Spring 2019

Zero and Low Energy Thresholds in Quantum Simulation

Yun Xuan Shi
San Jose State University

ZERO AND LOW ENERGY THRESHOLDS IN QUANTUM SIMULATION

A Thesis

Presented to

The Faculty of the Department of Physics & Astronomy

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Yun Xuan Shi

May 2019

The Designated Thesis Committee Approves the Thesis Titled

ZERO AND LOW ENERGY THRESHOLDS IN QUANTUM SIMULATION

by

Yun Xuan Shi

APPROVED FOR THE DEPARTMENT OF PHYSICS AND ASTRONOMY

SAN JOSÉ STATE UNIVERSITY

May 2019

Dr. Chris Pollett	Department of Computer Science
Dr. Peter Beyersdorf	Department of Physics and Astronomy
Dr. Thomas Madura	Department of Physics and Astronomy

ABSTRACT

ZERO AND LOW ENERGY THRESHOLDS IN QUANTUM SIMULATION

by Yun Xuan Shi

Quantum simulation is the process of simulating a quantum mechanical system using either a quantum or a classical computer. Because quantum mechanical systems contain a large number of entangled particles, they are hard to simulate on a classical computer. It is the task of computational complexity theorists to estimate the amount of resources to do the same number of operations on either classical or quantum devices. This report first summarizes the state of the art in the field of quantum computing, and gives an example of a model of quantum computer and examples of quantum algorithms that are currently being researched. Then our own research about *k*-local quantum Hamiltonians is discussed. We developed programs to determine if a particular kind of *k*-local Hamiltonian has zero-energy solutions. First, to familiarize ourselves with quantum algorithms, we implemented a recently discovered polynomial-time 2-QSAT algorithm called SolveQ. Then we wrote several versions of brute force 7-variable 3-QSAT solvers and conducted experiments for the threshold of satisfiability. We empirically determined that the thresholds for the four versions, Versions 3, 4, 5, and 6, are 0.741, 1.714, 1.714, and 0.571, respectively. In addition, experiments were conducted involving the 6-qubit Ising model, working on which caused us to realize how inefficient the classical computer really is at simulating quantum mechanical systems. Our conclusion is that quantum simulation is much less feasible than classical simulation on a classical computer.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Chris Pollett, for coming up with such a fascinating research opportunity for me, as well as this rewarding thesis topic. It is amazing that I have come so far, from knowing nothing about quantum computing to knowing so much. Prof. Pollett taught me the wonderful lesson of team work because we need other people in this life for things we cannot do on our own. It is with Prof. Pollett's help that I came to realize how I can become a contributing member of society after I graduate.

Thanks to all the professors on my thesis committee, for their diligent reviewing of this thesis and providing helpful comments. It is through their dedication that my thesis project was finally successfully completed.

I thank my parents for everything: allowing me to major in physics, and providing support and encouragement for me as I complete my degree. Without family support, it would have been hard for me to succeed. I am grateful for their assistance and guidance. I especially have to thank my father for his suggestions, support and encouragement, and for the fact that he was the one who first told me about quantum computing when I was young.

Basically, it was through everyone's tremendous support that I was able to complete this culminating project of my graduate studies. I am very fortunate to have met all these people. Physics is a challenging subject, and I really would not have come to understand physics like I do now without the guidance and enlightenment of all the teachers I met along the way.

Last but not least, I would like to thank SJSU for the diverse, supportive and welcoming environment that gave me so many positive memories that I will look back on years from now.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The dynamics of quantum particle systems are described by their wave functions, which evolve in time due to quantum Hamiltonians, operators that determine the total energy of the systems. This thesis is concerned with these Hamiltonians, and whether simulations of quantum algorithms can be done on classical computers. We are interested in the computation of the ground state (zero-energy) and low energy solutions of quantum Hamiltonians. We are also interested in the phase transition from where the quantum systems have zero or low energy solutions to where they do not. This phase transition is determined by a ratio between the number of clauses and total number of variables, called the freezing point ratio. It turns out that reaching above a certain number of clauses in a given problem instigates these transitions, for both classical and quantum simulations. Therefore, understanding such transitions provides insights into how quantum computation and classical computation differ in strength. In this chapter, we describe our project in the context of what is known about quantum Hamiltonians.

A *k*-local Hamiltonian is an operator that looks like: $H = \sum_j H_j$ where $j=1,2,\ldots,m$ for *m* clauses, and $H_j = P_j(C_j \otimes I_{2^{n-k}})P_j^{-1}$. Here C_j is a *k*-qubit gate, and P_j is a permutation matrix for *n* qubits. We are interested in *k*-local Hamiltonians because it is known that we can express *k*-SAT problems as decision problems of whether there are zero-energy solutions to the specific type of *k*-local Hamiltonians used to solve *k*-SAT problems (Beaudrap & Gharibian, 2015). In classical complexity theory, *k*-SAT is the problem of computing a satisfying assignment for a Boolean formula that consists of several constituent OR clauses of at most *k* variables or their negations.

This problem is known to be NP-hard (Nondeterministic Polynomial Time Hard), the class of problems which are at least as hard as those in the class of problems that have polynomial-time verifiable proofs. In history, *k*-local Hamiltonians were first considered by Kitaev, in the context of the class QMA (Quantum Merlin Arthur), the quantum analogue of NP. Kitaev was interested in this topic, because if we can solve *k*-local Hamiltonian problems on a quantum computer then we can solve all QMA problems, and thereby NP problems, since *k*-local Hamiltonians are the hardest type of QMA problems. We also study *k*-local Hamiltonians because models existing in nature, such as the Hubbard and Ising models, that can be used for information processing, can be represented as *k*-local Hamiltonians. It is known that any *k*-local Hamiltonian can be built using polynomial resource quantum circuits, even though building them with classical circuits is inefficient because the circuits will likely require exponential resources. This is why a quantum computer is superior. A model of the quantum simulation is discussed in source (Nielsen & Chuang, 2010). When one studies a physical system, one is often interested in its eigenenergy spectrum and the corresponding set of eigenvectors. A problem is called QMA-complete if it is both in QMA and is QMA-hard, and like NP-complete, it has polynomial-time verifiable proofs. Determining the lowest energy solutions for *k*-local Hamiltonians is known to be QMA-complete, even for *k* as low as *k*=2. This hardness also holds for a subclass of *k*-local Hamiltonians known as *k*-QSAT, which is the quantum equivalent of classical *k*-SAT problems. A good survey of quantum Hamiltonian complexity is (Gharibian, Huang, Landau & Shin, 2014). In this chapter, we present our work in the context of what is known about *k*-local Hamiltonians.

A literal is a Boolean variable or its negation. A *k*-SAT problem is a constraint satisfaction problem in which several disjunctions (OR clauses) of literals are joined together in a conjunction (AND clause). Let *n* be the number of variables in such a Boolean formula. For *k*-SAT, we restrict the number of variables in each clause to a maximum of *k*. Given such a formula φ, the solution must satisfy all of the OR clauses in the problem simultaneously. Since a *k*-SAT clause can be viewed as a special case of a *k*-QSAT clause, let us first understand what *k*-SAT clauses are. As an example, let us consider a simple OR clause of two variables, $(x_1 \vee \bar{x}_2)$. We can write the Hamiltonian constraint matrix representing this clause as H=$|0\rangle\langle 0| \otimes |1\rangle\langle 1|$. $|0\rangle\langle 0|$ is used because x_1 is not negated; $|1\rangle\langle 1|$ is used because x_2 is negated. Consider the basis quantum state as $|x_1\rangle \otimes |x_2\rangle$. The solutions to this OR clause are the null vectors of the constraint matrix H. So in this example, $|x_1\rangle \otimes |x_2\rangle$ = 00, 10 or 11 are the solutions, but 01 is not a solution. What was just described is a 2-SAT clause, but it is possible to construct a 3-SAT or *k*-SAT one by applying the same reasoning using more terms joined together in a tensor product. We know that for *k*-SAT, each OR clause in the problem can consist of just one forbidden vector with a probability of one; whereas, each OR clause in the *k*-QSAT problem contains 2^k terms with the sum of the probabilities equal to one. A *k*-QSAT clause is of the form $(I_k - |v\rangle\langle v|) \otimes I_{n-k}$ in which $|v\rangle$ are forbidden vectors summed together for a total of 2^k terms. For 3-QSAT, $|v\rangle$ might look like:

$$a|000\rangle + b|001\rangle + c|010\rangle + d|011\rangle + e|100\rangle + f|101\rangle + g|110\rangle + h|111\rangle$$

Notice that there are 8 terms, since 2^3=8. (Also note that each three digit term is an abbreviation for a matrix that is composed of three matrices joined together in a tensor product. For example, the term $|011\rangle$ means $|011\rangle = |0\rangle\langle 0| \otimes |1\rangle\langle 1| \otimes |1\rangle\langle 1|$). Because each coefficient represents the probability of its corresponding term, we have $a+b+c+d+e+f+g+h=1$.

To find the solution of *m* of these 3-QSAT clauses, H_j, joined by AND symbols, we simply add up the coefficients for each of the H_j and find the null vector of this entire matrix by doing Gaussian elimination, then look for zeros along the diagonal of the resulting matrix. One problem with implementing this on a computer is floating point round-off error in Gaussian elimination. One option to handle this problem is to set a small enough number as our threshold that must not be exceeded for a data slot to be counted as a zero.

