Quantum Compiling

by

Aram Harrow

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Physics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

Author .....
Department of Physics
May 18, 2001

Certified by .....
Neil Gershenfeld
Associate Professor
Thesis Supervisor

Accepted by .....
David E. Pritchard
Senior Thesis Coordinator, Department of Physics

Quantum Compiling

by

Aram Harrow

Submitted to the Department of Physics
on May 18, 2001, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Physics

Abstract

A key assumption in quantum computation is the ability to efficiently generate arbitrary unitary operators to high precision. However, it is often more reasonable to assume that we are only able to implement a finite set of primitive gates, from which we need to efficiently generate good approximations to any other single-qubit operation. Fortunately, the Solovay-Kitaev theorem shows that for almost all sets of primitive gates, the required string length is only polynomial in the number of bits of precision. In the first part of this thesis, we give a constructive proof of the theorem, numerically evaluate its effectiveness and compare it with the performance guarantee that it gives. In contrast with the polynomial cost of Solovay-Kitaev, elementary counting arguments give a lower bound for string length that is only linear in the bits of accuracy. In the second part we give a novel non-constructive proof that it is almost always possible to saturate this lower bound, and in the third part, we demonstrate a particular set of primitive gates that shows promise in being able to efficiently generate optimal compilations. We also discuss practical methods of realizing these gates fault-tolerantly.

Thesis Supervisor: Neil Gershenfeld
Title: Associate Professor

Contents

Acknowledgments

"Nothing will come of nothing" –King Lear

This work could never have been accomplished without the inspiration, guidance and support of innumerable friends and colleagues. I would particularly like to thank Isaac Chuang for introducing me to the problem and providing invaluable advice throughout the course of the project. The two of us colloborated with Ben Recht and Xinlan Zhou, who together have guided me through most of the mathematical difficulties that I have encountered. In the past year, I have also benefitted enormously from being able to work on NMR quantum computation in the physics and media group of the MIT media lab. I am deeply indebted to Yael Maguire for being a committed friend and mentor, Neil Gershenfeld, for among other things, causing me to rethink my conceptions of theory and practice, and Jason Taylor for, as he put it, "network code and spiritual guidance," upon both of which I have been utterly dependent.

Coming up with the results of the third chapter was possible only after I had bounced my ideas off a number of brilliant minds. In particular, it is unlikely that I would have succeeded without the discussions I had with Santosh Vempala, David Jerison, Persi Diaconis and Seth Lloyd. Finally, the software I wrote for chapter 2 required more computing cycles that I alone had access to, and I am grateful for the assistance in running it that I received from Jack Holloway, Matt Feinberg, Carla Pellicano, David Signoff, Gabe Weinberg and Dan Benhammou, as well as most of my physics and media coworkers.

Chapter 1

Introduction

The insight that information is physical has given rise to a beautiful intersection of mathematics, computer science, physics and engineering known as quantum computing. Though we are far from being able to build a useful quantum computer, simply knowing that such a device is possible has forced us to think about the physical meanings of information and computation in new and interesting ways.

This thesis, like much other work in the field, begins with a fairly technical and specific problem in quantum computation and ends with potentially new ways of looking at related fields such as quantum control. The problem, which I call *quantum compiling*, is fairly simple to express: an experimenter wishes to implement an arbitrary unitary transformation on a quantum system to a given precision, but for whatever reason is only able to apply a small discrete set of unitary operations. How many of these operations do we need to apply to achieve a precision ϵ? And how can we come up with this string of operations in a computationally efficient fashion?

These questions have obvious practical importance for writing programs for quantum computers, but can also be related to properties of the unitary groups that can apply to manipulating a wide range of quantum systems.

1.1 Outline

The first chapter of this thesis is devoted to background material. Although it would be possible to define quantum computation without reference to any more than elementary linear algebra, it is often far more fruitful to view it as an outgrowth of either quantum mechanics or classical complexity theory. With this is mind, the next two sections of the introduction will each derive the same definition of quantum computing in quite different ways. The remainder of the chapter reviews some important proofs related to quantum compiling. While the introduction surveys results that one cannot expect to learn fully from my limited treatments of them, the rest of the thesis should require no more than knowledge basic linear algebra and probability, and where it uses more advanced concepts, it proves most of the necessary assumptions.

The second chapter concerns itself with the Solovay-Kitaev theorem, which to date is the only substantial result in quantum compiling for a single qubit. I provide a constructive proof of the theorem for a single qubit, discuss generalizations, numerically evaluate the algorithm's effectiveness and compare with theoretical performance guarantees. Source code is provided in the appendix.

The third chapter proves upper and lower bounds for generic compiling that are within a constant factor of each other. Though these results are optimal, the upper bounds given are unfortunately not constructive. To the best of my knowledge, this result is novel.

While most of the results of this thesis apply to almost any set of basic quantum gates, in the fourth chapter I discuss a particular set of base single-qubit gates that are in many ways optimal. Though I have not been able to develop an efficient algorithm that uses fewer of these gates than the

generic Solovay-Kitaev approach, I report partial progress on an algorithm that would asymptotically achieve the lower bound and discuss what more is necessary.

The final chapter considers generalizations and discusses the major open questions that remain.

1.2 Quantum Mechanics

Anybody who is not shocked by quantum theory has not understood it –Niels Bohr

1.2.1 Axioms of quantum mechanics

Although most of what is presently known about basic quantum mechanics was worked out in the early twentieth century, we are far from having worked out all of the implications. The quantum mechanical picture of particles and wave functions is profoundly counter-intuitive and despite its universal acceptance, there are few people who are truly comfortable with it. One of the most striking examples of this quantum weirdness is *superposition*, the ability of quantum systems to exist in arbitrary linear combinations of allowable classical states. This accounts for, among other things, the fact that we observe interference patterns in the double-slit experiment even when only a single photon passes through at a time.

Mathematically, we say that the state of a system exists as a unit column vector in the Hilbert space $\mathbb{C}^N$, where N can be thought of as being loosely related to the number of possible classical states if every continuous variable were quantized. In the Dirac formalism, a column vector is called a *ket* and denoted $|\psi\rangle$, while its *adjoint* (or conjugate transpose) is a *bra* (or row vector), which is written as $\langle\psi|$. This suggests writing the inner product between two vectors $|\psi\rangle$ and $|\varphi\rangle$ as $\langle\psi|\varphi\rangle$, with respect to the canonical form, or as $\langle\psi|\hat{Q}|\varphi\rangle$, for the inner product with respect to an arbitrary Hermitian form $\hat{Q}$. For such a Hermitian form, its *expectation* (with respect to some state $|\psi\rangle$ is represented $\langle Q\rangle$, and is defined to be $\langle\psi|\hat{Q}|\psi\rangle$. Outer products are also written the way one would expect, as $|\psi\rangle\langle\varphi|$, and their action on an arbitrary vector $|n\rangle$ is defined to be $|\psi\rangle\langle\varphi|n\rangle = \langle\varphi|n\rangle|\psi\rangle$, as one would expect from simply writing the terms next to each other. The tensor product of $|\psi\rangle \in \mathbb{C}^M$ and $|\varphi\rangle \in \mathbb{C}^N$ is written $|\psi\rangle \otimes |\varphi\rangle \in \mathbb{C}^{MN}$, or when context permits, $|\psi\rangle|\varphi\rangle$. If $|\psi\rangle = \sum_m a_m|m\rangle$ and $|\varphi\rangle = \sum_n b_n|n\rangle$, then

$$|\psi\rangle \otimes |\varphi\rangle \equiv \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} a_m b_n |mN+n\rangle, \tag{1.1}$$

where the basis states of $\mathbb{C}^d$ are represented by $|0\rangle, \ldots, |d-1\rangle$. Most of this treatment can be found in elementary textbooks on quantum mechanics (such as [Grif] and [Gas]) and linear algebra ([Axl] and [Art]).

For quantum computing, we often use the two-dimensional Hilbert space corresponding to the spin of a spin-1/2 particle. If we have n spin-1/2 particles, then the resulting Hilbert space is $\overbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2}^{n}$ or $\mathbb{C}^N$, where $N = 2^n$. An immediate consequence of this is that representing the state of a quantum system requires exponentially more information (i.e. N complex numbers) than the $\log N$ bits it would take to represent a classical system that can occupy only one of N possible states. Since N is typically exponential in the size of the system, this has important practical implications for simulating quantum dynamics on classical computers. This curious gap, between the size of the actual quantum system and the exponentially larger size of the classical computer necessary to simulate its behavior led Feynman and other physicists to propose instead performing quantum simulations on some sort of quantum computer.[Fey82] After all, there shouldn't be any fundamental differences between the local Hamiltonians that nature applies and the sorts of Hamiltonians that can be generated in the laboratory. This intuition was formalized in several different papers (see for example [AL97]).

Quantum simulation is only as powerful as the simulator, of course, and is predicated on the assumption that Nature is governed by a Hamiltonian that can be easily represented within the models of quantum computation currently being considered. This means that if our understanding of quantum mechanics is revised, perhaps from the inclusion of gravity, then polynomial time simulation may require some new and more powerful sort of quantum computer. In the meantime, though, we content ourselves with describing the non-relativistic theories of quantum mechanics that the current concepts of quantum computers are based on.

Deriving these quantum mechanical principles requires surprisingly few assumptions. If $|n\rangle$ indexes an orthonormal basis of states and a measurement is performed in that basis, let the probability that $|\psi\rangle$ is found in the state $|j\rangle$ be $|\langle j|\psi\rangle|^2$. The fact that this quantity is squared means that L^2 is the fundamental norm of quantum mechanics, a result that will have important implications. Immediately, we see that any wave vector has to have L^2 norm one, that multiplying the wave vector by an arbitrary scalar $e^{i\phi}$ has no effect, and that defining the expectation to be $\langle\psi|\hat{Q}|\psi\rangle$ gives the correct answer, if measurement randomly gives an eigenvalue of $\hat{Q}$ with the appropriate probabilities. We say that an *observable* is represented by a matrix $\hat{Q}$ that has all real eigenvalues (so that measurements make physical sense) and is hermitian. Furthermore, for measurements to be meaningful, they should be repeatable, so if a measurement returns the result λ, then the wave vector should be projected onto the space of eigenvectors of the obserable with eigenvalue λ, which implies that $\hat{Q}$ must be diagonalizable and hence hermitian. So in principle we can measure the expected eigenvalue of any hermitian matrix.

A more interesting question is to ask what transformations are possible on a wave vector. If we assume linearity, then the requirement that our vectors remain normalized means that any change over time can be expressed by a unitary matrix, and since overall phase is physically unobservable, we can argue without loss of generality that the matrix has determinant one. The set of unitary $N \times N$ matrices is called the *unitary group* and is denoted $U(N)$ and the subset with determinant one is known as the *special unitary group* or $SU(N)$. Since the structure of $SU(N)$ will be central to the question of quantum compilation, it will be useful to review some facts about the group before continuing.

1.2.2 The Special Unitary group

A matrix U is said to be *unitary* if U is invertible and $U^\dagger = U^{-1}$, where † denotes conjugate transpose. The product of two unitary matrices is also unitary and $\det(AB) = \det(A) \cdot \det(B)$, so $U(N)$ and $SU(N)$ are groups. An important property of unitary matrices is that they leave the standard inner product invariant, i.e. $\forall u, v \in \mathbb{C}^N, M \in U(N), \langle Mu, Mv\rangle = u^\dagger M^\dagger M v = u^\dagger v = \langle u, v\rangle$, or equivalently, that they can be thought of as changes of complex orthonormal bases. To change the basis of a column vector we multiply it on the left by our unitary matrix U and to change the basis of a matrix M we conjugate it, i.e. $U^\dagger M U$.

In order to make these ideas more concrete, we consider the simplest possible unitary group, $U(1)$. This is simply the set of complex numbers with norm one, or equivalently, $\{e^{i\theta}|\theta \in \mathbb{R}\}$. A more interesting example is $SU(2)$, the set of transformations possible on a single two-level quantum system, if we ignore overall phase. For any $U \in SU(2)$, the conditions $U^\dagger U = I$ and $\det U = 1$ mean that we can write U in the form,

$$U = \begin{bmatrix} a + ib & c + id \\ -c + id & a - ib \end{bmatrix}, \tag{1.2}$$

with $a, b, c, d \in \mathbb{R}$ and $a^2 + b^2 + c^2 + d^2 = 1$. Thus, $SU(2)$ is homeomorphic to S^3, the unit 3-sphere in $\mathbb{R}^4$, which in turn is locally homeomorphic to $\mathbb{R}^3$, so we say that $SU(2)$ has dimension 3. If we

introduce the *Pauli spin matrices*

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \tag{1.3}$$

then (**??**) can be rewritten as $U = aI + bi\sigma_z + ci\sigma_y + di\sigma_x$. The $\{\sigma_x, \sigma_y, \sigma_z\}$ square to one, and multiplication among them is similar to the ordinary vector cross product, i.e. $\sigma_i\sigma_j = \delta_{ij}I + i\epsilon_{ijk}\sigma_k$, where

$$\epsilon_{ijk} = \begin{cases} 1 & \text{for } ijk \text{ an even permutation of 123} \\ -1 & \text{for } ijk \text{ an odd permutation of 123} \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

Let $su(2)$ denote the set of all trace-zero skew-hermitian 2×2 matrices. Then we can consider $su(2)$ a real three-dimensional vector space with basis $i\sigma_x$, $i\sigma_y$, $i\sigma_z$. To make the correspondence between $su(2)$ and $\mathbb{R}^3$ more explicit, define

$$\vec{\sigma} = \hat{x}\sigma_x + \hat{y}\sigma_y + \hat{z}\sigma_z, \tag{1.5}$$

so that $i\vec{\sigma}\cdot\vec{v} = v_1 i\sigma_x + v_2 i\sigma_y + v_3 i\sigma_z$. Define the commutator (or *Lie bracket*) of two elements of $su(2)$ to be $[v, w] \equiv vw - wv$. It is straightforward to verify that $su(2)$ is closed under commutation, and that commutation fulfills all the requirements (bilinearity, skew symmetry and the Jacobi identity) for $su(2)$ to be considered a *Lie algebra*.

In fact, $su(2)$ is also the Lie algebra of $SU(2)$, formally defined to be the space of infinitesimal vectors tangent to the identity matrix. For any matrix A, define e^A by the same power series used for scalars,

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!} \tag{1.6}$$

Any element of $su(2)$ can be written as $\theta i\vec{\sigma}\cdot\vec{v}$, where $\theta \in \mathbb{R}$ and $|v| = 1$. Now the matrix exponential is simply $e^{\theta i\vec{\sigma}\cdot\vec{v}} = \cos\theta + i\vec{\sigma}\cdot\vec{v}\sin\theta$, a form in which any element of $SU(2)$ can be written. This allows us to easily calculate the effects of conjugating by an element of $SU(2)$. If we consider u and v to be equivalently either matrices in $su(2)$ or vectors in $\mathbb{R}^3$ then

$$e^{-\theta v} u e^{\theta v} = u\cos 2\theta + \frac{[u, v]}{2}\sin 2\theta = \vec{u}\cos 2\theta + (\vec{u}\times\vec{v})\sin 2\theta \tag{1.7}$$

Thus conjugation by $e^{\theta v}$ is equivalent to rotating by an angle 2θ about the $\vec{v}$ axis. This suggests a natural homeomorphism from $SU(2)$ to $SO(3)$, the rotation group for $\mathbb{R}^3$, which is a double covering because the angle is mapped from θ to 2θ.

Most of the same results generalize in straightforward fashion to $SU(N)$. If $U = e^u$, then one can verify that

- $\det U = 1$ if and only if $\mathrm{Tr}\ u = 0$.
- U is unitary ($UU^\dagger = I$) if and only if u is skew-hermitian ($u^\dagger = -u$)

Thus $su(N)$ is the set of all trace-zero skew-hermitian $N \times N$ matrices, which can be thought of as an $N^2 - 1$ dimensional real vector space. This is an easy way of seeing that $SU(N)$ also has dimension $N^2 - 1$, since in general we cannot make arguments as simple as our identification of $SU(2)$ with the 3-sphere. In fact, a famous theorem in topology holds that no sphere other than S^3 and S^1 (which is homeomorphic to $U(1)$) can have a group structure.

We will often find it useful to study the subgroups of $SU(N)$. For any $A_1, \ldots, A_l \in SU(N)$, denote the subgroup that they generate by $\langle A_1, \ldots, A_l\rangle$. This is defined to be the smallest group that contains $A_1, \ldots, A_l$, or equivalently, to be group of all finite words (including the empty word I) comprised of $A_1, \ldots, A_l, A_1^{-1}, \ldots, A_l^{-1}$. For $SU(2)$, it has been known since the Greeks that any

finite groups of this form are isomorphic to either S_4, A_4, A_5, C_n, or D_n; either a cyclic or dihedral group of any order, or the symmetries of a Platonic solid. Another common sort of subgroup is a *one-parameter subgroup*, which is defined for any $A \in su(N)$ to be $\{e^{At} : t \in \mathbb{R}\}$. One-parameter subgroups of $SU(2)$ are isomorphic to $U(1)$, but in general we can only conlude that one-parameter subgroups are isomorphic to R^+.

We will also need a notion of a norm, not just for elements of $SU(N)$, but for linear operators in general. First define the familiar L^k vector norm to be

$$\|v\|_k \equiv \left(\sum_i |v_i|^k\right)^{1/k} \tag{1.8}$$

Unless otherwise specified we will always use the L^2 vector norm. Taking the limit as $k \to \infty$ gives the L^∞ norm, or $\|v\|_\infty \equiv \sup\{|v_i\}$. By analogy, we define the L^∞ operator norm to be

$$\|M\|_\infty = \sup\left\{\|vf\| \,\middle|\, \|v\| = 1\right\}. \tag{1.9}$$

The reason why we work with $SU(N)$ instead of $U(N)$ is that the overall phase of our system is an irrelevant degree of freedom. However, using $SU(N)$ does not completely eliminate this. Though phase is no longer a continuous degree of freedom, elements of $SU(N)$ can be multiplied by $\exp(2\pi ik/N)$ for any integer k and they will still have determinant one. The subgroup of $U(N)$ that has no phase information at all is the rarely-discussed *projective special unitary group* $PSU(N) \equiv SU(N)/\mathbb{C}^*$. However, $SU(N)$ is often more convenient to work with – for one thing, because it is a real algebraic group (it can be described as the locus of zeroes of a set of real polynomials in its entries) unlike $PSU(N)$ – and the small difference in size between the two groups usually does not affect our results. However, in chapter two, though the theory uses $SU(2)$, the performance guarantees are slightly stricter than necessary, since most of the actual computations were performed using $PSU(2)$, a group equivalent to $SO(3)$.

1.2.3 Quantum mechanics as computational resource

With the above formalism established, most of the important principles of quantum computation have already been described. The canonical example of a quantum computer is a set of n spin-1/2 particles that can be coupled together or manipulated individually. If spin up and spin down are labelled $|0\rangle$ and $|1\rangle$, then we can define the *computational basis* to be the set of all n-fold tensor products of these two states. It is natural to express the basis states as n-bit binary numbers, or simply as $|0\rangle, |1\rangle, \ldots, |N-1\rangle$, where $N = 2^n$ is the dimension of the Hilbert space.

To use such a quantum computer, there are four basic requirements. First, one must be able to initialize the system to a ground state, which we call $|0\rangle$. Then, it must be possible to implement a set of local Hamiltonians that are computationally universal for reasonably precise amounts of time. We will return to computational universality, but for now assume that one ought to be able to implement any Hamiltonian that is in the tensor product of $su(4)$ with $n-2$ copies of the 2×2 identity matrix. In other words, any unitary transformation on two neighboring spins. The system must also resist decoherence and other sources of errors, a subject which we will largely avoid discussing. Finally, one must be able to perform a measurement in the computational basis, or equivalently, one should be able to measure each spin individually.

The above description says little about the computational power of such a device, and without making the notion of computational power more precise it will be difficult to do so. Nevertheless there are still a few strengths of quantum computation that we should be able to have intuition for at this point. One that has already been mentioned is quantum simulation. If we can effect any local Hamiltonian, then we should be able to simulate any quantum system that has entirely local dynamics, a problem which is generally thought to be computationally hard for classical computers.

