DYNAMICAL STABILITY OF QUANTUM ALGORITHMS

by

MARK MCGREW MIMS, B.S. Phys., B.S. Math.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2000

DYNAMICAL STABILITY OF QUANTUM ALGORITHMS

APPROVED BY
DISSERTATION COMMITTEE:

Supervisor: _____

_____

_____

_____

_____

In memory of my uncle, Warren J. “Whip” Mermilliod, III.

His influence upon my life, albeit brief, was lasting.

Acknowledgments

I'd like to thank Professor Sudarshan, not only for his encouragement and diligent tutelage, but for his infinite patience during my brief flirtation with "applied capitalism" in the midst of my education. Many thanks go to Professor Cécile DeWitt for taking me under her wing as an undergraduate, and for continuing guidance and support over the years. Thanks to my family and friends for their generous support and patience throughout. Many people have contributed in one way or another to this work: Mark Byrd, Chris Howe, Aikmeng Kuah, Konstatin Mardanov, Usman Roshan, Anil Shaji, Todd Tilma, and Tom Yudichack. Helpful editors include Scott Glover and Dustin McCartney. For keeping things on the lighter side, thanks to Ali and "the little algorithm that could," and finally, thanks to Cheryl and Lecia for the occaisional beer.

DYNAMICAL STABILITY OF QUANTUM ALGORITHMS

Publication No. _____

Mark McGrew Mims, Ph.D.
The University of Texas at Austin, 2000

Supervisor: E.C.G. Sudarshan

Grover's quantum search algorithm is modelled as a geodesic of the Fubini–Study metric on the space of pure quantum states. This model is extended to encompass mixed states, and then generalized noise is added. A numerical simulation shows that the noise introduced does not slow the algorithm down, but only reduces the maximum probability of successfully finding the searched for item. The maximum noise that the bare algorithm can tolerate is shown to reduce as the size of the database increases. With a database of size N, this maximum noise scales like N^{-3}.

Table of Contents

List of Figures

Chapter 1

Introduction

Quantum computation is an exciting new field of research for a number of reasons[37]:

- Quantum computers can solve some hard problems.
- Quantum errors can be corrected.
- Quantum hardware can be constructed.

The field lies on the boundary between Physics, Mathematics, and Computer Science. It is developing at a rapid pace because theoretical and experimental researchers are working closely together in quite a unified effort.

The age of single photon and single atom experiments is upon us. The experimental techniques for controlling quantum computations either already exist or are expected to exist in the near future. Researchers on the theoretical side are contributing things like quantum error correcting codes that can be used to increase the quantum computer's tolerance for noise, as well as the algorithms used to solve various problems on a quantum computer.

What is it all about? Why all the hype? To see this, it is useful to look at some of the differences between a classical and quantum computer... to examine some of the limitations of the classical computers in use today.

Cosider the representation and evaluation of a function on a classical computer. Take, for instance, some function $f\colon \mathbb{Z} \to \mathbb{Z}$ where

$$x \mapsto f(x). \tag{1.1}$$

Now, an integer is represented on a classical computer, in base-two, by a finite set of binary digits or *bits*. If the computer represents integers by say n bits,

$$\begin{aligned} x =& \underbrace{0110010\ldots0}_{n \text{ bits}} \\ =& b_{n-1}b_{n-2}\ldots b_1 b_0 \\ =& \quad \text{(in base 10)} \quad \sum_{i=0}^{n-1} b_i 2^i, \end{aligned} \tag{1.2}$$

then only 2^n different integers can be represented. The function $f\colon \mathbb{Z} \to \mathbb{Z}$ really becomes $f\colon \mathbb{Z}_{2^n} \to \mathbb{Z}_{2^n}$.

Next, consider some very basic questions about the function f. Is it constant? Is it monotonic? Is it periodic? If so, find the period. How can these questions be answered on a classical computer? Well, the only thing to do is evaluate the function for *all* inputs. If the function is constant for all inputs, it is constant. In general, this is expensive to compute... prohibitively so. One might truthfully say that this sort of menial task is exactly what computers excel at and so, who cares? What's the big deal? Well, consider the case of determining the periodicity of a function. This is used, for instance, in certain factoring algorithms that allow a classical computer to factor a large

integer into it's component prime factors.[1] Again, in order to tackle this problem, the function must be evaluated for *all* inputs. It is the difficulty of this very task that lies behind the security of most cryptosystems in use today. These methods rely upon the fact that it takes a *long* time to determine the periodicity of a function and hence, the factors of a large integer.

The question to now ask is... can we do better? The answer is a resounding yes! A *quantum* computer can answer some of these questions with considerably less work. This is what has caused all of the hype. They are not better, or even good, at everything. However, it has been shown that if they could be built, quantum computers can solve certain problems *much* faster that their classical cousins.

In order to see what quantum computers are all about, first consider the representation of, say, integers on a quantum computer. The *bits* of a classical computer have a quantum mechanical analogue, the quantum bit or *qubit*. A quantum bit is some two–state system. Exactly what kind of two–state system is not specified here. There are many different implementations of qubits currently being investigated in laboratories around the world, and it is not clear which physical implementation will work best. It should be noted that a major issue in determining which physical system to use as a quantum computer is the noise inherent to that system. Research such as this dissertation is exactly the type of work necessary to determine the tolerances that the physical implementations must be held to, and ultimately, to determine which physical systems will be workable in an actual quantum computer. Thinking of a qubit

[1] This is covered in detail in chapter 3.

as a simple spin–$\frac{1}{2}$ system is probably best for now. A bit is represented by either spin up or down

$$\begin{aligned} 0 \quad &\mapsto \quad |0\rangle = |\downarrow\rangle \\ 1 \quad &\mapsto \quad |1\rangle = |\uparrow\rangle \end{aligned} \tag{1.3}$$

An integer represented on a classical computer by a set of n bits

$$x = (01100011\ldots0) \tag{1.4}$$

can be represented as a set of n qubits

$$\begin{aligned} |x\rangle =& |01100011\ldots0\rangle \\ =& \bigotimes_{i=0}^{n-1} |b_i\rangle. \end{aligned} \tag{1.5}$$

Now, consider the representation and evaluation of functions such as (1.1) on a *quantum* computer. If x becomes $|x\rangle$, and then the value of the function is just another integer represented on the computer by $|f(x)\rangle$. How can $f(x)$ be evaluated? How can the state $|f(x)\rangle$ be obtained from the state $|x\rangle$? Well, closed quantum systems evolve by unitary transformations, so an operator such as

$$\mathbf{U_f}|x\rangle = |f(x)\rangle \tag{1.6}$$

just might be a good start. Actually, the unitarity of the above operator is dependent upon properties of the function f. A better approach is to take the operator

$$\mathbf{U_f}|x\rangle|0\rangle = |x\rangle|f(x)\rangle, \tag{1.7}$$

or really

$$\mathbf{U_f}|x\rangle|a\rangle = |x\rangle|f(x) \oplus a\rangle \qquad (\text{any } a), \tag{1.8}$$

which is unitary no matter the function used. Exactly how such unitary operators can be constructed is covered in detail later in chapter 2.

Now, given such a unitary operator, the function f can be evaluated for some input $|x\rangle$ on a quantum computer. We can evaluate a function for some input. Big deal. Is this all just a fancier way of writing the same classical problem? The same constraints seemingly apply. *i.e.*, determining if the function is constant still seems to require evaluations of the function for all inputs. Well... no.

The classical evaluation of a function relies upon gates that can act on a bit with a definite value, 0 or 1. The quantum function evaluation is accomplished by unitary transformations that can act, not only on states such as $|0\rangle$ or $|1\rangle$, but on superpositions of states such as $a|0\rangle + b|1\rangle$. This is the crucial difference – that quantum computations can act on superpositions of states. This fact allows the quantum computer to take advantage of a sort of pseudo–parallelism to beat out the classical computer in solving certain types of problems.

Consider the example of finding the period of a periodic function f.[2] Instead of taking

$$\mathbf{U_f}|x\rangle|0\rangle = |x\rangle|f(x)\rangle \tag{1.9}$$

for all inputs $|x\rangle$, consider acting on the state

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle|0\rangle \tag{1.10}$$

[2]Periodic here refers to some periodicity besides the obvious one due to the computer's finite representation of integers as elements of the ring $\mathbb{Z}_{2^n}$.

which is in an equally weighted superposition of all input states $|x\rangle$. The function evaluation operator

$$\mathbf{U_f}\left[\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|0\rangle\right] = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|f(x)\rangle \tag{1.11}$$

evaluates the function simultaneously for *all* inputs! This is a powerful idea. The resulting state contains all information about the function. Of course, we don't necessarily have *access* to all of this information, but the right measurements can allow certain questions to be answered. The rest of the "algorithm" to find the period consists of a measurement of the second ket

$$\left(\mathbf{1}\otimes|c\rangle\langle c|\right)\sum_{x=0}^{2^n-1}|x\rangle|f(x)\rangle = \sum_{\{x|f(x)=c\}}|x\rangle|c\rangle \tag{1.12}$$

which collapses the superposition of all $|x\rangle$ into a superposition of just those where $f(x) = c$. The periodicity of the function f is still contained within this superposition and can be extracted via a Fourier transform. This entire process takes *considerably* less time than the classical version.

This Dissertation

There are several quantum algorithms to solve various problems on a quantum computer that exhibit various degrees of speedup over their classical counterparts. The quantum search algorithm is one such algorithm that is interesting for several reasons: it can easily be adapted to solve a host of other problems, it exhibits a speedup over a classical search algorithm, it is particularly simple to describe, and it can be described dynamically. This is often the first algorithm that experimentalists attempt to implement, and so it is a natural candidate for a robustness analysis.

My current research concentrates on a dynamical model of this quantum search algorithm. The hope is that, in future work, a dynamical approach can be taken for quantum alorithms in general. The vast arsenal of tools from the discipline of dynamical systems could then be brought to bear upon the analysis of the stability of various quantum algorithms.

For this dissertation, a faithful numerical model of the search algorithm is developed. Noise is then added in a generic way that will encompass any source and type of physical noise. The goal is to determine the robustness of the bare algorithm in the presence of various types of noise.

This dissertation is organized as follows: Chapters 2 and 3 contain background information on quantum computation and quantum algorithms. Chapter 4 describes the Fubini–Study metric and relates this to a statistical distance in Hilbert space. Chapter 5 applies the Fubini–Study metric to a dynamical model of the quantum search algorithm. It introduces the necessary techniques for numerical stability analysis that are then applied to the model of the algorithm in the presence of noise. Results are presented in chapter 6, with data included in appendix A. Appendix B discusses details of the numerical model and the tools used in this analysis.

Chapter 2

Quantum Computation

In the early 80's, Yuri Manin[32] and Richard Feynman[21] examined the difficulty of simulating a quantum mechanical system. Feynman argued that, in order to effectively simulate a physical quantum system, a conventional "classical" computer would require an unreasonably large amount of overhead. So no matter how clever the programmer, this task would almost always require enormous amounts of computer time and storage space. Feynman also mentioned that this difficulty might be overcome by using an inherently quantum mechanical computer. Benioff[2] had already shown how to, at least in theory, use quantum mechanical processes as a basis for conventional computation, and Feynman[22] later refined this approach.

If a computer that is quantum mechanical in nature were to be built, it would be able to handle the complexity required for the task of simulating a quantum system. Could this same computer also be used to solve other complex problems? Could this *quantum* computer be put to work to solve *classical* problems? The answer, remarkably, is yes! This approach resulted in several people[14, 15, 16, 4, 40, 38, 28] describing *quantum algorithms*, or programs that will run on a quantum computer to solve various types of problems. In

the process, it was discovered that some problems that are extremely hard to solve on a classical computer were easily solved on a quantum computer.

Perhaps the most notable of these quantum algorithms is the one designed by Peter Shor[38] in 1994. This now famous algorithm quickly finds the prime factors of a composite integer, a daunting task for even the most powerful of classical computers.

Computer scientists use a system of complexity classes to classify the difficulty of various problems. It turns out that Shor's algorithm reduces the problem of factorization to a completely different complexity class. This problem can be solved in polynomial time on a quantum computer. *i.e.*, the time required to solve the problem scales as a polynomial in the size of the input (the number to be factored). The fastest algorithm to solve the problem on a classical computerhas a running time that is exponential in the size of the input.

Consider finding the 65–digit factors of a 130–digit number. Today, this can be done in about a month using a network of hundreds of computers. From this result, we can estimate that it will take about 10^{10} years to factor a 400–digit number using today's technology. In fact, most popular cryptosystems today, for instance those used to protect online credit card transactions, nuclear missle secrets, etc., rely upon the fact that this particular problem is so intractably difficult to solve. Well, enter the *quantum* computer.

Once a suitably large quantum computer is built, Shor's algorithm could conceivably factor this 400–digit number in the order of years – maybe less. The complexity of the problem has been considerably reduced. Need-

less to say, this has generated a considerable amount of interest in quantum computation.

Another major quantum algorithm is the search algorithm. Lov Grover[28] developed a version of the common search algorithm that will run on a quantum computer. The fastest known classical algorithms that will pick a single item out of a large unordered list run in a time on the order of the size of the list. While Shor's algorithm enjoys an exponential speedup over any classical solutions to factorization, Grover's algorithm only has a square root speedup over classical search algorithms. It will pick an item from an unordered list in about the square root of the size of the list. Nevertheless, for a large list, this is a substantial speedup.

Grover's algorithm is interesting for several reasons. The general search algorithm is most amenable to being transformed into a solution to a different problem. Various other problems can be transformed into search problems and hence enjoy a speedup when run on a on a quantum computer[18, 26, 27, 7, 44, 8, 12, 20]. Also, Grover's algorithm is perhaps the easiest algorithm to model and simulate. Since the largest physical quantum computer built is only a few bits, we are still at a stage where numerical and analytical simulation is very important to advancement of the field. In fact, this is strong motivation for this dissertation. In developing and analyzing dynamical models of Grover's algorithm, can we learn more about how to develop similar models for general quantum algorithms? Grover's algorithm is simple enough that, hopefully, it can lead the way.

Before pressing further into the exciting world of quantum computation,

it will be useful to review some notions of classical computation. There are several excellent overviews of quantum computation in the literature [37, 39, 9, 41, 31, 17, 36] and I will not attempt to reproduce them here. Instead, the following sections are streamlined to include only what is necessary for the remaining work. They are taken, along with parts of chapter 3, shamelessly from the likes of Preskill[37], Shor[39], Braunstein[9], and Steane[41].

The reader is encouraged to pursue the subject of quantum computation further as there are many interesting areas that, unfortunately, would be a bit far afield for the current work. In particular, I avoid such topics as: quantum information theory, entanglement, optimal measurements, teleportation, quantum cloning, quantum cryptography (key distribution), and the issue of actually implementing a quantum computer. Additionally, I touch only briefly upon the extremely rich and interesting subject of quantum coding and error correction.

2.1 The Theory of Computation

2.1.1 Classical Theory of Computation

A classical computer can be viewed simply as a system that evaluates a function of the following form

$$f\colon \{0,1\}^n \mapsto \{0,1\}^m. \tag{2.1}$$

This "computer" takes n bits of input and uniquely determines m bits of output. Since each of m components of this function is just a simpler function to a single bit, the basic operation of a classical computer is just

$$f\colon \{0,1\}^n \mapsto \{0,1\}. \tag{2.2}$$

These simple functions with one–bit outputs are commonly referred to as *decision problems*. They encode the solution to a problem whose answer, based on the input, requires only a YES or NO. There are many such decision problems that can be useful, but it is important to note that there are also many apparently non–decision–type problems that can be rephrased in terms of decision problems.

To be evaluated on an actual computing device, each of the basic functions must then be broken down into a sequence of elementary logic operations. These elementary operations or *logic gates* can then be implemented on various types of physical devices.

Consider a simple function

$$f\colon \{0,1\}^n \mapsto \{0,1\}. \tag{2.3}$$

The claim is that the evaluation of this function can be reduced to a sequence of elementary logical operations. This can be seen by first considering the set of inputs to the function. Among the 2^n distinct inputs, distinguish between those inputs x for which the function evaluates to $f(x) = 1$ and those for which $f(x) = 0$. For each element $x^{(i)} \in f^{-1}(1)$, define a *characteristic function* for this input as

$$f^{(i)}(x) = \begin{cases} 1 & x = x^{(i)} \\ 0 & \text{otherwise} \end{cases}. \tag{2.4}$$

This is the first step to a representation of the original function in terms of simpler gates, leaving an intermediate representation of the function in terms of characteristic functions

$$f(x) = \bigvee_i f^{(i)}(x). \tag{2.5}$$

The function f is the logical OR ($\vee$) of the characteristic functions for all inputs that evaluate to 1. *i.e.*, all $x \in f^{-1}(1)$. The way to interpret this representation is that $f(x)$ can evaluate to 1 by either $x^{(1)}$ evaluating to 1 or $x^{(2)}$ evaluating to 1 or...

The next step is to consider how the characteristic functions themselves can be represented in terms of elementary logic gates. The characteristic function for the input

$$x^{(i)} = 111\ldots1 \tag{2.6}$$

could be constructed as

$$f^{(i)} = x_1 \wedge x_2 \wedge \ldots \wedge x_n, \tag{2.7}$$

the logical AND of all n bits of the input $x^{(i)}$. Each $x_j \in \{0, 1\}$ is the j–th bit of $x^{(i)}$. For this particular input (2.6), all x_j's are 1. Any other input $x^{(i')}$ will have at least one zero bit. The characteristic function for this input is again the logical AND of all bits, but the zero bits are flipped with the logical NOT ($\neg$) operation. For example, consider the input

$$x^{(i')} = 0110\ldots. \tag{2.8}$$

The corresponding characteristic function would be

$$f^{(i')}(x) = (\neg x_1) \wedge x_2 \wedge x_3 \wedge (\neg x_4) \wedge \ldots. \tag{2.9}$$

The function f can now be represented in terms of three elementary logical operations AND, OR, and NOT. This representation of the function is called, in the computer science literature, "disjunctive normal form." A

small number of elementary gates suffice to evaluate exactly any function of a finite input. This is an important result in computer science and allows for arbitrarily complex computations to be built up out of very simple physical devices.

Circuits and Complexity

Computers have a few basic components that can perform elementary operations on bits or pairs of bits. A *computation* is a finite sequence of such operations (a *circuit*) applied to a specified string of input bits. The complexity of a computation can be represented by the complexity of the corresponding circuit. For a decision problem, a reasonable measure of the difficulty of the problem is the *size* of the smallest circuit that computes the corresponding function

$$f\colon \{0,1\}^n \mapsto \{0,1\}. \tag{2.10}$$

Here, size is measured by the number of elementary operations (gates) that make up the circuit used to compute f. If more that one gate is allowed to act at once, then the *time* it takes to complete the computation can be a measure of its difficulty. In this case, the *depth* of the circuit is the maximum number of time steps required to complete, or the maximum length of a directed path from the input to the output of the circuit. The *width* of a circuit is the maximum number of gates that can act at any given timestep.

Note that the size of a circuit discussed above will, in general, scale with the number of bits, the size, of the input. The complexity of the circuit can then be determined by how the size of the circuit will scale with the size of the input. Consider a circuit C_f that evaluates a decision problem corresponding

to the function f. The input to this particular circuit has n bits. This circuit is considered to be of *polynomial size* if the size of the circuit grows no faster than some polynomial of n, or

$$\text{Size}\,(C_f) \leq \text{Poly}(n). \tag{2.11}$$

Decision problems with circuits of polynomial size are considered to be "easy," otherwise, they are referred to as "hard." A *complexity class* of these problems can be defined as (**P** for Polynomial)

$$\mathbf{P} = \{\text{decision problems with polynomial–sized circuits}\}. \tag{2.12}$$

Decision problems in **P** are easy, otherwise, they are hard.

2.1.2 Quantum Theory of Computation

The circuit model of classical computation will need to be generalized for a quantum computer. The universal gates in these quantum circuits will need to be implemented as unitary tranformations of quantum states, and hence will need to be reversible.

Reversibility

A reversible computation will evaluate an invertible function such as

$$f\colon \{0,1\}^n \to \{0,1\}^n. \tag{2.13}$$

This computation will need to be able to run in either direction, obtaining output from input and vice–versa, and so the function will need to be bijective. Bijective maps of $\mathbb{Z}_2^n$ to itself can be represented by permutations of the set.

Note that a non–invertible function

$$f\colon \{0,1\}^n \to \{0,1\}^m \tag{2.14}$$

can be made into an invertible function

$$\tilde{f}\colon \{0,1\}^{n+m} \to \{0,1\}^{n+m} \tag{2.15}$$

such that

$$\tilde{f}\colon \begin{bmatrix} \left.\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{matrix}\right\} \text{n} \\ \left.\begin{matrix} 0 \\ \vdots \\ 0 \end{matrix}\right\} \text{m} \end{bmatrix} \mapsto \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix}, \tag{2.16}$$

and hence into a reversible computation.

Now, how can this reversible function be made up from basic gates? Are there *reversible* universal gates? Classical logic gates are, in general, not reversible. The trivial gate, which takes an input directly to an output, as well as the `NOT` gate are obviously reversible. `AND` and `OR` are not. However, there exist non–trivial reversible gates. The next simplest is the conrolled–`NOT` (`CNOT`) or the exclusive–`OR` (`XOR`) gate

$$\texttt{XOR}\colon (x,y) \mapsto (x, x \oplus y). \tag{2.17}$$

This gate flips the second bit when the first is set as shown in figure (2.1) and table (2.1). Notice that this could be used as a copy operation if the second bit (y) is initially set to zero (figure (2.2))

$$\texttt{XOR}\colon (x,0) \mapsto (x,x). \tag{2.18}$$

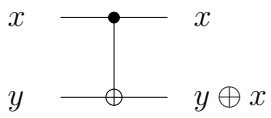

Figure 2.1: A controlled–NOT (CNOT) or exclusive–OR (XOR) gate.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Truth table for an exclusive–OR (XOR) gate.

Perhaps the most useful reversible gate is the controlled–controlled–NOT (CCNOT) gate

$$\texttt{CCNOT}\colon (x, y, z) \mapsto (x, y, z \oplus xy), \tag{2.19}$$

which flips the third bit only if the first two are 1 and does nothing otherwise.

The CCNOT gate is also known as the *Toffoli gate*, and, unlike the reversible two–bit gates, is itself a universal gate for Boolean logic. A Boolean circuit can be built to compute any reversible computation using Toffoli gates alone, provided input bits can be fixed and output bits can be ignored. For more on this, the reader is referred to the literature. Preskill[37], for example, addresses this in more detail.

Note that a reversible circuit can simulate a circuit composed of irreversible gates. This can be done efficiently without many additional memory resources (bits) and without appreciable slowdown. Since this can be done so

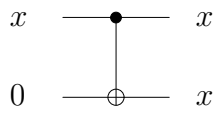

Figure 2.2: A controlled–NOT (CNOT) gate can be used to perform a COPY.

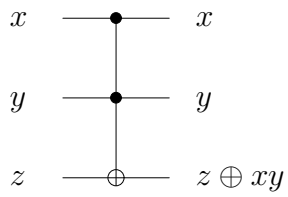

Figure 2.3: A controlled–controlled–NOT (CCNOT) gate

easily, the complexity of a reversible circuit will essentially be the same as for the irreversible circuit.

Quantum Circuits

A classical computer has "bits," while a quantum computer will have *quantum bits*, or "qubits." A quantum bit will be the state of a two–state quantum system. Say, for instance for some spin system,

$$\begin{aligned} |1\rangle &= |\uparrow\rangle \\ |0\rangle &= |\downarrow\rangle, \end{aligned} \tag{2.20}$$

More will be discussed about the actual physical implementation of these qubits later.

Sets of bits, will be tensor product states of qubits.

$$|0110\ldots\rangle = |0\rangle \otimes |1\rangle \otimes |1\rangle \otimes |0\rangle \otimes \ldots. \tag{2.21}$$

It is assumed, at any time during a computation, that the Hilbert space for

the entire quantum computer will factor into tensor products of smaller dimensional Hilbert spaces. For the most part, it is assumed these will be two–dimensional Hilbert spaces in order to more closely resemble classical *binary* computations. However, notice that nothing precludes working with *qutrits* or, in general, *qudits* (d–state systems) as the basic unit of computation.

A classical computer has circuits, or sets of gates, that act on sets of bits, or registers. Likewise, the quantum computer will have *quantum circuits*, which consist of sets of *quantum gates* which act on sets of qubits. Each quantum gate will consist of a unitary transformation that acts on a fixed number of qubits.

Each of the reversible classical gates described above have quantum analogues. In particular, the quantum version of the Toffoli gate, called the *Deutsch gate* (shown in figure (2.4)) will conditionally apply a rotation $\mathbf{R}$ to

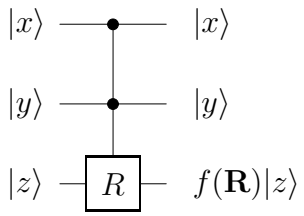

Figure 2.4: The Deutsch gate is the quantum analogue of the Toffoli gate. Like it's classical counterpart, it too is a universal gate.

the third qubit by

$$f(R) = \begin{cases} R, & xy = 1 \\ 1, & \text{otherwise.} \end{cases} \tag{2.22}$$

The rotation applied

$$\begin{aligned} R = R_x(\theta) &= \exp\left(i\frac{\theta}{2}\sigma_x\right) \\ &= \left(\cos\frac{\theta}{2} + i\sigma_x \sin\frac{\theta}{2}\right) \end{aligned} \tag{2.23}$$

is a rotation about the x axis by some angle θ. This gate is a universal gate for quantum circuits.