We now discuss our work related to *k*-local Hamiltonians and *k*-QSAT. We first explored available tools for implementing quantum algorithms on the internet and implemented two famous algorithms, Grover's and Shor's algorithms. As we could find no tool directly geared for *k*-local Hamiltonian experiments, we wrote new tools to do this ourselves. Our first attempt at writing a quantum algorithm was Bravyi's 2-QSAT algorithm (Gharibian, 2015), which is named SolveQ. This is a polynomial time algorithm for finding 0-energy solutions for the 2-QSAT instances. Through coding, we found that SolveQ finds only one solution out of a large set of possible solutions, and sometimes it will also output that the problem is unsolvable when it is solvable. SolveQ is still a valid quantum algorithm because it does find a solution with probability greater than 1/2. The next *k*-QSAT solver that we wrote was one we imagined on our own for

solving general *k*-QSAT problems the brute force way, called "brute force," because this algorithm will give us all possible solutions to a single problem. This employs the Gaussian elimination approach that was described earlier. Although the code was written for *k*-QSAT problems, we actually did not collect data for any other value of *k* other than k=3. This is because the algorithms we implemented run in time $O(N^3)$ where $N=2^7$ since 7 is the number of qubits considered. This was the maximum size that could be computed quickly on the computer that was used.

After writing these quantum simulators in Java, simulating *k*-SAT and *k*-QSAT by brute force, experiments were conducted on the ability of the computer to solve these constraint satisfaction problems. We were interested in the threshold of satisfiability, classically known as the freezing point zone, where most of the problem instances go from satisfiable to not. As the number of clauses increases, the number of non-zero terms on the diagonal of the forbidden matrix, our quantum Hamiltonian, increases very fast; and the likelihood that the formula is satisfiable decreases very swiftly. Only in a narrow range, near the threshold or freezing point, is the problem actually "hard"— a word reserved in computational complexity to describe problems to which all other problems within a class can be reduced to. In source (Bravyi, Moore & Russel, 2014), it was shown that this freezing point ratio for classical algorithms is a number in the range 3.520-4.490, and for quantum algorithms it is in the range 0.818-3.594. This indicates that quantum algorithms freeze a lot more easily than classical algorithms, which leads to the conclusion that quantum clauses are more "constraining" than classical clauses. To test this idea, we generated 3-QSAT instances with from 5 to 35 clauses and found whether there are null vectors to the matrix representing their sum. For each fixed

number of clauses between 5 and 35, we completed 50 trials and computed the average number of times the instances were satisfiable. This allows us to plot graphs of average count of satisfiable instances versus number of clauses to determine an experimental freezing point ratio of 0.571 (Version 6), which demonstrates the fact that quantum constraints are more constraining since they take much fewer clauses to freeze, as seen by our own experiment as well as the value calculated in (Bravyi, Moore & Russel, 2014).

The next section consists of preliminaries, in which we discuss the basics of quantum mechanics and computational complexity needed to understand our results. We introduce Shor's and Grover's algorithms as examples of the types of algorithms employed by the quantum computer to do computation, and they are better at performing specified tasks than any algorithm used by classical computers. We then move on to our own results, in which we describe our implementation of Shor's and Grover's algorithms in jQuantum, which were built with the intent of testing whether there are existing quantum simulators for carrying out our threshold experiments. Next we discuss the designs of the quantum simulator programs that we wrote: first describing the 2-QSAT program and then the brute force *k*-QSAT solver followed by the Ising and Antiferromagnetic Heisenburg Models. We discuss the *k*-QSAT problem as a specific type of *k*-local Hamiltonian problem and survey known results. This is followed by a description of the freezing point experiments conducted. We then describe our Quantum Ising Model simulator, which is not yet practical due to limitations of the classical computer, but are nonetheless an interesting idea for future research. The last chapter is the Conclusion.

Chapter 2

Preliminaries

2.1 Quantum Mechanics

In order to understand the work we did on the quantum threshold of satisfiability, it is necessary to have a basic understanding of quantum mechanics, computational complexity, and quantum information processing. In this chapter, we attempt to provide a brief background about the physics behind quantum computers. The quantum Hamiltonians that we will discuss in this chapter derive from the Schrodinger equation, which is one of the most fundamental equations in physics. The Schrodinger equation is a partial differential equation describing the conservation of energy of a particle. The left side of the Schrodinger equation is the Hamiltonian applied to the wave function. The right hand side of the Schrodinger equation is an imaginary constant times the time-derivative of the wave function. The wave function is also called the quantum state. The two forms of the Schrodinger equation are:

$$\text{Time-independent: } H|\varphi\rangle = E|\varphi\rangle$$

$$\text{Time-dependent: } H|\varphi\rangle = i\eta\frac{\partial|\varphi\rangle}{\partial t}$$

Because we want to generate a variety of quantum gates, the version we are more interested in is the time-dependent version of the Schrodinger equation. The Hamiltonian on the left represents some energy related operation on the quantum state— in the ion trap, lasers are used as the energy sources for generating the Hamiltonian. The Hamiltonians in the Schrodinger equation are represented by Hermitian matrices, which

are matrices that have the property $H = H^{+}$. Hermitian matrices have the following properties:

1) Diagonal entries are real.
2) A matrix that has only real entries is Hermitian if and only if it is a symmetric matrix. A real and symmetric matrix is simply a special case of a Hermitian matrix.
3) Hermitian matrices are normal matrices, which are matrices that commute with their adjoints.
4) A Hermitian matrix can always be diagonalized by some unitary matrix. The resulting diagonal matrix has only real entries.
5) The eigenvalues of a Hermitian matrix are all real and its eigenvectors are linearly independent meaning they are orthogonal. Sometimes the eigenvalues are degenerate.
6) The sum of any two Hermitian matrices is also Hermitian.
7) The inverse of an invertible Hermitian matrix is Hermitian as well.
8) The product of two Hermitian matrices is Hermitian if and only if they commute.
9) $\langle \psi | H | \psi \rangle$ is always real for Hermitian matrix H.
10) Hermitian matrices do not form a vector space over complex numbers since I_n is Hermitian but $i\ I_n$ is not. However, the complex Hermitian matrices do form a vector space over the real numbers.

11) Take the eigenvectors of Hermitian matrix H and put them in a matrix. This matrix diagonalizes the Hermitian matrix.

12) The sum of a square matrix and its conjugate transpose is Hermitian. The difference of a square matrix and its conjugate transpose is skew-Hermitian. This implies that the commutator of two Hermitian matrices is skew-Hermitian. Thus, an arbitrary square matrix C can be written as the sum of a Hermitian matrix and a skew-Hermitian matrix.

13) The determinant of a Hermitian matrix is real.

14) A Hermitian matrix can be decomposed into its real and imaginary parts: the real part is symmetric and the imaginary part is skew-symmetric. Skew-symmetric describes a matrix whose transpose is the matrix multiplied by negative one.

2.2 Qubits

A quantum computer stores information in terms of qubits, which are two-level systems. Mathematically, the quantum states we are interested in are composed of these two-level systems that can be represented in a 2-dimensional vector space with basis $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. In physics, these two basis vectors are written using the shorter ket notation $|0\rangle$ and $|1\rangle$. A general state in the 2-dimensional state space can be written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Where α and $\beta \in$ complex and $|\alpha|^2 + |\beta|^2 = 1$. Mathematically, a qubit is represented by this vector with its origin at the origin of a sphere and the tip on the surface of the sphere. This sphere is called the Bloch sphere, and it has a radius of one. Quantum states are formed through the coupling of several qubits. It should be

noted that all vectors representing one quantum state before decoherence must have a length of one, even if the state is an entangled state. Such states are called pure states. Unlike pure states for quantum systems, qubit registers that have been partially measured or have interacted with the external environment, through a process called decoherence, often ends up in what is known as a mixed state. For a mixed state, $|\alpha|^2 + |\beta|^2 < 1$. Mixed states can be represented by points inside the Bloch sphere. A mixed qubit state has three degrees of freedom: the angles θ and φ and the length *r*. A mixed state that is generated through decoherence, might be unusable as quantum information. A qubit is very different from a classical bit, because if a classical bit is described as a vector, then it can only be either a vector pointing vertically upward, or a vector pointing vertically downward; whereas, a qubit can take on a whole sphere of values (Maslov, 2017). The three parameters are the phase of the complex coefficient α , the phase of the complex coefficient β and the length *r*, the square root of the square of the norms of the two coefficients.

Physical systems built from more than one qubit are represented in a state space which is the tensor product of the constituent states. Below is an example of how to take tensor products:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \bullet \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 0 \bullet \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

This can be generalized to arbitrary vectors and matrices. Solutions to the Schrodinger equation indicate how states change with respect to time as unitary matrices are applied to them. A unitary matrix is a matrix that preserves the norm of the quantum

state. Therefore, while a system is evolving, it will be a linear combination of different qubit states, with the square of the norm of the coefficients always adding up to one, but these coefficients change over time. When we observe the system, that is take a measurement, quantum mechanics says the measured qubits end up in a fixed state that will never change again. Measurements are modeled by measurement operators, operators with the following trait: $\sum_i M_i^+ M_i = I$. When measuring ρ, the density matrix $|\psi\rangle\langle\psi|$ of the current state $|\psi\rangle$, with respect to $\{M_i\}$, we obtain outcome *i* with probability given by: $\Pr(i \mid \rho) = Tr(M_i \rho M_i^+)$ (Gharibian, Huang, Landau & Shin, 2014). Even though a single qubit take on a whole sphere full of values, it can only be measured along a single axis at a time. But measuring the qubit changes its state from whatever it was before the measurement, to whatever state the measurement produced after the measurement, for example, for a 7-qubit state, this might be a tensor product of one states and zero states like 1101001— this state now occurs with a probability of one and will never change again. As a simple example on a single qubit, we can take measurements $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$. We see that it is true that although a qubit is originally mathematically very different from a classical bit, once it has been measured via M_0 or M_1, it becomes no different from a classical bit, which is no longer a superposition of the two possibilities. To summarize, once a qubit has been measured, it no longer retains memory of its past, which can never be restored. At this stage, the wave function is said to be collapsed, and it is collapsed forever. We can choose to collapse the entire register or only parts of it as will be discussed later in this paper.