Another problem where quantum computers outperform classical computers, this time provably so, is in unstructured search. Suppose we have a function $f : \{0,1\}^n \to \{0,1\}$, which we can compute but not analyze. The problem is to find a value of x such that $f(x) = 1$. It is easy to see that for a classical computer, we cannot do better than evaluating $f(0), f(1), f(2), \ldots$ until f returns a one, since no ordering is better than any other and when f returns zero we gain no information. In the worst case, when $f(x) = 1$ only for a single value of x, this takes $N/2$ evaluations of $f(x)$ on average. Using a method known as Grover's algorithm, however, quantum computers can find x such that $f(x) = 1$ in only $O(\sqrt{N})$ steps. The algorithm first prepares the quantum system in a even superposition of all of the computational basis states, i.e.

$$|\psi\rangle = N^{-1/2} \sum_{x=0}^{N-1} |x\rangle \tag{1.10}$$

This can be accomplished by applying a Hadamard transformation,

$$H = \frac{i}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.11}$$

to each qubit. Note that the leading term is $N^{-1/2}$ so that we will have $\langle\psi|\psi\rangle = 1$. Though we will not prove the effectiveness of the algorithm, the basic idea is that for every evaluation of $f(x)$ followed by an appropriate unitary transformation, we can increase the *amplitude* of the $|x\rangle$ state (for some x with $f(x) = 1$) by a roughly constant amount. Since the probabilty of finding the system in a state is the squared magnitude of the amplitude of that state, we only need $O(\sqrt{N})$ to have an $O(1)$ chance of success.

The key of this quadratic speedup is the L^2 norm used in quantum mechanics and it is not surprising that quantum systems (or even phase-coherent classical systems) often exhibit this sort of improvement in dealing with highly general problems.

To better define the sorts of problems that we will solve with quantum computers, we will now turn to computer science.

1.3 Complexity Theory

Quantum computation can also be considered completely formally, as an extension of theories of classical computation. Instead of thinking of quantum computers as quantum systems for which we are able to control the Hamiltonian and apply measurement operators, it is sometimes fruitful to instead consider them to be phase-coherent randomized Turing machines.

1.3.1 Classical Complexity Theory

Complexity theory, broadly speaking, is the study of how easily certain problems can be solved with entirely mechanical techniques. The space of problems is fairly simple to describe; determining whether a particular string is a member of a set of strings (or *language*) is easily general enough. What makes the field possible is that most mechanical methods of solving problems, from using Pentiums to billiard balls, are loosely equivalent.

The generally accepted formalism for unifying classical computation is the *Turing machine*. A Turing machine has a set of states Q, three of which are marked *start*, *accept*, or *reject*, an alphabet Σ, which it uses for both its input and for a read/write work space, and a transition function $f : Q \times \Sigma \times \Sigma \to Q \times \Sigma \times \{L, R\}$. This function represents one step of running the machine; it maps the machine's state (in Q), the next character of the input stream (in Σ) and the current character of the tape (also Σ) onto a new state (in Q). It also indicates a character (in Σ) for the machine to write on the old tape location and a direction (either L or R) to move the tape head by a step. The

machine halts only when it reaches an accept or reject state. We say that a Turing machine *decides* a language if it accepts strings that are in the language and rejects strings that are not, all within a finite amount of time.

This mode of computation seem a little abstruse and rather inefficient; performing most computationally interesting tasks requires an immense amount of zipping back and forth on the tape. However, the *weak Turing-Church thesis* holds that Turing machines are computationally universal, meaning that any other imaginable method of computing can be simulated by a Turing machine. It is of course impossible to prove this in general without a complete description of the physical laws governing the universe, but a counter-example has never been found. Turing machines that have multiple tapes, or that have registers and random-access memory like modern computers or even that can branch non-deterministically and accept if any computational branch accepts, are all provably equivalent to the basic model of a Turing machine. This is of course a fairly loose equivalence: the class of decidable languages is the same, but the time and space required to do so can vary wildly based on implementation.

For practical questions about computing, we need a notion of the time complexity (or sometimes space complexity) of an algorithm or a language. The time complexity of an algorithm is said to be $O(f(n))$ if there exists a constant c such that on any input of length n, the algorithm takes no longer than $cf(n)$ time to halt. Space complexity is similarly defined in terms of the total amount of tape that the Turing machine accesses. This allows us to define time (and space) complexity classes $\mathbf{TIME}(f(n))$ (and $\mathbf{SPACE}(f(n))$) to be the set of all languages that are decided by a Turing machine in time (or space) $f(n)$. The complexity class that is loosely equivalent to what is practically computable is generally held to be $\mathbf{P}$, the class of languages computable in polynomial time, or $\mathbf{P} = \cup_k \mathbf{TIME}(n^k)$. We often call these languages *polytime* computable, or even simply *efficiently* computable. In contrast *computationally hard* problems are thought (but usually not proved) to require super-polynomial time or often exponential time. The much powerful class of polynomial space languages is analogously defined to be $\mathbf{PSPACE}$.

By adding more capabilities to polynomial-time Turing machines we can obtain other interesting complexity classes. One fairly practical feature is the ability to generate random bits. The class of *bounded-error probabilistic polynomial-time* languages is denoted $\mathbf{BPP}$ and consists of the languages that are decided with error no greater than $2/3$ by poly-time Turing machines with access to random bits. This is thought to most accurately describe classical computing capabilities, and it is an open question whether $\mathbf{BPP} = \mathbf{P}$. Note that the $2/3$ success rate is a somewhat arbitrary cut-off; if our probability of success if $\frac{1}{2} + \frac{1}{p(n)}$ for some polynomial $p(n)$ in the size of the input, then in a polynomial number of steps we can amplify our probability of success exponentially close to one. However, what if our criterion for success was simply that the machine always rejected bad inputs, but had some non-zero probability (perhaps exponentially small) of accepting on any good input? This is known as *non-determinism*, as it can equivalently be thought of as having the ability to simultaneously try all possible values of the random bits and accept if any setting of them results in an accepting computation. The class of languages decided by such machines is known as $\mathbf{NP}$. Its complement, $\mathbf{coNP}$, can be thought of as the class of languages decided by a random Turing machine that always accepts members of the language and has a non-zero probability of rejecting any string not in the language.

The concept of polynomial time is central enough that it is often useful to speak of *polynomial time reducibility*. A language A is polynomial time reducible to a language B (denoted $A \leq_P B$) if there exists a polynomial time Turing machine M whose output is a member of B if and only if its input is a member of A. Thus, if B is solvable in polynomial time, then A is as well. This is a useful way of showing that languages are roughly equally difficult. For example, the *Cook-Levin Theorem* holds that any language in $\mathbf{NP}$ is polynomial time reducible to SAT, which is the language of boolean formulas such as $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$ which are true for some assignment of the x_i. The proof involves writing the computational history of a polytime verifier as a polynomial size boolean formula which has a satisfying assignment if and only if there exists a certificate which can cause the

verifier to accept. Armed with this, we say that a language $\mathcal{L}$ is **NP**-complete if 1) $\mathcal{L} \in \mathbf{NP}$ and 2) SAT (and hence any language in **NP**) is polytime reducible to $\mathcal{L}$. A language is **NP**-hard if only the second condition is met.

Proving languages to be **NP**-complete is useful because if a single **NP**-complete language is decidable in polynomial time, then $\mathbf{P} = \mathbf{NP}$, which is generally considered to be a most improbable state of affairs. If we give up on **NP**-complete languages, though, then there are still several interesting problems thought to be in **NP**and not **P**, such as factoring a large composite number or determining whether two graphs are isomorphic. Such problems are informally said to constitute the class **NPI**, and are loosely thought to be the area where we can expect the most improvement on quantum computers.

Before formally defining quantum computation, we introduce the classical notion of *circuit complexity*. A circuit has a fixed number of input wires and some number of gates (either AND, OR, or NOT) that take as inputs either input wires or other gates. One gate is designated *output*, and the circuit is said to accept or reject its input based on the value taken on by the output gate. A nearly equivalent definition of a circuit, which we will use interchangeably, is to say that there are input bits and work bits (one of which is marked as output) and that the circuit runs by executing a fixed series of gates, each of which takes inputs either from the input bits or the work bits and writes its output to a work bit. This second definition has the advantage of being easily expressible in terms of linear algebra. If there are a total of N input and work bits, then there are 2^N different possible states, and we can represent a state $|\psi\rangle$ by a column vector in $\mathbb{Z}_2^{2^N}$ with a single entry equal to one and the rest filled with zeroes. This is equivalent to saying that states have L_1 norm one. Every gate $\mathcal{G}_i$ can then be expressed as a matrix of zeroes and ones with exactly one non-zero term in each column. The result of the final check of the output bit can be expressed as $\|Q|\psi\rangle\|_1$, where Q is the matrix that projects onto the subspace of accepting final states. Thus the entire computation is expressed by $\|Q\mathcal{G}_n \cdots \mathcal{G}_1|\psi_0\rangle\|_1$.

Expressing a simple circuit in terms of exponentially large matrices made up entirely of zeroes and ones may seem a little pointless right now, but this formalism will allow for some interesting generalizations. For example, to add randomness to a circuit, we consider our state space to instead be all vectors in $\mathbb{R}^{2^N}$ whose entries sum to one and we allow our gates to be composed of nonnegative real numbers, requiring only that each column sum to one. Our measurement stays the same, but the result is now interpreted as the probability that the circuit accepts. Nondeterministic circuits are defined the same way, but now the criteria for deciding a language are that the machine always reject inputs not in the language and have a nonzero probability of accepting inputs in the language.

In general, circuits can solve the same sorts of problems that Turing machines can, but suffer from the drawback of only handling input of a fixed size. To address this we consider *circuit families* $\{C_n\}_{n\geq 1}$, where C_n is a circuit with n inputs. We say that a circuit family is polynomial size if there exists a polynomial $p(n)$ such that C_n contains no more than $p(n)$ gates. A circuit family $\{C_n\}$ is said to decide a language if any word of size n is in the language if and only if it is accepted by C_n. Finally, define a *uniform circuit family* to be a family $\{C_n\}$ such that there exists a Turing machine M that, given a read-only input tape, a work tape that is size $O(\log n)$ and a write-only output tape, will output a description of C_n on input n. Such a Turing machine is known as a *log-space transducer*. Now it is possible to show that **P** is equivalent to the set of the languages decidable by polynomial-size uniform circuit families.

1.3.2 Quantum Complexity Theory

This leads naturally to one of the most common ways to think about quantum computation: the *quantum coprocessor model*. Here a language is decided by a standard classical polytime Turing machine (sometimes also restricted to log-space) which can construct a quantum circuit, prepare its input, run it one or more times, and measure its output and use it to help decide the problem. We will clarify in a moment exactly what a quantum circuit is, but for now we can define three complexity

classes: **EQP**, **BQP** and **NQP**. For all three the classical Turing machine is restricted to polynomial time, and the number of gates of the quantum circuit (and by extension its number of bits) can only be polynomial size. The differences between the three complexity classes concern the acceptance criteria. If a language can be decided by a polynomial-size uniform quantum circuit family (or simply a *quantum algorithm*) with zero probability of error, then it belongs in **EQP**. For a number of reasons, some of which will become apparent in the next few sections, this class is not particularly useful to consider. If the probability of error is less than $\frac{1}{2} - \frac{1}{p(n)}$ for some polynomial $p(n)$, then the language is in **BQP** and if the quantum algorithm always rejects strings not in the language and has a non-zero probability of accepting strings in the language, then we say the language belongs to **NQP**.

In defining a quantum circuit, we have somewhat more latitude than with classical circuits. Instead of an operation being represented by a truth table, a quantum gate on k qubits is an element in $SU(2^k)$. Just as the Toffoli gate (flip bit 3 if bits 1 and 2 are set) is univeral for classical reversible computation ([Ben89]), arbitrary single qubit operations and a CNOT (flip bit 2 if bit 1 is set) can be shown to be computationally universal for quantum algorithms. With these operations available as fundamental quantum gates, it is straightforward to show that **BPP**$\in$**BQP** and that **NP**$\in$**NQP**.

1.4 Fault Tolerance

Error rates for solid-state electronics are on the order of 10^{-22} per gate per second, so for all practical purposes idealizing classical computers as Turing machines is fairly reasonable. Unfortunately, the intrinsic fault-tolerance of classical computers relies on continually dissipating enormous amounts of information in ways that are difficult to imagine for most schemes of quantum computation. As a result, dealing effectively with errors and decoherence is one of the primary experimental and theoretical challenges in quantum computing. We will survey a few of the main principles of fault-tolerant computing, since they provide a framework for the compiling problem.

1.4.1 Effects of small perturbations

An obvious approach to fault-tolerance is to devise fault-tolerant schemes for each individual operation and hope that the errors do not grow too large by the end of the quantum circuit. Fortunately, even imperfect quantum gates are generally unitary and so will at worst preserve the size of any perturbation to one element of the circuit even if each gate has a small error. This intuitive claim was formalized by Bernstein and Vazirani ([BV97] and [Vaz98]). However, our statement of the result oversimplifies slightly. Errors quantum gates can also take the form of decoherence or coupling with the environment, which can result in dissipative and non-unitary processes, about which all we can conclude is that their operator norm is at most one. This requires the following result:

Theorem 1 (Bernstein and Vazirani) *If $U_1, \ldots, U_m, V_1, \ldots, V_m$ are $N \times N$ matrices such that $\|U_i\|, \|V_i\| \leq 1$ and $\|U_i - V_i\| < \delta$, then $\|U_m \cdots U_2 U_1 - V_m \cdots V_2 V_1\| < m\delta$.*

Proof The idea is that if we replace a single U_i in the product $U_m \cdots U_1$ with V_i, then the entire operator will still change by only δ. Changing one operator at a time and using the triangle inequality will then give us the desired result. More formally, since we are interested in the L^∞ operator norm, we will bound the largest possible discrepancy for any starting state $|\phi_0\rangle$. Let $|\phi_k\rangle \equiv U_k \cdots U_2 U_1 |\phi_0\rangle$ denote our state after we have acted on it with the first k of the U gates and let $|\psi_k\rangle \equiv V_m \cdots V_{k+1} U_k \cdots U_1 |\phi_0\rangle = V_m \cdots V_{k+1} |\phi_k\rangle$ be the k^{th} step in a series of "hybrid" states where the U_i are replaced by V_i one at a time. The so-called hybrid argument is named after a similar technique from cryptography. Note that

$$\|U_m \cdots U_1 - V_m \cdots V_1\| = \sup \left\{ \| |\psi_m\rangle - |\psi_0\rangle \| \;\middle|\; \|\phi_0\| = 1 \right\} \tag{1.12}$$

Further, for any k,

$$\begin{aligned} \||\psi_k\rangle - |\psi_{k-1}\rangle\| &= \|V_m \cdots V_{k+1} U_k U_{k-1} \cdots U_1 |\phi_0\rangle - V_m \cdots V_{k+1} V_k U_{k-1} \cdots U_1 |\phi_0\rangle\| & (1.13)\\ &= \|(V_m \cdots V_{k+1}) \left(U_k U_{k-1} \cdots U_1 |\phi_0\rangle - V_k U_{k-1} \cdots U_1 |\phi_0\rangle\right)\| & (1.14)\\ &\leq \|U_k U_{k-1} \cdots U_1 |\phi_0\rangle - V_k U_{k-1} \cdots U_1 |\phi_0\rangle\| = \|(U_k - V_k)|\phi_{k-1}\rangle\| & (1.15)\\ &\leq \|U_k - V_k\| \leq \delta & (1.16) \end{aligned}$$

By the triangle inequality,

$$\||\psi_m\rangle - |\psi_0\rangle\| \leq \sum_{k=1}^{m} \||\psi_k\rangle - |\psi_{k-1}\rangle\| \leq m\delta, \tag{1.17}$$

completing the proof. ■

The hybrid argument is a general and broadly useful technique. A slightly more sophisticated use of it can prove that the square-root speedup of Grover's algorithm is optimal, and thus that there exists an oracle A such that $\mathbf{NP}^A \not\subseteq \mathbf{BQTIME}(o(2^{n/2}))^A$ ([Vaz98]).

To apply this to a quantum algorithm, we need to relate the operator distance to the *total variation distance*, which is defined to be the largest difference in probability distributions that can result from applying the two operators. We state without proof another simple result due to Bernstein and Vazirani:

Lemma 2 *Let $|\psi\rangle, |\phi\rangle \in \mathbb{C}^N$ such that $\langle\psi|\psi\rangle = \langle\phi|\phi\rangle = 1$ and $\||\psi\rangle - |\phi\rangle\| \leq \epsilon$. Then no measurement can distinguish $|\psi\rangle$ and $|\phi\rangle$ with probability greater than 4ϵ.*

1.4.2 Error-correcting codes

In ways analogous to classical error-correcting codes, we can encode qubits in ways that can reliably reject small independant errors. On its face, this would seem unbelievable: not only do quantum computers encode analog data, but measuring them to see if an error has occured destroys their information! It turns out that these two seeming problems actually resolve each other. By performing suitable transformations prior to measurement, we can measure the *error* without gaining any knowledge about the data. This measurement forces an outcome where there is either no error, or a known and correctable error.

The theory behind this is remarkably deep, but for this paper we are interested only in the implications of error-correction for compiling. Once we have encoded our qubits with quantum error-correcting codes, we need to modify our gates suitably both to work in the new representation and to be able to automatically reject small errors. In order to do so, it will turn out that any error-correcting code only allows a finite set of gates to be performed fault-tolerantly. A typical set of gates might be CNOT, the three Pauli gates and the S, T and H gates, which are defined by

$$S = \begin{bmatrix} e^{i\pi/4} & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}, \quad T = \begin{bmatrix} e^{i\pi/8} & 0 \\ 0 & e^{-i\pi/8} \end{bmatrix} \quad \text{and} \quad H = \frac{i}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.18}$$

These fault-tolerant gates must comprise only a discrete set, since if they were to form a continuum then we would have no guarantee that any gate we applied wasn't off by a slight amount. This means that even though physical systems typically have terms in their Hamiltonian that can be turned on and off for discrete amounts of time, by the time we are dealing with protected quantum data we only have a finite repertoire of unitary transformations available to us.

The payoff, however, is immense. The simple, but important, *threshold theorem* says that we can perform arbitrarily long computations if our error rate is below some constant that depends on our physical assumptions but that is often thought to be roughly between 10^{-4} and 10^{-6} per gate.

1.5 Universal quantum gates

Either because error-correcting codes restrict us to a discrete state of possible quantum operations, or because our quantum computer's implementation only allows certain Hamiltonians, we often find ourselves only able to apply a limited set of unitary transformations. Since quantum algorithms are typically written with the assumption that CNOTs and arbitrary single-qubit operations are possible, it is often necessary to build up reasonable approximations to the algorithm we want out of strings of the gates that we are able to apply. This is the general problem of quantum compiling.

Before discussing questions of compiling efficiently, we first need to establish when it is even possible. Certainly if the only single qubit operations we are able to apply form some finite subgroup of $SU(2)$, then there will be a limit to how well we can approximate certain elements of $SU(2)$, regardless of how long a string of base gates we compose. To make things more concrete, say that we are able to perform base gates $A_1, \ldots, A_l$ and their inverses. Then $\langle A_1, \ldots, A_l \rangle$ is the set of operations that we are able to perform. We say that $A_1, \ldots, A_l$ is a *computationally universal* set of base gates if $\langle A_1, \ldots, A_l \rangle$ is dense in $SU(2)$, or equivalently if for any $U \in SU(2)$ and any $\epsilon > 0$, we can approximate U to within a distance ϵ with some string of base gates.

Fortunately, almost all sets of base gates turn out to be computationally universal, as shown independently by [Llo95] and [DBE95]). Here *almost all* means all but a set of measure zero.

Proposition 3 *Almost any set of gates is computationally universal.*

Proof Following [Llo95], we will prove the result only using the first two gates in the set. Call them A and B and let them equal e^a and e^b for some $a, b \in su(N)$. Let $\mathcal{L}$ denote the algebra generated by a and b through commutation. Consider the set of matrices of the form

$$U = e^{at_1} e^{bt_2} e^{at_3} e^{bt_4}, \cdots \tag{1.19}$$

for some finite number of t_k. As each additional term is added, the dimension of the submanifold that can be reached by varying $t_1, t_2, \ldots$ increases by one until it reaches the dimension of $\mathcal{L}$. If $\dim \mathcal{L} = \dim SU(N)$, then the space of transformations that can be reached in $\dim \mathcal{L}$ has non-zero measure, and since $SU(N)$ is compact, the entire group can be reached within a finite multiple of $\dim \mathcal{L}$ steps.

For generic A and B, what is the dimension of $\mathcal{L}$? The only way it can fail to be $\dim SU(N) = N^2 - 1$ is if e^{at} and e^{bt} both lie in the same $(N^2 - 1)$-dimensional representation of some other Lie group with fewer generators. The classification of these Lie groups depends on N, but in general each one has only a finite number of inequivalent representations. Since each such representation has measure zero, then with probability one e^{at} and e^{bt} generate all of $SU(N)$.