A particularly useful two–qubit quantum gate is a controlled–**U** gate (figure (2.5)). If the first qubit has value 1, this gate will apply the transfor-

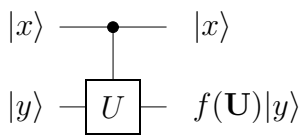

Figure 2.5: The controlled–**U** gate.

mation **U** to the second qubit, otherwise, no action is taken. This quantum gate will be used in almost all quantum algorithms in one form or another.

A computation on a quantum computer will start with a register of quantum bits in some initial configuration. The circuit of quantum gates will act on these qubits to perform whatever computations are necessary. Finally, a Von Neumann measurement is performed on the register of qubits, projecting each onto some measurement basis $\{|0\rangle, |1\rangle\}$. The outcome of this measurement is the result of the computation.

Before more is discussed about quantum algorithms and the actual computations involved, there are still a few more basic tools to be developed.

Practically every computation involves the use of at least one Hadamard transformation.

The Hadamard Transformation

The one–dimensional Hadamard transformation takes a quantum bit into the superposition given by

$$\begin{aligned} |0\rangle &\mapsto \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \\ |1\rangle &\mapsto \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right). \end{aligned} \tag{2.24}$$

This transformation has a matrix realization given by

$$H_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.25}$$

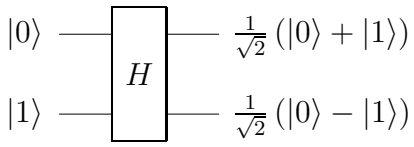

Figure 2.6: A quantum circuit applying a Hadamard transformation to two qubits.

The version of the Hadamard transformation that sees the most use is the N–dimensional Hadamard transformation defined as the N–fold tensor product of one–dimensional Hadamard transformations

$$H = H_N = \bigotimes^{N} H_1. \tag{2.26}$$

This is often used as the first step in many quantum algorithms. First, start with the *zero register*

$$|\mathbf{0}\rangle = |0\ldots0\rangle = \bigotimes^{N} |0\rangle, \tag{2.27}$$

and then act

$$\begin{aligned} H|\mathbf{0}\rangle =& H_N|0\ldots0\rangle \\ =& \bigotimes^{N} H_1|0\rangle \\ =& \bigotimes^{N} \frac{1}{\sqrt{2}}\left(|0\rangle+|1\rangle\right) \\ =& \frac{1}{\sqrt{2^N}}\Big\{|0\ldots0\rangle+|0\ldots01\rangle+\quad\cdots\quad+|1\ldots1\rangle\Big\}. \end{aligned} \tag{2.28}$$

Now, each of the kets on the bottom–RHS of equation (2.28) can be thought of as just a number expressed in binary. *i.e.*,

$$\begin{aligned} H|\mathbf{0}\rangle =& \frac{1}{\sqrt{2^N}}\Big\{\underbrace{|0\ldots0\rangle}_{0}+\underbrace{|0\ldots01\rangle}_{1}+\cdots+\underbrace{|1\ldots1\rangle}_{2^N-1}\Big\} \\ =& \frac{1}{\sqrt{2^N}}\sum_{x=0}^{2^N-1}|x\rangle. \end{aligned} \tag{2.29}$$

This will be useful in function evaluation. *i.e.*, Consider a function such as $f\colon \{0,1\}^N \to \{0,1\}$. Since the following are equivalent

$$\begin{gathered} (0,0,0,\ldots,1,0) \mapsto f(0,0,0,\ldots,1,0) \\ \underbrace{(000\ldots10)}_{2} \mapsto f\underbrace{(000\ldots10)}_{2} \\ (2) \mapsto f(2), \end{gathered} \tag{2.30}$$

one can simply think of this function as $x \mapsto f(x)$. So, as a first step, the Hadamard transformation acts on a quantum zero register, preparing it in a superposition of all possible values of x

$$H|\mathbf{0}\rangle = \frac{1}{\sqrt{2^N}}\sum_{x=0}^{2^N-1}|x\rangle. \tag{2.31}$$

A Hadamard transformation can also act on a *general* state $|x\rangle =$

$|x_1 \dots x_N\rangle$ (not necessarily all 0's), which can be written as

$$\begin{aligned} H\colon |x\rangle \quad &\mapsto \quad \bigotimes_{i=1}^{N} H_1 |x_i\rangle \\ &\mapsto \quad \bigotimes_{i=1}^{N} \frac{1}{\sqrt{2}} \left(|0\rangle + (-1)^{x_i}|1\rangle\right) \\ &\mapsto \quad \bigotimes_{i=1}^{N} \left(\frac{1}{\sqrt{2}} \sum_{y_i \in \{0,1\}} (-1)^{x_i y_i} |y_i\rangle \right) \\ &\mapsto \quad \frac{1}{\sqrt{2^N}} \sum_{y=0}^{2^N-1} (-1)^{x \cdot y} |y\rangle, \end{aligned} \tag{2.32}$$

where $x \cdot y$ denotes a dot product over $(\mathbb{Z}_2)^N$, or simply a binary dot product. Note that the final form of equation (2.32) is actually the Fourier transform of $|x\rangle$ over $(\mathbb{Z}_2)^N$.

Chapter 3

Quantum Algorithms

A quantum algorithm is a set of states and operations performed with a quantum computer in order to arrive at an answer to a particular problem. There are no simple prescriptions for constructing a quantum algorithm. In fact, there are no rules by which one can take an existing classical algorithm, an algorithm developed to run on a classical computer, and construct a corresponding quantum algorithm. There are not even many tools or concepts used in developing classical algorithms that carry over into the quantum domain.

As discussed in Chapter 2, the advent of universal quantum gates allows the representation of any computable function on a quantum computer. However, the uniquely quantum operations that must be performed require a wider range of tricks to be used when developing algorithms to be run on a quantum computer.

Despite the difficulty involved, there have been a few, seemingly inspired, algorithms developed for a quantum computer. In fact, the entire field essentially owes its existence to an algorithm developed by Peter Shor[38, 39] to solve FACTORIZATION[1], where the quantum version enjoys an *exponen-*

[1]The convention, originating in computer science, of YELLING the proper names of

tial speedup over the classical. The discovery of this algorithm has reduced the complexity class of this problem, a feat to which many in computer science aspire.

The following sections outline several aspects of quantum algorithms. A few existing algorithms are described, along with the basic framework needed to determine their computational complexity.

3.1 Simple Quantum Algorithms

3.1.1 Deutsch's Algorithm

Deutsch[14, 15] gives a simple example of a problem that can be solved faster on a quantum computer than its classical counterpart. Consider some black box that evaluates a function $f\colon \{0,1\} \to \{0,1\}$, where each evaluation, for whatever reason, takes a long period of time to compute, say 24 hours. The problem is to determine, as quickly as possible, information about the function using only inputs and outputs of the black box.

The obvious approach is to simply set an input value, then read the output value when the computation is done (24 hours later). Do this for all possible input values (0 or 1), then, 48 hours later, you have all possible information about the function.

What if that time is not good enough? Even if we are only interested in limited information about the function, such as is it *constant* ($f(0) = f(1)$) or *balanced* ($f(0) \neq f(1)$), it would still require full function evaluation (48 hours).

---

particular problems to be solved will be followed throughout this text.

Is there a quicker way? Yes, if there exists a quantum black box that computes $f(x)$, then the problem can be solved with a single function evaluation.

First, insure reversability of the computation, even if the quantum black box itself is not reversible, by constructing the unitary transformation

$$U_f\colon |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle, \tag{3.1}$$

where the second bit is flipped if $f(x) = 1$ and left alone if $f(x) = 0$.

Just as in the classical case, this computer can determine if the function is constant or balanced with multiple evaluations of $f(x)$. Still taking 48 hours to arrive at an answer.

Because the black box is a *quantum* computer, it can accept as an input a superposition of states $|0\rangle$ and $|1\rangle$. If the second qubit is initially prepared in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, then

$$\begin{aligned} U_f\colon |x\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad &\to \quad |x\rangle \otimes \frac{1}{\sqrt{2}}\big(|f(x)\rangle - |1 \oplus f(x)\rangle\big) \\ &= \quad |x\rangle \otimes (-1)^{f(x)} \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\,. \end{aligned} \tag{3.2}$$

Now, prepare the first qubit in the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The black box now acts

$$\begin{aligned} U_f\colon \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad &\to \\ &\frac{1}{\sqrt{2}}\big((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\big) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\,. \end{aligned} \tag{3.3}$$

Now, we simply perform a measurement that projects the first qubit into the basis

$$|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)\,, \tag{3.4}$$

to learn, after a single function evaluation, that the function is balanced if you measure $|+\rangle$, constant if $|-\rangle$.

So, here, the quantum computer gets an answer in around 24 hours, where the classical one would require 48. This speedup is attributed to the concept of *quantum parallelism*. The quantum computer is not limited to computing just a single $f(0)$ or $f(1)$ at a time, but in acting on superpositions of states, it behaves as if there were a sort of parallelism in execution. The computer essentially evaluates all inputs at the same time, storing the information in correlations between the states that make up the input and output registers. This information is only really available via specifically prepared measurements, but it is there. The output of the quantum computation encodes this "global" information about the function.

The quantum circuit to compute Deutsch's algorithm is shown in figure (3.1).

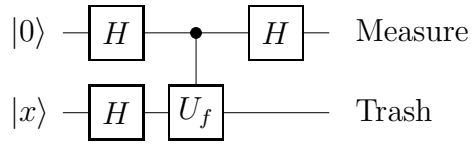

Figure 3.1: Deutsch's circuit

The Deutsch–Jozsa Algorithm

Of course, the simple function $\{0,1\} \to \{0,1\}$ discussed above is not particularly useful for anything other than providing a simple example with which to exhibit a quantum algorithm. However, the same principles apply to a more useful (section (2.1.1)) function $f\colon \{0,1\}^N \to \{0,1\}$. Again, construct the

unitary operator

$$U_f \colon |x\rangle|0\rangle \to |x\rangle|f(x)\rangle. \tag{3.5}$$

Then, via a Hadamard tranformation (*c.f.*, section (2.1.2)), choose the input register to be in the state

$$\left[\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)\right]^N = \frac{1}{\sqrt{2^N}} \sum_{x=0}^{2^N-1} |x\rangle. \tag{3.6}$$

A single function evaluation leaves

$$\frac{1}{\sqrt{2^N}} \sum_{x=0}^{2^N-1} |x\rangle|f(x)\rangle, \tag{3.7}$$

which encodes "global" information about the function. The trick is to now extract this information via specialized measurements.

Consider the classical version of the above computation. Depending on what sort of information we are trying to obtain about the function, there are *many*(!) function evaluations required. For $N \gg 1$, this quickly becomes intractable. The quantum case required only one. For obvious reasons, this is considered "massive quantum parallelism" in the literature.

3.1.2 Simon's Algorithm

In the last section it was shown how quantum algorithms can exhibit a speedup over algorithms designed to run on a classical computer. How much faster can certain problems be solved? Do the problems remain in the same complexity class (*c.f.*, chapter (2)), or like for Shor's algorithm, will the quantum algorithm change the actual complexity class of the problem?

Simon's algorithm[40] is the simplest algorithm where the problem to

be solved is classically *hard*, but can be computed in polynomial time on a quantum computer.

Consider a function

$$f\colon \{0,1\}^n \to \{0,1\}^n \tag{3.8}$$

that is 2–to–1 and periodic when viewed as a map $(\mathbb{Z}_2)^n \to (\mathbb{Z}_2)^n$ (as opposed to $\mathbb{Z}_{2^n} \to \mathbb{Z}_{2^n}$). *i.e.*, If the period of f is a, then

$$f(x) = f(x \oplus a). \tag{3.9}$$

The problem is: given a quantum black box that evaluates such a function, find its period.

According to Shor[39], this problem is classically hard because the function will need to be evaluated an exponentially large number of times to find a.

The quantum solution to the problem starts with two zero registers (as defined in section (2.1.2))

$$|0\rangle|0\rangle. \tag{3.10}$$

Now prepare an equally weighted superposition of all n-bit strings in the first register by using the Hadamard transformation

$$(H \otimes \mathbf{1})\,|0\rangle|0\rangle = \left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle\right)|0\rangle, \tag{3.11}$$

and then query the oracle

$$U_f\colon \quad \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle|0\rangle \quad \mapsto \quad \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle|f(x)\rangle. \tag{3.12}$$

Now, a measurement of the second register, let's say it is measured to be in the state $|c\rangle$, will collapse the first register into a superposition of all states $|x_0\rangle$ such that $f(x_0) = c$. For the function specified in this algorithm, there are two such values, x_0 and $x_0 \oplus a$. After the measurement of the second register, the first register then should look like

$$\frac{1}{\sqrt{2}}\big(|x_0\rangle + |x_0 \oplus a\rangle\big). \tag{3.13}$$

Note that, at this point, a measurement of this first register tells nothing about a. However, taking another Hadamard transformation (see equation 2.32) on the first register gives

$$H\frac{1}{\sqrt{2}}\big(|x_0\rangle + |x_0 \oplus a\rangle\big) = \frac{1}{\sqrt{2^{n+1}}}\sum_{y=0}^{2^n-1}\Big[(-1)^{x_0\cdot y} + (-1)^{(x_0\oplus a)\cdot y}\Big]|y\rangle. \tag{3.14}$$

This is non–zero only for $a\cdot y = 0$, and so the first register can then be rewritten as

$$H\frac{1}{\sqrt{2}}\big(|x_0\rangle + |x_0 \oplus a\rangle\big) = \frac{1}{\sqrt{2^{n-1}}}\sum_{a\cdot y=0}(-1)^{x_0\cdot y}|y\rangle, \tag{3.15}$$

which is just an equally weighted superposition of states $|y\rangle$ such that $a\cdot y = 0$. A measurement of the state in the first register will yield any one of these states with equal likelihood. Now, simply repeat the entire algorithm n times, each time obtaining a particular value of y, say y_i for trial i. Then, solve the set of equations

$$\begin{aligned} a\cdot y_1 &= 0 \\ a\cdot y_2 &= 0 \\ &\vdots \\ a\cdot y_n &= 0 \end{aligned} \tag{3.16}$$

for a.

The value a is then the period of the function f and the solution to the problem. This solution was obtained in only polynomial time – a considerable speedup over the case of a classical computer.

3.2 Prime Factorization

For an excellent overview of Shor's algorithm, see [9]. More detailed accounts of the algorithm can be found in [38, 39].

The goal of the following algorithm is to factor an integer N. For the first step, randomly choose an integer x that is relatively prime to N. If x and N did share a factor, Euclid's algorithm could be used to efficiently reduce the problem to a simpler one.

Next, consider the function

$$f(a) = x^a \pmod{N}, \tag{3.17}$$

where x is the integer chosen above. Evaluations of this function will look like

$$\begin{aligned} &1, x, \ldots, x^{r-1}, x^r, x^{r+1}, \ldots \\ &\underbrace{1, x, \ldots, x^{r-1}}_{\text{r terms}}, \underbrace{1\ , x\quad , \ldots ,\quad}_{\text{r terms}}\underbrace{1, x, \ldots\qquad}_{\text{r terms}}, \end{aligned} \tag{3.18}$$

where the top row is just $f(a)$ and the bottom is reduced modulo N. The number r, the smallest power where values start to repeat, is then the period of the function f. This period is, in general, difficult to obtain classically – requiring many function evaluations until a pattern is recognizable. However, Shor's algorithm can efficiently compute the period of a function such as f. So,

now the reader must wonder, what does this period have to do with factoring N?

In this algorithm, consider only even periods r. For randomly chosen x, the period r of the function (3.17) will be even fifty percent of the time. If r is odd, start over with a new (randomly chosen) x. This should not take too many tries[38, 19].

The (now even) period r is just the smallest number such that

$$x^r \equiv 1 \pmod{N}. \tag{3.19}$$

Rewrite this as the difference of squares

$$\left(x^{\frac{r}{2}}\right)^2 - 1 \equiv 0 \pmod{N}, \tag{3.20}$$

so that

$$\left(x^{\frac{r}{2}} + 1\right)\left(x^{\frac{r}{2}} - 1\right) \equiv 0 \pmod{N}. \tag{3.21}$$

This says one or the other of the factors in the product on the left hand side of (3.21) share a factor with N. An answer is now on the horizon. To obtain this answer, just compute[2]

$$\gcd\left(x^{\frac{r}{2}} + 1, N\right) \tag{3.22}$$

and

$$\gcd\left(x^{\frac{r}{2}} - 1, N\right). \tag{3.23}$$

Any non–trivial divisor will be a factor of N, and hence a solution to the problem.

[2] *Efficient* classical algorithms exist to compute the greatest common divisor of two numbers[29].

3.2.1 Shor's Algorithm

How can a quantum circuit find the period of a function such as (3.17)? First, the algorithm starts with two n–bit zero–registers

$$|0\rangle|0\rangle. \tag{3.24}$$

Next, as probably anticipated by now, comes a Hadamard transformation on the first register

$$(H \otimes 1)\,|0\rangle|0\rangle, \tag{3.25}$$

leaving

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle|0\rangle. \tag{3.26}$$

Now, construct the unitary operator

$$U_f \colon |x\rangle|0\rangle \to |x\rangle|f(x)\rangle \tag{3.27}$$

that evaluates the function

$$f(a) = x^a \pmod N. \tag{3.28}$$

Act with this operator on (3.26) to get

$$U_f \left(\sum_x |x\rangle|0\rangle \right) = \sum_x |x\rangle|f(x)\rangle. \tag{3.29}$$

The two quantum registers are now entangled. . . operations on one will affect the other. For the next step, perform a measurement such as

$$1 \otimes |c\rangle\langle c| \tag{3.30}$$

on the second register to get

$$\sum_{x \in f^{-1}(c)} |x\rangle|c\rangle. \tag{3.31}$$

This has collapsed the first register into an equally weighted superposition of states $|x\rangle$ such that $f(x) = c$.

All that's really needed at this point is the period of the function f, so the next step is simply a Fourier transform of the first register in order to find r.

The Quantum Discrete Fourier Transform

An N–dimensional quantum register $|x\rangle$ under a Fourier transform will look like

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi i x y}{N}} |y\rangle. \tag{3.32}$$

This operation is, of course, unitary and hence reversible. There have also been very efficient techniques developed to calculate this via a quantum circuit[13] of one– and two–qubit quantum gates.

A quantum Fourier transform applied to (3.31) will give

$$\sum_{x \in f^{-1}(c)} \sum_{y=0}^{N-1} e^{\frac{2\pi i x y}{N}} |y\rangle |c\rangle. \tag{3.33}$$

Now, with a little cleanup work, measurements of the first register yield the period r. The sum over y in (3.33) will yield constructive interference from the Fourier coefficients only when $\frac{y}{N}$ is a multiple of $\frac{1}{r}$. So, a measurement of the first register will give some (random) multiple of $\frac{1}{r}$ with high probability. This must be repeated until the multiple of $\frac{1}{r}$ is relatively prime to r, allowing r to be uniquely determined.

3.2.2 An Example

An example of Shor's algorithm is presented here directly from [9].

Consider factoring the number $N = 91$. Initially, we randomly choose $x = 3$ and find the function $f(a) = 3^a \pmod{91}$ will evaluate to

$$\begin{array}{rrrrrrrrrr} \text{a:} & 0, & 1, & 2, & 3, & 4, & 5, & 6, & 7, & \ldots \\ 3^a\text{:} & 1, & 3, & 9, & 27, & 81, & 243, & 729, & 2187, & \ldots \\ 3^a \pmod{91}\text{:} & 1, & 3, & 9, & 27, & 81, & 61, & 1, & 3, & \ldots \end{array} \tag{3.34}$$

The version of the algorithm running on a quantum computer will, of course, compute the period with a Fourier transform. Here, however, we'll resort to *wet–ware* of the human sort. By inspection of (3.34), the period $r = 6$. This is even, so the algorithm may proceed. Rearranging $3^6 \equiv 1 \pmod{91}$ as

$$3^6 - 1 \equiv 0 \pmod{91}, \tag{3.35}$$

and so

$$\begin{aligned} \left(3^3 + 1\right)\left(3^3 - 1\right) &\equiv 0 \pmod{91} \\ (28)(26) &\equiv 0 \pmod{91}. \end{aligned} \tag{3.36}$$

So, this implies that either

$$\gcd(91, 26) = 13 \tag{3.37}$$

or

$$\gcd(91, 28) = 7 \tag{3.38}$$

will be a non–trivial factor of 91. In this case, both are factors of $91 = 7 \times 13$, and the problem is completed.

3.3 Searching a Quantum Database

Grover's algorithm is a version of the ubiquitous searching algorithm that will run on a quantum computer. Consider the *quantum* solution to a simple

search problem of the following form: A single item is searched for in a set, a *database*, of unordered items. This item is guaranteed to be listed in the database exactly once, but the location of the item within the database is unknown.

Items in the database can be enumerated by the states of a quantum system. In particular, a database containing N items can be represented by an N–state system consisting of $n \geq \log(N)$ coupled 2–state systems

$$|\Psi\rangle = \sum_{i=0}^{2^N-1} a_i|i\rangle. \tag{3.39}$$

This representation, discussed at great length in chapter (2), should be quite familiar to the reader by now. A simple quantum algorithm to "search" this quantum database for a particular item consists of unitary evolution that amplifies the probability of finding the resultant state in the *desired* direction while simultaneously suppressing the probabilities for all other components. *i.e.*, for the expansion

$$|\Psi\rangle = a_{\text{desired}}|\text{desired}\rangle + \sum_{i\neq\text{desired}} a_i|i\rangle, \tag{3.40}$$

the search algorithm would consist of a unitary operator that took $a_{\text{desired}} \mapsto 1$ while $a_{\text{all others}} \mapsto 0$.

Grover's algorithm consists of a set of unitary operators that *approximately* accomplish the task described above in $O(\sqrt{N})$ iterations.

3.3.1 The Algorithm

Grover's algorithm is briefly outlined in figure (3.2). In the first step, an

Grover's Quantum Search Algorithm

1. First, start with an equally weighted superposition $|S\rangle$ of all possible states.
2. Next, repeat the following sequence $O(\sqrt{N})$ times[a]:
   (a) Query the oracle. Given any state $|S\rangle$, rotate the phase of the component in the direction of the desired state by π radians.
   (b) Flip the state about its "average" by applying what Grover refers to as the diffusion transform, $D|S\rangle$.
3. Finally, measure the resulting state. This should be the desired state with probability of at least $\frac{1}{2}$.

[a] exactly when to stop is important here.

Figure 3.2: An overview of Grover's algorithm.

equally weighted superposition of all possible states is, of course, obtained by hitting the n-dimensional zero register with the Hadamard transformation

$$H|0\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle = \begin{bmatrix} \frac{1}{\sqrt{2^n}} \\ \frac{1}{\sqrt{2^n}} \\ \vdots \\ \frac{1}{\sqrt{2^n}} \end{bmatrix}. \tag{3.41}$$

The numbers in the 2^n–dimensional column vector on the right of equation (3.41) are the (now equal) weights of each of the 2^n basis vectors for $\mathbb{Z}_2^n$ that make up the state of the register. For simplicity, write $N = 2^n$ so that

$$H|0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle = \begin{bmatrix} \frac{1}{\sqrt{N}} \\ \frac{1}{\sqrt{N}} \\ \vdots \\ \frac{1}{\sqrt{N}} \end{bmatrix}. \tag{3.42}$$

The Oracle

The concept of an *oracle* in step (2a) of figure (3.2) can be confusing at first glance. The idea is that there are criteria by which the search algorithm decides that the desired item has been found. This is often thought of in terms of a function

$$f\colon \quad \{\text{Search Domain}\} \quad \mapsto \quad \{Y, N\}, \tag{3.43}$$

that obtains a "yes" result for only a single *desired* input. Of course, this function will look a little more like

$$f\colon \quad \{0,1\}^n \quad \mapsto \quad \{0,1\} \tag{3.44}$$

for a search domain consisting of $N = 2^n$ inputs. When the details of this function evaluation are known, searches can be optimized accordingly. On the other hand, when this function is evaluated as a *black box*, where nothing is known about the function except perhaps bounds on the complexity of evaluation, then this is considered an **oracle query**.

In classical searches, an oracle query can either be a black box or given by an explicit decision function. A simple and familiar example of such a decision function might be a lexicographical comparison of the name on a file folder to some desired name.

For Grover's algorithm, the decision function is evaluated as a black box. Nothing is known about the function evaluation except that it will be implemented via some quantum circuit like that introduced by the Deutsch–Josza algorithm (section 3.1.1). How can this be implemented physically if nothing is known about the function evaluation? Consider the query of the oracle in Grover's algorithm:

Query the oracle. Given any state $|S\rangle$, rotate the phase of the component in the direction of the desired state by π radians.