The quantum Hamiltonians we consider operate on registers of qubits. To implement qubits in the real world, each qubit must be some sort of particle that is small enough so that it is inexpensive to make into a marketable computer. We conclude this section on qubits by briefly mentioning several real world systems that have been used to model qubits. All of the following particles have being used for qubits: photons, electrons and ions. For example, the polarization of a stream of light particles can be used to implement a quantum bit, with quartz crystals for quantum gates, and polarizers for measurement. To connect these real-world systems to *k*-local Hamiltonian, the subject of this paper, we will conclude this chapter with a discussion of a specific implementation called an ion trap that uses Ca40+ ions for its qubits.

2.3 Quantum Gates

For quantum computation to be possible, we must be able to process our qubits. First, we must be able to initialize all of our qubits to a known state. Second, we must be able to rotate individual qubits, measure individual qubits, perform operations that entangle pairs of qubits, and stay free of outside interference or decoherence for as long as it takes to finish our computation. We now describe the kinds of operations typically used in quantum computers: simple quantum logic gates.

Quantum logic gates operate on sets of qubits. Mathematically speaking, they cause a set of qubits to undergo unitary transformations. A unitary transformation corresponds to rotations of the quantum state vector on the Bloch sphere. For systems of qubits, it corresponds to something like a higher dimensional rotation. A standard basis measurement is an operation through which information is gained about the state of the qubit. For $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the result will be either up ($|\alpha|^2$) or down ($|\beta|^2$), with

the probability given in parentheses. A classical computer is built using classical logic gates: AND, OR and NOT. These gates are built from transistors and resistors, and operate by manipulation of currents and voltages. Mathematically, these classical gates are functions that take in two true-false inputs to generate one true-false output. The quantum computer uses quantum gates and different physical entities, such as spin and energy, to achieve information processing, which is what this section is about. To design a quantum algorithm, one must let go of one's classical intuition, using truly quantum effects to achieve the desired algorithmic end. Both quantum and classical algorithms, can be expressed as quantum Hamiltonians, which can then be built into quantum circuits. In our research of *k*-QSAT problems, we express classical OR clauses as quantum Hamiltonians, which we imagine are possible to build into quantum circuits.

Like a classical computer, a quantum computer has both hardware and software, both of which use algorithms to solve problems. Whereas classical computers use classical logic gates: AND, OR, NOT; quantum computers use quantum gates such as Pauli X, Y and Z, and CNOT. We will give the matrices for these in a moment. Coming up with a good quantum algorithm is difficult, so there are currently few quantum algorithms that actually perform better than their classical counterparts (Gharibian, Huang, Landau & Shin, 2014). Some examples of quantum algorithms are Deutch Josza, Shor's and Grover's algorithms. After calculating the results on a quantum computer, the next challenge is to extract the correct solution from a sea of wrong possibilities. The result of each quantum measurement is probabilistic. To determine the output of a quantum computation, one performs a large number of trials and takes the result with the highest frequency. This can be easily done using a normal computer and an algorithm which

allows you to tally the occurrences, reset the original assumptions, and try again with the quantum computer. It usually takes 3 or 4 runs for you to achieve your answer. The algorithms needed are the Grover's Algorithm and Shor's Algorithm. The articles (Wikipedia, 2017) are good starting points for understanding the Shor's and Grover's algorithms.

All quantum gates are unitary, so that they are reversible and therefore nondestructive of quantum information. Mathematically, a complex matrix U is unitary if $UU^{+} = I$. We write U^{+} for $(U^{*})^{T}$. Unitary matrices have the following properties:

1) Multiplication by unitary matrices preserves inner product that is $\langle Ux, Uy \rangle = \langle x, y \rangle$

2) U is normal (matrix N is normal means: $[N^{+}, N] = 0$), and U has complex eigenvalues that lie on the unit circle.

3) U is diagonalizable means given a unitary matrix V and a diagonal matrix D (also unitary), we have $U = V^{+}DV$.

4) $|\det(U)| = 1$

5) The eigenvectors of U are orthogonal.

6) U can be written as $U = e^{iH}$ where H is a Hermitian matrix.

7) Both rows and columns of U form an orthonormal basis.

8) U is called an "isometry," meaning that it preserves the norm of a vector.

9) The sum of two unitary matrices is not guaranteed to be unitary.

10) The product of two unitary matrices is also unitary.

The net effect of evolving a quantum system according to a Hamiltonian from the starting quantum state for a fixed amount of time is a unitary operation applied to the

quantum state. Three unitary gates we are especially interested in are the Hadamard, CNOT , and Phase gates:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-i2\pi/2^n} \end{pmatrix}$$

The Hadamard gate is a single-qubit gate used to achieve quantum parallelism. Both the $|0\rangle$ or $|1\rangle$ states are changed into equal superpositions of the $|0\rangle$ and $|1\rangle$ states after the Hadamard gate. The $|0\rangle$ state is changed into a sum of the $|0\rangle$ and $|1\rangle$ states, while the $|1\rangle$ state is changed into a difference of the $|0\rangle$ and $|1\rangle$ states. The sign probably makes a difference in generating probability distributions to achieve information processing. Another way of stating this is that the Hadamard gate is for changing back and forth between the computational basis and the bell basis. Here the bell basis is $|+\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$ and $|-\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right)$.

A CNOT gate is a two-qubit gate, often used to achieve entanglement, or achieve correlated data. Practically, the CNOT gate is a simple two qubit gate that does nothing to the control qubit, and flips the value of the target qubit if the control qubit is $|1\rangle$. The target output of the CNOT gate corresponds to the result of a classical XOR gate. Furthermore, the CNOT gate is used for generating entanglement. Many two-qubit states cannot be completely described by the tensor product of the first qubit and the second qubit. We call such states entangled**.** A common application of the CNOT gate is to

maximally entangle two qubits in tensor product into two qubits described by the bell states, which can be written in terms of the bell basis.

$$\left|\Psi^{+}\right\rangle = \frac{1}{\sqrt{2}}\left(\left|01\right\rangle + \left|10\right\rangle\right) = \frac{1}{\sqrt{2}}\left(\left|+\right\rangle_A \otimes \left|+\right\rangle_B - \left|-\right\rangle_A \otimes \left|-\right\rangle_B\right)$$
$$\left|\Psi^{-}\right\rangle = \frac{1}{\sqrt{2}}\left(\left|01\right\rangle - \left|10\right\rangle\right) = \frac{1}{\sqrt{2}}\left(\left|+\right\rangle_A \otimes \left|-\right\rangle_B - \left|+\right\rangle_A \otimes \left|-\right\rangle_B\right)$$
$$\left|\Phi^{+}\right\rangle = \frac{1}{\sqrt{2}}\left(\left|00\right\rangle + \left|11\right\rangle\right) = \frac{1}{\sqrt{2}}\left(\left|+\right\rangle_A \otimes \left|+\right\rangle_B + \left|-\right\rangle_A \otimes \left|-\right\rangle_B\right)$$
$$\left|\Phi^{-}\right\rangle = \frac{1}{\sqrt{2}}\left(\left|00\right\rangle - \left|11\right\rangle\right) = \frac{1}{\sqrt{2}}\left(\left|+\right\rangle_A \otimes \left|-\right\rangle_B + \left|+\right\rangle_A \otimes \left|-\right\rangle_B\right)$$

We will illustrate one of the four cases now. Suppose we have the input state $\frac{1}{\sqrt{2}}\left(\left|1\right\rangle + \left|0\right\rangle\right) \otimes \left|0\right\rangle$, and we put the state through a CNOT gate, we now have the bell state $\frac{1}{\sqrt{2}}\left(\left|11\right\rangle + \left|00\right\rangle\right)$, which is maximally entangled.

Entanglement is a non-local property that allows a set of qubits to express higher correlation than is possible in classical systems. When several qubits are entangled, each particle can no longer be described independently of the other particles. For example, with entanglement, you have changed your solution set from {00, 01} to {00, 11}. And what if you want the two qubits to have different values? Then the CNOT gate changes {01, 11} to {01, 10}; notice you have now the two qubits taking on opposite values. In this example, in choosing if you want the first qubit to be 0 (as in {00, 01}) or the second qubit to be 1 (as in {01, 11}), you have just processed your information and greatly narrowed down the possibilities. We see that before the CNOT you have not made any mathematical statement about the two qubits. After the CNOT gate, you have made the

statement that they must be the same as in ({00, 11}), or that they are opposites as in ({01, 10}). There is now a relation between the first and second qubit—hence they are entangled. In effect, before the quantum gate, the individual qubits are in an undefined state, but afterwards they are correlated. This correlation between several qubits is the complete description of the state of the set of qubits; if we choose the same basis to measure both qubits and compare our results, the measurements will be correlated with a probability greater than 50%. We will be able to know the second qubit's value just from looking at the first qubit's value. In this manner we can logically relate many qubits and achieve computation. In the real world, the variables symbolize some physical characteristic of the system being simulated, and sometimes, the variables are going to depend on one another, or share a correlation.

Controlled Phase gates are two or multiple-qubit gates that are important in quantum information processing because they are useful for doing more complex computations that involve series. One example of their use is to implement the Quantum Fourier Transform (QFT), a critical component of many quantum algorithms. Phase gates are a family of single-qubit gates that leave the $|0\rangle$ state unaltered, and multiply a phase to the $|1\rangle$ state. As a part of the QFT, controlled phase gates are used for finding the period of a function, which allows us to do factoring. The phase gates are built out of the z-rotation gates, which allow us to rotate the qubit by arbitrarily small angles each time. This is how you can build the $Z^{-1/2^n}$ gate. Without the rotation gates, it would be impossible to build phase gates with *n* greater than one. So fortunately, the rotation gates can be implemented.