However, we only have access to a discrete set of gates which we cannot apply for continuous amounts of time. With probability one, this turns out not to restrict us. For there to exist m such that $\|A^m - e^{at}\| < \epsilon$ it is only necessary that the eigenvalues of A have phases that are irrational multiples of π, and likewise for B. This is the problem of simultaneous Diophantine approximation, and for almost all choices of A there is a solution with m of order ϵ^{N-1}. Since there are $O(N^2)$ terms in (**??**) and we can choose our errors to roughly cancel out, we require $O(N^2 \epsilon^{N-1})$ gates to approximate an arbitrary gate in $SU(N)$ to a precision ϵ. ■

If we assume that quantum algorithms are written in terms of CNOTs and single-qubit operations then we can apply the above result to show the total cost of compiling. On the other hand, if we simply have a universal family of two-qubit operations from which we need to simulate CNOTs then the cost will be somewhat higher.

Proposition 4 *Any n-gate quantum algorithm that has k bits of output can be simulated with error no greater than ϵ in $O(n^4 \log k/\epsilon)$ gates from any universal family of two-qubit gates or with $O(n^2 \log 1/\epsilon)$ gates from a universal family of single-qubit gates together with a CNOT.*

Proof If we approximate each gate to an accuracy of $1/12n$, then by the hybrid argument (theorem **??**) and lemma **??**, the total program will fail no more than 1/3 of the time. If we need to generate arbitrary gates in $SU(4)$ to do so, then this will multiply our program length by $O(n^3)$ and if we have a CNOT and only need to generate arbitrary gates in $SU(2)$, then this step will only multiply our number of gates by $O(n)$. Now we can simply repeat the program $O(\log k/\epsilon)$ times and have the controlling classical computer set each bit of the final answer to whatever value that bit of the answer was set to more often in the $O(\log k/\epsilon)$ separate runs. Using a Chernoff bound, it is possible to show that this results in an error rate for each bit of the final answer of $O(\epsilon/k)$ or a probability of getting any bit wrong no greater than $O(\epsilon)$. ■

This polynomial slowdown eliminates the benefit of Grover's algorithm and often makes algorithms far too long to be practical, but at least it preserves **BQP**. As a first stab at quantum compiling, this has been a failure, but not a hopeless one.

1.5.1 Free subgroups

The above construction doesn't use nearly all of the words available to us. Define a *reduced word* in $\langle A_1, \ldots, A_l\rangle$ to be a word that contains no substrings of the form $A_i A_i^{-1}$ or $A_i^{-1} A_i$. In general there are $(2l)(2l-1)^{n-1}$ reduced words of length n with l generators. An interesting question would be to ask how many of them are typically distinct? It turns out that with probability one *every* reducted word is different. When this occurs, and the only relations among $A_1, \ldots, A_l$ are the trivial ones $A_i A_i^{-1} = A_i^{-1} A_i = I$, then we say that $\langle A_1, \ldots, A_l\rangle$ is a *free group.*

Proposition 5 *If there exists a single free subgroup of $SU(N)$ with l generators, then for $A_1, \ldots, A_l \in SU(N)$ chosen uniformly at random, $\langle A_1, \ldots, A_l\rangle$ is free with probability one.*

Proof Fix a reduced word $w(A_1, \ldots, A_l)$ of length m comprised of $A_1, \ldots, A_l, A_1^{-1}, \ldots, A_l^{-1}$ and let it act as a function from $SU(N)^l$ to $SU(N)$. If we examine the domain and range term-by-term, then w can be expressed as N^2 polynomials, $w_{ij}(A_1^{11}, \ldots, A_1^{NN}, \ldots, A_l^{11}, \ldots, A_l^{NN})$ of degree no higher than m in any variable, after we remove all relations among the A_k^{ij} that come from belonging to $SU(N)$. Obviously if $A_1 = \cdots = A_l = I$, then $w = I$. On the other hand, since there exists a free subgroup of $SU(N)$, there exist $A_1, \ldots, A_l \in SU(N)$ with no relation among them, implying that $w(A_1, \ldots, A_l) \neq I$. Thus w is not constant and at least one of the w_{ij} is a nonconstant polynomial of degree at most m. For w to equal I, we must have that w_{ij} is one if $i = j$ or zero otherwise. However, this occurs on a set of measure zero, since it corresponds to the set of solutions of a nonconstant polynomial. Thus, any finite word w evaluates to the identity on a set of measure zero.

Since there are only countably many such words, the set of $A_1, \ldots, A_l$ for which *any* non-trivial word evaluates to the identity is a countable union of measure zero sets, which also has measure zero. Thus with probability one, a random set of l elements from $SU(N)$ together with its inverses has no relations (other than of the form $A_i A_i^{-1} = I$) and thus generates a free subgroup of $SU(N)$. ■

In Chapter 4, we will demonstrate a free subgroup of $SU(2)$ and by extension $SU(N)$. **??** This will allow us to conclude that

Corollary 6 *Almost all sets of elements of $SU(N)$ generate free subgroups.*

However, products of uniform random matrices are not themselves uniform or even pairwise independant, so knowing that most matrices generate free groups doesn't give us any important information about the distribution of their strings.

Moreover, generators that are not free are dense in $SU(N)^l$. This is because every matrix is arbitrarily close to some matrix whose eigenvalues have phases that are rationally related to π. Though we will not explore this line further, it has interesting implications for the eigenvalues of the mixing operators that we use in chapter 3.

It is interesting to note that the above proof does not hold for infinite words. Define the group commutator $[U, V]$ by $[U, V] = UVU^{-1}V^{-1}$. Now for any $A_0, B_0, C_0 \in SU(N)$, define for all $k \geq 1$, $A_k = [B_k, C_k]$, $B_k = [A_k, C_k]$, and $C_k = [A_k, B_k]$. As $k \to \infty$, A_k, B_k and C_k all approach the identity for a set of A_0, B_0, C_0 with non-zero measure. The proof of this follows trivially from the Solovay-Kitaev theorem in the next chapter; in fact, constructing sequences of strings that converge to the identity on a set of non-zero measure is the underlying principle behind the sort of approximation performed by the Solovay-Kitaev theorem. Unfortunately, as we will later see, this approach is unlikely to improve on the polylogarithmic string length achieved by Solovay-Kitaev.

1.5.2 Multi-qubit compiling

For most of this paper we will discuss compiling for an arbitrary fixed number of qubits (or rather for $SU(N)$ for constant N), but with the assumption that we are only discussing one or two qubits. This is because compiling algorithms that scale to an arbitrary number of qubits have to be treated in a fundamentally different way. For one thing for n qubits the matrices and vectors involved are exponentially large in n, so we can never efficiently write down or examine more than a polynomial fraction of them. Moreover, efficient generic compiling is no longer a reasonable idea. This is seen most easily by restricting ourselves only to classical functions from n bits to 1 bit, something clearly simpler than a quantum algorithm. Such functions are defined on 2^n different inputs and thus there are a doubly exponential number of truth tables necessary to represent all such functions. This means that representing one function take 2^n bits no matter how we describe it: by writing out the truth table, by making a circuit of NAND gates, or by describing a quantum algorithm. Doing generically better than this would require universally compressing data, which is of course impossible.

Perhaps we could abandon the idea of generic unitary transformations and discuss only programs that can be written out in terms of a polynomial number of elementary gates. Then perhaps we could attempt to find a shorter program that did the same thing, or one that accomplished the same thing using our restricted set of gates, but that did so in a way that was more efficient than simply compiling each one and two qubit gate separately. It turns out that most of the simplest things that we would want our compiler to do, such as determine whether our compiled code always works the same way as the code we were given, are **NP**-complete for classical algorithms, and **NQP**-complete for quantum algorithms.

Of course, normal computers compile large programs and optimize code all the time. The key is not to look at programs as large matrices or to try to make enormous improvements, but to simply use a number of local heuristics that can be locally shown to leave the results of an algorithm unchanged. This brings up the real reasons that this paper does not discuss compiling for many qubits, though it may turn out to be an interesting problem. First, because we have to look at programs in fundamentally different ways, which depend far more on the algorithm being implemented, and second because this form of quantum compiling is not particularly different from many kinds of classical compiling, which have been worked on in some form or another for decades.

With that in mind, we will now give a constructive proof of one of the main results in quantum compiling to date, the Solovay-Kitaev theorem.

Chapter 2

The Solovay-Kitaev theorem

The constructions given for compiling in the last chapter are in many ways quite desirable: they work for any dense set of gates, can be efficiently computed on a classical Turing machine and only result in a polynomial slowdown (thus preserving the complexity class **BQP**). However, this polynomial slowdown is rather irksome. For one thing, it renders useless the quadratic speedup of quantum computers for unstructured or black box problems. For another, we know it is possible to do far better than $O(1/\epsilon)$ basic gates for a precision of ϵ. From counting the number of different ways to string together basic gates, we obtain a lower bound that is exponentially better, $O(\log 1/\epsilon)$; moreover, in the next chapter, we demonstrate non-constructively that it is always possible to saturate this lower bound up to a constant factor.

Although achieving the $O(\log 1/\epsilon)$ lower bound appears to be quite difficult, we can still do much better than $O(1/\epsilon)$ by exploiting the nonabelian structure of $SU(N)$. Using some simple observations about commutators, the Solovay-Kitaev theorem demonstrates that it is possible to generically compile to precision ϵ with $O(\log^c(1/\epsilon))$ gates, where c is a constant that varies between 2 and 4 depending on the implementation. This *polylogarithmic* slowdown is often a quite reasonable cost, as it leaves any polynomial quantum speedup intact.

2.1 Statement of the theorem

The theorem was proved independently by Solovay (in an unpublished manuscript) and Kitaev ([Kit97]). One other derivation exists in [NC00] and elaborated in unpublished work done by this author with Xinlan Zhou, Ben Recht and Isaac Chuang. For the most part, this is the one that we follow. The precise statement of the theorem is as follows:

Theorem 7 (Solovay-Kitaev) *For any $A_1, \ldots, A_l \in SU(N)$ such that $\langle A_1, \ldots, A_l \rangle$ is dense in $SU(N)$, there exists a constant C and a procedure for approximating any $U \in SU(N)$ to precision ϵ with a string of $A_1, \ldots, A_l$ and their inverses of length no greater than $C \log^c(1/\epsilon)$, where $c \approx 4$ and C is independant of U and ϵ. This procedure can be implemented in time polynomial in $\log(1/\epsilon)$.*

We will prove this theorem backwards, beginning with its implications, an overview of its proof and the resulting algorithm, and then proceeding backwards into the gritty details and lemmas that constitute it and verify its correctness. First we state a corollary about what the theorem means for quantum circuits.

Corollary 8 *For any family of universal gates, there exists a constant C such that any quantum circuit with n arbitrary gates can be constructed from fewer than $Cn \log^c(n) \log(1/\delta)$ universal gates if the probability of error is δ.*

Proof The corollary is proved in the same way as proposition **??**. After reducing the error per gate to $O(1/n)$ with a now polylogarithmic cost, the error for the entire circuit becomes $O(1)$ and classical probability amplification becomes possible. ■

The highest-level picture of the Solovay-Kitaev theorem is that of successive approximation. We will use a crude constant-time strategy to make an initial approximation to our desired gate, then will attack the remaining distance with techniques that rely on being near the identity. Specifically, near the identity the group commutator (which is easily expressible with strings of matrices) approaches the algebra commutator (which is easily calculable), and by moving back and forth between these two ways of looking at operators we can express precise matrices near the identity as strings of less precise matrices that are farther from the identity.

Before continuing, we need to define a few terms. If A and B are sets of points, then say that A is an ϵ-net for B if $\forall b \in B, \exists a \in A$ such that $|a - b| < \epsilon$. Define $S(\epsilon)$ to be the set of matrices within a distance ϵ of the identity. To make our analysis more concrete we will prove the theorem for the $SU(2)$ case. To this end, define $u(\vec{x}) \equiv e^{i\vec{\sigma}\cdot\vec{x}}$. Also, since $SU(2)$ is a finite dimensional space, all norms can be related to each other by constant factors. Instead of the L^∞ norm, we will find it more convenient to work with the L^2 or *trace norm*, defined to be $\|M\| = \mathrm{Tr}\,\sqrt{MM^\dagger}$. If elements of $SU(2)$ are associated with points on the surface of the 3-sphere, then this is simply equivalent to twice the Euclidean distance. This implies a natural operator distance $D(U, V) \equiv \|U - V\|$, which we will often refer to simply as precision. Finally, we will use the commutator $[U, V] \equiv UVU^\dagger V^\dagger$ both for members of $SU(2)$ and $su(2)$.

2.2 Algorithm

We are now ready to begin a constructive proof of the theorem. The strategy will be to construct a series of epsilon nets, $\Gamma_0, \Gamma_1, \ldots$ such that Γ_0 is an ϵ_0-net for $SU(2)$ and for $k > 0$, Γ_k is an $\epsilon(k)^2$-net for $S(\epsilon(k))$. For consistency, let $\epsilon(0)^2 = \epsilon_0$. The initial net Γ_0 can be created in constant time and below we will discuss the details of doing so. Afterwards, every other net can be constructed recursively, by defining $\Gamma_k = \{ABA^\dagger B^\dagger | A, B \in \Gamma_{k-1}\}$. We will prove tighter bounds below, but in general each iteration squares the epsilon of the net and multiplies its string length by four, so Γ_k has strings of length $O(4^k)$ and $\epsilon(k)^2 = O(\epsilon_0^{2^k})$. This suggests an obvious means of performing successive approximation: as long as $\epsilon(k) > \epsilon^2(k-1)$ we can find the closest gate in Γ_0 to our target, then the closest gate in Γ_1 to the remaining difference, and so on, until after k steps we have a string of length $O(4^k)$ and an approximation of accuracy $O(\epsilon_0^{2^k})$. Thus k must be $O(\log\log(1/\epsilon))$. This indeed obtains a polylogarithmic string length for generic sets of gates, but there is a hidden exponential computational cost. Since the amount of volume in a small ball of $SU(2)$ scales roughly as r^3, an ϵ-net for $S(\delta)$ cannot contain any fewer than $O((\delta/\epsilon)^3)$ points. Since Γ_k is an $\epsilon^2(k)$-net for $S(\epsilon(k))$, it must contain $O(\epsilon^3(k))$, or $O(\epsilon_0^{3\cdot 2^k})$ points. For $k = O(\log\log(1/\epsilon))$, this is a number of points exponential in $\log(1/\epsilon)$, even if we only take as few commutators as are necessary to fill the patch that we need to guarantee convergence.

The key trick in constructing an efficient algorithm is to note that since the Γ_k are constructed recursively, we can generate elements of them on the fly instead of precomputing all the points in the net. This is accomplished by the following procedure:

> Subroutine `FACTOR(`C`)` returns A, B such that $\vec{a} \times \vec{b} = \vec{c}$ and $\vec{a} \perp \vec{b}$, where $A = u(\vec{a})$, $B = u(\vec{b})$ and $C = u(\vec{c})$.
>
> Procedure $[A, B]$ =`FACTOR(`C`)`
> Write $\vec{c} = (c_1, c_2, c_3)$ in polar coordinates
> $c_1 = c\sin(\theta)\cos(\phi)$, $c_2 = c\sin(\theta)\sin(\phi)$, $c_3 = c\cos(\theta)$
> Then let $\vec{a}_p \equiv (\sqrt{c}, \theta + \pi/2, \phi)$ and $\vec{b}_p \equiv (\sqrt{c}, \pi/2, \phi + \pi/2)$

Return $[u(\vec{a}), u(\vec{b})]$

The correctness of this will be proved below. For now note that $\|u(\vec{x}) - I\| = 2\sin(|x|/2)$.

We will also need define a procedure ϵ_0-APPROX(U) that takes constant time and returns a string of bounded length that is an ϵ_0 approximation to U. Note that this is the only place in the algorithm where we reference the base gates that we are working with; from now on they are abstracted away as simply generators of an ϵ_0 net.

Now we can finally define the main procedure:

Function $\texttt{SK}(U, n)$ takes in any $U \in SU(2)$ and returns $V \in \langle A_1, \ldots, A_l \rangle$ such that $\|U - V\| < \epsilon^2(n)$.

Function $V =\texttt{SK}(U, n)$
If $n = 0$
    $V = \epsilon^2(0)$-$\texttt{APPROX}(U, G_I)$.
Else
    $W = \texttt{SK}(U, n-1)$
    $[A, B] =\texttt{FACTOR}(UW^\dagger)$
    $V = [\texttt{SK}(A, n-1), \texttt{SK}(B, n-1)]W$
End
Return V.

We claim that this algorithm will work correctly if $\epsilon(k)^2 < \epsilon(k+1)$ for all k, which will always hold when $\epsilon_0 < \epsilon_c$ for some critical value ϵ_c. Continuing to work backwards, we will defer choosing ϵ_c until the end of the proof.

By definition $\texttt{SK}(U, 0)$ returns a $\epsilon^2(0)$-approximation to U. The proof of the algorithm's correctness now proceeds by induction. Assume $\texttt{SK}(U, n-1)$ returns an $\epsilon^2(n-1)$-approximation of U. We now want to show that $\texttt{SK}(U, n)$ is an $\epsilon^2(n)$-approximation of U, where $\epsilon^2(n) = C\epsilon^3(n-1)$ for some constant C. Note when $W =\texttt{SK}(U, n-1)$, $\|UW^\dagger - I\| < \epsilon^2(n-1)$. Let $u(\vec{c}) = UW^\dagger$, and using $\|u(\vec{c}) - I\| = 2\sin(c/2)$, we have $c < \epsilon^2(n-1) + O(\epsilon^6(n-1))$. Let $[A, B] =\texttt{FACTOR}(UW^\dagger)$, and $u(\vec{a}) = A, u(\vec{b}) = B$. It follows that $a, b = \sqrt{c} < \epsilon(n-1) + O(\epsilon^5(n-1))$. Let $u(\vec{a}') = A' =\texttt{SK}(A, n-1)$, and $u(\vec{b}') = B' =\texttt{SK}(B, n-1)$. Since A', B' are the $\epsilon^2(n-1)$-approximations to A and B respectively, we have $a', b' < \epsilon(n-1) + \epsilon^2(n-1)$. The output from $\texttt{SK}(U, n)$ is $[A', B']W$, and

$$\begin{aligned} & \|[A', B']W - U\| && (2.1)\\ = \; & \|[A', B'] - UW^\dagger\| && (2.2)\\ = \; & \|[u(\vec{a}'), u(\vec{b}')] - u(\vec{c})\| && (2.3)\\ \leq \; & \|[u(\vec{a}'), u(\vec{b}')] - u(\vec{a}' \times \vec{b}')\| + \|u(\vec{a}' \times \vec{b}') - u(\vec{c})\|, && (2.4) \end{aligned}$$

Using the facts (proved in the next subsection) that given $a, b < \delta$,

$$\|[u(\vec{a}), u(\vec{b})] - u(\vec{a} \times \vec{b})]\| \leq d_1\delta^3 + O(\delta^4), \tag{2.5}$$

$$\|u(\vec{a}) - u(\vec{b})\| \leq \|\vec{a} - \vec{b}\| + d_2\delta^3 + O(\delta^4) < \|veca - \vec{b}\|, \tag{2.6}$$

for some constants $d_1 > 0$ and $d_2 < 0$, we have

$$\|[u(\vec{a}'), u(\vec{b}')] - u(\vec{a}' \times \vec{b}')\| = d_1\epsilon^3(n-1) + O(\epsilon^4(n-1)) \tag{2.7}$$

and

$$\begin{aligned} \|u(\vec{a}' \times \vec{b}') - u(\vec{c})\| &= \|u(\vec{a}' \times \vec{b}') - u(\vec{a} \times \vec{b})\| && (2.8)\\ &= \|\vec{a}' \times \vec{b}' - \vec{a} \times \vec{b}\| + O(\epsilon^6(n-1)) && (2.9)\\ &= 2\epsilon^3(n-1) + O(\epsilon^4(n-1)) && (2.10) \end{aligned}$$

Therefore,

$$\|U - [A', B']W\| \leq (d_1 + 2)\epsilon^3(n-1) + O(\epsilon^4(n-1))\,. \tag{2.11}$$

By choosing $C = d_1 + 2$, $\epsilon^2(n) = C\epsilon^3(n-1)$, $\mathtt{SK}(U, n)$ returns an $\epsilon^2(n)$-approximation of $U \in SU(2)$.

2.3 Efficiency of the algorithm

In this subsection, we analyze the efficiency of the above algorithm. We first show that when the algorithm converges, finding an ϵ-approximation to any $U \in SU(2)$ requires $O(\log^\alpha(1/\epsilon))$ calls to ϵ_0-APPROX with $\alpha \approx 2.7$.

$\mathtt{SK}(U, n)$ is an recursive algorithm. When $n > 0$, $\mathtt{SK}(U, n)$ calls $\mathtt{SK}(U, n-1)$ three times, and $\mathtt{SK}(U, 0)$ calls ϵ_0-APPROX once. Thus, to compute $\mathtt{SK}(U, n)$, it requires $1 + 3 + 3^2 + \cdots + 3^n = (3^{n+1} - 1)/2$ calls to ϵ_0-APPROX.

