a quantum circuit consisting of unitary operations. This means that the inverse Fourier transform is also the hermitian conjugate of the Fourier transform, see section 2.1.13 on unitarity. As the Fourier

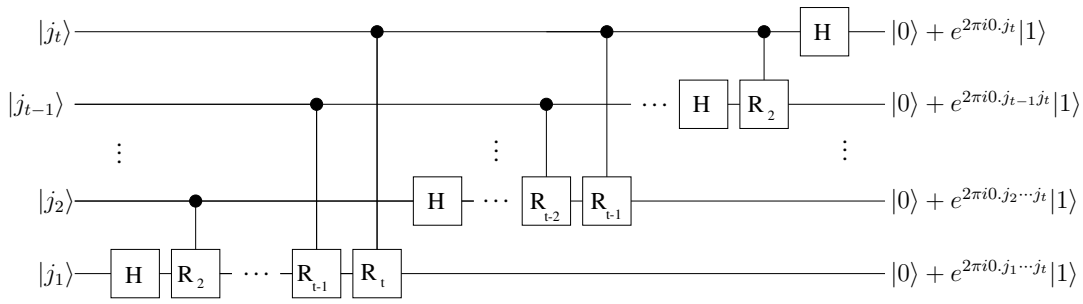


Figure 3.3: Fourier transform circuit without swap gates

transform is a product of operators, the inverse is the reverse order product with all operators hermitian conjugated,

$$FT = \prod_{i=1}^m \hat{\Omega}_i \Rightarrow (FT)^{-1} = (FT)^\dagger = \prod_{i=m}^1 \hat{\Omega}_i^\dagger. \quad (3.57)$$

Since the Hadamard operation is hermitian and unitary

$$H^2 = \hat{1} \Rightarrow H^{-1} = H,$$

and R_k is unitary, $R_k^\dagger = R_k^{-1}$, we obtain the gate shown in fig. 3.4.

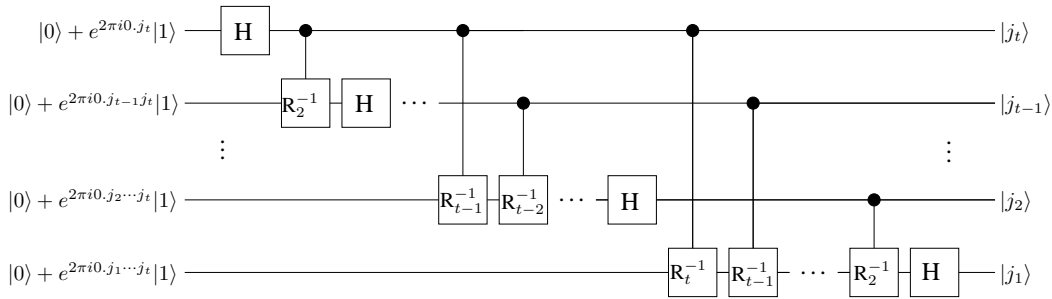


Figure 3.4: Inverse Fourier transform circuit without swap gates

As we see in fig. 3.4 when compared to eq. (3.40), the order of the qubits in the output state has to be reversed to $|j_1 j_2 \dots j_{t-1} j_t\rangle$, the same as the Fourier transform. The swap gates have not been shown in the figure and we do not need them in this thesis. We do employ the inverse Fourier transform,

but we measure the states right after having performed the inverse transform and so can swap the results mathematically.

As an example, if $|001\rangle$ is the measurement of an inversely Fourier transformed state (using the tensor product from eq. (2.22)), then this would mean that the first two qubits, from above, are in the $|0\rangle$ state, and the third is in the $|1\rangle$ state. Having reversed the order of the qubits, the correct result should be that the first qubit is in the $|1\rangle$ state, and the second and third in the $|0\rangle$ state. This is achieved by reversing the binary digits of the binary measurement result, 001 becomes 100 and so on.

In our simulation we use the inverse Fourier transform just before measuring, and therefore we need not include the swapping gates, but the correct circuit should have swapping gates at the end.

3.4 Phase estimation

The quantum simulation algorithm of section 3.1.1 is incomplete as an algorithm to extract the eigenvalues of a Hamiltonian, it merely changes the phase of each eigenstate by using the time evolution operator,

$$\begin{aligned}\hat{U}|\alpha\rangle &= e^{-i\hat{H}\Delta t}|\alpha\rangle \\ &= \sum_k e^{-iE_k\Delta t}\alpha_k|k\rangle,\end{aligned}\tag{3.58}$$

where $|\alpha\rangle$ is a superposition of eigenstates $|k\rangle$ of \hat{H} ,

$$|\alpha\rangle = \sum_k \alpha_k|k\rangle.\tag{3.59}$$

This means that to find E_k we must estimate the phase of each eigenstate, the algorithm we use to do that is shown below.

Description

The phase estimation algorithm is used to find the eigenvalues of a unitary operator U , with eigenvectors $|k\rangle$ and corresponding eigenvalues $e^{2\pi i\phi_k}$.

In the case of the time evolution operator $e^{-i\hat{H}\Delta t}$, we have the eigenstates $|k\rangle$, each with the eigenvalues $e^{-iE_k\Delta t}$. The phase estimation algorithm finds

$$\phi_k \equiv -E_k\Delta t/2\pi.\tag{3.60}$$

In the phase estimation algorithm we use what we call work qubits. A work qubit is a qubit that is not used directly in simulating the state of the

system, but is used to store information. When we describe the total system by a tensor product or by circuit diagrams, the work qubits are always first. The total state of the work and simulation qubits described by a tensor product is

$$|system\rangle = \sum_i |work\rangle_i \otimes |simulation\rangle_i, \quad (3.61)$$

where we have used a superposition to illustrate the fact that there can be entanglement between the work and simulation qubits.

The algorithm uses t work qubits in addition to the s simulation qubits representing the eigenstates of the operator. The work qubits will, at the end of the algorithm, give a binary representation of ϕ_k , which we extract by measurement.

Input

What we need to perform the algorithm is the unitary operator \hat{U} as a set of gates (see section 4.4).

We need s simulation qubits whose state, $|i\rangle$, can be written as a superposition of eigenstates of U ,

$$|i\rangle = \sum_k \langle k|i\rangle |k\rangle \equiv \sum_k c_k |k\rangle. \quad (3.62)$$

We need t work qubits, each one initialized to $|0\rangle$, giving us the total input state

$$|initial\rangle = \underbrace{|0\rangle \otimes |0\rangle \cdots \otimes |0\rangle}_{work} \otimes \underbrace{\sum_k c_k |k\rangle}_{simulation}, \quad (3.63)$$

where we have used the enumeration representation of the tensor product of basis states, see eq. (2.23), in the simulation part of the input state.

3.4.1 The algorithm

The algorithm uses first the Hadamard transformation on each work qubit to transform it into the $(|0\rangle + |1\rangle)/\sqrt{2}$ state, then it uses the U operator on the simulation qubits conditional on the work qubits. This extracts part of the phase because of eq. (3.38), similar to the effect of the conditional R_k gate in the Fourier transform, see eq. (3.55). Then finally we use the inverse Fourier transform to extract the eigenvalues.

Hadamard

First we apply the Hadamard gate to each work qubit,

$$|0\rangle|0\rangle \cdots |0\rangle \rightarrow \frac{1}{2^{t/2}}(|0\rangle + |1\rangle) \cdots (|0\rangle + |1\rangle), \quad (3.64)$$

where we have skipped the \otimes signs as in eq. (2.21).

Controlled U^{2^j}

The next step is to use a series of controlled U^{2^j} gates to extract the eigenvalues and store them in the work qubits. The controlled U^{2^j} operation is $e^{-i\hat{\mathbf{H}}\Delta t 2^j}$ with eigenvalues $e^{2\pi i \phi_k 2^j}$. This can be constructed by changing Δt to $2^j \Delta t$, alternately performing the controlled U operation 2^j times.

Because of eq. (3.38) we are only able to extract the non-integer part of ϕ , that means $\phi \leq 1$. We skip using the k subscript on the eigenvalue ϕ for now. This means it can be approximated as a binary fraction $\phi = 0.\phi_1\phi_2 \cdots \phi_t$, see eq. (3.37) for an example.

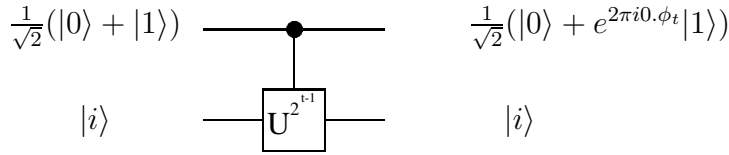
In eq. (3.65) below, we use the results from eq. (3.39)

$$e^{2\pi i 0.\phi_1 \cdots \phi_t 2^j} = e^{2\pi i \phi_{j+1} \cdots \phi_t}.$$

The action of a controlled U^{2^j} operation on $|i\rangle$, the simulation qubits, with the $(t-j)^{th}$ work qubit as the control qubit is shown in eq. (3.65) (disregarding the other work qubits for now).

$$\begin{aligned} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |i\rangle &= \sum_k \frac{c_k}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |k\rangle \\ &\rightarrow \sum_k \frac{c_k}{\sqrt{2}}(|0\rangle \otimes |k\rangle + |1\rangle \otimes \hat{\mathbf{U}}^{2^j} |k\rangle) \\ &= \sum_k \frac{c_k}{\sqrt{2}}(|0\rangle \otimes |k\rangle + e^{2\pi i 0.\phi_1 \cdots \phi_t 2^j} |1\rangle \otimes |k\rangle) \\ &= \sum_k \frac{c_k}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.\phi_1 \cdots \phi_t 2^j} |1\rangle) \otimes |k\rangle \\ &= \sum_k \frac{c_k}{\sqrt{2}}(|0\rangle + e^{2\pi i \phi_{j+1} \cdots \phi_t} |1\rangle) \otimes |k\rangle \end{aligned} \quad (3.65)$$

Here and in the following we will not tag the ϕ 's with k 's to represent each eigenvalue, this is just to remind the reader that all eigenstates can have different eigenvalues. We will tag them later on when we have need for it.

Figure 3.5: The conditional $U^{2^{t-1}}$ gate

In fig. 3.5 we have depicted the same operation using quantum gates. A controlled $U^{2^{t-1}}$ operates on the target state $|i\rangle$, with the work qubit as the control qubit. The state of the work qubit then changes to $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.\phi_t}|1\rangle)$.

We have now extracted $0.\phi_t$, next we do a controlled $U^{2^{t-2}}$ operation with the second work qubit as control, and then extract $0.\phi_{t-1}\phi_t$. We continue to do this for all the work qubits, changing the j of the U^{2^j} operation. The circuit of the phase estimation algorithm so far is shown in fig. 3.7.

Multiple qubit conditional operations

In the phase estimation algorithm we use the conditional U^{2^j} operation, where U^{2^j} is an operation consisting of several multi-qubit gates. To create a conditional multi-qubit gate we merely make a conditional gate of every single- or two-qubit gate in U^{2^j} . If for example the operation A consists of gate B on qubit two and gate C on qubit three, then the conditional A operation with qubit one as control is seen in fig. 3.6.

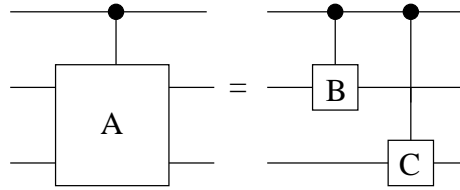


Figure 3.6: Multiple qubit conditional operation

The circuit

Doing this for all the work qubits sequentially (the order in which we do this is irrelevant, since the eigenvalues are the same), we get the following result (we start with work qubit one on the left),

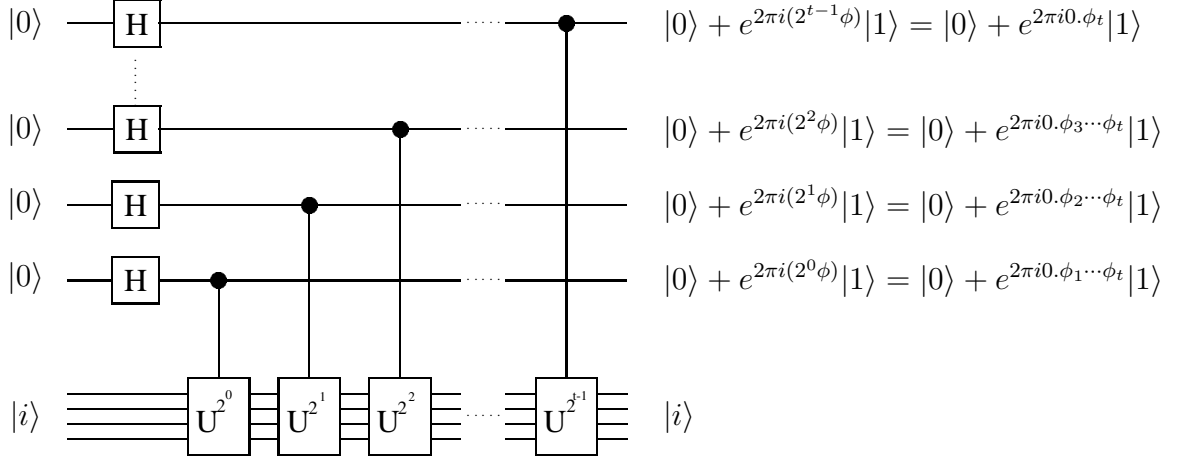


Figure 3.7: Phase estimation circuit

$$\sum_k \frac{c_k}{2^{t/2}} (|0\rangle + e^{2\pi i 0.\phi_t} |1\rangle) (|0\rangle + e^{2\pi i 0.\phi_{t-1}\phi_t} |1\rangle) \cdots (|0\rangle + e^{2\pi i 0.\phi_1\phi_2\cdots\phi_{t-1}\phi_t} |1\rangle) \otimes |k\rangle \quad (3.66)$$

When compared to the Fourier transform in eq. (3.40), we see that $\phi = j/2^t$, where j now is the Fourier transformed number from eq. (3.40). This is because $\phi \leq 1$ and we have used $\phi = 0.\phi_1\phi_2\cdots\phi_t$, whereas $j = j_1j_2\cdots j_t$. We use the definition of the Fourier transform from eq. (3.33), and get as a result

$$\sum_k \frac{c_k}{2^{t/2}} \sum_l e^{2\pi i \phi l} |l\rangle \otimes |k\rangle. \quad (3.67)$$

The state of the work qubits for each term in the sum over k is the Fourier transform of $|\phi 2^t\rangle = |\phi_1\phi_2\cdots\phi_{t-1}\phi_t\rangle$, using both the binary and integer enumeration of basis states from eqs. (2.22) and (2.23). By applying the inverse Fourier transform on the work qubits, we will have a superposition of states with each state having the same probability $|c_k|^2$ as the eigenstates of U . For each of these states, the enumeration of the work qubits will be a binary representation of $\phi_k 2^t$, where the k subscript denotes the ϕ of state $|k\rangle$. The superposition follows from the linearity of the Fourier transform ,

$$\sum_{j=0}^{N-1} a_j |j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} a_j y_k |k\rangle,$$

and we get

$$\sum_k \frac{c_k}{2^{t/2}} \sum_l e^{2\pi i \phi l} |l\rangle \otimes |k\rangle \rightarrow \sum_k c_k |\phi_k 2^t\rangle \otimes |k\rangle. \quad (3.68)$$

Having used the inverse Fourier transform we measure the work qubits and find ϕ_k with probability $|c_k|^2$. In fig. 5.6 we have found the probability spectrum $|c_k|^2$ for a system using the phase estimation algorithm, as we can see the probability maxima show us the eigenvalues of U , ϕ_k .

3.5 Measurements on the system

To get a result from a quantum computer we must measure the state, the same way we must read information from the memory of a classical computer. What a measurement is physically will differ for each realization of the quantum computer. We might measure whether electrons have spin “up” or “down”, we might measure whether an atom is in the ground state or excited etc.

Quantum mechanics is probabilistic, we measure results with a probability distribution determined by the state of the system. To find the probability distribution of eigenstates of a system we must perform a number of measurements on identically prepared states, we cannot perform the same measurement twice due to the collapse of the state, see section 2.1.12. This cannot be solved by copying the state once we have produced it, and then measuring each copy; the no-cloning theorem [23] says that it is generally impossible to clone a quantum state. We have to prepare each state, perform a set of operations, every time before we measure on the state. Some algorithms though, like Shor’s factoring algorithm, does not need the total probability distribution, and is not that hampered by this serious limitation of quantum computers.

3.5.1 The projection

The state of the system after measurement on parts of it, must still be normalized to one. If we consider the operator $\widehat{\mathbf{M}} = |m\rangle\langle m|$, which projects the state $|i\rangle$ onto the state $|m\rangle$, to ensure normalization the projection must be

$$\frac{\widehat{\mathbf{M}}|i\rangle}{\sqrt{\langle i|\widehat{\mathbf{M}}^\dagger\widehat{\mathbf{M}}|i\rangle}} = \frac{c_M|m\rangle}{\sqrt{\langle i|\widehat{\mathbf{M}}^\dagger\widehat{\mathbf{M}}|i\rangle}}. \quad (3.69)$$

3.5.2 Measuring a single qubit

A measurement on a quantum computer will consist of measurements on single qubits, the state of every qubit will give us one state of the total system.

There are two possible results when measuring a qubit, it can be projected into either the $|0\rangle$ state or the $|1\rangle$ state. The probability of each is the amplitude squared. For the general unentangled state $|\psi\rangle = a|0\rangle + b|1\rangle$, the probability of measuring state $|0\rangle$ is $|a|^2$, and $|b|^2$ for measuring $|1\rangle$.

The projection operator

To find an operator that projects only one qubit into a desired state, we must use tensor products. If we have an operation on a single qubit, that means we perform the unity operation on the rest of the qubits, see section 5.1.2.

For a one qubit operation, $\widehat{\mathbf{M}}$, operating on the j^{th} qubit of a t qubit system, tensor products will give an $\widehat{\mathbf{M}}_{tot}$ for the whole system. Here indices mark which qubits the operators work on, and $\widehat{\mathbf{1}}^i = \bigotimes_{k=1}^i \widehat{\mathbf{1}}$. That is, a unity operation on i qubits,

$$\widehat{\mathbf{M}}_{tot} = \widehat{\mathbf{1}}^{j-1} \otimes \widehat{\mathbf{M}}_j \otimes \widehat{\mathbf{1}}^{t-j}.$$

Using this method of constructing operators, we can construct projection operators to measure all states.

3.5.3 Measurement in phase estimation

The last part of the phase estimation algorithm is the measurement. The system is in this state after the controlled U operations and inverse Fourier transform

$$\sum_k c_k |\phi_k 2^t\rangle \otimes |k\rangle,$$

see eq. (3.68).

As we can see in fig. 5.6, the probability distribution gives us ϕ_k by the position of the maximas. There are 2^t possible values, ϕ , for the enumeration of the work qubit states. In the ideal case, when all ϕ_k 's can be expressed as binary numbers with as many bits as there are work qubits, and our simulation of the U operator is perfect, the probability will be zero for all $\phi \neq \phi_k$. That is, we will never measure a value of the work qubits that is not an eigenvalue of U .

3.5.4 Simulating measurement

How can we simulate the process of measurement? When simulating the system on our classical computers we know the complete state, and we can

calculate the probability spectrum $|c_k|^2$ from the state in eq. (3.68). Since the total state is a superposition of work qubit basis states tensor multiplied to eigenstates of U , we can project the total state into one of these terms. We will show below that we can use this to calculate $|c_k|^2$. Then we can simulate the measurement using a “random” number generator.

How can we find $|c_k|^2$ when using the column matrix representation of the system state? By constructing a projection operator on the work qubits that is the outer product of the work qubit basis state corresponding to c_k , $|\phi_k 2^t\rangle\langle\phi_k 2^t|$, applying it to the system and calculating the inner product of the projection

$$\left(|\phi_k 2^t\rangle\langle\phi_k 2^t| \otimes \hat{\mathbf{1}}\right) \left(\sum_i c_i |\phi_i 2^t\rangle \otimes |i\rangle\right) = c_k |\phi_k 2^t\rangle \otimes |k\rangle.$$

By not normalizing the projected state the inner product gives us $|c_k|^2$,

$$|c_k |\phi_k 2^t\rangle \otimes |k\rangle|^2 = |c_k|^2 \langle\phi_k 2^t|\phi_k 2^t\rangle \langle k|k\rangle = |c_k|^2. \quad (3.70)$$

If it were normalized it would be the projection seen in eq. (3.69).

Random choosing algorithm

Having the probability spectrum, we now need an algorithm to simulate the measurement.

- First we pick a random number, $i \in [0, 1]$.
- Then we start with the lowest eigenvalue, $k = 0$. If $|c_0|^2 > i$ we set the measured eigenvalue to be ϕ_0 . If not, move to the next step.
- If $\sum_{k=0}^f |c_k|^2 = |c_0|^2 + |c_1|^2 > i$ then the measured eigenvalue is set to be ϕ_1 . We continue in this fashion, adding the probability of the next eigenvalue, until $\sum_{k=0}^f |c_k|^2 > i$. When it is, the measured eigenvalue is set to be ϕ_f .
- The measurement is done.