There are criteria for whether a Hamiltonian can be used to build a quantum gate. In quantum mechanics we have learned that Hamiltonians are Hermitian, and quantum gates are unitary. So why is there a discrepancy? Although there are two versions of the Schrodinger equation, the time-dependent and time-independent versions, we are interested in the time-dependent case ($H|\psi\rangle = i\eta\frac{\partial}{\partial t}|\psi\rangle$) for making our quantum gates. This means that when the quantum state passes through a Hamiltonian, which is a Hermitian matrix, it will evolve according to the following rule: $|\psi(t)\rangle = e^{-iHt/\eta}|\psi_0\rangle$. As you can easily derive for yourself $e^{-iHt/\eta}$ is a unitary matrix, and it depends on time, this is why quantum gates perform only unitary operations and are time-dependent; even though the quantum Hamiltonian itself is Hermitian and not unitary. As discussed later in the ion trap section, the laser is used to generate *H*, which, when acted on the quantum state, causes it to evolve in time according to the time-dependent Schrodinger equation until the desired gate action is achieved. The fact that quantum gates are time-dependent has the implication that after you have reached the desired result, you must turn off the quantum gate.

For more information about quantum circuits, please consult source (Nielsen & Chuang, 2010). The survey (Gharibian, Huang, Landau & Shin, 2014) contains information about the decomposition of single and two-qubit gates into rotation matrices—which represents what the quantum gate does.

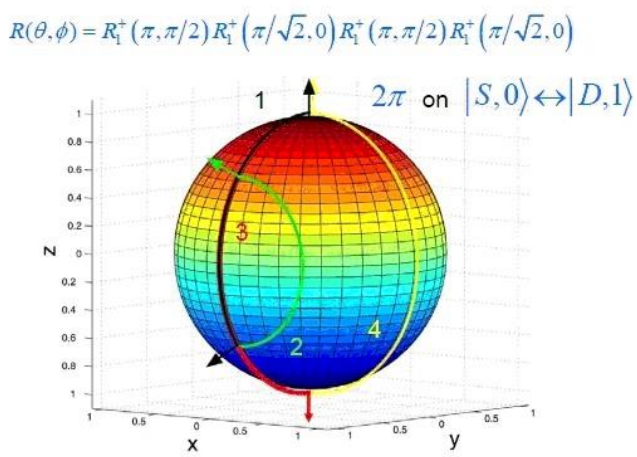

Figure 1. Quantum Gate

2.4 Computational Complexity Theory

Complexity theory is about classifying computational problems according to their inherent difficulty, and relating those classes to one another. A problem can be thought of as a binary relation *P* on a set *I* of instances and a set of solutions to those instances. A pair $(i,s) \in P$ says that *s* is a solution for problem *P* with input *i*. To keep things simple, we assume that instance *i* and solution *s* are encoded as binary strings.

Before quantifying how hard a problem is, we need to fix a computational model. Since the 1930s, many computational models have been proposed, out of which the most general have all been shown to be computationally equivalent. This is known as the Church Turing Thesis, a description of which can be found in (Moore & Mertens, 2011). We will focus on three models: Turing Machines, uniform circuit families and uniform quantum circuit families. A Turing Machine (TM) is a device that operates on a two-way infinite tape. The tape consists of a sequence of squares, each of which contains a single

symbol from a fixed finite alphabet together with a blank square symbol. A TM machine has a tape head which sits over a single tape square at a time. A TM does one-step computations by using the contents of the tape square currently under its tape head and the awareness of its current state which must come from a finite set of states. Using these two pieces of information, a transition function is performed to find a new symbol to write to the square and also whether the tape head should stay put or move left one square or right one square. If the TM enters a special state called the Halt state, then the contents of the tape from the square currently under the tape head to the first blank square is called the output. An instance of a problem can be input to a TM as a blank tape except for the squares to the right of the tape head on which the instance is encoded using the TM's alphabet which is assumed to contain at least 0 and 1. A TM is said to solve a problem *P* if whenever it is given an input instance of *P*, it can compute after some number of steps an output *s* such that $(i, s) \in P$. Since we assumed *P* was total, at least one *s* exists for every *i*. We use *T(n)* and *S(n)* to denote, respectively, the maximum number of steps used and maximum number of squares seen to compute a solution on inputs of length *n*. These are called the time and space complexity of one problem.

A second computational model is that of the uniform circuit family. A uniform circuit is a directed acyclic graph with the following properties: Nodes with in-degree 0 are called inputs and are labeled with Boolean variables, 0 or 1. Nodes of out-degree 0 are called outputs. Internal nodes are labeled with gate types. Without loss of generality (Moore & Mertens, 2011), we will assume the possible gate types are AND, OR and NOT. Fixing values for its inputs we can proceed from the inputs to the outputs by evaluating each gate at an internal node, each of whose inputs are known. The size of a

circuit is the number of gates it contains. A circuit family { $C_1, C_2, ...$ } is a set of circuits, such that C_i takes *i* as input variables. A circuit family is uniform if there is a TM which on input *i* written in binary operates for at most T(|i|) steps, where |i| is the length of *i* in binary, and outputs an encoding of C_i in binary. We say a circuit family solves a problem *P* if when given an instance *x* of the problem, if we take the circuit $C_{|x|}$ and evaluate it on input *x*, we get an *s* such that (*x*, *s*) is in *P*.

A decision problem *P* is a problem such that the only solution on an instance *i* of *P* is either the value 0 or the value 1. The class P, polynomial time, is the class of decision problems that can be solvable by TMs whose runtime *T(n)* are bounded by a polynomial. It turns out as stated in (Moore & Mertens, 2011), P could be alternatively have been defined as the class of decision problem solvable by a uniform circuit family { C_n } such that the size of the encoding of C_n in binary is bounded by a polynomial.

We could even restrict the C_n a bit more—to make the circuits leveled. A circuit is leveled if all paths from an input to a certain gate are the same length. If one has a leveled circuit then we define the level of a gate to be the path length to its inputs. For a leveled circuit, the level *i+1* gates only depend on the level *i* gates and the mapping between these layers could be computed as an application of a unitary matrix. This motivates our last model of computation, the uniform quantum circuit family.

A quantum circuit is a unitary matrix defined as a product of layers where each layer is either a permutation matrix or a tensor product of CNOT, Hadamard gates, and Z gates. A quantum circuit family{ C_n , M_n } is a set of quantum circuits C_n such that C_n operates on *m>n* qubits—*n* input qubits, *m-n* ancillary qubits, and there is some measurement M_n

on the *m* qubits. The family is uniform if there is a polynomial time |n| -bounded TM which can output encodings of the gates used by each of its layers for both C_n and M_n. A uniform quantum circuit family solves a problem *P* if when *x* is an instance of the problem, then if one takes the quantum circuit $C_{|x|}$ and applies it to $|x\rangle|0\rangle^{m-n}$ and then takes the measurement; the measure output *s* and *x* satisfy $(x,s) \in P$ with probability at least 3/4. The class BQP is the class of decision problems solvable by uniform quantum circuit families such that the number *m* for each *n* is bounded by a polynomial and the number of layers in C_n is bounded by a polynomial in *n*.

It is unknown if P=BQP. As factorization can be solved using the Shor's algorithm which is built into polynomial sized quantum circuit families, but this algorithm is not polynomial, and no polynomial TM algorithm for the problem is known, thus it is suspected that $P \neq BQP$. In the study of computation complexity another important classical complexity class is NP. It consists of those decision problems *P* that are verifiable by a polynomially time bounded TM, *M*, which operates on encodings of ordered pairs, a polynomial *q(n)*, and we have $(x, 1) \in P$ iff $\exists_y$, $|y| \leq q(|x|)$ and *M* on *(x,y)* outputs 1.

It is a famous open problem whether P=NP; a $1,000,000 prize is offered by the Clay Math Institute for its solution. *k*-SAT is known to be in NP: $(\varphi, 1) \in k$-SAT iff φ has at least one satisfying assignment υ. In polynomial time given φ and υ, we can check whether υ satisfies φ by evaluating each clause of φ according to υ.

Given two decision problems P_1 and P_2, we say P_1 polynomial time reduces to P_2 if there is a polynomial time bounded TM which computes a P function f on its inputs such that $(x,1) \in P_1$ iff $(f(x),1) \in P_2$.

Given a class of decision problems *C* and a problem *P*, we say *P* is complete for *C* if $P \in C$ and given any $P' \in C$ we have P' is polynomial time reducible to *P*. It is known *k*-SAT is NP-complete (Moore & Mertens, 2011).

We complete this section by introducing two more complexity classes, BPP and QMA. BPP is used to capture problems whose algorithms might involve access to random coin tosses. A decision problem P is in BPP if there is a TM, *M*, and a polynomial *q*, such that $(x,1) \in P$ iff for at least ¾ of strings with $|y| \leq q(|x|)$ and when (x,y) is operated on by *M* outputs 1. It is known $P \subseteq BPP \subseteq BQP$, but it is unknown if either containment is strict. It is also unknown the relationship of either BPP or BQP to NP. The class QMA is defined to be the class of decision problems *P* such that there is a polynomial sized uniform quantum circuit family $\{C_n, M_n\}$ and a polynomial *q*, and we have $(x,1) \in P$ iff $\exists_y$ $|y| \leq q(|x|)$ and where $C_{|x|+|y|}, M_{|x|+|y|}$ outputs 1 with probability at least ¾ on input *(x,y)*.

It is known $BQP \subseteq QMA$ and $NP \subseteq QMA$, but it is not known if either containment is strict, or whether the given two classes, BQP and NP, are equal. The decision problem, "Given an encoding of the matrices of a *k*-local Hamiltonian and an energy value *v* does this Hamiltonian have an eigenenergy less than *v*?" is known to be QMA-complete, even for *k*=2 (Gharibian, Huang, Landau & Shin, 2014). A special case of the *k*-local Hamiltonian problem is whether a *k*-QSAT instance has a 0-energy solution. This too is known to be QMA complete for $k \geq 3$ (Gharibian, Huang, Landau & Shin, 2014).