Given $\epsilon^2(n) = C\epsilon^3(n-1)$, we have

$$\epsilon(n) = \frac{(C\epsilon(0))^{(3/2)^k}}{C}\,. \tag{2.12}$$

For $\epsilon^2(n) \leq \epsilon$, it is sufficient to choose n such that

$$\left(\frac{3}{2}\right)^n \geq \frac{\log(C^2\epsilon)}{2\log(C\epsilon(0))} \tag{2.13}$$

Thus, the running time is proportional to $3^{n+1} = \left[\frac{\log(C^2\epsilon)}{2\log(C\epsilon(0))}\right]^{\log 3/\log(3/2)} = O(\log^\alpha(1/\epsilon))$ with $\alpha = \log 3/\log(3/2) \approx 2.7$.

Next we show that the algorithm returns an ϵ-approximation of $U \in SU(2)$ using $O(\log^\beta(1/\epsilon))$ basic gates with $\beta \approx 4.0$. Let $\langle A_1, \ldots, A_l\rangle_{l_0}$ be an $\epsilon^2(0)$-net for $SU(2)$, and l_n be the maximum length of the gate returned by $\mathtt{SK}(U, n)$. Then $l_n \leq 5l_{n-1} \leq 5^n l_0$. Using n satisfying Eq. (**??**), we have

$$l_{n(\epsilon)} \leq 5^n l_0 \leq l_0 \left[\frac{\log(C^2\epsilon)}{2\log(C\epsilon(0))}\right]^{\log 5/\log(3/2)} = O(\log^\beta(1/\epsilon)), \tag{2.14}$$

where $\beta = \log 5/\log(3/2) \approx 4.0$.

Thus, the proposed algorithm is efficient in both the time resource and the length of the approximation.

2.4 The initial epsilon net

For the algorithm to converge, the initial ϵ_0-net should have ϵ_0 sufficiently small, such that $\epsilon(n)$ decreases quickly with increasing n. From Eq. (**??**), this requires $C\epsilon(0) < 1$. In this section, we find the sufficient value of $\epsilon(0)$ for the algorithm to converge.

Lemma 9 *Given $a, b < \delta$,*

$$\|[u(\vec{a}), u(\vec{b})] - u(\vec{a} \times \vec{b})]\| = d_1\delta^3 + O(\delta^4)\,, \tag{2.15}$$

with $d_1 \leq \frac{4}{3\sqrt{3}}$.

Proof Let $A = i\vec{\sigma}\cdot\vec{a}$ and $B = i\vec{\sigma}\cdot\vec{b}$. Then $e^{-iA} = u(\vec{a})$ and $e^{-iB} = u(\vec{b})$. Also note that $e^{-[A,B]} = u(\vec{a}\times\vec{b})$. Then given $a, b < \delta$

$$\begin{aligned}
[e^{-iA}, e^{-iB}] &= e^{-iA}e^{-iB}e^{iA}e^{iB} && (2.16)\\
&= I - [A,B] + \frac{i}{2}[A+B, A^2+2AB+B^2] + O(\delta^4) && (2.17)
\end{aligned}$$

and

$$e^{-[A,B]} = I - [A,B] + O(\delta^4) \quad (2.18)$$

Therefore,

$$\begin{aligned}
M &\equiv [e^{-iA}, e^{-iB}] - e^{-[A,B]} && (2.19)\\
&= \frac{i}{2}[A+B,[A,B]] + O(\delta^4) && (2.20)\\
&= -\frac{1}{8}[(\vec{a}+\vec{b})\cdot\vec{\sigma}, (\vec{a}\times\vec{b})\cdot\vec{\sigma}] + O(\delta^4) && (2.21)\\
&= -\frac{i}{4}[(\vec{a}+\vec{b})\times(\vec{a}\times\vec{b})]\cdot\vec{\sigma} + O(\delta^4), && (2.22)
\end{aligned}$$

where $\vec{\sigma}$ is defined to be $\hat{x}\sigma_x + \hat{y}\sigma_y + \hat{z}\sigma_z$. Let $\lambda_{1,2}$ be the two eigenvalues of M. $|\lambda_{1,2}| = \frac{1}{4}\|(\vec{a}+\vec{b})\times(\vec{a}\times\vec{b})\| + O(\delta^4)$ ([HJ]) Let the angle between $\vec{a}$ and $\vec{b}$ be θ. Therefore

$$\begin{aligned}
\|(\vec{a}+\vec{b})\times(\vec{a}\times\vec{b})\| &= \|(\vec{a}+\vec{b})\|ab|\sin(\theta)| && (2.23)\\
&= \sqrt{a^2+b^2+2ab\cos(\theta)}ab|\sin(\theta)| && (2.24)\\
&\leq \sqrt{2}\delta^3\sqrt{1+|\cos(\theta)|}|\sin(\theta)| && (2.25)\\
&\leq \frac{8}{3\sqrt{3}}\delta^3, && (2.26)
\end{aligned}$$

and

$$\begin{aligned}
&\|[u(\vec{a}), u(\vec{b})] - u(\vec{a}\times\vec{b})\| && (2.27)\\
= \;& \mathrm{tr}|\sqrt{MM^\dagger}| = |\lambda_1| + |\lambda_2| \leq \frac{4}{3\sqrt{3}}\delta^3 + O(\delta^4) && (2.28)
\end{aligned}$$

Thus, $d_1 \leq \frac{4}{3\sqrt{3}} \leq 0.77$. ■

Since C is chosen to be $d_1 + 2 \leq 2.77$, for $C\epsilon(0) < 1$, it is sufficient to have $\epsilon(0) < 1/C = 0.361$. Thus, an initial $\epsilon^2(0)$-net with $\epsilon^2(0) \leq \epsilon_c = 0.13$ is sufficient for the approximation algorithm to converge.

Finally we prove a lemma used in the last section.

Lemma 10 *Given* $a, b < \delta$,

$$\|u(\vec{a}) - u(\vec{b})\| \leq \|\vec{a}-\vec{b}\| + d_2\delta^3 + O(\delta^4), \quad (2.29)$$

for some constant $d_2 < 0$.

Proof:

$$\begin{aligned} M &= u(\vec{a}) - u(\vec{b}) && (2.30)\\ &= -i(A-B) - \frac{A^2-B^2}{2} + i\frac{A^3-B^3}{6} + O(\delta^4) && (2.31)\\ &= -i(A-B) - \frac{(a^2-b^2)I}{8} + i\frac{a^2A-b^2B}{24} + O(\delta^4)\,, && (2.32)\end{aligned}$$

where we have used $(2A)^2 = (\vec{a}\cdot\vec{\sigma})^2 = a^2 I$. Thus

$$\begin{aligned} MM^\dagger &= (A-B)^2 - \frac{(A-B)(a^2A-b^2B)}{24} && (2.33)\\ &- \frac{(a^2A-b^2B)(A-B)}{24} + \left(\frac{a^2-b^2}{8}\right)^2 I + O(\delta^5) && (2.34)\\ &= (\frac{1}{4}\|\vec{a}-\vec{b}\|^2 + \beta)I + O(\delta^5)\,, && (2.35)\end{aligned}$$

where $\beta = -\frac{a^4+b^4}{3\cdot 2^4} + \frac{(a^2-b^2)^2}{2^6} - \frac{(a^2+b^2)\vec{a}\cdot\vec{b}}{3\cdot 2^4} \leq 0$. Let $\alpha = \|\vec{a}-\vec{b}\|$, and $\sqrt{MM^\dagger} = \sqrt{(\frac{1}{4}\alpha^2+\beta)I + O(\delta^5))}$. Then $\|u(\vec{a}) - u(\vec{b})\| = \mathrm{tr}|\sqrt{MM^\dagger}| = \alpha\sqrt{1+4\beta/\alpha^2 + O(\delta^3)}$ Since $\beta \leq 0$, we obtain $\|u(\vec{a}) - u(\vec{b})\| \leq \alpha + d_2\delta^3 + O(\delta^4)$, where $d_2 \leq 0$.

2.5 Determining Epsilon

The above arguments prove the convergence of our algorithm given the assumption that ϵ_0-APPROX(U) will always return an approximation that is accurate to within ϵ_0. Since ϵ_0-APPROX(U) simply searches through an ϵ-net G for the closest operator to U, an equivalent problem is to determine ϵ, given G, and verify that $\epsilon < \epsilon_c$. More formally, if $G = \{g_1, \ldots g_n\}$, then we define

$$f(x) \equiv \inf_i |x - g_i| \qquad \text{and} \qquad \epsilon = \sup_{x\in SU(2)} f(x) \tag{2.36}$$

Since our methods for constructing G do not easily lend themselves to analytic calculations of ϵ, we will resort to proving that G is an ϵ-net by computerized enumeration of cases. On its face, finding the maximum value of $f(x)$ does not seem particularly straightforward, as there are a continous range of choices for $x \in SU(2)$ to check. However since $SU(2)$ is compact, it will suffice to check all the local maxima of $f(x)$ and since $f(x)$ is differentiable (along some direction) almost everywhere, we expect the local maxima to have measure zero and likely be a finite set. Say that x is an *extreme* point if it is a *global* maximum, i.e.

$$\forall y \in SU(2), f(x) \geq f(y) \tag{2.37}$$

With the following lemma, we prove that the extreme points are all contained within a set of size $\binom{n}{4}$.

Lemma 11 *Any extreme point is equidistant from its four closest neighbors in G.*

Proof Let x be an extreme point. Order the elements of G by their distance to x, so that $f(x) = |x-g_1| \leq |x-g_2| \leq \ldots$. First, we proceed by contradiction and suppose that $|x-g_2| = |x-g_1|+\delta$, for some $\delta > 0$. Define $B(a,\epsilon)$ to be the open ball of radius ϵ centered on a. Then for all $x' \in B(x, \delta/2)$, $|x'-g_1| = \inf_i |x'-g_i| = f(x')$. However, since distance to g_1 has a local minimum only at g_1 (which must be different from x if x is to be an extreme point), there must exist some $x' \in B(x,\delta/2)$ such that $|x'-g_1| > |x-g_1|$, which is impossible, since x is an extreme point. Thus, our assumption that

$|x - g_2| \neq |x - g_1|$ must have been false, and we have that all extreme points are equidistant from their two closest neighbors.

Similar arguments will also apply to g_3 and g_4. If we consider elements of $SU(2)$ to be points on the unit 3-sphere, then the set of points equidistant from g_1 and g_2 is simply the hyperplane given by

$$E(g_1, g_2) \equiv \left\{ y \,\middle|\, |y - g_1| = |y - g_2| \right\} = \left\{ y \,\middle|\, y \cdot (g_1 - g_2) = \left(\frac{g_1 + g_2}{2} \right) \cdot (g_1 - g_2) \right\} \tag{2.38}$$

Now, if $|x - g_3| = |x - g_1| + \delta$, we consider distance from g_3 on the domain $E(g_1, g_2) \cap B(x, \delta/2)$ and again find that δ must be zero. Thus, x is located on $E(g_1, g_2, g_3)$, the plane that is equidistant from g_1, g_2 and g_3. Repeating this process one last time, we find that x is in $E(g_1, g_2, g_3, g_4)$, which is a line passing through the origin. However, x is also on the unit sphere, which when intersected with $E(g_1, g_2, g_3, g_4)$ will constrain x to a pair of points. Since we can no longer construct an open ball within $E(g_1, g_2, g_3, g_4)$, we stop here. ■

This suggests an obvious algorithm for determining ϵ. For every different way that we can select four points, consider the x and $-x$ that are equidistant from all four as possible extreme points. Verify that these possible extreme points are indeed closer to the four points that they're equidistant from than to any other points in the net and then ϵ will be the greatest minimum distance of any extreme point. Since there are $O(n^4)$ ways to choose 4 points and for each possible extreme point we need to check its distance against every point in the net, the total running time is $O(n^5)$. This is polynomial time and exact, but not really feasible for $n > 100$ or so.

In order to make this more efficient, we instead divide up $SU(2)$ into p different patches and iterate through each patch, considering only extreme points that fall inside it. This allows us to run smaller versions of the $O(n^5)$ check on only the points that are close enough to the patch to make a difference. To determine this, we still need to check each point's minimum distance to a patch, so the total running time is $O(p(n + m^5))$, where m is the average number of points that are near a patch. On the other hand, if we already have a lower bound on ϵ, say from examining a small number of extreme points, then whenever an entire patch is within ϵ of some point in our net, then we know that no extreme point within the patch can affect ϵ and we can skip the entire patch.

The final algorithm is then as follows. Initialize ϵ to zero and increase it whenever we find an extreme point that achieves a higher value of ϵ. For each patch, loop through all points in the net and compute upper and lower bounds on their distance to the patch. This part of the algorithm takes $O(pn)$ time. If any point's upper bound for distance to the patch is less than ϵ then reject the entire patch. If the upper bound for point A is less than the lower bound for point B then we can safely reject point B. To perform this efficiently, we make a single sweep through all n upper bounds and record the lowest one; call it d. Then we make another sweep through all n points, rejecting those with lower bound greater than d. This processing is still part of the $O(pn)$ part. At this point, if there are m points left, then we perform an $O(m^5)$ check on them, considering only extreme points that fall within the patch we are currently considering. This step can loosely be said to have a running time of $O(pm^5)$, although of course m varies from patch to patch.

This implementation also lends itself readily to parallel processing. A server can allocate different patches to different machines, which need only communicate the indices of extreme points that achieve new values of epsilon. Since total execution time can vary wildly based on CPU speed, process priority and the number of clients for a particular ϵ-net, we instead measure execution time in terms of the number of calls to the distance function. As the number of points grew from 4,000 to 100,000 the number of distance calls ranged from a little over a billion to over 500 billion, which demonstrates roughly quadratic asymptotic behavior.

2.6 Results

Before running the Solovay-Kitaev algorithm, we need to construct an intial epsilon net. This was accomplished by generating all strings of bounded length made up of T and H gates, which were defined in (**??**). We then calculated ϵ for each net. As loosely predicted in the next chapter, we found that ϵ decreased exponentially with string length.

Given an initial epsilon net, we then ran the algorithm on a number of random matrices to different levels of recursion and plotted the resulting string length against the approximation accuracy achieved. We expect a polynomial dependence of string length on $\log 1/\epsilon$ and this is confirmed a the linear fit on the log plot with a slope of roughly 4. The final data points on this plot are below the fit because of the

limitations of machine precision.

2.7 Generalization

Although the proofs and algorithms in the section are specialized to the $SU(2)$ case, this was largely done in order to achieve precise results for the constant factors involved. Everything generalizes readily to $SU(N)$ and indeed to any compact Lie group with the same group/algebra commutator property as $SU(N)$, although there is a prefactor that is exponential in N corresponding to the cost of creating the inital ϵ_0-net.

Kitaev proposed a different algorithm, which is asymptotically more efficient; it uses $O(\log^2(1/\epsilon))$ gates. However, the constant prefactors are enough to make the requisite level of computation impractical for modern computers. This chapter was intended to be a proof of principle. Any actual quantum computer will likely use some more complicated compiler that takes into account the limitations of the actual system instead of being generically and asymptotically powerful. Still, generic results such as the Solovay-Kitaev theorem are a useful starting place for any such technique.

Chapter 3

Almost all quantum gates are efficiently universal

The Solovay-Kitaev theorem proves that almost all families of quantum gates can cover $SU(N)$ with a cost polylogarithmic in $1/\epsilon$, but it makes no claims about whether we can do better. On the other hand, we will demonstrate a family of the gates in the next chapter that achieves a provably optimal string length that is only logarithmic in $1/\epsilon$. The goal of this chapter will be to close the gap between the upper and lower bounds for generic families of quantum gates. We find that that all dense families of gates can achieve an ϵ-net with string length $O(\log 1/\epsilon)$, but that it is impossible to bound the prefactor with probability one.

3.1 Background

Proving the main result of this section will use some basic principles from functional analysis, reviews of which can be found in [AG] and [Halm]. Much of finite-dimensional linear algebra is directly applicable to dealing with linear operators on Hilbert spaces, but there are a few places where we need to be careful. This section will review some of these principles and lay out the definitions used in the rest of the chapter.

Define dg to be a Haar measure on $SU(N)$ normalized so that $\int dg = 1$. Consider the Hilbert space $L^2(SU(N))$ with norm defined by the usual inner product $\langle \psi, \varphi \rangle \equiv \int \bar{\psi}(g)\varphi(g)dg$. We denote the set of linear transformations (or endomorphisms) on $L^2(SU(N))$ by $\mathrm{End}(L^2(SU(N)))$. Using the L^2 Hilbert space norm we define an L^∞ operator norm for $\mathrm{End}(L^2(SU(N)))$ (linear operators on $L^2(SU(N))$) by

$$\|M\|_\infty = \sup\left\{ \|Mf\| \,\middle|\, f \in L^2(SU(N)), \|f\| = 1 \right\} \tag{3.1}$$

Let Δ be the usual Laplacian operator on $L^2(SU(N))$, defined by $\Delta f(\vec{x}) = \sum_i \frac{\partial^2 f(\vec{x})}{\partial^2 x_i}$. Under Δ, our Hilbert space decomposes into a countable direct sum of finite-dimensional orthogonal invariant eigenspaces, which for $L^2(SU(2))$ are just copies of the familiar spaces of spherical harmonics. This means that any $M \in \mathrm{End}(L^2(SU(N)))$ that commutes with Δ will also leave these subspaces invariant. Since M can be considered a direct sum of its restrictions to a series of finite-dimensional vector spaces, then we can analyze it with techniques that apply to finite dimensional matrices. In particular, if M is also hermitian then M is diagonalizable on each subspace and its L^∞ norm is simply the supremum of its spectrum. So,

$$\|M\|_\infty = \sup\left\{ |\lambda| \,\middle|\, \exists f \neq 0 \text{ s.t. } Mf = \lambda f \right\} \tag{3.2}$$

and as a result,

$$\|M^n\|_\infty = \|M\|_\infty^n \tag{3.3}$$

For any $U \in SU(N)$, let $\tilde{U}$ be a linear functional on $L^2(SU(N))$ defined by $\tilde{U}f(x) = f(Ux)$. Note that $\tilde{U}$ fixes the constant function, commutes with Δ, is hermitian and has no eigenvalues with norm greater than one, so $\|\tilde{U}\| = 1$. For any $A_1, \ldots, A_l \in SU(N)$, define the mixing operator $T(A_1, \ldots, A_l) \in \mathrm{End}(L^2(SU(N)))$ by

$$T(A_1, \ldots, A_l) = \frac{1}{2l} \sum_{i=1}^{l} \tilde{A}_i + \tilde{A}_i^{-1} \tag{3.4}$$

All such T are hermitian, commute with Δ, and by the triangle inequality have no eigenvalues greater than one. Where there is no ambiguity, we will often simply write T instead of $T(A_1, \ldots, A_l)$.

Denote the set of words of length n made up of $A_1, \ldots, A_l, A_1^{-1}, \ldots, A_l^{-1}$ by $W_n(A_1, \ldots, A_l)$, or when the $A_1, \ldots, A_l$ are understood, sometimes simply W_n. This set has $(2l)^n$ elements, though not all are distinct, since substrings such as $A_1 A_1^{-1}$ and $A_2 A_2^{-1}$ are equivalent. For any integer n, expanding T^n gives

$$T^n = \sum_{w \in W_n} \frac{\tilde{w}}{(2l)^n} \tag{3.5}$$

Define $P \in \mathrm{End}(L^2(SU(N)))$ to be the integration operator on $L^2(SU(2))$, or the projection operator onto the space of constant functions, i.e.

$$Pf(g) = \int_{SU(N)} f(g) dg \tag{3.6}$$

Clearly P is also hermitian and commutes with T. In fact, $TP = P$ and $P^2 = P$, so

$$(T-P)^n = T^n + \sum_{k=1}^{n} \binom{n}{k} (-1)^k P \qquad \text{and} \qquad 0 = (P-P)^n = \sum_{k=0}^{n} \binom{n}{k} (-1)^k P \tag{3.7}$$

Subtracting the second equation from the first gives that

$$(T-P)^n = T^n - P \tag{3.8}$$

Finally, we define $\Lambda(A_1, \ldots, A_l)$ by

$$\Lambda(A_1, \ldots, A_l) = \|T(A_1, \ldots, A_l) - P\|_\infty \tag{3.9}$$

3.2 Some gates are efficiently universal

Lemma 12 (Lubotsky, Phillips and Sarnak) *Let*

$$V_1 = \frac{1 + 2i\sigma_x}{\sqrt{5}}, \quad V_2 = \frac{1 + 2i\sigma_y}{\sqrt{5}} \text{ and, } \quad V_3 = \frac{1 + 2i\sigma_z}{\sqrt{5}}. \tag{3.10}$$

Let $d = \Lambda(V_1, V_2, V_3)$. *Then* $d = \frac{\sqrt{5}}{3} < 1$.