The code we have used for this algorithm can be found in section B.4. An example is shown in fig. 3.8 where we have simulated two random measurements on the state

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix},$$

where each result has the probability 0.25. If the random number generated is 0.375 we measure the $|0\rangle|1\rangle$ state, if the number is 0.93 we measure the $|1\rangle|1\rangle$ state.

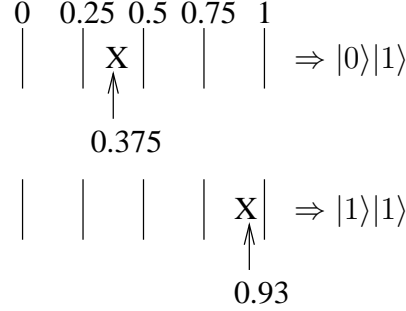


Figure 3.8: Simulating measurement

3.5.5 Finding E_k

The measured work qubits give us the energy corresponding to the enumeration of the basis state measured,

$$E_k = -(\phi_k 2^t) 2\pi / (2^t \Delta t), \quad (3.71)$$

where $\phi_k 2^t \in [0, 2^t - 1]$ is the enumeration. When using an imperfect simulation, which will always be the case, we will measure eigenvalues which are not correct, but the methods we develop here and in chapter 5 will find the correct answers from an imperfect simulation.

After performing the measurements and obtaining the probability spectrum, we then plot the probability as a function of the energy. One set of probabilities gives us a value for each eigenvalue of the system that is within the boundaries of the simulation. Several calculations gives us a set of values for each eigenvalue. Then we can calculate the mean value and the mean standard deviation, and present a value with an estimated error. See section 5.3 where the result of one such simulation is shown.

3.5.6 Phase estimation and negative ϕ

The phase-estimation algorithm is designed to find the eigenvalues of \hat{U} , $2\pi\phi_k = -E_k \Delta t$ in our case. The final result is a state where the work qubits

are in the state $|\phi_k 2^t\rangle$, where $\phi_k 2^t$ is a binary number. If $\phi < 0$ this will not work, therefore we must alter the Hamiltonian so that all E_k are negative,

$$\hat{\mathbf{H}}' \equiv \hat{\mathbf{H}} - E_{max} \quad (3.72)$$

$$\Rightarrow \hat{\mathbf{H}}'|k\rangle = (\hat{\mathbf{H}} - E_{max})|k\rangle = (E_k - E_{max})|k\rangle. \quad (3.73)$$

3.5.7 Information required for simulating

Aside from the Hamiltonian and the size of the system, we also need to know two other things to obtain correct results from the simulation.

We need an upper bound, the maximum eigenvalue, so that we can create a Hamiltonian with all negative eigenvalues, see section 3.5.6.

To decide on a proper time interval, Δt , we need a lower bound, so that $-E_k \Delta t < 2\pi$. If it is larger than this, $\phi > 1$ and we will not find the correct eigenvalues due to the properties of the exponential function shown in eq. (3.38).

This means that to find the whole energy spectrum, all the eigenvalues, we need an upper and lower bound on the eigenvalues of $\hat{\mathbf{H}}$. This can be found by performing simulations and adjusting E_{max} and Δt till we get satisfactory results.

3.5.8 Analyzing parts of the energy spectrum

By adjusting the upper energy boundary, E_{max} see section 3.5.6, and the time interval of the time evolution operator, Δt , see eq. (3.60), we can analyze what part of the energy spectrum we wish.

The different energies we can measure are given by eq. (3.60) and eq. (3.73),

$$E_k = E_{measure} = E_{max} - \phi_k \frac{2\pi}{\Delta t}. \quad (3.74)$$

The maximal value is E_{max} , and the minimal value is given by eq. (3.39) and the number of work qubit basis states, 2^t ,

$$\min\{E_{measure}\} = E_{max} - \frac{2^t - 1}{2^t} \frac{2\pi}{\Delta t}. \quad (3.75)$$

By adjusting E_{max} and Δt we can choose where to look for eigenvalues of the time evolution operator. We can, for example, use it to look for the ground state energy, without performing as many calculations as we would need to find the entire energy spectrum.

3.5.9 Initializing qubits

The work qubits must be initialized to be in the $|0\rangle$ state, as we see in section 3.4. The simulation qubits, however, can be arbitrarily initialized to suit the purpose of the simulation. If the simulation qubits start one simulation in a random state, then after the system has gone through the whole quantum circuit and the work qubits have been measured upon, the simulation qubits will be in an eigenstate of the time evolution operator, see eq. (3.68). This means that if we do the procedure once more without changing the state of the simulation qubits, we will measure the same value. One of the requirements of a quantum computer is that initializing qubits to arbitrary states must be possible, so we can initialize the simulation qubits to the same state each time. A large set of measurements will then give us the probabilities of each eigenstate and thereby the energies of the system. It may be that it is less time consuming to initialize the simulation qubits to a new random state before each simulation. In that case there is no single set of coefficients, $\{|c_k|^2\}$, the random measurements are drawn from. The total probability spectrum for the measurements will be the average probability spectrum of all states,

$$\{|c_k|^2\} = \frac{1}{\sqrt{2^s}}\{1, 1, 1, \dots\}, \quad (3.76)$$

here s is the number of simulation qubits. In this case, the maximas of our probability distribution graphs will be of equal height, see fig. 5.6. Our classical calculations are not slowed down by using the same simulation input state for each simulation, on the contrary it is far more effective calculating $\{|c_k|^2\}$ for one state and use that to calculate the energies simulating several measurements. If we had to perform the phase estimation algorithm once for each measurement, the running time would increase by as many times we needed to measure to get a good result.

3.5.10 The circuit

In this chapter we have developed methods to analyze a quantum mechanical system, a recipe that can be used to design a quantum computer circuit which will obtain the desired results. In fig. 3.9 we see a schematic circuit of the phase estimation algorithm.

The measurements are illustrated by the dial at the end of the work qubit line.

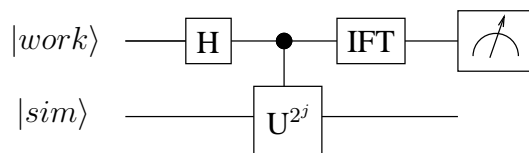


Figure 3.9: Simulation circuit

Chapter 4

The Model

In this chapter we develop the quantum computer circuits necessary for simulating the Heisenberg model, a one dimensional chain of spin 1/2 particles. It is where most of our own work is done, using the recipe given in [16]. At present there are only few calculations which employ quantum computing algorithms for solving many-body problems, see for example [18] where they have both used simulations and experiments to calculate diffusion on 16 two-qubit processors.

4.1 The Heisenberg model

In order to construct a quantum circuit for solving a many-body problem, we choose in this work the one-dimensional Heisenberg model, given by the Hamiltonian

$$\hat{\mathbf{H}} = h \sum_k \sigma_k^z \pm J \sum_k \vec{\sigma}_k \cdot \vec{\sigma}_{k+1}. \quad (4.1)$$

This is a one-dimensional chain of particles, with only one degree of freedom, the spin of the particle. The energy of the system is determined by the directions of each individual spin, σ_z , and the relationships between neighboring spins, $\vec{\sigma}_k \vec{\sigma}_{k+1}$. The Ising spin model is a classical version of this problem, whereas the Hubbard model stands for the quantum mechanical analog.

In eq. (4.2) we define a new Hamiltonian to deal with the negative ϕ problem, see section 3.5.6 for more information, we subtract an upper energy bound to get only negative eigenvalues.

What we want to find are the energy levels of this system, the eigenvalues of $\hat{\mathbf{H}}$. To get the results in dimensionless units, we divide the expression by $\pm J$, and get

$$\hat{\mathbf{H}}' = \frac{\hat{\mathbf{H}}}{\pm J} = \sum_k \frac{h}{\pm J} \sigma_k^z + \sum_k \vec{\sigma}_k \cdot \vec{\sigma}_{k+1} - \frac{E_{max}}{\pm J} \equiv \sum_k h' \sigma_k^z + \sum_k \vec{\sigma}_k \cdot \vec{\sigma}_{k+1} - E'_{max}. \quad (4.2)$$

h' and E'_{max} are now dimensionless parameters.

4.1.1 Time evolution

We follow the simulation algorithm as described in chapter three. The first thing we have to do is decide what the qubits will represent. In this case it is straightforward, as the basic components of the system are two-level quantum states, just like qubits. Each qubit will represent a single spin.

The Hamiltonian we use is a sum over operations on single spins, or qubits, and operations on two neighboring qubits,

$$\hat{\mathbf{H}}' = \sum_k \left[h' \sigma_k^z + (\sigma_k^x \sigma_{k+1}^x + \sigma_k^y \sigma_{k+1}^y + \sigma_k^z \sigma_{k+1}^z) - E'_{max} \right].$$

With the general versions of eqs.(3.2) and (3.3),

$$\begin{aligned} e^{-i(A + (\sum_{i=0}^N B_i))\Delta t} &= e^{-iA\Delta t/2} e^{-i\sum_{i=0}^N B_i\Delta t} e^{-iA\Delta t/2} + O(\Delta t^3), \\ &= e^{-iA\Delta t/2} e^{-iB_0\Delta t/2} e^{-i\sum_{i=1}^N B_i\Delta t} e^{-iB_0\Delta t/2} e^{-iA\Delta t/2} + O(\Delta t^3), \end{aligned}$$

we see that we can approximate $\hat{\mathbf{U}} = \exp(-i\hat{\mathbf{H}}'\Delta t)$ as a product of operations on single qubits and neighboring pairs of qubits, see section 3.1.1. These operations can now be performed independently of the rest of $\hat{\mathbf{U}}$ as long as Δt is small. Our choice of Hamiltonian yields the following time evolution operator,

$$e^{-i\hat{\mathbf{H}}'\Delta t} = e^{-i(\sum_k [h' \sigma_k^z + (\sigma_k^x \sigma_{k+1}^x + \sigma_k^y \sigma_{k+1}^y + \sigma_k^z \sigma_{k+1}^z) - E'_{max}])\Delta t}, \quad (4.3)$$

implying that we need to explicitly evaluate the following terms in our quantum circuits,

$$\begin{aligned} &e^{-ih' \sigma_k^z \Delta t}, \\ &e^{-i\sigma_k^x \sigma_{k+1}^x \Delta t}, \\ &e^{-i\sigma_k^y \sigma_{k+1}^y \Delta t}, \\ &e^{-i\sigma_k^z \sigma_{k+1}^z \Delta t}, \\ &e^{iE'_{max} \Delta t}. \end{aligned} \quad (4.4)$$

4.2 Operators for two level unitary matrices

The series expansion of an operator $\hat{\mathbf{A}} = \exp(-i\xi\hat{\mathbf{B}}\Delta t)$ is

$$\hat{\mathbf{A}} = \sum_{j=0}^{\infty} \frac{(-i\xi\hat{\mathbf{B}}\Delta t)^j}{j!} \quad (4.5)$$

$$= \sum_{j=0}^{\infty} \frac{(-i\xi\hat{\mathbf{B}}\Delta t)^{2j}}{(2j)!} + \sum_{j=0}^{\infty} \frac{(-i\xi\hat{\mathbf{B}}\Delta t)^{2j+1}}{(2j+1)!}. \quad (4.6)$$

We can then use this feature in rewriting the exponentials of eq. (4.4). The operator $\sigma_k^i \sigma_{k+1}^i$ squared, where the Pauli matrix i operates on qubit k and $k+1$, is

$$\begin{aligned} \sigma_k^i \sigma_{k+1}^i &= \sigma_k^i \otimes \sigma_{k+1}^i \Rightarrow \\ (\sigma_k^i \sigma_{k+1}^i)^2 &= (\sigma_k^i)^2 \otimes (\sigma_{k+1}^i)^2 = \hat{\mathbf{1}}_k \otimes \hat{\mathbf{1}}_{k+1}, \end{aligned} \quad (4.7)$$

where we have used the relation $\sigma_i^2 = \hat{\mathbf{1}}$. The above equation simplifies greatly when $\hat{\mathbf{B}}^{2j} = \hat{\mathbf{1}}$ and $\hat{\mathbf{B}}^{2j+1} = \hat{\mathbf{B}}$, resulting in

$$\begin{aligned} \hat{\mathbf{A}} &= \sum_{j=0}^{\infty} \frac{(-i\xi\Delta t)^{2j}}{(2j)!} \hat{\mathbf{1}} + \hat{\mathbf{B}} \sum_{i=0}^{\infty} \frac{(-i\xi\Delta t)^{2j+1}}{(2j+1)!} \\ &= \cos(\xi\Delta t) \hat{\mathbf{1}} - i \sin(\xi\Delta t) \hat{\mathbf{B}}. \end{aligned} \quad (4.8)$$

With the above equations we can rewrite the different operators of eq. (4.4).

4.2.1 σ_z

In the case of $\hat{\mathbf{B}} = \sigma_z$ and $\xi = h'$, this already is a one qubit operation known as $R_z(\theta)$, the rotation operator about the z -axis.

$$\begin{aligned} e^{-i\sigma_z\Delta t} &= \cos(h'\Delta t) \hat{\mathbf{1}} - i \sin(h'\Delta t) \sigma_z \\ &= \begin{pmatrix} \cos(h'\Delta t) - i \sin(h'\Delta t) & 0 \\ 0 & \cos(h'\Delta t) + i \sin(h'\Delta t) \end{pmatrix} \\ &\equiv \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} = R_z(\theta), \end{aligned} \quad (4.9)$$

where $\theta \equiv 2h'\Delta t$.

4.2.2 $\sigma_k^i \sigma_{k+1}^i$

This is an operation on two qubits, k and $k+1$, and we have to split it into a product of one qubit operations and $CNOT$ gates. First we have the theorem which states that any unitary matrix can be decomposed into a product of two level unitary matrices, and then we use that a two level unitary matrix can be represented as a quantum gate by using only one qubit operations and $CNOT$ gates, see section 3.2 on universality. First we shall factorize $\sigma_k^i \sigma_{k+1}^i$ into two-level unitary operations for x , y , and z , then we shall find the circuit for each factor.

4.2.3 $\sigma_k^x \sigma_{k+1}^x$

In this case we use eq. (4.8) with $\hat{\mathbf{B}} = \exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$,

$$U_x \equiv e^{-i\sigma_k^x \sigma_{k+1}^x \Delta t} = \cos \Delta t \hat{\mathbf{1}} - i \sin \Delta t \sigma_x \otimes \sigma_x,$$

or in matrix form

$$U_x = \begin{pmatrix} \cos \Delta t & 0 & 0 & -i \sin \Delta t \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ -i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix}.$$

We use the Pauli matrices as seen in the gate description of the X , Y and Z gates in section 2.4, and the rules for tensor products of matrices found in appendix A.1.

Two level unitary matrices

The goal is to find a set of two level unitary matrices U_j such that

$$U_k U_{k-1} \cdots U_2 U_1 U = \hat{\mathbf{1}}. \quad (4.10)$$

For a 4×4 matrix, k is less than or equal to 6. For our specific Hamiltonian k is two. What we do first is define U_1 such that $U_1 U$ is a matrix where every component but the first in the left hand column is zero. We set up U_1 with two unknowns, a and b ,

$$U_1 \equiv \begin{pmatrix} a & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -i \sin \Delta t & 0 & 0 & -\cos \Delta t \end{pmatrix}.$$

This gives us $U_1 U_x$ as

$$U_1 U_x = \begin{pmatrix} a \cos \Delta t - bi \sin \Delta t & 0 & 0 & -ai \sin \Delta t + b \cos \Delta t \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ b(i \sin \Delta t - i \sin \Delta t) & 0 & 0 & (i \sin \Delta t)^2 - (\cos \Delta t)^2 \end{pmatrix}.$$

If we now choose $a = -\cos \Delta t$ and $b = -i \sin \Delta t$ we get

$$U_1 U_x = - \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The inverse of $U_1 U_x$ is a two level unitary matrix, which means eq. (4.10) is already fulfilled, and we have U_x as a product of two two-level unitary matrices, $U_x = U_1^{-1}(U_1 U_x)$. For further use we define $U_1 U_x \equiv B$. To find the inverse of U_1 we find it's Hermitian conjugate U_1^\dagger ,

$$U_x = U_1^{-1} B = \begin{pmatrix} \cos \Delta t & 0 & 0 & -i \sin \Delta t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (4.11)$$

4.2.4 $\sigma_k^y \sigma_{k+1}^y$

Here we have $U = \cos \Delta t \hat{1} - i \sin \Delta t (\sigma_y)_k (\sigma_y)_{k+1}$, which has the following matrix form

$$U_y = \begin{pmatrix} \cos \Delta t & 0 & 0 & i \sin \Delta t \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix}. \quad (4.12)$$

Decomposing it gives us

$$U_y = U_1^{-1} B = \begin{pmatrix} \cos \Delta t & 0 & 0 & i \sin \Delta t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (4.13)$$

where B is the matrix defined in the previous section.

4.2.5 $\sigma_k^z \sigma_{k+1}^z$

For the z -case it is even simpler as U itself is a two level unitary matrix,

$$U_z = \begin{pmatrix} \cos \Delta t - i \sin \Delta t & 0 & 0 & 0 \\ 0 & \cos \Delta t + i \sin \Delta t & 0 & 0 \\ 0 & 0 & \cos \Delta t + i \sin \Delta t & 0 \\ 0 & 0 & 0 & \cos \Delta t - i \sin \Delta t \end{pmatrix}. \quad (4.14)$$

Since $\cos \Delta t - i \sin \Delta t = \exp(-i\Delta t)$ we have

$$U_z = e^{-i\Delta t} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{2i\Delta t} & 0 & 0 \\ 0 & 0 & e^{2i\Delta t} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

4.3 Two-level unitary matrices to quantum gates

In this part we make the final connection between the Hamiltonian and the quantum gates. Here we follow the procedure given in section 3.2.2 for creating quantum circuits representing two-level unitary operators.

The gates for the two types

The two-level unitary matrices that are part of our time evolution operator all operate on either the $|00\rangle$ and $|11\rangle$ basis states, or the $|01\rangle$ and $|10\rangle$ basis states. To create a circuit implementing the operator we must use the swapping technique developed in section 3.2.2. The idea is to use gates to swap the coefficients of the two states affected so that they become the coefficients of two neighboring basis states, and then perform an appropriate controlled operation on the two qubits. When we swap back, the coefficients of the target states have been changed correctly.

The circuit is a factorization of the two-level unitary operator into swap operations and one conditional operation. The conditional operation is the single-qubit operation represented by the matrix of non-trivial components of the two-level unitary matrix, see eq. (3.15). The circuit looks the same for any operation working on the same two states, except that the matrix of non-trivial components, \tilde{V} , is different in each case.

We now must find the gate implementation of each, these are the two

kinds of matrices we have to deal with

$$V_1 = \begin{pmatrix} a & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ c & 0 & 0 & d \end{pmatrix}, V_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (4.15)$$

See table 3.1 for a list of the different conditional operations. The non-trivial components are represented by \tilde{V} ,

$$\tilde{V} \equiv \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

V_1

First we find the circuit for matrix V_1 . The two affected states are $|0\rangle|0\rangle$ and $|1\rangle|1\rangle$, the same case as in eq. (3.23). We swap from the $|0\rangle|0\rangle$ state to the $|0\rangle|1\rangle$ state by applying a flip gate, the X gate, to qubit two, conditional on qubit one being in the state $|0\rangle$,

$$|0\rangle \otimes |0\rangle \rightarrow |0\rangle \otimes |1\rangle. \quad (4.16)$$

Now we can apply the \tilde{V} gate to qubit one, conditional on qubit two being in the $|1\rangle$ state, achieving the desired effect on the coefficients. Then finally we swap the $|0\rangle|0\rangle$ and $|0\rangle|1\rangle$ states back. The circuit implementing V_1 is shown in fig. 4.1.

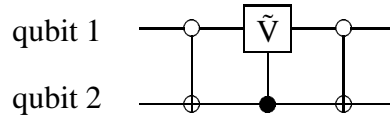


Figure 4.1: V_1 circuit

The gate representing the swap operation, seen in eq. (3.25), is the first of the three gates in the circuit. The second gate is the conditional \tilde{V} operation, the same operation that is shown in eq. (3.26).

V_2

In this case the affected states are $|01\rangle$ and $|10\rangle$. We see that the two states are different in both qubits, and we need to perform a swap from $|0\rangle|1\rangle$ to $|0\rangle|0\rangle$,

$$|0\rangle \otimes |1\rangle \rightarrow |0\rangle \otimes |0\rangle. \quad (4.17)$$

First we have to flip the second qubit (X gate), conditional on the first being zero. Then we apply the \tilde{V} operation to the first qubit, conditional on the second being zero. Swapping back we complete the circuit, as seen in fig. 4.2. The difference between this and the previous circuit is merely that the conditional operation is performed if the control qubit is in the $|0\rangle$ state rather than the $|1\rangle$ state.

