

Jean François Vanthuele
BYU - Aug. 99 -

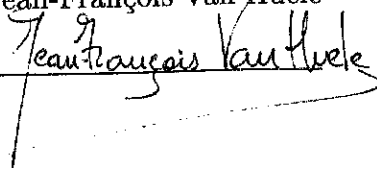
STUDY OF QUANTUM COMPUTATION
THROUGH SIMULATION OF
SHOR'S ALGORITHM
ON A CLASSICAL COMPUTER

Brent E. Kraczek

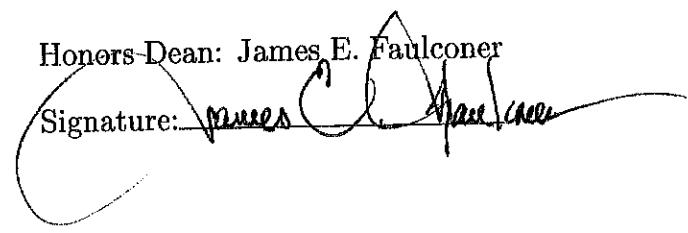
May 1999

Submitted to Brigham Young University in partial fulfillment
of graduation requirements for University Honors

Advisor: Jean-François Van Huele

Signature: 

Honors Dean: James E. Faulconer

Signature: 

Abstract

In order to understand the nature of quantum computation it is useful to examine what makes a quantum algorithm different from a classical one. This work presents an in-depth analysis of Shor's algorithm, including a simulation programmed for a classical computer.

Contents

Abstract	ii
1 Introduction	1
2 Classical Computation	3
2.1 Classical computation: Little theoretical change	3
2.2 Microchip Development: Slowing at last	4
2.3 The physical limit	6
2.4 How quantum computation may someday be an important part of the computer industry	6
2.5 Algorithm efficiency	7
3 Fundamentals of quantum computation	10
3.1 Quantum Turing machines	10
3.2 Qubits and the quantum register	11
3.2.1 Qubits	12
3.2.2 Superposition: The ability to be in two states at once .	12
3.2.3 Entanglement: Einstein's hang-up with quantum me- chanics	13
3.2.4 Quantum algebra: Using superposition and entangle- ment for exponential speed increase	14
3.2.5 Decoherence: The bane of quantum computation . . .	16
3.3 Algorithms for the new computer	16
4 The basis of Shor's algorithm	18
4.1 The number theoretic basis of Shor's algorithm	19
4.1.1 Notation and multiplication of integers modulo N . . .	19
4.1.2 Factoring on a periodic function	20

4.2	An overview of Shor's algorithm	21
4.2.1	Preparing the main register	22
4.2.2	Find the period by performing the DFT_q :	23
4.2.3	Interpretation of the result	23
5	Classical simulation of Shor's algorithm	25
5.1	Selection of main numbers: N, y, q	26
5.2	Preparation of the main register	27
5.3	Finding the spectrum: The DFT_q	28
5.3.1	The quantum theory of DFT_q	29
5.3.2	The classical implementation of DFT_q	30
5.4	Extracting and making use of the period	34
5.5	Discussion	36
6	Conclusion	37
Appendices		
A	Program simulating Shor's algorithm	38
A.1	Shor.cpp	38
A.2	shor_main.h	39
A.3	DFT_Ekert.h	43
A.4	displayREGISTER.h	48
A.5	shor_foundation.h	52
A.6	numbers.h	53
A.7	complex.h	55
B	Numerically exploring Shor's algorithm	58
B.1	trail.cpp	58
B.2	Numeric examples of how Shor's algorithm fails	61
C	Two other topics from number theory	64
C.1	The Euclidean algorithm for finding the gcd	64
C.2	Finding fractional approximations using a continued fraction	65
	Bibliography	68

Chapter 1

Introduction

People today expect their computers to become out-dated every couple of years. This trend, driven largely by the break-neck speed of microprocessor development is slowly coming to an end. This may open the market to alternative technologies. The computers of this century, based on continual improvements of the same model, may be joined by radically different technologies. Among these contenders is quantum computation, which, in theory, takes advantage of physical differences between the microscopic and macroscopic worlds to perform computations in a completely different manner than any "classical" computer.

In the last five years quantum computation has gone from an obscure mathematical model to a hot research field [Tau96]. As with any proposed technology it is hard to say how successful the field will be. Quantum computers will certainly not surpass classical ones any time soon. But certain algorithms developed for quantum computers show promise beyond what will ever be possible on a classical computer, and may drastically change the field of data manipulation and security. One such algorithm, Shor's factorization algorithm, will be the main focus of this study.

Shor's algorithm was the main cause of the increased interest in quantum computation. Quantum computation was first discussed in 1981 [Feyn82], and by 1985 the philosophy and general theory behind it were firmly in place [Deu85], claiming a potential for exponential speed increase over classical computers. But no useful algorithms had been developed to motivate sufficient research to overcome the difficulties in engineering a quantum computer. Shor changed this in 1994 by showing that factorization, for which

no efficient classical algorithm is known, would be efficient on a quantum computer [Sho94]. The difficulty of factoring large numbers is the basis of RSA cryptography—the main public key cryptography system in use that allows information to be securely transferred over the Internet [RSA98]. If a quantum computer could be built different methods of cryptography would have to be used.

In order to better understand quantum algorithms and develop a useful simulation for further study, I have designed and programmed a simulation of Shor's algorithm to run on a classical computer. Through careful examination of this algorithm and its simulation we will discuss the theoretical differences between quantum and classical computation, and show how quantum computation has made an existing algorithm efficient.

In chapter two we will discuss relevant aspects of classical computation and discuss how quantum computation may some day enter the computer industry. Chapter three discusses the theoretical basis of quantum computation, briefly describing the effects that make quantum and classical computation different. In chapter four we will examine the number theoretic basis of Shor's algorithm and overview the algorithm through a simple example. In chapter five we will investigate the specifics of Shor's algorithm and show how they were implemented in the simulation. The appendices include all the code in the working version of my simulation, a section on numeric exploration of the number theory behind the algorithm and a couple basic number theory topics for reference.

The diskette included with the thesis contains the source code for `Shor.cpp` and `trial.cpp`, together with the appropriate header files, together with a compiled version of `Shor.cpp`, `Shor.out`, for use on an HP PA RISC 2.0 workstation. The diskette also contains the \LaTeX 2_ε file for this thesis.

Chapter 2

Some elements of classical computation

Every existing computer today is “classical”. Even though the microprocessor is composed of millions of field effect transistors, the operation of which depends on quantum mechanical phenomena, the mathematical basis of the computation is classical. The primary difference between quantum and classical computation is in the nature of the bits the data are stored in and how they are manipulated. Since it will shortly be physically impossible for microprocessor components to continue shrinking at the rate they have, new technologies are posed to find their way into the computational world. Among these, quantum computation is a strong contender. Although we will delay an actual definition of classical computation until chapter three, this chapter will discuss aspects of classical computation useful in gaining an understanding of quantum computation.

2.1 Classical computation: Little theoretical change

Microprocessor performance has advanced at an amazing rate in the second half of this century. Computing power has increased exponentially as microelectronics have shrunk in size. While it is common to speak of “technical progress”, the mathematical basis and methods of technological advancement have changed little and has been accompanied by extreme increases in production cost because of the difficulties in fabricating smaller and smaller

devices. In addition, physical limitations may cause the microprocessor industry to change significantly in the near future, making room for new technologies as well.

Whether a computing device is a PC or a supercomputer, it can be mathematically described as a Turing machine [Tur37]. All classical computers, no matter how complex, can be described by Turing's model. According to Williams and Clearwater, Turing's idea came from mathematician David Hilbert's question, "Could a machine be built that would be able to prove the truth of any mathematical conjecture?" Taking this question more literally than Hilbert probably intended, Turing designed a machine to mimic the way a mathematician thinks when writing a proof [WC97].

Turing decided that his model should not be a blueprint for a specific machine, but general enough to describe all possible systems and avoid all physical assumptions. His machine had an infinite strip of paper with cells on it and a read-write head. The cells are either marked or unmarked (in current terminology marked with a one or zero). The read-write head, like a mathematician, would use earlier premises to dictate later ones, moving up and down the strip, its actions dictated by the patterns of symbols it reads.

The Turing machine is simple enough to be used in mathematical proofs, but complex enough to represent every computer in use. Turing claimed (and periodically demonstrated) that no matter what method someone used to create a different model, it was mathematically equivalent to his. According to Williams and Clearwater, "Given enough time and memory, there is not a single computation that a supercomputer can perform that a personal computer cannot also perform. In the strict theoretical sense, they are equivalent" [WC97].

2.2 Microchip Development: Slowing at last

From the invention of the microchip it has ruled computation, developing into a \$150 billion industry. The number of components per chip has doubled every couple year since the chip's invention, driving the industry forward. Analysts now say that industry growth may slow in the near future. In the next twenty years either economics or the laws of physics will force the business to change.

In 1964 Gordon Moore, co-founder of Intel, saw that the number of components on a microchip doubled every year. The prediction that this would continue became known as Moore's law. Since 1964 Moore's law has held with occasional alteration. According to Alfred Brenner, Deputy Director of the Institute for Defense Analysis, the rate had slowed to a doubling every 18 months by the end of the seventies, which slowed again at the end of the eighties to every 24 months. Market analysis now shows that from 1997-2007 the number will double every three years [Bre97]. Beyond 2007 some scientists and analysts are now saying that this trend is even less sure.

According to Dan Hutcheson, analyst for VSLI research group, and his partner Jerry Hutcheson who serves on the Semiconductor Industry Association Roadmap Council, the progression of technology was not a smooth development. "It was more like a harrowing obstacle course that repeatedly required chip makers to overcome significant limitations in their equipment and production processes." Anytime one group of engineers predicted a "show stopping" problem, that was too costly to overcome, someone else would solve it [HH96].

This trend may be coming to its end. Robert F. Service, research writer for Science magazine reports that the industry is looking hard for new technologies to keep things going. The current methods of chip production may be able to fit 250 times as many components on a single chip as are there are currently, but then the technology will need significant advances [Ser96]. Hutcheson and Hutcheson agree, saying that the industry is in need of breakthroughs to continue, both in lithography (the method of manufacture) and in the materials used [HH96]. Some scientists don't believe there will be a crunch. Bijan Divari, silicon integration engineer for IBM reminds us that industry has continued in this fashion for thirty years, and with the developments taking place now there are many possibilities on the horizon [Ser96]. Robert Keyes, also of IBM, acknowledges the limits, but says that there are many more possible new technologies on the horizon that will develop as understanding of the physics of new materials increases [Key92].

Technology, however, isn't the only issue; the microprocessor industry anticipates having trouble increasing investment and may have to change their way of doing business. According to Hutcheson and Hutcheson, while the cost of 1 megabyte memory chip has gone from \$550,000 to \$38 in the last 25 years, the price of building the production facility has gone from \$4 million to \$1.2 billion. Business methods will have to change as prices

increase and the rate of development slows. The computer industry will change just like the automotive, railroad and airline industries did, going from advancement in straight power to specialized markets. These industries also grew quickly in the beginning, but eventually the need for bigger and faster machines didn't match the costs, and companies began to diversify their products [HH96].

2.3 The physical limit

Although the industry is more worried about the barriers in the near future, some scientists are looking at ultimate physical limits. Clearwater and Williams say that if the current trends hold chip components will reach the size of an atom by 2020. Not only would this be the decisive end to any decrease in size, sometime before the components reached that size they would enter the realm of quantum mechanics [WC97, Mil96].

Another physical problem with shrinking components is that the amount of heat given off by the integrated circuits increases with the density of the components. Irreversible computation, which uses logic gates to get one bit of information from two, emit heat with the destruction of information. The laws of thermodynamics require that for each bit erased the computer gives off $k_B T$ in heat [Lan91]. A microchip based on irreversible logic that has components the size of an atom would produce more than enough heat to melt itself. Although we will not discuss this in greater detail, since the evolution of a quantum system is reversible, a quantum computer would be a reversible computer and not suffer from the heat problem. (For further discussion on reversible and irreversible computation, see [Mil98] and [FT82].)

2.4 How quantum computation may someday be an important part of the computer industry

The three most promising alternatives to reduction in size of microcircuitry are nanocircuitry, optical computation and quantum computation. Although our topic is quantum computation, the other two technologies help illustrate how quantum computation may enter the industry. These technologies also

offer different ways of approaching problems that would allow more efficient algorithms for specific problems.

Strictly speaking nanocircuitry would be the effect of continual size reduction of microcircuitry. As the size of microprocessor elements is reduced, increased attention must be placed on the quantum mechanics. Nanocircuits will be small enough that either they will utilize quantum mechanical effects or be limited by them. In experimental systems careful design has allowed tight control of current flow and behavior of groups of particles. Gerard Milburn, a leading physicist in the field of quantum optics, predicts that, within a decade, quantum nanocircuits will have important niche markets [Mil96].