Proof The proof is due to Lubotsky, Phillips and Sarnak ([LPS86] and [LPS87]). It relies on Deligne's proof of the Weil conjecture regarding asymptotic coefficients of modular forms and is somewhat beyond the scope of this paper, though some of the number theoretic background will be discussed in the next chapter. ■

This shows that there exists a family of quantum gates for $SU(2)$ whose mixing operator T has a second largest eigenvalue that is strictly less than one. We will later demonstrate that this means that

almost all families of quantum gates in $SU(2)$ require only $O(\log 1/\epsilon)$ gates to approximate anything else. However we will require slightly more effort to extend the result to $SU(N)$. To this end, if I_k denotes the $k \times k$ identity matrix, then, for any $U \in SU(2)$ and $2 \leq j \leq N$, define $\beta_j^{(N)}(U)$ to be

$$\beta_j^{(N)}(U) = \begin{pmatrix} I_{j-2} & 0 & 0 \\ 0 & U & 0 \\ 0 & 0 & I_{N-j} \end{pmatrix} \in SU(N) \tag{3.11}$$

We will typically omit the N where it is understood.

Lemma 13 (Diaconis and Shahshahani) *Let $\{G_j^i\}, 1 \leq i < j \leq N$ be a series of $\binom{n}{2}$ independent random matrices in $SU(2)$ that are chosen uniformly according to a Haar measure. Then*

$$\left(\beta_n(G_n^1) \cdot \beta_{n-1}(G_{n-1}^1) \cdots \beta_2(G_2^1)\right) \cdots \left(\beta_n(G_n^{n-2}) \cdot \beta_n(G_n^{n-1})\right) \beta_n(G_n^{n-1}) \tag{3.12}$$

is uniformly distributed in $SU(N)$.

Proof The idea behind the proof is fairly straightforward. In $\mathbb{R}^3$ one can generate a uniform $SO(3)$ rotation by first performing a random rotation in the (x, y) plane and then rotating the z axis by a random amount. The generalization of this intuition to complex numbers and N dimensions is found in [DS87]. ■

Proposition 14 *Let V_1, V_2, V_3 be defined as in (**??**). For any $N > 2$, let $k = 3(N-1)$ and define $U_1, \ldots, U_k$ by*

$$\begin{array}{ccc} U_1 = \beta_2(V_1), & U_2 = \beta_2(V_2), & U_3 = \beta_2(V_3), \\ U_4 = \beta_3(V_1), & U_5 = \beta_3(V_2), & U_6 = \beta_3(V_3), \\ \vdots & \vdots & \vdots \\ U_{k-2} = \beta_n(V_1), & U_{k-1} = \beta_n(V_2), & U_k = \beta_n(V_3). \end{array} \tag{3.13}$$

Then $\Lambda(U_1, \ldots, U_k) < 1$.

Proof For $2 \leq j \leq N$, define the set γ_j by

$$\gamma_j \equiv \left\{U_{3(j-2)+1}, U_{3(j-2)+2}, U_{3(j-2)+3}, U_{3(j-2)+1}^\dagger, U_{3(j-2)+2}^\dagger, U_{3(j-2)+3}^\dagger\right\} \tag{3.14}$$

Using this, we define R_m to be the subset of $W_{m\binom{n}{2}}$ where all words have the form given in lemma **??**. In other words,

$$R_m = \left\{G_n^{1,1} \cdots G_n^{1,m} G_{n-1}^{1,m} \cdots G_{n-1}^{1,m} \cdots G_2^{1,1} \cdots G_2^{1,m} G_n^{2,1} \cdots G_n^{n,m} \,\middle|\, G_j^{i,m} \in \gamma_j\right\} \tag{3.15}$$

From lemma **??** we have that $\|T^n(U_{3j+1}, U_{3j+2}, U_{3j+3}) - \beta_{j+2}(P)\| = \|T(V_1, V_2, V_3) - P\| = d^n$ for some $d < 1$. Thus, using the hybrid argument (theorem **??**) and lemma **??** gives that

$$\Lambda(R_m) = \left\| \sum_{w \in R_m} \frac{\tilde{w}}{|R_m|} - P \right\| \leq \binom{n}{2} d^m \tag{3.16}$$

Now,

$$
\begin{aligned}
(\Lambda(U_1,\ldots,U_k))^{m\binom{n}{2}} &= \|T(U_1,\ldots,U_k)-P\|^{m\binom{n}{2}} = \|(T(U_1,\ldots,U_k)-P)^{m\binom{n}{2}}\| && (3.17)\\
&= \|T^{m\binom{n}{2}}(U_1,\ldots,U_k)-P\| = \left\|\sum_{w\in W_{m\binom{n}{2}}}\frac{\tilde{w}-P}{(2l)^{m\binom{n}{2}}}\right\| && (3.18)\\
&\leq \frac{1}{(2l)^{m\binom{n}{2}}}\left(\sum_{w\notin R_m}\|\tilde{w}-P\| + \left\|\sum_{w\in R_m}(\tilde{w}-P)\right\|\right) && (3.19)\\
&\leq \frac{(2l)^{m\binom{n}{2}} - |R_m| + \left\|\sum_{w\in R_m}(\tilde{w}-P)\right\|}{(2l)^{m\binom{n}{2}}} && (3.20)\\
&= 1-\frac{|R_m|}{(2l)^{m\binom{n}{2}}}(1-\Lambda(R_m)) = 1-\frac{1-\binom{n}{2}d^m}{(2l)^{m\binom{n}{2}}} && (3.21)
\end{aligned}
$$

If we choose m large enough so that $\binom{n}{2}d^m < 1$, then this will be less than one, and we will have $\Lambda(U_1,\ldots,U_k) < 1$. ■

3.3 Almost all gates are efficiently universal

It should be fairly clear that if the mixing operator for a set of gates has all but one eigenvalue bounded by a constant less than one, then powers of it will converge exponentially to a uniform distribution. This intuition will be formalized in a moment, though first we need to show that this property holds for most choices of base gates.

Lemma 15 *Suppose there exists $U_1,\ldots,U_k \in SU(N)$ such that $\Lambda(U_1,\ldots,U_k) = d < 1$. Then for any $A_1,\ldots,A_l \in SU(N)$ such that $\langle A_1,\ldots,A_l\rangle$ is dense in $SU(N)$, $\Lambda(A_1,\ldots,A_l) < 1$.*

Proof Since $\langle A_1,\ldots,A_l\rangle$ is dense in $SU(N)$, we have $\forall\, M_1,\ldots,M_k \in SU(N), \forall\epsilon > 0$, $\exists n \in \mathbb{Z}, q_1,\ldots,q_k \in W_n(A_1,\ldots,A_l)$ such that $|q_i - M_i| < \epsilon$ for $1 \leq i \leq k$. Since Λ is continuous, there exists some other value of n, with corresponding $q_1,\ldots,q_k \in W_n(A_1,\ldots,A_l)$ such that

$$|\Lambda(q_1,\ldots,q_k) - \Lambda(M_1,\ldots,M_k)| < \epsilon \tag{3.22}$$

Choose $U_1,\ldots,U_k$ as in (**??**), let $\epsilon = (1-\Lambda(U_1,\ldots,U_k))/2$ and obtain n and $q_1,\ldots,q_k$ so that (**??**) holds. If necessary, reduce ϵ to ensure that ϵ is less than half the distance between any pair of U_i or U_i^{-1}, so $q_1,\ldots,q_k$ are necessarily distinct. Now we apply a similar technique to the one used in the last proposition.

$$
\begin{aligned}
(\Lambda(A_1,\ldots,A_l))^n &= \|T-P\|^n = \|(T-P)^n\| = \|T^n - P\| && (3.23)\\
&= \left\|\sum_{w\in W_n}\frac{\tilde{w}-P}{(2l)^n}\right\| \leq \frac{(2l)^n - 2k + \left\|\sum_{i=1}^k (q_i + q_i^{-1}) - 2kP\right\|}{(2l)^n} && (3.24)\\
&= 1 + \frac{2k}{(2l)^n}(\Lambda(q_1,\ldots,q_k)-1) \leq 1 + \frac{2k}{(2l)^n}(\Lambda(U_1,\ldots,U_k)+\epsilon-1) && (3.25)\\
&= 1 - \frac{2k}{(2l)^n}\epsilon < 1 && (3.26)
\end{aligned}
$$

■

In their paper referenced earlier, Lubotsky, Phillips and Sarnak raised the question of whether the spectrum of $T - P$ had one as an accumulation point for almost all randomly chosen $A_1, \ldots, A_l$. This settles the question in the negative. Now we are ready to prove our main result.

Theorem 16 *For any $A_1, \ldots, A_l \in SU(N)$ such that $\langle A_1, \ldots, A_l \rangle$ is dense in $SU(N)$, $\exists c$ such that $\forall U \in SU(N), \epsilon > 0, \forall n > c \log 1/\epsilon, \exists w \in W_n$ such that $|w - U| < \epsilon$.*

Proof Define $T = T(A_1, \ldots, A_l)$ as above and consider $(T^n - P)\tilde{U}^{-1}$. Since U is unitary, $\tilde{U}^{-1}$ is unitary as well, and

$$\|(T^n - P)\tilde{U}^{-1}\| = \|T^n - P\| = \|(T - P)^n\| = \|T - P\|^n = \Lambda(A_1, \ldots, A_l)^n < \alpha^n \tag{3.27}$$

for some $\alpha(A_1, \ldots, A_l) < 1$.

Define $\chi \in L^2(SU(N))$ by

$$\chi(x) = \begin{cases} 1 & \text{for } |x - I| < \epsilon/2 \\ 0 & \text{otherwise} \end{cases} \tag{3.28}$$

Let $V = \|P\chi\| = \|\chi\|^2$ be the measure of the ball around the identity of radius $\epsilon/2$. We won't perform this integration, but it suffices to note that there exists a constant k such that $V > k\epsilon^{N^2-1}$, since $SU(N)$ is an $(N^2 - 1)$-dimensional manifold.

Now we use the Cauchy-Schwartz inequality to give

$$\left|\left\langle \chi, (T - P)^n \tilde{U}^{-1} \chi \right\rangle\right| \leq \|\chi\| \|(T - P)^n \tilde{U}^{-1} \chi\| \leq \|\chi\|^2 \|(T - P)^n \tilde{U}^{-1}\| < \alpha^n V \tag{3.29}$$

Another way to compute the same inner product is

$$\left\langle \chi, (T^n - P)\tilde{U}^{-1} \chi \right\rangle = \left\langle \chi, T^n \tilde{U}^{-1} \chi \right\rangle - \left\langle \chi, P\tilde{U}^{-1} \chi \right\rangle = \left\langle \chi, T^n \tilde{U}^{-1} \chi \right\rangle - V^2 \tag{3.30}$$

Combining these gives that $\left|\langle \chi, T^n \tilde{U}^{-1} \chi \rangle - V^2\right| < \alpha^n V$. This means that there exists c which depends only on $A_1, \ldots A_l$ such that if $n > c \log 1/\epsilon$ then $\langle \chi, T^n \tilde{U}^{-1} \chi \rangle > 0$. Specifically, we must choose

$$n > \frac{N^2 - 1}{\log(1/\alpha)} \log(1/\epsilon) + \log k \tag{3.31}$$

When this occurs it means that

$$\int \chi(g) \sum_{w \in W_n} \frac{\chi(wU^{-1}g)}{(2l)^n} dg > 0, \tag{3.32}$$

which implies that $\exists x \in SU(N)$ and $w \in W_n$ such that $\chi(x) \neq 0$ and $\chi(wU^{-1}x) \neq 0$. Thus $|x - I| < \epsilon/2$ and $|wU^{-1}x - I| < \epsilon/2$, implying that $|w - x^{-1}U| < \epsilon/2$. Combining these and using the triangle inequality gives $|w - U| < \epsilon$. ■

3.4 Optimality of the result

This proves the theorem, but can we do any better? Note that an ϵ-ball in $SU(N)$ has measure of order ϵ^{N^2-1}, so if we expect to cover all of $SU(N)$ with strings of length n, then we will require $(2l)^n k \epsilon^{N^2-1} > 1$, or equivalently,

$$n = \frac{N^2 - 1}{\log 2l} \log 1/\epsilon + k \tag{3.33}$$

Thus, up to the constant α, the result is optimal. Eliminating the constant linear factor is unlikely, but it would be nice to find a value of α that did not depend on $A_1, \ldots, A_l$.

It turns out that this is also impossible. Recall from the introduction that the sets of base gates that do not generate dense subgroups of $SU(N)$ have measure zero in $SU(N)^l$. Now consider the base gates that are dense but are very close to base gates that are not dense. More formally, define

$$B(\delta) = \{(A_1, \ldots, A_l) \quad | \quad \langle A_1, \ldots, A_l \rangle \text{ is dense, } \exists A'_1, \ldots, A'_l, \\ \langle A'_1, \ldots, A'_l \rangle \text{ is not dense and } |A_i - A'_i| < \delta\} \tag{3.34}$$

Since non-dense base gates exist, but have measure zero, $B(\delta)$ has non-zero measure for all $\delta > 0$. Now, from theorem **??** it follows that for $(A_1, \ldots, A_l) \in B(\delta)$, everything in $W_n(A_1, \ldots, A_l)$ lies within $n\delta$ of some non-dense subgroup. An example might be a one-parameter subgroup. Since we can make δ arbitrarily small, any value of α that does not depend on the base gates will fail on a set with non-zero measure.

Note that unlike most results about quantum compiling, this argument also holds if the base gates are parametrized; say, $A_1, \ldots, A_l$ are elements of the algebra $su(N)$ and a single operation now has the form $e^{\pm A_i t}$, for any $t > 0$. Clearly the above proof demonstrates that there exist sets with non-zero measure which requires arbitrarily many steps, even if they are continuous. If we measure cost not in terms of number of steps, but by the total time taken, then we have to modify the argument slightly. For small values of t, $|e^{A_i t} - e^{A'_i t}|$ is on the order of $t\delta$, but for large t the difference never gets any higher than δ. This means that no matter how many steps we take, in time t, we will stay within $t\delta$ of some non-dense subgroup and the same result holds.

Most of the above lower bounds could be proved more easily by simply taking the $A_1, \ldots, A_l$ to be infinitesimally small. However, the more general problem of operators that very nearly commute or very nearly generate finite subgroups is important to consider.

Chapter 4

A class of gates that allows optimal compiling

In many ways, the results of the previous two chapters represent the limits of their respective approaches. At its heart, the Solovay-Kitaev theorem relies on successive approximation to compile using any universal set of gates. It might seem as though we could find a better method of successive approximation, since using the commutator technique to generate finer and finer epsilon-nets multiplies the string length by four each time and only uses a vanishingly small fraction of the total number of possible strings. However, suppose that instead of repeatedly quadrupling our string length, we had available a nonunitary superoperator σ that would shrink a single unitary matrix halfway towards the identity. Then applying σ^n would turn a ϵ-net for $S(2\epsilon)$ into a $2^{-n}\epsilon$-net for $S(2^{-(n-1)}\epsilon)$. This obviously extends into a method of compiling that requires $O(\log^2(1/\epsilon))$ gates, which is asymptotically no better than Kitaev's version of the algorithm. This example is rather artificial, but it illustrates the problems of any successive approximation scheme. First, by the arguments of the last chapter, we know that ϵ-nets generally require at least $\log 1/\epsilon$ gates. Second, in order for the compiling technique to be polynomial time computable, we need the size of the region covered by the ϵ-net to stay roughly proportional to ϵ, or at worst, grow relative to ϵ as $\log^c(1/\epsilon)$. This means that we need to progress through roughly $\log 1/\epsilon$ different epsilon-nets, which range in number of gates per point from $O(1)$ to $O(\log 1/\epsilon)$, resulting in a cost quadratic in $\log 1/\epsilon$. Thus, successive approximation is unlikely to offer us any asymptotic improvements at this point, although it is quite difficult to prove that no new creative method could ever perform asymptotically better in polynomial time.

The generality of the Solovay-Kitaev algorithm also makes it difficult to improve. Recall that after the initial ϵ_0-net is created, the base gates are abstracted away into the ϵ_0-APPROX procedure. Although this cannot be a very strict black box, since its contents can be completely determined in constant time, treating ϵ_0-APPROX as a black box means only needing to know about the starting gates to $O(\epsilon_0)$, which also suggests that this method will have trouble achieving optimal results.

The results of the third chaper suffer from the opposite flaws. Though asymptotically optimal, they offer no guidance in coming up with any sort of constructive method of achieving the $O(\log 1/\epsilon)$ upper bound. In fact, since there is no practical way of evaluating $\|T - P\|$, we do not even have a constructive way of determining what prefactor is necessary for any particular set of gates.

Both are unhelpful in our goal of efficiently saturating the lower bound in large part because they have to work with arbitrary base gate sets, which we have no way of putting stronger conditions on other than that they generate dense subsets, or comprise ϵ-nets. If we are to make progress, then it may be necessary to specialize our arguments to a particular set of base gates. In this chapter, the gates we will consider will be $SU(2)$ matrices corresponding to rotations by $\arccos(\pm\frac{3}{5})$ about the X, Y, and Z axes.

4.1 Quaternions

To understand the motivation for this choice of gates, we will first need to develop some basic number theoretic concepts. Most of this section is taken from [LPS87].

Let $H[\mathbb{Z}]$ denote the ring of quaternions with integer entries, i.e.

$$H[\mathbb{Z}] = \left\{ a + bi\sigma_x + ci\sigma_y + di\sigma_z \,\middle|\, a, b, c, d \in \mathbb{Z} \right\} \tag{4.1}$$

For $\alpha \in H[\mathbb{Z}]$, define $\bar{\alpha} = \alpha^\dagger = a - bi\sigma_x - ci\sigma_y - di\sigma_z$ to be its conjugate and $N(\alpha) = \alpha\bar{\alpha} = a^2 + b^2 + c^2 + d^2 \in \mathbb{Z}$ to be its norm.

There is a natural homomorphism φ from the non-zero quaternions into $SU(2)$. It is given by

$$\varphi(\alpha) = \frac{\alpha}{\sqrt{N(\alpha)}} \tag{4.2}$$

From this it is easy to see that the units in the quaternion ring are the eight elements with norm one, namely $\pm 1, \pm i\sigma_x, \pm i\sigma_y, \pm i\sigma_z$. A well-known result from number theory (see [Har]) holds that the number of ways to express a positive number n as the sum of four squares is

$$r_4(n) = 8 \sum_{\substack{d|n \\ 4 \not| d}} d \tag{4.3}$$

Equivalently, $r_4(n)$ is the number of $\alpha \in H[\mathbb{Z}]$ with $N(\alpha) = n$.

For p a prime, with $p \equiv 1 \pmod 4$, we will consider the set of $\alpha \in H[\mathbb{Z}]$ such that $N(\alpha) = p$. Since squares are congruent to either 0 or 1 mod 4, only one of a, b, c, d can be odd. This means that every α' with $N(\alpha') = p$ has a unique associate $\alpha = \epsilon\alpha'$, for some unit ϵ such that $N(\alpha) = p$, $\alpha \equiv 1 \pmod 2$ and $a > 0$. By (**??**) there are $p+1$ such α and we can divide them into $\sigma = (p+1)/2$ conjugate pairs which are denoted

$$\{\alpha_1, \bar{\alpha}_1, \alpha_2, \bar{\alpha}_2, \ldots, \alpha_\sigma, \bar{\alpha}_\sigma\} = S_p \tag{4.4}$$

The example that we will typically work with is

$$S_5 = \{1 + 2i\sigma_x, 1 - 2i\sigma_x, 1 + 2i\sigma_y, 1 - 2i\sigma_y, 1 + 2i\sigma_z, 1 - 2i\sigma_z\}, \tag{4.5}$$

which φ maps to the $V_1, V_2, V_3, V_1^\dagger, V_2^\dagger, V_3^\dagger$ of (**??**).

Let $R_m(\alpha_1, \bar{\alpha}_1, \ldots, \alpha_\sigma, \bar{\alpha}_\sigma)$ denote a reduced word of length m, which, like the reduced words we introduced in chapter 1, is defined to mean a word in $\alpha_1, \ldots, \bar{\alpha}_m$ that contains no subwords of the form $\alpha_i\bar{\alpha}_i$ or $\bar{\alpha}_i\alpha_i$. The number of such words of length l is $(p+1)p^l$.

Quaternions are interesting because they have properties both of $SU(2)$ and of the integers. The one that we will find most useful is unique factorization.