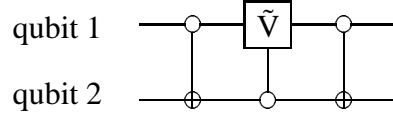


Figure 4.2: V_2 circuit

Having shown how each circuit is designed, we go through all the operators of the time evolution listing the circuits representing them.

4.3.1 $\exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$

As we saw in section 4.2.3, eq. (4.11), the $\exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$ operator factorizes into two two-level unitary matrices. We call the left one A , and the right one B , and the 2×2 matrix of their non-trivial components we denote by \tilde{A} and \tilde{B} ,

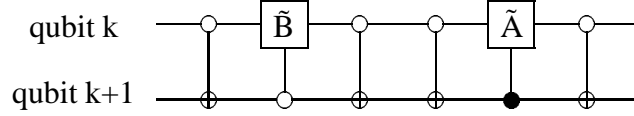
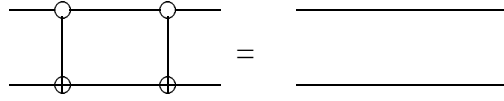
$$A = \begin{pmatrix} \cos \Delta t & 0 & 0 & -i \sin \Delta t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix}, \tilde{A} \equiv \begin{pmatrix} \cos \Delta t & -i \sin \Delta t \\ -i \sin \Delta t & \cos \Delta t \end{pmatrix}, \quad (4.18)$$

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tilde{B} = \tilde{A}. \quad (4.19)$$

We get one circuit for each matrix, and since matrix multiplication is non-commutative, we have to apply the B circuit first, and then the A circuit. This gives us the total circuit for the operation $\exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$ in fig. 4.3.

However, since two $XCNOT$ gates are the same as the identity transformation,

we get the the total circuit as seen in fig. 4.5.

Figure 4.3: First circuit for “ XX ” operatorFigure 4.4: $XCNOT$ twice is unity

Checking the circuit

We have included an example to show that the circuit indeed does perform the same operation as the operator $\exp(-i\sigma_k^x \sigma_{k+1}^x \Delta t)$.

The effect of multiplying matrix A , eq. (4.18), with a general vector is

$$A \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \cos \Delta t - id \sin \Delta t \\ b \\ c \\ -ia \sin \Delta t + d \cos \Delta t \end{pmatrix}, \quad (4.20)$$

where the vector (a, b, c, d) is a representation of the state $|i\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$, using the binary enumeration of eq. (2.22).

In our computation we first flip qubit two conditional on qubit 1 being $|0\rangle$. That means that the state $|00\rangle$ shifts to $|01\rangle$ and $|01\rangle \rightarrow |00\rangle$. The total system, $|i'\rangle$, is now $b|00\rangle + a|01\rangle + c|10\rangle + d|11\rangle$.

Then we apply \tilde{A} to qubit one conditional on qubit two being $|1\rangle$. That means we apply \tilde{A} to the basis states $|01\rangle$ and $|11\rangle$,

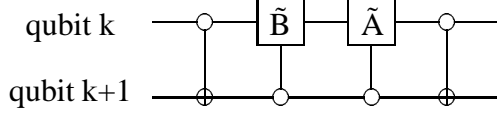
$$a|01\rangle + d|11\rangle = (a|0\rangle + d|1\rangle)|1\rangle.$$

The effect of \tilde{A} on one qubit in the state $a|0\rangle + d|1\rangle$ is given by

$$\begin{pmatrix} \cos \Delta t & -i \sin \Delta t \\ -i \sin \Delta t & \cos \Delta t \end{pmatrix} \begin{pmatrix} a \\ d \end{pmatrix} = \begin{pmatrix} a \cos \Delta t - id \sin \Delta t \\ -ia \sin \Delta t + d \cos \Delta t \end{pmatrix}.$$

This shows that after the conditional \tilde{A} on qubit one we have the total state

$$|i''\rangle = b|00\rangle + (a \cos \Delta t - id \sin \Delta t)|01\rangle + c|10\rangle + (-ia \sin \Delta t + d \cos \Delta t)|11\rangle. \quad (4.21)$$

Figure 4.5: Final circuit for “ XX ” operator

Now we flip $|01\rangle$ to $|00\rangle$ and $|00\rangle$ to $|01\rangle$, and we have the final state

$$|i'''\rangle = (a \cos \Delta t - id \sin \Delta t)|00\rangle + b|01\rangle + c|10\rangle + (-ia \sin \Delta t + d \cos \Delta t)|11\rangle.$$

This is exactly what we got in eq. (4.20), namely

$$\begin{pmatrix} a \cos \Delta t - id \sin \Delta t \\ b \\ c \\ -ia \sin \Delta t + d \cos \Delta t \end{pmatrix}.$$

4.3.2 $\exp(-i\sigma_k^y \sigma_{k+1}^y \Delta t)$

For the y case we have a slightly different circuit for the two matrices given in section 4.2.4 eq. (4.13). We call the operator $U_y = CB$, where the B matrix is the same as in eq. (4.18) and C is given by

$$C \equiv \begin{pmatrix} \cos \Delta t & 0 & 0 & i \sin \Delta t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ i \sin \Delta t & 0 & 0 & \cos \Delta t \end{pmatrix}, \tilde{C} \equiv \begin{pmatrix} \cos \Delta t & i \sin \Delta t \\ i \sin \Delta t & \cos \Delta t \end{pmatrix}, \quad (4.22)$$

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \Delta t & -i \sin \Delta t & 0 \\ 0 & -i \sin \Delta t & \cos \Delta t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tilde{B} = \tilde{A} = \begin{pmatrix} \cos \Delta t & -i \sin \Delta t \\ -i \sin \Delta t & \cos \Delta t \end{pmatrix}. \quad (4.23)$$

This gives us the total gate for U_y in fig. 4.6, using the same equality for two $XCNOT$ gates after one another.

4.3.3 $\exp(-i\sigma_k^z \sigma_{k+1}^z \Delta t)$

In this case we have one matrix operating on the $|01\rangle$ and the $|10\rangle$ states, and in addition we have one operator $e^{-i\Delta t \hat{\mathbf{1}}^{4 \times 4}}$, given in section 4.2.4 eq. (4.13).

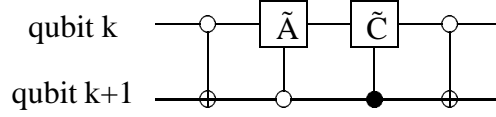


Figure 4.6: Circuit for the “YY” operator

The $\hat{\mathbf{1}}^{4 \times 4}$ operator is the 4×4 unity operator, so that $e^{-i\Delta t} \hat{\mathbf{1}}^{4 \times 4}$ is an operation that changes the phase of the states. Since $\hat{\mathbf{1}}^{4 \times 4} = \hat{\mathbf{1}}^{2 \times 2} \otimes \hat{\mathbf{1}}^{2 \times 2}$, the operation $e^{-i\Delta t} \hat{\mathbf{1}}^{4 \times 4}$ on the two-qubit state is a one-qubit operation \tilde{E} on one of the qubits, where E must be $e^{-i\Delta t} \hat{\mathbf{1}}^{2 \times 2}$, because of the properties of factors in tensor products, $aA \otimes bB = abA \otimes B$. We define the following operations

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{2i\Delta t} & 0 & 0 \\ 0 & 0 & e^{2i\Delta t} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tilde{D} = \begin{pmatrix} e^{2i\Delta t} & 0 \\ 0 & e^{2i\Delta t} \end{pmatrix}, \tilde{E} = \begin{pmatrix} e^{-i\Delta t} & 0 \\ 0 & e^{-i\Delta t} \end{pmatrix}. \quad (4.24)$$

This gives us the gate for the operator $\exp(-i\sigma_k^z \sigma_{k+1}^z \Delta t)$ in fig. 4.7.

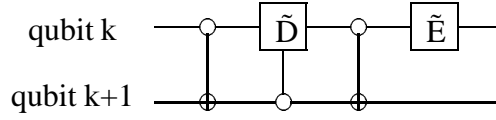


Figure 4.7: Circuit for the “ZZ” operator

4.3.4 $\exp(iE_{max}\Delta t)$

The final part of our time evolution simulation circuit is the E_{max} part. This is a scalar times the unity operator, and therefore commutes with the rest of the Hamiltonian meaning that there is no error in factorizing out $\exp(iE_{max}\Delta t)$ of U .

The operator we wish to implement as a circuit is

$$e^{iE_{max}\Delta t} \hat{\mathbf{1}}.$$

Because $a|\phi\rangle \otimes |\psi\rangle = |\phi\rangle \otimes a|\psi\rangle$ we only need to apply the phase $E_{max}\Delta t$ to one qubit. In our simulation we choose to apply it to the first simulation qubit, see fig. 4.8.

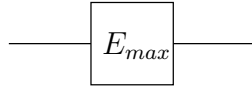


Figure 4.8: Gate ensuring negative eigenvalues

4.4 The final circuit

Now we have all the factors of the approximated time evolution operator described as quantum circuits, and we just have to combine them to get a total circuit, running through the same circuit a number of times with small Δt to minimize error. The circuit in fig. 4.9 is the approximated time evolution operator for one term of the Hamiltonian, eq. (4.2), $i = 1$, where we have used the $\mathcal{O}(\Delta t^2)$ approximation. The total time evolution operator will perform this circuit once for each qubit, with the qubit in question as the top line and the next qubit as the bottom line.

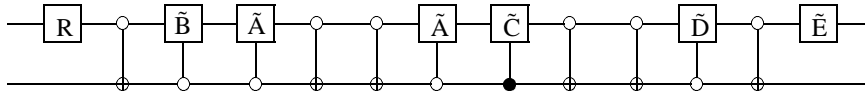


Figure 4.9: Final circuit for one term of the Hamiltonian

Chapter 5

The simulation

In this chapter we discuss the algorithms we use on our classical computers to implement the quantum simulation algorithm. Then we discuss the results from our calculations, and elaborate on the computational complexity of the quantum version and the classical version of the algorithm.

5.1 Algorithms

In this section we present the algorithms used to simulate the quantum simulation algorithm.

5.1.1 Operations, gates and matrices

In our simulation of a quantum system we have a set of operations called a circuit. Each operation is split into the universal basic operations, represented by the two types of gates discussed in the previous chapters, viz. the *CNOT* gate and single-qubit gate. All operations are represented as matrices, see section 2.2.2. This we can simulate using the rules of tensor products found in appendix A.1. We generate a $2^N \times 2^N$ matrix operating on the state vector, representing one operation. If we have N qubits, and each qubit is operated on by gate A_i , represented by a 2×2 matrix for the single-qubit operator, then the matrix for the total operator, $\hat{\mathbf{A}}$, is the tensor product of each of the matrices for the A_i operators,

$$\hat{\mathbf{A}} = A_1 \otimes A_2 \otimes \cdots \otimes A_N = \bigotimes_{i=1}^N A_i. \quad (5.1)$$

5.1.2 Single-qubit gate

To generate a matrix for any single-qubit operator, we use eq. (5.1). If, for example, we wish to apply the X -gate, see section 2.4, to qubit i in a system of total N qubits, then the abstract operator is

$$\hat{\mathbf{x}}(i) \equiv \hat{\mathbf{1}}(i-1) \otimes X \otimes \hat{\mathbf{1}}(n-i), \quad (5.2)$$

where

$$\hat{\mathbf{1}}(k) \equiv \bigotimes_k \hat{\mathbf{1}}^{2 \times 2} \quad (5.3)$$

is the unity operator on k qubits.

Implementation of the tensor product

As an example we take a two-qubit system, where we wish to do the X operation on the second qubit. The operators on the two qubits are,

$$A^1 = \hat{\mathbf{1}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, A^2 = X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Then the total operator of the two-qubit system will be

$$\begin{aligned} \hat{\mathbf{A}} &= A^1 \otimes A^2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} A_{11}^1 A^2 & A_{12}^1 A^2 \\ A_{21}^1 A^2 & A_{22}^1 A^2 \end{pmatrix} \\ &\Rightarrow \hat{\mathbf{A}} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \end{aligned}$$

5.1.3 Controlled operations

The most common conditional operation is the controlled X gate, also known as the controlled not gate or *CNOT* gate, see section 2.4 for its matrix representation. In addition because of the computational basis matrix representation we use, see section 2.3.1, the matrix is different if the second qubit is the control qubit instead. We also use a general controlled single-qubit operation, called *CU*, where U is a 2×2 matrix. See section 2.3.2 for more details on what a controlled operation is. In [16] page 181 it is shown that every *CU* operation can be expressed as a composite gate using only *CNOT* and single qubit gates. The different types of controlled operations described below are essentially the same for a quantum computer, but all have different matrices. Here we calculate the matrix representations of the different controlled operations.

5.1.4 CU

First we show the controlled U gate, or CU where the C stands for controlled or conditional. It is a gate that applies the single qubit gate U to the target qubit if the control qubit is in the $|1\rangle$ state. The controlled operation has two different matrix representations, depending on whether the target qubit or the control qubit is first. The uppermost qubit line in the circuit diagram represents the first qubit.

First we show the gate where the control qubit is first, see fig 5.1.

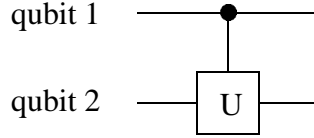


Figure 5.1: Control U gate

To find the matrix, we must examine the effect on an arbitrary state. This is the unitary operation U ,

$$U \equiv \begin{pmatrix} U_a & U_b \\ U_c & U_d \end{pmatrix}. \quad (5.4)$$

It's effect on a general one qubit state is:

$$U|\xi\rangle = U(\xi_a|0\rangle + \xi_b|1\rangle) = (U_a\xi_a + U_b\xi_b)|0\rangle + (U_c\xi_a + U_d\xi_b)|1\rangle. \quad (5.5)$$

If we now look at a general two qubit state and apply the U operator on the second qubit if the first is $|1\rangle$, and do not apply it if the first is $|0\rangle$, we get

$$\begin{aligned} & CU(a|0\rangle \otimes |0\rangle + b|0\rangle \otimes |1\rangle + c|1\rangle \otimes |0\rangle + d|1\rangle \otimes |1\rangle) \\ &= a|0\rangle \otimes |0\rangle + b|0\rangle \otimes |1\rangle + c|1\rangle \otimes (U_a|0\rangle + U_c|1\rangle) + d|1\rangle \otimes (U_b|0\rangle + U_d|1\rangle) \\ &= a|0\rangle \otimes |0\rangle + b|0\rangle \otimes |1\rangle + (cU_a + dU_b)|1\rangle \otimes |0\rangle + (cU_c + dU_d)|1\rangle \otimes |1\rangle. \end{aligned} \quad (5.6)$$

The matrix representation of the operation is,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_a & U_b \\ 0 & 0 & U_c & U_d \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ b \\ cU_a + dU_b \\ cU_c + dU_d \end{pmatrix}, \quad (5.7)$$

which shows us the form of a controlled U operation,

$$CU \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_a & U_b \\ 0 & 0 & U_c & U_d \end{pmatrix}. \quad (5.8)$$

In the specialized case of $CNOT$, which uses the Pauli matrix X as U , the matrix is

$$CNOT \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \quad (5.9)$$

5.1.5 XCX

The XCX gate performs the operation U on the second qubit if the first is in the state $|0\rangle$. This is implemented by using a CU gate and two X gates. We flip the state of the control qubit with an X gate, then employ the CU gate, then flip the control qubit back with another X gate, see fig. 5.3.

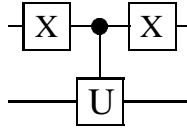
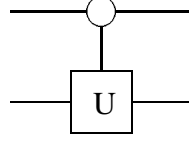


Figure 5.2: XCX gate as combination of X and CU

The matrix representation of this operation is found by matrix multiplication of $(\hat{X} \otimes \hat{1})CU(\hat{X} \otimes \hat{1})$. See section 5.1.2 on how to calculate single-qubit gates, such as $\hat{X} \otimes \hat{1}$. Which gives us,

$$XCX \equiv \begin{pmatrix} U_a & U_b & 0 & 0 \\ U_c & U_d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (5.10)$$

The conditional operation where the target qubit is in the state $|0\rangle$ instead of $|1\rangle$ is depicted by empty circles in circuit diagrams, whereas the regular ones are depicted by dots. The graphical representation of the XCX gate is seen in fig. 5.3.

Figure 5.3: XCU gate

5.1.6 UC

This is a control U operation as in section 5.1.4 except that the first qubit is the target and the second the control qubit, while the same on a quantum computer, the two gates become different when using matrix representations.

To find it's matrix representation we use the same procedure as for the CU operation. First we find the effect on a general two qubit state, and then extrapolate the matrix. The effect of U on a general single-qubit state is given by

$$U|\xi\rangle = U(\xi_a|0\rangle + \xi_b|1\rangle) = (U_a\xi_a + U_b\xi_b)|0\rangle + (U_c\xi_a + U_d\xi_b)|1\rangle. \quad (5.11)$$

The UC operation on a two-qubit state then becomes,

$$UC(a|0\rangle \otimes |0\rangle + b|0\rangle \otimes |1\rangle + c|1\rangle \otimes |0\rangle + d|1\rangle \otimes |1\rangle) \quad (5.12)$$

$$= a|0\rangle \otimes |0\rangle + b(U_a|0\rangle + U_c|1\rangle) \otimes |1\rangle + c|1\rangle \otimes |0\rangle + d(U_b|0\rangle + U_d|1\rangle) \otimes |1\rangle. \quad (5.13)$$

We see that only the states where the second qubit is $|1\rangle$ are affected, the state then becomes

$$a|0\rangle \otimes |0\rangle + (bU_a + dU_b)|0\rangle \otimes |1\rangle + c|1\rangle \otimes |0\rangle + (bU_c + dU_d)|1\rangle \otimes |1\rangle. \quad (5.14)$$

The same operation using matrices is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & U_a & 0 & U_b \\ 0 & 0 & 1 & 0 \\ 0 & U_c & 0 & U_d \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} a \\ bU_a + dU_b \\ c \\ bU_c + dU_d \end{pmatrix},$$

we see that the matrix representation is

$$UC = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & U_a & 0 & U_b \\ 0 & 0 & 1 & 0 \\ 0 & U_c & 0 & U_d \end{pmatrix}. \quad (5.15)$$

The circuit diagram of the UC gate is shown in fig. 5.4.

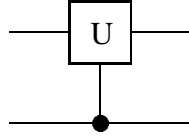


Figure 5.4: UC gate

5.1.7 XUC

This is the UC gate, except that the target qubit is operated on if the control qubit is in the $|0\rangle$ state. We use the same procedure as in section 5.1.5, flipping the state of the control qubit before and after the UC gate to obtain the XUC gate.

The matrix representation follows from the matrix multiplication $(\hat{\mathbf{1}} \otimes \hat{\mathbf{X}})UC(\hat{\mathbf{1}} \otimes \hat{\mathbf{X}})$,

$$XUC \equiv \begin{pmatrix} U_a & 0 & U_b & 0 \\ 0 & 1 & 0 & 0 \\ U_c & 0 & U_d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (5.16)$$

The gate is seen in fig. 5.5.

Figure 5.5: XUC gate

In section 3.1 we have listed these conditional two-qubit operations.

5.1.8 Conditional operations on non-neighboring qubits

When we perform conditional operations on qubits that are not next to each other, the matrix representation of that operator is different. We must combine the result from section 5.1.2 and the matrix representation of each conditional operation.

As an example we will find the matrix representation of the $CNOT$ gate with qubit t as target and qubit c as control, the control qubit comes before the target qubit, $c < t$. We see that the matrix representation of $CNOT$ can be written as a sum of two tensor product operators

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \hat{\mathbf{1}} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes X. \quad (5.17)$$

This is the same for all the conditional operators we have discussed, and the fact that they cannot be written as a pure tensor product of operators show that they are entangling operators. A conditional operator working on a product state yields an entangled state, see section 2.2.4 on entanglement.

We now have the operator represented as a sum of tensor products. To generalize it to the case where the two qubits are not neighbors we use the procedure in section 5.1.2 on each term. The operation being performed on the $c - 1$ first qubits is the unity operator, as it is with qubits $c + 1$ through $t - 1$ and $t + 1$ to n , where n is the total amount of qubits. The total operator then becomes

$$\begin{aligned} & \hat{\mathbf{1}}(c-1) \otimes \overbrace{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}}^c \otimes \hat{\mathbf{1}}(n-c) \\ & + \hat{\mathbf{1}}(c-1) \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \hat{\mathbf{1}}(t-1-c) \otimes \underbrace{X}_t \otimes \hat{\mathbf{1}}(n-t), \end{aligned} \quad (5.18)$$

here we have also used the definition in eq. (5.3) that $\hat{\mathbf{1}}(m)$ is the unity operator on m qubits.

This procedure where we view the conditional operators as sum of tensor product operators, and generalize them to work on any qubit, is applicable to all operators and is used frequently in our simulation. Especially in the conditional U^{2j} operation in the phase estimation, where all the work qubits are at one time the control qubit in a controlled operation with each simulation qubit as target.