2.5 Quantum Information Processing

In this section we consider some of the advantages of quantum over classical computation. The three mechanisms that allow a quantum computer to process information more efficiently than a classical computer are quantum parallelism, entanglement and measurement. Quantum parallelism is achieved through a Hadamard gate, and one way to achieve entanglement is through the CNOT gate. Quantum parallelism gives you a large set of possibilities and allows simultaneous evaluation of a function for a large number of inputs, while entanglement deletes most of the possibilities, so that you are left with only the solutions. Quantum parallelism is mathematically relevant to our topic because it allows all the values that can be denoted by the q qubits to become initialized to equal probability, which is the best way to start out our computation. Everything that comes after this point will alter the probability of all the possible states, amplifying some and diminishing others, until we arrive at the correct value with the highest probability. Both quantum parallelism and entanglement are used in the Shor's algorithm. Because of quantum parallelism and entanglement, we are able to solve problems much faster because we can accomplish in one step what we are used to accomplishing in N steps. Problems that take a million years to solve on a classical computer only take one hour to solve on a quantum computer.

The other way to generate entanglement other than through the CNOT gate is through the process called measurement. Let us illustrate through an example. Suppose we want to do a function evaluation using two registers: Register 1 for x, and Register 2 for $f(x)$. For example, function $f(x)$ has a period of four, and register 1 has 7 qubits. We start off with a quantum parallelism or equal superposition of when x=0, 1, 2, 3…127. So that we

have the state: $\frac{1}{\sqrt{128}}\left(|0, f(0)\rangle + |1, f(1)\rangle + |2, f(2)\rangle ... + |127, f(127)\rangle\right)$. If we perform a measurement on the second register and we obtain a single value *m*, in doing this we have deleted all options except the ones for x=0,4,8,12,16…124, which are the values of x for which f(x) = m. Now we have:

$\frac{1}{\sqrt{32}}\left(|0, f(0)\rangle + |4, f(4)\rangle + |8, f(8)\rangle ... + |124, f(124)\rangle\right)$. In the case of quantum parallelism, we had a tensor product of the seven qubits in the first register, but after measurement we have deleted all entries that are not multiples of 4. Now entanglement exists in the first register, that is, the first register can no longer be written as a tensor product of its constituent single qubits anymore.

A restriction on measurements is that you can measure along only one axis at a time. The result of the measurement will either be an up along that axis or a down along the axis. After the measurement the quantum state will lose its probabilistic characteristic and cannot be used again. On the one hand, this is bad because you might need more than one copy of the qubit during computation, but on the other hand this is good because this means measurement is a powerful tool that can be used in information processing, in which we narrow down the states that will be useful to us for solving our problem. In measurement, you collapse the value of the wave function of your qubit register to just a single value. But measurement is a random process, and you never know exactly which one in a set of values you will collapse your wave function to. So this is done with the assumption that all values lead to the same result, finding the period of the function is one example of such applications. The Shor's algorithm itself is a period finding algorithm, and it is based on the idea that all measurements lead to the same period and are therefore

equally welcomed. After measuring register 1 of the Shor's algorithm in the final step of computation, we obtain a random integer multiple of *Q/r* which means we still have some data processing to do before we can actually extract *r*. The quantum computer cannot operate alone—it needs a classical computer for processing the solutions.

Measurement appears to be one way we can change the state of a qubit other than passing it through a quantum gate. For a quantum programmer who wants to adjust the qubits in a quantum computer, however, this may not be a good choice. After all, the results are random! Although some quantum algorithms, such as the Shor's algorithm, use measurement in the middle of a computation, most of the time measurement is reserved for the end, when the programmer learns the *final result* of the computation.

2.6 Shor's Algorithm

To factor a number *N* we need *q* qubits; we find the number of qubits in register 1, *q*, by finding a *q* such that $N^2 \leq 2^q < 2N^2$. In the Figure 2 below, we have q=5, this means the number we are factoring is actually only N=5; as you can see this is a trivial example, but from Figure 2, it is possible to infer what a larger circuit looks like. We find the number of qubits in register 2 via the equation $n = \lceil \log_2(N) \rceil$. Thus, in the above example we have q=5, N=5, n=3. To anyone interested in quantum computing, one of the most interesting quantum algorithms is the Shor's algorithm, which can solve factoring in polynomial time. Many cryptography systems on the web rely on factoring being hard, so if quantum computers can be realized, this algorithm will impact how information is secured. The Shor's Algorithm is an example of a quantum Hamiltonian. Below is an outline of the Shor's Algorithm (Wikipedia, 2017):

1. Pick a random number $a < N$.

2. Compute $\gcd(a, N)$. This may be done using the Euclidean algorithm.

3. If $\gcd(a, N) \neq 1$, then this number is a nontrivial factor of *N*, so we are done.

4. Otherwise use the period finding subroutine to find the period *r* of function $f(x) = a^x \bmod N$. We note that the function $f(x)$ is a periodic function of period *r*. Thus, if given $f(0) = a^0 \bmod N = 1$, we can deduce $f(r) = a^r \bmod N = 1$, $f(2r) = a^{2r} \bmod N = 1$ and so on. For any integer multiple of *r*, this means, if $a^r \bmod N = 1$, then :

$(a^{r/2})^2 \bmod N = 1$

$[(a^{r/2})^2 - 1] \bmod N = 0$

$[a^{r/2} - 1][a^{r/2} + 1] \bmod N = 0$

That is, at least one of $a^{r/2} - 1$ and $a^{r/2} + 1$ is a multiple of *N*.

5. If *r* is odd, go back to step 1.

6. If $a^{r/2} \bmod N = -1$ go back to step 1.

7. $\gcd(a^{r/2} - 1, N)$ and $\gcd(a^{r/2} + 1, N)$ are both nontrivial factors of *N*. We are done.

For example: $N = 15$, $a = 7$, $r = 4$, $\gcd(7^2 \pm 1, 15) = \gcd(49 \pm 1, 15)$) where $\gcd(48, 15) = 3$ and $\gcd(50, 15) = 5$. We will use this example in our discussion of the Shor's Algorithm.

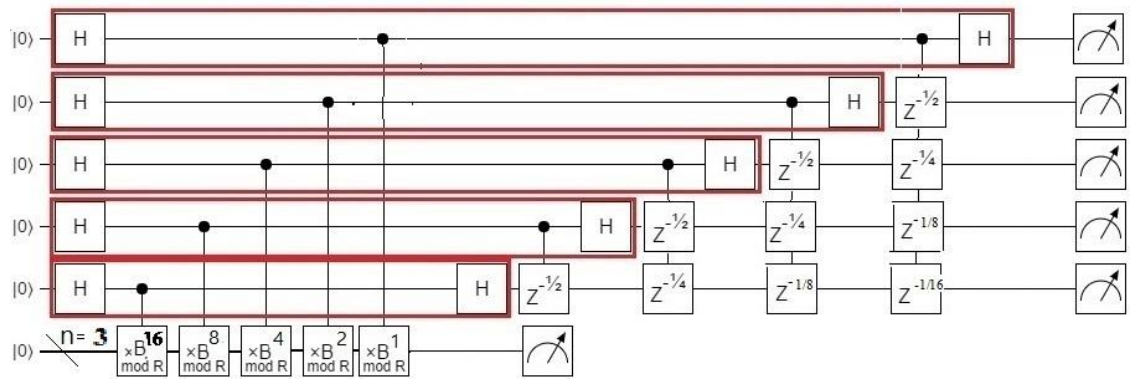

Figure 2. Shor's Circuit

Classical algorithms such as Newton's method are guaranteed to converge, unlike quantum algorithms. For the quantum algorithm, there is high chance of the algorithm converging, but it does not happen for certain. The idea is that you keep amplifying the probability by repeating the computation, allowing you to finally reach the ground state. Each time you run the algorithm is independent of the time before, but you have made adjustments to your boundary condition so that there is a smaller set of solutions for you to look at, thus enhancing the probability of achieving the correct solution.

The Shor algorithm circuit is used to factor numbers into their prime factorization. This circuit consists of two registers, the first register is called the period register, and the second register is called the computational register. To accomplish computation, we need *q* qubits in a first register and *n* qubits in a second register. *q* qubits can be used to express numbers in the range 0 to 2^q-1. We call the number 2^q as *Q*. To factor *N*, in the first register, we need a *Q* value that is in the range $N^2 \le Q < 2N^2$. We also need a *n* value, for the second register, that is equal to $n = \lceil \log_2 N \rceil$. The function $f(x) = a^x \bmod N$ is periodic and it is contained in the second register. This means if $f(r) = a^r \bmod N = 1$ then $f(kr) = a^{kr} \bmod N = 1$ for *k* equal to integers. The largest a

period r can be is N, since the function M mod N will always equal zero when $M=kN$, meaning N is the largest period for any modular function, including the modular exponentiation function defined here denoted as $f(x)$. Which leads us to why we pick $N^2 \leq Q < 2N^2$--so that we have at least N to $2N$ periods to work with when we perform our period finding subroutine; we definitely need more than one period of data for the purpose of finding the period. In fact, when we apply Shor's algorithm to encryption-breaking or factoring problems, this assumption will automatically be satisfied since $r << N$.

Register 1 must have enough qubits q to represent integers as large as $Q-1$. Register 2 must have enough qubits n to represent integers as large as $N-1$. Load register 1 with all zeros, and register 2 with all zeros as well, and we are ready to perform a computation. The Hadamard gates create equally weighted superpositions of all states represented by integers from 0 to $Q-1$.

For our next example with q=7, how the computational register works is as follows. First it takes the highest digit (the digit representing the 2^6 digit), and finds the modulus N of this digit. Let us call this number a_1. Then it multiplies the number a_1 by the next highest digit (the digit representing the 2^5 digit) and finds the modulus N of this number, Let us call the result a_2. Then we multiply a_2 by the next highest digit (the digit representing the 2^4 digit), and so on until we get to the lowest digit. By the end, it has calculated the function $f(x) = a^x \bmod N$, where x is a number between 0 and 127, and a is, for the example here, the number 7. In this particular example $a = q$ but in general, they can be different.