Proposition 17 *Every $\beta \in H[\mathbb{Z}]$ with $N(\beta) = p^k$ has a unique representation*

$$\beta = p^l \epsilon R_m(\alpha_1, \ldots, \bar{\alpha}_\sigma) \tag{4.6}$$

where $l \leq \frac{1}{2}k, m + 2l = k$, and R_m is a reduced word of length m in $\alpha_1, \ldots, \alpha_\sigma$.

Proof First we prove that such a factorization exists. It can be shown (see [Dic]) that quaternions with odd norm form a left and right Euclidean ring. Moreover, $\alpha \in H[\mathbb{Z}]$ is prime if and only if $N(\alpha)$ is a prime integer. Since $N(\beta) = p^k \equiv 1 \pmod 2$, we can write $\beta = \gamma\delta$, where $N(\gamma) = p^{k-1}$ and $N(\delta) = p$. We can in turn write $\delta = \epsilon\alpha$ for some unit ϵ and some $\alpha \in S_p$, yielding $\beta = \gamma\epsilon\alpha$.

Proceeding by induction we obtain $\beta = \epsilon s_1 s_2 \cdots s_k$ with $s_j \in S_p$. Finally, we cancel where appropriate to give the desired factorization.

To prove uniqueness, we count the number of factorizations in two ways. Summing over all possible values of l gives

$$8\left(\sum_{0\leq l<k/2}(p+1)p^{k-2l-1}+\delta(k)\right) \tag{4.7}$$

where $\delta(k) = 1$ if k is even and $\delta(k) = 0$ if k is odd. This expression simplifies to $8(p^{k+1}-1)/(p-1)$. But from (**??**) we have that the number of quaternions with norm p^k is $8(1+p+p^2+\cdots+p^k = 8(p^{k+1}-1)/(p-1)$, so there is a one-to-one correspondence between quaternions of p^k and factorizations in the form described. ■

One immediate consequence of this lemma is that no reduced word with non-zero length can equal a multiple of 1. Thus the V_1, V_2, V_3 of (**??**) generate a free group in $SU(2)$, and from proposition **??**, almost all sets of elements of $SU(N)$ generate free groups.

4.2 A Factoring Algorithm

An immediate corollary of unique factorization existing for quaternions of norm p^k is that we can find this factorization in time $O(pk)$. The algorithm is as follows. Given $\beta \in H[\mathbb{Z}]$ such that $N(\beta) = p^k$, successively try dividing β (on the right) by each element of S_p. Recall that β can be expressed as $\beta = \gamma\epsilon\alpha$, for ϵ a unit, $\alpha \in S_p$, and $N(\gamma) = p^{k-1}$. Clearly, dividing on the right by α will still leave an integer quaternion. Conversely, if dividing β on the right by any other $\alpha' \in S_p$ resulted in an integer quaternion (or equivalently, in no remainder), then we could write $\beta = \gamma'\epsilon'\alpha'$, for ϵ' a unit and $N(\gamma') = p^{k-1}$. Factoring γ and γ' would now give two factorizations of β which differed in the last position (since $\alpha \neq \alpha'$), which contradicts the proposition that β had a unique factorization. Thus every time we test every element of S_p we are guaranteed to correctly find exactly one divisor of β. Then we can divide and continue until $N(\beta) = p$.

This would seem to lend itself well to an efficient approximation scheme for elements of $SU(2)$. The missing step is the following:

> Procedure $[\beta, k]$=`QUAT-APPROX(`U`, `ϵ`)`
>
> Given $U \in SU(2)$ and $\epsilon > 0$.
> Find $\beta \in H[\mathbb{Z}]$ and $k \in \mathbb{Z}$.
> Such that:
>
> - $k = O(\log 1/\epsilon)$
> - $|U - \beta/5^k| < \epsilon$
> - $N(\beta) = 5^{2k}$

Clearly, if `QUAT-APPROX` were possible to implement in polynomial (in $\log 1/\epsilon$) time, then efficient $O(\log 1/\epsilon)$ factoring would be possible. Unfortunately there is no known method for doing so, or even of performing substantially better than a blind search of integer quaternions in the neighborhood of $5^k U$ (for some reasonably large k). To evaluate the cost of this strategy, note that the surface of a 3-sphere with radius 5^k is three-dimensional and hence has surface volume $O(5^{3k})$. However, from (**??**), the number of integer quaternions on the surface of this 3-sphere is $O(5^{2k})$, so we can expect to have to search a volume of $O(5^k)$. From $|U - \beta/5^k| < \epsilon$, we have that k must generically be at least $O(\log 1/\epsilon)$, so blind search has a cost that is exponential in $O(\log 1/\epsilon)$.

At the heart of `QUAT-APPROX` is the problem of finding integer lattice points on a large and fairly loosely defined region of the surface of a 3-sphere, which has connections to a several active areas of

research in mathematics. Number theory, for example, has been applied to results such as Malyshev's (described below), of which unfortunately almost all are non-constructive and asymptotic. Finding integer solutions to systems of polynomial equations is often dealt with in algebraic geometry, but rarely with the same notions of distance that are vital here. It would seem that new mathematical tools will be necessary to solve this problem.

Even if we have no idea of how to efficiently find solutions to `QUAT-APPROX`, we at least know that such solutions exist.

Proposition 18 *Procedure* `QUAT-APPROX` *has valid solutions for any* U *and* ϵ.

Proof For any region Γ of a 3-sphere of radius $R = 5^k$ delimited by a finite number of half-planes (such as most of the region within $R\epsilon$ of RU), let a be the fraction of integer points on the surface of the 3-sphere to fall within Γ, and b be the fraction of surface volume of the 3-sphere that belongs to Γ. A theorem due to Malyshev (see [Lin] or [Mal62]) holds that the difference between a and b grows more slowly than any polynomial in R. If $k \geq c(\log 1/\epsilon)/(\log 5)$, then $R = 5^k \geq \epsilon^{-c}$ for any constant c that we wish. The region of the 3-sphere where we can find valid β is the intersection of its surface with the ball of radius $R\epsilon$ around RU, which has volume $O((R\epsilon)^3) = O(\epsilon^{3(1-c)})$. From Malyshev's theorem and the scaling arguments of the last paragraph, we have that the density of points is at least $O(R^{-1-\delta}) = O(\epsilon^{c(1+\delta)})$ for any $\delta > 0$ that we please. Thus, it suffices to choose c such that

$$O(1) < O(\epsilon^{3(1-c)}\epsilon^{c(1+\delta)}) = O(\epsilon^{3-c(2-\delta)}) \tag{4.8}$$

Thus choosing $c > 2$ will be asymptotically sufficient. ■

This gives an alternate proof of theorem **??**, though for a specific set of gates instead for a set of measure one.

4.3 Implementation Concerns

Although most fault-tolerant schemes do not directly permit constructing rotations by arccos 3/5 about the axes, there is a fairly simple method due to Preskill of implementing them from S, H and Toffoli gates together with measurment and and classical feedback ([NC00]). The construction is as follows. Initialize a two qubit to be $|00\rangle$. Apply a hadamard transformation (H) to each qubit to obtain $(|00\rangle + |01\rangle + |10\rangle + |11\rangle)/2$. Perform a Toffoli gate with a third qubit $|\psi\rangle$ as the target, resulting in

$$(|00\rangle|\psi\rangle + |01\rangle|\psi\rangle + |10\rangle|\psi\rangle + |11\rangle\sigma_x|\psi\rangle)/2 \tag{4.9}$$

Now apply an S gate to $|\psi\rangle$, apply another Toffoli gate from the first two qubits upon the third, apply hadamard gates to the first two qubits and measure them in the computational basis. Working through the algebra shows that if the value 00 is measured, then $|\psi\rangle$ will have been rotated about the Z axis by arccos 3/5, as desired. This has a 5/8 probability of occuring. If it does not, then a σ_z gate will have been applied to $|\psi\rangle$, which we will need to undo before attempting the arccos 3/5 rotation again. The expected number of times that we need to repeat this process is 8/5.

The constructions for rotations by arccos 3/5 are similar. Thus with most fault-tolerant methods of computing, we can implement the gates in S_5 exactly in expected constant time.

Chapter 5

Conclusions and further directions

We have demonstrated a constructive proof of the Solovay-Kitaev theorem, tightened the upper and lower bounds for compiling to within a constant factor of each other and demonstrated a class of gates that shows promise for efficiently compiling to within a constant factor of optimal. In many ways, these sorts of analyses are only the start of a method of looking at $SU(N)$ that can have broad applications.

At the end of chapter three, we concluded that almost every family of gates covered $SU(N)$ in a way that was asymptotically exponential, but that always had a prefactor α. Finding general techniques of determining α seems to be a difficult but quite interesting problem.

This is a largely unanswered question and to best of our knowledge, the only calculations that have been made are on the two extremes: the base gates in lemma **??** are provably optimal for number-theoretic reasons, and $B(\delta)$ (from (**??**)) captures what are probably the worst possible base gates, for reasons related to the subgroup structure of $SU(N)$. This question also has practical significance. Many physical implementations of quantum computers exist in low-coupling limits, where many-qubit operations are much slower than single-qubit operations because the available Hamiltonians are very close to completely separable Hamiltonians.

For example, in an two-spin NMR system, a good approximation for the Hamiltonian is

$$H = \omega_1 \sigma_z \otimes I + \omega_2 I \otimes \sigma_z + J \sigma_z \otimes \sigma_z + \mathrm{RF}_1 \otimes I + I \otimes \mathrm{RF}_2, \tag{5.1}$$

where ω_1 and ω_2 are hundreds of megahertz, J is somewhere from 1 to 100 Hz, and RF_1 and RF_2 are arbitrary elements of $su(2)$ that are on the order of ω_1 and ω_2 respectively. This Hamiltonian clearly allows us to achieve any transformation in $SU(4)$, but in practice the speed at which we can do so is limited by J. This can be represented formally by saying that all of the basic unitary transformations that are available to us lie within some small neighborhood of the subgroup $SU(2) \times SU(2)$.

If this sort of analysis could be formalized and generalized, then it could have quite useful conclusions for our ability to effect quantum control in general. In the future, when deceptive quantum computer manufacturers play up their single-qubit Megahertz, it might be nice to be able to derive a value of α that determines bounds on their speed in practice.

Compiling in $SU(N)$ is also useful for more than strictly simulating algorithms with a fixed set of gates. A common problem in quantum computing is that "qubits" are not actually two-level systems, but have several other accessible energy states that our perturbations will sometimes send them into. Rather then treating this with the usual techniques of error-correction, Tian and Lloyd noted that this leakage into other dimensions simply calls for compiling single-qubit operations in $SU(N)$ to take into account the effects of transitions in and out of the two energy levels being used for computation. This is a difficult problem, but one that it will be increasinly necessary to solve in the future.

Finally, this work also poses some interesting open mathematical questions. There have been remarkably few studies of decomposing matrices into approximate products of a fixed set of matrices, and any algorithm that generically compiled gates to strings of length $O(\log 1/\epsilon)$ would likely have

to come up with some new mathematical techniques to do so. The problem of implementing the procedure `QUAT-APPROX` in polynomial time is also deeper than it might seem at first and is surprisingly challenging.

Optimizing quantum compilers is a problem that in many ways parallels the entire field of quantum computation. Although it is uncertain whether we will actually reach our goal, or what the implications will be once we get there, all we can know is that along the way we will develop a powerful set of broadly useful tools, and encounter a fascinating array of difficult problems.

Bibliography

[AG] N. I. Akhiezer and I. M. Glazman *Theory of Linear Operators in Hilbert Space, vol. I* Pitman Advanced Publishing Program. 1981

[AL97] D. Abrams and S. Lloyd "Simulation of Many-Body Fermi Systems on a Universal Quantum Computer" quant-ph/9703054, Phys.Rev.Lett. **79** 2586-2589 (1997)

[Art] M. Artin *Algebra* Prentice Hall. 1991

[Axl] S. Axler. *Linear Algebra Done Right*. Springer-Verlag. 1996

[Bar95] A. Barenco *et al*. Elementary gates for quantum computation. Mar 23, 1995. quant-ph/9503016

[Ben89] C. H. Bennett. Time-space trade-offs for reversible computation. *SIAM J. Comput.*, 18:766-776, 1989.

[DBE95] D. Deutsch, A. Barenco, and A. Ekert. "Universality in Quantum Computation," Proc. R. Soc. of Lond. A, quant-ph/9505018 1995.

[Dic22] L.E. Dickson, "Arithmetic of Quaternions," Proc. London Math. Soc. **20(2)** 225-232, 1922

[DS87] P. Diaconis, M. Shahsahani, "The subgroup algorithm for generating uniform random variables." Prob. Eng. Info. Sci. **1** 15-32, 1987.

[Fey82] R. P. Feynman. "Simulating physics with computers." Int. J. Theor. Phys. **21** 467, 1982

[Fey86] R. P. Feynman. "Quantum Mechanical Computers." *Foundations of Physics*, 16(6):507-531, 1986

[Gas] S. Gasiorowicz. *Quantum physics*. John Wiley and Sons. 1996

[Grif] D. Griffiths. *Introduction to Quantum Mechanics*. Prentice Hall. 1995

[Hal] P. Halmos. *A Hilbert Space Problem Book*. Springer-Verlag. 1982.

[HJ] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge Univ. Press 1985

[Kit] A. Y. Kitaev, "Quantum Computation: Algorithms and Error Correction," Russ. Math. Surv. **52**:1991 (1997)

[Lin] Y. V. Linnik. *Ergodic Properties of Algebraic Fields* trans M. S. Keane. Springer-Verlag 1968

[Llo95] S. Lloyd. "Almost any quantumm logic gate is universal." *Phys. Rev. Lett.*, 75(2):346, 1995

[LPS86] A. Lubotsky, R. Phillips and P. Sarnak. "Hecke operators and distributing points on the sphere I" I. Comm. Pure Appl. Math. **39**, Supplement I, S149-S186, 1986

[LPS87] A. Lubotsky, R. Phillips and P. Sarnak. "Hecke operators and distributing points on the sphere II" I. Comm. Pure Appl. Math. **40**, 401-420, 1987

[Mal62] A. V. Malyshev, "On representations of integers by positive quadratic forms," Trudy Math. Inst. Akad. Nauk. **45** (1962)

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press. 2000

[Sip] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company. 1997

[TL00] L. Tian and S. Lloyd "Resonant cancellation of off-resonant effects in a multilevel qubit" quant-ph/0003034 (2000)

[Vaz98] U. Vazirani, "On the Power of Quantum Computation." Phil. Trans. R. Soc. Lond. A **356**, 1759-1768 (1998)

Appendix A

Source code

Although it is traditional to include all the source code used in a thesis, it would not be practical to list here the more than 5000 lines of functional code used for the project. Instead, only three files are included. The first, `su2.h`, contains the definitions, but not the full source code, for the general-purpose functions dealing with $SU(2)$ that are used in the other two files. The second, `sk.cc` implements the Solovay-Kitaev algorithm, and calls an external program `th_net` to create the initial epsilon net simply by using a few heuristics to print all distinct strings of T and H gates up to a particular length. Finally `epsilon.cc` is the program to determine the value of ϵ for a given ϵ-net. It is the most computationally intensive, and as a result of its optimizations, by far the longest. The code listed only implements the client side, however the operation of the server is fairly straightforward, and can easily be implied by the behavior of the client.

A.1 Common functions

```
/*
  SU2 utility stuff

  Aram Harrow, Jan-May 2001
*/

#ifndef __SU2_H
#define __SU2_H

#define PI 3.141592653589793238462643
#define MAX_DIST 2.0
#define MAX_DIST_SQR 4.0
#define del(x) do {if (x) delete x; x=NULL;} while(0)

inline double sqr(double x) {return x*x;}

template <class T> inline int sgn(T x)
{return (x>0) ? 1 : ((x<0) ? -1 : 0);}

template <class T> T MAX (T a,T b)
{return (a>b)?a:b;};

template <class T> T MIN (T a,T b)
{return (a<b)?a:b;};
```

```
template <class T> T RESTRICT(T x, T lb, T ub)
{return (x<lb) ? lb : (x>ub ? ub : x);}

/* return a random number uniformly distributed between 0 and 1 */
double uniform();

/* determine the L2 norm of a length-n vector */
double vec_norm(double *v, int n);

/* print a vector in matlab format */
void print_vec(char *name, double *vec, int n);

/* make a 4-vector into a unit vector */
double normalize(double mat[4]);

/* make a orthogonal to b, where b is a unit vector */
void subtract_off(double a[4], double b[4]);

/* given a time in seconds return a string of the form "4h 22m 37s" */
char *time_str(int time);

/*
   Two formats are used to represent elements of SU(2), usually called
   "mat" (for matrix) and "param" (for parameterized).
   mat format means four numbers (a,b,c,d) such that
   U = a + biZ + ciX + diY
   where X = [0 1]    Y = [0 -i]    Z = [1  0]
             [1 0]        [i  0]        [0 -1]

   param format is (x,y,z) s.t. for random points on the 3-sphere they
   are independant and uniformly distributed from 0 to 1
   The relation is
   a = sqrt(x) * cos(2 * pi * y)
   b = sqrt(x) * sin(2 * pi * y)
   c = sqrt(1-x) * cos(2 * pi * z)
   d = sqrt(1-x) * sin(2 * pi * z)

   Distance is Euclidean.
*/
void param_to_mat(double *param, double *mat);
void mat_to_param(double *mat, double *param);
double param_dist(double *p1, double *p2);
double mat_dist_sqr(double *m1, double *m2);
double mat_dist(double *m1, double *m2);

/* computes C = A*B, where the matrices are in mat format */
void matrix_mult(double *a, double *b, double *c);

/* find A and B s.t. [A,B] is close to U and |A-I|, |B-I| are small */
void factor(double *U, double *a, double *b);

/* for U in mat format, this computes the v in the Lie algebra such
   that e^v = U.  v is represented as a 3-vector. */
void mat_to_vec(double *U, double *v);

/* given v in the Lie algebra, computes U = e^v in mat format */
```

```
void vec_to_mat(double *v, double *U);

/* converts from cartesian to spherical coordinates */
void cart_to_sph(double *cart, double *sph);

/* converts from spherical to cartesian coordinates */
void sph_to_cart(double *sph, double *cart);

/* set a matrix to the identity */
void set_identity(double *U);

/*  Generates a random matrix in mat format. */
void rand_mat(double *mat);

/*  Generates a random matrix in param format. */
void rand_param(double *param);

/*
 * str is a string made up of 'XYZTH'.
 * Each character represents a primitive matrix (lowercase means inverse).
 * U is set to the product of all the primitive matrices.
 */
void eval_str(char *str, double *U);

/* turn a matrix into its inverse */
void invert(double *U);

/* reflect a matrix onto the upper half-sphere */
void reflect(double *U);

/*
 * Invert a string of matrices.
 * This is equivalent to reversing the order and exchanging uppercase
 * and lowercase.
 */
void invert_str(char *str);

/* maximum distance from the center of a patch to any corner */
double param_patch_radius(double low[], double high[]);

/*
 * returns min_{x,y,z in [low, high]} |(x,y,z)-pt|
 * where everything is in param format.
 * This is used to reject points that are far from a patch before
 * performing the O(m^5) check.
 *
 * Recall that the distance between x1,y1,z1 and x2,y2,z2 is
 * sqrt(2 * (1 - sqrt(x1*x2) * cos(2*pi*(y1-y2)) -
 *               sqrt((1-x1)*(1-x2) * cos(2*pi*(z1-z2)))))
 *
 */
double min_dist_to_patch(double pt[], double low[], double high[]);

/* whether to ignore sign when computing distance */
extern bool so3_dist;
```

```
#endif /* __SU2_H */
```