5.2 Eigenvalues by diagonalization

The one-dimensional Heisenberg model can be solved by generating the Hamiltonian as a matrix for the given amount of particles, and then diagonalizing it to find the eigenvalues and eigenvectors. In table 5.1 we list the eigenvalues and corresponding degeneracy of the one dimensional Heisenberg model eq. (4.2), for a system of two, three and four particles. The \hbar' parameter of the Hamiltonian is set to one.

We have only included the result for the $s = 2, 3$ and 4 cases, which are the ones we calculate with the quantum computer simulation. All the eigenvalues are integers so far, but it is only a coincidence, and is not the case for $s = 5$ and larger systems.

$s = 2$		$s = 3$		$s = 4$	
E	deg	E	deg	E	deg
4	1	6	1	8	1
2	1	4	1	6	1
0	1	2	1	4	1
-6	1	0	1	2	3
		-2	2	0	4
		-4	2	-2	3
				-4	1
				-6	1
				-8	1

Table 5.1: Exact eigenvalues with corresponding degeneracy

Solution for $s = 2$

This is the case we simulated most often, and we have used it's eigenvalues and eigenvectors to test the simulations. This problem can be solved analytically with the following eigenvectors v_i and eigenvalues E_i , again with $h' = 1$. The results are seen in table 5.2.

$$\begin{aligned}
 E_1 &= 4 & E_2 &= 2 & E_3 &= 0 & E_4 &= -6 \\
 v_1 &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} & v_2 &= \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} & v_3 &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} & v_4 &= \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \end{pmatrix}
 \end{aligned}$$

Table 5.2: Eigenvectors and eigenvalues for $s = 2$ **5.3 Extracting the eigenvalues**

After running through the phase estimation algorithm we start the measurement simulation. The measured values of the work qubits are stored in an array of two to the power of number of work qubits, 2^t , elements. We use the binary transformation found in section 3.3.3. An array keeps track of how many times each energy is measured, and we use it to find the probability spectrum. The relation between E_k and ϕ_k is given in eq. (3.71).

After we have obtained the work qubit measurements we plot the probability amplitudes of each eigenvalue as a function of the eigenvalues, $E_k = -2\pi\phi/\Delta t$, see figure 5.6. The height of the spikes are unimportant as it illustrates a random superposition of eigenstates, only the position of the maxima on the horizontal axis are important. A program analyzing the data gives us the eigenvalues from the probability distribution.

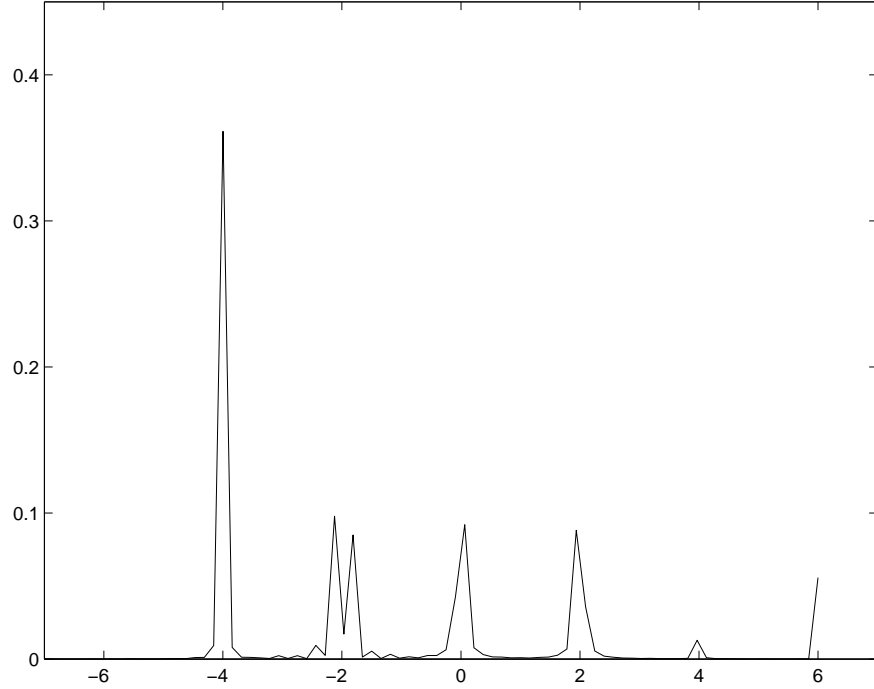


Figure 5.6: The result of the program. Parameters $N = 9$, $t = 6$, $s = 3$

5.3.1 Finding the maxima

Inspection is not satisfactory as a means of determining the eigenvalues, we need to do it numerically. We single out the spikes on the graph corresponding to a single eigenvalue, and then calculate where a single spike would have been if we had perfect resolution. To do that we either single out the areas manually or divide the graph into areas where the probability is larger than some limit, then calculate the average energy weighted by probability.

The weighted average for a given interval, where p_i is the probability and

E_i is the energy, is given by

$$\overline{E} = \frac{\sum_i p_i E_i}{\sum_i p_i}. \quad (5.19)$$

Mechanical analysis

The program we wrote tests every element of the probability array and compares it to a preset limit. If the element is less than or equal to the limit, we move on. If it is larger we make a mark and proceed checking until we find an element below the limit, then we make another mark. These marks constitute the boundaries of a spike, and we use them to calculate the average energy weighted by the probability for that part of the graph, i.e. the eigenvalue.

As an example we have run the program on figure 5.6 with a probability limit of 0.01 (we have discarded areas where the probability is less than 0.01), in table 5.3 we first show the exact solutions and then the numerical results.

E_{exact}	6	4	2	0	-2	-4
$E_{numerical}$	5.9999	3.9628	1.9725	0.0020	-1.9815	-4.0026

Table 5.3: The results of the analysis

This is a fine line to walk, and knowledge of the system beforehand is useful. If the limit is 0.02 instead we loose the $E = 4$ result, and find another false eigenvalue near $E = -2$. If we set the limit to 0.009 we get an extra value near $E = -2$. We see that this simulation is not optimal, and later in the chapter we will find ways to optimize our simulations, so that the calculations will yield more accurate results.

A set of values

A set of values for the energy levels is obtained by performing the same calculation several times, by calculating several probability distributions, and using each to obtain a set of eigenvalues. These calculations need not have the same input state for the simulation qubits, as discussed in section 3.5.9. We then calculate the mean value for each energy level and a mean standard deviation as a measure of the statistical error. To simulate an experiment where the answer is unknown, we estimate the mean standard deviation using the mean energy levels.

We include one such simulation where we ran our program 17 times and averaged the results. We calculate the energy levels of a two qubit system using seven work qubits with the parameters seen in table 5.4.

Parameter	value
N	9
s	2
E_{max}	5
$n\Delta t$	0.5
h'	1

Table 5.4: Parameters

Here s is the number of simulation qubits, E_{max} is the upper boundary of the energies measured and h' is the parameter of the Hamiltonian, see eq. (4.2). A graph from one of the simulations is shown in fig. 5.9.

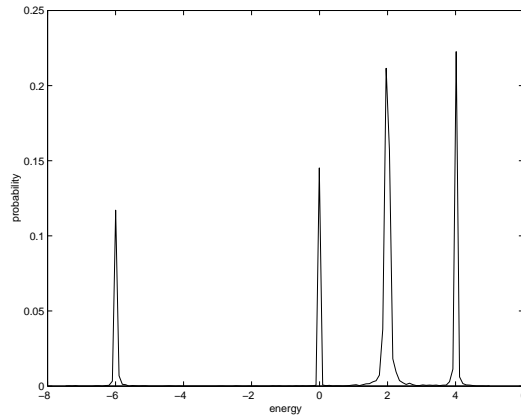


Figure 5.7: The probability spectrum as a function of the energy

The resulting eigenvalues of the Heisenberg Hamiltonian, eq. (4.1), are shown in table 5.5.

We see that two of the energies calculated are not within one standard deviation of the correct answer.

5.4 Minimizing errors

In the previous section we showed how we find the eigenvalues using the probability distribution from several simulations. The results were not exact

Exact	Calculated	Error
4	4.013	± 0.001
2	1.985	± 0.003
0	-0.0075	± 0.0004
-6	-5.996	± 0.001

Table 5.5: Calculated energy levels

however, and in this section we discuss the different errors that can and do arise in our calculations, and the degree to which they affect the outcome. Most importantly we discuss how to minimize their effect.

There are two main errors that we encounter. Spikes on the graph that should not be there, see fig. 5.8, or the spikes are not centered on the correct eigenvalue. These errors lead to calculating more eigenvalues than there are, and to systematic errors in the eigenvalues, see the shift of energies in in table 5.5.

5.4.1 The exponential function approximation

The exponential function approximation in eq. (3.3) causes an error in the time evolution proportional to $\mathcal{O}(\Delta t^3)$. This error depends on the Hamiltonian, and therefore, the number of simulation qubits, s . It also depends on the time interval used in the U operator. In eq. (3.4) we show how the error is reduced by splitting the time interval into n equal parts, where the total time now is $n\Delta t$, and iterating U n times.

We will compare the results from simulations with varying Δt and n to see how many iterations, n , we need to obtain the eigenvalues adequately.

From eqs. (3.2) and (3.3) we see that there is a different error dependence on the size of the time interval for the two approximations. We compare the amount of iterations needed for each approximation, how many gates we use in each simulation and determine which is more effective in total computer time.

The $\mathcal{O}(\Delta t^2)$ approximation

First we use the approximation from eq. (3.2), which has a higher error, but requires less operations. To estimate how many iterations of U are needed we ran several simulations with varying n and constant $n\Delta t$ and compared the results. The system is a nine qubit system, with two simulation qubits and seven work qubits. The two qubit Heisenberg system has four eigenvalues,

see fig. 5.1. Seven work qubits for a two qubit system means we have $2^7/2^2 = 2^5 = 32$ points on the graph per eigenvalue. We use the same parameters as in table 5.4.

The case where $n = 1$ is shown in fig. 5.8, and it is clear that more precision is needed to pinpoint the energies.

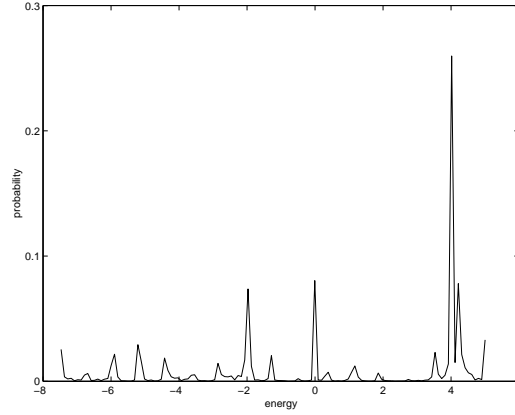


Figure 5.8: Energy spectrum, $\mathcal{O}(\Delta t^2)$ approximation, $n = 1$

In contrast we show the case where $n = 100$ in fig. 5.9. This is close to the exact value, the energies calculated are seen in table 5.6.

E_{exact}	4	2	0	-6
$E_{numerical}$	4.011	1.955	-0.007	-5.976

Table 5.6: Measured eigenvalues for $n = 100$

The simulation with $n = 100$ was the first where there were no splitting of the spikes and provides us with a good estimate of the eigenvalues. The conclusion must be, after several different n 's have been tested, that we need at least 100 iterations when using this approximation.

The $\mathcal{O}(\Delta t^3)$ approximation

Here we show the results using the approximation in eq. (3.3), with the same parameters as above, $N = 9$, $s = 2$, $t = 7$, $n\Delta t = 0.5$, $E_{max} = 5$, $h' = 1$. For one iteration, fig. 5.10, the graph is not sharp enough to find the eigenvalues.

For $n = 30$, fig. 5.11, however, the graph is perfect and the analysis from section 5.3.1 gives us the energies with great precision, see table 5.7.

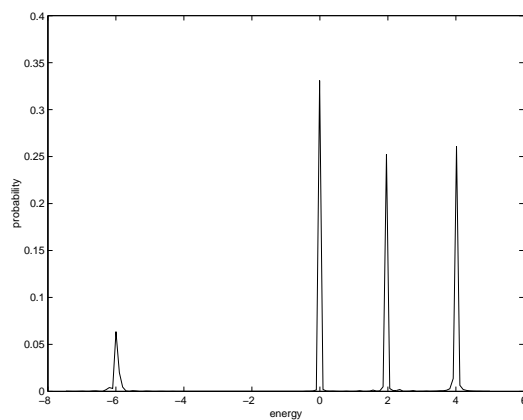


Figure 5.9: Energy spectrum, $\mathcal{O}(\Delta t^2)$ approximation, $n = 100$

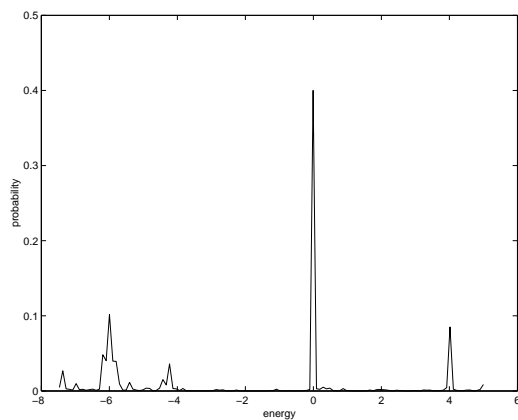
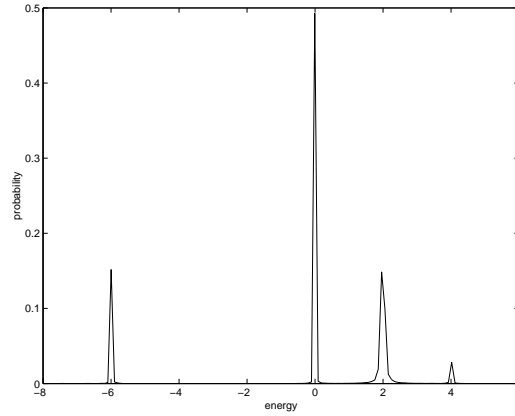


Figure 5.10: Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 1$

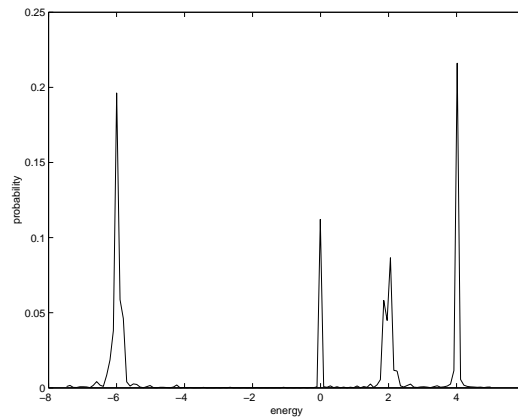
E_{exact}	4	2	0	-6
$E_{numerical}$	4.013	1.989	-0.007	-6.000

Table 5.7: Energies for $n = 30$

When we use only ten iterations the numerical result is good, although the graph is not quite as sharp, fig. 5.12. The energies measured are shown in table 5.8.

Figure 5.11: Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 30$

E_{exact}	4	2	0	-6
$E_{numerical}$	4.011	1.990	-0.007	-5.982

Table 5.8: Energies for $n = 10$ Figure 5.12: Energy spectrum, $\mathcal{O}(\Delta t^3)$ approximation, $n = 10$

Conclusion on approximations and iterations

The $\mathcal{O}(\Delta t^3)$ approximation requires roughly twice as many operations per iteration as the $\mathcal{O}(\Delta t^2)$ approximation, see eq. (3.3). On the other hand the $\mathcal{O}(\Delta t^3)$ approximation needs only ten iterations as a minimum in this case, whereas the $\mathcal{O}(\Delta t^2)$ approximation require 100 as a minimum. This means

we can save roughly a factor of five in computer time by using the more accurate approximation from eq. (3.3).

5.4.2 Work qubits vs. simulation qubits

In this part we discuss how many work qubits, t , are needed to effectively calculate the eigenvalues of a system with s simulation qubits. First we discuss the resolution of the probability spectrum, then we discuss the systematic errors that occur due to the binary approximation, see eq. (3.35) where an integer is described using binary numbers (it is not the same ϕ as we use in the phase estimation algorithm, which is less than one).

Resolution

A system of s simulation qubits has potentially 2^s different eigenvalues, that means there are less than or equal to 2^s spikes on the graph of the probability distribution. If we have t work qubits the resolution of the probability distribution is 2^t , there are 2^t work qubit basis states. To get a graph where we can discern 2^s maxima we obviously need more than 2^s points. The amount of points per eigenvalue is $2^t/2^s = 2^{t-s}$, where $t - s$ is the difference between the number of work qubits and simulation qubits. If we have two more work qubits than simulation qubits, we have four points on the graph per eigenvalue, not counting degeneracy.

In figure 5.13 we illustrate this, here we have $N = 4$, $t = 2$ and $s = 2$. With these parameters it is not possible to single out a specific eigenvalue.

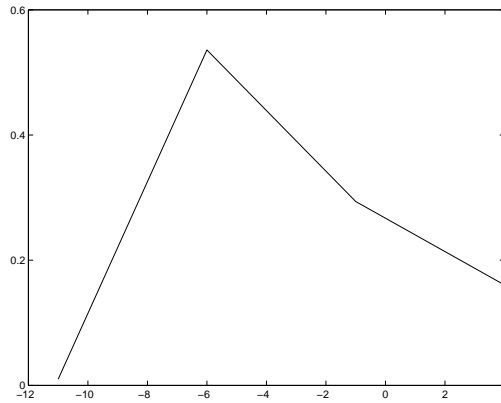


Figure 5.13: Energy spectrum, parameters $N = 4$, $t = 2$, $s = 2$

In section 5.4.1 we gave the results for several simulations with $t - s =$

5 more work qubits than simulation qubits. As we saw we can estimate the energies quite accurately (see section 5.12 for the best results). The results show that a surplus of five work qubits is sufficient to calculate the eigenvalues.

Systematic errors of the binary approximation

The resolution of the energy scale is 2^t , where t is the number of work qubits. The eigenvalues of the system are $\phi_k 2^t = -E_k \Delta t 2^t / 2\pi$ (see eq. (3.60) and eq. (3.68)). If $\phi_k 2^t$ is not an integer, the $\phi_k 2^t$ measured will be wrong by $\delta \leq 1/2$, where δ is the difference between the exact $\phi_k 2^t$ and the nearest integer. This leads to systematic errors in the energy levels calculated, because the system has to “choose” between two states that are near the eigenvalue.

As an elaborated example we look at the calculation done in table 5.5. Here we see that two of the energies are wrong by more than one standard deviation. From the parameters of the simulation we calculate the $\phi_k 2^t$ value of each eigenvalue of the Hamiltonian. In this simulation $2^t = 2^7 = 128$, so there are 128 integer values for $\phi_k 2^t$ to take. The value of $n\Delta t$ is 0.5. The third column in table 5.9, $|\delta|$, is the exact $\phi_k 2^t$ subtracted from the nearest integer.

E_k	$\phi_k 2^t$	$ \delta $
4	10.19	0.19
2	30.56	0.44
0	50.93	0.07
-6	112.05	0.05

Table 5.9: Systematic error due to binary number approximation

We see that $E_k = 4$ and $E_k = 2$ have the $\phi_k 2^t$'s that are farthest from an integer, and they are also the eigenvalues that have the largest systematic error.

To show this more explicitly we ran an identical simulation where we had changed Δt so that $\phi_k 2^t = 96$ for the eigenvalue -6 , see table 5.10. In this simulation we see that the calculated values for $E_k = 0$ and $E_k = 4$ are both systematically wrong by about 0.020. They are also the two values with the greatest $|\delta|$.

To compensate for this error we can run simulations with different $n\Delta t$ to get different systematical errors in each. The average of these simulations will then give a better result, but with a higher standard deviation. To show

E_k	$\phi_k 2^t$	$ \delta $	Calculated	Error
4	8.73	0.27	3.974	0.004
2	26.18	0.18	2.014	0.001
0	43.64	0.36	-0.020	0.002
-6	96.00	0.00	-6.000	0.000

Table 5.10: Energy spectrum, $\phi_k 2^t = 96$ for $E_k = -6$

this we ran a third calculation exactly like the two previous ones, except with a different $n\Delta t$. The total averaged result is seen in fig. 5.11.

Exact	Calculated	Error
4	4.001	± 0.01
2	1.997	± 0.007
0	-0.0157	± 0.003
-6	-6.005	± 0.005

Table 5.11: Averaged over three different $n\Delta t$'s

In total 53 simulations are used to achieve these results. Even now, the systematic error for the $E_k = 0$ eigenvalue is so great that the estimated value is not within one standard deviation. To obtain the exact value, more simulations with different time is needed. The best results will be achieved if $|\delta|$ goes from zero to $1/2$ for all the eigenvalues. That way we will average out the systematic error. If $\phi_k = \psi_n + \delta/2^t$ is the exact eigenvalue of the system then

$$\phi_k = \psi_n + \delta/2^t = \frac{-E_k \Delta t}{2\pi} \equiv a_k \Delta t. \quad (5.20)$$

If we now make a small change in the the time interval, $\Delta t \rightarrow \Delta t + T$, then

$$\phi'_k = \psi_n + \delta'/2^t = a_k(\Delta t + T) \Rightarrow \delta'/2^t = \delta/2^t + a_k T. \quad (5.21)$$

We see that varying $a_k T$ from zero to $1/2^t$ will make δ obtain all values from $-1/2$ to $1/2$, but only for ϕ_k . If we have another eigenvalue $\phi_\nu = a_\nu \Delta t$, then with the same variation, T , in Δt the variation in ϕ_ν will be

$$\delta'_\nu = \delta_\nu + \frac{a_\nu}{a_k}. \quad (5.22)$$

If we choose E_k to be the greatest eigenvalue of the system then $a_k < a_\nu$, making sure the δ 's change from $-1/2$ to $1/2$ at least once for all eigenvalues

of the system. This means we have to make an educated guess at the a_k value to use, adjusted by the results of simulations.