Optical computing is the oldest of the three technologies, and may show how quantum computing will enter the market. Research in optical computing began before 1960, but optical computers have not been able to compete with semi-conductor based systems. A group of special editors chosen for an *Applied Optics* feature issue on optical computation, say that optical computation is now considered a strong contender. The Institute of Electrical and Electronics Engineers (IEEE) has held conferences on optical computing since the '70's. In the '90's there have been demonstrations of "practical optical computing systems". Although optical computers are still a technical novelty, many technologies developed for optical computing, including fiber optics, are used commercially in some form of computer hardware [Li96].

As shown by developments in optical computing, the industry is prepared to accept new technologies where their performance is superior. As microchip advances slow we are likely to see more of these technologies entering electronic devices. Quantum computation, in particular, has the attractive advantage that its calculations can be performed with an exponential speed increase.

2.5 Algorithm efficiency

One method for increasing computer performance is finding more efficient algorithms. A clever new algorithm can greatly decrease the time required to perform desired task. For example the discrete Fourier transform is commonly used in data analysis to find the spectrum of experimental data. Noticing that many of the calculations in the transform are redundant, a new algorithm improved performance from $O(n^2)$ to $O(n \ln n)$. This second

form is known as the fast Fourier transform (FFT), and can greatly reduce time required for data analysis. Where the FFT takes 1 second to do the spectral analysis on 100,000 data points the older or slow Fourier transform would take two and a half hours to produce the same results on the same computer.

In computer science the most common method of rating algorithm performance is through "big-oh" notation, written $O(f(n))$. Big-oh notation will not give specific information about run time (which is dependent on implementation, compiler and computer the program is run on), but offers information on how the time taken for operation scales with the problem size. If we have a problem, where the scaling variable is n (we may have n entries in a database) we can analyze the algorithm in relation to n . Using an example in C code,

```
for (i=0; i<=n; i++){
  for (j=0; j<=n; j++){

    /* program performs some operations here, where *
     * the number of operations is independent of n */

  }
}
```

we see that we could have many operations occur. If in place of the “some operations” we place 25 different commands, the run time of the example code, $T(n)$, will scale as $T(n) = 25n^2$. But the time each of the operations takes may vary from computer to computer. We may even decide that some of the commands are unnecessary, and reduce them to 21, by adding another loop that scales linearly with n , so $T(n) = 21n^2 + 5n$. Overall, though, either way the dominant portion of the code scales with n^2 , so we say it is $O(n^2)$. (For a more formal discussion of runtime and big-oh notation see chapter 3 of [AU95].)

Improved algorithms can greatly increase computer performance, but for many problems no efficient algorithms are known to exist. In general, we say an algorithm with $O(f(n))$ is considered efficient (or “tractable”) if $f(n)$ grows like a polynomial, and inefficient (or “intractable”) if $f(n)$ grows faster than a polynomial.

Chapter 3

Fundamentals of quantum computation

In this section we will examine some basic quantum mechanical principles necessary to understanding the nature of quantum computing, including what makes a quantum computer different from a classical one and an overview of the quantum effects quantum computational theory relies on: entanglement, superposition and decoherence. Our scope will be limited to the potential of quantum Turing machines and what may give them an edge over classical Turing machines.

3.1 Quantum Turing machines

In 1981 Richard Feynman questioned Turing's claim that all systems could be described by his model, asking whether a Turing machine would be able to simulate quantum mechanics. In answer to the questions raised by Feynman, David Deutsch, redefined the Turing machine, allowing him to make a distinction between classical and quantum Turing machines.

In 1981, at the first Symposium on Computers in Physics at MIT, Richard Feynman raised a question about whether it would be possible to build a "universal quantum simulator". Feynman stated that a Turing machine may not be able to serve this purpose since there are a lot of things we simply don't know about quantum mechanics, and classical computers may not be able to simulate them for us. He asked if it would be possible to build a computer such that, with "a suitable class of machines you could

simulate any quantum system.” He said that this machine would not be a Turing machine, but like a classical or universal computer, this simulator should be able to handle any quantum system [Feyn82].

In response to Feynman’s question, Deutsch looked at the implications of a quantum computer [Deu85]. Deutsch decided that regardless of Turing’s attempts to define his machine independent of any sort of design of a computer, his definition was too general. The machine has to assume some physical assumptions, and needs to be matched to the “existing structure of physics”. According to Deutsch,

The reason we find it possible to construct, say, electronic calculators, and indeed, why we can perform mental arithmetic, cannot be found in mathematics or logic. The reason is that the laws of physics happen to permit the existence of physical models for the operations of arithmetic such as addition, subtraction and multiplication. If they did not, these familiar operations would be non-computable functions. We might still know of them in mathematical proofs . . . but we could not perform them.

Deutsch continues, “Every existing model of computation is effectively classical. . . . The more urgent motivation [to develop a model of quantum computation] is, of course, that classical physics is false.”

Whether or not one agrees that classical physics is false, it is the difference between quantum and classical physics that makes a quantum computer different. The mathematics of quantum mechanics does allow us to simulate things in the classical world (according to the correspondence principle), but it also allows us to treat things differently. Deutsch’s model for a quantum Turing machine provides the fundamental mathematical basis for quantum computation, just as Turing’s original model does for classical computing.

3.2 Qubits and the quantum register

The major effects that must be addressed in quantum computation are how entanglement, superposition and decoherence act on quantum bits or qubits. The combination of the first two working together is what allow the exponential speed increase promised by specific algorithms. They allow the qubits to

work together in ways not possible with classical bits. The third effect, decoherence, destroys the effects of the other two, making physical realization of quantum computation difficult.

3.2.1 Qubits

Just as the basis of data in classical computers is the bit or binary digit, the basis of data in a quantum computer is the qubit or quantum binary digit. A working quantum computer would be composed of a memory register of many such qubits, each bit being stored by a quantum system (possibly a two-level atom in an ion trap, see [MW95, MW96, Ste96, HR96], or as multiple spins in an NMR system, see [GC97, Eco97]). We can write each bit in familiar spin form representing spin-up and spin-down

$$|\uparrow\rangle \text{ vs. } |\downarrow\rangle$$

or in a binary form representing on and off

$$|1\rangle \text{ vs. } |0\rangle.$$

Multiple qubits can be placed together in a quantum register, similar to how a classical register is put together. We can put a series of bits together, representing them either as arrows, binary, or in the decimal equivalent of the binary number:

$$|\uparrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\uparrow\uparrow\rangle = |101101011\rangle = |363\rangle.$$

For our purposes we will generally use the decimal notation for representing quantum registers.

3.2.2 Superposition: The ability to be in two states at once

A quantum state defines the “range of possible locations” of a particle in discrete, quantized energy states [Ste96]. It is generally believed that the particle is not just somewhere, but everywhere in that range of locations, in a superposition of states, similar to how a superposition of solutions to a differential equation is also a solution.

According to quantum mechanics, the particle's state will never be measured in a combination of states, but as one or the other. Gilles Brassard explains how this probability works by giving the example of an electron orbiting an atom. The electron can be moved from one state to a higher state by shining a specific frequency of light on it for a specific length of time. But if you only shine the light on the atom for half the needed time and then take a measurement, there's a fifty percent chance of finding the electron in one state or the other. We can use a system like this for a qubit, with one possible state representing a 0 while the other represents a 1. This qubit is different than a classical bit because it represents neither 0 nor 1, but a combination of the two [Bra97].

If we consider a particle in a superposition of two states, $|\uparrow\rangle$ and $|\downarrow\rangle$, we can write the state of the particle as

$$\psi = \alpha |\uparrow\rangle + \beta |\downarrow\rangle \quad (3.1)$$

where α and β are the probability amplitudes of the respective states. Measurement of the bit will produce only one of the states, with a probability of $|\alpha|^2$ finding $|\uparrow\rangle$ and a probability of $|\beta|^2$ of finding $|\downarrow\rangle$. If we are measuring the spin state of a particle we are guaranteed to have either spin-up or spin-down, so the total probability $|\alpha|^2 + |\beta|^2$ is one [Gas96].

3.2.3 Entanglement: Einstein's hang-up with quantum mechanics

The other quantum phenomenon that gives quantum computing its power is entanglement. Entangled quantum particles form a somehow correlated system in which their quantum properties are connected until measurement. A common example of an entangled state of two fermions is the spin-singlet state,

$$\psi = \frac{|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle}{\sqrt{2}}, \quad (3.2)$$

which is a superposition of two two-qubit states. If we write this in the form

$$\psi = \alpha |\uparrow\downarrow\rangle + \beta |\downarrow\uparrow\rangle, \quad (3.3)$$

then $\alpha = 1/\sqrt{2}$ and $\beta = -1/\sqrt{2}$. When we measure one of the bits we have a 50-50 probability of measuring either state, spin-up or spin-down. Saying

that the particles are entangled means that if we measure the first bit and get an answer of spin-up we are guaranteed to get an answer of spin-down when we measure the second (where we still had an even chance on finding the second particle in either state before we measured the first). This holds true even if the particles are separated by a long distance and the second is measured so soon after the first that any communication between the two would have to exceed the speed of light.

This phenomenon, known as the Einstein-Podolsky-Rosen (EPR) Paradox has no classical analog. Somehow particles know the states of the other particles they are entangled to. Einstein, Podolsky and Rosen published this paradox as an argument against quantum mechanics, saying that the theory can not be complete as it implies communication faster than the speed of light. The phenomenon predicted by EPR was demonstrated in a series of experiments in 1981-2 led by Alain Aspect (see [AGR81, AGR82, ADR82], or for a more accessible, comprehensive explanation, chapter 6 of [GZ97]).

A quantum register would be composed of many correlated particles, each of which can be acted on individually or collectively. Any operation on a single qubit would cause each state the register is in to change in that bit. The register will remain in this state until a measurement is made. If the qubits are all entangled, measurement of the state

$$\psi = \frac{1}{\sqrt{2}} |\uparrow\downarrow\uparrow\uparrow\downarrow\uparrow\downarrow\uparrow\uparrow\rangle + \frac{1}{\sqrt{2}} |\downarrow\uparrow\downarrow\downarrow\uparrow\downarrow\uparrow\downarrow\downarrow\rangle, \quad (3.4)$$

will be completely determined by the first measurement that is made.

3.2.4 Quantum algebra: Using superposition and entanglement for exponential speed increase

The promise of exponential speed increase comes from the interplay between superposition and entanglement. Since a register of n bits can be in a superposition of any or all of the possible values, it can effectively act as 2^n registers simultaneously, allowing for 2^n calculations to be performed at once. There is a hitch, however, in that when the register is measured only one value is retrieved.

If we begin with two registers whose states are represented by ψ and ϕ ,

$$\psi = \alpha_\psi |1\rangle + \beta_\psi |2\rangle + \gamma_\psi |4\rangle,$$

$$\phi = \alpha_\phi |0\rangle + \beta_\phi |1\rangle + \gamma_\phi |5\rangle, \quad (3.5)$$

$$(3.6)$$

and perform an addition of ϕ to ψ , the state held by register ψ is now

$$\tilde{\psi} = \alpha_{\tilde{\psi}} |1\rangle + \beta_{\tilde{\psi}} |2\rangle + \gamma_{\tilde{\psi}} |3\rangle + \delta_{\tilde{\psi}} |4\rangle + \varepsilon_{\tilde{\psi}} |5\rangle + \zeta_{\tilde{\psi}} |6\rangle + \eta_{\tilde{\psi}} |7\rangle + \theta_{\tilde{\psi}} |9\rangle, \quad (3.7)$$

where

$$|\alpha_{\tilde{\psi}}|^2 + |\beta_{\tilde{\psi}}|^2 + \dots + |\theta_{\tilde{\psi}}|^2 = 1. \quad (3.8)$$

Here the exact number of calculations done depends on the number of states ψ and ϕ are each originally in. A problem arises when we go find out what each of those values are—when we measure $\tilde{\psi}$ we get only one of the answers and we can never know what the other states were.

This poses a sticky problem for quantum computation. In an n qubit register we can at best perform 2^n calculations, but unless we can find an efficient way to use each of these we effectively only perform one calculation, and our answer is somewhat random. If we begin with a 3-qubit register in a superposition of all possible states

$ 000\rangle$	$ 100\rangle$
$ 001\rangle$	$ 101\rangle$
$ 010\rangle$	$ 110\rangle$
$ 011\rangle$	$ 111\rangle$

perform an addition of several states on it we could calculate eight answers at once, but there are only eight possible states. If all original states have the same probability, our answer is just a random, 3-qubit number.

To get around this problem quantum algorithms are generally based on a quantum version of the fast Fourier transform (FFT), which deals with the probability amplitudes of each of the possible states of the register in a similar manner to how the classical FFT operates on a list of data. This does, however, require that algorithms be found that rely on periodicity. In such algorithms the quantum speed increase is at best exponential, and can change exponential algorithms to polynomial.

3.2.5 Decoherence: The bane of quantum computation

The last of the three major effects involved in quantum computation, decoherence poses the biggest hurdle to a working computer. Any interaction between the environment and a qubit effectively measures that qubit, determining (at least partially) the state of the other qubits and causing the qubit to decohere or disentangle. If a qubit in the quantum register experiences any interaction with the outside environment that qubit will decohere and the quantum algorithms will not work. Current experimental quantum systems have been unable to keep sufficient numbers of entangled qubits isolated from the environment to perform significant quantum computations.