In our example we have the following function values:

$$7^0 \bmod 15 = 1 \qquad 7^4 \bmod 15 = 1$$
$$7^1 \bmod 15 = 7 \qquad 7^8 \bmod 15 = 1$$
$$7^2 \bmod 15 = 4 \qquad 7^{16} \bmod 15 = 1$$
$$7^3 \bmod 15 = 13 \qquad 7^{32} \bmod 15 = 1$$
$$7^{64} \bmod 15 = 1$$

To help illustrate this example, we go through the following procedure when we want to find the function $f(23) = 7^{23} \bmod 15 = 7^{16+0+4+2+1} \bmod 15$:

$$7^{16} \bmod 15 = 1$$
$$1 \bullet 7^0 \bmod 15 = 1$$
$$1 \bullet 7^4 \bmod 15 = 1$$
$$1 \bullet 7^2 \bmod 15 = 4$$
$$4 \bullet 7^1 \bmod 15 = 28 \bmod 15 = 13$$

As can be seen, when the computational register is measured, the result will be one of the four values: 7, 4, 13 and 1. So after measurement, the wave function is collapsed to one of these four options. However, no matter which option results, the period *r* will be 4.

At this point, the computational register is $\frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} \left| x, 7^x \bmod 15 \right\rangle$. Measuring the computational register, gives a value $7^x \bmod 15 = k$, and the state is:

$$\frac{1}{\sqrt{\| A \|}} \sum_{f(x')=k, x' \in A} \left| x', k \right\rangle$$

So all the states $|x, y\rangle$ that do not have $y=k$ from the superposition will be deleted.

The next part of Shor's algorithm to understand is the Inverse Quantum Fourier Transform. Initially, the binary number $j = j_1 j_2 j_3 .. j_n$, where n denotes the lowest bit and j is represented by the quantum state $|j_1 ..... j_n\rangle$. Applying the Hadamard gate, which is also called the R_1 gate, to the first qubit produces the state:

$$\frac{1}{2^{1/2}}\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right) \otimes |j_2 ..... j_n\rangle .$$

After the entire quantum Fourier transform algorithm, we have:

$$\frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle\right) \otimes ... \otimes \left(|0\rangle + e^{2\pi i 0.j_1 j_2 ... j_n}|1\rangle\right)}{2^{n/2}}$$

In the above equation, the lowest qubit is the left most qubit. The next term from the right is the second highest qubit, and so on. The lowest qubit will be passed through a Hadamard gate. The next will passed through a Hadamard gate and a R_2 gate. The next lowest qubit will be passed through Hadamard, R_2 and R_4. The next lowest qubit will be passed through Hadamard, R_2, R_4 and R_8. And so on. All the rotation gates R are controlled gates.

After the discrete Inverse Quantum Fourier Transform on register 1 has been computed, if a measurement is performed, the result of this measurement has a very high probability of being a multiple of Q/r, $m = a \bullet Q/r$, where r is the desired period. The

last step is to take the value measurement *m*, and on a classical computer do some processing to calculate *r* based on the knowledge of *m* and *Q*. Unfortunately there is no known way to access all the amplitudes after the Inverse Quantum Fourier Transform from a quantum computer by measurement, but with one measurement, the correct solution can be obtained with high probability. The Inverse Quantum Fourier Transform is the key to a general procedure known as phase estimation, which in turn is the key for Shor's algorithm. It is obtained by reversing the circuit for the QFT. This provides a pretty good estimation of the phase or period *r*. Even though the wave function has all the phase for all the qubits and all the probabilities, you cannot extract them from the computer, because in measurement the quantum state is collapsed. After the Inverse Quantum Fourier Transform we have:

$$m = \frac{1}{\sqrt{\| A \|}} \sum_{x' \in A} \frac{1}{\sqrt{Q}} \sum_{y=0}^{Q-1} \left| y, k \right\rangle * e^{2\pi i x' y / Q}$$

Each term in the summation contains $e^{2\pi i a}$ where $a = \frac{x' y}{Q}$. *Q* is the period of the function after the Inverse Quantum Fourier Transform, *x'* is the independent variable of the function and *y* is the index of summation running from 0 to *Q-1*. $e^{2\pi i a}$ is largest when *a* is an integer, and *a* is most likely to be an integer when *x'=r*. Thus, when we measure the period register at the end of the Shor's algorithm, we get a value for *y* such that $y = m = a \bullet \left(\frac{Q}{r} \right)$, where *a* is an integer. This is because by inspection of the

probability formula we extract from the coefficient of our final quantum state we see that that the probability is greatest when *yr/Q* is an integer *a*.

2.7 Grover's Algorithm

To introduce Grover's algorithm, we can describe it as a database search algorithm that requires a "quantum oracle" operator which can recognize solutions to the problems and give them a negative sign. Shor's algorithm provides an example of the power of quantum computing, but Grover's Algorithm shows a way in which a quantum computer can solve problems more quickly than a classical algorithm but less quickly than the Shor's algorithm. The Grover's algorithm can be described as finding a needle in a haystack in just $O(\sqrt{N})$ queries. The Grover's algorithm makes use of another example of a quantum Hamiltonian. Below is an outline of the Grover's Algorithm (Wikipedia, 2017):

1. Initialize the system to the state $|s\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle$.
2. Perform the Grover iteration r(N) times, The function r(N) is described below
   1. Apply the Oracle operator U_ω.
   2. Apply the Diffusion operator U_s.
3. Perform the measurement. The measurement result will be eigenvalue λ_ω with probability approaching 1 for N>>1. From λ_ω, $|\omega\rangle$ may be obtained.

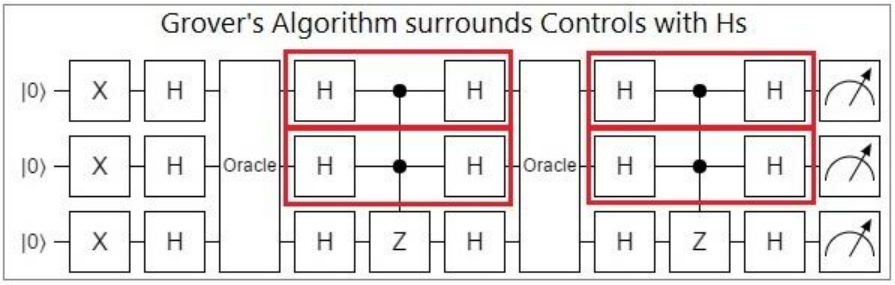

Figure 2. Grover's Circuit

To describe the algorithm in detail, it is convenient to start by describing two vectors $|u\rangle$ and $|v\rangle$, which are composed of quantum states within the haystack. $|u\rangle$ is the equal superposition of all possible states in the haystack, $|j\rangle$; whereas, $|v\rangle$ is the equal superposition of all states in the haystack, except the solution state, $|i\rangle$. Then as described on the previous page, there are two operators that operates on the qubits, the Oracle and Diffusion operators, or U_ω and U_s. The Diffusion operator reflects the quantum state $|\psi\rangle$ about the vector $|u\rangle$, while the Oracle operator reflects the quantum state $|\psi\rangle$ about the vector $|v\rangle$. Another way to describe what the Oracle operator does is that it multiplies a constant of -1 to the solution state $|i\rangle$ which is a component of $|\psi\rangle$, and leaves the other components of $|\psi\rangle$ unaltered. By successively multiplying the quantum state by the Oracle then Diffusion operators, which is equivalent to rotating about the fixed $|v\rangle$ then $|u\rangle$ for an angle of π, we cause the quantum state $|\psi\rangle$ to gradually approach the state that is perpendicular to $|v\rangle$, also known as the needle's location, $|i\rangle$ (Moore, Mertens, 2011).

At the beginning of the computation, we start with $|\psi\rangle = |u\rangle$, which means the density matrix of the haystack is $\rho = |u\rangle\langle u|$, with $|u\rangle = \sum_{j=1}^{N} |j\rangle$; at the end of the computation the haystack is $\rho = \hat{I} / N$. What happened in between is the process, called decoherence, which removed all quantum phase information until only classical probability remains. In other words, as entanglement increased, all off diagonal elements of the density matrix became zero, leaving only the diagonal entries nonzero. A simple measurement of entanglement *S* is the sum of all the entries in the density matrix. We saw that at the beginning of the algorithm, *S*=*N*, while at the end, *S*=1; giving us a change of *N*-1 in *S*. It can be derived mathematically that each Oracle operator changes *S* by a maximum of $2\sqrt{N}$. Thus for very large values of *N*, the number of steps the Grover's algorithm takes to search a data base is approximately $t = \sqrt{N}/2$. To make this value more precise, how many iterations of Oracle and Diffusion operators we need is calculated via the formula $t = \frac{\pi}{4}\sqrt{N}$. In fact, it was shown by in source (Moore, Mertens, 2011) that no other operator allows us to find the needle more quickly than the Oracle operator, since the Oracle operator imposes the largest possible phase shift or an angle of π, thus we say: the Grover's algorithm is optimal.

Grover's algorithm allows us to find, with high probability, the unique input to a black box function that produces a particular output value. The Grover's algorithm we discussed above solves a problem in $\sqrt{N}$ steps whereas a classical algorithm cannot solve the problem in *N* steps or fewer. It was shown later (Nielsen & Chuang, 2010) that some variants of the Grover's algorithm can search an *N*-item database at most in $\sqrt[3]{N}$ steps.