For the two-qubit system the smallest a_k is $a_k = (E_{max} - 4)/2\pi$. In table 5.12 we show the results of a calculation with the same parameters as the previous simulations, except that we vary $n\Delta t$ in equal steps from 0.5 to $0.5 + 1/2^t a_k = 0.5 + 1/(2^t(5-4)/2\pi) = 0.5 + 2\pi/2^t$ in 20 different simulations.

Exact	Calculated	Error
4	3.996	± 0.007
2	1.996	± 0.006
0	-0.004	± 0.007
-6	-6.003	± 0.007

Table 5.12: Varying Δt

This time all the mean values are within one standard deviation of the exact value.

Conclusion on number of work qubits

We have shown that the problem of the binary approximation can be solved without increasing the number of work qubits. And that the difference between the number of work qubits and simulation qubits should not have to increase as the number of simulation qubits increase.

5.4.3 Analysis errors

The analysis algorithm we use is not flawless and may introduce errors. These errors, however, are usually minimized before the results from the analysis are accepted, and they are reduced as the resolution and number of iterations increase.

The factor that affects the precision and running time of each individual calculation is how many measurements are used to obtain the probability distribution that we use to determine the eigenvalues. The number of measurements needed to obtain a given accuracy is proportional to the number of work qubit basis states 2^t . This means that as the system increases the number of measurements will increase, but linearly and will not cause a dramatic speedup.

In the examples we have used the number of measurements per graph have been 10000. This number could be reduced and we would still have good results.

5.4.4 Conclusion on errors

The two problems that affect our calculations can be solved without too much increase in consumed computer power.

When the number of iterations is doubled, the running time of the program is not doubled, due to other parts of the computation than the conditional U circuit. That means increasing the number of iterations increases the running time less than linearly, and is not a problem.

The work qubit problem is different. Adding a work qubit to the system at least quadruples the memory needed and the time consumed. The errors arising from too few work qubits can be reduced if we run several simulations with $n\Delta t$ always slightly different, this eliminates the systematic errors from the binary number approximation, and shows us that we can get good results without many work qubits.

5.5 Computational complexity

Efficiency of algorithms is measured in how much the resources required to perform a calculation increase as the size of the problem increases. In this section we will show explicitly that the quantum simulation algorithm is much more effective on a quantum computer than on a classical computer, and indeed is much more effective than any general classical quantum simulation algorithm.

To illustrate the amount of information and computations needed to perform simulations of quantum mechanical systems, we will look at the Heisenberg model used in this thesis. The complexity is of the same order for the quantum simulation algorithm and for the diagonalization of the Hamiltonian. We will also look at how many gates are needed to perform the computation on a quantum computer.

5.5.1 Complexity

The efficiency of algorithms is measured in how much the capacity needed and the time consumed increases as the size of the problem increases. A linear increase of running time means that the running time doubles as the size of the problem, n , doubles. We then say it is an $\mathcal{O}(n)$ problem in time.

This is an example of polynomial problems, which are problems where the running time and/or memory needed increases as a polynomial of n . An $\mathcal{O}(n^3)$ problem takes eight times as long to compute if n is doubled.

Problems exponential in n are exceedingly difficult to expand, and exponential algorithms are therefore called inefficient. For example an $\mathcal{O}(2^n)$ problem in time where n is doubled would take $2^{2n}/2^n = 2^n$ times as long to compute. An increase of n to $n+1$ doubles the time. It is clearly not feasible to perform these kinds of calculations when n is large, see section 5.5.2 for an upper boundary on the classical calculation of the quantum simulation algorithm.

5.5.2 Classical complexity

First we see how much memory is needed on our classical computers for a program simulating a 13 qubit quantum computer. To simulate a system of 13 qubits we need a state vector with 2^{13} elements. An operator represented by a matrix has $2^{2 \cdot 13} = 2^{26}$ elements. Each element consists of two complex numbers. We use double precision in our programs, that means 64 bits of memory are used to represent each number. A byte is eight bits, that means the computer uses $8 = 2^3$ bytes per number. Two complex numbers per element gives us $16 = 2^{3+1} = 2^4$ bytes for every element in the matrix or $2^{26+4} = 2^{30}$ bytes total. One kilobyte is $2^{10} = 1024$ bytes, and one gigabyte is $(2^{10})^3 = 2^{30}$ bytes, which is the memory needed to store the 13 qubit operator.

To perform the calculations efficiently we need to have the complete matrix representing an operator in the dynamic memory of the computer. Some home computers today, 2003, can have three gigabytes of dynamical memory and could simulate this. However, every time the size of the system is increased by one qubit, the number of elements in the matrix is quadrupled. A 15 qubit operator matrix would then require four gigabytes, and such a simulation cannot be done on the above computer. There are, however, computers that can perform much better than these home computers. Parallel computers can have thousands of gigabytes of memory, and in [17] they simulate up to 30 qubits using parallel computing.

The many-body computation performed in this thesis using the quantum simulation algorithm is an algorithm exponential in both running time and memory. The number of elements in a matrix representing an N qubit operator is 2^{2N} , and the number of mathematical operations in one matrix vector multiplication is 2^{2N} . That means that it is a problem of exponential complexity, $\mathcal{O}(2^{2N})$, both in time and memory needed.

Time

In table 5.13 we show the time, in seconds, it took to run the program on our computers. In this example we only ran U once, that is $n\Delta t = \Delta t$. The total number of qubits is N and the number of simulation qubits is s .

s	$N = 4$	$N = 5$	$N = 6$	$N = 7$	$N = 8$	$N = 9$
2	1	3	17	173	3173	27909
3			25	313	3516	33142
4					3867	30981

Table 5.13: Time as function of N and s

The numbers themselves are not interesting, and may not even have the correct ratios due to other processes that may have been running at the same time. We use the ratios to show that the time it takes to run the program increases exponentially as the number of qubits increase. A plot of the logarithm of time as a function of the number of qubits nearly produces a straight line.

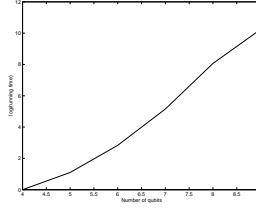


Figure 5.14: Logarithm of time vs number of qubits for $s = 2$

5.5.3 Quantum computer complexity

The complexity of the algorithm on a quantum computer must be defined as the increase of number of operations as the number of qubits increase. In section 3.4 we discuss the controlled U^{2^j} operations needed for the phase estimation algorithm. These operations can either be achieved by performing $U(\Delta t)$ 2^j times, or by designing the new operation $U(2^j \Delta t)$. A quantum computer consisting of operations controlled by a classical computer or a universal quantum computer in it's own right will be programmable in such a way that the $U(2^j \Delta t)$ option is possible. This is the best solution since the number of operations otherwise would increase exponentially.

The time evolution operator is linear in the number of simulation qubits, s , because of the Hamiltonian which has a constant number of operations per qubit. Using the phase estimation circuit, fig. 3.7, we estimate the total number of operations. If t is the number of work qubits, then the rough number of operations for that figure is kts for the U operations, where k is some constant. This means that the circuit in fig. 3.7 is linear in the number of work qubits, t , and the number of simulation qubits, s . The number of gates needed to implement it increases as $\mathcal{O}(st)$.

In addition to that circuit the phase estimation algorithm applies an inverse Fourier transform on the work qubits, see fig. 3.4. The number of operations performed in this circuit is,

$$1 + 2 + 3 + \cdots + t = \frac{1+t}{2}t = \frac{t^2 + t}{2}.$$

When t is large the number of operations in the inverse Fourier transform increases as $\mathcal{O}(t^2)$. The inverse Fourier transform, however, is applied only once for each simulation, while U is iterated several times.

The total number of qubits is the sum of the simulation and work qubits, $N = s + t$. If the ratio between work and simulation qubits is constant, $t = aN$ and $s = (1-a)N$, the total complexity of the simulation algorithm is quadratic in N , $\mathcal{O}(N^2)$. Equally if the difference between s and t is constant, $t - s = b$, then $st = s(s+b) \sim s^2 \sim N^2$, and the total complexity is $\mathcal{O}(N^2)$.

Gates

As we said in section 2.3.2, all conditional U gates can be simulated by a total of six $CNOT$ and single-qubit gates. When we apply a conditional U operation in our program we add six operations to the number of operations performed, not only one, thus acquiring the correct number of gates. In these simulation we have iterated U only once. Table 5.14 show how many gates are needed to perform one measurement on an N qubit system with s simulation qubits. The first line shows the operations required to simulate the conditional U operations only, while the second line shows all the operations in the simulation.

Examining the numbers we see that the number of operations increase linearly with the number of work qubits for the same number of simulation qubits. The same is true for the number of simulation qubits. If we compare the total amount of operations we see the near quadratic increase in N when the ratio s/t is constant.

s	$N = 4$	$N = 5$	$N = 6$	$N = 7$	$N = 8$	$N = 9$
2	160	240	320	400	480	560
tot	165	249	334	420	507	595
3			336	448	560	672
tot			345	462	580	699
4					576	720
tot					590	740

Table 5.14: Number of gates

5.5.4 Polynomial in N

The time the quantum simulation algorithm on a quantum computer takes is quadratic in the number of qubits. This is far better than the classical simulation whose time is exponential in the number of qubits, like all general classical simulations of quantum systems. This means that once a quantum computer can perform one simulation, it does not take much time or resources to increase the size of the problem. This is why Feynman proposed this use of quantum computers, and it may well be their most important task.

5.5.5 Time

To get a certain measure of the time it would take to perform these calculations on a quantum computer, we compare the number of gates used with the maximum number of operations allowed on some proposed physical systems.

To be able to use as a quantum computer a quantum system must be accessible to manipulation. At the same time we need a stable system that does not couple with the environment and change the state randomly before the calculation is finished. To quantify these two parameters are important, the decoherence time and the operation time. They can only be given as rough estimates for a given system, but provide us with some information on the suitability of the system as a quantum computer.

The decoherence time, τ_Q , is the longest time a quantum computer calculation can take. After that the system will interact with its surroundings and will no longer provide the correct result.

The operation time, τ_{op} , is the time needed to perform a unitary operation on at least two qubits.

The time of the entire calculation divided by the time of each operation gives the maximum number of operations, $n_{op} = \tau_Q/\tau_{op}$. This is a crude estimate, but serves as a tool to rule out certain models of quantum computers.

The field of quantum computers today is a field searching high and low for working models of quantum computation. It is the modern day quest for the holy grail of quantum physics. We have listed in table 5.15 these numbers for a few proposed models, bear in mind that these models and numbers are from [16] which was written in 2000, and the models listed here will probably not be used in the form originally proposed. New ideas abound and especially the field of condensed matter physics looks promising, with hopes for integrated quantum circuits for large scale production.

System	τ_Q	τ_{op}	n_{op}
Quantum dot	10^{-6}	10^{-9}	10^3
Electron spin	10^{-3}	10^{-7}	10^4
Optical cavity	10^{-5}	10^{-14}	10^9

Table 5.15: Decoherence time and operation time

Comparing with the number of gates in fig. 5.14 we see that for example the optical cavity system could perform approximately 10^6 iterations of U on an $N = 9$ system before measuring, while the electron spin system could only perform about ten iterations per measurement before decoherence. This may not seem as many operations at all, but this is the number needed per measurement. The whole process must be started over many times to produce a probability spectrum, and because of that the decoherence is a problem that can be overcome.

We see here that a physical process can be simulated with a number of quantum operations, and a quantified amount of information. Indeed all information is physical, it is a fundamental truth of information technology. Inspired by this Seth Lloyd estimated in [14] the amount of information the universe could have registered and the amount of elementary operations it could have performed, to be no more than 10^{120} operations on 10^{90} bits. The idea that the amount of information is limited is a consequence of the theory that the universe is finite.

Chapter 6

Conclusion

Here we present our conclusions and possible perspectives on the future of quantum computers.

6.1 What we have done

We have introduced quantum computing, in particular the quantum Fourier transform, the phase estimation algorithm and the quantum simulation algorithm.

We have shown how an arbitrary unitary transformation can be broken down into single-qubit unitary transformations and the two-qubit *CNOT* unitary transformation. We have also shown a set of single-qubit transformations that can approximate any single-qubit transformation to arbitrary precision.

We have used the quantum simulation algorithm to solve the eigenvalue problem of the Heisenberg model. This we have done by developing the quantum gates needed to simulate the time evolution operator of the system. We have written a program that simulates this quantum circuit and analyzed the results.

6.2 A new computational algorithm

The quantum simulation algorithm is a new algorithm for solving many-body problems. It is slower on a classical computer than other more commonly used algorithms, such as monte-carlo integration, exact diagonalization and perturbative many-body techniques.

We have shown how it can extract eigenvalues of Hermitian operators to an arbitrary precision, and how the computational errors can be reduced

effectively. Decreasing the time steps of the simulation gives a more precise time evolution, and varying the time steps over several simulations eliminates the error arising from the binary number approximation.

We have shown the complexity of both the classical and quantum implementations. While of exponential complexity on classical computers, it is of polynomial complexity when run on a quantum computer, and therefore a relatively small quantum computer could greatly increase our knowledge of many-body quantum systems. Even though solving a two-spin, $s = 2$, system probably will be faster on a classical computer, the different complexities ensures that the quantum computer quite quickly will be faster than the classical one as the size of the system increases.

6.3 The future of quantum computers

In this section we discuss the problems of making quantum computers, the obstacles to overcome before large scale quantum computations can be performed.

The furthest one has come at present is being able to factorize 15 into five and three, seemingly not an astonishing feat, but it is just an example of several primitive quantum computers that exist today. Indeed, nature is thought to be a quantum computer performing an enormous amount of computations every instant, it is “just” for us as physicists to harness that power. The limitations lie not in theory, but in practice, we need to find a system balancing coherence, ability to manipulate and scalability. Scalability is the major problem today, the ability to include more qubits in the system. No system realised yet has the size to perform any revolutionary calculations, and the hunt is raging. The finders will almost certainly be awarded a Nobel prize and suffer an eternity of fame.

There are many pessimists and sceptics, even those who downright oppose the use of funds in the research of quantum computers, believing that quantum computing is a fashion word invoked merely to acquire funds. No one can see what will come and some sceptic’s claims have already been put to shame by experiments, for example M. I. Dyakonov [5] did not, in 2001, believe the factorization of 15, or even six, could be accomplished using Shor’s algorithm. There are no known theoretical boundaries excluding the possibility of large quantum computers. The only logical thing to do as a scientific community is to work hard at finding a system that works, and if it proves impossible, then that too is progress.

6.3.1 Research today

As we have discussed, much research is devoted to the experimental part of quantum computing, as exemplified by [20], an article describing the experiment where seven qubits are used to factorize 15. New schemes have been proposed for making quantum computers. One is reviewed in [4], where the idea is to “pour” a Bose-Einstein condensate into an optical lattice, with individual atoms trapped in the laser lattice used as qubits.

Leinaas and Myrheim predicted in 1977 [11] that in one or two dimensions particles could have statistics other than those of fermions and bosons. Later these quasi-particles were dubbed anyons, and may have found unexpected use as Kitaev in 1997 [10] proposed to use non-abelian anyons as qubits. This is said to be an intrinsically fault tolerant setup, though purely theoretical at this point. In 2000 Lloyd [13] proposed using abelian anyons instead, a less fault tolerant system, but perhaps more experimentally accessible.

There is still general theoretical work to be done. Guifré Vidal showed in [22] that simulating quantum systems that are only slightly entangled on classical computers is not inefficient, that the complexity is polynomial not exponential. In other words, that it is possible to simulate low entanglement systems on classical computers, and therefore one only need a quantum computer for quantum computing algorithms with a certain degree of entanglement.

Many groups are working on new algorithms that improve on classical ones like the famous algorithms of Shor [19] and Grover [9]. Especially it is sought after the holy grail of computer science, a polynomial time solution of an NP-complete problem. NP stands for non-deterministically polynomial, and is a group of problems that can be solved in polynomial time on a deterministic computer (a theoretical construct that always chooses the right path towards the answer). Polynomial time means that if the size of the problem is increased, the running time is increased polynomially, while exponential time means that for example adding one qubit to a system means it takes twice as long to simulate it. NP-complete problems are a subgroup of NP problems and it is not known whether they can be solved polynomially, but it is known that if one of them can, the whole group can. There are hopes that one will find a polynomial solution to an NP-complete problem using quantum computation.

A quantum computer will have to be partly classical, at least in the beginning, even if just for measurement and controlling operations. It is usual, however, to envision the quantum computer as a complete computer, independent of “archaic” computers and capable of being programmed to different tasks. This of course is the ideal, but may be far away in the

future, as it will need thousands, tens of thousands, hundreds of thousands or more qubits. However, as we suggested in section 6.2 we might not need that large a quantum computer to find uses for it. A recent article [21] deals with a hybrid computer, where a classical computer cooperates with a quantum computer to perform calculations. They have combined the classical fast Fourier transform and the quantum Fourier transform, and their work suggest that a semi-quantum computer could be an improvement over a pure classical computer, even in the case of a small number of qubits.

6.4 My thoughts on the future

The fruits we reap from the quantum simulation algorithm are still to come. In the future hopefully large scale quantum computers will be able to simulate quantum systems not possible to simulate on classical computers. A system described by 30 qubits has roughly a billion basis states, 2^{30} , and cannot be solved for all eigenvalues by any existing computer. A quantum computer of only 100 qubits will be a quantum leap forward in the simulation of quantum systems. And should with a good margin be able to simulate a system of 30 simulation qubits, 2^{30} states, see the arguments in section 5.4.2. Seth Lloyd showed this in [12] and argued that simulation might be the most important aspect of quantum computers.

When people discuss quantum computers, scientists and layfolk alike, they tend to emphasize the quantum algorithms that improve on the speed of classical equivalents, but still can be performed on classical computers, like Grover's search algorithm [9] and Shor's factoring algorithm [19]. These are fascinating results which illustrate the potential power of quantum computers, but I believe the most important application of quantum computers will be the one proposed by Feynman in 1982 [8], simulating quantum mechanical systems. This at least I predict will be the case for the near future, the infancy and childhood of quantum computers, however long a period that might be. Later on, when their size has been greatly increased, quantum computers will be multi-purpose computers that can be programmed to solve any solvable problem.

Who can tell what sorts of problems can be solved, and what sorts of technologies can be developed from this advance, the only certainty is that if large enough quantum computers ever become a reality they will at least greatly boost the fields of nano-technology, molecular biology, high energy physics and quantum mechanics.

I think it's shining through that I am an optimist, and I believe there is good reason, even though it is far from certain and a long way to go.

Perhaps humanity will never reach the stars, but we have already reached deep within nature, and the impact of the findings of the last century have yet to seriously impact the philosophies by which most humans live their lives. I believe quantum computers will revolutionize our perception of nature and life itself as philosophy slowly catches up with science.

Appendix A

Tensor Products

A.1 Tensor product or outer product

A tensor is a multi-linear function. It takes vectors or dual-vectors to complex numbers. If T is a tensor and $|a\rangle$ and $|b\rangle$ are vectors, then $T(|a\rangle, |b\rangle) = x + iy$ is the complex number you get from putting these vectors into the multi-linear function. Multi-linear only means that it has more than one argument. Vectors too are functions, they take dual-vectors to complex numbers. Also called the inner product, but we still associate a geometric abstract meaning to them. In the same way, we have an abstract geometric understanding of a tensor without it's arguments.

A general tensor with two vectors as arguments is written as a tensor-product of two dual-vectors, $T = \langle a| \otimes \langle b|$. Where $\langle a|$ and $\langle b|$ are dual-vectors. The definition of $T(|u\rangle, |v\rangle)$ is

$$T(|a\rangle, |b\rangle) = \langle a|u\rangle \langle b|v\rangle, \quad (\text{A.1})$$

where $\langle a|u\rangle$ is the inner product of dual-vector a and vector u .

In the same way we can make tensors made of vectors which take dual-vectors as arguments. If we have two vector spaces, U of n dimensions with an element $|u\rangle$, and V of m dimensions with an element $|v\rangle$. The tensor product of $|u\rangle$ and $|v\rangle$ is a tensor $T = |u\rangle \otimes |v\rangle$.