A.2 Solovay-Kitaev implementation

```
/*
   Solovay-Kitaev implementation

   Aram Harrow Jan 2001
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <unistd.h>
#include <vector>

#include "su2.h"

int dist_count = 0;
bool laconic = true;

double *dbl_index_comp_array;
int dbl_index_comp_func(const void *a, const void *b)
{
  double d=dbl_index_comp_array[*(int*)a]-dbl_index_comp_array[*(int*)b];
  return d<0 ? -1 : d>0 ? 1 : 0;
}

/* this class represents our initial epsilon net */
class fixed_net
{
public:
  char *approx(double *U);
  void create(int hash_size);
  void save(char *fn);
  bool load(char *fn);
  fixed_net() {n=0; strings=NULL; string_data=NULL; matrices=NULL;}
  ~fixed_net()
  {
    n=0;
    if (strings) delete strings;
    if (string_data) delete string_data;
    if (matrices) delete matrices;
  }

private:
  char **strings;
  char *string_data;
  int str_data_len;
  double *matrices;
  double *dist_to_id;
  int n;
  double epsilon;
} global_net;
```

```
void fixed_net::create(int hash_size)
{
  char fn[80], cmd[200], buf[100];
  double m[4];
  int i, ptr, len, *order;
  FILE *pipe;
  double eye[4], *_dist_to_id;
  vector<double> _matrices;
  vector<int> _strings;
  vector<char> _string_data;

  /* check for already saved net */
  sprintf(fn, "precomputed-net-%d.txt", hash_size);
  if (load(fn)) return;

  /* create covering with hash_size^3 patches */
  sprintf(cmd, "./covering %g 1 1 1", 1.0 / hash_size);
  if (!laconic) printf("%s\n", cmd);
  pipe = popen(cmd, "r");
  assert(pipe);
  ptr = 0;
  while (!feof(pipe))
    {
      fscanf(pipe, "%lf%lf%lf%lf%d%*[ ]%[a-zA-Z]",
     &m[0], &m[1], &m[2], &m[3], &len, buf);
      if (!len) buf[0] = 0;
      assert(len == (int)strlen(buf));
      _matrices.push_back(m[0]);
      _matrices.push_back(m[1]);
      _matrices.push_back(m[2]);
      _matrices.push_back(m[3]);
      _strings.push_back(ptr);
      ptr += len + 1;
      for (i=0 ; i<len ; i++)
_string_data.push_back(buf[i]);
      _string_data.push_back(0);
    }
  fclose(pipe);
  n = _matrices.size() / 4;

  /* order by distance to the identity */
  assert(_dist_to_id = new double[n]);
  set_identity(eye);
  for (i=0 ; i<n ; i++)
    _dist_to_id[i] = mat_dist(eye, &_matrices[i*4]);

  dbl_index_comp_array = _dist_to_id;
  assert(order = new int[n]);
  for (i=0 ; i<n ; i++) order[i] = i;
  qsort(order, n, sizeof(int), dbl_index_comp_func);

  /* convert from vectors to arrays and reorder */
  assert(dist_to_id = new double[n]);
  assert(matrices = new double[4 * n]);
  assert(strings = new (char *)[n]);
  assert(string_data = new char[str_data_len = _string_data.size()]);
```

```
  memcpy(string_data, _string_data.begin(), str_data_len);
  for (i=0 ; i<n ; i++)
    {
      strings[i] = string_data + _strings[order[i]];
      dist_to_id[i] = _dist_to_id[order[i]];
      memcpy(&matrices[i<<2], &_matrices[order[i]<<2], 4 * sizeof(double));
    }
  delete order;
  delete _dist_to_id;

  /* sanity check */
  for (i=0 ; i<n ; i++)
  {
    double test[4];
    eval_str(strings[i], test);
    assert(fabs(double(mat_dist(eye, &matrices[i*4]) - dist_to_id[i]))
   < 1e-9);
    assert(double(mat_dist(test, &matrices[i*4])) < 1e-6);
    if (i) assert(dist_to_id[i-1] <= dist_to_id[i]);
  }

  /* determine epsilon */
  FILE *temp_file;
  char temp_fn[30] = "sk-to-epsilon-XXXXXX";
  double eps;
  close(mkstemp(temp_fn));
  temp_file = fopen(temp_fn, "wt");
  assert(temp_file);
  for (i=0 ; i<4*n ; i++)
    fprintf(temp_file, "%g%c", (double)matrices[i], ((i+1)&3) ? ' ' : '\n');
  fclose(temp_file);
  sprintf(cmd, "./epsilon -laconic -load %s", temp_fn);
  if (!laconic) printf("%s\n", cmd);
  pipe = popen(cmd, "r");
  assert(pipe);
  fscanf(pipe, "%lf", &eps);
  fclose(pipe);
  unlink(temp_fn);

  /* save for future use */
  save(fn);
}

/* return the best approximation to a matrix U */
char *fixed_net::approx(double *U)
{
  int i, closest=0;
  double d, best=4.0, eye[4], U_to_id;
  set_identity(eye);
  U_to_id = mat_dist(eye, U);
  for (i=0 ; i<n ; i++)
    {
      dist_count++;
      d=mat_dist(U, &matrices[i*4]);
      if (d < best)
{
```

```
  best = d;
  closest = i;
}
      else
if (dist_to_id[i] > U_to_id + best)
  break; /* it's only downhill from here */

    }
  return strings[closest];
}

void fixed_net::save(char *fn)
{
  FILE *out = fopen(fn, "wt");
  int i;
  fprintf(out, "%d %d\n", n, str_data_len);
  for (i=0 ; i<n ; i++)
    fprintf(out, "%s\n", strings[i]);
  fclose(out);
}

bool fixed_net::load(char *fn)
{
  FILE *in = fopen(fn, "r");
  double eye[4];
  char *ptr;
  int i;

  if (!in) return false;
  if (3 != fscanf(in, "%d%d\n", &n, &str_data_len)) return false;
  if (!laconic)
    printf("Reading %d strings from %s\n", n, fn);
  assert(matrices = new double[n * 4]);
  assert(dist_to_id = new double[n]);
  assert(strings = new (char *)[n]);
  assert(ptr = string_data = new char[str_data_len]);
  set_identity(eye);
  for (i=0 ; i<n ; i++)
    {
      assert(!feof(in));
      strings[i] = ptr;
      while (isalpha(*ptr++ = getc(in))) ;
      ptr[-1] = 0;
      //for (j=0 ; j<4 ; j++) fscanf(in, "%lf ", &matrices[i*4 + j]);
      eval_str(strings[i], &matrices[i*4]);
      dist_to_id[i] = mat_dist(&matrices[i*4], eye);
      /*printf("'%s' %g %g %g %g\n", strings[i], matrices[i*4],
matrices[i*4 + 1], matrices[i*4 + 2], matrices[i*4 + 3]);*/
    }
  fclose(in);
  assert(ptr == string_data + str_data_len);
  return true;
}

/* strings that can grow dynamically */
struct long_string
```

```
{
  char *str;
  int len, max;

  void append(char *s);
  void init(int M) {assert(str=new char[max=M]); len=0;}
};

/* round x up to the nearest power of 2 */
int roundup(int x)
{
  int n=0;
  while (x>>=1) n++;
  return (1 << (n+1));
}

void long_string::append(char *s)
{
  int extra = strlen(s)+1;

  if (len + extra >= max)
    {
      char *temp = new char[max = roundup(len+extra+1)];
      memcpy(temp, str, len);
      delete str;
      str = temp;
    }
  memcpy(str + len, s, extra);
  len += extra - 1;
}

/*
   sk(U, n) returns a eps(n)^2 approximation to U.
   The calling procedure has the responsibility for deleting the result.
*/
char *sk(double *U, int n)
{
  long_string result;
  char *initial, *a_str, *b_str;
  double V[4], W[4], A[4], B[4];

  if (!n)
    {
      initial = global_net.approx(U);
      result.init(roundup(strlen(initial)));
      result.append(initial);
      return result.str;
    }

  initial = sk0(U, n-1);
  eval_str(initial, W);
  invert(W);
  matrix_mult(U, W, V);
  reflect(V);
  factor(V, A, B);
  a_str = sk0(A, n-1);
```

```
  b_str = sk0(B, n-1);
  result.init(strlen(initial) + 2 * (strlen(a_str) + strlen(b_str)) + 10);
  result.append(a_str);
  result.append(b_str);
  invert_str(a_str);
  invert_str(b_str);
  result.append(a_str);
  result.append(b_str);
  result.append(initial);

  delete a_str;
  delete b_str;
  delete initial;

  return result.str;
}

void sk_results(double *U, int depth, bool raw)
{
  double dbl;
  char *temp;
  double V[4];
  dist_count = 0;
  temp = sk(U, depth);
  if (!raw)
    {
      eval_str(temp, V);
      dbl = mat_dist(U, V);
      printf("%d %d %g\n", strlen(temp), dist_count, dbl);
    }
  else
    printf("%s\n", temp);
  fflush(stdout);
  delete temp;
}

void print_help()
{
  printf("format is sk <options> \n");
  printf(" -size <n>: set the size of the starting epsilon-net (def: 40)\n");
  printf(" -rand <depth> <n>: evaluate n random points to a given depth\n");
  printf(" -raw: output full string instead of length, eps and # compares\n");
  printf(" -prolix: print stupid status messages\n");
  exit(1);
}

int main(int argc, char **argv)
{
  double U[4];
  int size = 40, n=1, depth = 2, i;
  bool rand = false, raw = false;

  so3_dist = true;

  for (i=1 ; i<argc ; i++)
    {
```

```
      if (!strcmp(argv[i], "-size"))
{
  if (++i == argc) print_help();
  if (!(size = atoi(argv[i]))) print_help();
  continue;
}
      if (!strcmp(argv[i], "-rand"))
{
  if (++i == argc) print_help();
  depth = atoi(argv[i]);
  if (++i == argc) print_help();
  n = atoi(argv[i]);
  rand = true;
  if (depth < 0)
    depth = -depth;
  else
    srand(time(0));
  continue;
}
      if (!strcmp(argv[i], "-raw"))
{
  raw = true;
  continue;
}
      if (!strcmp(argv[i], "-prolix"))
{
  laconic = false;
  continue;
}
      print_help();
    }

  global_net.create(size);

  if (rand)
    while (n--)
      {
rand_mat(U);
sk_results(U, depth, raw);
      }
  else
    /* take requests from stdin */
    while (cin.good())
      {
cin >> U[0] >> U[1] >> U[2] >> U[3] >> depth;
assert(fabs(double(vec_norm(U, 4)) - 1) < 1e-6);
sk_results(U, depth, raw);
      }

  return 0;
}
```