Denote the desired state in the search by $|\omega\rangle$. Any state can then be expanded in terms of $|\omega\rangle$ and everything else $|\omega_\perp\rangle$ as

$$|S\rangle = |\omega\rangle\langle\omega|S\rangle + |\omega_\perp\rangle\langle\omega_\perp|S\rangle. \tag{3.45}$$

The oracle query $\mathbf{U}_\omega$ acting on $|S\rangle$

$$\mathbf{U}_\omega|S\rangle = \Big(\mathbf{1} - 2|\omega\rangle\langle\omega|\Big)|S\rangle \tag{3.46}$$

simply flips the state $|S\rangle$ about the (hyper)plane spanned by $|\omega_\perp\rangle$. *i.e.*, The phase of the component of the state in the direction of the desired state is rotated by π radians (see figure (3.3)). The resulting state will look like

$$\mathbf{U}_\omega|S\rangle = \Big(\mathbf{1} - 2|\omega\rangle\langle\omega|\Big)|S\rangle = -|\omega\rangle\langle\omega|S\rangle + |\omega_\perp\rangle\langle\omega_\perp|S\rangle. \tag{3.47}$$

Note that this operation can be implemented without knowing exactly where $|\omega\rangle$ is in the state expansion. This operation still works despite the fact that it's not possible to write this down in terms of a matrix such as

$$\mathbf{U}_\omega|S\rangle = \begin{bmatrix} 1 & 0 & \cdots & & 0 \\ 0 & \ddots & & & \\ \vdots & & e^{i\pi} & & \vdots \\ & & & \ddots & \\ 0 & & \cdots & & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{N}} \\ \vdots \\ \frac{1}{\sqrt{N}} \\ \vdots \\ \frac{1}{\sqrt{N}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{N}} \\ \vdots \\ -\frac{1}{\sqrt{N}} \\ \vdots \\ \frac{1}{\sqrt{N}} \end{bmatrix}, \tag{3.48}$$

because you don't initially know where to put the phase shift. This is not a problem because it *is* possible to write an algorithm, even classically, that examines each input and tests for successful ("yes") function evaluation. This

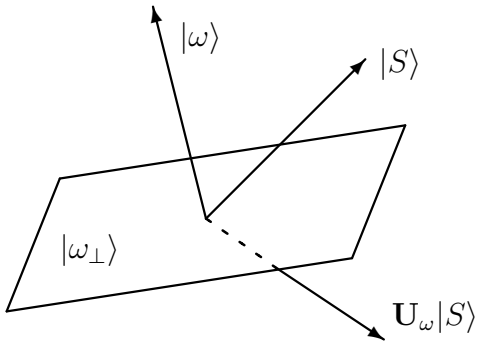

Figure 3.3: The oracle query in Grover's algorithm reflects the state $|S\rangle$ through the plane $|\omega_\perp\rangle$.

issue has already been resolved by simply writing the transformation as the operator

$$\mathbf{U}_\omega = \big(\mathbf{1} - 2|\omega\rangle\langle\omega|\big). \tag{3.49}$$

Inversion About the Average

Step (2b) of Grover's algorithm (figure 3.2) is an inversion of the state about the *average* state. This so called *diffusion transformation* can be written

$$\mathbf{U_s} = 2|s\rangle\langle s| - \mathbf{1}, \tag{3.50}$$

where, as before,

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \tag{3.51}$$

is the equally–weighted superposition (the "average") of all computational basis states. The diffusion transformation preserves $|s\rangle$, but flips the sign of any vector orthogonal to $|s\rangle$... it *rotates*[3] a vector around $|s\rangle$.

[3]Of course, this is really a flip.. it is a rotation only for certain dimensions.

Now, rewrite (3.50) as

$$\begin{aligned}\mathbf{U_s} =& \frac{2}{N}\sum_{i,j}|i\rangle\langle j| - \mathbf{1} \\ =& \frac{2}{N}\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & & \\ \vdots & & \ddots & \vdots \\ 1 & \cdots & & 1 \end{pmatrix} - \mathbf{1} \\ =& \begin{pmatrix} \frac{2-N}{N} & \frac{2}{N} & \cdots & \frac{2}{N} \\ \frac{2}{N} & \frac{2-N}{N} & & \\ \vdots & & \ddots & \vdots \\ \frac{2}{N} & \cdots & & \frac{2-N}{N} \end{pmatrix}.\end{aligned} \tag{3.52}$$

Each iteration in Grover's algorithm can be written as the unitary transformation

$$\mathbf{U}_{\text{Grover}} = \mathbf{U}_s\mathbf{U}_\omega, \tag{3.53}$$

which is just a query of the oracle followed by a flip about the average. This combined unitary operator cannot be written in matrix form (because of $\mathbf{U}_\omega$), but the effect upon the input state can be calculated. Consider the state

$$|\Psi\rangle = k|\omega\rangle + l|\omega_\perp\rangle. \tag{3.54}$$

A single Grover iteration applied to this state gives

$$\begin{aligned}
\mathbf{U}_{\text{Grover}}|\Psi\rangle &= \mathbf{U_s}\mathbf{U}_\omega\left(k|\omega\rangle + l|\omega_\perp\rangle\right) \\
&= \mathbf{U_s}\left(-k|\omega\rangle + l|\omega_\perp\rangle\right) \\
&= \left\{2|s\rangle\langle s| - \mathbf{1}\right\}\left(-k|\omega\rangle + l|\omega_\perp\rangle\right) \\
&= \left\{\frac{2}{N}\sum_{i,j}|i\rangle\langle j| - \mathbf{1}\right\}\left(-k|\omega\rangle + l|\omega_\perp\rangle\right) \\
&= k|\omega\rangle - l|\omega_\perp\rangle + \frac{2}{N}\sum_{i,j}\left\{-k|i\rangle\langle j|\omega\rangle + l|i\rangle\langle j|\omega_\perp\rangle\right\} \\
&= k|\omega\rangle - l|\omega_\perp\rangle + \frac{2}{N}\sum_{i}|i\rangle\left\{-k\sum_j\langle j|\omega\rangle + l\sum_j\langle j|\omega_\perp\rangle\right\}.
\end{aligned} \tag{3.55}$$

Since the ket $|\omega\rangle$ was chosen to be an element of the orthonormal basis $|i\rangle$,

$$\sum_j\langle j|\omega\rangle = 1, \qquad \text{and} \qquad \sum_j\langle j|\omega_\perp\rangle = N-1. \tag{3.56}$$

A single step of Grover then looks like

$$\mathbf{U}_{\text{Grover}}|\Psi\rangle = k|\omega\rangle - l|\omega_\perp\rangle + \frac{2}{N}\sum_i|i\rangle\left(-k + l(N-1)\right). \tag{3.57}$$

To find the coefficients for the resulting state after the Grover iteration is applied, take

$$\begin{aligned}
k_{\text{new}} =& \langle\omega|\mathbf{U}_{\text{Grover}}|\Psi\rangle \\
=& k\langle\omega|\omega\rangle + \frac{2}{N}\left(-k + l(N-1)\right)\langle\omega|\omega\rangle \\
=& \frac{N-2}{N}k + \frac{2(N-1)}{N}l
\end{aligned} \tag{3.58}$$

and

$$\begin{aligned}
l_{\text{new}} =& \langle\omega_\perp|\mathbf{U}_{\text{Grover}}|\Psi\rangle \\
=& -l\langle\omega_\perp|\omega_\perp\rangle + \frac{2}{N}\left(-k + l(N-1)\right)\langle\omega_\perp|\omega_\perp\rangle \\
=& -\frac{2}{N}k + \frac{N-2}{N}l.
\end{aligned} \tag{3.59}$$

So, under the action of Grover's algorithm, the state at any given iteration can be described by

$$|\Psi_i\rangle = k_i|\omega\rangle + l_i|\omega_\perp\rangle, \tag{3.60}$$

where k_i and l_i are determined iteratively by

$$\begin{aligned} k_{i+1} =& \frac{N-2}{N}k_i + \frac{2(N-1)}{N}l_i \\ l_{i+1} =& -\frac{2}{N}k_i + \frac{N-2}{N}l_i. \end{aligned} \tag{3.61}$$

Measurement

The final step in Grover's algorithm is simply to measure the resulting state after the appropriate number of iterations. The desired state should then be measured with probability of at least $\frac{1}{2}$.

3.3.2 Complexity Analysis

In order to understand the complexity of Grover's algorithm, it is useful to discuss the complexity associated with a similar search on a classical computer.

The relevant classical search problem is as follows: start with a large unsorted database containing $N >> 1$ items. Find one particular item in this database... search for a "needle in a haystack" as Grover puts it[28]. As discussed earlier, this database can be represented by a table or a function $f(x)$ where $x \in \{0, 1, \ldots, N-1\}$. The item searched for is guaranteed to be in the database exactly once, so $f(x) = a$ for only a single value of x. Now, given a, find x.

If the database has been sorted, x can be found by evaluating the function for only $\log N$ possible values of the domain before the value a is

obtained.

Suppose N is a power of 2, say $N = 2^n$. Then the search would start by evaluating $f(x)$ for $x = 2^{n-1} - 1$, and then comparing the result to the desired result, a. If $f(x)$ is greater than a, then, taking advantage of the fact that the database is ordered, repeat the search step by evaluating $f(x)$ on $x = 2^{n-2} - 1$ and comparing, etc.

Consider, for a nice concrete example, a database consisting of file folders. The oracle function f is simply to read the name on the folder, the value of the desired folder being the name a. If the database is ordered, lexicographically say, then the search would start by grabbing the middle folder ($x = 2^n - 1$), reading the name on the folder (evaluating the function f), and then comparing this name to the name a. If the name a is later in the list, choose a folder three quarters of the way back and try again. In a worst–case scenario, you will have to look at $\log_2 N$ different folders to find the one with the name a. Another example of this kind of search is, knowing a person's name, looking up their phone number in the phone book. These are all searches of an ordered database.

On the other hand, if the database is *not* ordered the search becomes much more difficult. Take, for instance, the seemingly intractable problem of using the phone book to look up a person's name given their phone number. This is essentially looking for an item in an unordered list – searching for a needle in a haystack. Considerably more function evaluations would be required in this instance. It requires $\frac{N}{2}$ lookups or evaluations before the probability of success is greater than $\frac{1}{2}$.

On a *quantum computer*, Grover's algorithm manages to get to a probability of success of at least $\frac{1}{2}$ in only about $\sqrt{N}$ evaluations. A careful analysis of why this should be true is in chapter 5.

3.3.3 Multisearch and Other Advanced Searches

Several variations of the standard search algorithm can be performed on a quantum computer. Instead of searching for a single item in a list, you may be looking for one of a set of items that fit the search criteria, a multisearch. Grover's algorithm handles this type of search the same way it handles searching for a single object. Start from an equally–weighted superposition

$$|\Psi\rangle = \frac{1}{\sqrt{N}} \sum_i |i\rangle. \tag{3.62}$$

Instead of simply amplifying a single coefficient while simultaneously suppressing all others, amplify the set of all items meeting the search criteria. The resulting state

$$|\Psi_{\text{result}}\rangle = \sum_{i \in \{\text{desired}\}} a|i\rangle + \sum_{j \notin \{\text{desired}\}} b|j\rangle \tag{3.63}$$

is an equally weighted superposition of all states meeting the search criteria, which, when measured, is equally likely to be in any one of these states (here, b is small). The primary difference between the single search and multisearch versions of Grover's algorithm is the stopping time. A multisearch must be stopped before it's simpler cousin.

Consider the case when there are t solutions to the problem. *i.e.*, t values x in the database that meet the search criteria ($f(x) =$ "yes"). Let $A = \{x|f(x) =$ "yes"$\}$, and $B = \{x|f(x) =$ "no"$\}$. Then, as in equation

(3.54), consider

$$|\Psi\rangle = \sum_{x \in A} k|x\rangle + \sum_{y \in B} l|y\rangle. \tag{3.64}$$

The state at any iteration i of Grover's algorithm will be

$$|\Psi_i\rangle = \sum_{x \in A} k_i|x\rangle + \sum_{y \in B} l_i|y\rangle, \tag{3.65}$$

where the coefficients can be calculated iteratively

$$\begin{aligned} k_{i+1} =& \frac{N-2t}{N} k_i + \frac{2(N-t)}{N} l_i \\ l_{i+1} =& -\frac{2t}{N} k_i + \frac{N-2t}{N} l_i. \end{aligned} \tag{3.66}$$

This version of the algorithm will find an answer in about $\sqrt{\frac{N}{t}}$ iterations, and must be stopped *before* the $\sqrt{N}$ iterations for the single item search.

Arbitrary Initial Distributions

Another extension of the simple quantum search is to start the search from an arbitrary distribution. Instead of starting with an equally–weighted superposition of states, allow the search to run from any initial distribution. This approach shows the stability of Grover's algorithm with respect to noise in the initial setup[5].

Chapter 4

Quantum Geometry

4.1 Statistical Distance and Hilbert Space

In classical probability, the *distance* between two distinguishable events in state space can be characterized in terms of the maximum number of mutually distinguishable intermediate events. This notion was generalized to quantum states by Wooters[46], who found that the *statistical distance* between two quantum states coincides with the *metric distance* between two rays in Hilbert space.

Optimization of the statistical distinguishability of quantum states leads to a natural Hermitian metric on the space of density operators[10]. This metric is, for pure states, the usual Fubini–Study metric on projective Hilbert space[24], and can be extended to general density operators for mixed states[10].

4.1.1 Classical Statistical Distance

It is worth noting, that statistical distance on a classical probability space is not just the Euclidean distance between two probabilities $|p_1 - p_2|$. This is due to the fact that probabilities near $\frac{1}{2}$ are harder to distinguish due to higher statistical fluctuations than probabilities near 0 or 1.

As an example, consider two differently weighted coins. Coin i has an *a priori* probability p_i of obtaining heads in a given toss. The probability space is one dimensional – just the probability p_i of heads for each coin. Define, following Wooters[46], the statistical distance between two coins in this probability space as

$$d(p_1, p_2) = \lim_{n\to\infty} \frac{1}{\sqrt{n}} \times \left[\begin{array}{c} \text{maximum number of mutually distinguishable} \\ \text{(in } n \text{ trials) intermediate probabilities} \end{array} \right], \tag{4.1}$$

where two intermediate probabilities p and p' are distinguishable in n trials if

$$|p - p'| \geq \Delta p + \Delta p'. \tag{4.2}$$

An expression for Δp can be given in the spirit of experimental uncertainty as RMS deviation

$$\Delta p = \left[\frac{p(1-p)}{n} \right]^{\frac{1}{2}}. \tag{4.3}$$

This gives

$$\begin{aligned} d(p_1, p_2) &= \lim_{n\to\infty} \frac{1}{\sqrt{n}} \times \int_{p_1}^{p_2} \frac{dp}{2\Delta p} \\ &= \lim_{n\to\infty} \frac{1}{\sqrt{n}} \times \int_{p_1}^{p_2} \frac{\sqrt{n} dp}{2\sqrt{p(1-p)}} \\ &= \int_{p_1}^{p_2} \frac{dp}{2\sqrt{p(1-p)}}, \end{aligned} \tag{4.4}$$

or

$$d(p_1, p_2) = \cos^{-1}\left(\sqrt{p_1}\sqrt{p_2} + \sqrt{q_1}\sqrt{q_2} \right), \tag{4.5}$$

where $q_1 = 1 - p_1$ and $q_2 = 1 - p_2$, which is obviously quite different from the ordinary Euclidean distance $|p_2 - p_1|$ on the space of probabilities.

It is easy to generalize this distance when the system has more that just two outcomes, p and q. When there are countably infinite outcomes, the

notion of statistical distance becomes

$$d(p_1, p_2) = \cos^{-1}\left[\sum_i \sqrt{p_1^{(i)}}\sqrt{p_2^{(i)}}\right]. \tag{4.6}$$

For this notion of distance to apply to a quantum mechanical system, some sort of measurement apparatus must be chosen. The statistical distance between two states can then be given *with respect to this measuring device*. For instance, if $\phi_1, \ldots, \phi_N$ form an eigenbasis for the measuring device A, then the statistical distance between two different states ψ_1 and ψ_2 is given by

$$d_A(\psi_1, \psi_2) = \cos^{-1}\left[\sum_{i=1}^{N} \big|(\phi_i, \psi_1)\big|\big|(\phi_i, \psi_2)\big|\right]. \tag{4.7}$$

The "absolute" statistical distance will be the maximum statistical distance over all possible measuring devices, which is given, not surprisingly, by

$$d(\psi_1, \psi_2) = \cos^{-1}\Big[\big|(\psi_1, \psi_2)\big|\Big]. \tag{4.8}$$

4.1.2 A Quantum Mechanical Approach

Consider two pure states:

$$\begin{aligned} |\psi\rangle &= \sum_j \sqrt{p_j} e^{i\phi_j}|j\rangle, \\ |\tilde{\psi}\rangle = |\psi\rangle + |d\psi\rangle &= \sum_j \sqrt{p_j + dp_j}\, e^{i(\phi_j + d\phi_j)}|j\rangle. \end{aligned} \tag{4.9}$$

For $|\psi\rangle$ and $|\tilde{\psi}\rangle$ normalized, we expect

$$\langle\psi|\tilde{\psi}\rangle = cos(\theta), \tag{4.10}$$

where θ is the "angle" between these two states in (projective) Hilbert space. As seen in the last section, this angle presents a measure of the notion of state

distinguishability. *i.e.*,

$$\frac{1}{4}ds^2_{\text{Pure States}} = \left[\cos^{-1}\left(\left|\langle\tilde{\psi}|\psi\rangle\right|\right)\right]^2. \tag{4.11}$$

Notice that this can also be written

$$\frac{1}{4}ds^2_{\text{Pure States}} = \left[\cos^{-1}\left(\left|\langle\tilde{\psi}|\psi\rangle\right|\right)\right]^2 = 1 - \left|\langle\tilde{\psi}|\psi\rangle\right|^2 = \langle d\psi_\perp|d\psi_\perp\rangle. \tag{4.12}$$

If we define $|d\psi_\perp\rangle = |d\psi\rangle - |\psi\rangle\langle\psi|d\psi\rangle$, then

$$\langle d\psi_\perp|d\psi_\perp\rangle = \langle d\psi|d\psi\rangle - |\langle\psi|d\psi\rangle|^2. \tag{4.13}$$

Since (from (4.9)) $|d\psi\rangle = |\tilde{\psi}\rangle - |\psi\rangle$, we can write

$$\begin{aligned}
\langle d\psi|d\psi\rangle =& \left(\langle\tilde{\psi}| - \langle\psi|\right)\left(|\tilde{\psi}\rangle - |\psi\rangle\right) \\
=& \langle\tilde{\psi}|\tilde{\psi}\rangle - \langle\tilde{\psi}|\psi\rangle - \langle\psi|\tilde{\psi}\rangle + \langle\psi|\psi\rangle \\
=& 2 - \langle\psi|\tilde{\psi}\rangle - \langle\tilde{\psi}|\psi\rangle \\
=& 2 - \left(\sum_j \sqrt{p_j}\sqrt{p_j + dp_j}e^{id\phi_j}\right) - \left(\sum_j \sqrt{p_j}\sqrt{p_j + dp_j}e^{-id\phi_j}\right) \\
=& 2 - 2\sum_j \sqrt{p_j}\sqrt{p_j + dp_j}\,\cos(d\phi_j).
\end{aligned} \tag{4.14}$$

Now, we can expand to second order in $d\phi$ and dp to get

$$\begin{aligned}
\langle d\psi|d\psi\rangle \approx& 2\left\{1 - \sum_j p_j\left(1 + \frac{dp_j}{p_j}\right)^{\frac{1}{2}}\left[1 - \frac{(d\phi_j)^2}{2}\right]\right\} \\
\approx& 2\left\{1 - \sum_j p_j\left[1 + \frac{1}{2}\left(\frac{dp_j}{p_j}\right) - \frac{1}{8}\left(\frac{dp_j}{p_j}\right)^2\right]\left[1 - \frac{(d\phi_j)^2}{2}\right]\right\} \\
=& 2\left\{1 - \sum_j \left(p_j + \frac{1}{2}dp_j - \frac{1}{8}\frac{(dp_j)^2}{p_j} - \frac{1}{2}p_j(d\phi_j)^2\right)\right\}.
\end{aligned} \tag{4.15}$$

Now, $\{p_i\}$ are probabilities, so $\sum_j p_j = 1$ and $\sum_j dp_j = 0$, which gives

$$\langle d\psi|d\psi\rangle \approx \sum_j \left(\frac{1}{4}\frac{(dp_j)^2}{p_j} + p_j(d\phi_j)^2\right). \tag{4.16}$$

Similarly,

$$\begin{aligned}\langle\psi|d\psi\rangle =&\langle\psi|\left(|\tilde{\psi}\rangle - |\psi\rangle\right) = \langle\psi|\tilde{\psi}\rangle - \langle\psi|\psi\rangle \\ =&\langle\psi|\tilde{\psi}\rangle - 1 \\ =&\sum_j \sqrt{p_j}\sqrt{p_j + dp_j}\, e^{id\phi_j} - 1.\end{aligned} \tag{4.17}$$

Expanding to first order in dp and $d\phi$, we get

$$\begin{aligned}\langle\psi|d\psi\rangle &\approx \left\{\sum_j p_j \left[1 + \frac{1}{2}\left(\frac{dp_j}{p_j}\right)\right][1 + id\phi_j]\right\} - 1 \\ &\approx \sum_j \left(p_j + \frac{1}{2}dp_j + ip_j d\phi_j\right) - 1 \\ &= i\sum_j p_j d\phi_j,\end{aligned} \tag{4.18}$$

and then equation (4.13) gives

$$\begin{aligned}\langle d\psi_\perp|d\psi_\perp\rangle &= \langle d\psi|d\psi\rangle - |\langle\psi|d\psi\rangle|^2 . \\ &\approx \sum_j \left(\frac{1}{4}\frac{(dp_j)^2}{p_j} + p_j(d\phi_j)^2\right) - \left(\sum_j p_j d\phi_j\right)^2 \\ &\approx \sum_j \frac{1}{4}\frac{(dp_j)^2}{p_j} + \left[\sum_j p_j(d\phi_j)^2 - \left(\sum_j p_j d\phi_j\right)^2\right].\end{aligned} \tag{4.19}$$

The absolute statistical distance for pure states is given by

$$ds^2_{\text{Pure States}} = \sum_i \frac{1}{4}\frac{(dp_i)^2}{p_i} + \left(\sum_i p_i(d\phi_i)^2 - \left(\sum_i p_i d\phi_i\right)^2\right). \tag{4.20}$$

4.2 The Geometry of Quantum Mechanics

4.2.1 Complex Projective Space

The complex projective space $\mathbb{C}P^n$ is the set of complex lines through the origin of $\mathbb{C}^{n+1}$. A point $z = (z^0, \ldots, z^n) \in \mathbb{C}^{n+1}$ defines a line through the

origin provided $z \neq 0$. Another point $w \in \mathbb{C}^{n+1}$ lies on the same line as that defined by z if there exists a complex number $\alpha \neq 0$ such that $w = \alpha z$. This defines an equivalence relation $\sim$. *i.e.*, $w \sim z$ if there exists $\alpha \in \mathbb{C}^{n+1} \setminus \{0\}$ such that $w = \alpha z$. Then, $\mathbb{C}P^n$ can be defined as $\mathbb{C}P^n = (\mathbb{C}^{n+1} \setminus \{0\}) / \sim$.

The set of $n+1$ coordinates $z^0, \ldots, z^n$ are referred to as *homogeneous coordinates* for $\mathbb{C}P^n$. However, $\mathbb{C}P^n$ is an n-dimensional complex manifold with (n) coordinates defined as follows. Consider the coordinate patch U_i defined as the set of all complex lines through the origin of $\mathbb{C}^{n+1}$ such that $z^i \neq 0$. Define coordinates on this patch by

$$\xi^j_{(i)} = \frac{z^j}{z^i}. \tag{4.21}$$

These are *inhomogeneous coordinates* for $\mathbb{C}P^n$.

For a point $z = (z^0, \ldots, z^n)$ in the intersection of two coordinate patches $U_i \cap U_j$, the two inhomogeneous coordinates for this point $\xi^k_{(i)} = z^k/z^i$ and $\xi^k_{(j)} = z^k/z^j$ are related by the transition functions

$$\psi_{ij}: \quad \xi^k_{(j)} \quad \mapsto \quad \xi^k_{(i)} = \left(\frac{z^j}{z^i}\right) \xi^k_{(j)}, \tag{4.22}$$

which makes $\mathbb{C}P^n$ a nice smooth complex manifold.

Inhomogeneous coordinates for $\mathbb{C}P^n$ will be used exclusively throughout the remainder of this dissertation. Consequently, I will adapt a slight abuse of notation to simplify many later expressions. The $n+1$ *homogeneous* coordinates for a point in $\mathbb{C}^{n+1}$ will be expressed as $(z^0, z^1, \ldots, z^n)$, whereas the n *inhomogeneous* coordinates for the corresponding point in $\mathbb{C}P^n$ will be expressed, not as $(\xi^1_{(0)}, \ldots, \xi^n_{(0)})$ in the U_0 coordinate patch, but simply as

$(z^1, \ldots, z^n)$, with the coordinate patch implied. Nothing will be covered that is dependent upon which particular coordinate patch is used.

For more on complex manifolds and/or projective spaces see [33], [47], or [34].