Product spaces

We can compose another vector space as a product space of U and V , $U \otimes V$, seen in section 2.1.2. This product space is the set of all the tensors made of vectors from U and V , and is mn dimensional. The elements of $U \otimes V$ are linear combinations of tensor products of the elements of U and V .

If we look at the qubit, we have the two-dimensional vector space V describing the system of one qubit. Then $|0\rangle \otimes |1\rangle$ is an element of $V \otimes V$, so is $|0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle$.

Some of the basic properties of the tensor product are

1. z is a complex number, $|v\rangle$ is an element of V and $|u\rangle$ an element of U , then

$$z(|u\rangle \otimes |v\rangle) = (z|u\rangle) \otimes |v\rangle = |u\rangle \otimes (z|v\rangle).$$

2. $|v_a\rangle$ and $|v_b\rangle$ are elements of V , $|u\rangle$ is an element of U , then

$$(|v_a\rangle + |v_b\rangle) \otimes |u\rangle = |v_a\rangle \otimes |u\rangle + |v_b\rangle \otimes |u\rangle.$$

3. $|v\rangle$ is an element of V and $|u_a\rangle$ and $|u_b\rangle$ are elements of U , then

$$|v\rangle \otimes (|u_a\rangle + |u_b\rangle) = |v\rangle \otimes |u_a\rangle + |v\rangle \otimes |u_b\rangle.$$

They all follow directly from the definition in equation A.1.

A.1.1 Matrix tensor products

We now look at tensor products, or outer products, in a specific matrix representation, the Kronecker product. We need this when we look at our vectors as column matrices, and our operators as square matrices.

Definition 1 *If A is an $(m \times n)$ matrix, and B is an $(r \times s)$ matrix, then*

$$A \otimes B = \overbrace{\begin{pmatrix} A_{11}B & A_{12}B & \cdots & A_{1n}B \\ A_{21}B & A_{22}B & \cdots & A_{2n}B \\ \vdots & \vdots & \vdots & \vdots \\ A_{m1}B & A_{m2}B & \cdots & A_{mn}B \end{pmatrix}}^{ns} \Bigg\}^{mr}.$$

The notation $A_{11}B$ means that we take the element A_{11} and multiply every element of B with it. Therefore the matrix $A \otimes B$ in the above equation has mn submatrices of rs elements, giving it a total of $mnr s$ elements.

In eq. (2.20) we see an example of a tensor-product of two two-dimensional column matrices, granting the four-dimensional column matrix used to describe a state in a two qubit system.

There are different ways to denote a tensor product between two kets, $|u\rangle|v\rangle$, $|uv\rangle$ and $|u, v\rangle$ all mean $|u\rangle \otimes |v\rangle$ and are used in many texts.

Appendix B

The Program

In this appendix we will discuss the programming languages, programming techniques and programs employed in this thesis, so that others may understand the code properly.

B.1 Programming

The language chosen is C++, a language that is low level like C, but has incorporated classes and objects, and is thus a more object oriented language. The last few years there has been great progress in making C++ faster at calculations, and it is becoming a serious contender to fortran. The reason I chose C++ over fortran was only one of personal desire to learn C++.

Unlike fortran, C++ has no built in data types or classes for complex matrices or matrix operations. There is no standard package that implements them either, but a plethora of different ones. For this thesis we chose the Blitz++ package, which can be found and downloaded at

<http://oonumerics.org/blitz/download/>

It allows for a great deal of detailed manipulation on arrays of every sort.

The C++ compiler used is g++, on a Linux OS.

B.1.1 C++ and Blitz++

There are several free manuals on C++ to be downloaded for those who wish to learn. We will assume, however, that the reader has some prior knowledge of C++.

We will just explain a few specialities of the Blitz++ package. An Array is a class that incorporates arrays that can be of any rank and dimension.

That is, every element can have any number of indices, and there can be any number of elements “in” one index. It can be a vector, a matrix, a three dimensional grid etc.

Arrays

The declaration of a complex, double precision (eight bytes represent each number), vector of dimension N is

```
Array<complex<double>,1> Vector(N);
```

A square matrix of similar dimension would be declared like this:

```
Array<complex<double>,2> Matrix(N);
```

Range objects

To manipulate parts of an array we use the Range objects of the Blitz++ package. It gives the starting and ending index of a part of an array to manipulate.

```
Range      x(0,3);
```

```
Vector(x)  = Vector4;
```

This Statement changes the first four elements of *Vector* to the four elements of *Vector4*.

```
Matrix(x,x) = Matrix4x4;
```

This statements puts the four by four matrix *Matrix4x4* into the upper left corner of *Matrix* as a submatrix.

firstIndex

Another speciality of the Blitz++ package is the ability to define firstIndex, secondIndex and so forth, variables. These variables when used as an index instead of a number, run through the whole dimension.

```
firstIndex i;
```

```
A(i,0) = 1;
```

Is the same as:

```
for(int i=0; i < dimA; i++)
```

```
  A(i,0) = 1;
```

This can make the code more compact and easier to write, and sometimes easier to read.

B.2 Bruteforce diagonalization

Here we go through the program used to generate the Hamiltonian as a matrix for the one dimensional Heisenberg model of a given number of spins. What the program does is take each operator on a qubit or a pair of qubits, and finds the operator on the whole system using eq. 5.1.

$$\hat{\mathbf{A}} = A^1 \otimes A^2 \otimes \cdots \otimes A^N = \bigotimes_{i=1}^N A^i.$$

To use this, we need to implement the tensor product.

B.2.1 Tensor product code

Below is the code we use for calculating the tensor product of two square matrices. Input is the matrices A and B, output is C, which is returned in the final statement. The first three lines is the declaration of the method/function `outerProduct`.

```
Array<complex<double>,2>
Gate::outerProduct(Array<complex<double>,2> A,
                   Array<complex<double>,2> B){
    firstIndex i;
    secondIndex j;

    int dimA = A.rows();
    int dimB = B.rows();
    int dim = dimA*dimB;
    Array<complex<double>,2> C(dim, dim);

    for (int a=0; a < dimA; a++){
        for (int b=0; b < dimB; b++){
            C(Range(a*dimB, (a+1)*dimB - 1), Range(b*dimB, (b+1)*dimB - 1))
              = A(a,b)*B(i,j);
        }
    }
    return C;
}
```

B.2.2 Diagonalization

To diagonalize the matrix we can use several different methods for C++ or call fortran in our program, but we simply imported the matrix into matlab and used this short code to find the eigenvalues and eigenvectors.

```
clear;
load io.txt;
[V,E]=eig(io);
```

io.txt is the input file we generate in our C++ program, V is where the eigenvectors is stored, and E is where the eigenvalues is stored.

B.3 Simulation

For the simulation program we made three classes, however most non-standard objects used are from the Blitz++ package.

- **State** This class creates and stores the statevector of a system of appropriate size, it calculates the effects of operations and keeps count of the number of operations performed.
- **Gate** This class generates operators of correct dimension using tensor products. Some methods here are CNOT, CU and singleQubitGate.
- **Simulation** This is just a shell for the simulation program, and where all the physics is done. When a simulation object is initialized it generates the σ , Hadamard and 2×2 unity matrices. It reads in information from files to be used in the simulation. It also initializes two objects of the State class and two objects of the Gate class, one of each for the simulation qubits, and one set for the whole system. It then calls the different methods using the State and Gate objects in performing the simulation.

B.3.1 single-qubit gate

The code here belongs to a method called singleQubitGate. It calls on the outerProduct method shown above.

In the program the call to generate a matrix corresponding to a single qubit operation and multiplying it with the state vector is

```
state->operate(singleGate(gate, x, i);
```

Where state is a pointer to the State object representing the state vector of the system, and gate is a pointer to the Gate object with all the methods for generating operators. In the singleGate method, the method singleQubitGate in the Gate object is called, and this again calls the outerProduct method like this:

```
Array<complex<double>,2>
Gate::singleQubitGate(Array<complex<double>,2> I,
                      int qOperatedOn) {

    Array<complex<double>,2> temp(dimension,dimension);
    qubitOperatedOn = qOperatedOn;
    /*Dimensions of the unity matrices seen in eq. 5.3 */
    dima = (int) pow(2.0,(double)(qubitOperatedOn - 1));
    dimb = (int) pow(2.0,(double)(noOfQubits - qubitOperatedOn));

    Range a(firstDim, dima -1), b(dimension - dimb , dimension -1),
        tmp(dimension - 2*dimb , dimension -1), dim(firstDim, dimension-1);

    temp = 0;
    for (int i=0; i < dimension; i++){
        temp(i,i)=1;
    }/* Creates a unity matrix which we split with Range(,) to create proper
        dimensional unity matrices*/

    if( qubitOperatedOn == secondDim) {
        temp = outerProduct( I, temp(b, b));
    }//if
    /*Only need one outerProduct because first qubit is operated on,
        secondDim≡ 1*/

    else{
        if( qubitOperatedOn == noOfQubits){
            temp = outerProduct( temp(a, a), I);
        }//Here it's the last qubit we operate on

        else{
            temp(tmp, tmp) = outerProduct( I, temp(b, b));
            temp = outerProduct(temp(a,a), temp(tmp,tmp));
```

```

    }//else
} //else

return temp;

} //end of singleQubitGate method

```

Now, using this method for all singlequbit operations and a similar one using the recipe for controlled operations in appendix 5.1.3, any simulation of a unitary operation on a quantum computer can be simulated on our classical computer.

B.4 The measurement

Here we show the implementation of the algorithm found in section 3.5.

```

for (int i=0; i < noOfMeasurements; i++){
    m = (double) rand()/RAND_MAX;

    j = 0;
    double totalProb = (*probability)(j);
    while( m > totalProb){

        j++;
        totalProb += (*probability)(j);

    } //while
    (*measured)(i) = j;
    (*timesMeasured)(j) += 1;

} //for

```

Probability is here a pointer to an array containing the probability amplitudes of each state. Measured is an array with an element for every measurement telling what the result was of that particular measurement. TimesMeasured is an array telling how many times the different values of the work qubits have been measured.

Rand

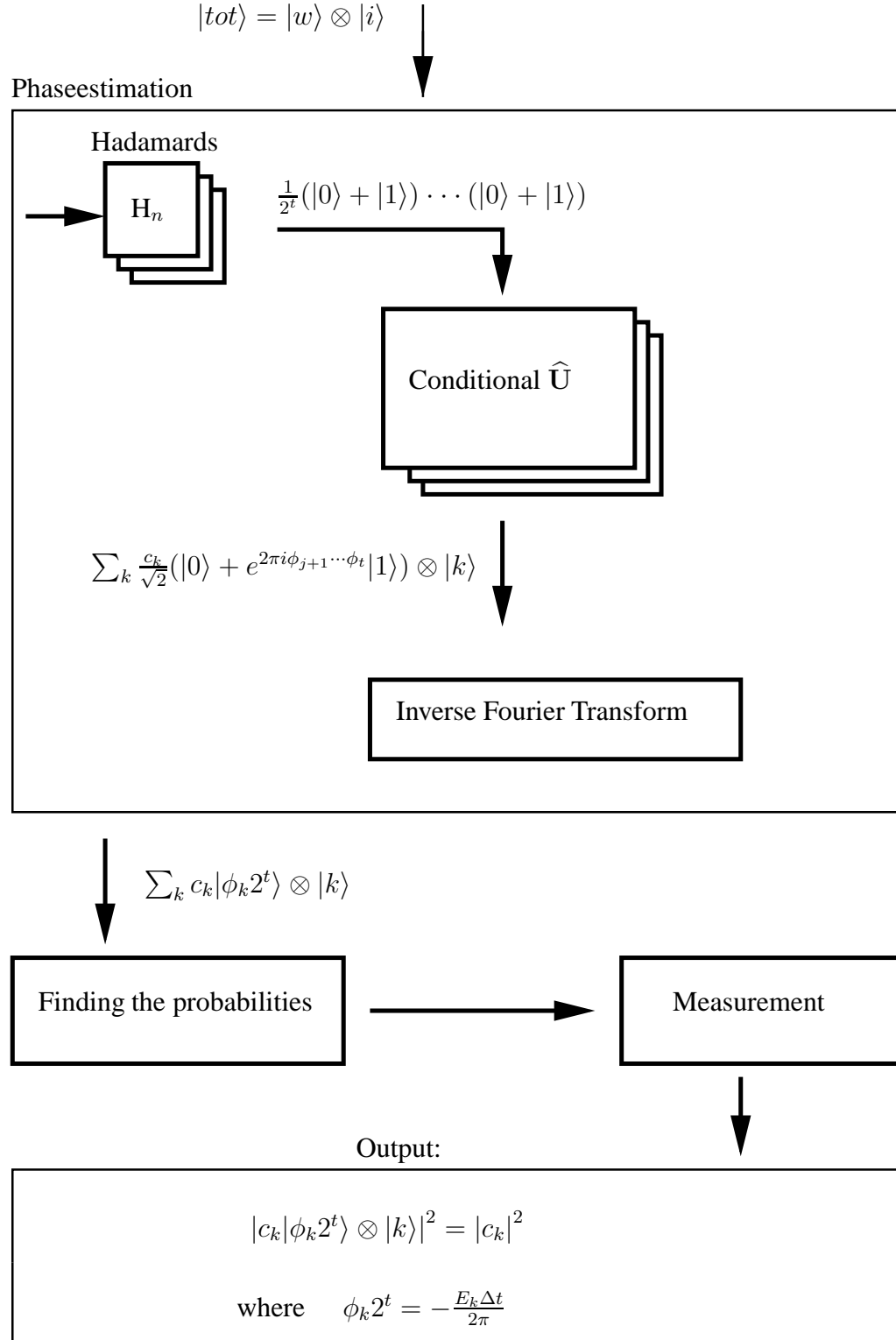
The *rand* method generates an integer from zero to a large number *RAND_MAX*, by dividing by *RAND_MAX* we get a number between zero and one. The

rand method does of course not generate a random number, but calculates one from the previous giving a sequence that is distributed uniformly between the bounds. By seeding the method with the system time we get a different series every time.

B.5 Flow chart of the program

In this section we give a graphical representation of the flow of data in our program. First we have the input state $|w\rangle \otimes |i\rangle$, and we see how the state evolves throughout the different parts of the program. The time evolution is the simulation algorithm described in the thesis. Then we apply the phase estimation algorithm, which includes the discrete inverse Fourier transform . Then a purely computational algorithm needed to find the probability of each state, so that we can simulate measurements in the last bulk of the program.

B.5.1 Outline of the program



Appendix C

Program listing

C.1 Simulation class

C.1.1 Header file

```
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <blitz/array.h>
#include <complex>
#include <fstream.h>
#include "State.h"
#include "Gate.h"

using namespace blitz;

#ifndef Simulation_H
#define Simulation_H

const double pi = 3.141592653589793238;

class Simulation{

public:
    int operationsPerformed;           /*We add one for each gate simulated*/
    int dimension;                    /*Dimension of state, 2^(noOfQubits)*/
    int noOfQubits;                   /*Total number of qubits*/
    int workQubits;                   /*Number of work qubits*/
    int simQubits;                    /*Number of qubits used in the direct simulation*/
```

```

int firstSimQubit;           /*Here is the first non-work qubit, before this
                                they all are work, after and including we have
                                simulation qubits.*/

int workDim;
int simDim;

Gate* qGate;                 /*This is the pointer to the gates we use*/
Gate* sGate;                 /*This is used to create gates for the phaseestimation*/
State* qState;               /*This is the pointer to the state of our system*/
State* sState;
/*Pauli matrices and Hadamard*/
Array<complex<double>,2> x, y, z, H, I;

/*An array with probabilities of eigenstates, and one with the eigenstates*/
Array<double,1>* probability;
Array<complex<double>,1>** eigenstates;

/*An array containing the measured eigenvalues*/
Array<double,1>* measured;

/*An array with the number of times each eigenvalue is measured, to calculate
the statistical probability of each eigenvalue used in the simulation of
measurements*/
Array<int,1>* timesMeasured;

/*A method to calculate the inner product of a vector*/
double innerProduct(Array<complex<double>,1> A);

void checkNormalisation();

/*ConSTRUCTOR!!!!!!!!!!*/
Simulation(const char* simFile, const char* vFile,
           const char* mFile);

/*This is where all the commands are given, we have put them in a method for
clarity*/
void simulate();
void readFromFile(const char* simFile);
void writeToFile(const char* simFile);
void writeOutEnergies();

```

```

/*Below is declared the methods that operate on the state, calling
on gates from the gate class using the pointer declared above*/

Array<complex<double>,2> singleGate(Gate*
gatePointer,Array<complex<double>,2> E,
                                int target);
Array<complex<double>,2> CNOT(Gate* gatePointer, int c, int x);
Array<complex<double>,2> NOTC(Gate* gatePointer, int x, int c);
Array<complex<double>,2> CU(Gate* gatePointer,
Array<complex<double>,2> U, int c,
                        int u);
Array<complex<double>,2> UC(Gate* gatePointer,
Array<complex<double>,2> U, int u,
                        int c);

void Z(int target, State* state, Gate* gate);
void XX(int target, State* state, Gate* gate);
void YY(int target, State* state, Gate* gate);
void ZZ(int target, State* state, Gate* gate);

void CZ(int control, int target);
void CXX(int control, int target);
void CYY(int control, int target);
void CZZ(int control, int target);

/*Below are the methods which run through a cycle depending on the size of
the system and call the above methods and other gates to run
the simulation, phaseestimation and inverse Fourier transform.*/

void U(State* state, Gate* gate); /*This runs the simulation*/

/*This runs  $U^{\wedge}(\text{exponent})$  conditional on control qubit being  $|1\rangle$ */
void conditionalU(int controlQubit, int exponent);
void phaseEst(); /*The phaseestimation*/
void inverseFourierTransform();
void fourierTransform();
void findingProbability();
void measurement();

```

```

    /*This is an alternative to CU*/
    void altCU(int controlQubit, int exponent);

private:
    double origDeltaT;
    double phaseMultiplier;
    double deltaT;           /*This is n*time intervall from the time evolution
                               operator*/
    double h;                /*The parameters of the Hamilton, h is h/(+/-J)*/
    int noOfTimesteps;       /*NoOf times we run the time evolution with the same
timestep*/
    int noOfMeasurements;    /*NoOf times we prepare the same state and
measure*/
    double Emax;
};

#endif

```

C.1.2 CPP file

```

#include "Simulation.h"
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <complex>
#include <fstream.h>
#include <blitz/array.h>

```

```

using namespace blitz;

```

```

void Simulation::simulate(){

```

```

    Array<complex<double>,2> temp(dimension, dimension);
    temp = 0;
    for(int i=0; i < dimension; i++)

```

```

    temp(i,i)=1;

    qState→m.reference(temp);

    /*Running the tests*/
    //U(qState, qGate);
    /*cout << "This is the state of the system after it has run through U: \n"
       << qState->vector << endl;*/
    phaseEst();

    inverseFourierTransform();

    findingProbability();
    measurement();
    writeOutEnergies();

    cout << "\nNumber of operations performed is: " <<
    qState→operationsPerformed
        <<endl;

} //test

Simulation::Simulation(const char* simFile, const char* vFile,
                      const char* mFile){

    /*Initilization of method parameters and Hamiltonian parameters*/
    readFromFile(simFile);

    phaseMultiplier = 1;

    simDim = (int) pow(2.0, (double)(noOfQubits - firstSimQubit + 1));
    workDim = (int) pow(2.0, (double)(firstSimQubit - 1));

    /*Initialization of Gate class*/
    qGate = new Gate(noOfQubits);
    sGate = new Gate(noOfQubits - firstSimQubit + 1);

```

```

    cout << "Dimension of qGate:  " << qGate→dimension << ", and
sGate:  " <<
    sGate→dimension << endl;

```

```

    /*Initialization of State class*/

```

```

    qState = new State(noOfQubits, firstSimQubit);
    sState = new State(noOfQubits-firstSimQubit+1, 1);
    dimension = qGate→dimension;

```

```

    /*Initialization of paulis and Hadamard, matrices used throughout the
program*/

```

```

    Array<complex<double>,2> tempx(2,2), tempy(2,2), tempz(2,2),
tempH(2,2), tempI(2,2);
    tempx = 0,1,
        1,0;
    x.reference(tempx);
    tempy = 0, complex<double> (0, -1),
        complex<double> (0, 1), 0;
    y.reference(tempy);
    tempz = 1, 0,
        0, -1;
    z.reference(tempz);
    tempH = 1, 1,
        1, -1;
    tempH /= sqrt(2);
    H.reference(tempH);
    tempI=1,0,
        0,1;
    I.reference(tempI);

```

```

    /*This is where the simulation is started*/

```

```

    simulate();

```

```

} //End of constructor

```

```

void Simulation::readFromFile(const char* simFile){

```

```

    ifstream ifs(simFile);
    if (ifs.bad())