One proposed solution to this problem is through quantum error code correction. In classical computation error correction codes are used to guarantee against accidental flips in the value of a specific bit. Similar methods have been developed for quantum computation (for further information see [Sho96], [EJ96]).

Quantum computing's main strength lies in its theoretical ability to use massively entangled systems. Even if the time required to perform single operations were close to that of classical computers (which is not yet the case) the quantum computer would have to decisively beat classical computers to gain serious consideration. Search algorithms like Grover's aren't very impressive when a much cheaper, classical system takes less than a second. Large problem sizes are needed to justify using a quantum computer. Regardless of the quantum system chosen, a quantum computer deciphering a 56-bit encryption code would require at least 56 entangled states (plus many more for error correction codes). Haroche and Raimond have claimed that more than a few dozen entangled states will never be realized, which is a much smaller number than is needed to justify quantum computing for Shor's algorithm [HR96]. If large entangled systems cannot be created, quantum computing may never be realized.

3.3 Algorithms for the new computer

Deutsch's model for computing laid the groundwork for quantum computing, but, like Turing's model, it didn't build a computer. His model lay largely unexamined until the discovery of quantum algorithms showed how powerful quantum computing can be. Quantum algorithms can perform tasks at

speeds that are mathematically impossible for a classical Turing machine, including (but essentially limited to) factoring large numbers and searching unsorted lists. There is still a clear lack of useful algorithms.

As already mentioned, the field remained in obscurity until 1994 when Peter Shor of AT&T Bell Labs announced that he had found a way to factor large numbers in polynomial time. There is no known way to do this efficiently on a classical computer. This has made factoring an important tool in data encryption. This demonstration of the power of quantum computing, albeit theoretical, caused interest in quantum computing to skyrocket, with many people searching for other useful algorithms.

The next important algorithm, Grover's algorithm, provides a dramatic example of the potential power of quantum computers. Grover's algorithm is for a "needle-in-a-haystack" type search. Brassard explained that the best method of searching on a classical computer is a sequential search, which is $O(n)$, while Grover's algorithm can perform the search in $O(\sqrt{n})$ [Bra97]. Put in more dramatic terms by Graham Collins, associate editor of *Physics Today*, for a search through $n = 2^{56}$ possibilities (the common Data Encryption Standard uses a 56-bit key), a classical computer trying one million different possibilities a second would take, on average, 1000 years to find a match. A quantum computer running Grover's algorithm at the same speed would take under 4 minutes [Col97].

Although there are a couple other algorithms for quantum computation (for a review of algorithms, see [Joz98]), the lack of useful algorithms is still a concern. The importance of Shor's algorithm—the threat of being able to break the RSA codes—may generate enough interest to build a prototype quantum computer, but if other useful algorithms are not found, the world may only ever need one quantum computer.

Chapter 4

The basis of Shor's algorithm

The genius of Shor's algorithm is using quantum computation to make a known method of factoring efficient. As was discussed in the last chapter a quantum algorithm must find a way to get more than one answer out for the exponential speed increase to be useful. Shor's algorithm uses the quantum discrete Fourier transform modulo q (DFT_q) to analyze the information in the register before measurement, so that almost all possible answers provide enough information to complete the calculation. Since all the major known quantum algorithms depend on the DFT_q [Joz98], analysis of Shor's algorithm serves as a good illustration of how successful quantum algorithms work.

The most widely used standard for asymmetric data encryption, RSA cryptography, relies on the fact that large numbers are hard to factor by using the product of a pair of large primes [RSA 98]. Even for smaller numbers calculating the product,

$$229 \cdot 283 = z$$

can be done much faster than solving

$$x \cdot y = 64,807.$$

for x and y .

The best known factoring algorithms for classical computers are considered "sub-exponential", and intractable. The number field sieve, which best handles large numbers, is $O\left(e^{cn^{1/3}(\ln n)^{2/3}}\right)$, where c is a constant and n is the number of bits needed to represent the number being factored [WC97]. Any

time there is worry that the number of bits used in RSA security is too small the standard can be increased. Although factoring large numbers is generally hard, the Rabin algorithm [Rab80] can efficiently determine whether a number is prime or composite, providing an inexhaustable source of primes. Shor's algorithm may be the only way to break this.

4.1 The number theoretic basis of Shor's algorithm

Factoring is a problem from classical number theory, and Shor's algorithm is primarily rooted in it. Most of the algebra takes place in the the multiplicative group of integers modulo N , where N is the number being factored.

4.1.1 Notation and multiplication of integers modulo N

Since a majority of the math in Shor's algorithm deals with the set of integers modulo N , and N is always the number to be factored, numbers in the equivalence class $x(\text{mod}N)$ will generally be written as $\bar{x} \equiv x(\text{mod}N)$. This serves to distinguish these numbers from regular integers, which also play a part in the algorithm. Hence, if $N = 5$, numbers will be written $\bar{3} = \bar{8} = \bar{13} = \bar{18}$. On the other hand if $\bar{x} = \bar{5}$ we cannot be sure that $x = 5$.

Multiplication in the integers mod N is surprisingly simple. This greatly aids in any realization of the algorithm since raising an integer to modest powers quickly requires a tremendous amount of memory. It is also useful in understanding the algorithm. If $X = a + bN$ and $Y = c + dN$, then

$$\begin{aligned} \overline{X \cdot Y} &= \overline{ac + N(ad + bc + bdN)}, \\ \overline{X \cdot Y} &= \overline{ac}, \\ \overline{X \cdot Y} &= \overline{(a + bN) \cdot (c + dN)}. \end{aligned}$$

Hence,

$$\overline{X \cdot Y} = \overline{\bar{X} \cdot \bar{Y}}. \quad (4.1)$$

Thus any register involved in the program needs only to be large enough to handle numbers on the same order N .

4.1.2 Factoring on a periodic function

Given an N to factor, and a random integer y , where $1 < y < N$ and $\gcd(y, N) = 1$, then $\gcd(y^a, N) = 1$ and $\overline{y^a} \neq \overline{0}$ for any integer a . Now we let

$$g(a) \equiv \overline{y^a}, \quad a = 0, 1, 2, 3, \dots \quad (4.2)$$

Since there are only N equivalence classes in the integers mod N , there are only N possible distinct values of g . Furthermore, since $\overline{y^a} \neq \overline{0}$, there are at most $N - 1$ elements in g and one of them is not $\overline{0}$. Using Eq. (4.1), for any two integers b and c , if $g(b) = g(c)$ then $g(b+1) = g(c+1)$. Since there are only a distinct number of values of g , for some r , $g(b+r) = g(b)$. Thus g is periodic with period r and each element of g occurs only once in a period. (In algebraic terms, g is a cyclic sub-group of the multiplicative group of integers modulo N with generator y and order r [DF91].)

This periodicity can be used to find the factors of N . The specific values of g are $g(0) = \overline{1}$, $g(1) = \overline{y}$, . . . $g(r) = \overline{1}$, . . . , we see that $\overline{y^r} = \overline{1}$. This implies that

$$\begin{aligned} \overline{y^r} &= \overline{1}, \\ \overline{y^r - 1} &= \overline{0}, \\ \overline{\left(y^{\frac{r}{2}}\right)^2 - 1} &= \overline{0}, \end{aligned}$$

which, by factoring, gives

$$\overline{\left(y^{\frac{r}{2}} + 1\right) \left(y^{\frac{r}{2}} - 1\right)} = \overline{0}. \quad (4.3)$$

This implies that $\left(y^{\frac{r}{2}} + 1\right) = mp$ and $\left(y^{\frac{r}{2}} - 1\right) = nq$, where m , n , p and q are integers and $pq = N$.

The two factors of N , p and q can be found by taking the greatest common denominator of N with each of the factors in the left hand side of Eq. (4.3):

$$\begin{aligned} \gcd\left(y^{\frac{r}{2}} + 1, N\right) &= \gcd\left(\overline{y^{\frac{r}{2}} + 1}, N\right) = p, \\ \gcd\left(y^{\frac{r}{2}} - 1, N\right) &= \gcd\left(\overline{y^{\frac{r}{2}} - 1}, N\right) = q. \end{aligned} \quad (4.4)$$

This gives us

$$\gcd\left(\overline{y^{\frac{r}{2}} + 1}, N\right) \times \gcd\left(\overline{y^{\frac{r}{2}} - 1}, N\right) = N. \quad (4.5)$$

If $\overline{y^{r/2} + 1} = \bar{0}$, then $p = N$ (or if $\overline{y^{r/2} + 1} = \bar{0}$ then $q = N$) and Shor's algorithm yields the trivial factors. Otherwise p and q are non-trivial factors. Shor claims that the probability of finding trivial factors is $1 - 1/2^k$ where k is the number of distinct prime factors of N . So if we can efficiently calculate enough values of $\overline{y^a}$ to find r we can efficiently factor N [Sho94].

What values of r are possible? It is clear that $1 < r < N$, but there is little guaranteed beyond that. To serve as an example, a simple program can find the possible values of r for each possible y . Using $N = 77$ as an example, the different r values generated are 2, 3, 5, 6, 10, 15, 30 and 76. It is easy to see how the even r 's would not produce irrational numbers, but the occasional odd r 's raise some questions. These often occur with perfect squares (when $y = 36$, $r = 5$), but not always. When $y = 23$, $r = 3$ and Shor's algorithm produces factors of 7 and 1. Further analysis of this problem can be found in Appendix B.

4.2 An overview of Shor's algorithm

We now turn to Shor's algorithm, using it on a simple example—factoring the number 21. Our purpose here is to give a brief overview, unmuddled by details. In the following chapter we will examine the details of each step of Shor's algorithm, and discuss their classical simulation. The example used here, the factoring of 21, is not interesting by itself, but the method would be used for much larger numbers in a quantum computer.

A majority of Shor's algorithm relies on classical number theory, and can be performed on a classical computer. The heart of the algorithm, though, is the Quantum Discrete Fourier Transform ($\text{DF}T_q$), which finds the spectrum of a single quantum register in a superposition of values much the same way

the fast Fourier transform finds the spectrum of a list of values (stored in an array of registers). The other parts of the algorithm serve to either prepare the register for the DFT_q or interpret the result received from the DFT_q .

4.2.1 Preparing the main register

We begin with a number to factor, N and a random number y that is relatively prime to N :

$$N = 21, \quad y = 13.$$

In practice it isn't essential to choose a number relatively prime to N . Any random odd number will do. Most numbers will be relatively prime, but if the random number chosen is a factor then we have our answer.

We now find $q = 2^L$ such that

$$\frac{q}{2} < N < q. \quad (4.6)$$

$$q = 512.$$

We then prepare a quantum register with $3L/2$ entangled qubits. We prepare the first L qubits so that each of the q states formed by these qubits have equal probability amplitude.

$$\sum_{a=0}^{q-1} f(a) |a\rangle, \quad (4.7)$$

where $f(a) = \frac{1}{\sqrt{q}}$.

We prepare the other $L/2$ qubits so that this auxiliary register contains the function $g(a) = \overline{y^a}$. The state of our register is now

$$\sum_{a=0}^{q-1} f(a) |a\rangle |\overline{y^a}\rangle. \quad (4.8)$$

where $f(a)$ is the probability amplitude of the state $|a\rangle$.

As shown above g is periodic. For $N = 21$ and $y = 13$ we have

$$|y^a\rangle = \begin{matrix} |1\rangle \\ |13\rangle \\ |1\rangle \\ |13\rangle \\ |1\rangle \\ \vdots \end{matrix} \quad (4.9)$$

We now measure the auxiliary register and throw out the answer. In our example we will assume that our measurement yields the answer "1".

We do not keep the answer because we are only interested in the period of g . Since this auxiliary register was entangled to the main register the probabilities of the states in the main register now reflect that period. For all a such that $g(a) = 13$, $f(a) = 0$.

$$f(a) = \begin{cases} \frac{1}{\sqrt{256}} & , a = 0, 2, 4, \dots, 510 \\ 0 & , a = 1, 3, 5, \dots, 511 \end{cases} \quad (4.10)$$

4.2.2 Find the period by performing the DFT_q :

We now use the DFT_q to find the spectrum of the probability amplitudes of the different possible values in the main register.

$$\text{DFT}_q : \sum_{a=0}^{q-1} f(a) |a\rangle \longrightarrow \sum_{c=0}^{q-1} \tilde{f}(c) |c\rangle \quad (4.11)$$

$$\tilde{f}(c) = \begin{cases} \frac{1}{\sqrt{2}} & , c = 0, 256 \\ 0 & , \text{otherwise} \end{cases} \quad (4.12)$$

4.2.3 Interpretation of the result

We now measure $|c\rangle$ and call this value c . We use a continued fraction to find a fraction in lowest terms that approximates c/q ,

$$\frac{c'}{r} \approx \frac{c}{q}. \quad (4.13)$$

The continued fraction is discussed in greater detail in chapter 5 and in appendix C. We can find r if $c = 0$. There is a 50% probability of measuring each, $c = 0$ or $c = 256$, so we have a 50% chance of getting r . If we measure $c = 256$ then we get $r = 2$ and can use this in Eqs. (4.4) to find the factors of N

$$\overline{y^{\frac{r}{2}}} = \overline{13^{2/2}} = 13.$$

$$\gcd(13 \pm 1, 21) = 7, 3. \quad (4.14)$$

Once again these last steps—the continued fraction and finding the greatest common denominator—can be done using well-known methods from classical number theory. The power of Shor's algorithm is in the efficiency of the DFT_q .