4.2.2 The Fubini–Study Metric

A natural Hermitian metric, the Fubini–Study metric[23, 43], can be defined on complex projective space. This metric is given, in inhomogeneous coordinates, as

$$ds^2_{FS} = g_{\overline{i}j}\, dz^i \otimes d\overline{z}^j, \tag{4.23}$$

where

$$g_{\overline{i}j} = 4\,\frac{\delta_{ij}\left(1+|z|^2\right) - \overline{z}_i z_j}{\left(1+|z|^2\right)^2}. \tag{4.24}$$

Fubini–Study as Statistical Distance

The Fubini–Study metric is a notion of distance on a projective Hilbert space that corresponds exactly to expressions of statistical distance previously discussed. To see this, consider the line element from the Fubini–Study metric (equation (4.24))

$$\begin{aligned} ds^2 =& \left(\frac{\delta_{ij}\left(1+|z|^2\right) - \overline{z}_i z_j}{\left(1+|z|^2\right)^2}\right) dz^i d\overline{z}^j \\ =& \frac{dz^i d\overline{z}^i}{\left(1+|z|^2\right)} - \frac{\overline{z}_i dz^i z_j d\overline{z}^j}{\left(1+|z|^2\right)^2}. \end{aligned} \tag{4.25}$$

In polar coordinates

$$\begin{aligned} z_j =& r_j e^{i\phi_j} \\ \overline{z}_j =& r_j e^{-i\phi_j}, \end{aligned} \tag{4.26}$$

with derivatives

$$\begin{aligned} dz_j &= (dr_j + ir_j d\phi_j)\, e^{i\phi_j} \\ d\overline{z}_j &= (dr_j - ir_j d\phi_j)\, e^{-i\phi_j}, \end{aligned} \tag{4.27}$$

this metric looks like

$$\begin{aligned} ds^2 = \frac{1}{(1+r^2)}\left[\sum_i (dr_i)^2 + \sum_i r_i^2 \,(d\phi_i)^2\right] \\ - \frac{1}{(1+r^2)^2}\left[\left(\sum_i r_i dr_i\right)^2 + \left(\sum_i r_i^2 d\phi_i\right)^2\right]. \end{aligned} \tag{4.28}$$

Now, to see the connection with the statistical distance described in section (4.1.2), consider coordinates for an $(n+1)$–state system

$$|\Psi\rangle = a_0|0\rangle + \cdots + a_n|n\rangle \tag{4.29}$$

with

$$a_i = \sqrt{p_i}e^{i\phi_i} \qquad i = 0 \ldots n+1. \tag{4.30}$$

We note here that the p_i are probabilities and hence obey

$$\begin{aligned} \sum_{i=0}^{n} p_i &= 1 \\ \sum_{i=0}^{n} dp_i &= 0. \end{aligned} \tag{4.31}$$

If these coordinates are viewed as $n+1$ *homogeneous* coordinates to $\mathbb{C}P^n$, then the n corresponding *inhomogeneous* coordinates can then be obtained (in coordinate patch U_0, where $a_0 \neq 0$) as

$$z_i = \frac{a_i}{a_0} = \frac{\sqrt{p_i}e^{i\phi_i}}{\sqrt{p_0}e^{i\phi_0}} = \sqrt{\frac{p_i}{p_0}}e^{i(\phi_i - \phi_0)}. \tag{4.32}$$

Physically, freedom of normalization and overall phase allow

$$|\Psi\rangle = a_0\left(|0\rangle + \frac{a_1}{a_0}|1\rangle + \cdots + \frac{a_n}{a_0}|n\rangle\right) \tag{4.33}$$

to be written simply as

$$|\Psi\rangle = \frac{1}{\sqrt{1+|z|^2}}\Big(|0\rangle + z_1|1\rangle + \cdots + z_n|n\rangle\Big), \tag{4.34}$$

with

$$z_i = \sqrt{\frac{p_i}{p_0}}e^{i\phi_i}. \tag{4.35}$$

The polar form of the Fubini–Study metric (4.28), can now be written in terms of probabilities p_i to obtain (4.20), the expected expression for statistical distance. If

$$r_i = \sqrt{\frac{p_i}{p_0}} \qquad (i = 1 \ldots n), \tag{4.36}$$

then

$$r^2 = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \frac{p_i}{p_0} = \frac{1}{p_0}\sum_{i=1}^{n} p_i. \tag{4.37}$$

Now, since $\sum_{i=0}^{n} p_i = 1$,

$$\begin{aligned} \sum_{i=1}^{n} p_i &= 1 - p_0 \\ \sum_{i=1}^{n} dp_i &= -\,dp_0. \end{aligned} \tag{4.38}$$

Then,

$$r^2 = \frac{1}{p_0} - 1, \tag{4.39}$$

and the factor of $1+r^2$ in (4.28) becomes simply $\frac{1}{p_0}$. The Fubini–Study metric then becomes

$$ds^2 = p_0\left[\sum_i (dr_i)^2 + \sum_i r_i^2\,(d\phi_i)^2\right] - p_0^2\left[\left(\sum_i r_i dr_i\right)^2 + \left(\sum_i r_i^2 d\phi_i\right)^2\right]. \tag{4.40}$$

Now, since

$$dr_i = \frac{1}{2}\left(p_i^{-\frac{1}{2}} p_0^{-\frac{1}{2}} dp_i - p_i^{\frac{1}{2}} p_0^{-\frac{3}{2}} dp_0\right) \tag{4.41}$$

leads to

$$(dr_i)^2 = \frac{(dp_i)^2}{4p_i p_0} + \frac{p_i (dp_0)^2}{4p_0^3} - \frac{dp_i dp_0}{2p_0^2} \tag{4.42}$$

and

$$r_i dr_i = \frac{dp_i}{2p_0} - \frac{p_i dp_0}{2p_0^2}, \tag{4.43}$$

then (4.40) becomes

$$\begin{aligned} ds^2 = p_0 \Bigg[\sum_i \left(\frac{(dp_i)^2}{4p_i p_0} + \frac{p_i (dp_0)^2}{4p_0^3} - \frac{dp_i dp_0}{2p_0^2} \right) + \frac{1}{p_0} \sum_i p_i \left(d\phi_i \right)^2 \Bigg] \\ - p_0^2 \Bigg[\left(\sum_i \frac{dp_i}{2p_0} - \sum_i \frac{p_i dp_0}{2p_0^2} \right)^2 + \frac{1}{p_0^2} \left(\sum_i p_i d\phi_i \right)^2 \Bigg]. \end{aligned} \tag{4.44}$$

Now, the expression for $r_i dr_i$ simplifies somewhat

$$\begin{aligned} \sum_i r_i dr_i =& \sum_i \frac{dp_i}{2p_0} - \sum_i \frac{p_i dp_0}{2p_0^2} \\ =& \frac{1}{2p_0} \left(\sum_i dp_i \right) - \frac{dp_0}{2p_0^2} \left(\sum_i p_i \right) \\ =& \frac{1}{2p_0} \left(-dp_0 \right) - \frac{dp_0}{2p_0^2} \left(1 - p_0 \right) \\ =& - \frac{dp_0}{2p_0} - \frac{dp_0}{2p_0^2} + \frac{dp_0}{2p_0} \\ =& - \frac{dp_0}{2p_0^2}. \end{aligned} \tag{4.45}$$

The line element then becomes

$$\begin{aligned} ds^2 = p_0 \Bigg[\sum_i \left(\frac{(dp_i)^2}{4p_i p_0} + \frac{p_i (dp_0)^2}{4p_0^3} - \frac{dp_i dp_0}{2p_0^2} \right) + \frac{1}{p_0} \sum_i p_i \left(d\phi_i \right)^2 \Bigg] \\ - p_0^2 \Bigg[\left(- \frac{dp_0}{2p_0^2} \right)^2 + \frac{1}{p_0^2} \left(\sum_i p_i d\phi_i \right)^2 \Bigg], \end{aligned} \tag{4.46}$$

or

$$ds^2 = \sum_i \left(\frac{(dp_i)^2}{4p_i} + \frac{p_i (dp_0)^2}{4p_0^2} - \frac{dp_i dp_0}{2p_0} \right) + \sum_i p_i \left(d\phi_i \right)^2 - \frac{dp_0^2}{4p_0^2} - \left(\sum_i p_i d\phi_i \right)^2. \tag{4.47}$$

The first sum can be reduced

$$\begin{aligned}
&\sum_i \left(\frac{(dp_i)^2}{4p_i} + \frac{p_i (dp_0)^2}{4p_0^2} - \frac{dp_i dp_0}{2p_0} \right) \\
&= \sum_i \frac{1}{4} \frac{(dp_i)^2}{p_i} + \frac{dp_0^2}{4p_0^2} \left(\sum_i p_i \right) - \frac{dp_0}{2p_0} \left(\sum_i dp_i \right) \\
&= \sum_i \frac{1}{4} \frac{(dp_i)^2}{p_i} + \frac{dp_0^2}{4p_0^2} (1 - p_0) - \frac{dp_0}{2p_0} (-dp_0) \\
&= \sum_i \frac{1}{4} \frac{(dp_i)^2}{p_i} + \frac{dp_0^2}{4p_0^2} + \frac{dp_0^2}{4p_0},
\end{aligned} \tag{4.48}$$

and then (4.47) becomes

$$\begin{aligned}
ds^2 &= \left(\sum_i \frac{1}{4} \frac{(dp_i)^2}{p_i} + \frac{dp_0^2}{4p_0^2} + \frac{dp_0^2}{4p_0} \right) + \sum_i p_i \left(d\phi_i \right)^2 - \frac{dp_0^2}{4p_0^2} - \left(\sum_i p_i d\phi_i \right)^2 \\
&= \sum_{i=1}^{n} \frac{1}{4} \frac{(dp_i)^2}{p_i} + \frac{1}{4} \frac{dp_0^2}{p_0} + \sum_{i=1}^{n} p_i \left(d\phi_i \right)^2 - \left(\sum_{i=1}^{n} p_i d\phi_i \right)^2 \\
&= \sum_{i=0}^{n} \frac{1}{4} \frac{(dp_i)^2}{p_i} + \sum_{i=1}^{n} p_i \left(d\phi_i \right)^2 - \left(\sum_{i=1}^{n} p_i d\phi_i \right)^2 .
\end{aligned} \tag{4.49}$$

Now, since the overall phase ϕ_0 was chosen to remain zero when specifying inhomogeneous coordinates ((4.32) to (4.35)), then $d\phi_0 = 0$, and all sums in (4.49) can be extended to include the zero case.

So then

$$ds^2 = \sum_{i=0}^{n} \frac{1}{4} \frac{(dp_i)^2}{p_i} + \sum_{i=0}^{n} p_i \left(d\phi_i \right)^2 - \left(\sum_{i=0}^{n} p_i d\phi_i \right)^2, \tag{4.50}$$

the statistical distance (4.20) for pure states, can be characterized geometrically as the Fubini–Study metric on projective Hilbert space.

4.2.3 Geodesics

An explicit expression, in inhomogeneous coordinates, for the equations of motion for geodesics of the Fubini–Study metric will be useful in the remaining chapters.

The inverse of the Fubini–Study metric, as written in equation (4.24), is easily shown to be

$$g^{\bar{i}j} = \frac{1}{4}\left(1+|z|^2\right)\left[\delta^{ij} + \overline{z}^i z^j\right]. \tag{4.51}$$

Since $\mathbb{C}P^n$ is a Kähler manifold[33], the only non–vanishing Christoffel symbols are Γ^a_{bc} and $\Gamma^{\overline{a}}_{\overline{bc}}$ ($= \overline{\Gamma^a_{bc}}$), given by

$$\begin{aligned} \Gamma^a_{bc} =& g^{\overline{s}a} g_{c\overline{s},b} \\ \Gamma^{\overline{a}}_{\overline{bc}} =& g^{\overline{a}s} g_{s\overline{c},b}. \end{aligned} \tag{4.52}$$

The geodesic equations become

$$\begin{aligned} \ddot{z}^l + \Gamma^l_{ik}\dot{z}^i\dot{z}^k =& 0 \\ \ddot{\overline{z}}^l + \Gamma^{\overline{l}}_{\overline{ik}}\dot{\overline{z}}^i\dot{\overline{z}}^k =& 0. \end{aligned} \tag{4.53}$$

Using equation (4.24), we can calculate partials of the metric

$$\begin{aligned} g_{i\bar{j},k} =& 4\frac{\partial}{\partial z^k}\left(\frac{\delta_{ij}\left(1+|z|^2\right) - z_i\overline{z}_j}{\left(1+|z|^2\right)^2}\right) \\ =& 4\left\{\frac{\partial}{\partial z^k}\left(\frac{\delta_{ij}}{\left(1+|z|^2\right)}\right) - \frac{\partial}{\partial z^k}\left(\frac{z_i\overline{z}_j}{\left(1+|z|^2\right)^2}\right)\right\} \\ =& 4\left\{-\frac{\delta_{ij}\overline{z}_k}{\left(1+|z|^2\right)^2} + 2\frac{z_i\overline{z}_j\overline{z}_k}{\left(1+|z|^2\right)^3} - \frac{\delta_{ik}\overline{z}_j}{\left(1+|z|^2\right)^2}\right\} \\ =& \frac{4}{\left(1+|z|^2\right)^3}\left\{2\,z_i\overline{z}_j\overline{z}_k - \left(1+|z|^2\right)\left[\delta_{ij}\overline{z}_k + \delta_{ik}\overline{z}_j\right]\right\}, \end{aligned} \tag{4.54}$$

and

$$g_{i\overline{j},\overline{k}} = \frac{4}{\left(1+|z|^2\right)^3}\Big\{2\, z_i\overline{z}_j z_k - \left(1+|z|^2\right)\left[\delta_{ij}z_k + \delta_{jk}z_i\right]\Big\}. \tag{4.55}$$

Equation (4.52) gives Christoffel symbols

$$\begin{aligned}\Gamma^l_{ik} =& g^{\overline{j}l} g_{i\overline{j},k} \\ =& \frac{1}{\left(1+|z|^2\right)^2}\Big\{\delta^{jl} + \overline{z}^j z^l\Big\}\Big\{2z_i\overline{z}_j\overline{z}_k - \left(1+|z|^2\right)\left[\delta_{ij}\overline{z}_k + \delta_{ik}\overline{z}_j\right]\Big\} \\ =& \frac{1}{\left(1+|z|^2\right)^2}\Big\{2z_i\overline{z}^l\overline{z}_k - \left(1+|z|^2\right)\left[\delta^l_i\overline{z}_k + \delta_{ik}\overline{z}^l\right] \\ &\qquad + 2\left(\overline{z}\cdot\overline{z}\right)z^l z_i\overline{z}_k - \left(1+|z|^2\right)\left[\overline{z}_i z^l\overline{z}_k + \left(\overline{z}\cdot\overline{z}\right)\delta_{ik}z^l\right]\Big\},\end{aligned} \tag{4.56}$$

and

$$\begin{aligned}\Gamma^{\overline{l}}_{i\overline{k}} = \frac{1}{\left(1+|z|^2\right)^2}\Big\{& 2\overline{z}_i z^l z_k - \left(1+|z|^2\right)\left[\delta^l_i z_k + \delta_{ik}z^l\right] \\ &+ 2\left(z\cdot z\right)\overline{z}^l\overline{z}_i z_k - \left(1+|z|^2\right)\left[z_i\overline{z}^l z_k + \left(z\cdot z\right)\delta_{ik}\overline{z}^l\right]\Big\}.\end{aligned} \tag{4.57}$$

The geodesic equations of motion (4.53) now become $\dot{z} = w$,

$$\begin{aligned}\dot{w}^l = \frac{1}{\left(1+|z|^2\right)^2}\Big\{& \Big[\left(1+|z|^2\right)\left(\overline{z}\cdot w\right)\Big]w^l \\ &+ \Big[\left(1+|z|^2\right)\left(w\cdot w\right) - 2\left(z\cdot w\right)\left(\overline{z}\cdot w\right)\Big]\overline{z}^l \\ &+ \Big[\left(1+|z|^2\right)\left[\left(\overline{z}\cdot w\right)\left(\overline{z}\cdot w\right) + \left(\overline{z}\cdot\overline{z}\right)\left(w\cdot w\right)\right] - 2\left(\overline{z}\cdot\overline{z}\right)\left(z\cdot w\right)\left(\overline{z}\cdot w\right)\Big]z^l\Big\},\end{aligned} \tag{4.58}$$

along with Hermitian conjugates $\dot{\overline{z}} = \overline{w}$ and

$$\dot{\overline{w}}^l = \frac{1}{\left(1+|z|^2\right)^2}\Bigg\{\Big[\left(1+|z|^2\right)(z\cdot\overline{w})\Big]\overline{w}^l + \Big[\left(1+|z|^2\right)(\overline{w}\cdot\overline{w}) - 2\,(\overline{z}\cdot\overline{w})\,(z\cdot\overline{w})\Big]z^l + \Big[\left(1+|z|^2\right)\left[(z\cdot\overline{w})\,(z\cdot\overline{w}) + (z\cdot z)\,(\overline{w}\cdot\overline{w})\right] - 2\,(z\cdot z)\,(\overline{z}\cdot\overline{w})\,(z\cdot\overline{w})\Big]\overline{z}^l\Bigg\}. \tag{4.59}$$

4.3 Mixed States

Based on the work of Bures[11], Uhlmann[45] obtained an explicit expression for the distance between two mixed–state density matrices, ρ_1 and ρ_2. This distance is given by

$$d_{\text{Bures}}\left(\rho_1,\rho_2\right) = \sqrt{2}\left\{1 - \text{Tr}\,\left[\left(\rho_1^{\frac{1}{2}}\rho_2\rho_1^{\frac{1}{2}}\right)^{\frac{1}{2}}\right]\right\}^{\frac{1}{2}}. \tag{4.60}$$

Note that, for two neighboring density matrices, ρ and $\rho + d\rho$, this distance coincides with the Riemannian metric on the space of density matrices (not just pure) obtained by Braunstein and Caves[10]. The finite distance described above is sufficient for purposes of this dissertation.

Chapter 5

A Dynamical Approach to Quantum Algorithms

5.1 Grover

Grover's algorithm was introduced and discussed at great length in chapter (3). There, it was mentioned that Grover's algorithm can be seen as a rotation of states in Hilbert space. This section will establish this result a little more rigorously and show how it is used in an analysis of the complexity of the algorithm.

Consider the discrete action of Grover's algorithm as described in the end of section (3.3). This will describe a search for *one*, state $|\omega\rangle$, out of a list of N states. At any given step in the process, the quantum database is in the state

$$|\Psi\rangle = k|\omega\rangle + \sum_{j\in\omega_\perp} l|j\rangle, \tag{5.1}$$

for some k and l, which, at say the i-th iteration, look like

$$\begin{aligned} k_{i+1} =& \frac{N-2}{N}k_i + \frac{2(N-1)}{N}l_i \\ l_{i+1} =& -\frac{2}{N}k_i + \frac{N-2}{N}l_i. \end{aligned} \tag{5.2}$$

Proposition 5.1.1. *The coefficients k and l in (5.2) obey*

$$\begin{aligned} k_i =& \sin\left[(2i+1)\,\theta\right] \\ l_i =& \frac{1}{\sqrt{N-1}} \cos\left[(2i+1)\,\theta\right] \end{aligned} \tag{5.3}$$

for $\sin\theta = \frac{1}{\sqrt{N}}$.

Proof. (By induction) Starting at step zero with an equally weighted superposition of all states

$$|\Psi_0\rangle = \frac{1}{\sqrt{N}}|\omega\rangle + \sum_{j\in\omega_\perp} \frac{1}{\sqrt{N}}|j\rangle \tag{5.4}$$

implies $k_0 = l_0 = \frac{1}{\sqrt{N}}$. Inserting $i = 0$ into (5.3), you get

$$k_0 = \sin(\theta) = \frac{1}{\sqrt{N}} \tag{5.5}$$

and

$$\begin{aligned} l_0 =& \frac{1}{\sqrt{N-1}} \cos(\theta) \\ =& \frac{1}{\sqrt{N-1}} \sqrt{1-\sin^2(\theta)} \\ =& \frac{1}{\sqrt{N-1}} \sqrt{1-\frac{1}{N}} = \frac{1}{\sqrt{N}}. \end{aligned} \tag{5.6}$$

So, the proposition holds for $i = 0$.

Now, consider the $(i+1)$–th iteration

$$\begin{cases} k_{i+1} = & \sin\left[(2(i+1)+1)\,\theta\right] \\ l_{i+1} = & \frac{1}{\sqrt{N-1}} \cos\left[(2(i+1)+1)\,\theta\right] \end{cases}$$

$$\begin{cases} k_{i+1} = & \sin\left[(2i+1)\,\theta + 2\theta\right] \\ l_{i+1} = & \frac{1}{\sqrt{N-1}} \cos\left[(2i+1) + 2\theta\right] \end{cases} \tag{5.7}$$

$$\begin{cases} k_{i+1} = & \sin\left[(2i+1)\,\theta\right]\cos(2\theta) + \cos\left[(2i+1)\,\theta\right]\sin(2\theta) \\ l_{i+1} = & \frac{1}{\sqrt{N-1}} \left\{\cos\left[(2i+1)\,\theta\right]\cos(2\theta) - \sin\left[(2i+1)\,\theta\right]\sin(2\theta)\right\}. \end{cases}$$

By the induction hypothesis,

$$\begin{cases} k_{i+1} = & k_i \cos(2\theta) + \sqrt{N-1}\, l_i \sin(2\theta) \\ l_{i+1} = & \frac{1}{\sqrt{N-1}} \left\{ \sqrt{N-1}\, l_i \cos(2\theta) - k_i \sin(2\theta) \right\}, \end{cases} \tag{5.8}$$

and since

$$\sin(\theta) = \frac{1}{\sqrt{N}} \quad \Rightarrow \quad \cos(2\theta) = \frac{N-2}{N} \quad \text{and} \quad \sin(2\theta) = \frac{2\sqrt{N-1}}{N}, \tag{5.9}$$

then

$$\begin{cases} k_{i+1} = & k_i \frac{N-2}{N} + l_i \frac{2(N-1)}{N} \\ l_{i+1} = & l_i \frac{N-2}{N} - k_i \frac{2}{N}. \end{cases} \tag{5.10}$$

Since this is (5.2) as expected, $j \Rightarrow j+1$. □

Now, the cyclical nature of this evolution means that the algorithm must be stopped manually when the probability for success will be high. Proposition 5.1.1 allows the "stopping time" for the algorithm to be easily estimated. The expressions for k and l in (5.3) take extreme values when

$$(2i+1)\theta = \frac{\pi}{2}, \tag{5.11}$$

and so, along with the fact that $\theta \sim \frac{1}{\sqrt{N}}$ for large N, the algorithm must be stopped in about $\lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ iterations. For a more detailed analysis of stopping time for Grover's algorithm (and multisearch too) see [28, 6]. It should also be noted that Grover's algorithm has been shown[3, 48] to be optimal. *i.e.*, there are no other search algorithms more efficient than Grover.

5.1.1 Grover is a Geodesic

Earlier in this chapter Grover's algorithm was expressed in terms of discrete recurrence relations. In the algorithm, the coefficients of the state

$$|\Psi\rangle = k|\omega\rangle + \sum_{j \in \omega_\perp} l|j\rangle \tag{5.12}$$

evolve according to

$$\begin{aligned} k_i &= \sin\left[(2i+1)\,\theta\right] \\ l_i &= \frac{1}{\sqrt{N-1}}\cos\left[(2i+1)\,\theta\right]. \end{aligned} \tag{5.13}$$

This evolution can be viewed as a discrete path in state space as shown in figure (5.1).

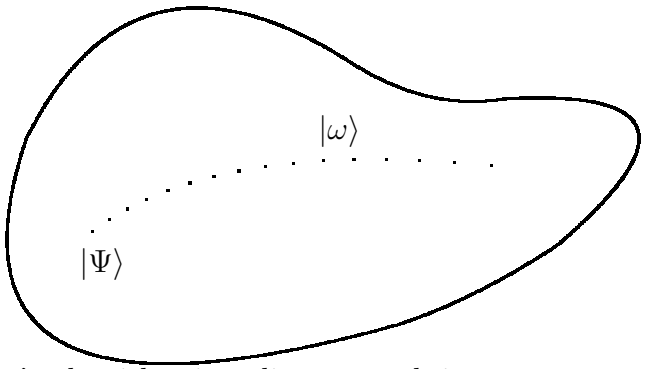

Figure 5.1: Grover's algorithm is a discrete path in state space starting from an equally–weighted superposition $|\Psi\rangle$ and continuing on through the desired state $|\omega\rangle$. Note that the algorithm keeps going and must be stopped at the anticipated running time (When the algorithm will be near the desired state).

Now, consider the continuous path on the space of states given by

$$\gamma\colon \mathbb{R} \to \mathbb{C}P^N\colon t \mapsto \begin{pmatrix} \sin(t) \\ \frac{1}{\sqrt{N-1}}\cos(t) \\ \vdots \\ \frac{1}{\sqrt{N-1}}\cos(t) \end{pmatrix}. \tag{5.14}$$

The discrete points generated by Grover's algorithm *lie upon this path*. This path can then be used to imbed Grover's algorithm (figure 5.2).

Proposition 5.1.2. *The path γ, as defined in (5.14), is a geodesic of the Fubini–Study metric on $\mathbb{C}P^N$.*

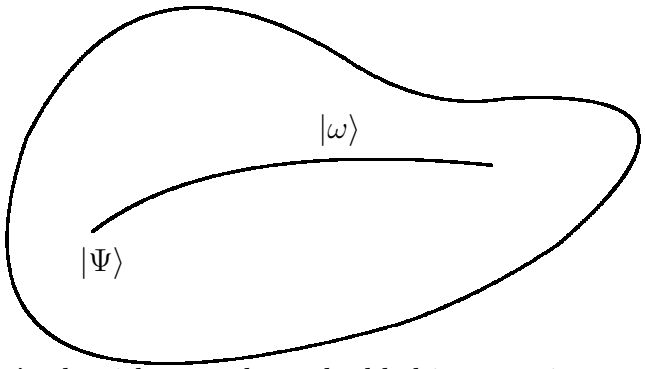

Figure 5.2: Grover's algorithm can be embedded in a continuous path in state space. Note that this path is a geodesic of the Fubini–Study metric.