```



```

    {
        cerr << "Unable to open file:  " << simFile << endl;
        exit(1);
    }

    ifs >> noOfQubits >> firstSimQubit >> h >> deltaT >> Emax >>
noOfTimesteps >>
        noOfMeasurements;

    cout << "Number of qubits:" << noOfQubits << "\nFirst simulation
qubit "
        << firstSimQubit
        << "\nh " << h << "\nTimestep "
        << deltaT << "\nEmax " << Emax << "\nNo.  of timesteps " <<
noOfTimesteps
        << "\nNo.  of measurements " << noOfMeasurements << endl;

    origDeltaT = deltaT;

} //readFromFile

void Simulation::writeToFile(const char* filename){

    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    ofs << noOfQubits << " " << firstSimQubit << " " << h << " " <<
        deltaT << endl;

} //writeToFile

```

```

/*This is the time evolution*/
void Simulation::U(State* state, Gate* gate){
    deltaT=deltaT/2;
    Array<complex<double>,2> AB(2,2);
    AB=I*exp(complex<double> (0,Emax*deltaT));

    for(int a=0; a < noOfTimesteps; a++){
        state→operate(singleGate(gate, AB, state→firstSimQubit));
        for(int i=state→firstSimQubit; i ≤ state→noOfQubits; i++){
            Z(i, state, gate);
            ZZ(i, state, gate);
            YY(i, state, gate);
            XX(i, state, gate);
            XX(i, state, gate);
            YY(i, state, gate);
            ZZ(i, state, gate);
            Z(i, state, gate);
        }//for
        state→operate(singleGate(gate, AB, state→firstSimQubit));
    }//for
    deltaT=deltaT*2;
} //U

```

```

void Simulation::conditionalU(int controlQubit, int exponent){

    phaseMultiplier = exponent;
    Array<complex<double>,2> AC(2,2);
    AC=I*exp(complex<double> (0,Emax*deltaT*phaseMultiplier));

    for(int a=0; a < noOfTimesteps; a++){
        qState→operate(CU(qGate, singleGate(sGate, AC,
sState→firstSimQubit),controlQubit,firstSimQubit));
        for(int i=firstSimQubit; i ≤ noOfQubits; i++){

```

```

        CZ(controlQubit, i);
        CZZ(controlQubit, i);
        CYY(controlQubit, i);
        CXX(controlQubit, i);
    }//for
} //for

} //conditional U

/*Here we run the phase estimation algorithm*/
void Simulation::phaseEst(){

    /*Hadamard on each work qubit*/
    for(int i = 1; i < firstSimQubit; i++)
        qState→operate(singleGate(qGate, H, i));

    /*Controlled big  $U^{2^t}$ */
    // int tempNoOfOperations = sState→operationsPerformed;
    // for(int t = 1; t < firstSimQubit; t++)
    // conditionalU(firstSimQubit - t, (int) pow(2.0, (double)(t-1)));

    // sState→operationsPerformed=
    (sState→operationsPerformed-tempNoOfOperations)*6
    // + tempNoOfOperations;

    /*Alternative phaseEst, faster*/
    Array<complex<double>,2> tomp(simDim, simDim);
    tomp = 0;
    for(int i=0; i < simDim; i++)
        tomp(i,i)=1;

    for(int t = 1; t < firstSimQubit; t++){
        sState→m.reference(tomp);
        altCU(firstSimQubit - t, (int) pow(2.0, (double)(t-1)));
    }
} //phase estimation

```

```

Array<complex<double>,2> Simulation::singleGate(Gate* gatePointer,
        Array<complex<double>,2> E, int target){

    return(gatePointer→singleQubitGate(E, target));

} //singleGate

Array<complex<double>,2> Simulation::CNOT(Gate* gatePointer, int c,
int x){

    return(gatePointer→gCNOT(c, x));

} //CNOT

Array<complex<double>,2> Simulation::NOTC(Gate* gatePointer, int x,
int c){

    return(gatePointer→gNOTC(x, c));

} //NOTC

Array<complex<double>,2> Simulation::CU(Gate* gatePointer,
Array<complex<double>,2> U, int c, int u)

    return(gatePointer→gCU(U, c, u));

} //CU

Array<complex<double>,2> Simulation::UC(Gate* gatePointer,
Array<complex<double>,2> U, int u, int c)

    return(gatePointer→gUC(U, u, c));

} //CU

void Simulation::Z(int target, State* state, Gate* gate){

```

```

    Array<complex<double>,2> Rz(2,2); //Because it is the rotation
operator
    Rz = exp(complex<double> (0,-1)*h*deltaT),0,
        0,exp(complex<double> (0,1)*h*deltaT);

    state→operate(singleGate(gate, Rz, target));

} //Z

```

```

void Simulation::XX(int target, State* state, Gate* gate){

```

```

    Array<complex<double>,2> A(2,2);
    A = cos(deltaT), complex<double> (0, -sin(deltaT)),
        complex<double> (0, -sin(deltaT)), cos(deltaT);

```

```

    if(target ≠ state→noOfQubits){
        state→operate(singleGate(gate, x, target));
        state→operate(CNOT(gate, target, target + 1));
        state→operate(singleGate(gate, x, target));

```

```

        state→operate(singleGate(gate, x, target +1));
        state→operate(UC(gate, A, target, target +1));
        state→performOperations(5);
        state→operate(singleGate(gate, x, target +1));

```

```

        state→operate(UC(gate, A, target, target +1));
        state→performOperations(5);

```

```

        state→operate(singleGate(gate, x, target));
        state→operate(CNOT(gate, target, target + 1));
        state→operate(singleGate(gate, x, target));

```

```

    } // if

```

```

    /*Bound ends condition*/

```

```

    else{
        state→operate(singleGate(gate, x, state→noOfQubits));
        state→operate(NOTC(gate, state→firstSimQubit,

```

```

state→noOfQubits));
    state→operate(singleGate(gate, x, state→noOfQubits));

    state→operate(singleGate(gate, x, state→firstSimQubit));
    state→operate(CU(gate, A, state→firstSimQubit,
state→noOfQubits));
    state→performOperations(5);
    state→operate(singleGate(gate, x, state→firstSimQubit));

    state→operate(CU(gate, A, state→firstSimQubit,
state→noOfQubits));
    state→performOperations(5);

    state→operate(singleGate(gate, x, state→noOfQubits));
    state→operate(NOTC(gate, state→firstSimQubit,
state→noOfQubits));
    state→operate(singleGate(gate, x, state→noOfQubits));
} //else

} //XX

```

```

void Simulation::YY(int target, State* state, Gate* gate){

```

```

    Array<complex<double>,2> A(2,2), C(2,2);
    A = cos(deltaT), complex<double> (0, -sin(deltaT)),
        complex<double> (0, -sin(deltaT)), cos(deltaT);
    C = cos(deltaT), complex<double> (0, sin(deltaT)),
        complex<double> (0, sin(deltaT)), cos(deltaT);

```

```

if(target ≠ state→noOfQubits){
    state→operate(singleGate(gate, x, target));
    state→operate(CNOT(gate, target, target + 1));
    state→operate(singleGate(gate, x, target));

    state→operate(singleGate(gate, x, target + 1));
    state→operate(UC(gate, A, target, target + 1));
    state→performOperations(5);
    state→operate(singleGate(gate, x, target + 1));

    state→operate(UC(gate, C, target, target + 1));

```

```

state→performOperations(5);

state→operate(singleGate(gate, x, target));
state→operate(CNOT(gate, target, target + 1));
state→operate(singleGate(gate, x, target));
} //if

/*Bound ends condition*/
else{

state→operate(singleGate(gate, x, state→noOfQubits));
state→operate(NOTC(gate, state→firstSimQubit,
state→noOfQubits));
state→operate(singleGate(gate, x, state→noOfQubits));

state→operate(singleGate(gate, x, state→firstSimQubit));
state→operate(CU(gate, A, state→firstSimQubit,
state→noOfQubits));
state→performOperations(5);
state→operate(singleGate(gate, x, state→firstSimQubit));

state→operate(CU(gate, C, state→firstSimQubit,
state→noOfQubits));
state→performOperations(5);

state→operate(singleGate(gate, x, state→noOfQubits));
state→operate(NOTC(gate, state→firstSimQubit,
state→noOfQubits));
state→operate(singleGate(gate, x, state→noOfQubits));
} //else

} //YY

```

```

void Simulation::ZZ(int target, State* state, Gate* gate){

```

```

    Array<complex<double>,2> D(2,2), E(2,2);

```

```

    D = exp(complex<double> (0,2*deltaT)), 0,

```

```

0, exp(complex<double> (0,2*deltaT));

E = exp(complex<double> (0,-deltaT/2.0)), 0,
0, exp(complex<double> (0,-deltaT/2.0));

if(target  $\neq$  state→noOfQubits){
    state→operate(singleGate(gate, E, target));
    state→operate(singleGate(gate, E, target+1));

    state→operate(singleGate(gate, x, target));
    state→operate(CNOT(gate, target, target + 1));
    state→operate(singleGate(gate, x, target));

    state→operate(singleGate(gate, x, target + 1));
    state→operate(UC(gate, D, target, target + 1));
    state→performOperations(5);
    state→operate(singleGate(gate, x, target + 1));

    state→operate(singleGate(gate, x, target));
    state→operate(CNOT(gate, target, target + 1));
    state→operate(singleGate(gate, x, target));

} // if

/*Bound ends condition*/
else{
    state→operate(singleGate(gate, E, state→firstSimQubit));
    state→operate(singleGate(gate, E, state→noOfQubits));

    state→operate(singleGate(gate, x, state→noOfQubits));
    state→operate(NOTC(gate, state→firstSimQubit,
state→noOfQubits));
    state→operate(singleGate(gate, x, state→noOfQubits));

    state→operate(singleGate(gate, x, state→firstSimQubit));
    state→operate(CU(gate, D, state→firstSimQubit,
state→noOfQubits));
    state→performOperations(5);
    state→operate(singleGate(gate, x, state→firstSimQubit));

    state→operate(singleGate(gate, x, state→noOfQubits));

```



```

        state→operate(NOTC(gate, state→firstSimQubit,
state→noOfQubits));
        state→operate(singleGate(gate, x, state→noOfQubits));

    }//else

} //ZZ

```

```

void Simulation::CZ(int control, int target){

    int sTarget = target - firstSimQubit + 1;

    Array<complex<double>,2> Rz(2,2);//Name because it is the rotation
operator
    Rz = exp(complex<double> (0,-1)*h*deltaT*phaseMultiplier),0,
        0,exp(complex<double> (0,1)*h*deltaT*phaseMultiplier);

    qState→operate(CU(qGate, singleGate(sGate, Rz, sTarget), control,
firstSimQubit));

} //CZ

```

```

void Simulation::CXX(int control, int target){

    int sNoOfQ = noOfQubits - firstSimQubit +1;//number of simulation
qubits
    int sTarget = target - firstSimQubit + 1;
    Array<complex<double>,2> A(2,2);
    A = cos(deltaT*phaseMultiplier), complex<double> (0,
-sin(deltaT*phaseMultiplier)),
        complex<double> (0, -sin(deltaT*phaseMultiplier)),
cos(deltaT*phaseMultiplier);

```

```

if(target  $\neq$  noOfQubits){

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));
    qState→operate(CU(qGate, UC(sGate, A, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));

    qState→operate(CU(qGate, UC(sGate, A, sTarget, sTarget + 1),
control,
                                firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));

} // if

/*Bound ends condition*/
else{

    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));
    qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ), control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));

```

```

        qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));
        qState→operate(CU(qGate, CU(sGate, A, 1, sNoOfQ),control,
firstSimQubit));
        qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));
        qState→performOperations(5);

        qState→operate(CU(qGate, CU(sGate, A, 1, sNoOfQ),control,
firstSimQubit));
        qState→performOperations(5);

        qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ),control,
firstSimQubit));
        qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ),control,
firstSimQubit));
        qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ),control,
firstSimQubit));

    }//else
}//CXX

```

```

void Simulation::CYY(int control, int target){

    int sNoOfQ = noOfQubits - firstSimQubit +1;
    int sTarget = target - firstSimQubit + 1;

    Array<complex<double>,2> A(2,2), C(2,2);
    A = cos(deltaT*phaseMultiplier), complex<double> (0,
-sin(deltaT*phaseMultiplier)),
        complex<double> (0, -sin(deltaT*phaseMultiplier)),
cos(deltaT*phaseMultiplier);
    C = cos(deltaT*phaseMultiplier), complex<double> (0,
sin(deltaT*phaseMultiplier)),
        complex<double> (0, sin(deltaT*phaseMultiplier)),
cos(deltaT*phaseMultiplier);

```

```

if(target  $\neq$  noOfQubits){

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));
    qState→operate(CU(qGate, UC(sGate, A, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));

    qState→operate(CU(qGate, UC(sGate, C, sTarget, sTarget + 1),
control,
                                firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                                firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));

} // if

/*Bound ends condition*/
else{
    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));
    qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ), control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));

```

```

    qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));
    qState→operate(CU(qGate, CU(sGate, A, 1, sNoOfQ),control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));

    qState→operate(CU(qGate, CU(sGate, C, 1, sNoOfQ),control,
firstSimQubit));
    qState→performOperations(5);

    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ),control,
firstSimQubit));
    qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ),control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ),control,
firstSimQubit));

    }//else

} //CYY

```

```

void Simulation::CZZ(int control, int target){

    int sNoOfQ = noOfQubits - firstSimQubit +1;
    int sTarget = target - firstSimQubit + 1;
    Array<complex<double>,2> D(2,2), E(2,2);

    D = exp(complex<double> (0,2*deltaT*phaseMultiplier)), 0,
        0, exp(complex<double> (0,2*deltaT*phaseMultiplier));

    E = exp(complex<double> (0,-deltaT*phaseMultiplier/2.0)), 0,
        0, exp(complex<double> (0,-deltaT*phaseMultiplier/2.0));

    if(target ≠ noOfQubits){

        qState→operate(CU(qGate, singleGate(sGate, E, sTarget), control,
firstSimQubit));
    }
}

```

```
    qState→operate(CU(qGate, singleGate(sGate, E, sTarget+1), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                        firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, UC(sGate, D, sTarget, sTarget + 1),
control,
                        firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget + 1), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
```

```
    qState→operate(CU(qGate, CNOT(sGate, sTarget, sTarget + 1),
control,
                        firstSimQubit));
```

```
    qState→operate(CU(qGate, singleGate(sGate, x, sTarget), control,
firstSimQubit));
```

```
    }//if
```

```
    /*Bound ends condition*/
```

```
    else{
```

```
        qState→operate(CU(qGate, singleGate(sGate, E, sNoOfQ), control,
firstSimQubit));
```

```
        qState→operate(CU(qGate, singleGate(sGate, E, 1), control,
firstSimQubit));
```

```
        qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));
```

```
        qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ),control,
firstSimQubit));
```

```
        qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
```

```

firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));
    qState→operate(CU(qGate, CU(sGate, D, 1, sNoOfQ),control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, 1),control,
firstSimQubit));

    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));
    qState→operate(CU(qGate, NOTC(sGate, 1, sNoOfQ),control,
firstSimQubit));
    qState→operate(CU(qGate, singleGate(sGate, x, sNoOfQ), control,
firstSimQubit));

    }//else

} //CZZ

void Simulation::inverseFourierTransform(){

    Array<complex<double>,2> Rk(2,2);

    for(int i=1; i < firstSimQubit; i++){
        /*Rks for the below work qubits*/
        for(int j=1; j < i; j++){
            Rk = 1, 0,
                0, exp(complex<double> (0, -2*pi/pow(2.0,(double)(i-j + 1))));
            qState→operate(CU(qGate, Rk, j, i));
        }//for
        qState→operate(singleGate(qGate, H, i));
    }//for

} //IFT

```

```

void Simulation::fourierTransform(){

```

```

Array<complex<double>,2> Rk(2,2);

for(int k=firstSimQubit -1; k≥1; k--){
    qState→operate(singleGate(qGate, H, k));

    for(int l = k-1; l≥1; l--){
        Rk = 1, 0,
        0, exp(complex<double> (0, 2*pi/pow(2.0,(double)( k-l+1))));

        qState→operate(CU(qGate, Rk, l,k ));
    }//for
} //for

} //FT

/*This method is for the alternative phase estimation*/
void Simulation::altCU(int controlQubit, int exponent){
    phaseMultiplier = exponent;
    deltaT=deltaT*phaseMultiplier;
    U(sState, sGate);
    qState→operate(CU(qGate, sState→m, controlQubit, firstSimQubit));

    deltaT=origDeltaT;

} //altCU

```

Second file to keep files small

```

#include "Simulation.h"
#include "Gate.h"
#include "State.h"
#include <time.h>
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <complex>
#include <fstream.h>
#include <blitz/array.h>

```



```
using namespace blitz;
```

```
void Simulation::findingProbability(){

    int slot = 0;

    Array<complex<double>,2> unitySim(simDim, simDim),
    phiMatrix(workDim, workDim),
        projector(dimension, dimension);

    Array<complex<double>,1> temp(dimension);

    /*The probabilities*/
    probability = new Array<double,1>(workDim);
    *probability =0;

    /*This is the projector operator*/
    projector = 0;

    /*This is the unity operator on the simQubits*/
    unitySim = 0;
    for(int i = 0; i < simDim; i++)
        unitySim(i,i) = 1;

    /*This is the projector on the work qubits*/
    phiMatrix = 0;

    /*Declaring matrix to hold eigenstates*/
    eigenstates = new Array<complex<double>,1>* [workDim];
    for (int i=0; i < workDim; i ++)
        eigenstates[i] = new Array<complex<double>,1>(dimension);

    *eigenstates[0] = qState→vector;

    /*Projecting to find probabilities*/
    for(int k = 0; k < workDim; k++){
```

```

    Range nonZero(k*simDim, (k+1)*simDim - 1);
    projector(nonZero, nonZero) = unitySim;

    *eigenstates[k]=(qGate→matVecMult(projector, qState→vector));
    projector(nonZero, nonZero) = 0;

    /*Finding the probabilities for each eigenvalue*/
    (*probability)(k) = innerProduct(*eigenstates[k]);

} //for

cout << "Probability " << *probability << endl;

/*Here is a test to see if the probabilitis sum up to one*/
double sum = 0;
for (int i = 0; i < workDim; i++)
    sum += (*probability)(i);

cout << "Total probability: " << sum << endl;

/*Writing probabilities and eigenstates to file*/
qGate→writeToFile("eigenstatesNot.data", eigenstates, workDim);

for(int i=0; i < workDim; i++){

    *eigenstates[i] = *eigenstates[i]/sqrt((*probability)(i));
    //cout << "Normalised eigenstate\n" << *eigenstates[i] << endl;
    qGate→writeToFile("normalizedEigenstates.data", eigenstates,
workDim);

}

/*Saving the probabilities*/
qGate→writeToFile("probability.data", probability);

} //probability

void Simulation::measurement(){

```

```

double m=0;
int j;

/*This is where each measurement goes*/
measured = new Array<double,1>(noOfMeasurements);

/*Times each eigenvalue is measured*/
timesMeasured = new Array<int,1>(workDim);
*timesMeasured = 0;

/*The random number generator is seeded with the system time*/
srand(time(0));

/*Generate a random number to pick an eigenvalue*/
for(int i=0; i < noOfMeasurements; i++){
    m = (double) rand()/RAND_MAX;

    j = 0;
    double totalProb = (*probability)(j);
    while( m > totalProb){

        j++;
        totalProb += (*probability)(j);

    }//while
    (*measured)(i) = j;
    (*timesMeasured)(j) += 1;

//for

qGate→writeToFile("everyMeasurement.data", measured);
qGate→writeToFile("frequency.data", timesMeasured);

cout << "\n\nHere is the number of times each eigenvalue is
measured:  \n"
        << *timesMeasured << endl;

/*Here we calculate the measured probability*/
Array<double,1> probs(workDim);

```

```

for(int i=0; i < workDim; i++)
    probs(i) = (double)(*timesMeasured)(i)/(double) noOfMeasurements;

cout << "\nThe measured probability:\n" << probs << endl;
qGate→writeToFile("measuredProb.data", &probs);

} //measurement

double Simulation::innerProduct(Array<complex<double>,1> A){

    double c=0.;

    for(int i=0; i < A.rows(); i++){

        c += real(A(i))*real(A(i));
        c += imag(A(i))*imag(A(i));

    }

    return c;

} //innerProduct

void Simulation::checkNormalisation(){
    /*This method checks whether the state is normalised, taking the
       inner product of the state-vector itself*/

    double prob = innerProduct(qState→vector);

    cout << "The inner product of the state is :  " << prob << endl;

} //checkNormalisation

void Simulation::writeOutEnergies(){

```

```

    /*This method writes out the energies corresponding to the binary values of
       the work qubits*/

    double time =deltaT*noOfTimesteps;
    /*Help variables*/
    double fie;
    int tamp;
    Array<int,1> wb(firstSimQubit - 1);
    Array<double,1> energies(workDim), phi(workDim),
energyScale(workDim),
    probOfEnergy(workDim);

    for(int i=0; i<workDim; i++){
        wb=0;
        fie=0;
        tamp=i;
        for(int j=0; j<firstSimQubit - 1; j++){
            wb(j)=tamp%2;
            tamp = tamp/2;
            fie+=wb(j)*pow(2.0, (double)(firstSimQubit - 1-j -1));
        }//for

        phi(i)=(double)fie/(double)workDim;
        energies(i) = phi(i)*(-2*pi/time);
        energyScale((int)fie)=energies(i);

        probOfEnergy((int)fie)=(*probability)(i);

    }//for

    for(int k=0; k<workDim; k++)
        energyScale(k)=energyScale(k)+Emax;

    qGate→writeToFile("scale.data", &energyScale);
    qGate→writeToFile("EnergyProb.data", &probOfEnergy);
    qGate→writeToFile("energies.data", &energies);
    qGate→writeToFile("phis.data", &phi);

} //writeOutEnergies