Chapter 5

Classical simulation of Shor's algorithm

Careful examination of Shor's algorithm provides a good model for how a quantum algorithm can be applied to a classical problem to find a tractable solution. It also illustrates the inherent differences between classical and quantum computation. Emphasis will be placed on these differences, showing how the algorithm has been realized in our classical simulation. The entire code of the simulation can be found in Appendix A. To aid in understanding the program each step will be examined thoroughly and frequent reference will be made to the functions in which the specific portion of the implementation can be found. (The function names will be written `function()` in `headerfile.h`.) The flow of the simulation is controlled by the function `run_shor()` found in `shor_main.h`.

For large numbers (such as are used in RSA cryptography) most steps in the algorithm are tractable on a classical computer, and would presumably be executed on one, except for calculating enough values of

$$g(a) = \overline{y^a}$$

to generate a period and then finding that period. Quantum computation solves this problem because quantum gates operate bitwise on the qubits of superposed states rather than whole bytes. A series of gate operations can be used to generate g efficiently, and a second, the quantum discrete Fourier transform (DFT_q) can efficiently find r . This simulation sets out to perform Shor's algorithm on a classical computer as a quantum computer

would. Because of the differences of operation between classical and quantum computers some of the code is inefficient by general programming standards.

Although it will be a common complaint that our classical computer cannot perform quantum operations, there are two great advantages of our simulation over what an actual quantum computer would do: when we want to see what is in our register we don't end the program and when we take a measurement we can see what all the values in our register are at once. With a quantum computer only one value can ever be extracted. For this reason if/when a quantum computer is built classical simulation may still play an important role in algorithm design.

5.1 Selection of main numbers: N , y , q

The first task in performing Shor's algorithm, is to determine the numbers to be used throughout the algorithm. Given N , an integer to factor, we choose a random integer, y , such that $1 < y < N$. For Shor's algorithm to definitely work we need $\gcd(y, N) = 1$. So we use the Euclidean algorithm to check to see if the numbers are co-prime. In our program, any time we need to find a greatest common denominator we call `GCD()` from `numbers.h`. An explanation of the Euclidean algorithm can be found in Appendix C.

In practice it would be nice if N and y were not co-prime, since the Euclidean algorithm would find a factor of N and we would be done. However for a 200-digit number with two prime factors (as used in RSA cryptography) this isn't very likely. If it does happen (and since the numbers our program factors are 2-digit this occurs frequently) the program simply returns the two factors and is done.

If N and y are co-prime the next step is to determine the number of entangled qubits needed to factor N . This can also easily be done on a classical computer (and is performed in `set_mainNums()` in `shor_main.h`). For the DFT_q to be efficient we need to find integers q and L such that

$$\begin{aligned} q &= 2^L, \\ \frac{q}{2} &< N^2 < q. \end{aligned} \tag{5.1}$$

The DFT_q needs to operate on a register capable of containing the states $a = 0, 1, 2, \dots, q-1$ to be efficient, so we need L qubits to hold 2^L states. An

additional set of qubits, also entangled with the first L , are needed to create the "auxiliary register" in which the function

$$g(a) = \overline{y^a} \quad (5.2)$$

is placed to create a period. Since for all a , $g(a) < N$, the number of bits required to hold g is $\lceil \log_2 N \rceil = L/2$, so in all $3L/2$ qubits are required for our quantum register. Additional bits are required for error code correction, which we will do not consider in our simulation.

In a working quantum computer the maximum number of qubits a computer can handle will determine whether or not the computer is able to factor a number. In our simulation, too, we will see this happen, although in a different way.

5.2 Preparation of the main register

The main register is prepared by superposing in it all possible values $a = 0, 1, 2, \dots, q - 1$ so each state has an equal probability amplitude. The subregister is then generated by the main register with the value $\overline{y^a}$ entangled to each corresponding value a , $|a\rangle |\overline{y^a}\rangle$. This can also be done efficiently on a quantum computer, first by determining the value of $\overline{y^{2^i}}$ for each $i = 0, 1, 2, \dots, L - 1$ through repeated squaring modulo N and then multiplying specific $\overline{y^{2^i}}$'s according to the binary expansion of a . Thus if $a = 35$, $a = 100011_2$, and so

$$\begin{aligned} \overline{y^a} &= \overline{y^{2^{32+2+1}}} \\ &= \overline{y^{2^{32}} \cdot y^{2^2} \cdot y^{2^1}} \end{aligned} \quad (5.3)$$

This operation can be performed simultaneously on all values of a using quantum gates analogous to the gates a classical computer would use to perform the operation on a single value of a . This is the first example of where the quantum algorithm achieves exponential speed-up.

Once the auxiliary register is created it is measured. Since the qubits in this auxiliary register were entangled to the main register, this changes the probabilities of different states in the main register. The state of the main register is changed from

$$\sum_{a=0}^{q-1} h(a) |a\rangle |y^a \bmod N\rangle \quad (5.4)$$

where $h(a) = \frac{1}{\sqrt{q}}$, to

$$|\phi_l\rangle = \sum_{a=0}^{q-1} f(a) |a\rangle \quad (5.5)$$

where if m is the value that's measured, l is the smallest a for which $g(a) = m$, $n = 0, 1, 2, \dots$, r is the period of g and k is the number of values a for which $g(a) = m$,

$$f(a) = \begin{cases} \frac{1}{\sqrt{k}} & , \quad a = l + rn \\ 0 & , \quad a \neq l + rn \end{cases} , \quad a = 0, 1, \dots, q-1. \quad (5.6)$$

The function f is the probability amplitude of the different values in the main register and is now zero everywhere except at periodic values of a . It is interesting to note that f does not depend directly on m . It turns out that the period r is all that is important and the actual value of m is useless information. Finding an m is done solely to make f periodic.

On a classical computer each calculation must be performed individually, and this becomes a daunting task. We are better off not even trying to find all the values, but finding r as is done in Appendix B. However, since we want to mimic the operation of a quantum computer and our numbers are small we will calculate all the values, but not in the same fashion. In `createMeasure_register` we first find m randomly by choosing an a ("chosen") calculating $m = y^{\text{chosen}}$. We then cycle through all q values of $g(a)$, counting how many times $g(a) = g(m)$, which gives us our value k ("occurs"). We then cycle through $g(a)$ again, and set $f(a) = 1/\sqrt{k}$ when $g(a) = l$ and $f(a) = 0$ otherwise. Although this may seem contrived, the act of measuring the auxiliary register on a quantum computer automatically sets these equal probabilities, and a classical computer has no mechanism to do such.

5.3 Finding the spectrum: The DFT_q

With f containing a periodic function the next step is to find that period. This is done through the quantum discrete Fourier transform modulo q

(DFT_q), which is a quantum adaptation of the fast Fourier transform (FFT). Where the FFT finds the spectrum of an ordered set of values contained in many registers, the DFT_q finds the spectrum of an ordered set of probability amplitudes contained in a single quantum register. The discussion and notation here will basically follow the discussion by Ekert and Jozsa [EJ96], adapted for the purpose of understanding our simulation.

5.3.1 The quantum theory of DFT_q

A quantum gate performs an operation on a qubit or multiple qubits in the quantum register, just as a classical logic gate performs an operation on a bit or multiple bits. The DFT_q uses two quantum gates. The first,

$$A_j = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad (5.7)$$

is a gate that operates on a single qubit. The second,

$$B_{j,k} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi/2^{k-j}} \end{pmatrix}, \quad (5.8)$$

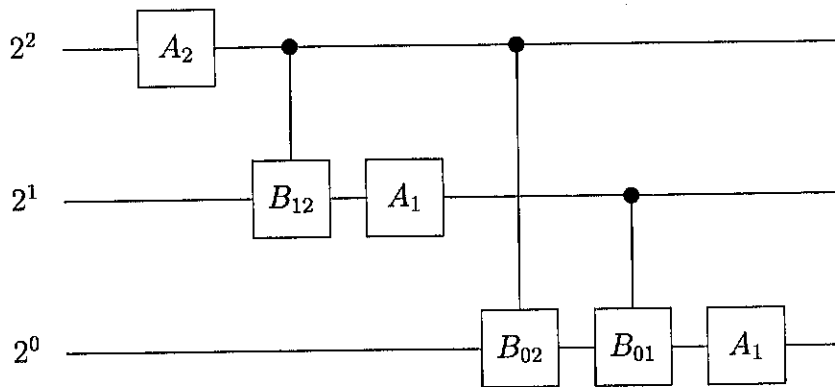
operates on two qubits. The $1/\sqrt{2}$ in A is a normalization factor. All gates in a quantum computer are required to be unitary, meaning that as long as the total probability of the quantum register was one, after any number of gates have operated on it, it will still be one.

The transform is performed by making L passes on the register, running from $j = L - 1$ to $j = 0$. Each pass begins with a series of B gates, beginning with $B_{j,j+1}$ and ending with $B_{j,L-1}$, followed by A_j . On the first pass, since $j = L - 1$, there are no operations done by B gates, on the second there is one, on the third, two, etc. So on a three bit register the DFT_q is performed by the following matrix equation:

$$\tilde{\mathbf{V}} = A_0 B_{01} B_{02} A_1 B_{12} A_2 \mathbf{V}, \quad (5.9)$$

where \mathbf{V} is our initial state, $\tilde{\mathbf{V}}$ is the transformed state, and gates operate from right to left on \mathbf{V} . The schematic for this operation is shown in Fig (5.1).

Figure 5.1: Schematic for 3-bit DFT_q represented by Eq (5.9). (Adapted from [EJ96].)



At the end of these passes we define a new counter, by letting qubits of c be the same as the qubits of a read in reverse order. So if $|a\rangle = |001\rangle = |1\rangle$, then $|c\rangle = |100\rangle = |4\rangle$ and $f(a) = \tilde{f}(c)$. So the effect of the entire DFT_q on our register is

$$\text{DFT}_q : \sum_{a=0}^{q-1} f(a) |a\rangle \rightarrow \sum_{c=0}^{q-1} \tilde{f}(c) |c\rangle,$$

where

$$\tilde{f}(c) = \begin{cases} \frac{1}{\sqrt{r}} & , \quad c = 0, \frac{q}{r}, \frac{2q}{r}, \dots, \frac{(r-1)q}{r} \\ 0 & , \quad \textit{otherwise} \end{cases} \quad (5.10)$$

We will return to this equation shortly.

5.3.2 The classical implementation of DFT_q

Since the DFT_q is a quantum algorithm we cannot perform it with the efficiency described above. This portion of the simulation (contained in `DFT_Ekert.h`) requires using $q \times q$ matrices, requiring a great deal of computer memory for even the smallest values of N , limiting the numbers we can factor. Classical simulation does, however, allow us to watch the evolution of the state of the main register throughout the DFT_q .

For the purposes of understanding the DFT_q in our simulation it is helpful to look at the function, $f(a)$, as a column vector containing the probability amplitudes of the different states. Since the probability amplitudes are potentially complex values we simulate the register by creating a $q \times 1$ array of complex variables, indexed from 0 to $q - 1$. We let the complex value in the array cell with index number 0 represent the probability amplitude of $|0\rangle$, $f(0)$, the complex number in array cell with index number 1 represent $f(1)$ and so on.

As shown in Damian Menscher's thesis [Men97], it is helpful to look at gate operations using vectors and matrices. If we begin with three qubits,

$$\begin{aligned} |A\rangle &= \alpha_0 |0\rangle + \alpha_1 |1\rangle, \\ |B\rangle &= \beta_0 |0\rangle + \beta_1 |1\rangle, \\ |C\rangle &= \gamma_0 |0\rangle + \gamma_1 |1\rangle, \end{aligned} \tag{5.11}$$

the state of the register formed by the entangled qubits can be represented as

$$\mathbf{V} = \begin{pmatrix} \alpha_0 \beta_0 \gamma_0 \\ \alpha_0 \beta_0 \gamma_1 \\ \alpha_0 \beta_1 \gamma_0 \\ \alpha_0 \beta_1 \gamma_1 \\ \alpha_1 \beta_0 \gamma_0 \\ \alpha_1 \beta_0 \gamma_1 \\ \alpha_1 \beta_1 \gamma_0 \\ \alpha_1 \beta_1 \gamma_1 \end{pmatrix}. \tag{5.12}$$

Since our array's index corresponds directly to the possible values of $|a\rangle$, a gate operation on this vector can now be represented by a unitary matrix, M , which we can multiply by \mathbf{V} .