Proof. In inhomogeneous coordinates, γ becomes

$$z_i = \frac{1}{\sqrt{N-1}} \cot(t) \tag{5.15}$$

for all i. A little sweat will show that these satisfy the equations of motion (4.58) for a geodesic of the Fubini–Study metric on $\mathbb{C}P^N$. □

Note that γ coincides with the large database limit of Grover's algorithm. This can easily be seen from the large N limit of (5.3).

5.1.2 Is there a Hamiltonian?

The diffusion transformation in Grover's algorithm, as defined in section (3.3.1), looks like

$$\mathbf{U_s} = \begin{pmatrix} \frac{2-N}{N} & \frac{2}{N} & \cdots & \frac{2}{N} \\ \frac{2}{N} & \frac{2-N}{N} & & \\ \vdots & & \ddots & \vdots \\ \frac{2}{N} & \cdots & & \frac{2-N}{N} \end{pmatrix}. \tag{5.16}$$

With a little work, this can be rewritten as

$$\mathbf{U_s} = -\exp\left\{ i\frac{\pi}{N} \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \right\}. \tag{5.17}$$

For the purpose of argument, assume that the location (index) of the target state is known ahead of time. In this case, the phase shift in Grover's algorithm (3.46) can be written as

$$\mathbf{U}_\omega = \exp\left\{ i\pi \begin{pmatrix} 0 & & & \\ & \ddots & & \\ & & 1 & \\ & & & \ddots \\ & & & & 0 \end{pmatrix} \right\}, \tag{5.18}$$

and so the entire Grover iteration can be written

$$\begin{aligned} \mathbf{U}_{\text{Grover}} =& \mathbf{U}_s \mathbf{U}_\omega \\ =& -\exp\left\{ i\frac{\pi}{N} \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \right\} \exp\left\{ i\pi \begin{pmatrix} 0 & & & \\ & \ddots & & \\ & & 1 & \\ & & & \ddots \\ & & & & 0 \end{pmatrix} \right\}. \end{aligned} \tag{5.19}$$

This should help in finding a Hamiltonian that generates the Grover evolution. This might prove useful in an analytical stability analysis of Grover's algorithm, but this has yet to be worked out.

5.2 Dynamical Stability

The primary model used in this dissertation describes Grover's algorithm as a geodesic of the Fubini–Study metric as discussed in the previous section. A

numerical model of the algorithm is developed to determine how the algorithm behaves in the presence of noise.

5.2.1 A Numerical Model of Grover's Algorithm

A database consisting of $N = 2^n$ items, is represented (using n qubits) as an N–state system whose states are given as points in $\mathbb{C}P^{N-1}$. The evolution of the algorithm is then modelled by the numerical integration of the geodesic equations of motion (4.58) on this space. Appropriate initial conditions are supplied to mimic the behaviour of the actual algorithm, which, at this stage, involves only pure states. The unitary evolution of the algorithm simply remains on the shell of pure states.

5.2.2 Noise

The time evolution of a quantum mechanical system is most easily described by unitary evolution. Unfortunately (fortunately?), physical systems do not exist in isolation. There are always interactions with either the environment or other physical systems that cause a quantum mechanical system to evolve by possibly more complicated means than these simple unitary transformations.

A somewhat more general approach to quantum evolution is to consider dynamical evolution of a quantum mechanical system described as simply a map, a *dynamical map*[25] of density matrices to density matrices. Notice that an even more general approach is to look at maps of density matrices, where the image itself doesn't necessarily have to be a physical density matrix. It could be simply a subsystem of a physical system, and hence it's trace could be less than one. For purposes of this dissertation, only dynamical maps that

take physical density matrices to physical density matrices are considered.

The numerical simulation of Grover's algorithm described in the previous section integrates the geodesic equations of motion. This can describe not only the evolution of pure states, but more generally, the evolution of density matrices. At every time step, this numerical integration can then be viewed as simply a dynamical map taking density matrices to density matrices. [1]

Now, any noise introduced into the dynamical evolution of a quantum mechanical system must be described within the context of dynamical maps. This approach encompasses any kind of noise that can be introduced, regardless of the source or if the noise is Hamiltonian or non-Hamiltonian.

In order to faithfully bring noise into the numerical model of Grover's algorithm, mixed states must be represented in that model (see section B.1.3). Mixed state density matrices can be written as

$$\rho = \sum \lambda_i \rho_i, \tag{5.20}$$

where the ρ_i are pure state density matrices, and i can run to the dimension of the component pure states ($N = 2^{\text{Number of Qubits}}$ for the present model).

Without noise, the algorithm happily evolves along the shell of pure states. When noise is added in, the state will be taken off of the shell of pure states and "in" via a slight mixture of other pure states. These states are chosen orthogonal to the direction of the target state in the search, which result in the state moving in a physically realistic (inward) direction throughout the simulation.

[1] Really, this particular mapping maps derivatives as well, and can be thought of as a mapping of the state in a quantum phase space.

The simulation simultaneously runs two states that start on the shell as pure states... one with noise, one without. At several times during the run, the Bures distance between the two states can be calculated (A.3), and the probability of measuring the target state for each displayed (figure 6.1). Data for several runs can be collected and compiled to give information about the stability of the algorithm.

Chapter 6

Conclusions

In discussing results, the first thing to note is that a quantum computer sure would have been useful in actually performing this simulation. As Feynman noted[21], simulating a quantum system by a classical computer requires an exponentially large amount of overhead. This was clearly evident in this research.

Decoherence and the experimental control and manipulation of different physical devices introduce noise into a computation. How much noise can the computation withstand before it fails? Exactly how does a computation fail in the presence of various types of noise? Any means of getting a handle on the stability of these algorithms warrants attention.

The current work uses a numerical simulation of a dynamical description of Grover's quantum search algorithm to determine the answers to some of these questions.

6.1 The Effects of Noise on Grover's Algorithm

Noise does not slow the algorithm down. It only reduces the probability that the algorithm will succeed. This is shown in figure 6.1.

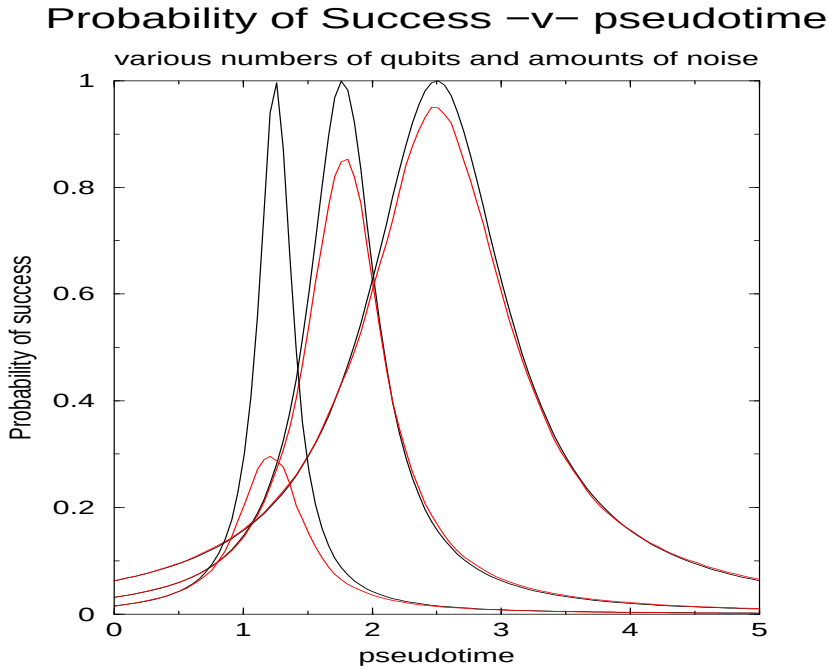

Figure 6.1: Probability of success -v- time for different noise levels and different numbers of qubits. Black is a run without noise, red is the same run with noise. Notice that where the peaks occur in "time" do not change due to noise. The relative times of the peaks are artificial in this graph.

Define the maximum amount of noise that the algorithm can tolerate as the point at which the probability of success falls below $\frac{1}{2}$. For n qubits, this maximum amount of tolerable noise falls off as $\frac{1}{n^3}$, as shown in figure 6.2.

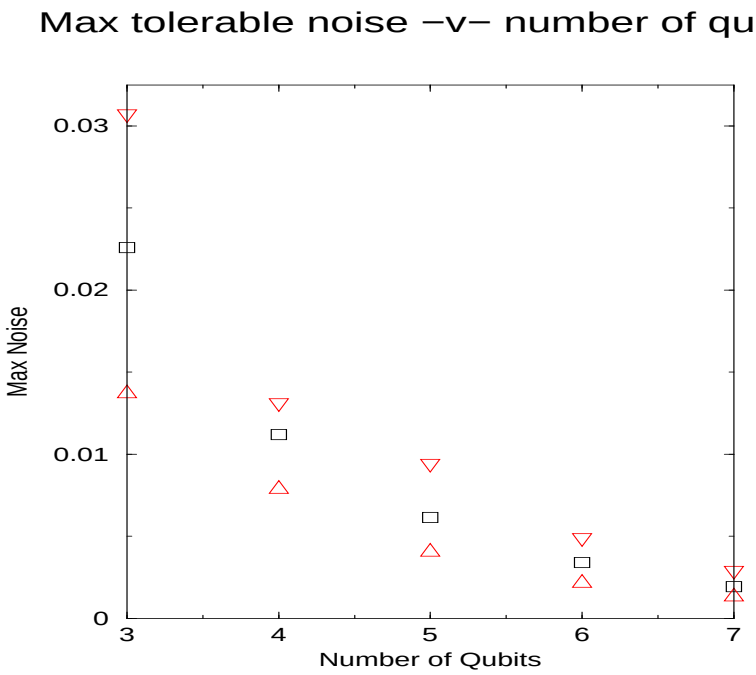

Figure 6.2: Maximum tolerable noise as a function of the number of qubits. This curve is approximately fit by $y = 0.5812x^{-2.886}$.

Appendix A contains:

- Bures-v-time for different noise levels and different numbers of qubits.
- Maximum probability of success -v- noise level for given number of qubits.
- Maximum probability of success -v- number of qubits for given noise levels.

6.2 Topics for Further Investigation

1. How does quantum error correction help? Consider, for instance, digitized sound stored on an audio compact disk. A huge majority of the information stored on such a CD consists of coding and methods to prevent errors in digital data from destroying the audio content of the CD. Little of this information would be necessary without noise. Well, a similar kind of redundancy can be built around quantum information. Quantum errors can be corrected.

   My results determine how much noise the bare algorithm can tolerate. Experimentalists would like to know to what tolerances they need to control candidate technologies for qubits. A more interesting result for them, the so–called bottom line, would be how much noise, not the bare algorithm, but the algorithm *with error correction* can tolerate.

   It also may be interesting to determine explicitly how device noise corresponds to "bit flips" in the quantum error correction literature.

2. How can decoherence–free subspaces help? Little is known about the geometry of the space of states of a quantum system. Investigation into this geometry yields decoherence–free regions of state[30]. Can this be harnessed by, not necessarily exact, but more robust versions of certain quantum algorithms?

3. Can a Hamiltonian be produced for Grover's algorithm? Can this be used for a more analytical stability analysis? Can error correction be added to this?

4. Can other algorithms such as Shor's be treated dynamically?

Appendices

Appendix A

The Data

Here, the data is presented as a series of figures organized into sections depending upon the information displayed.

Uncertainties in the numerical results are due almost exclusively to roundoff error. The computer is limited in the amount of space used to approximate a real number, and this limitation causes rounding to occur at every step of the calculation. These uncertainties are difficult to explicitly propagate through the calculations, but a good approximation of roundoff errors can be obtained by adjusting the resolution of integration. The limits of uncertainty in the following figures were obtained by doubling/halving the step size while simultaneously halving/doubling the number of steps taken.

A.1 Probability of success -v- maximum noise level for various numbers of qubits

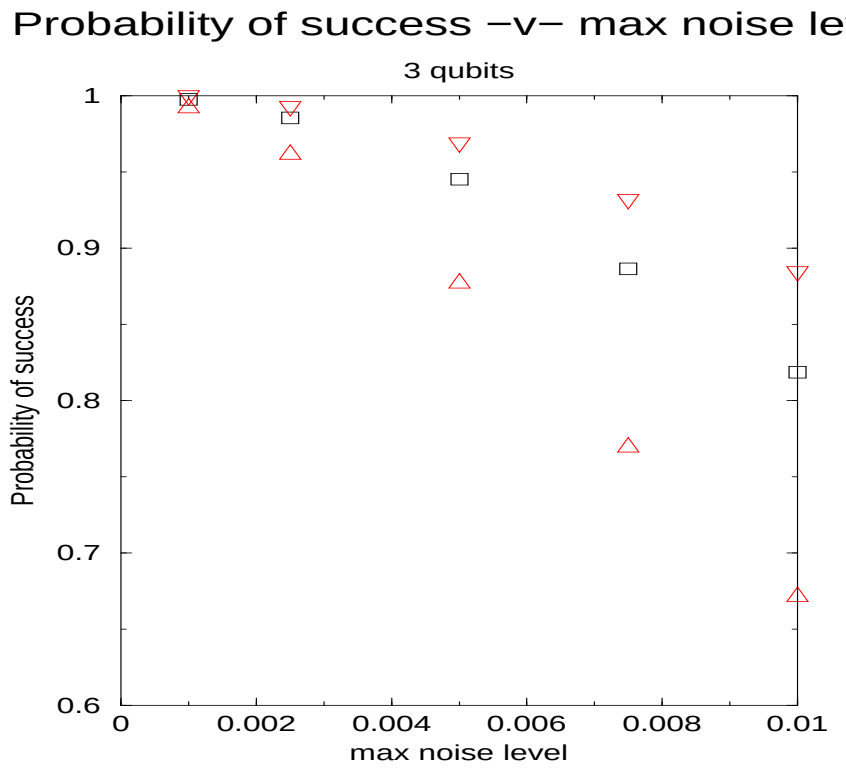

Figure A.1: Probability of sucess plotted as a function of the maximum noise allowed. This is done for 3 qubits.

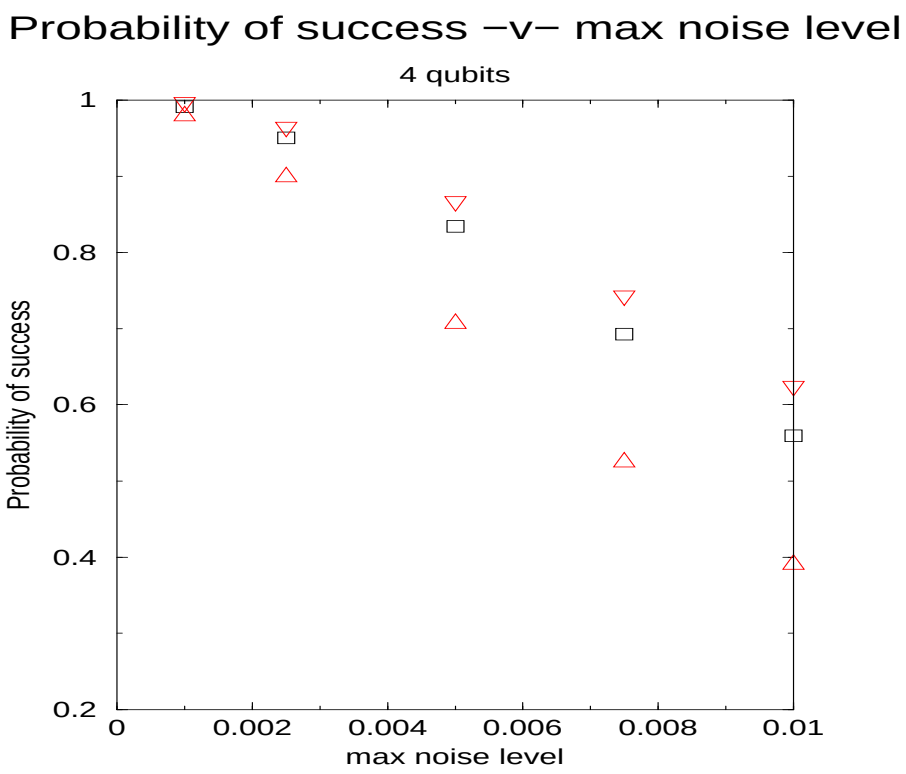

Figure A.2: Probability of sucess plotted as a function of the maximum noise allowed. This is done for 4 qubits.

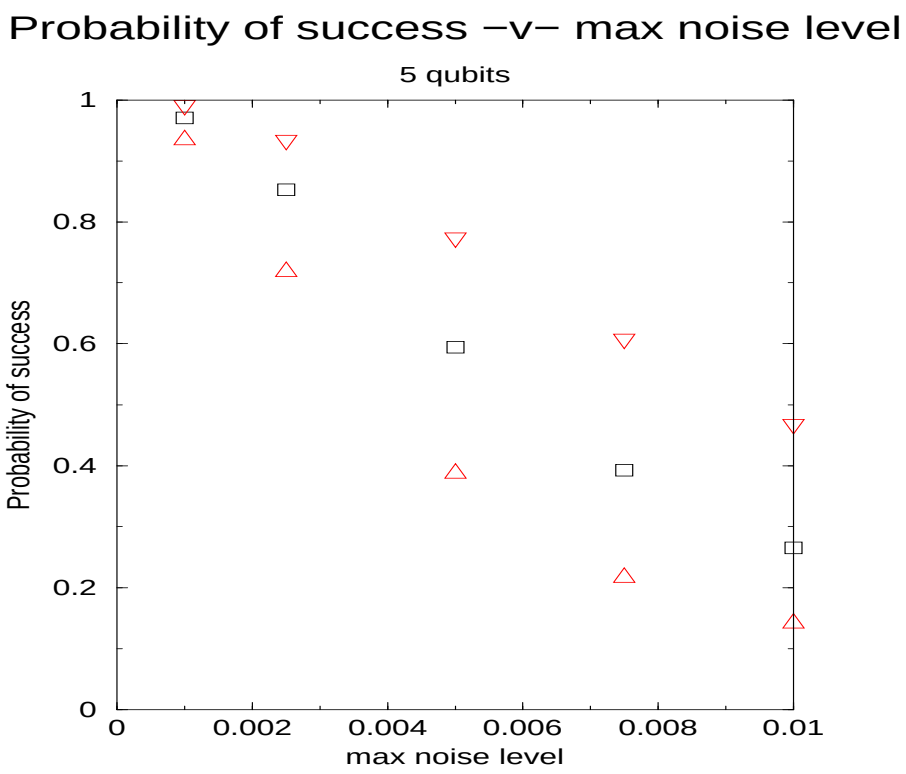

Figure A.3: Probability of sucess plotted as a function of the maximum noise allowed. This is done for 5 qubits.

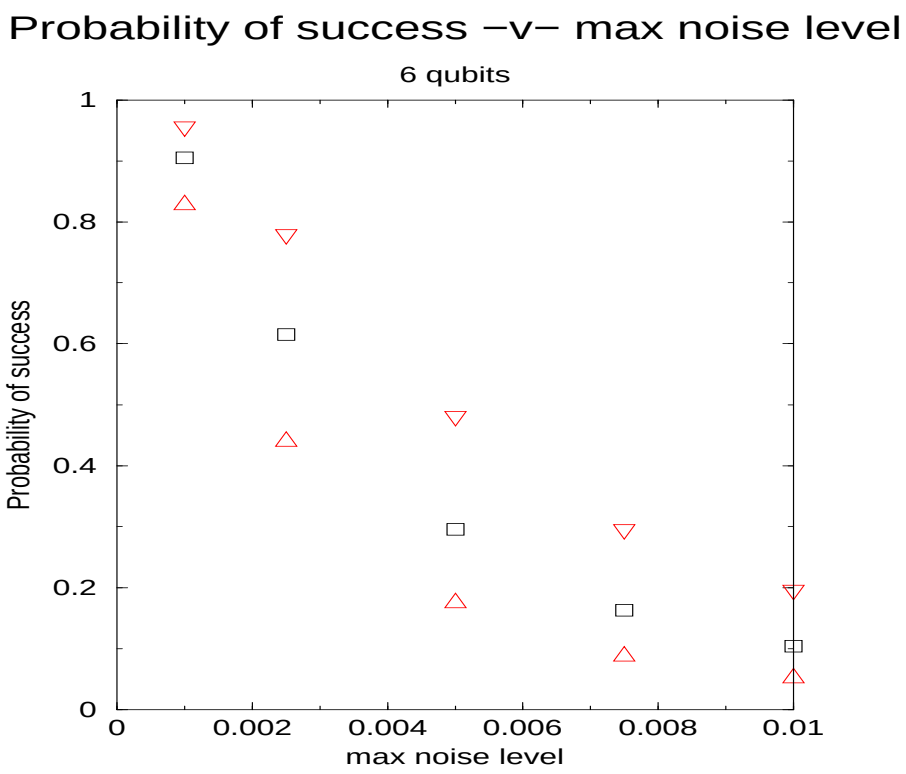

Figure A.4: Probability of sucess plotted as a function of the maximum noise allowed. This is done for 6 qubits.

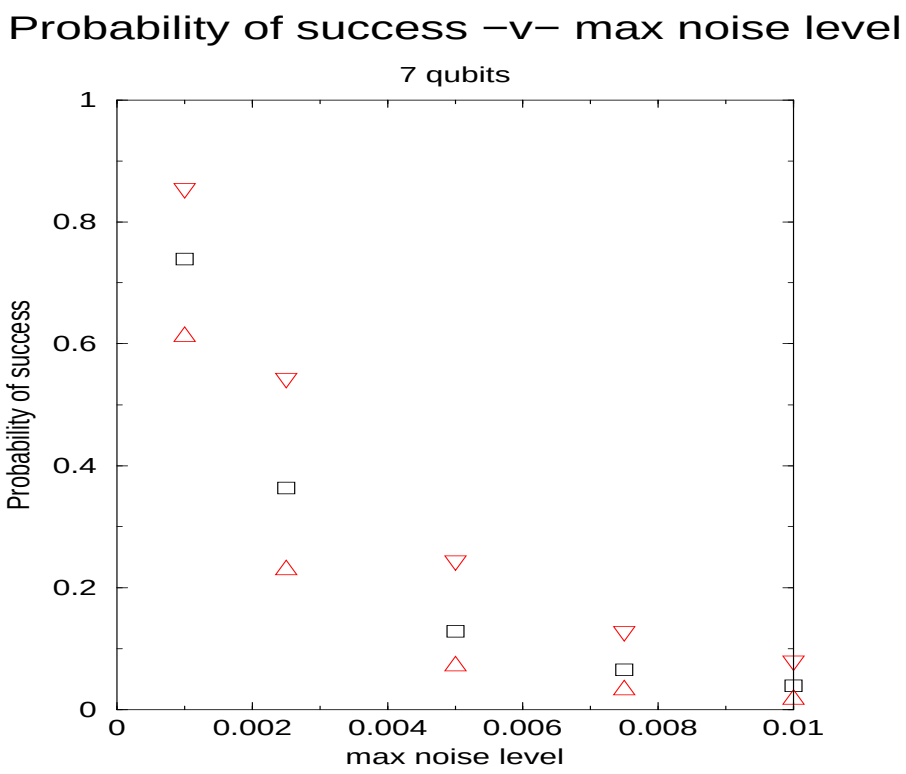

Figure A.5: Probability of sucess plotted as a function of the maximum noise allowed. This is done for 7 qubits.

A.2 Probability of success -v- number of qubits for various amounts of noise

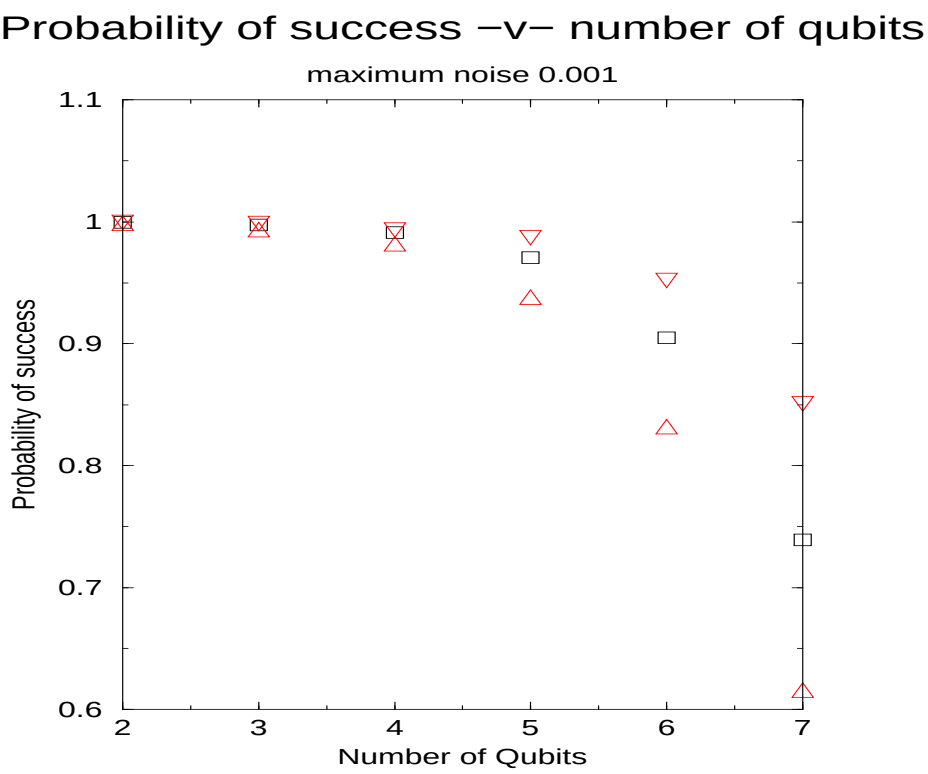

Figure A.6: Probability of sucess plotted as a function of the number of qubits. This is done for a noise threshold of 0.001.