A.3 Epsilon finding

```
/*
```

```
   Given an epsilon-net, or set of points (stars) on the 3-sphere
   such that all points on the 3-sphere are within epsilon of some
   star, we wish to determine epsilon.  The basis behind the
   algorithm is that we only need examine points that are
   equidistant from the four closest stars.

   Aram Harrow Jan-May 2001
   spiritual guidance and network code from Jason Taylor
*/

#include <time.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <unistd.h>
#include <dirent.h>
#include <signal.h>

#include "su2.h"
#include "utils.h"

bool laconic = false, expand = false;

/* helper functions for qsort */
int int_comp_func(const void *a, const void *b)
{return sgn(*(int*)a - *(int*)b);}

double *dbl_index_comp_array;
int dbl_index_comp_func(const void *a, const void *b)
{
  double d=dbl_index_comp_array[*(int*)a]-dbl_index_comp_array[*(int*)b];
  return d<0 ? -1 : d>0 ? 1 : 0;
}

int dbl_index_comp_func_3(const void *a, const void *b)
{return sgn(dbl_index_comp_array[*(int*)a*3] -
    dbl_index_comp_array[*(int*)b*3]);}

int *int_index_comp_array;
int int_index_comp_func(const void *a, const void *b)
{
  int i=int_index_comp_array[*(int*)a]-int_index_comp_array[*(int*)b];
  return i<0 ? -1 : i>0 ? 1 : 0;
}

/*
   Implements a 3-D space-filling hilbert curve.
   It uses int_index_comp_array, to which we assume inv_gray has already
   been applied.  This is used to attempt to make neighboring points
   approximately near each other.
*/
int gray_triple_compare(int x, int y)
{
```

```
  int i, j, bit, diff;
  for (i=0 ; i<3 ; i++)
    for (bit=512<<(i*10), j=0 ; j<10 ; bit>>=1, j++)
      if ((diff = sgn((x & bit) - (y & bit))) != 0)
return diff;
  return 0;
}

/* if the bits of a are written a[1], a[2], ... a[10], then return
   a number of the form a[1], b[1], c[1], a[2], ... c[10] */

/* helper define */
#define GO(x,bit,offset) ((x&(1<<bit))<<(2*bit+offset))
int gray_order(int a, int b, int c)
{
  return GO(a,0,0) + GO(a,1,0) + GO(a,2,0) + GO(a,3,0) + GO(a,4,0) +
    GO(a,5,0) + GO(a,6,0) + GO(a,7,0) + GO(a,8,0) + GO(a,9,0) +
    GO(b,0,1) + GO(b,1,1) + GO(b,2,1) + GO(b,3,1) + GO(b,4,1) +
    GO(b,5,1) + GO(b,6,1) + GO(b,7,1) + GO(b,8,1) + GO(b,9,1) +
    GO(c,0,2) + GO(c,1,2) + GO(c,2,2) + GO(c,3,2) + GO(c,4,2) +
    GO(c,5,2) + GO(c,6,2) + GO(c,7,2) + GO(c,8,2) + GO(c,9,2);
}
#undef GO

int gray_triple_comp_func(const void *a, const void *b)
{
  int x, y;
  x = int_index_comp_array[*(int *)a];
  y = int_index_comp_array[*(int *)b];
  return gray_triple_compare(x, y);
}

/*
 * This data structure stores the net of points for which we want to
 * compute epsilon and contains most of the core functionality of the
 * code except for communication with other processes.
 */
struct net
{
  /* points represented in both formats */
  double *param, *mat;
  int no_pts;

  /* hash size */
  int hs[3];

  /* hash multiplier: hm[i] = 1.0 / hs[i] */
  double hm[3];

  /* keep track of the hash indices of each point */
  int *hash_order;

  net()
  {
    param = mat = NULL;
    no_pts = -1;
```

```
    hash_order = NULL;
  }

  ~net() {reset();}

  /* most of these are explained in the print_help() function */
  bool sanity_check();
  void random_points(int n);
  void skewed_random(int n);
  void uniform_param(int n);
  bool load_points(char *fn, bool skip=false);
  bool load_web(char *url, char *cache_fn);
  bool load_cache(char *fn, char *url);
  void process_loaded_points();
  void process_points();
  void save_points(char *fn);
  void print_nearest();
  double check_distance(double m[], bool silent=false);
  double check_random(int n);
  double check_ga(int iter, int n=1000);
  double check_slide(int n);
  double compute_midpoint(double *mid, int *indices);
  void set_no_blocks(int no_blocks);
  double find_epsilon();
  void print_points();
  int hash_val(double *p);
  void reset();
};

/*
 * parallel is a base class which handles the parallelism (or lack
 * thereof) of the epsilon finder.  It is responsible for returning
 * which block should be checked, when the program is finished and
 * keeping track of epsilon and a 4-tuple of verifying points.  Its
 * descendants are file_parallel (for parallel processes that
 * communicate via a shared directory) and web_parallel (for processes
 * that use the eps-server.cc web server to coordinate themselves.)
 * The base class parallel will cause the program to run alone
 * (i.e. no parallelism).
 */

struct parallel
{
  /* total number of blocks to be processed */
  int no_blocks;

  /* current block */
  int cur;

  /* highest value of epsilon seen so far */
  double epsilon;

  /* indices of four points that prove this */
  int verify[4];

  /* last time we've checked for a new value of epsilon with read_world() */
```

```
  time_t last_time;

  /* how long to wait before calling read_world() */
  int time_between_checks;

  /* the epsilon net controlling the process */
  net *net_ptr;

  /* whether someone else finished the block we're currently working on */
  bool scooped;

  /* count the amount of computation done */
  long long int dist_count;
  int m5_count, pt_count;

  /* create a unique identifier for each net/hash combination */
  char consistency_check[6];

  parallel()
  {
    verify[0] = verify[1] = verify[2] = verify[3] = 0;
    cur = -1;
    no_blocks = 0;
    epsilon = 0.0;
    last_time = 0;
    time_between_checks = 10;
    net_ptr = NULL;
    scooped = false;
    dist_count = 0;
    m5_count = pt_count = 0;
    consistency_check[0] = 0;
  }

  virtual ~parallel() {reset();}

  virtual void init(int size, net *data);
  virtual void read_world() {last_time = time(0);}
  virtual int next_block() {cur++; return cur=((cur==no_blocks)?-1:cur);}
  virtual void finish_block() {}
  virtual void report_results() {}
  virtual void finish_all() {}
  virtual void describe(int hash_size, int no_pts, char *desc) {}
  virtual void reset() {}

  /* check things every once in a while */
  void check_world() {if (time(0)-last_time>time_between_checks) read_world();}

  /* try to update epsilon from another process */
  void update_eps(double eps, int ver[4]);
} *multi = NULL;

/* All processes share a directory and each one has a seperate file
 * that shows its progress.  The filenames contain the consistency check
 * (to determine who shares with whom) and a unique host identifier to
 * avoid filename collisions.
 */
```

```
struct file_parallel : public parallel
{
  /* 0 untouched, 1 claimed, 2 finished */
  int *blocks;

  bool all_claimed;
  char *dir;
  char *fn1;
  FILE *file1;

  file_parallel() : parallel()
  {
    blocks = NULL;
    dir = fn1 = NULL;
    file1 = NULL;
  }

  virtual void init(int size, net *data);
  virtual void read_world();
  virtual int next_block();
  virtual void finish_block();
  virtual void report_results();
  virtual void reset();
};

/* file parallel functionality omitted for brevity */

/*
 * Parallelism is handled by a central server which allocates blocks
 * and saves a common set of verifying points.  This code is in
 * eps-server.cc.
 */
struct web_parallel : public parallel
{
  char *url1, *url2;
  int client_id;

  web_parallel() {client_id = -1; url1 = url2 = NULL;}
  ~web_parallel();

  char *wget_trap(char *addr, char *fn);
  char *wget_cmd(char *cmd, bool no_trap=false);

  virtual void describe(int hash_size, int no_pts, char *desc);
  virtual void init(int size, net *data);
  virtual void read_world();
  virtual int next_block();
  virtual void finish_block();
  virtual void report_results();
  virtual void finish_all();
  virtual void reset();
};

char base64(int from)
{
  from &= 63;
```

```
  return (from < 10) ? '0' + from :
    ((from < 36) ? 'a' - 10 + from :
     (from < 62) ? 'A' - 36 + from : '_');
}

void parallel::init(int size, net *data)
{
  int i, consistency;

  no_blocks = size;
  net_ptr = data;

  /* create a consistency check from the data and number of blocks */
  consistency = no_blocks;
  hash_ints(net_ptr->hash_order, net_ptr->no_pts, &consistency);
  for (i=0 ; i<5 ; i++)
    {
      consistency_check[i] = base64(consistency);
      consistency >>= 6;
    }
  consistency_check[5] = 0;
  printf("consistency check: %s\n", consistency_check);
}

void parallel::update_eps(double eps, int ver[])
{
  double eps_real;
  int i;
  if (eps > epsilon + 3e-6)
    {
      eps_real = net_ptr->compute_midpoint(NULL, ver);
      if (fabs(eps - eps_real) > 1e-2)
printf("false witness!! eps_real=%f eps=%f points=%d %d %d %d\n",
       eps_real, eps, ver[0], ver[1], ver[2], ver[3]);
      else
{
  epsilon = eps_real;
  for (i=0 ; i<4 ; i++) verify[i] = ver[i];
  printf("from another process, epsilon is now %f\n", epsilon);
}
    }
}

char *web_parallel::wget_trap(char *addr, char *fn)
{
  char *buf;
  int failure = 0;
  while (1)
    {
      buf = wget(addr, fn);
      if (!buf)
{
  assert(buf = new char[200]);
  sprintf(buf, "#Couldn't connect to host %s\n", addr);
}
      if (buf[0] != '#') return buf;
```

```
      printf("Server error:\naddr=%s\nfn=%s\nerror=%s\n", addr, fn, buf+1);
      delete buf;

      if (++failure == 6)
{
  printf("Six failures, time for closing.  Arrr.\n");
  exit(1);
}
      sleep(failure);
    }
}

char *web_parallel::wget_cmd(char *cmd, bool no_trap)
{
  char *buf, *result;
  buf = new char[strlen(url2) + strlen(cmd) + 100];
  sprintf(buf, "%s?%s+%d+%s", url2, consistency_check, client_id, cmd);
  result = no_trap ? wget(url1, buf) : wget_trap(url1, buf);
  delete buf;
  return result;
}

void web_parallel::describe(int hash_size, int no_pts, char *desc)
{
  char *buf = new char[strlen(desc)*3 + 100];

  sprintf(buf, "describe+%d+%d+", no_pts, hash_size);
  copy_web_str(buf + strlen(buf), desc);
  wget_cmd(buf, false);
  delete buf;
}

web_parallel::~web_parallel()
{
  reset();
  del(url1);
  del(url2);
}

void web_parallel::reset()
{
  char *temp = wget_cmd("stop", true);
  del(temp);
}

void web_parallel::init(int size, net *data)
{
  char buf[100], *result;
  int no_proc;

  parallel::init(size, data);
  client_id = -1;
  sprintf(buf, "start+%d", size);
  result = wget_cmd(buf);
  sscanf(result, "%d%d", &client_id, &no_proc);
  printf("%d other processes running with consistency check %s.\n",
```

```
 no_proc, consistency_check);
  if (!no_proc) printf("Either you are the first process or you're doing something differently from everyone el
  delete result;
}

void web_parallel::read_world()
{
  char *result;
  double eps;
  int ver[4], x;

  result = wget_cmd("status");
  sscanf(result, "%lf%d%d%d%d", &eps, ver, ver+1, ver+2, ver+3);
  update_eps(eps, ver);
  delete result;

  if (cur != -1)
    {
      result = wget_cmd("scooped");
      sscanf(result, "%d", &x);
      delete result;
      scooped = x;
    }

  last_time = time(0);
}

int web_parallel::next_block()
{
  char *result = wget_cmd("request");
  sscanf(result, "%d", &cur);
  scooped = false;
  delete result;
  return cur;
}

void web_parallel::finish_block()
{
  delete wget_cmd("finish");
  report_results();
}

void web_parallel::report_results()
{
  char buf[200];
  sprintf(buf, "update+%f+%d+%d+%d+%d+%lld+%d+%d", epsilon,
  verify[0], verify[1], verify[2], verify[3],
  dist_count, m5_count, pt_count);
  delete wget_cmd(buf);
}

void web_parallel::finish_all()
{delete wget_cmd("stop");}

void net::reset()
{
```

```
  no_pts = -1;
  if (param) delete param;
  if (mat) delete mat;
  if (hash_order) delete hash_order;
}

bool net::load_points(char *fn, bool skip)
{
  FILE *in = fopen(fn, "rt");
  double *temp;
  int max_pts, i=0;

  if (!in)
    {
      fprintf(stderr, "Couldn't open input file %s\n", fn);
      return false;
    }
  assert(mat = new double[max_pts = 512]);
  no_pts = 0;
  while (!feof(in))
    {
      fscanf(in, "%lf ", &mat[no_pts]);
      if (skip && ((i++ & 7) > 3)) continue;
      if (++no_pts == max_pts)
{
  assert(temp = new double[max_pts *= 2]);
  memcpy(temp, mat, sizeof(double) * no_pts);
  delete mat;
  mat = temp;
}
    }
  fclose(in);
  process_loaded_points();
  process_points();
  return true;
}

void net::save_points(char *fn)
{
  FILE *out = fopen(fn, "wt");
  int i;
  for (i=0 ; i<no_pts*4 ; i++)
    fprintf(out, "%f%c", mat[i], (i+1)&3 ? ' ' : '\n');
  fclose(out);
}

void net::process_loaded_points()
{
  int i;

  if (!laconic)
    printf("%d doubles loaded, %d points\n", no_pts, no_pts/4);
  no_pts /= 4;
  assert(param = new double[no_pts*3]);
  for (i=0 ; i<no_pts ; i++)
    {
```

```
      if (fabs(vec_norm(&mat[i*4], 4)-1) > 1e-3)
printf("error in line %d: %f %f %f %f not on sphere.\n",
       i+1, mat[i*4], mat[i*4+1], mat[i*4+2], mat[i*4+3]);
      mat_to_param(&mat[i*4], &param[i*3]);
    }
}

bool net::load_web(char *url, char *cache_fn)
{
  char *p1, *p2, *result, *tok;
  vector<double> pts;

  split_ip(url, &p1, &p2);
  result = wget(p1, p2);
  delete p1;
  delete p2;
  if (!result) return false;
  no_pts = 0;
  tok = strtok(result, " \n");
  while (tok)
    {
      pts.push_back(atof(tok));
      tok = strtok(NULL, " \n");
    }
  assert(mat = new double[no_pts = pts.size()]);
  memcpy(mat, pts.begin(), no_pts * sizeof(double));
  process_loaded_points();
  process_points();
  if (cache_fn) save_points(cache_fn);
  return true;
}

bool net::load_cache(char *fn, char *url)
{
  if (fexist(fn))
    {
      if (!laconic) printf("%s found, using cached version.\n", fn);
      return load_points(fn);
    }
  else
    {
      if (!laconic) printf("%s not found, downloading from %s\n", fn, url);
      return load_web(url, fn);
    }
}

/* returns epsilon achieved or 0.0 if something else is closer */
double net::compute_midpoint(double *mid, int *indices)
{
  double a1[4], a2[4], a3[4], a4[4], a4neg[4], dpos, dneg, m0[4], tm[4];
  double a4param[3];
  int i;

  param_to_mat(&param[indices[0]*3], m0);

  param_to_mat(&param[indices[1]*3], tm);
```

```
  for (i=0 ; i<4 ; i++)
    a1[i] = tm[i] - m0[i];

  param_to_mat(&param[indices[2]*3], tm);
  for (i=0 ; i<4 ; i++)
    a2[i] = tm[i] - m0[i];

  param_to_mat(&param[indices[3]*3], tm);
  for (i=0 ; i<4 ; i++)
    a3[i] = tm[i] - m0[i];

  normalize(a1);
  subtract_off(a2, a1);
  normalize(a2);
  subtract_off(a3, a1);
  subtract_off(a3, a2);
  normalize(a3);
  do
    {
      for (i=0 ; i<4 ; i++) a4[i] = uniform();
      subtract_off(a4, a1);
      subtract_off(a4, a2);
      subtract_off(a4, a3);
    }
  while (sqr(a4[0])+sqr(a4[1])+sqr(a4[2])+sqr(a4[3])<1e-8);
  normalize(a4);

  dpos = mat_dist_sqr(a4, m0);
  for (i=0 ; i<4 ; i++) a4neg[i] = -a4[i];
  dneg = mat_dist_sqr(a4neg, m0);

  if (dneg < dpos)
    {
      dpos = dneg;
      for (i=0 ; i<4 ; i++) a4[i] = a4neg[i];
    }

  if (mid) memcpy(mid, a4, 4 * sizeof(double));

  /* test to see that no point is closer */
  dpos = sqrt(dpos);
  dneg = dpos * 0.99999;  /* use a relaxed inequality */
  mat_to_param(a4, a4param);
  for (i=0; i<no_pts ; i++)
    if (param_dist(a4param, &param[i*3]) < dneg)
      return 0.0;
  return dpos;
}

void net::set_no_blocks(int no_blocks)
{
  int i;
  for (i=0 ; i<3 ; i++)
    {
      hs[i] = no_blocks;
      hm[i] = 1.0 / hs[i];
```

```
    }

  multi->init(hs[0] * hs[1], this);
}

/* For every x in our set, add the point -x, because we're actually
 * working with SO(3), not SU(2).  Then remove all
 * duplicates.  Things are confusing, since we don't ever actually
 * allocate storage for the new points, but just index them implicitly.
 */
void net::process_points()
{
  int i, j, k, midway, *hash_val, *temp;
  double *m1, *m2, temp_m[8], p[3], *old_mat, *old_param, avg, d;

  /* order the points by hash value */
  midway = no_pts;
  no_pts *= 2;
  assert(temp = new int[no_pts]);
  assert(hash_val = new int[no_pts]);
  assert(hash_order = new int[no_pts]);
  for (i=0 ; i<no_pts ; i++)
    {
      /* stupid legacy stuff */
      memcpy(p, &param[(i%midway)*3], 3 * sizeof(double));
      if (i>=midway)
{
  p[1] = fmod(p[1] + 0.5, 1.0);
  p[2] = fmod(p[2] + 0.5, 1.0);
}

      /* hash as finely as possible to ensure duplicates are found */
      hash_val[i] = inv_gray(int((p[2]-1e-9)*1024)) +
((inv_gray(int((p[1]-1e-9)*1024)) +
  (inv_gray(int((p[0]-1e-9)*1024)) << 10)) << 10);
      temp[i] = gray_order(inv_gray(int((p[0]-1e-9)*1024)),
   inv_gray(int((p[1]-1e-9)*1024)),
   inv_gray(int((p[2]-1e-9)*1024)));
      hash_order[i] = i;
    }
  int_index_comp_array = temp;
  my_qsort_int(temp, hash_order, no_pts);
  i=j=0;
  m1 = temp_m + 4; m2 = temp_m;
  memcpy(m1, &mat[4*(hash_order[0]%midway)], 4*sizeof(double));
  avg = 0;
  while (1)
    {
      hash_order[j++] = hash_order[i++];
      if (i == no_pts) break;
      swap(m1, m2);
      memcpy(m1, &mat[4*(hash_order[i]%midway)], 4*sizeof(double));
      if ((hash_order[i]>=midway) != (hash_order[i-1]>=midway))
for (k=0 ; k<4 ; k++)
  m2[k] = -m2[k];
      d = mat_dist_sqr(m1, m2);
```

```
      avg += d;
      if (d < 1e-10) j--;
    }

  no_pts = j;

  if (!laconic)
    printf("After processing, %d points remain.\n average distance = %f\n",
   no_pts, avg/no_pts);

  delete temp;

  /* actually rearrange them according to the new ordering */
  old_param = param;
  assert(param = new double[no_pts * 3]);
  old_mat = mat;
  assert(mat = new double[no_pts * 4]);
  for (i=0 ; i<no_pts ; i++)
    {
      j = hash_order[i];
      k = (j>=midway);
      if (k) j -= midway;
      memcpy(&mat[i*4], &old_mat[j*4], 32);
      memcpy(&param[i*3], &old_param[j*3], 24);
      if (k)
{
  for (k=0 ; k<4 ; k++)
    mat[i*4 + k] = -mat[i*4 + k];
  param[i*3 + 1] = fmod(param[i*3 + 1] + 0.5, 1.0);
  param[i*3 + 2] = fmod(param[i*3 + 2] + 0.5, 1.0);
}
      hash_order[i] = hash_val[hash_order[i]];
    }
  delete old_mat;
  delete old_param;
  delete hash_val;
}

/* the patch-based method of finding epsilon:
   time = O(p * (n + m^5)), where p is the number of patches, n is the number
   of points and m is the typical numbers of neighbors of each patch */
double net::find_epsilon()
{
  int *nearby, no_nearby, expand_by;
  int x, y, z, i, j, k;
  double loc[3], loc2[3], center[3], b4param[3], rad, dist, closest=0, *min_dist;
  double b1[4], b2[4], b3[4], b4[4], dpos, mata1[4], temp_mat[4];
  double b4neg[4], dneg, dtest;
  int a1, a2, a3, a4, a5;
  bool fuckit;
  long long int last_dist_count = 0;
  int good_count = 0, within_count = 0;
  int nearby_aborted = 0, none_nearby = 0, patch_count = 0;

  multi->epsilon = 0.0;
```

```
  assert(nearby = new int[no_pts]);
  assert(min_dist = new double[no_pts]);

  expand_by = MAX(5, no_pts / 1000);

  delete mat;
  mat = new double[0];

  multi->read_world();

  /*
     The main algorithm: iterate through all patches and look for
     extremal points inside, resulting from nearby 4-tuples.
  */
  for (multi->next_block() ; multi->cur!=-1 ; multi->next_block())
    {
      x = multi->cur / hs[1];
      y = multi->cur % hs[1];
      loc[0] = x * hm[0];
      loc2[0] = loc[0] + hm[0];
      center[0] = loc[0] + hm[0] / 2;
      loc[1] = y * hm[1];
      loc2[1] = loc[1] + hm[1];
      center[1] = loc[1] + hm[1] / 2;

      for (z=0, loc[2]=0.0 ; z<hs[2] ; z++, loc[2]=loc2[2])
{
  multi->check_world();
  if (multi->scooped) break;
  patch_count++;

  loc2[2] = loc[2] + hm[2];
  center[2] = loc[2] + hm[2] / 2;

  /* 1) compute patch radius */
  rad = param_patch_radius(loc, loc2);
  multi->dist_count++;
  fuckit = false;

  /* 2) first look at nearby points */
  closest = MAX_DIST;
  i = k = 0;
  j = no_pts;
  a1 = inv_gray(int((center[2]-1e-9)*1024)) +
    ((inv_gray(int((center[1]-1e-9)*1024)) +
      (inv_gray(int((center[0]-1e-9)*1024)) << 10)) << 10);

  while (i<j)
    {
      k = (i + j) >> 1;
      a2 = gray_triple_compare(hash_order[k], a1);
      if (a2 > 0)
j = k;
      else
i = k + 1;
    }
```

```
  i = MAX(0, k - expand_by);
  j = MIN(no_pts, k + expand_by);
  for ( ; i<j ; multi->dist_count++, i++)
    if ((dist = param_dist(&param[i*3], center)) < closest)
      if ((closest = dist) + rad < multi->epsilon)
fuckit = true;

  /* 3a) now iterate through all points, discarding those
     whose  minimum distance is greater than closest + rad */

  no_nearby = 0;
  for (i=0 ; !fuckit && i<no_pts ; i++)
    {
      dist = min_dist_to_patch(&param[i*3], loc, loc2);
      multi->dist_count++;

      if (dist < closest + rad)
{
  /* add this point to the list of nearby candidates */
  min_dist[no_nearby] = dist;
  nearby[no_nearby++] = i;

  dist = param_dist(&param[i*3], center);

  if (dist < closest)
    if ((closest = dist) + rad < multi->epsilon)
      fuckit = true;
  multi->dist_count++;
}
    }

  /* if we're never going to affect epsilon, then don't try */
  if (fuckit) continue;

  /* 3b) once we have the best value of closest that we'll get,
     go back through and cull as many points as possible */
  closest += rad;
  for (i=j=0 ; i<no_nearby ; i++)
    {
      min_dist[j] = min_dist[i];
      nearby[j] = nearby[i];
      if (min_dist[i] < closest) j++;
    }
  no_nearby = j;

  /* 4) do the O(m^5) part - iterate through all 4-tuples and
     check if the point falls in the patch and has no closer points
  */
  if (no_nearby > 500)
    printf("m=%d\n\n  The horror!  The horror!\n", no_nearby);
  multi->m5_count++;
  for (a1=0 ; a1<no_nearby ; a1++)
    {
      param_to_mat(&param[nearby[a1]*3], mata1);
      for (a2=a1+1 ; a2<no_nearby ; a2++)
{
```

```
  param_to_mat(&param[nearby[a2]*3], temp_mat);
  for (i=0 ; i<4 ; i++)
    b1[i] = temp_mat[i] - mata1[i];
  if (normalize(b1) < 1e-9) continue;
  for (a3=a2+1 ; a3<no_nearby ; a3++)
    {
      param_to_mat(&param[nearby[a3]*3], temp_mat);
      for (i=0 ; i<4 ; i++)
b2[i] = temp_mat[i] - mata1[i];
      subtract_off(b2, b1);
      if (normalize(b2) < 1e-9) continue;
      for (a4=a3+1 ; a4<no_nearby ; a4++)
{
  /* this whole inner loop kind of hurts */
  multi->dist_count += 5;
  multi->pt_count++;
  fuckit = false;

  param_to_mat(&param[nearby[a4]*3], temp_mat);
  for (i=0 ; i<4 ; i++)
    b3[i] = temp_mat[i] - mata1[i];
  subtract_off(b3, b1);
  subtract_off(b3, b2);
      continue;
    }
  do
    {
      for (i=0 ; i<4 ; i++) b4[i] = uniform();
      subtract_off(b4, b1);
      subtract_off(b4, b2);
      subtract_off(b4, b3);
    }
  while (normalize(b4) < 1e-9);

  dpos = mat_dist(b4, mata1);

  /* either b4 or -b4 is the point we want */
  for (i=0 ; i<4 ; i++) b4neg[i] = -b4[i];
  dneg = mat_dist(b4neg, mata1);
  if (dneg < dpos)
    {
      dpos = dneg;
      for (i=0 ; i<4 ; i++) b4[i] = b4neg[i];
    }

  /* test to see if it's in the patch */
  mat_to_param(b4, b4param);
  for (i=0 ; i<3 ; i++)
    if (b4param[i] < loc[i] || b4param[i] > loc2[i])
      fuckit = true;

  if (!fuckit) within_count++;

  /* test to see that no point is closer */
  for (a5 = 0; a5<no_nearby ; a5++)
    if (param_dist(b4param,&param[nearby[a5]*3])
```

```
< dpos*0.99999)
      {
fuckit = true;
break;
      }
  multi->dist_count += a5;

  if (fuckit) continue; /* for a4=... */

  good_count++;

  if (dpos > multi->epsilon)
    {
      multi->epsilon = dpos;
      if (!laconic)
{
  printf("Epsilon is now %f\n", multi->epsilon);
  fflush(stdout);
}

      multi->verify[0] = nearby[a1];
      multi->verify[1] = nearby[a2];
      multi->verify[2] = nearby[a3];
      multi->verify[3] = nearby[a4];
      multi->report_results();

      dneg=MAX_DIST; j=0;
      for (a5=0;a5<no_pts;a5++)
if ((dtest=param_dist(param + a5*3, b4param))<dneg)
  {dneg=dtest; j=a5;}

      if (fabs(dneg-dpos)>1e-4)
{
  printf("bad extreme point.\n");
  print_vec("loc", loc, 3);
  print_vec("loc2", loc2, 3);
  printf("rad=%f, closest=%f\n", rad, closest);
  for (i=0;i<no_nearby;i++)
    {
      char buf[40];
      sprintf(buf, "pnear(%d,:)", i+1);
      print_vec(buf, param + nearby[i]*3, 3);
    }
  printf("a1=%d; a2=%d; a3=%d; a4=%d\n",
 a1+1, a2+1, a3+1, a4+1);
  print_vec("b4", b4, 4);
  print_vec("closest param", param+j*3, 3);
  printf("min_dist_to_patch=%f\n", min_dist_to_patch(param+j*3, loc, loc2));
  printf("dpos=%f, actual=%f\n", dpos, dneg);
  assert(0);
}
    }
}
    }
}
    }
```

```
}

      multi->finish_block();
      if (!laconic && (multi->dist_count > last_dist_count + 10000000))
{
  last_dist_count = multi->dist_count;
  printf("%d patches, %d aborted nearby, %d full scans\n",
 patch_count, nearby_aborted, none_nearby);
  printf("%d O(m^5) checks, %lld distance checks, %d 4-tuples, %d closest\n",
 multi->m5_count, multi->dist_count, multi->pt_count, good_count);
  fflush(stdout);
}
    }

  delete nearby;
  delete min_dist;

  if (!laconic)
    {
      printf("%d patches, %lld distance checks, %d O(m^5), %d 4-tuples\n",
     patch_count, multi->dist_count, multi->m5_count, multi->pt_count);

      compute_midpoint(b4, multi->verify);
      printf("%d %d %d %d\n", multi->verify[0], multi->verify[1], multi->verify[2], multi->verify[3]);
      print_vec("An example of an extremal point is:\n  x", b4, 4);
      printf("\nThis point has a least distance of %f\n", multi->epsilon);
    }
  multi->finish_all();
  return multi->epsilon;
}

/* This lists all the command-line options.  Note that some of the
   corresponding functions have been omitted from this code listing
   for brevity */
void print_help()
{
  printf("epsilon [options]\n");
  printf("In case of trouble, make sure the parameters are in the listed order.\n");
  printf("  -laconic: print only the final output\n");
  printf("  -timer: set the random number seed to the system time\n");
  printf("  -rand <n>: test n random points\n");
  printf("  -skewed <n>: try to pick random points far from each other\n");
  printf("  -uniform <n>: pick n^3 points uniformly in parameter space\n");
  printf("  -load <fn>: load points for a file\n");
  printf("  -load-skip <fn>: skip 2 out of every 4 lines\n");
  printf("  -load-web <url>: load the points from a web site\n");
  printf("  -load-web-cache <fn> <url>: only download the points once\n");
  printf("  -hash <n>: hash into n intervals (default=100)\n");
  printf("  -sanity: first make sure the primitives work correctly\n");
  printf("  -check <a> <b> <c> <d>: check the distance for a point\n");
  printf("  -check-ind <a> <b> <c> <d>: specify 4-tuple indices instead\n");
  printf("  -print-ind <a> <b> ...: print out several points.\n");
  printf("  -check-rand <n>: check the distance for n random points\n");
  printf("  -check-ga <n>: do a GA distance check for n iterations\n");
  printf("  -check-slide <n>: slide from random points to extreme points\n");
  printf("  -expand: turn on a questionable optimization\n");
```

```
  printf("  -use-hash: another dubious optimization\n");
  printf("  -parallel <dir>: many processes share the same directory\n");
  printf("  -distributed <url>: parallel processes using a web server\n");
  printf("  -nice <n>: reduce process priority with the nice command\n");
  printf("  -master-skip <n>: skip first n lines of the master file\n");
  printf("  -master <url>: load a list of commands from a master list\n");
  exit(1);
}

/* if the program crashes, then try to tell the central server first */

/* signal-handling code due to Jason Taylor omitted to keep things short */

/* Most of the main function was also omitted.
   Below is an example of how the interface is used */
int main(int argc, char **argv)
{
  net N;
  N.set_no_block(200);
  N.load_points("filename");
  printf("epsilon is %f\n", N.find_epsilon());
  return 0;
}
```