To perform the DFT_q we need to create matrices which will operate on the entire vector \mathbf{V} simulating the action of the A and B matrices, which operate on a specific bit. For example, since $A_1 \mathbf{V}$ yields

$$\begin{aligned} \beta_0 &\mapsto \frac{\beta_0 + \beta_1}{\sqrt{2}}, \\ \beta_1 &\mapsto \frac{\beta_0 - \beta_1}{\sqrt{2}}, \end{aligned} \tag{5.13}$$

we would like to find an A_1 such that

$$A_1 \begin{pmatrix} \alpha_0\beta_0\gamma_0 \\ \alpha_0\beta_0\gamma_1 \\ \alpha_0\beta_1\gamma_0 \\ \alpha_0\beta_1\gamma_1 \\ \alpha_1\beta_0\gamma_0 \\ \alpha_1\beta_0\gamma_1 \\ \alpha_1\beta_1\gamma_0 \\ \alpha_1\beta_1\gamma_1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} \alpha_0\beta_0\gamma_0 + \alpha_0\beta_1\gamma_0 \\ \alpha_0\beta_0\gamma_1 + \alpha_0\beta_1\gamma_1 \\ \alpha_0\beta_1\gamma_0 - \alpha_0\beta_0\gamma_0 \\ \alpha_0\beta_1\gamma_1 - \alpha_0\beta_0\gamma_1 \\ \alpha_1\beta_0\gamma_0 + \alpha_1\beta_1\gamma_0 \\ \alpha_1\beta_0\gamma_1 + \alpha_1\beta_1\gamma_1 \\ \alpha_1\beta_1\gamma_0 - \alpha_1\beta_0\gamma_0 \\ \alpha_1\beta_1\gamma_1 - \alpha_1\beta_0\gamma_1 \end{pmatrix}. \quad (5.14)$$

The desired matrices can be created through tensor products of the A and B matrices and the 2×2 identity matrix, I . So the matrix performing the operation A_1 in the above example is

$$\begin{aligned} A_1 = I \otimes A \otimes I &= \begin{pmatrix} \frac{1}{\sqrt{2}}I & \frac{1}{\sqrt{2}}I & 0 \cdot I & 0 \cdot I \\ \frac{1}{\sqrt{2}}I & -\frac{1}{\sqrt{2}}I & 0 \cdot I & 0 \cdot I \\ 0 \cdot I & 0 \cdot I & \frac{1}{\sqrt{2}}I & \frac{1}{\sqrt{2}}I \\ 0 \cdot I & 0 \cdot I & \frac{1}{\sqrt{2}}I & -\frac{1}{\sqrt{2}}I \end{pmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix}. \end{aligned} \quad (5.15)$$

All the A_j matrices for a three qubit transform can be constructed in a similar manner.

Since L varies depending on N , we need a more general equation for our matrices. The general A_j matrix for an arbitrary number of qubits, L , is

$$A_j = I^0 \otimes I^1 \otimes \dots \otimes I^{j-1} \otimes A^j \otimes \dots \otimes I^{L-1}, \quad (5.16)$$

where the superscripts denote the position of the matrices in the sequence of tensor products.

As can be guessed, the A matrices require large amounts of memory, and serve as the limiting factor in the program. Since A is a $q \times q$ square matrix, the maximum size of q (QMax, set in `shor_found.h`) can be set according to the limits of the computer system (1024 on the HP PA RISC 2.0 machine). The operation of an A_j gate is performed on the main register as a square matrix multiplied by a vector.

The B matrices could be constructed in a similar fashion to the A , but are not since all the entries in any given B matrix off of the main diagonal are zero. We can place the main diagonal entries in an array and multiply these by the main register, element by element.

We mentioned earlier that one advantage of simulating quantum algorithms on a classical computer is that we can watch the state of the main register as the DFT_q progresses. On a quantum computer when a measurement is made we get a single value out. In our simulation we can display a histogram of the entire probability of the register. In `displayREGISTER.h` we have a couple different versions of the display function, and can use these to examine different aspects about the DFT_q 's operation. General operation would use `displayREGISTERhist()` to produce a simple histogram in the commandline window after each A matrix is multiplied by the register, allowing us to watch the DFT_q at work, which could not be done with a quantum computer. At the end of the DFT_q we see the period. Below is output from the DFT_q with $N = 21$ and $y = 17$:

```

0 - 27 *****
28 - 55
56 - 83 *
84 - 111 *****
112 - 139
140 - 167
168 - 195 *****
196 - 223
224 - 251
252 - 279 *****
280 - 307
308 - 335
336 - 363 *****

```

```

364 - 391
392 - 419
420 - 447 *****
448 - 475
476 - 503
504 - 511

```

for which $r = 6$.

The other functions in `displayREGISTER.h` allow us to check different aspects of our program. Equ. (5.10) predicts that there should be a maximum at $c = 2q/r = 512/3$. But c is supposed to be an integer. We can zoom-in on this value using `displayREGISTERrange_hist()`

```

169
170 ***
171 *****
172 *
173

```

to see that the probabilities are spread around the value. In comparison, where c is an integer we have only one value.

```

254
255
256 *****
257
258

```

We will see that the spread probability is a problem later on.

5.4 Extracting and making use of the period

Once the DFT_q is completed a measurement is taken of the main register resulting in one of the possible values of c ,

$$c = 0, \frac{q}{r}, \frac{2q}{r}, \dots, \frac{(r-1)q}{r}. \quad (5.17)$$

To determine a value of c in our simulation we calculate a random value between 0 and 1 and add the total probability of the register sequentially until we surpass the random value. We next attempt to use a continued fraction to find r . Continued fractions can be used to find a fractional approximation of rational or irrational numbers to a desired tolerance. The resulting fraction is in lowest terms, and is judged as being a good or bad approximation by an inequality, which for Shor's algorithm is

$$\left| \frac{c}{q} - \frac{c'}{r} \right| \leq \frac{1}{2q}. \quad (5.18)$$

According to a specific algorithm we use the continued fraction to find an approximation for our measured value c . We then check to see if our approximation is within the desired bound. If it is we call this r (or a factor of r). If it is not we continue with the algorithm to generate a fraction that more closely approximates the given value. (The continued fraction algorithm is briefly explained in Appendix C.)

Using the continued fraction we can find r or one of its factors. From there we can try this number in Eq. (4.5). Using the above example, where $N = 21$ and $y = 17$, $c = 171$ tells us that $c'/r = 1/3$. By multiplying small integer values by 3 for r 's to try in

$$N = \gcd\left(\overline{y^{\frac{r}{2}} + 1}, N\right) \times \gcd\left(\overline{y^{\frac{r}{2}} - 1}, N\right) \quad (5.19)$$

we find that $r = 6$ and our program has successfully factored 21.

We can see, though, that we will not always easily find r and we may need to start over. If we measure $c = 0$ then there is no way to extract useful information. Also, some numbers we measure won't satisfy the inequality until the denominator is q . This is true if we measure 170 rather than 171 in the above example. The only approximation within the bound is $c' = c$ and $r = q$, which doesn't satisfy Eq. (4.5).

5.5 Discussion

The computer program `Shor.cpp` in Appendix A works well for $N = 15$ and $N = 21$. Most other composite numbers less than 31 do not work as well, for reasons from number theory (as discussed in Appendix B). The difference in time taken between $N = 15$ ($q = 256$) and $N = 21$ ($q = 512$) gives a taste of the exponential slow-down that occurs with the algorithm being run on a classical computer. We have also been able to watch the effect of the DFT_q on a quantum register, which is something that could never be done on a quantum computer.

Chapter 6

Conclusion

The adage, "Nothing is constant but change", seems to apply well to the computer industry. The coming years will provide new challenges and opportunities, which may include alternative technologies. It is hard to say whether quantum computing is one of them. This simulation serves as another reminder that the algorithms have arrived before the hardware. We have been able to get a good picture of how a quantum computer differs from a classical one.

In our simulation of Shor's algorithm we have seen how a quantum computer works differently than a classical one. The ability of the quantum computer to act on single qubits to change the probability of all the possible states of the register has no classical analog, and our code resulted in exponential slow-down.

There are a couple of things which could be done to improve the program. The most inefficient part of the simulation is the A matrix. I chose to construct the entire matrix since this best matches the gate representation as discussed in chapter 5. One change that could be made in this C++ code would be to change the A matrix to a smaller matrix using bitwise math, decreasing the required memory dramatically. Since the only values the matrix needs to hold are 0, 1 and -1, this could be done with two bits per matrix entry. Another obvious change would be to not construct the A matrices, but perform the operations based on the bitwise representation of a . I now think that a careful treatment based on the binary representation of a would more closely mimic the operation of the A gate than the matrix. It would also save time and memory, allowing the factoring of larger numbers.

Appendix A

Program simulating Shor's algorithm

The following is the complete text of the simulation program of Shor's algorithm. The program has been subdivided into several header files, which are interdependent. The order the files have been listed in is according to when they are called in the program: Shor.cpp, shor_main.h, DFT_Ekert.h, displayREGISTER.h, shor_found.h numbers.h and complex.h. Further explanation of what the functions do can be found in the comments in the code.

A.1 Shor.cpp

```
/****** Shor.cpp *****/
* Main program for Shor.cpp. The program is really in the header files. *
/*******/

#include "shor_main.h"

main(){
    int N;
    shor_main Shor_main;

    // Input number to factor
    cout << endl << endl << "Input a number to factor: ";
    cin >> N;

    /* Run Shor's algorithm (the Shor object file). If the algorithm returns TRUE
```

```

        then display factors. If not, say the algorithm failed. */

if (Shor_main.run_Shor(N)){
    cout << endl << "The factors of " << N << " are "
        << Shor_main.mainNums.fac1 << " and " << Shor_main.mainNums.fac2
        << "." << endl << endl << endl;
}
else
    cout << endl << "Shor's algorithm found: "
        << Shor_main.mainNums.fac1 << " and " << Shor_main.mainNums.fac2
        << "." << endl << endl << endl;
}

```

A.2 shor_main.h

```

/***** shor_main.h *****/
* This file contains the necessary program to run Shor's *
* algorithm on a classical computer (except for the discrete*
* Fourier transform, which is found in DFT_Ekert.h ) *
*****/

#include "DFT_Ekert.h"

class shor_main {
public:
    int Q; //Although also in mainNums, very useful;
    shor_mainNumbers mainNums; //Main numbers for shor's algorithm
    set_mainNums (int); /* Based on N (number to be factored) determines
        other main numbers */
    superposition mainReg; //This is the main register for shor's algorithm
    createMeasure_register(); //Initializes main register to start algorithm
    BOOLEAN run_Shor (int); /* Runs shor's algorithm as the primary function
        * call in the main program. */
    int measure_transfrmdRegister();
};

/***** run_Shor *****/

BOOLEAN shor_main::run_Shor(int N){ /* This is the object that runs
    Shor's algorithm */
    DFT_Q dft;
    int position, ct, r;
    BOOLEAN rFound=FALSE;

```

```

// Based on the input, create numbers needed in shor's algorithm and then
// determine whether the computer can factor the number (if Q > QMax)
srand (time(NULL));

set_mainNums (N);
if (mainNums.Q > QMax){
    cout << "N is too large for this program to factor it." << endl;
    return FALSE;
}

// Check whether random number chosen is a factor. If it is exit to main prog.
position = (int) GCD (N, mainNums.y);
if (position != 1){
    mainNums.fac1 = position;
    mainNums.fac2 = N/position;
    return TRUE;
}

Q = mainNums.Q; //Determine size of register needed
//This is actually already done in set_mainNums()
dft.Q = Q;
dft.mainNums = mainNums;

cout << endl << "Preparing and measuring part of main register. ";
createMeasure_register();
cout << endl;
for (ct=0; ct<Q; ct++)
    dft.mainReg[ct] = mainReg[ct];
cout << endl;

cout << "Performing discrete fourier transform on the main register. ";
dft.perform_dft();
for (ct=0; ct<Q; ct++)
    mainReg[ct].probVal = dft.mainReg[ct].probVal;
cout << endl;

cout << "Measuring main register. ";
position = measure_transfrmdRegister();

cout << endl;

cout << "Using continued fraction to approximate period.";
r = contFrac (position, mainNums.Q);

```

```

cout << endl;

cout << "Determining factors. ";
for (ct = 1; ct < 5; ct++){
    mainNums.fac1 = (int) GCD ((int) pow(mainNums.y, (float)(r*ct/2)) +1, N);
    mainNums.fac2 = (int) GCD ((int) pow(mainNums.y, (float)(r*ct/2)) -1, N);
    if ((mainNums.fac1 !=1) && (mainNums.fac2 != 1)){
        cout << endl << "Period was " << r*ct << "." << endl;
        rFound=TRUE;
        ct = 5;
    }
}

cout << "Position was " << position << endl
    << "Original random number was " << mainNums.y << endl;

if (rFound == FALSE){
    cout << "Program was unable to determine r.";
    return FALSE;
}
return TRUE;
}

/***** set_mainNums *****/
BOOLEAN shor_main::set_mainNums (int N){
    mainNums.N = N;
    mainNums.L = (int)((log(N*N))/LN2 +1);
    mainNums.Q = (int) pow(2, mainNums.L);
    //mainNums.y = (rand() % (N-2)) + 2;
    cout << endl << "Please input an integer: " ;
    cin >> mainNums.y;

    return 0;
}

/***** createMeasure_register *****/
/****Creates main register for the algorithm*/

shor_main::createMeasure_register(){

    int a, chosen, occurs=0;
    float probability;

    //Creates a "superposition" of values in part of our register
    mainReg[0].yamodN = 1;