Neither search method will allow quantum computers to solve NP-Complete problems in polynomial time, but Grover's algorithm does provide a quadratic or cubic speedup. Even a quadratic speedup is considerable when N is large. If we are given $y=f(x)$, Grover's algorithm allows us to calculate x when given y. Grover's algorithm can also be used for estimating the mean and median of a set of numbers, and for solving the collision problem. A modification of Grover's algorithm called quantum partial search is one in which we are not interested in finding the exact address of the target item, instead only the first few digits of the address are of interest. As mentioned earlier, it is known that Grover's algorithm is optimal. That is, any algorithm that accesses the database only by using the operator U_ω must apply U_ω at least as many times as Grover's Algorithm. Reading a full database item by item and converting it into such a representation may take a lot longer than Grover's search. To account for such effects, Grover's algorithm can be viewed as solving an equation or satisfying a constraint. In such applications, the oracle is a way to check the constraint and is not related to the search algorithm. This separation of checking and finding usually prevents algorithmic optimizations, whereas conventional search algorithms often rely on such optimizations and avoid an exhaustive search.

2.8 Ion trap

We conclude our preliminary discussion of quantum computation, k-local Hamiltonians and complexity theory with a brief introduction to research being conducted to physically implement a quantum computer. Although this topic is not strictly necessary to understanding the rest of this paper, it is useful to know how the quantum techniques discussed in this paper may be physically implemented. Right now,

there are three or four different types of quantum computers being researched, including the ion trap, the quantum annealer and the Josephson junction. The most promising model is the ion trap. To understand ion traps, recall that quantum Hamiltonians are constructed from physical observables. In the ion trap, we manipulate vibrational and electronic states with lasers to achieve computation. Electrons exist in orbitals. Orbitals describe where you are most likely to find an electron with an energy given in the energy level shown in your atomic energy level diagram. Orbitals have fixed characteristic energies that can be described abstractly as "energy levels" on an atomic energy level diagram. When photons of frequency that matches the gap in energy levels shine on the atom or ion, an electron is promoted into the upper energy level, an orbital with a higher energy; the electron can then drop back with a non-zero probability to the lower energy level, emitting fluorescence. These toggling of states are used by an ion trap computer to perform quantum computations.

The ion trap is currently the best model for quantum computers because of its negligible loss rate of the ions themselves, and of the coherence of the stored quantum information. Trapping the ions with electromagnetic forces in a vacuum isolates them almost perfectly from their environment and thus permits extremely long storage times of the fragile quantum information. Ion traps are also popular because the states can be initialized and measured with high accuracy. Lasers are used to initialize, do computation, and finally measure the states of the qubits. The Hamiltonian for a single trapped ion has both a simple mechanical oscillator component and an internal energy component; so that the energy spectrums involve the coupling of photonic and mechanical effects.

Designing the gates for the ion trap involves designing the lasers' effect on the ions and finding the perfect combination of mechanical movement and electronic energy transition of the Hamiltonian, to make the needed rotation of the quantum state in a certain amount of time. This Hamiltonian is basically what the laser does, in other words it is our quantum gate, and will "rotate" our qubit by any angle we wish. The quotes on "rotate" are because, it is not a physical spin that we are rotating, but a conceptual spin that is represented by two energy levels in an atom.

In the ion trap setting, both initialization and measurement can be implemented with a very high accuracy. Single and two qubit gates are implemented via one photon or two photon transitions. In choosing atoms to use for our qubits, the following requirements need to be met: 1) The electronic level structure should be simple to allow the realization of a closed two level system without the need for too many lasers. 2) The levels used for the qubit transition should have a negligible spontaneous decay. 3) The levels should allow for efficient laser cooling and detection. Most of the experiments have been done with: Be9+, Ca40+, Ba138+, Mg25+, Hg199+, and Yb171+. We will use the Ca40+ ion in our discussion.

The ions used in an ion trap are all positive ions, which means they repel one another. The only way to get them close together so that they are coupled is by freezing. As the temperature is lowered, the charge due to the nucleus becomes very strong and induces a dipole on a nearby ion of the same type, so that bounds are formed. For Ca40+, this freezing point is actually much lower than room temperature. With laser cooling, temperatures that are far beyond reach of cryostats can be realized. Because the laser freezes the ions, we end up with what is called the "icy model," which basically means

the ions are coupled, and the coupling coefficient J of the model allows every ion to talk to every other ion. Through freezing, 30 to 40 qubits can be coupled so as one can do quantum computation with them. The reason why one currently can only do about 20 qubits for quantum computation rather than 100 is that our ability to make a vacuum is limited and the trapped ions collide with the molecules of the residue gas, causing the crystal to melt in about one minute, so the system needs to be re-cooled, which takes about 1 or 2 minutes to recover. So far the longest they can keep the crystal frozen by lowering residue gas pressure is for a month.

Besides freezing, another problem encountered is that when the number of ions in the trap increases, it gets more and more difficult to kick the ion string with a single photon (or in the Raman approach, two photons). In the mathematical description, the Lamb-Dick parameter gets smaller and slows down the operations on the sideband, causing further problems to arise due to the more complex normal mode spectrum, and decreasing ion-ion spacing with increasing ion number. Mathematically, these restrictions do not change exponentially with the number of qubits, but they do prohibit scaling to large numbers of ions for practical reasons. There are at least five ideas regarding how to overcome these problems: 1) Split up the ion string in small portions and move the ions around. 2) Couple the ions via cavities and photons. 3) Prepare entanglement via joint florescence photon detection and use this entanglement as a resource to teleport information between two traps. 4) Wire up ion traps, and use the image charges induced by the ion motions to couple the ions in different traps. 5) Use the radial modes of the ion string. One of the problems in the cavity QED-systems introduced was the unwanted occurrence of flying qubits. In order to solve this problem there are now attempts to use

stored ions in combination with optical cavities. Researchers have now proved that they can place a single trapped Ca40+ ion at will inside an optical resonator (Haffner, 2008).

An ion trap can be built using a Ca40+ ion in which two such orbitals or energy levels are used to make the quantum states $|0\rangle$ and $|1\rangle$. When you shine laser light onto the Ca40+ ion, the electron starts in two possible energy levels, and only one of these states glows—this is how we can detect whether the atom is in state $|0\rangle$ or $|1\rangle$. The upper energy level that corresponds to the state $|1\rangle$ in the Ca40+ trap is the P(1/2) level; and is the state that glows. The lower energy level corresponding to the state $|0\rangle$ in the Ca40+ trap is the D(5/2) state, and it is the state that remains dark.

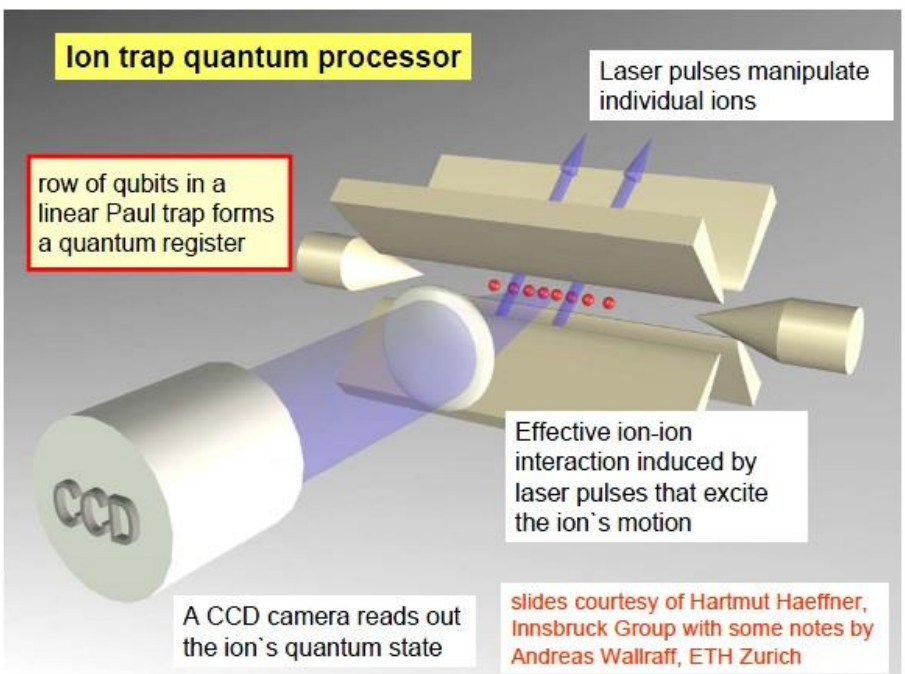

Figure 3. Ion Trap

The Ca40+ ion fluoresces when radiated on the S(1/2) to P(1/2) transition only when it is projected into the P(1/2) state. To avoid pumping into the metatstable D(3/2) level, one uses additional light on the D(3/2) to P(1/2) transition, so that an electron in D(3/2) actually ends up in P(1/2). If the ion was projected into the D(5/2) level, it remains dark. The absence of photon detection events signals projection onto the D(5/2) level. Knowing the arrival times of the photons allows one to take into account the decay of the D(5/2) level. In a single-qubit operation, the electric field of the laser builds up a quadrupole moment that is oscillating in phase at the laser frequencies corresponding to P(1/2) and D(5/2) to S(1/2) energy gaps (Haeffner, 2008). When all the population is transferred to the excited D(5/2) level, the phase reference is lost and changing the phase of the excitation field has no effect.