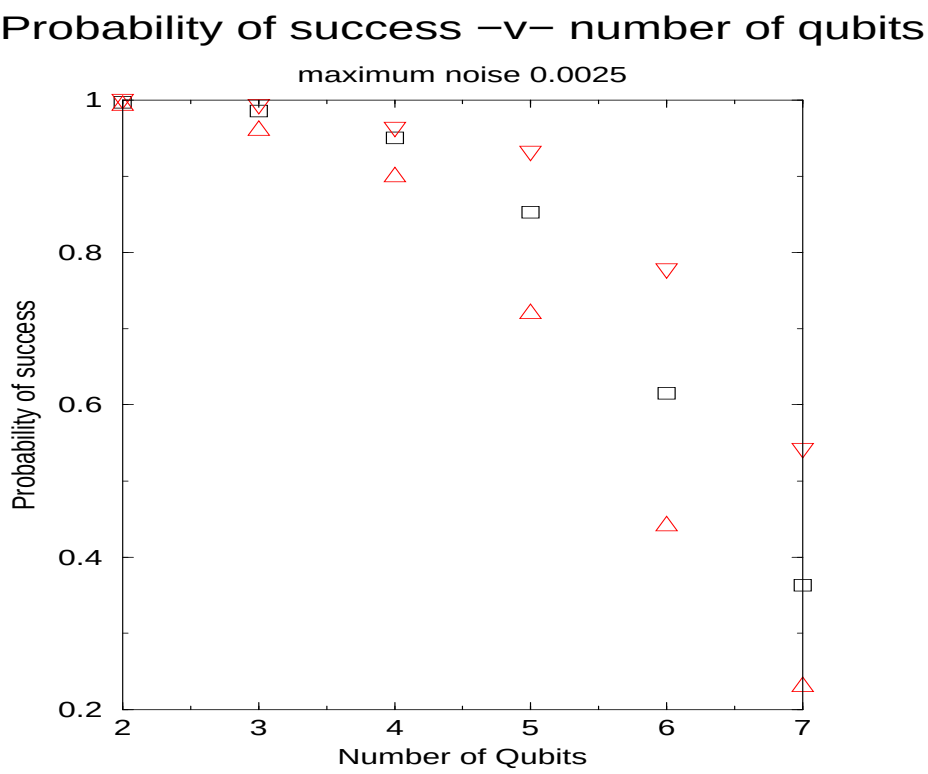

Figure A.7: Probability of sucess plotted as a function of the number of qubits. This is done for a noise threshold of 0.0025.

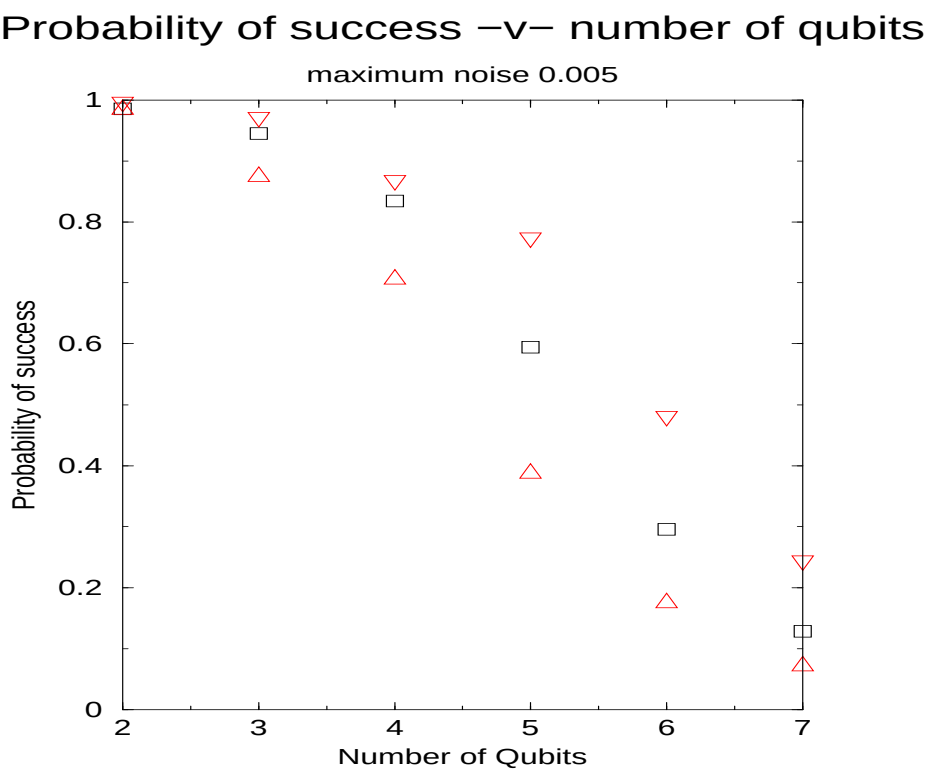

Figure A.8: Probability of sucess plotted as a function of the number of qubits. This is done for a noise threshold of 0.005.

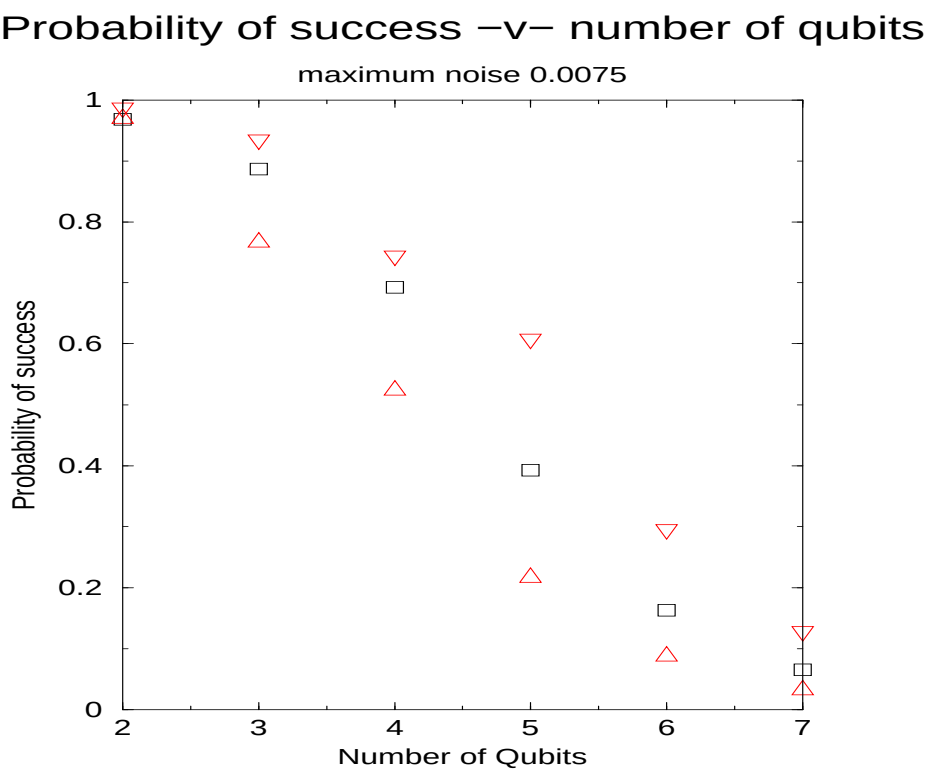

Figure A.9: Probability of sucess plotted as a function of the number of qubits. This is done for a noise threshold of 0.0075.

A.3 Bures distance -v- pseudotime for various numbers of qubits and amounts of noise

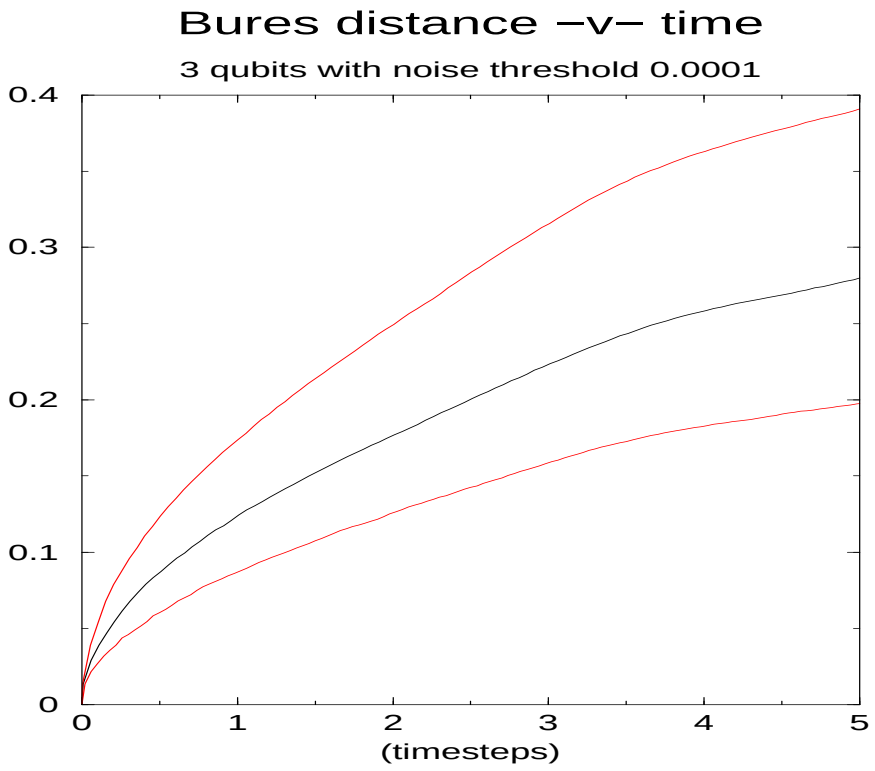

Figure A.10: Bures distance as a function of time for 3 qubits with noise threshold of 0.0001.

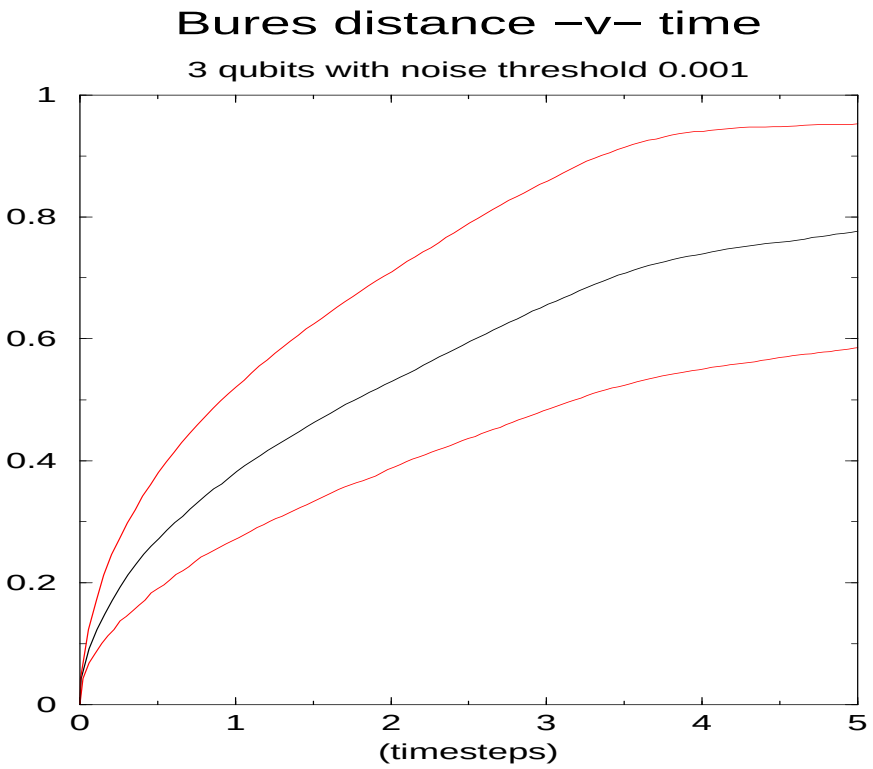

Figure A.11: Bures distance as a function of time for 3 qubits with noise threshold of 0.001.

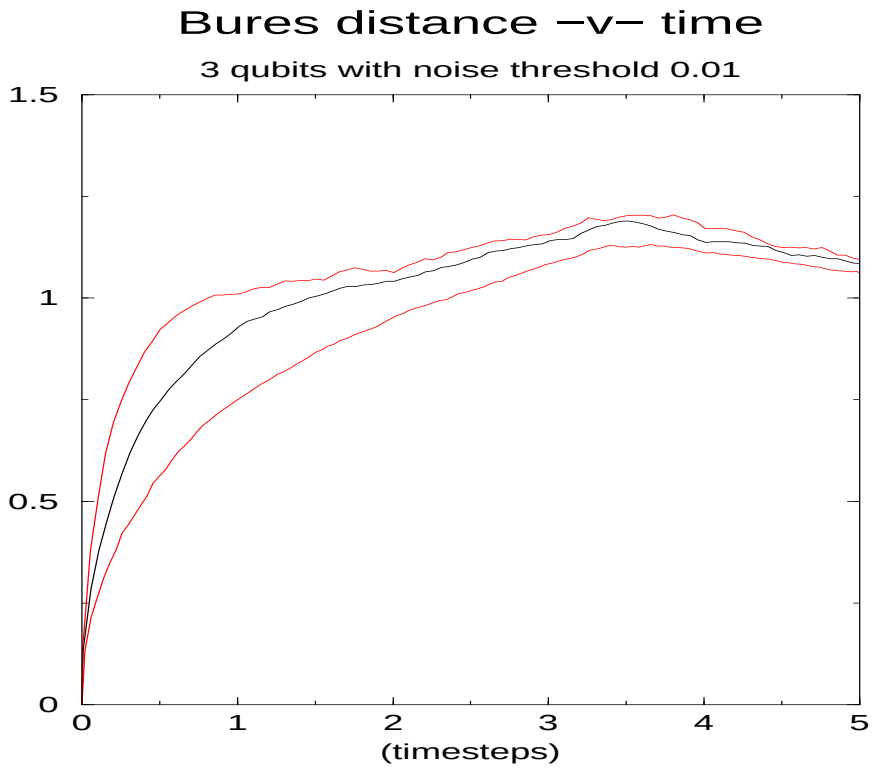

Figure A.12: Bures distance as a function of time for 3 qubits with noise threshold of 0.01.

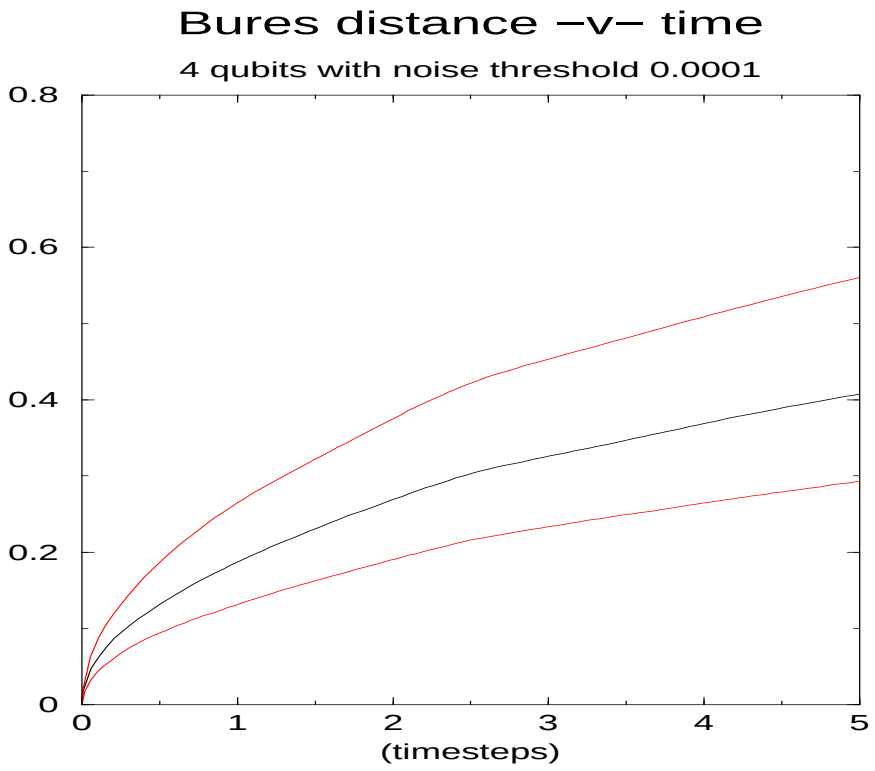

Figure A.13: Bures distance as a function of time for 4 qubits with noise threshold of 0.0001.

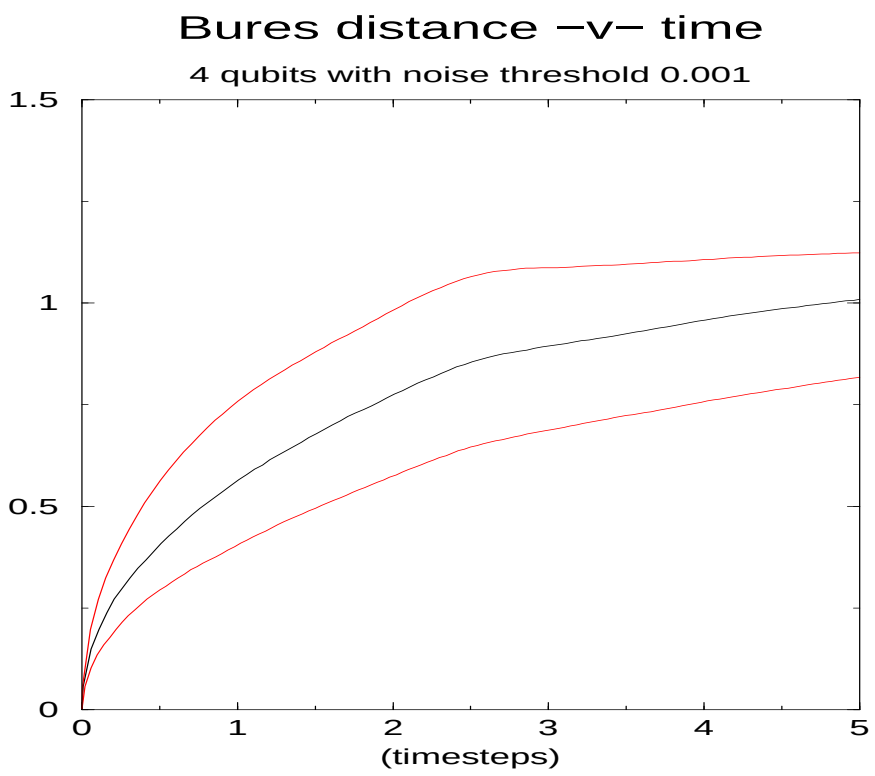

Figure A.14: Bures distance as a function of time for 4 qubits with noise threshold of 0.001.

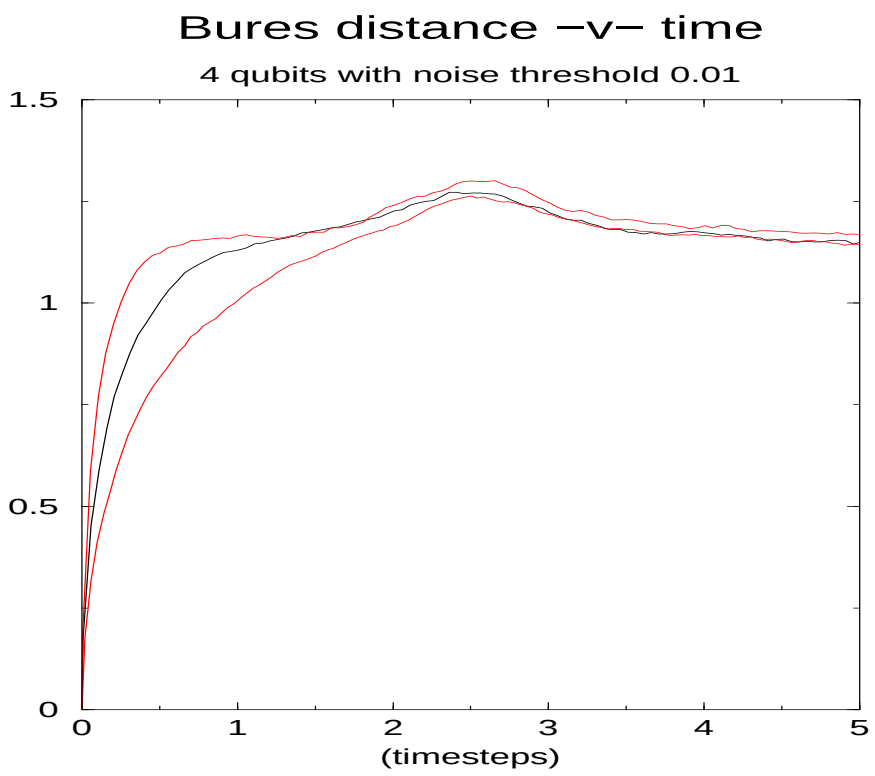

Figure A.15: Bures distance as a function of time for 4 qubits with noise threshold of 0.01.

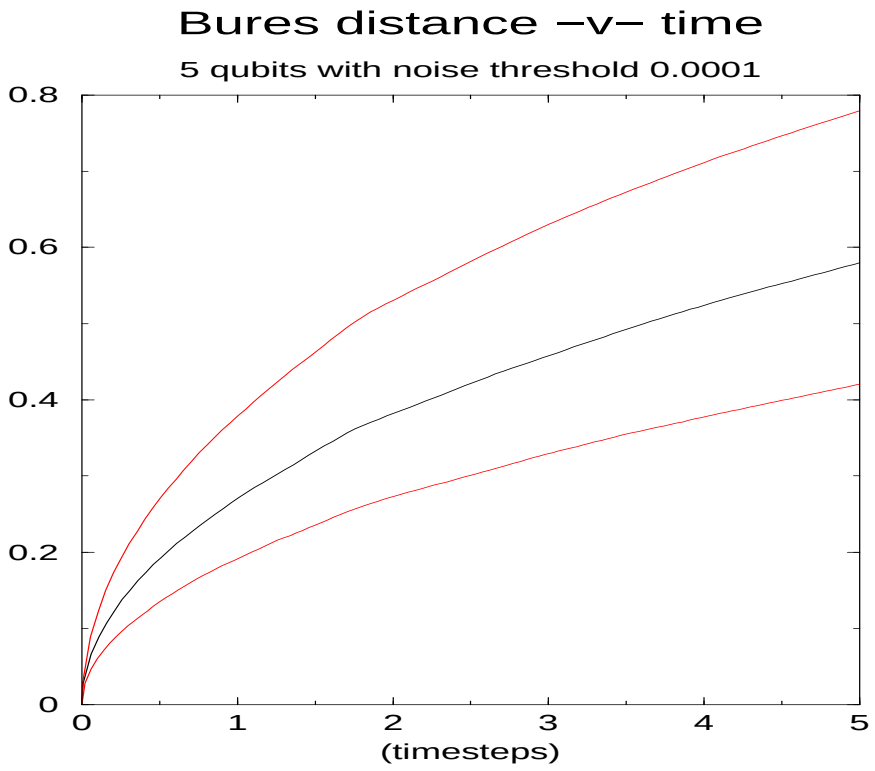

Figure A.16: Bures distance as a function of time for 5 qubits with noise threshold of 0.0001.

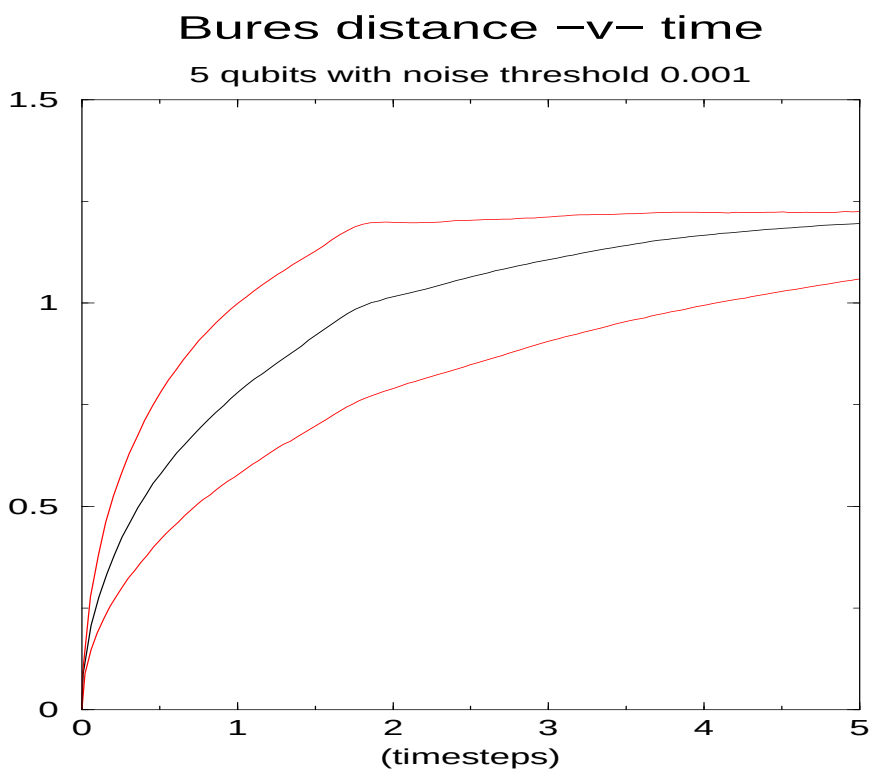

Figure A.17: Bures distance as a function of time for 5 qubits with noise threshold of 0.001.