```

C.2 Gate class

C.2.1 Header file

```

#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <blitz/array.h>
#include <complex>
#include <fstream.h>

using namespace blitz;

#ifndef Gate_H
#define Gate_H

class Gate{

public:
    Array<complex<double>,2> matrix;
    int noOfQubits, dimension;
    int qubitOperatedOn, dima, dimb, dim;

    Gate(int noOfQ);

    void writeToFile(const char* filename);
    void writeToFile(const char* filename, Array<complex<double>,1>**
array, int dim);
    void writeToFile(const char* filename, Array<double,1>* in);
    void writeToFile(const char* filename, Array<int,1>* in);
    void writeToFile(const char* filename, Array<complex<double>,1>*
in);

    /*Returns the matrix of the class*/
    Array<complex<double>,2> matrixx();

    /*Calculates the tensor product of two matrices*/

```

```

    Array<complex<double>,2>
outerProduct(Array<complex<double>,2> A,
              Array<complex<double>,2>
B);
    /*Creates a single qubit gate*/
    Array<complex<double>,2>
singleQubitGate(Array<complex<double>,2> I,
                int qOperatedOn);
    /*Performs U on the second qubit conditional on the first being |1>*/
    Array<complex<double>,2> gCU(Array<complex<double>,2> U, int
c,
                                int u);
    /*Performs U on the first qubit conditional on the second being |1>*/
    Array<complex<double>,2> gUC(Array<complex<double>,2> U, int
u,
                                int c);
    /*Performs X on the second qubit conditional on the first being |1>*/
    Array<complex<double>,2> gCNOT(int c, int x);

    /*Performs X on the first qubit conditional on the second being |1>*/
    Array<complex<double>,2> gNOTC(int x, int c);

    /*Matrix multiplication*/
    Array<complex<double>,2> mult(Array<complex<double>,2> A,
Array<complex<double>,2> B);

    /*Matrix*vector*/
    Array<complex<double>,1> matVecMult(Array<complex<double>,2>
A,
                                        Array<complex<double>,1>
B);

};

#endif

```

C.2.2 CPP file

```
#include "Gate.h"
#include "State.h"
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <complex>
#include <fstream.h>
#include <blitz/array.h>
```

```
using namespace blitz;
```

```
Gate::Gate(int noOfQ){

    noOfQubits = noOfQ;
    dimension = (int)pow(2.0, (double)(noOfQubits));

} //conSTRUCTOR!!!
```

```
void Gate::writeToFile(const char* filename) {

    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file: " << filename << endl;
        exit(1);
    }

    ofs << matrix << endl;

} //writeToFile
```



```

Array<complex<double>,2>
Gate::singleQubitGate(Array<complex<double>,2> I,
                      int qOperatedOn) {
    /*Creates a matrix corresponding to a single qubit gate*/

    Array<complex<double>,2> temp(dimension,dimension);

    qubitOperatedOn = qOperatedOn;

    dima = (int) pow(2.0,(double)(qubitOperatedOn - 1)); //exponent
    dimb = (int) pow(2.0,(double)(noOfQubits - qubitOperatedOn));

    Range a(firstDim, dima - 1), b(dimension - dimb , dimension - 1),
        tmp(dimension - 2*dimb , dimension - 1), dim(firstDim, dimension-1);

    temp = 0;
    for (int i=0; i < dimension; i++){
        temp(i,i)=1; //unity matrix for whole system
    }//for

    if( qubitOperatedOn == secondDim) {
        temp = outerProduct( I, temp(b, b));

    }//if
    //Only need one outerProduct because first qubit is operated on

    else{
        if( qubitOperatedOn == noOfQubits){

            temp = outerProduct( temp(a, a), I);

        }//Here it's the last qubit we operate on

        else{
            temp(tmp, tmp) = outerProduct( I, temp(b, b));
            temp = outerProduct(temp(a,a), temp(tmp,tmp));
            //This is OP tensor identity matrix for the rest of the qubits
        }//else
    }
}

```

```

    }//else

    return temp;

} //singleQubitGate


Array<complex<double>,2> Gate::gCU(Array<complex<double>,2> U,
int c, int u){

    /*In the case of multiqubit U operators, u is defined as the first of
    the qubits the Gate operate on, only one qubit Gates or next neighbour
    Gates are available. That is why we have two sets of dimensions for the
    help matrices*/

    int a, b, d, e, uDim;
    b = 1;
    uDim = U.rows();

    a = (int) pow(2.0,(double)(u - c - 1)); //dimensions of help matrices
    d = (int) pow(2.0,(double)(noOfQubits - c + 1));
    e = (int) pow(2.0,(double)(c - 1));

    Array<complex<double>,2> temp(dimension, dimension),
        temp2(dimension, dimension), temp3(dimension, dimension),
        upper(thirdDim, thirdDim),
        lower(thirdDim, thirdDim); // help matrices


    temp3 = 0;
    temp2 = 0;
    temp = 0;
    for (int i=0; i < dimension; i++){
        temp2(i,i)=1; //unity matrix for whole system
    } //for

    upper = 1,0,
        0,0;//This is part of the Control operation

```

```

    lower = 0,0,
        0,1;

    temp3 = singleQubitGate(upper, c);

    //Ranges for help matrices

    Range two(firstDim, 2*a - 1), I2(firstDim, a-1), three(firstDim, d - 1),
        I3(firstDim, e - 1);

    b = (int) pow(2.0, (double)(noOfQubits - u + 1));

    Range one(dimension - b, dimension -1), I1(dimension - b/uDim,
dimension -1);

    temp(one,one) = outerProduct(U, temp2(I1, I1));
    temp(two,two) = outerProduct(lower, temp2(I2,I2));
    temp(three, three) = outerProduct( temp(two, two), temp(one, one));
    temp = outerProduct(temp2(I3, I3), temp(three, three));
    temp3 += temp;

    return temp3;
} //CU

Array<complex<double>,2> Gate::gUC(Array<complex<double>,2> U,
int u,
                                int c){
    /*In the case of multiqubit U operators, u is defined as the first of
    the qubits the Gate operate on, only one qubit Gate or next neighbour
    Gates are available. That is why we have two sets of dimensions for the
    help matrices*/

    int a, b, d, e, uDim;

```

```

b = 1;
uDim = U.rows();

//dimensions of help matrices, independent of size of U
b = (int) pow(2.0, (double)(noOfQubits - c));
d = (int) pow(2.0, (double)(noOfQubits - u + 1));
e = (int) pow(2.0, (double)(u - 1));

Array<complex<double>,2> temp(dimension, dimension),
    temp2(dimension, dimension), temp3(dimension, dimension),
    upper(thirdDim, thirdDim),
    lower(thirdDim, thirdDim); // help matrices

temp3 = 0;
temp2 = 0;
temp = 0;
for (int i=0; i < dimension; i++){
    temp2(i,i)=1; //unity matrix for whole system
} // for

upper = 1,0,
    0,0; //This is part of the Control operation

lower = 0,0,
    0,1;

temp3 = singleQubitGate(upper, c);

//Ranges for help matrices

Range one(dimension - 2*b, dimension - 1), I1(dimension - b, dimension
-1),
    three(firstDim, d - 1), I3(firstDim, e - 1);

if(uDim==2){
    a = (int) pow(2.0, (double)(c - u - 1));
    Range two(firstDim, 2*a - 1), I2(firstDim, a-1);

```

```

temp(one,one) = outerProduct(lower, temp2(I1, I1));
temp(two,two) = outerProduct(U, temp2(I2,I2));
temp(three, three) = outerProduct( temp(two, two), temp(one, one));
temp = outerProduct(temp2(I3, I3), temp(three, three));
temp3 += temp;

} //if
else{
    a = (int) pow(2.0, (double)(c - u - 2));
    Range two(firstDim, 4*a - 1), I2(firstDim, a-1);

    temp(one,one) = outerProduct(U, temp2(I1, I1));
    temp(two,two) = outerProduct(lower, temp2(I2,I2));
    temp(three, three) = outerProduct( temp(two, two), temp(one, one));
    temp = outerProduct(temp2(I3, I3), temp(three, three));

    temp3 += temp;

} //else

return temp3;

} //UC

```

```

Array<complex<double>,2>
Gate::outerProduct(Array<complex<double>,2> A,
                    Array<complex<double>,2>
B){

    firstIndex i;
    secondIndex j;

    int dimA = A.rows();
    int dimB = B.rows();
    int dim = dimA*dimB;

```

```

Array<complex<double>,2> C(dim, dim), D(dimB, dimB);

for (int a=0; a < dimA; a++){
    for (int b=0; b < dimA; b++){

        C(Range(a*dimB, (a+1)*dimB - 1), Range(b*dimB, (b+1)*dimB
-1)) = A(a,b)*B(i,j);

    }
}

return C;

} //outerProduct

Array<complex<double>,2> Gate::gCNOT(int c, int x){

    Array<complex<double>,2> X(2,2);
    X = 0,1,
        1,0;

    return gCU(X, c, x);

} //CNOT

Array<complex<double>,2> Gate::gNOTC(int x, int c){

    Array<complex<double>,2> X(2,2);
    X = 0,1,
        1,0;

```

```

    return gUC(X, x, c);

} // CNOT

Array<complex<double>,2> Gate::matrixx(){

    return matrix;

} // matrixx

Array<complex<double>,2> Gate::mult(Array<complex<double>,2> A,
                                   Array<complex<double>,2> B){
    /*This method multiplies two square matrices*/

    Array<complex<double>,2> temp(A.rows(), A.rows());

    if(A.rows()  $\neq$  B.rows()){
        cout << "Matrices do not match, multiplication impossible."
    << endl;
        return temp;
    } // if

    firstIndex i;
    secondIndex j;
    thirdIndex k;

    temp = sum(A(i,k)*B(k,j), k);

    return temp;

} // mult

```

```

Array<complex<double>,1>
Gate::matVecMult(Array<complex<double>,2> A,
                  Array<complex<double>,1> B){
    /*This method multiplies a matrix to a vector, and simulates an
       operator on a state*/

    Array<complex<double>,1> temp(A.rows());
    temp = 0;

    if(A.rows()  $\neq$  B.rows()){
        cout << "Matrixx does not match vector, multiplication
impossible." << endl;
        return temp;
    }//if

    firstIndex i;
    secondIndex j;

    temp = sum(A(i,j)*B(j), j);

    return temp;
} //matVecMult

/*Different writeToFile methods to handle different datastructures*/
void Gate::writeToFile(const char* filename,
Array<complex<double>,1>** array, int dim){

    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    for(int i=0; i < dim; i++)
        ofs << *array[i] << endl;

```



```
}//write
```

```
void Gate::writeToFile(const char* filename, Array<double,1>* in){
```

```
    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    ofs << *in << endl;
```

```
}//write
```

```
void Gate::writeToFile(const char* filename, Array<int,1>* in){
```

```
    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    ofs << *in << endl;
```

```
}//write
```

```
void Gate::writeToFile(const char* filename,
Array<complex<double>,1>* in){
```

```

    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    ofs << *in << endl;

} //write

```

C.3 State class

C.3.1 Header file

```

#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <blitz/array.h>
#include <complex>
#include <fstream.h>
#include "Gate.h"

using namespace blitz;

#ifndef State_H
#define State_H

class State {

public:
    int operationsPerformed;           /*We add one for each gate simulated*/
    int dimension;                     /*Dimension of state, 2^(noOfQubits)*/
    int noOfQubits;                    /*Total number of qubits*/
    int workQubits;                    /*Number of workqubits*/

```

```

int simQubits;           /*Number of qubits used in the direct simulation*/
int firstSimQubit;       /*Here is the first non-work qubit, before this
                           they all are work, after and including we have
                           simulation qubits.*/

Array<complex<double>,1>* vec;
Array<complex<double>,1> vector;/*This is the total state of the system*/
Array<complex<double>,1> getVector(){ return vector; };
Array<complex<double>,2> m;
    /*Matrix multiplication*/
Array<complex<double>,2> mult(Array<complex<double>,2> A,
Array<complex<double>,2> B);
    /*ConSTRUCTORS!!*/
State(int noOfQ, int firstSimQ); //testing
State(const char* filename, int noOfQ,int firstSimQ); //standard
State(const char* filename, int noOfQubits); /*reads state of each qubit*/

void readFromFile(const char* filename);
void writeToFile(const char* filename, Array<complex<double>,1>
vector);
void operate(Array<complex<double>,2> G);
void performOperations(int noOfOperations);
    /*A method to calculate the inner product of a vector*/
double innerProduct(Array<complex<double>,1> A);

};

#endif

```

C.3.2 CPP file

```

#include "State.h"
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <complex>
#include <fstream.h>
#include <blitz/array.h>

```

```
using namespace blitz;
```

```
State::State(int noOfQ, int firstSimQ){
```

```
    int g;
```

```
    double m, k;
```

```
    noOfQubits = noOfQ;
```

```
    firstSimQubit = firstSimQ;
```

```
    dimension = (int)pow(2.0, (double)(noOfQ));
```

```
    operationsPerformed = 0;
```

```
    simQubits = noOfQubits - firstSimQubit + 1;
```

```
    workQubits = noOfQubits - simQubits;
```

```
    Array<complex<double>,1> temp((int)pow(2.0, (double)(simQubits)));
```

```
    /*This is to seed the random number generator, the program is run
       so fast that two State objects declared consecutively will be run
       at the same time(0) and therefore two identical states are
       generated! I solve this by adding the number of qubits to the seed*/
    srand(time(0)+noOfQubits);
```

```
    /*This is to create random starting vectors, simulating
       an unprepared state*/
```

```
    for(int i=0; i < temp.rows(); i++){
```

```
        g = rand();
```

```
        if((g%2)==0)
```

```
            g = -g;
```

```
        m = (double) g/RAND_MAX;
```

```
        g = rand();
```

```
        if((g%2)==0)
```

```
            g = -g;
```

```
        k = (double) g/RAND_MAX;
```

```
        temp(i)=complex<double>(m,k);
```

```
    }
```

```
    m=innerProduct(temp);
```

```
    for(int i=0; i < temp.rows(); i++){
```

```
        temp(i)/=sqrt(m);
```

```

    }

    cout<<"This is the starting state "<<temp<<endl;
    vec = new Array<complex<double>,1>(dimension);
    Range I(0, (int)pow(2.0, (double)(simQubits)) - 1);
    (*vec)(I) = temp;
    vector.reference(*vec);

} //conSTRUCTOR!!!

State::State(const char* filename, int noOfQ, int firstSimQ) {

    noOfQubits = noOfQ;
    firstSimQubit = firstSimQ;

    dimension = (int)pow(2.0, (double)(noOfQ));
    operationsPerformed = 0;

    simQubits = noOfQubits - firstSimQubit + 1;
    workQubits = noOfQubits - simQubits;
    cout << "Dimension of state:  " << dimension << endl;

    readFromFile(filename);

} //conSTRUCTOR!!!

void State::writeToFile(const char* filename, Array<complex<double>,1>
vector) {

    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file:  " << filename << endl;
        exit(1);
    }

    ofs << vector << endl;

} //writeToFile

```

```

void State::readFromFile(const char* filename) {

    ifstream ifs(filename);
    if (ifs.bad())
    {
        cerr << "Unable to open file:  " << filename << endl;
        exit(1);
    }

    vec = new Array<complex<double>,1>(dimension);

    Array<complex<double>,1> temp;

    ifs >> temp;
    Range I(0, (int)pow(2.0, (double)(simQubits)) - 1);
    (*vec)(I) = temp;
    vector.reference(*vec);

} //readFromFile


void State::operate(Array<complex<double>,2> G) {

    /*Just to test my methods, also I use it for the alternative phaseEst*/
    m.reference(mult(G, m));

    Array<complex<double>,1> temp(dimension);
    complex<double> tmp;

    if( G.rows() ≠ dimension){
        cout << "Error:  Matrix input does not match vector!" << endl;
    } // if

    else{
        temp = 0;
    }
}

```

```

    for(int i=0; i < dimension; i ++){
        for( int j=0; j < dimension; j ++){

            temp(i) += G(i,j)*vector(j);

        }//for
    }//for

    operationsPerformed ++;
    vector.reference(temp);
} //else

} //operate

void State::performOperations(int noOfOperations){

    operationsPerformed += noOfOperations;

} //performOperations

Array<complex<double>,2> State::mult(Array<complex<double>,2> A,
                                   Array<complex<double>,2>
B){

    int dim = A.rows()*B.rows();

    Array<complex<double>,2> temp(dimension, dimension);
    temp = 0;

    if(A.rows() != B.rows()){
        cout << "Matrices do not match, multiplication impossible."
<< endl;
        return temp;
    } //if

    firstIndex i;
    secondIndex j;

```

```
thirdIndex k;

temp = sum(A(i,k)*B(k,j), k);

return temp;

} //mult


double State::innerProduct(Array<complex<double>,1> A){

    double c=0.;

    for(int i=0; i < A.rows(); i++){

        c += real(A(i))*real(A(i));
        c += imag(A(i))*imag(A(i));

    }

    return c;

} //innerProduct
```


Bibliography

- [1] A. Aspect, P. Grangier, and G. Roger, *Experimental Realization of Einstein-Podolsky-Rosen-Bohm Gedankenexperiment: A New Violation of Bell's Inequalities*, Physical Review Letters **49** (1982), 91–94.
- [2] J. S. Bell, *On the einstein-podolsky-rosen paradox*, Physics **1** (1964), 195–200.
- [3] P. Benioff, *The computer as a physical system: A microscopic quantum mechanical hamiltonian model of of computers as represented by turing machines*, Journal of Statistical Physics **22** (1980), 563–590.
- [4] G. Cennini, G. Ritt, C. Geckeler, R. Scheunemann, and M. Weitz, *Towards quantum logic with cold atoms in a co₂-laser optical lattice*, Quantum information processing, Wiley-VCH, New York, 2003, pp. 235–245.
- [5] M. I. Dyakonov, *Quantum computing: A view from the enemy camp*, preprint cond-mat/0110326v1 (2001).
- [6] A. Einstein, B. Podolsky, and N. Rosen, *Can quantum-mechanical description of physical reality be considered complete?*, Physical Review **47** (1935), 777–780.
- [7] Elitzer and Vaidman, *Quantum-mechanical interaction-free measurements*, Foundations of Physics (1993).
- [8] R. P. Feynman, *Simulating physics with computers*, International Journal of Theoretical Physics **21** (1982), 467–488.
- [9] L. K. Grover, *Quantum mechanics helps in searching for a needle in a haystack*, Physical Review Letters **79** (1997), no. 2, 325.
- [10] A. Y. Kitaev, *Fault-tolerant quantum computation by anyons*, Annals of Physics **303** (2003), 2–30.

- [11] J. M. Leinaas and J. Myrheim, *On the theory of identical particles*, Il Nuovo Cimento **37** (1977), 1–23.
- [12] S. Lloyd, *Universal quantum simulators*, Science **273** (1996), 1073–1078.
- [13] ———, *Quantum computation with abelian anyons*, preprint quant-ph/0004010v2 (2000).
- [14] ———, *Computational Capacity of the Universe*, Physical Review Letters **88** (2002), 237901.
- [15] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics **38** (1965), no. 8.
- [16] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, Cambridge, 2000.
- [17] J. Niwa, K. Matsumoto, and H. Imai, *General purpose parallel simulator for quantum computing*, preprint quant-ph/0201042 (2002).
- [18] M. A. Pravia, Z. Chen, J. Yezek, and D. G. Cory, *Experimental Demonstration of Quantum Lattice Gas Computation*, preprint quant-ph/0303183 (2003), 3183.
- [19] P. W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, Proceedings of the 35th Annual Symposium on Fundamentals of Computer Science (Los Alamitos, CA), IEEE Press, 1994.
- [20] M. Steffen, L. Vandersypen, G. Breyta, C. Yannoni, M. Sherwood, and I. Chuang, *Experimental Realization of Shor's quantum factoring algorithm*, American Physical Society, Annual APS March Meeting, March 18 - 22, 2002 Indiana Convention Center; Indianapolis, Indiana Meeting ID: MAR02, abstract #T23.001, March 2002, p. 23001.
- [21] R. O. Vianna, W. R. M. Rabelo, and C. H. Monken, *The Semi-Quantum Computer*, preprint quant-ph/0304085 (2003), 4085.
- [22] G. Vidal, *Efficient classical simulation of slightly entangled quantum computations*, preprint quant-ph/0301063 (2003), 1063.
- [23] W. K. Wootters and W. H. Zurek, *A single quantum cannot be cloned*, Nature **299** (1982), 802–803.