```

```

for (a = 1; a < Q; a++)
    mainReg[a].yamodN = (mainReg[a - 1].yamodN * mainNums.y) % mainNums.N;
chosen = 16; //mainReg[(rand() % (Q - 1)) + 1].yamodN; //Take measurement

//Create probability of function being measured
for (a = 0; a < Q; a++)
    if (mainReg[a].yamodN == chosen) occurs++;

probability = 1 / ( sqrt((float)occurs) );

for (a = 0; a < Q; a++){
    mainReg[a].probVal.Im = 0;
    if (mainReg[a].yamodN == chosen)
        mainReg[a].probVal.Re = probability;
    else mainReg[a].probVal.Re = 0;
}

for (a = 0; a < Q; a++)
    mainReg[a].probValmodSq = mainReg[a].probVal.Re * mainReg[a].probVal.Re;
return 0;
}

/***** measure_transfrmdRegister() *****/

int shor_main::measure_transfrmdRegister(){
    int c;
    float val, total;

    total = 0;
    for (c=0; c < Q; c++){
        mainReg[c].probValmodSq = modSqr_cplx (mainReg[c].probVal);
        total += mainReg[c].probValmodSq;
    }

    cout << "Total probability = " << total << ". ";
    val = ((float) (rand() +rand() +rand() +rand() +rand() +rand() +rand()
        +rand() +rand() +rand())) / (float) (10*RAND_MAX)) * total;

    total = 0;
    for (c = 0; val > total; c++)
        total += mainReg[c].probValmodSq;

    c--;
    return c;
}

```

A.3 DFT_Ekert.h

```

/***** DFT_Ekert.h *****/
* header file containing routines for a working classical *
* simulation of the quantum discrete fourier transform *
* in paper by Ekert and Jozsa (Rev of Mod Phys, July 96). *
*****/

#include "displayREGISTER.h"

typedef float MATRIX [QMax][QMax];

/***** DFT_Q class *****/
class DFT_Q {
public:

    superposition mainReg, newReg;
    perform_dft();
    shor_mainNumbers mainNums;
    int Q, L;
    MATRIX A; //The A matrix from Ekert and Jozsa's paper
    A_create (int); //Creates the proper A matrix for the current
    situation
    displayMatrix (MATRIX);
    void B_mult_REG (int, int);
    A_mult_REG();
    void bitReverse();
    void bitReverseNew(); //does same thing as bitReverse, but
    //only changes newReg and not mainReg
};

/***** perform_dft *****/
DFT_Q::perform_dft(){

    int ct1, ct2;
    Q = mainNums.Q;
    L = mainNums.L;

    for (ct1 = L - 1; ct1 >= 0; ct1--){
        for (ct2 = L - 1; ct2 > ct1; ct2--){
            B_mult_REG (ct1, ct2);
            cout << endl << "B" << ct2 << ct1;
        }
        A_create(ct1);
        A_mult_REG();
    }
}

```

```

    cout << endl << "A" << ct1;

// The following two lines display the state of the
// register after each pass through the DFT
    bitReverseNew();
    displayREGISTER(newReg, Q);

}

    bitReverse();
    for (ct1 = 0; ct1 < Q; ct1++)
        mainReg[ct1].probValmodSq = modSqr_cmplx(mainReg[ct1].probVal);
//displayREGISTER(newReg, Q);
    return 0;
}

/***** A_create *****/
* The purpose of this function is to create the A matrix capable *
* of operating on a single bit using tensor products, Since the *
* classical simulation doesn't work the way the quantum computer *
* would.
*****
* Position: The position is given so that if there are 4 bits in*
* the calculation (L == 4) then the ordering of our matrices is *
* 2^3, 2^2, 2^1, 2^0. The sequence of for loops
*
*****/

DFT_Q::A_create (int position){

    int pos_ct, // Position counter (which tensor are we multiplying),
                // for number manip
    i, j, ct_temp, tempval; // for number manipulation

// Fill A-Matrix with ones over square root of two
// (we'll get rid of them as we multiply the other matrices)
    for (i=0; i < Q; i++)
        for (j=0; j < Q; j++)
            A[i][j] = InvSQRT2;

//Multiply identity matrices before A
    for (pos_ct = L -1; pos_ct > position; pos_ct--){
        tempval=1;

```



```

    for (ct_temp=0; ct_temp < pos_ct; ct_temp++)
        tempval = tempval*2;

    for (i = 0; i < Q; i++)
        for (j = 0; j < Q; j++){
            if (((i/tempval)%2)!=(j/tempval)%2))
                A[i][j] = 0;
        }
    }

//Multiply A matrix (no for loop, but happens when pos_ct=position)
for (i = 0; i < Q; i++)
    for (j = 0; j < Q; j++){
        tempval=1;
        for (ct_temp=0; ct_temp < position; ct_temp++)
            tempval = tempval*2;
        if (((i/tempval)%2)==1) && (((j/tempval)%2)==1))
            A[i][j] = A[i][j] * -1;
    }

//Multiply identity matrices after A matrix
for (pos_ct = position-1; pos_ct >= 0; pos_ct--){
    tempval=1;
    for (ct_temp=0; ct_temp < pos_ct; ct_temp++)
        tempval = tempval*2;

    for (i = 0; i < Q; i++)
        for (j = 0; j < Q; j++){
            if (((i/tempval)%2)!=(j/tempval)%2))
                A[i][j] = 0;
        }
    }

return 0;
}

/***** A_mult_REG *****/
DFT_Q::A_mult_REG(){

    int ct1, ct2;
    cmplx sum;

    for (ct1 = 0; ct1 < Q; ct1++){

```

```

    sum.Re=0; sum.Im=0;
    for (ct2 = 0; ct2 < Q; ct2++){
        sum = add_cmplx(sum, scalMult_cmplx(A[ct1][ct2], mainReg[ct2].probVal));
    }
    newReg[ct1].probVal = sum;
}

for (ct1 = 0; ct1 < Q; ct1++){
    mainReg[ct1].probVal=newReg[ct1].probVal;

return 0;
}

/***** B_mult_REG *****/
void DFT_Q::B_mult_REG(int ind1, int ind2){

int difference, tempval1, tempval2, ct;
float diff2=1;
cplx exponential;

difference = ind1 - ind2;
if (difference < 0)
    difference = -difference;

for (ct=0; ct<difference; ct++)
    diff2*=2;

exponential.Re = cos (3.14159285358/diff2);
exponential.Im = sin (3.14159285358/diff2);

tempval2=1;
for (ct=0; ct < ind2; ct++)
    tempval2 = tempval2*2;

tempval1=1;
for (ct=0; ct < ind1; ct++)
    tempval1 = tempval1*2;

for (ct=0; ct < Q; ct++)
    if ((ct/tempval1)%2==1 && (ct/tempval2)%2==1)
        mainReg[ct].probVal=mult_cmplx(mainReg[ct].probVal, exponential);
}

/*****bitReverse()*****/

```

```

void DFT_Q::bitReverse (){
    int ct1, ct2, ct3, Log2_ct, tempval, newLocat, difference, diffPow2;

    for (ct1 = 0; ct1 < Q; ct1++){
        newLocat = 0;
        Log2_ct = (int)(log((float)ct1)/LN2+1);
        tempval = 1;
        for (ct2=0; ct2 < Log2_ct; ct2++){
            tempval = tempval*2;
            if (ct2 == 0)
                tempval /= 2;
            if (((ct1/tempval) % 2) == 1){
                difference = L - 1 - ct2;
                if (difference < 0)
                    difference = -difference;
                diffPow2 = 1;
                for (ct3 =0; ct3 < difference; ct3++){
                    diffPow2 *= 2;
                    newLocat+= diffPow2;
                }
            }
        }

        newReg [newLocat].probVal = mainReg [ct1].probVal;
    }

    for (ct1 = 0; ct1 < Q; ct1++)
        mainReg [ct1].probVal = newReg [ct1].probVal;
}

/*****bitReverseNew()*****/

void DFT_Q::bitReverseNew (){
    int ct1, ct2, ct3, Log2_ct, tempval, newLocat, difference, diffPow2;

    for (ct1 = 0; ct1 < Q; ct1++){
        newLocat = 0;
        Log2_ct = (int)(log((float)ct1)/LN2+1);
        tempval = 1;
        for (ct2=0; ct2 < Log2_ct; ct2++){
            tempval = tempval*2;
            if (ct2 == 0)
                tempval /= 2;
            if (((ct1/tempval) % 2) == 1){

```

```

        difference = L - 1 - ct2;
        if (difference < 0)
            difference = -difference;
        diffPow2 = 1;
        for (ct3 = 0; ct3 < difference; ct3++)
            diffPow2 *= 2;
        newLocat += diffPow2;
    }
}
newReg [newLocat].probVal = mainReg [ct1].probVal;
}
}

```

A.4 displayREGISTER.h

```

/***** displayREGISTER.h *****/
* Functions for displaying histograms of the probability *
* of measuring different states of main register *
*****/

#include "shor_found.h"

void displayREGISTER (superposition, int); // mainReg and Q
void displayREGISTERhist (superposition, int); // mainReg and Q
void displayREGISTERLog2_hist (superposition, int); // mainReg and Q
void displayREGISTERrange_hist (superposition, int, int, int); // mainReg, Q,
//center (where to center histogram) and radius
void displayREGISTERnum (superposition, int); // mainReg and Q
void displayTotalProb (superposition, int); // mainReg and Q

/***** displayReg Set *****/
* The idea is that the format of display may be chosen by *
* putting the function creating the desired format into this other *
* function. So in DFT_Ekert only this function needs to be called *
*****/

void displayREGISTER(superposition mainReg, int Q){

    int i, center, radius;
    superposition newReg;

    for (i=0; i<Q; i++)
        mainReg[i].probValmodSq = modSqr_cplx (mainReg[i].probVal);
}

```

```

    cout << endl << endl ;

// The following can be un-commented if one desires to use the
// displayREGISTERrange_hist function.

/*      cout << "Please input center and radius of histogram: " << endl;
        cout << "Center: "; cin >> center;
        cout << "Radius: "; cin >> radius;
        cout << "Magnification around " << center << endl;
        displayREGISTERrange_hist (mainReg, Q, center, radius);
*/

    displayREGISTERhist (mainReg, Q);
    displayTotalProb (mainReg, Q);
}

/***** displayREGISTERhist() *****/
* displays in a histogram format *
*****/

void displayREGISTERhist(superposition mainReg, int Q){
    int i, j, k, range, total_i;
    float total_f;

    range = Q / 18; /* range will be the number of numbers between      *
                    * which the histogram displays, ie. if range = 4,*
                    * then it will show entries for 0-3, 4-7 ...      */
    if (range < 1)
        range = 1;

    for (i=0; i<Q; i++)
        mainReg[i].probValmodSq = modSqr_cmplx (mainReg[i].probVal);
    for (i=0; i<Q; i+=range){
        if (i+range > Q)
            range = Q - i;

        total_f = 0;
        for (k=0; k<range; k++)
            total_f+=mainReg[i+k].probValmodSq;
        total_i=(int)(100 * total_f + 0.5);
        //The 0.5 is for rounding to nearest rather than integer part

        cout << setw (3) << i << " - " << setw(3) << i+range-1 << setw (3);
    }
}

```

```

    for (j = 0; j < total_i; j++)
        cout << "*";
    cout << endl;
}
}

/***** displayREGISTERLog2_hist() *****/
*   displays in a histogram format   *
*****/
void displayREGISTERLog2_hist(superposition mainReg, int Q){
    int i, j, k, range, total_i;
    float total_f;

    range = Q / 32; //range will be the number of numbers between which
        // the histogram displays, ie. if range = 4, then it will show
        //entries for 0-3, 4-7 ...

    if (range == 0)
        range = 1;

    for (i=0; i<Q; i++)
        mainReg[i].probValmodSq = modSqr_cplx (mainReg[i].probVal);
    for (i=0; i<Q; i+=range){
        if (i+range > Q)
            range = Q - i;

        total_f = 0;
        for (k=0; k<range; k++)
            total_f+=mainReg[i+k].probValmodSq;
        total_i=(int)(100 * total_f + 0.5);
        //The 0.5 is for rounding to nearest rather than integer part
        total_i = (int)(log((float)total_i)/LN2);

        cout << setw (3) << i << " - " << setw(3) << i+range-1 << setw (3);

        for (j = 0; j < total_i; j++)
            cout << "*";
        cout << endl;
    }
}

/***** displayREGISTERrange_hist() *****/
* displays in a histogram format, centered around *

```

```

* "center" showing "radius" values on either side *
*****/

void displayREGISTERrange_hist(superposition mainReg, int Q,
                               int center, int radius){
    int i, j, range_hi, range_low, total_i;
    float total_f;

    if (radius < 0)
        radius = -radius;

    range_low = center - radius; //range_low is lowest index shown
    if (range_low < 0) range_low = 0;

    range_hi = center + radius; //range_hi is highest index shown
    if (range_hi >= Q) range_hi = Q - 1;

    for (i=range_low; i <= range_hi; i++){
        total_f = modSqr_cmplx (mainReg[i].probVal);
        total_i=(int)(100 * total_f + 0.5);
        //The 0.5 is for rounding to nearest rather than integer part
        cout << setw (3) << i << " ";

        for (j = 0; j < total_i; j++)
            cout << "*";
        cout << endl;
    }
}

/***** displayREGISTERnum() *****/

void displayREGISTERnum(superposition mainReg, int Q){
    int i;

    for (i=0; i<Q; i++){
        if (mainReg[i].probValmodSq < 10.001){
            cout << setw (4) << i << " " << setprecision(10)
                << mainReg[i].probValmodSq << " " << setw (3)
                << mainReg[i].yamodN << endl;
        }
    }
}

/*****displayTotalProb()*****/