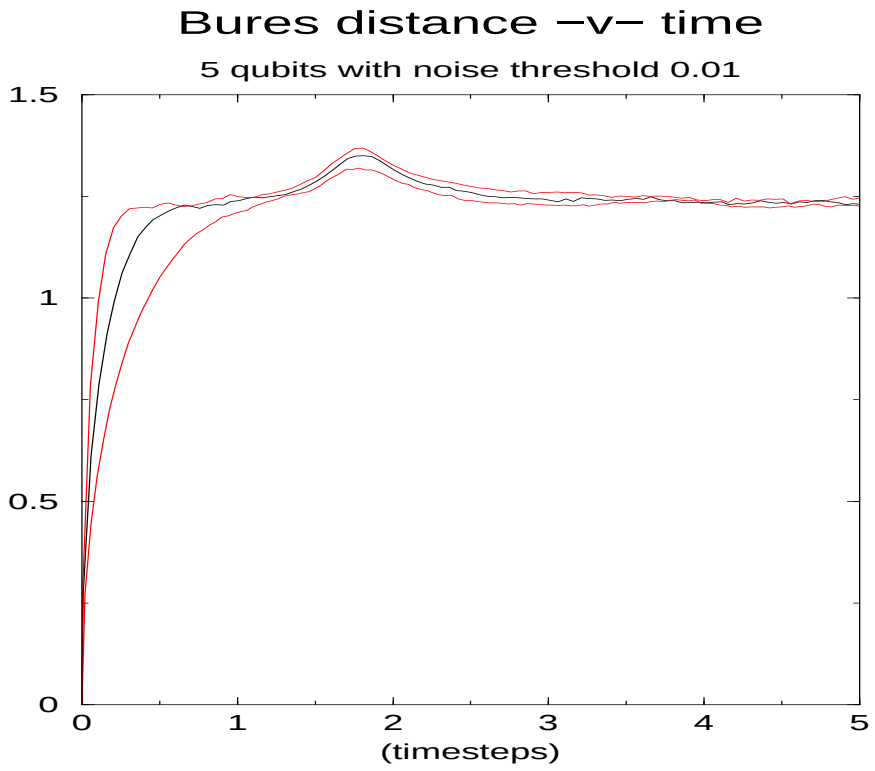

Figure A.18: Bures distance as a function of time for 5 qubits with noise threshold of 0.01.

Appendix B

The Code

This chapter contains details of the computer program used in the numerical simulation of Grover's algorithm. The code itself is interesting and possibly useful in other quantum simulations due to the general nature of the representations of quantum states.

This code is written in C++ and compiled with development snapshots of the GNU C++ compiler (`http://gcc.gnu.org/`). Any compiler compliant with the ANSI ISO C++ standard *should* compile the code, but I would read about how the compiler implements and instantiates templates before I tried it. The makeline used included

```
CFLAGS = -ansi -pedantic -Wall -O -gstabs+ -D_REENTRANT
CFLAGS += -D_GNU_SOURCE -ftemplate-depth-17 -funroll-loops
CFLAGS += -fstrict-aliasing
LIBS = -llapack -lblas -lg2c -lm
```

B.1 Classes Used

B.1.1 A General Quantum State

```
// State.h

#ifndef _STATE_H_
#define _STATE_H_

#include <complex>

#include <tnt/tnt.h>
#include <tnt/cmat.h>
#include <tnt/fmat.h>                                                    10
```

```
using namespace TNT;

#include <random/uniform.h>
using namespace ranlib;

class State {
  public:
    State(const int dim):
        _dimension(dim) {};
    virtual ~State() {};

    //accessors
    virtual Matrix<complex<double> > matrix( void ) const = 0;
    virtual void print( const double time ) const = 0;

    //mutators
    virtual void init( void ) = 0;
    virtual void step( const double time, const double stepSize ) = 0;
    virtual void perturb( Uniform<double>& generator,
                          const double upperBound ) = 0;
  protected:
    int _dimension;// of Hilbert space (_not_ projective Hilbert space!)
};

#endif //_STATE_H_
```

B.1.2 A Pure Quantum State

```
// PureState.h

#ifndef _PURESTATE_H_
#define _PURESTATE_H_

#include "myvalarray.h"
#include "State.h"

class PureState: public State {
  public:
    PureState( const int dim ):
        State(dim), _data( double(0), 4*(dim-1) ) {};
    virtual ~PureState() {};

    //accessors
    virtual Matrix<complex<double> > matrix( void ) const;
    virtual void print( const double time ) const ;
```

```
    //mutators
    virtual void init( void );
    virtual void init( const valarray<double>& z_i,
                       const valarray<double>& w_i,
                       const valarray<double>& zbar_i,
                       const valarray<double>& wbar_i );
    virtual void step( const double time, const double stepSize );
    virtual void perturb( Uniform<double>& generator, const double upperBound );

//private:
    valarray<double> _data;
};

#endif //_PURESTATE_H_
```

B.1.3 A Mixed Quantum State

```
// MixedState.h

#ifndef _MIXEDSTATE_H_
#define _MIXEDSTATE_H_

#include <vector>
#include <valarray>

#include "PureState.h"

class MixedState: public State {
  public:
    MixedState( const int dim ):
        State(dim), _lambda(0.0,dim) {};
    virtual ~MixedState();

    //accessors
    virtual Matrix<complex<double> > matrix( void ) const;
    virtual void print( const double time ) const ;

    //mutators
    virtual void init( void );
    virtual void step( const double time, const double stepSize );
    virtual void perturb( Uniform<double>& generator, const double upperBound );
  private:
    valarray<double> _lambda;
    vector<PureState*> _pureStates;
};
```

30

#endif //_MIXEDSTATE_H_

B.2 Libraries Used

- The ANSI/ISO standard C++ library includes template support, which is used heavily throughout this program. In particular, the `valarray` template combines the ease of using a template–based container with the numerical speed of a simple C array. The implementation for templates within the g++ compiler suite is still under development, so a heavy regimen of testing is recommended, paricularly with templates for expression evaluation. For more information about templates in C++ see[42], and for more information on how these are implemented by the GNU compilers check out `http://gcc.gnu.org/`.
- The Template Numerical Toolkit `TNT` Provides fairly functional matrix interfaces and compiles quickly despite being entirely template based. More on this alleged successor to things like `lapack++` can be found at `http://math.nist.gov/tnt`.
- The `blitz++` toolkit seems like it will be a great toolkit when completed. I simply used the component of the library that generates random numbers, preferring `TNT` for matrix work. This was done for two reasons, the blitz library is all templates and took *forever* to compile even the simplest things under the GNU compiler, and `TNT` provided examples for interfacing with `LAPACK` routines, where `blitx++` did not. More information about this library can be found at `http://www.oonumerics.org`.
- The ubiquitous Fortran library `LAPACK` was used for the rather droll task of diagonalizing density matrices. One of my primary reasons for including code in this dissertation is to provide examples of calling these legacy library routines from C++. The module `fwrap.C` is example code from the `TNT` slightly adapted to do eigenvalue and eigenvector problems for double precision complex Hermitian matrices, and is included below.

B.3 Modules

B.3.1 main

```
//
#include <string>
#include <fstream>
#include <unistd.h> // getopt
#include <cmath> // pow
#include <time.h> // time

#include "PureState.h"
#include "MixedState.h"
#include "display.h"


void usage( void ) {
    cout << "Usage: stepper [options]" << endl
         << "Where options are:" << endl
         << "-d prob-datafile " << endl
         << "-n num-steps" << endl
         << "-o bures-datafile" << endl
         << "-q num-qubits" << endl
         << "-s step-size" << endl
         << "-u noise-threshold" << endl;
    exit(1);
}

int main( int argc, char* argv[] ) {


    int numQubits = 4;
    int numSteps = 500;
    double stepSize = 0.0001;
    string outFileA = "output/Bures.out";
    string outFileB = "output/targetCoeff.out";
    double upperBound = 0.005; // biggest noise can get... kinda
                               // works between 0.001 and 0.01
    int opt = 0;
    while ( (opt = getopt( argc, argv, "d:n:o:q:s:u:" )) != -1 ) {
        switch ( opt ) {
        case 'd':
            outFileB = optarg;
            break;
        case 'n':
            numSteps = atoi(optarg);
            break;
        case 'o':
            outFileA = optarg;
            break;
        case 'q':
            numQubits = atoi(optarg);
```

```
            break;
        case 's':
            stepSize = atof(optarg);
            break;
        case 'u':
            upperBound = atof(optarg);
            break;
        default:
            usage();
        }
    }

    const int dimension = int( pow( 2, numQubits ) );

    Uniform<double> uniformGenerator;
    //uniformGenerator.seed( static_cast<unsigned int>( time(0) ) );

    // outputfile stuff
    char fileAppend[5 + sizeof(int) + 2*sizeof(double)] = "";
    sprintf( fileAppend, "-%d-n%f-s%f", numQubits, upperBound, stepSize );
    outFileA += fileAppend;
    outFileB += fileAppend;
    ofstream outFileStreamA( outFileA.c_str() );
    ofstream outFileStreamB( outFileB.c_str() );
    if ( !outFileStreamA || !outFileStreamB ) {
        cerr << "Oops!" << endl;
        exit(1);
    }

    cout << "Bures distance -v- time to "
         << outFileA << endl;
    cout << "target state prob -v- time to "
         << outFileB << endl;
    cout << "numQubits = " << numQubits << endl;
    cout << "numSteps = " << numSteps << endl;
    cout << "step size = " << stepSize << endl;
    cout << "upper bound = " << upperBound << endl;


    // initial conditions
    double t = 0.0;
    State* rho1 = new PureState(dimension);
    rho1->init();
    State* rho2 = new MixedState(dimension);
    rho2->init();

    try{
#ifdef TELL_ME
```

```
//          rho1->print(t);
//          rho2->print(t);
        printDiffs(outFileStreamA,t,rho1,rho2);
        printLeadingEVals(outFileStreamB,t,rho1,rho2);
#endif //TELL_ME

        int aHundredth = numSteps/100;
        for ( int i = 0; i < numSteps; i++ ){


            rho1->step( t, stepSize );
            rho2->step( t, stepSize );
            rho2->perturb( uniformGenerator, upperBound );
            t += stepSize;

            if ( 0 == i%aHundredth ) {
                //rho1->print(t);
                //rho2->print(t);
                printLeadingEVals(outFileStreamB,t,rho1,rho2);
                printDiffs(outFileStreamA,t,rho1,rho2);
            }
#ifdef TELL_ME
            showProgress(i,numSteps,numQubits);
            if ( i == numSteps - 1 ) {
//                rho1->print(t);
//                rho2->print(t);
                printDiffs(outFileStreamA,t,rho1,rho2);
                printLeadingEVals(outFileStreamB,t,rho1,rho2);
            }
//            char line[80];
//            cin.getline(line,80);
#endif //TELL_ME

        }//end of for

    }//end of try
    catch( const exception& ex ) {
        cerr << "exception " << ex.what() << endl;
    }

    delete rho1;
    delete rho2;

    outFileStreamA.close();
    outFileStreamB.close();

}
```

B.3.2 PureState

```
// State.C
#include <stdexcept>
#include "rk4.h"
#include "Matrices.h"
#include "exceptions.h"

#include "PureState.h"

void PureState::init(void) {

    if ( _data.size() % 4 ) throw;
    const int n = _data.size() / 4;

    try {
        valarray<double> z(1.0/sqrt(_dimension), n);
        valarray<double> w(-10.0, n);
        //valarray<double> w(0.0, n);
        valarray<double> zbar(0.0, n);
        valarray<double> wbar(0.0, n);
        for( int i = 0; i<n; i++ ) {
            _data[i] = z[i];
            _data[n+i] = w[i];
            _data[2*n+i] = zbar[i];
            _data[3*n+i] = wbar[i];
        }
    }
    catch(out_of_range) {
        cerr << "oops" << endl;
        exit(1);
    }

}

void PureState::init( const valarray<double>& z_i,
                      const valarray<double>& w_i,
                      const valarray<double>& zbar_i,
                      const valarray<double>& wbar_i ) {

    if ( _data.size() % 4 ) throw;
    const int n = _data.size() / 4;
```

```
    try {
        for( int i = 0; i<n; i++ ) {
            _data[i] = z_i[i];
            _data[n+i] = w_i[i];
            _data[2*n+i] = zbar_i[i];
            _data[3*n+i] = wbar_i[i];
        }
    }
    catch(out_of_range) {
        cerr << "oops" << endl;
        exit(1);
    }

}

void PureState::step( const double time, const double stepSize ) {

    _data += stepRk4( time, _data, stepSize );

}

Matrix<complex<double> > PureState::matrix( void ) const {

    Matrix<complex<double> > rho(_dimension,_dimension,0.0);

    try {
        const int n = _data.size() / 4;
        valarray<complex<double> > states(0.0, _dimension);
        states[0] = complex<double>( 1.0/sqrt( _dimension ), 0.0 );
        for( int i = 0; i<n; i++ ) {
            states[i+1] = complex<double>( _data[i], _data[2*n+i] );
        }
        //normalize...
        //states /= sqrt(abs(
        //             static_cast<valarray<complex<double> > >(states) *
        //             static_cast<valarray<complex<double> > >(states.apply(conj))
        //           ));
        double norm = sqrt(abs(
                    static_cast<valarray<complex<double> > >(states) *
                    static_cast<valarray<complex<double> > >(states.apply(conj))
                  ));
        if ( norm < ZERO ) throw Fpe("in PS::matrix()");
        states /= norm;

        for (int i=0; i<_dimension; i++ ) {
            for (int j=0; j<_dimension; j++ ) {
                rho(i+1,j+1) = states[i]*conj(states[j]);
            }
```

```
        }
    }
    catch(out_of_range) {
        cerr << "oops" << endl;
        exit(1);
    }

    return rho;
}

void PureState::print( const double t ) const {

    Matrix<complex<double> > rho = matrix();

    cout << "Pure state density matrix is : " << rho << endl;

    complex<double> trace = 0.0;
    for (int i=1; i<_dimension+1; i++) {
        trace += rho(i,i);
    }
    cout << "with trace : " << abs(trace) << endl;

}

void PureState::perturb( Uniform<double>& generator, const double upperBound ) {

//    cout << "PureState random number = "
//         << ( generator.random() - 0.5 ) * upperBound
//         << endl;

    if ( _data.size() % 4 ) throw;
    const int n = _data.size() / 4;

    try {
        for( int i = 0; i<n; i++ ) {
            _data[i] += ( generator.random() - 0.5 ) * upperBound;
        }
        for( int i = 2*n; i<3*n; i++ ) {
            _data[i] += ( generator.random() - 0.5 ) * upperBound;
        }
    }
    catch(out_of_range) {
        cerr << "oops" << endl;
        throw;
    }

}
```

B.3.3 MixedState

```
// State.C
#include <stdexcept>

#include "rk4.h"
#include "Matrices.h"
#include "exceptions.h"

#include "MixedState.h"

MixedState::~MixedState() {
    for ( int i = 0; i< _dimension; i++ ) {
        delete _pureStates[i];
    }
}

void MixedState::init( void ) {

    _lambda[0] = 1.0;

    try{
        valarray<double> z(1.0/sqrt(_dimension), _dimension - 1);
        //valarray<double> w(0.0, _dimension - 1);
        valarray<double> w(-10.0, _dimension - 1);
        valarray<double> zbar(0.0, _dimension - 1);
        valarray<double> wbar(0.0, _dimension - 1);

        for ( int i = 0; i< _dimension; i++ ) {
            _pureStates.push_back( new PureState(_dimension) );
            _pureStates[i]->init(z,w,zbar,wbar);
        }

        // first pure state is equally--weighted superposition of everything
        //_pureStates[0]->init(z,w,zbar,wbar);

        valarray<double> base(0.0, _dimension - 1);
        //w = valarray<double>(0.1, _dimension-1);
        w = valarray<double>(1.0, _dimension-1);
        //w = valarray<double>(0.0, _dimension-1);
        for ( int i = 1; i< _dimension; i++ ) {
            base = 0.0;
            //base[i-1] = 1e4;
            base[i-1] = 1e8;
            _pureStates[i]->init(base,w,zbar,wbar);
```

```
        }

    }
    catch(...) {
        cerr << "oops in Mixed::init" << endl;
        throw("MS:init???");
    }                                                                    50
}

void MixedState::step( const double time, const double stepSize ) {

    for ( int i = 0; i< _dimension; i++ ) {
        _pureStates[i]->step( time, stepSize );
    }

}
                                                                         60
Matrix<complex<double> > MixedState::matrix( void ) const {

    Matrix<complex<double> > rho(_dimension,_dimension,0.0);

    for(int k=0; k<_dimension; k++) {

        const int n = _pureStates[k]->_data.size() / 4;
        valarray<complex<double> > states(0.0, _dimension);
        states[0] = complex<double>( 1.0/sqrt( _dimension ), 0.0 );
        for( int i = 0; i<n; i++ ) {                                     70
            states[i+1] = complex<double>( _pureStates[k]->_data[i],
                          _pureStates[k]->_data[2*n+i] );
        }
        //normalize...
        //states /= sqrt(abs(
        //             static_cast<valarray<complex<double> > >(states) *
        //             static_cast<valarray<complex<double> > >(states.apply(conj))
        //           ));
        double norm = sqrt(abs(
                    static_cast<valarray<complex<double> > >(states) *       80
                    static_cast<valarray<complex<double> > >(states.apply(conj))
                  ));
        if ( norm < ZERO ) throw Fpe("Norm too small in MixedState::matrix");
        states /= norm;

        Matrix<complex<double> > tmpMat(_dimension,_dimension,0.0);
        for (int i=0; i<_dimension; i++ ) {
            for (int j=0; j<_dimension; j++ ) {
                //tmpMat(i+1,j+1) = _lambda[k]*states[i]*conj(states[j]);
                complex<double> tmp = _lambda[k]*states[i]*conj(states[j]);  90
                tmpMat(i+1,j+1) = complex<double>(
```

```
                    ( std::abs(tmp.real()) > ZERO ? tmp.real(): 0.0 ),
                    ( std::abs(tmp.imag()) > ZERO ? tmp.imag(): 0.0 ));

            }
        }

        rho = rho + tmpMat;// no "+=" in TNT!

    }                                                                    100

    //renormalize
    //rho /= trace(rho); // not in TNT!!!
    const double tr = trace(rho);
    if ( tr < ZERO ) throw Fpe("trace is too small");
    for ( int i=0; i<_dimension; i++ ) {
        for ( int j=0; j<_dimension; j++ ) {
            rho[i][j] /= tr;
        }
    }                                                                    110

    if ( has_a_nan(rho) ) throw Fpe("From Mixed_State::matrix()");

    return rho;

}

void MixedState::print( const double t ) const {

    Matrix<complex<double> > rho = matrix();                             120

    cout << "Density matrix is : " << rho << endl;

//    complex<double> trace = 0.0;
//    for (int i=1; i<_dimension+1; i++) {
//        trace += rho(i,i);
//    }
//    cout << "with trace : " << abs(trace) << endl;
//
//    for (int i=0; i< _dimension; i++ ) {                               130
//        cout << "Pure state component i = " << i << endl;
//        cout << "lambda i = " << _lambda[i] << endl;
//        _pureStates[i]->print(t);
//    }
}

void MixedState::perturb( Uniform<double>& generator, const double upperBound ) {

    double lambdaSum = 0.0;
```

```
    for ( int i=0; i<_dimension; i++ ) {
        _pureStates[i]->perturb( generator, upperBound );
        _lambda[i] += generator.random() * 0.5 * upperBound;
        lambdaSum += _lambda[i];
    }

    // keep lambda's normalized...
    // should they be able to be negative? No!
    if ( lambdaSum < ZERO ) throw Fpe("lambda too small in MS::perturb");
    _lambda /= lambdaSum;

}
```

B.3.4 derivs

```
#define _POSIX_SOURCE 1
// derivs.C
//
#include <stdexcept>
#include "myvalarray.h"

#include "derivs.h"

// return dy/dx for each particular equation
valarray<double> dydx( const double x, const valarray<double>& y ) {

    if ( y.size() % 4 ) throw;

    const int n = y.size() / 4;


    valarray<double> tmpdydx(0.0,y.size());
    try {
        // slices would be easier,
//        slice_array<double>& z = y[slice(0,n-1,1)];
//        slice_array<double>& w = y[slice(n,2*n-1,1)];
//        slice_array<double>& zbar = y[slice(2*n,3*n-1,1)];
//        slice_array<double>& wbar = y[slice(3*n,4*n-1,1)];
        //but oh well...
        valarray<double> z(0.0, n);
        valarray<double> w(0.0, n);
        valarray<double> zbar(0.0, n);
        valarray<double> wbar(0.0, n);
        for( int i = 0; i<n; i++ ) {
            z[i] = y[i];
            w[i] = y[n+i];
```

```
    zbar[i] = y[2*n+i];
    wbar[i] = y[3*n+i];
}

valarray<double> zdot(w);

valarray<double> wdot(0.0, n);
// calculate dot products first...
double zdotzbar = z*zbar;                                                    40
double wdotw = w*w;
double zdotw = z*w;
double zbardotw = zbar*w;
double zbardotzbar = zbar*zbar;
for( int i = 0; i<n; i++ ) {
    wdot[i] =
        ( zbar[i] * (
            ( 1 + zdotzbar )*(wdotw) - 2*(zdotw)*(zbardotw)
          )
          + w[i] * (                                                         50
              ( 1 + zdotzbar )*(zbardotw)
            )
          + z[i] * (
              ( 1 + zdotzbar )*(
                  (zbardotw)*(zbardotw)
                  + (zbardotzbar)*(wdotw)
              )
              - 2*(zbardotzbar)*(zdotw)*(zbardotw)
            )
        ) / ( 1 + zdotzbar )*( 1 + zdotzbar );                               60
}

valarray<double> zbardot(wbar);

valarray<double> wbardot(0.0, n);
// calculate dot products first...
double wbardotwbar = wbar*wbar;
double zbardotwbar = zbar*wbar;
double zdotwbar = z*wbar;
double zdotz = z*z;                                                          70
for( int i = 0; i<n; i++ ) {
    wbardot[i] =
        ( z[i] * (
            ( 1 + zdotzbar )*(wbardotwbar) - 2*(zbardotwbar)*(zdotwbar)
          )
          + wbar[i] * (
              ( 1 + zdotzbar )*(zdotwbar)
            )
          + zbar[i] * (
```

```
                    ( 1 + zdotzbar )*(                                    80
                        (zdotwbar)*(zdotwbar)
                        + (zdotz)*(wbardotwbar)
                    )
                    - 2*(zdotz)*(zbardotwbar)*(zdotwbar)
                )
            ) / ( 1 + zdotzbar )*( 1 + zdotzbar );
        }

        // recombine into tmpdydx... again, slices would be easier
        for( int i = 0; i<n; i++ ) {                                      90
            tmpdydx[i] = zdot[i];
            tmpdydx[n+i] = wdot[i];
            tmpdydx[2*n+i] = zbardot[i];
            tmpdydx[3*n+i] = wbardot[i];
        }

    }
    catch(out_of_range) {
        cerr << "oops" << endl;
        exit(1);                                                          100
    }

    return tmpdydx;

}
```

B.3.5 rk4

```
#define _POSIX_SOURCE 1
// rk4.C

#include "derivs.h"
#include "rk4.h"


extern valarray<double>
stepRk4( const double x, const valarray<double>& y, const double stepSize ) {
                                                                          10
    valarray<double> k1 = stepSize * dydx( x, y );

    valarray<double> k2 = stepSize * dydx( x + stepSize/2.0, y + k1/2.0 );

    valarray<double> k3 = stepSize * dydx( x + stepSize/2.0, y + k2/2.0 );

    valarray<double> k4 = stepSize * dydx( x + stepSize, y + k3 );
```

```
    return k1/6.0 + k2/3.0 + k3/3.0 + k4/6.0;
}
```

B.3.6 Bures

```
// Bures.C
//
#include "Matrices.h"

#include "exceptions.h"

#include "Bures.h"

extern double distBures( const Matrix<complex<double> >& mat1,
                         const Matrix<complex<double> >& mat2 ) {

    //cout << "Bures distance called.. between " << mat1 << endl;
    //cout << "and " << mat2 << endl;

    double ret(0.0);
    try {

        Matrix<complex<double> > tmp1 = sqrt( mat1 );
        Matrix<complex<double> > tmp2 = sqrt( tmp1 * mat2 * tmp1 );
        double tmp = 1.0-trace(tmp2);
        ret = sqrt( 2.0 * std::abs(tmp) );
        if ( isnan( ret ) ) throw Fpe("From distBures");
    }
    catch ( const exception& ex ) {
        cerr << "Oops in distBures" << endl;
    }

    return ret;
}
```