```

```

* Adds up the probability in the main register *
* to check that all operations are unitary *
*****/

void displayTotalProb(superposition mainReg, int Q){
    int c;
    float total;

    total = 0;
    for (c=0; c < Q; c++){
        mainReg[c].probValmodSq = modSqr_cmplx (mainReg[c].probVal);
        total += mainReg[c].probValmodSq;
    }

    cout << "Total probability = " << total << ". ";
}

```

A.5 shor_foundation.h

```

/*****shor_found.h*****/
* This is the foundation (hence _found) header file, *
* where most of the useful header files and defined *
* constants have been placed, as well as important *
* type definitions. *
*****/

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

#include "numbers.h"
#include "complex.h"

#define TRUE 1
#define FALSE 0

#define QMax 1024 // This is the maximum value of q we will allow.
// It is set according to memory limitations on
// the A matrix in DFT_Ekert.h.

// Numeric constants: These numbers will show up in our program
// so often it will be faster to define them as constants.
#define SQRT2 1.41421356237

```



```

#define InvSQRT2 0.707106781187
#define LN2 0.6931471805599

typedef struct reg{ // This is the probabilities register--
    cmplx probVal; // probVal is the probability function values (complex
    int yamodN; // of course) while yamodN is the value of y^a mod N
    float probValmodSq; // and probValmod is modulus of the probability value.
}; // Since we are really interested in an array of this
// type, representing the superpositions of the states,
// this is how it will be used, as type REG.

typedef reg superposition [QMax];
typedef int BOOLEAN;

struct shor_mainNumbers{
    int N, Q, L, y, fac1, fac2;
    /* Based on the model of Shor's algorithm set down by Ekert and Jozsa
    in their paper (in Rev. of Mod. Physics), these are the main numbers
    for a run of shor's algorithm, so they will be set at the beginning
    in the shor_mainStructure object and only passed as needed: N is
    number to factor, y is number coprime to N. */
};

```

A.6 numbers.h

```

/*****numbers.h*****/
* This header file contains the number theoretic *
* functions for use in Shor's algorithm. This *
* includes GCD--Euclid's method to find greatest *
* common denominator contfrac--an algorithm to *
* approximate a fraction by a simpler fraction. *
*****/

//Function Declarations

unsigned long GCD (unsigned long, unsigned long);
int contFrac (int, int);

/***** GCD *****/
* Uses Euclid's method to determine greatest *
* common denominator of two numbers. *
*****/

```

```

unsigned long GCD (unsigned long num1, unsigned long num2){

unsigned long big, small, temp;

if (num1 > num2){
    big = num1;
    small = num2;
}
else {
    big = num2;
    small = num1;
}

while (small!=0) {
temp = small;
small = big % small;
big = temp;
}

return big;

}

/***** contFrac() *****/
* Uses standard algorithm for finding a simpler *
* approximation to a fraction. *
*****/

#define LIMIT 15 /* Limits depth of fraction (irrational
    * numbers by definition are infinite) */
#define A_i 0 //Defining A-i, P_i and Q_i this way allows
#define P_i 1 //us to use these names for indexes in the
#define Q_i 2 //\array rather than numbers.

int contFrac(int num, int denom){

    int i, reg[3][LIMIT];
    double run_d, frac, test;

    test= 1/(2 * (double) denom);
    frac = (double) num / (double) denom;

    i=0;
    run_d= frac;
    reg[A_i][i]= (int) run_d;

```

```

    reg[P_i][i]= reg[A_i][i];
    reg[Q_i][i]= 1;

    i=1;
    run_d= 1/(run_d - (double) reg[A_i][i-1]);
    reg[A_i][i]= (int) run_d;

    reg[P_i][i]= reg[A_i][i]*reg[A_i][i-1] +1;
    reg[Q_i][i]= reg[A_i][i];

//i=2..LIMIT
for (i=2; i < LIMIT; i++){
    run_d= 1/(run_d - (double) reg[A_i][i-1]);
    reg[A_i][i]= (int) run_d;

    reg[P_i][i]= reg[A_i][i]*reg[P_i][i-1]+reg[P_i][i-2];
    reg[Q_i][i]= reg[A_i][i]*reg[Q_i][i-1]+reg[Q_i][i-2];

/* See if fraction is within 1/2Q of input fraction, if so it will
 * be returned: */
    if ((test>=frac-(float)reg[P_i][i]/(float)reg[Q_i][i]) &&
        (frac-(float)reg[P_i][i]/(float)reg[Q_i][i])>= -test){
        denom = reg[Q_i][i];
        i=LIMIT;
    }
}
return denom;
}

```

A.7 complex.h

```

/***** complex.h *****/
 * Basic routines in complex math, requiring *
 * type (herein defined) cplx. Specifically *
 * for simulation of quantum computation. *
 *****/

#include <math.h>
#include <iostream.h>
#include <iomanip.h>

#define RE      Re
#define re      Re
#define IM      Im

```

```

#define Im      Im

struct cmplx{
    float Re, Im; //Complex number, to be used with Re and Im
};

cmplx conj_cmplx (cmplx);
float modSqr_cmplx (cmplx);
cmplx add_cmplx (cmplx, cmplx);
cmplx sub_cmplx (cmplx, cmplx);
cmplx scalMult_cmplx (float, cmplx);
cmplx mult_cmplx (cmplx, cmplx);
cmplx div_cmplx (cmplx, cmplx);
    disp_cmplx (cmplx);

cmplx conj_cmplx (cmplx num1){ //complex conjugate of a number
    cmplx numOut;
    numOut.Re=num1.Re;
    numOut.Im=-num1.Im;
    return numOut;
}

float modSqr_cmplx (cmplx num1){ //modulus squared of a complex number
    float numOut;
    numOut = num1.Re * num1.Re + num1.Im * num1.Im;
    return numOut;
}

cmplx add_cmplx (cmplx num1, cmplx num2){ //addition of two complex numbers
    cmplx numOut;
    numOut.Re = num1.Re + num2.Re ;
    numOut.Im = num1.Im + num2.Im ;
    return numOut;
}

cmplx sub_cmplx (cmplx num1, cmplx num2){ //subtraction of two complex numbers
    cmplx numOut;
    numOut.Re = num1.Re - num2.Re ;
    numOut.Im = num1.Im - num2.Im ;
    return numOut;
}

cmplx scalMult_cmplx (float num1, cmplx num2){
//multiplication of a scalar by a complex number
    cmplx numOut;

```

```

    numOut.Re = num1 * num2.Re ;
    numOut.Im = num1 * num2.Im ;
    return numOut;
}

cmplx mult_cmplx (cmplx num1, cmplx num2){ //multiplication of two complex
numbers
    cmplx numOut;
    numOut.Re = num1.Re * num2.Re - num1.Im * num2.Im ;
    numOut.Im = num1.Re * num2.Im + num1.Im * num2.Re ;
    return numOut;
}

cmplx div_cmplx (cmplx num1, cmplx num2){ //divides num1 by num2 (both complex
no.
    cmplx numOut;
    numOut = mult_cmplx(num1, conj_cmplx(num2));
    numOut.Re = numOut.Re / modSqr_cmplx(num2);
    numOut.Im = numOut.Im / modSqr_cmplx(num2);
    return numOut;
}

disp_cmplx (cmplx numIn){ //Display a complex number in form    a + b i

    if (numIn.Im == 0 && numIn.Re == 0)
        cout << "0";
    else if (numIn.Re == 0)
        cout << numIn.Im << " i";
    else if (numIn.Im == 0)
        cout << numIn.Re;
    else if (numIn.Im < 0)
        cout << numIn.Re << " - " << -numIn.Im << " i";
    else
        cout << numIn.Re << " + " << numIn.Im << " i";

    return 0;
}

```

Appendix B

Exploring the number theory behind Shor's algorithm

The program `trial.cpp` was written to get a feel for the number theory behind Shor's algorithm (see [Mil76]). It does not follow Shor's algorithm strictly, since it does not use the DFT_q to find r , but the results it gives are the same as if Shor's algorithm were run on the same values of N and y . Instead we calculate individually each value of $\overline{y^a}$ beginning with $a = 1, 2, 3, \dots$ and running until $\overline{y^a} = \overline{1}$. The smallest non-zero value of a that gives $\overline{y^a} = \overline{1}$ is the period of $\overline{y^a}$ ($r = a$). This can be done quickly by using the fact that $\overline{y^a} + \overline{1} = \overline{y^a} \cdot \overline{y}$. Choosing random y values and running them through this algorithm is actually a fairly good method for factoring numbers up to 60,000. Here we will first give the code for `trial.cpp` in B.1 and then discuss the results in B.2.

B.1 `trail.cpp`

```
#include<math.h>
#include<iostream.h>
#include<iomanip.h>
#include<fstream.h>
#include#numbers.h#

//unsigned long GCD (unsigned long, unsigned long) -from in numbers.h
unsigned long r_yamodN (unsigned long, unsigned long);
unsigned long yamodN (unsigned long, unsigned long, unsigned long);
```

APPENDIX B. NUMERICALLY EXPLORING SHOR'S ALGORITHM 59

```

main(){
  const unsigned long N=25;
  unsigned long y, r, a, x, fac1, fac2, product, trivCount=0,
                onesCount=0, badCount=0, yct=0;
  float trivRat, usefulRatio;

  ofstream ones    ("ones.txt", ios::out);
  ofstream bad    ("bad.txt", ios::out);
  /* In these text files we list y's that give a product equal to *
   * one (ones.txt) or a product other than one or N (bad.txt).  */

  cout << endl << "    N" << "    y" << "    r" << " y^(r/2)"
        << " fac1" << " fac2" << " prod"<< endl
        << "-----" << endl;

  for (y=2; y<N; y++){
    if (GCD(N,y)==1){

      yct++;
      r= r_yamodN(y, N);
      a=(r-r%2)/2;
      x=yamodN(y, a, N);

      if (r%2==1)
        x= ((unsigned long)(x*sqrt((float)y)))/N;

      fac1=GCD(x+1,N);
      fac2=GCD(x-1,N);
      product = fac1 * fac2;

      cout << setw(6) << N << setw(6) << y << setw(6) << r << setw(8)
            << x << setw(6) << fac1 << setw(6) << fac2 << setw(6)
            << product << endl;

      if ((product==N)&&((fac1==N)|| (fac2==N))){
        trivCount++;
      }

      switch (product){
        case N:
          break;

        case 1:

```

APPENDIX B. NUMERICALLY EXPLORING SHOR'S ALGORITHM60

```

ones << "y=" << setw(6) << y <<" yields r= " << r
    << " and product= " << product << "." << endl;
onesCount++;
break;

default:
bad << "y=" << setw(6) << y <<" yields r= " << r
    << " and product= " << product << "." << endl;
badCount++;
break;
}
}
else
    cout << setw(6) << N << setw(6) << y << endl;
}

trivRat=((float)trivCount)/((float)yct);

usefulRatio=(float)(N-2-trivCount-onesCount)/(float)(N-2);

ones << endl << "For N= " << N << " there were " << yct
    << " possible y values, of which " << onesCount
    << " values of y that " << "gave an answer of 1, "
    << trivCount << " that gave trivial factors "
    << "and " << badCount << " that gave a wrong product." << endl;

cout << endl << "For N= " << N << " there were " << yct
    << " values relatively prime to N." << endl << "Of these "
    << onesCount << " values gave an answer of 1, "
    << trivCount << " gave" << endl << "trivial factors and "
    << badCount << " gave a wrong (but useful) product." << endl
    << "The probability of a y finding a useful factor is "
    << usefulRatio << "." << endl << endl;

ones.close;
bad.close;

return 0;
}

/***** r_yamodN *****/
/* An iterative approach to finding r from y^a mod N, given y and N */

unsigned long r_yamodN (unsigned long y, unsigned long N) {

```


APPENDIX B. NUMERICALLY EXPLORING SHOR'S ALGORITHM61

```

unsigned long out, ct; /* ct keeps track of iterations
                       out holds each successive y^ct(mod N) */
out= y % N;
ct=1;
while(out!=1){
    out= (out * y)% N;
    ct++;
}

return ct;
}

/***** yamodN *****/
/* An iterative approach to finding y^a mod N, given y, a and N */

unsigned long yamodN(unsigned long y, unsigned long a, unsigned long N)
{
    unsigned long ct, out=1;

    for (ct=1; ct <= a; ct++)
        out= (out * y) % N;
    return out;
}

```

B.2 Numeric examples of how Shor's algorithm fails

trial.cpp can help us gain a feel for the numbers that Shor's algorithm produces. Below is printed an output sample for $N = 77$. Not all the possible values of y are shown. To give a feel for the output $r = 2, \dots, 9$ is shown, followed by a couple other groups to give sample output. The program prints blank lines following y 's that are not relatively prime to N . For readability in this list spaces have been placed where y values have been skipped.