B.3.7 Matrices

```
// Matrices.h

#ifndef _MATRICES_H_
#define _MATRICES_H_
```

```
#include <complex>

#include <tnt/tnt.h>
#include <tnt/cmat.h>
#include <tnt/fmat.h>

using namespace TNT;

#include <random/uniform.h>
using namespace ranlib;

extern Fortran_Matrix<complex<double> >
          dagger( const Fortran_Matrix<complex<double> >& mat );

extern Matrix<complex<double> >
          dagger( const Matrix<complex<double> >& mat );

extern const double trace( const Matrix<complex<double> >& mat );

extern const bool has_a_nan( const Matrix<complex<double> >& mat );

extern const Matrix<complex<double> >
                sqrt( const Matrix<complex<double> >& mat );

extern const Vector<double>
                eigVals( const Matrix<complex<double> >& mat );

extern const Matrix<complex<double> >
                randomMatrix( Uniform<double>& generator,
                              const int dimension,
                              const double upperBound );

const static double ZERO = 1.0e-18;

#endif // _MATRICES_H_
```

```
// Matrices.C
//
#include <iostream>
#include <tnt/fmat.h>

#include "fwrap.h"
#include "exceptions.h"

#include "Matrices.h"

const double trace( const Matrix<complex<double> >& mat ) {
```

```
    if( mat.num_rows() != mat.num_cols() ) throw("Matrices not square");

    double sum = 0.0;
    for( int i=0; i<mat.num_rows(); i++ ) {
        sum += abs(mat[i][i]);
    }

    return sum;

}

Fortran_Matrix<complex<double> >
nan_to_zero( const Fortran_Matrix<complex<double> >& mat ) {

    Fortran_Matrix<complex<double> > tmp = mat;
    for(int i=1; i<=mat.num_rows(); i++) {
        for(int j=1; j<=mat.num_rows(); j++) {
            tmp(i,j) = complex<double>(
                //( mat(i,j).real() < 1e-20 )? 0.0 : mat(i,j).real(),
                //( mat(i,j).imag() < 1e-20 )? 0.0 : mat(i,j).imag()
                ( isnan( mat(i,j).real() ) )? 0.0 : mat(i,j).real(),
                ( isnan( mat(i,j).imag() ) )? 0.0 : mat(i,j).imag()
                );
        }
    }

    return tmp;

}

Fortran_Matrix<complex<double> >
dagger( const Fortran_Matrix<complex<double> >& mat ) {

    Fortran_Matrix<complex<double> > tmp = mat;
    for(int i=1; i<=mat.num_rows(); i++) {
        for(int j=1; j<=mat.num_rows(); j++) {
            tmp(j,i) = conj( mat(i,j) );
        }
    }

    return tmp;

}

Matrix<complex<double> >
dagger( const Matrix<complex<double> >& mat ) {
```

```
    Matrix<complex<double> > tmp = mat;
    for(int i=0; i<mat.num_rows(); i++) {
        for(int j=0; j<mat.num_rows(); j++) {
            tmp[j][i] = conj( mat[i][j] );
        }
    }

    return tmp;

}

Fortran_Matrix<complex<double> >
toFortranMatrix( const Matrix<complex<double> >& mat ) {

    Fortran_Matrix<complex<double> > tmpMat( mat.num_rows(),
                                             mat.num_rows(),
                                             0.0 );
    for( int i=1; i<=mat.num_rows(); i++ ) {
        for( int j=1; j<=mat.num_rows(); j++ ) {
            tmpMat(i,j) = mat(i,j);
        }
    }
    return tmpMat;

}

Matrix<complex<double> >
toCMatrix( const Fortran_Matrix<complex<double> >& mat ) {

    Matrix<complex<double> > tmpMat( mat.num_rows(), mat.num_rows(), 0.0 );
    for( int i=1; i<=mat.num_rows(); i++ ) {
        for( int j=1; j<=mat.num_rows(); j++ ) {
            tmpMat(i,j) = mat(i,j);
        }
    }
    return tmpMat;

}

// Make a diagonal matrix out of a vector of eigenvalues
const Matrix<complex<double> > makeMatrix( const Vector<double>& eigVals ) {

    Matrix<complex<double> > tmp( eigVals.size(), eigVals.size(), 0.0 );
    for( int i=0; i < eigVals.size(); i++ ) {
        tmp[i][i] = eigVals[i];
    }
    return tmp;
}
```

```
const Matrix<complex<double> >
sqrt( const Matrix<complex<double> >& mat ) {

    if( mat.num_rows() != mat.num_cols() ) throw("Matrices not square");

    Fortran_Matrix<complex<double> > tmpMat = toFortranMatrix( mat );

    Vector<double> eigVals( mat.num_rows() );
    Fortran_Matrix<complex<double> > eigVects = nan_to_zero(tmpMat);
    eigVals = Hermitian_eigenvalue_solve( eigVects );

    Fortran_Matrix<complex<double> > tmpMat2 =
        dagger(eigVects)*tmpMat*eigVects;
    const complex<double> zero(0.0,0.0);
    Fortran_Matrix<complex<double> > out(mat.num_rows(),
                                         mat.num_rows(),
                                         zero);
    for (int i=1; i<=mat.num_rows(); i++ ) {
        out(i,i) = sqrt( tmpMat2(i,i) );
    }

    //return toCMatrix(eigVects*out*dagger(eigVects));
    return toCMatrix( nan_to_zero(eigVects*out*dagger(eigVects)) );

}

const bool has_a_nan( const Matrix<complex<double> >& mat ) {

    for( int i=0; i<mat.num_rows(); i++ ) {
        for( int j=0; j<mat.num_cols(); j++ ) {
            if ( isnan(mat[i][j].real()) || isnan(mat[i][j].imag()) )
                return true;
        }
    }
    return false;

}

const Vector<double> eigVals( const Matrix<complex<double> >& mat ) {

    if( mat.num_rows() != mat.num_cols() ) throw("Matrices not square");

    Fortran_Matrix<complex<double> > tmpMat = toFortranMatrix( mat );

    Vector<double> eigVals( mat.num_rows() );
    Fortran_Matrix<complex<double> > eigVects = nan_to_zero(tmpMat);
    eigVals = Hermitian_eigenvalue_solve( eigVects );
```

```
    return eigVals;

}
                                                                        160
const Matrix<complex<double> > randomMatrix( Uniform<double>& generator,
                                              const int dimension,
                                              const double upperBound ) {

    Matrix<complex<double> > tmp( dimension, dimension,
                                  complex<double>(0.0,0.0) );
    for ( int i=0; i<dimension; i++ ) {
        tmp[i][i] = generator.random();
        for ( int j=0; j<i; j++ ) {
            tmp[j][i] = complex<double>( generator.random() - 0.5,          170
                                         generator.random() - 0.5 );
            tmp[i][j] = conj( tmp[j][i] );
        }
    }

    double trace = 0.0;
    for ( int i=0; i<dimension; i++ ) {
        trace += tmp[i][i].real();
    }
                                                                        180
    if ( trace < ZERO ) throw Fpe("trace too small");

    for ( int i=0; i<dimension; i++ ) {
        for ( int j=0; j<dimension; j++ ) {
            tmp[i][j] /= trace;
            tmp[i][j] *= upperBound;
        }
    }

    return tmp;                                                         190

}
```

B.3.8 display

```
#define _POSIX_SOURCE 1
// display.C
//
#include <iostream>

#include <sys/time.h> // all for getrusage and gettimeofday
```

```
#include <sys/resource.h>
#include <unistd.h>

#include "Bures.h"
#include "Matrices.h"
#include "fwrap.h"

#include "display.h"

void showProgress( const int step, const int numSteps, const int numQubits ) {

    static timeval startTime;
    if ( 0 == step ) {
        //first time through
        gettimeofday( &startTime, NULL );
        cout << "Stepper was started at timeofday: "
             << startTime.tv_sec << " seconds, "
             << startTime.tv_usec << " microseconds."
             << endl;
        cout << "Running "
             << numQubits
             << " qubits. . . |";
        cout.flush();
    }

    int aTenth = numSteps/10;
    if ( 0 == step%aTenth ) {
        cout << "=";
        cout.flush();
    }

    if ( step && 0 == step%(numSteps-1) ) {
        // end of the run.
        rusage usageStuff;
        if ( getrusage(RUSAGE_SELF,&usageStuff) ) {
            cerr << "stepper: can't get rusage" << endl;
        }
        timeval endTime;
        gettimeofday( &endTime, NULL );
        cout << "> Done!" << endl;
        cout << "Stepper finished at timeofday: "
             << endTime.tv_sec << " seconds, "
             << endTime.tv_usec << " microseconds."
             << endl;
        cout << " CPU time used: " << usageStuff.ru_utime.tv_sec
             << " seconds, " << usageStuff.ru_utime.tv_usec
             << " microseconds."
             << endl;
```

```
        timeval realTime = { 0 };
        timersub( &endTime, &startTime, &realTime );
        cout << " Real time used: "
             << realTime.tv_sec << " seconds, "
             << realTime.tv_usec << " microseconds."
             << endl;
    }

}

extern void printDiffs( ofstream& outFile,
                        const double time,
                        const State *const rho1,
                        const State *const rho2 ) {

    const Matrix<complex<double> >& mat1 = rho1->matrix();
    const Matrix<complex<double> >& mat2 = rho2->matrix();
    // errbnds on Bures?
    outFile << time * 100.0
            << ' '
            << distBures(mat1,mat2)
            << ' ';

    const Vector<double> eVals = eigVals(mat2);
//    const double eValErrBnd = get_eps() *
//        max( eVals[0], eVals[ eVals.size() - 1 ] );
// too small for now
    for ( int i=eVals.size() -1; i>=0; i-- ) {
        outFile << eVals[i]
                << ' ';
    }
    outFile << endl;

}

extern void printLeadingEVals( ofstream& outFile,
                               const double time,
                               const State *const rho1,
                               const State *const rho2 ) {

    const Matrix<complex<double> >& mat1 = rho1->matrix();
    const Matrix<complex<double> >& mat2 = rho2->matrix();
    outFile << time * 100.0
            << ' '
            << mat1(1,1).real()
            << ' '
            << mat2(1,1).real();
```

```
//    const Vector<double> eVals1 = eigVals(mat1);
//    outFile << max( eVals1[0], eVals1[ eVals1.size() - 1 ] )
//            << ' ';
//    const Vector<double> eVals2 = eigVals(mat2);
//    outFile << max( eVals2[0], eVals2[ eVals2.size() - 1 ] )
//            << ' ';
    outFile << endl;

}
```

B.3.9 exceptions

```
// exceptions.h

#ifndef _EXCEPTIONS_H_
#define _EXCEPTIONS_H_

#include <exception>

class Fpe: public exception {
  public:
    Fpe(const char* message): _message(message) {};

    // overrides exception::what()
    const char* what(void) const;

  private:
    const char* _message;
};


#endif // _EXCEPTIONS_H_
```

```
// exceptions.C
#include <iostream>

#include "exceptions.h"


const char* Fpe::what(void) const {
    cerr << Fpe::_message << endl;
    return "Oops... in Fpe::what";
}
```

B.3.10 fwrap

```
// fwrap.C
// Fortran Wrappers
#include <iostream>
#include <complex>

#include <tnt/tnt.h>
#include <tnt/vec.h>
#include <tnt/fmat.h>

#include <tnt/fortran.h>
//#include "fortran.h"

using namespace TNT;

#define F77_DGESV dgesv_
#define F77_DGELS dgels_
#define F77_DSYEV dsyev_
#define F77_DGEEV dgeev_
#define F77_ZHEEV zheev_
#define F77_ZHEEVD zheevd_
#define F77_DLAMCH dlamch_

extern "C"
{

    // linear equations (general) using LU factorizaiton
    //
    void F77_DGESV(cfi_ N, cfi_ nrhs, fda_ A, cfi_ lda,
        fia_ ipiv, fda_ b, cfi_ ldb, fi_ info);

    // solve linear least squares using QR or LU factorization
    //
    void F77_DGELS(cfch_ trans, cfi_ M,
        cfi_ N, cfi_ nrhs, fda_ A, cfi_ lda, fda_ B, cfi_ ldb, fda_ work,
            cfi_ lwork, fi_ info);

    // solve symmetric eigenvalues
    //
    void F77_DSYEV( cfch_ jobz, cfch_ uplo, cfi_ N, fda_ A, cfi_ lda,
        fda_ W, fda_ work, cfi_ lwork, fi_ info);

    // solve unsymmetric eigenvalues
    //
    void F77_DGEEV(cfch_ jobvl, cfch_ jobvr, cfi_ N, fda_ A, cfi_ lda,
        fda_ wr, fda_ wi, fda_ vl, cfi_ ldvl, fda_ vr,
        cfi_ ldvr, fda_ work, cfi_ lwork, fi_ info);
```

```
    // solve complex Hermitian eigenvalues and eigenvectors
    //
//    void F77_ZHEEV( cfch_ jobz, cfch_ uplo, cfi_ N, fda_ A, cfi_ lda,
//        fda_ W, fda_ work, cfi_ lwork, fda_ rwork, fi_ info);
    void F77_ZHEEV( cfch_ jobz, cfch_ uplo, cfi_ N, fda_ A, cfi_ lda,
        fda_ W, fda_ work, cfi_ lwork, fda_ rwork, fi_ info);

    // solve complex Hermitian eigenvalues and eigenvectors
    // with divide and conquer
    //
    void F77_ZHEEVD(cfch_ jobz, cfch_ uplo, cfi_ N, fda_ A, cfi_ lda,
        fda_ W, fda_ work, cfi_ lwork, fda_ rwork, cfi_ lrwork, fi_ iwork,
            cfi_ liwork, fi_ info);

    Fortran_double F77_DLAMCH(cfch_ ch);

}

// solve linear equations using LU factorization

using namespace TNT;

Vector<double> Lapack_LU_linear_solve(const Fortran_Matrix<double> &A,
    const Vector<double> &b)
{
    const Fortran_integer one=1;
    Subscript M=A.num_rows();
    Subscript N=A.num_cols();

    Fortran_Matrix<double> Tmp(A);
    Vector<double> x(b);
    Vector<Fortran_integer> index(M);
    Fortran_integer info = 0;

    F77_DGESV(&N, &one, &Tmp(1,1), &M, &index(1), &x(1), &M, &info);

    if (info != 0) return Vector<double>(0);
    else
        return x;
}

// solve linear least squares problem using QR factorization
//
Vector<double> Lapack_LLS_QR_linear_solve(const Fortran_Matrix<double> &A,
    const Vector<double> &b)
{
    const Fortran_integer one=1;
```

```
    Subscript M=A.num_rows();
    Subscript N=A.num_cols();

    Fortran_Matrix<double> Tmp(A);
    Vector<double> x(b);
    Fortran_integer info = 0;                                          100

    char transp = 'N';
    Fortran_integer lwork = 5 * (M+N); // temporary work space
    Vector<double> work(lwork);

    F77_DGELS(&transp, &M, &N, &one, &Tmp(1,1), &M, &x(1), &M, &work(1),
        &lwork, &info);

    if (info != 0) return Vector<double>(0);
    else                                                               110
        return x;
}

// ********************** Eigenvalue problems ******************

// solve symmetric eigenvalue problem (eigenvalues only)
//
Vector<double> Upper_symmetric_eigenvalue_solve(const Fortran_Matrix<double> &A)
{
    char jobz = 'N';                                                   120
    char uplo = 'U';
    Subscript N = A.num_rows();

    assert(N == A.num_cols());

    Vector<double> eigvals(N);
    Fortran_integer worksize = 3*N;
    Fortran_integer info = 0;
    Vector<double> work(worksize);
    Fortran_Matrix<double> Tmp = A;                                    130

    F77_DSYEV(&jobz, &uplo, &N, &Tmp(1,1), &N, eigvals.begin(), work.begin(),
        &worksize, &info);

    if (info != 0) return Vector<double>();
    else
        return eigvals;
}

                                                                       140
// solve unsymmetric eigenvalue problems
//
```

```
int eigenvalue_solve(const Fortran_Matrix<double> &A,
        Vector<double> &wr, Vector<double> &wi)
{
    char jobvl = 'N';
    char jobvr = 'N';

    Fortran_integer N = A.num_rows();

    assert(N == A.num_cols());

    if (N<1) return 1;

    Fortran_Matrix<double> vl(1,N); /* should be NxN ? **** */
    Fortran_Matrix<double> vr(1,N);
    Fortran_integer one = 1;

    Fortran_integer worksize = 5*N;
    Fortran_integer info = 0;
    Vector<double> work(worksize, 0.0);
    Fortran_Matrix<double> Tmp = A;

    wr.newsize(N);
    wi.newsize(N);

//  void F77_DGEEV(cfch_ jobvl, cfch_ jobvr, cfi_ N, fda_ A, cfi_ lda,
//      fda_ wr, fda_ wi, fda_ vl, cfi_ ldvl, fda_ vr,
//      cfi_ ldvr, fda_ work, cfi_ lwork, fi_ info);

    F77_DGEEV(&jobvl, &jobvr, &N, &Tmp(1,1), &N, &(wr(1)),
        &(wi(1)), &(vl(1,1)), &one, &(vr(1,1)), &one,
        &(work(1)), &worksize, &info);

    return (info==0 ? 0: 1);
}


// solve complex Hermitian eigenvalues and eigenvectors
//
// returns eigVects in A
Vector<double>
Hermitian_eigenvalue_solve( Fortran_Matrix<std::complex<double> >& A)
{

    char jobz = 'V';
    char uplo = 'U';
    Subscript N = A.num_rows();
```

```
    assert(N == A.num_cols());

    Vector<double> eigvals(N);
    //Fortran_integer worksize = 3*N;
    Fortran_integer worksize = 6*N;
    Vector<complex<double> > work(worksize);
    //Fortran_integer rworksize = 3*N;
    Fortran_integer rworksize = 6*N;
    Vector<double> rwork(rworksize);
    Fortran_integer info = 0;                                          200

//    void F77_DSYEV( cfch_ jobz, cfch_ uplo, cfi_ N, fda_ A, cfi_ lda,
//         fda_ W, fda_ work, cfi_ lwork, fda_ rwork, fi_ info);
    F77_ZHEEV(&jobz, &uplo, &N, (fda_)(&A(1,1)),
              &N, eigvals.begin(), (fda_)(work.begin()),
              &worksize, rwork.begin(), &info);

    if (info != 0) return Vector<double>();
    else
        return eigvals;                                                210
}

double get_eps(void) {

    char ch = 'E';
    return F77_DLAMCH(&ch);

}
```

B.3.11 myvalarray

```
#ifndef _MYVALARRAY_H_
#define _MYVALARRAY_H_

#include <valarray>
inline double
std::operator*( const valarray<double>& a, const valarray<double>& b ) {

    if ( a.size() != b.size() ) throw;

    double sum = 0.0;                                                  10
    for ( unsigned int i = 0; i < a.size(); i++ ) {
        sum += a[i]*b[i];
    }
```

```
    return sum;
}

#include <complex>
inline complex<double>
std::operator*( const valarray<complex<double> >& a,
                const valarray<complex<double> >& b ) {

    if ( a.size() != b.size() ) throw;

    complex<double> sum = 0.0;
    for ( unsigned int i = 0; i < a.size(); i++ ) {
        sum += a[i]*b[i];
    }

    return sum;
}

#endif // _MYVALARRAY_H_
```

Index

Bibliography

[1] J.J. Alvarez and C. Gòmez. A comment on Fisher information and quantum algorithms. v3, 1999. LANL ePrint `quant-ph/9910115`.

[2] P. Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *J. Stat. Phys.*, 22:563–591, 1980.

[3] Charles Bennett, Gilles Brassard, Ethan Bernstein, and Umesh Vazirani. Strenths and weaknesses of quantum computing. *SIAM Journal of Computing*, 26:1510, 1997. LANL ePrint `quant-ph/9701001`.

[4] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM J. Computing*, 26:1411–1473, 1997.

[5] Eli Biham, Ofer Biham, David Biron, Markus Grassl, and Daniel A. Lidar. 1998. LANL ePrint `quant-ph/9807027`.

[6] Michel Boyer, Gilles Brassard, Peter Hoyer, and Alain Tapp. Tight bounds on quantum searching. 1996. LANL ePrint `quant-ph/9605034`.

[7] G. Brassard, P. Hoyer, and A. Tapp. 1997. LANL ePrint `quant-ph/9705002`.

[8] G. Brassard, P. Hoyer, and A. Tapp. 1998. LANL ePrint `quant-ph/9805082`.

[9] Samuel L. Braunstein. Quantum computation. *unpublished lecture notes*, 1996.

[10] Samuel L. Braunstein and Carlton Caves. Statistical distance and the geometry of quantum states. *Phys. Rev. Lett.*, 72:3439–3443, 1994.

[11] D. J. C. Bures. *Trans. Am. Math. Soc.*, 135:199, 1969.

[12] N. J. Cerf, L. K. Grover, and C. P. Williams. 1998. LANL ePrint `quant-ph/9806078`.

[13] D. Coppersmith. An approximate fourier transform useful in quantum factoring. *IBM Research Report*, RC19642, 1994.

[14] D. Deutsch. Quantum theory as a universal physical theory. *Int. J. Theor. Phys.*, 24:1–41, 1985.

[15] D. Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proc. Roy. Soc. Lond. A*, 400:97–117, 1985.

[16] D. Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc. Roy. Soc. Lond. A*, 439:553–558, 1992.

[17] David Deutsch, Artur Ekert, and Rossella Lupacchini. Machines, Logic, and Quantum Physics. 1999. LANL ePrint `quant-ph/9911150`.

[18] C. Durr and P.Hoyer. 1996. LANL ePrint `quant-ph/9607014`.

[19] A. Ekert and R. Jozsa. Quantum computation and shor's factoring algorithm. *Rev. Mod. Phys.*, 68(3), 1996.

[20] E. Farhi and S. Gutmann. *Phys. Rev. A*, 57:2403, 1998.

[21] R. P. Feynman. Simulating physics with computers. *Int. J. Theor. Phys.*, 21:467–488, 1982.

[22] R. P. Feynman. Quantum mechanical computers. *Found. Phys.*, 16:507–531, 1985.

[23] G. Fubini. Sulle metriche definite da una forma Hermitiana. *Atti Instit. Veneto*, 6:501–513, 1903.

[24] G.W. Gibbons. Typical states and density matrices. *J. Geom. Phys.*, 8:147–162, 1992.

[25] V. Gorini, A. Kossakowski, and E.C.G. Sudarshan. *J. Math. Phys.*, 17:821, 1976.

[26] L. K. Grover. 1996. LANL ePrint `quant-ph/9607024`.

[27] L. K. Grover. *Phys. Rev. Lett.*, 79:4709, 1997.

[28] Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 78:325–328, 1996. Also LANL ePrint `quant-ph/9605043`, updated as `quant-ph/9706033`.

[29] G.H. Hardy and E.M. Wright. *An introduction to the theory of numbers.* Clarendon Press, 1979.

[30] J. Kempe, D. Bacon, D. A. Lidar, and K. B. Whaley. Theory of decoherence–free fault–tolerant universal quantum computation. v2, 2000. LANL ePrint `quant-ph/0004064`.

[31] Hoi-Kwong Lo, Sandu Popescu, and Tim Spiller. *Introduction to Quantum Computation and Information*. World Scientific, 2000.

[32] Yuri Manin. Computable and uncomputable (in russian). *Sovetskoye Radio*, 1980.

[33] Mikio Nakahara. *Geometry, Topology, and Physics*. Institute of Physics Publishing, 1990.

[34] Charles Nash. *Differential Topology and Quantum Field Theory*. Academic Press, 1991.

[35] B. Pablo-Norman and M. Ruiz-Altaba. Noise in grover's quantum search algorithm. 1999. LANL ePrint `quant-ph/9903070`.

[36] John Preskill. Quantum Information and Quantum Computing. 1996. Compiled lecture notes from various talks. Available online at `http://www.theory.caltech.edu/people/preskill/`.

[37] John Preskill. Lecture notes for physics 229: Quantum information and computation. 1998. Available online at `http://www.theory.caltech.edu/people/preskill/ph229/`.

[38] Peter W. Shor. Polynomial–time algorithms for prime factorization and discrete logarithms on a quantum computer. In *Proc. 35th Annual Symp. on Foundations of Computer Science*. IEEE Computer Society Press, 1994. Revised edition available, LANL ePrint `quant-ph/9508027`.

[39] Peter W. Shor. Introduction to quantum algorithms. 2000. LANL ePrint `quant-ph/0005003`.

[40] Daniel R. Simon. On the power of quantum computation. *SIAM J. Computing*, 26:1474–1483, 1997.

[41] Andrew Steane. Quantum computing. 1997. LANL ePrint `quant-ph/9708022`.

[42] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, 1997.

[43] E. Study. Kuerzeste wege im komplexen gebiete. *Math. Ann.*, 60:321–377, 1905.

[44] B. M. Terhal and J. A. Smolin. *Phys. Rev. A*, 58:1822, 1998.

[45] A. Uhlmann. *Rep. Math. Phys.*, 9:273, 1976.

[46] W.K. Wooters. Statistical distance and Hilbert space. *Phys. Rev. D*, 23(2):357–362, 1981.

[47] Kentaro Yano. *Differential Geometry on Complex and Almost Complex Spaces*. Pergamon Press, 1965.

[48] Christof Zalka. Grover's quantum searching algorithm is optimal. v2, 1997. LANL ePrint `quant-ph/9711070`.

Vita

Mark McGrew Mims was born on Christmas day, 1970 to Joseph M. and Bonnie W. Mims. He was raised in Louisiana, Alabama, and Texas. After graduating in 1988 from Langham Creek High School in Houston, he attended The University of Texas at Austin, graduating in 1992 with bachelor of science degrees in both physics and mathematics. Graduate work is being completed at the same institution under the supervision of Professor E.C.G. Sudarshan.

Permanent address: P.O. Box 8465
Austin, Texas 78713-8465

This dissertation was typeset with LaTeX[‡] by the author.

[‡]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.