N	y	r	$y^{(r/2)}$	fac1	fac2	prod
77	2	30	43	11	7	77
77	3	30	34	7	11	77
77	4	15	43	11	7	77

APPENDIX B. NUMERICALLY EXPLORING SHOR'S ALGORITHM62

77	5	30	34	7	11	77
77	6	10	76	77	1	77
77	7					
77	8	10	43	11	7	77
77	9	15	34	7	11	77
77	23	3	33	1	1	1
77	24	30	76	77	1	77
77	25	15	34	7	11	77
77	36	5	76	77	1	77
77	37	15	20	7	1	7
77	38	30	34	7	11	77
77	53	15	29	1	7	7
77	76	2	76	77	1	77

For $N=77$ there were 59 values relatively prime to N .
 Of these, 5 values gave an answer of 1, 17 gave
 trivial factors and 2 gave a wrong (but useful) product.
 The probability of a y finding a useful factor is 0.706667.

The "wrong product" results from a y -value that does not satisfy

$$N = \gcd(y^a + 1, N) \times \gcd(y^a - 1, N).$$

We have chosen to define the wrong product as the cases only where the product is not equal to 1 or N . By nature the wrong product will not be a problem since it must be the product of two numbers that divide N . If at least one of those two numbers is not equal to one then it is a factor of N .

For additional examples, several numbers were chosen and appear in table (B.1). The focus here is more on finding the probability that a single pass through Shor's algorithm will produce a useful answer. The numbers were chosen to be diverse, including a perfect square and a higher power, two numbers that are close (one factor in common and the other having close to the same value), a prime and small and large composites. The table shows for the given numbers the frequency at which problems (product equals one or factors are trivial) are seen.

The largest of these numbers, $N = 229 \times 283 = 64807$, was chosen as a "larger number", since although still much smaller than the numbers

APPENDIX B. NUMERICALLY EXPLORING SHOR'S ALGORITHM63

Table B.1: Results of `trial.cpp` on selected numbers. The final column, "Prob. Success", gives the probability that given a random y , $1 < y < N$ the algorithm will produce at least one useful factor.

N	Factors	Ones	Trivial Result	Wrong	Prob. Success
21	3×7	0	4	0	0.789474
25	5×5	1	17	1	0.217391
77	7×11	5	17	2	0.706667
223	prime	96	125	0	0
1521	$3^2 \times 13^2$	28	122	77	0.789474
16807	7^5	5022	7315	2064	0.265873
52003	$7 \times 17 \times 19 \times 23$	149	4	0	0.789474
63109	223×283	15139	15776	260	0.510118
64807	229×283	7779	8102	130	0.754942

used in RSA cryptography, it is near the maximal value that will not cause `trial.cpp` to overflow the C++ data type `unsigned long` on our system (in this case a 32-bit integer). In general, $\overline{y^a}$ could be close to N , which, when multiplied by N to calculate $\overline{y^{a+1}}$ could overflow the register if we do not guarantee that $N^2 < 2^{\text{sizeof}(\text{unsigned long})}$.

The columns of table B.1 were chosen to give a fair image of what results can be expected. The final column contains the probability that any random y , $1 < y < N$ will find a factor (including y 's that are not co-prime to N). This shows that for our limited selection the probability of finding a factor of a number that is not a pure power is always better than 1/2. This problem is addressed further in [EJ96].

Table B.1 shows that this method works best for composite numbers, and absolutely does not work for powers of a single prime (the only y 's that are counted as a useful factor for $N = 25$ and $N = 7^5$ are those that are not coprime to N or produce a "wrong factor", which is always a lower power of 5 or 7 respectively). Similarly, for $N = 3^2 13^2$ the squares were never split, the program always returning 9 and 169 as factors. As can be seen by 63109 and 64807 even numbers that are close to one another (both in value and factors) do not necessarily fare equally as well in this method. For all of the numbers tried, the probability of choosing a y randomly that will not find at least one useful factor is greater than twenty percent.

Appendix C

Two other topics from number theory

As stated in chapter 1, Shor's algorithm depends on several results from number theory. In addition to the core of number theory that allows us to create the periodic function \bar{y}^a two well known algorithms from number theory are needed, and will be discussed briefly here, the Euclidean algorithm, which finds the greatest common denominator of two numbers and continued fractions for finding fractional approximations. Proofs and more thorough discussion of these methods can be found in any basic text on number theory (see for example [HW68]).

C.1 The Euclidean algorithm for finding the gcd

The Euclidean algorithm is an efficient, well-known classical algorithm that is often used in number theory and ring theory. Since the Euclidean algorithm will be a major element of Shor's algorithm we will examine it here. The function `GCD()` in `numbers.h` in our program is based on this algorithm.

Given two numbers, p_0 and q_0 , where $p_0 > q_0$, if we let

$$\begin{aligned} q_1 &= p_0 \bmod q_0, & p_1 &= q_0, \\ q_2 &= p_1 \bmod q_1, & p_2 &= q_1, \\ q_3 &= p_2 \bmod q_2, & p_3 &= q_2, \\ & \vdots & & \vdots \\ 0 &= p_n \bmod q_n, \end{aligned} \tag{C.1}$$

where $q_1, q_2, \dots, q_n > 0$, then $\gcd(p_0, q_0) = q_n$.

Example: Find $\gcd(3315, 247)$:

$$\begin{aligned} 104 &= 3315 \bmod 247, \\ 39 &= 247 \bmod 104, \\ 26 &= 104 \bmod 39, \\ 13 &= 39 \bmod 26, \\ 0 &= 26 \bmod 13. \end{aligned}$$

The algorithm says that $\gcd(3315, 247) = 13$, which is true, since $3315 = 3 \times 5 \times 13 \times 17$ and $247 = 13 \times 19$.

C.2 Finding fractional approximations using a continued fraction

A continued fraction is an efficient algorithm that finds a fraction that either approximates or is equal to a decimal or another fraction, within a specified bound. The resultant fraction will be in lowest terms. This brief introduction is intended as a starting point. As with the greatest common denominator, a basic text on number theory can provide a proof and more details.

The continued fraction:

$$\frac{c'}{r} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

can easily be reduced to c'/r . Given a decimal (or fraction) d , a_0 is the greatest integer in the decimal, d (hereafter written $[d]$). We will denote the fractional remainder by f_0 . Each successive a_k is calculated by taking the greatest integer of one divided by the remainder of the previous calculation, so $a_k = 1/(f_{k-1})$. We then want to determine numbers p and q so that at each step, k , the value of the continued fraction can be found by

$$\frac{c'}{r} \approx \frac{p_k}{q_k}. \quad (\text{C.2})$$

The numbers p_k and q_k are calculated as follows:

$$\begin{aligned} p_0 &= a_0 & \text{and} & & q_0 &= 1, \\ p_1 &= a_1 \cdot a_0, & q_1 &= a_1, \\ & & & & & \vdots \\ p_k &= a_k \cdot p_{k-1} + p_{k-2}, & q_k &= a_k \cdot q_{k-1} + q_{k-2}. \end{aligned} \quad (\text{C.3})$$

The continued fraction can be used to find good fractional approximations of any decimal. There is a traditional bound that usually accompanies a discussion of the continued fraction, but since this is really just the tolerance of the approximation it is somewhat arbitrary. Shor's algorithm uses a bound based on q , so in my function (`contfrac()` found in `numbers.h`) I allow the desired bound to be input.

The example below shows the information output by version of `contfrac()` that was modified to display the continued fraction as it progressed. The same basic code used in my simulation of Shor's algorithm is used to determine a fractional approximation of π .

Select input style (anything else to quit):

1. Fraction
2. Decimal

Please input a decimal: 3.14159265359

Set limit (ie. for $1/(2*N^2)$ input $2*N^2$): 100000000

Constructing continued fraction:

$a_0 =$	3	$p =$	3	$q =$	1	$p/q =$	3
$a_1 =$	7	$p =$	22	$q =$	7	$p/q =$	3.14285707473755

a_ 2= 15	p= 333	q= 106	p/q= 3.14150953292847
a_ 3= 1	p= 355	q= 113	p/q= 3.14159297943115
a_ 4= 292	p= 103993	q= 33102	p/q= 3.14159274101257

As this example shows, this relatively simple algorithm can efficiently find approximations to as high a precision as desired.

Bibliography

- [AU95] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science, C Edition*. New York: Computer Science Press, 1995.
- [AGR81] A. Aspect, P. Grangier and G. Roger. Experimental tests of realistic local theories via Bell's theorem. *Physical Review Letters* 47 (1981): 460-463.
- [AGR82] A. Aspect, P. Grangier and G. Roger. Experimental realization of Einstein-Podolsky-Rosen-Bohm gedanken experiment: A new violation of Bell's inequalities. *Physical Review Letters* 49 (1982): 91-4.
- [ADR82] A. Aspect, J. Dalibard and G. Roger. Experimental tests of Bell's inequalities using time-varying analyzers. *Physical Review Letters* 47 (1981): 460-463.
- [Bra97] Gilles Brassard. Searching a quantum phone book. *Science* 275 (1997) :627-8.
- [Bre97] Alfred Brenner. Moore's Law. *Science* 275 (1997): 1551.
- [Col97] Graham P. Collins. Searching is less tiring with a bit of quantum magic. *Physics Today* Oct. 1997:19-21.
- [Deu85] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A* 400 (1985): 97-117.
- [DF91] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Englewood Cliffs, New Jersey: Prentice Hall, 1991, pp. 53-60
- [Eco97] Quantum computing: Cue the qubits. *Economist* 22 Feb. 1997:91-2.

- [EJ96] Artur Ekert and Richard Jozsa. Quantum computation and Shor's algorithm. *Reviews of Modern Physics* 68 (1996): 733-753.
- [Feyn82] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics* 21 (1982): 467-88.
- [FT82] E Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics* 21 (1982): 219-53.
- [Gar92] Alejandro L. Garcia. *Numerical Methods for Physics*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.
- [Gas96] Stephen Gasiorowicz. *Quantum Physics*, 2nd ed. New York: John Wiley and Sons, 1996.
- [GC97] Neil A. Gershenfeld and Isaac L. Chuang. Bulk spin-resonance quantum computation. *Science* 275 (1997): 350-6.
- [GZ97] George Greenstein and Arthur G. Zajonc. *The Quantum Challenge*. New York: Jones and Bartlett Publishers, 1997.
- [HR96] Serge Haroche, Jean-Michel Raimond. Quantum computing: Dream or nightmare. *Physics Today* Aug. 1996: 51-2.
- [HH96] D. Hutcheson and J. Hutcheson. Technology and economics in the semiconductor industry. *Scientific American* Jan. 1996: 54-56.
- [HW68] G. Hardy and E. Wright. *An Introduction to the Theory of Numbers*. New York: Oxford Press, 1968.
- [Joz98] Richard Jozsa. Quantum algorithms and the Fourier transform. *Proceedings of the Royal Society of London A* 445 (1998): 323-337.
- [Key92] Robert Keyes. The future of solid-state electronics. *Physics Today* 45 (1992): 42-8.
- [Lan91] R. Landauer. Information is Physical. *Physics Today* May 1991: 23-29.
- [Li96] Yao Li et. al. Optical computing: Introduction by feature editors. *Applied Optics* 10 Mar. 1996: 1177-9

- [Men97] amian Menscher. Modeling the quantum computer on the classical computer. Thesis, Brigham Young U., 1997.
- [Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*. 13: 300-317, 1976.
- [Mil96] Gerard J. Milburn. *Schrodinger's Machines*. New York: W. H. Freeman and Company, 1996.
- [Mil98] Gerard J. Milburn. *The Feynman Processor*. Reading, Massachusetts: Perseus Books, 1998.
- [MW95] Christopher Monroe, David Wineland, et. al. Demonstration of quantum logic gate. *Physical Review Letters* 18 Dec. 1995: 4714-17.
- [MW96] Christopher Monroe, David Wineland, et. al. Future of quantum computing proves to be debateable. *Physics Today* Nov. 96: 107-8.
- [Rab80] M. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12 (1980):128-38.
- [RSA 98] FAQ 4.0. RSA Laboratories. Available at www.rsa.com/rsalabs/faq/index.html, 1998.
- [Ser96] Service, Robert F. Can Chip Devices Keep Shrinking? *Science* 274 (1996): 1834-36.
- [Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. *Proceedings of the 35th IEEE Annual Symposium on the Foundations of Computer Science*. 35 (1994): 124-134.
- [Sho96] Peter W. Shor. Fault Tolerant quantum computation. *Proceedings of the 37th IEEE Annual Symposium on the Foundations of Computer Science*. 37 (1996): 56-65.
- [Ste96] Stein, Ben. It Takes Two to Tangle. *New Scientist* 28 Sept. 1996: 26-31.
- [Tau96] Gary Taubes. All together for quantum computing. *textitScience* 273 (1996): 1164.

- [Tur37] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42 (1937): 544-6.
- [WC97] Colin P. Williams. and Scott H. Clearwater. *Explorations in Quantum Computing*. Santa Clara: Springer-Telos, 1997.