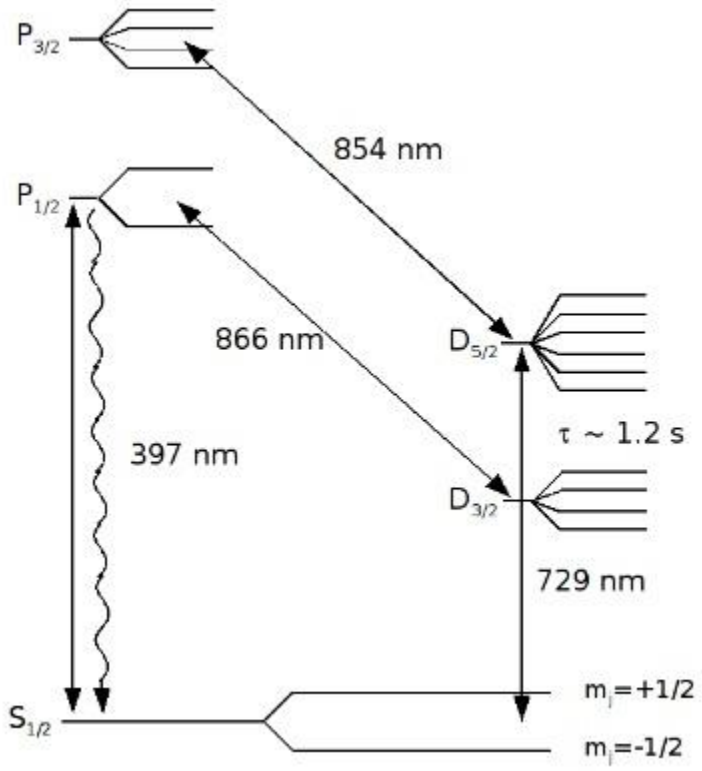

Figure 4. Energy Level Diagram

The lifetime of the metastable D-levels, that is both D(5/2) and D(3/2), are on the order of a second, thus allowing for long coherence times of the qubit. For sideband cooling, the ion is irradiated with S(1/2) to D(5/2) transition. The lifetime of the metastable D(3/2) level is artificially shortened by a laser coupling the D(5/2) to P(3/2) from 1.2s to 7 ns. In order to suppress heating during the excitation on the D(5/2) to P(3/2) transition and the subsequent decay process, it is advantageous that the ion string is already in the Lamb-Dicke limit.

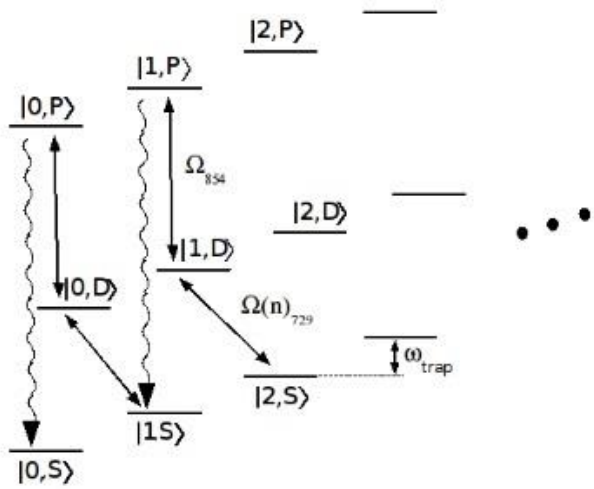

Figure 5. Laser-Ion Interaction

$$H = \hbar\Omega\sigma_{+}e^{-i(\Delta t-\varphi)}\exp\left(i\eta\left[ae^{-i\omega_t t} + a^{+}e^{i\omega_t t}\right]\right) + h.c.$$

Above is the quantum Hamiltonian for a single trapped ion interacting with near resonant laser light. Below is the Lamb-Dicke approximation for this quantum Hamiltonian.

$$H = \hbar\Omega\left\{\sigma_{+}e^{-i(\Delta t-\varphi)} + \sigma_{-}e^{i(\Delta t-\varphi)} + i\eta\left(\sigma_{+}e^{-i(\Delta t-\varphi)} - \sigma_{-}e^{i(\Delta t-\varphi)}\right)\left(ae^{-i\omega_t t} + a^{+}e^{i\omega_t t}\right)\right\}$$

When we use the Lamb-Dicke approximation, we can rewrite the Hamiltonian by expanding the exponential. The Lamb-Dicke approximation is good for when there is suppressed heating of the ion crystal.

Three cases of laser detuning delta are of particular interest:

$$\Delta = 0 : H_{car} = \hbar\Omega\left(\sigma_{+}e^{i\varphi} + \sigma_{-}e^{-i\varphi}\right)$$

$$\Delta = \omega_t : H_{+} = i\hbar\Omega\eta\left(\sigma_{+}a^{+}e^{i\varphi} - \sigma_{-}ae^{-i\varphi}\right)$$

$$\Delta = -\omega_t : H_{+} = i\hbar\Omega\eta\left(\sigma_{-}a^{+}e^{-i\varphi} - \sigma_{+}ae^{i\varphi}\right)$$

Using the above Hamiltonian for detuning with Δ=0, we directly find single qubit operations to be:

$$R(\theta,\phi) = \eta\Omega\left(i\theta / 2(\sigma_{+}e^{i\varphi} + \sigma_{-}e^{-i\varphi})\right)$$

The angle φ specifies the axis of rotation in the equatorial plane, and θ is the size of the rotation (Haeffner, 2008).

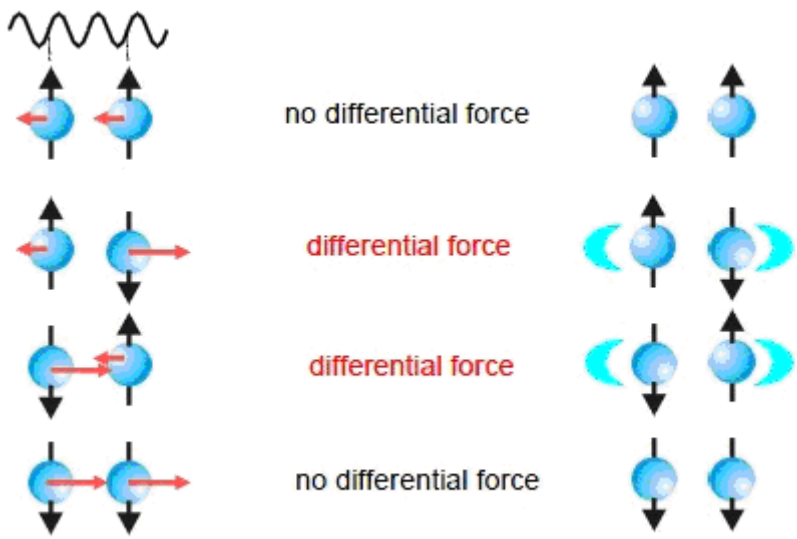

Figure 6. Stretch Mode Excitation

The single-qubit gates are implemented by the rotation gates described above. We now describe three implementations of two-qubit gates that have been realized for the ion trap. The first of these is the Cirac-Zoller gate, which is a controlled phase flip gate, and it can be adapted to perform CNOT operations. This two-qubit gate uses the motion of the ion crystal to couple ions to each other. For *N* ions, there are *3N* motional modes. Typically only one of these modes is used as the "quantum bus," and the other modes are all spectators. The operation of this gate is as follows. First use laser to flip the target ion based on the information of the bus mode, which acts as the control bit. Finally the

bus mode and control ions are reset to their initial value. We now describe the problem of performing a controlled phase flip gate on a single ion with the motional bus mode as the control bit. A controlled phase gate flip can be constructed with the help of an auxiliary atomic level. We first assume one qubit is stored in an ion internal state (0 or 1) another qubit is stored in the phonon state, or the quantum bus (also 0 or 1). Both qubits can be in any arbitrary superposition so we have: $|11\rangle$, $|10\rangle$, $|01\rangle$, $|00\rangle$. A laser is tuned to the frequency of the sum of the auxiliary and phonon levels, $\omega_{aux} + \omega_z$, to cause the transition between the auxiliary state $|2,0\rangle$ (out of $|2,0\rangle$ and $|2,1\rangle$)and only the state, $|1,1\rangle$. Because of the uniqueness of the tuned frequency, there are no other transitions. The phase and duration of the pulse is chosen to make a 2π pulse so that the result is a negative sign multiplied to the $|11\rangle$ state. In order to decode both qubits in ions, a swap gate is required which maps an internal qubit state to a phonon qubit state. This can be done by tuning the laser to the frequency $\omega_0 - \omega_z$, and arranging the phase and pulse duration such that a π-pulse is established.

Another two-qubit gate that has been implemented is the geometric phase gate. Two laser beams with different frequencies are used. This difference in frequency is a necessary design parameter for our purpose. During the gate operation, both ions are illuminated simultaneously. Also the ion-ion distance is adjusted to a multiple of optical lattice constant. A force oscillating with the frequency difference between the two laser beams act on each ion. If the two ions are in different electronic states, the wave function picks up a phase— it might acquire a phase of $\pi/2$ in cases where the two ions are in different electronic states, or no phase in case where the ions are in the same electronic

state. This modulation allows us to do interesting things like multiplying a sinusoidal phase to the middle two terms of our two-qubit quantum state. Then we can use a swap gate to further process our information. The magnitude of the phase in our phase gate depends on light intensity. Finally, due to detuning, the ion string returns to the original motional state after a period of time.

The last two-qubit gate we will describe is the Molmer-Sorensen gate, which is a gate that is somewhat related to the geometric phase gate. The main idea is to drive the collective spin flips of the involved ions using two laser fields, one tuned to the red sideband, the other tuned to the blue sideband. We drive the two-photon transitions on $|gg,n\rangle \leftrightarrow |ee,n\rangle$ as well as $|ge,n\rangle \leftrightarrow |eg,n\rangle$; stopping halfway entangles the two ions in the states $\frac{1}{\sqrt{2}}\left[|gg,n\rangle + i|ee,n\rangle\right]$ or $\frac{1}{\sqrt{2}}\left[|ge,n\rangle + i|eg,n\rangle\right]$. As a last remark, this gate is universal, but it is very difficult to keep the lasers interferometrically stable for use. A detailed discussion of the Ca40+ ion trap is available in (Haeffner, 2008). Compared to single-qubit gates, 2-qubit gates take longer runtime and have higher error rates.

Single-qubit operations are achieved through Rabi oscillations. Two-qubit operations are achieved through Cirac-Zoller gates, geometric phase gates and Molmer-Sorensen gates. Qubit measurement is achieved through individual ion fluorescence. We use flying qubits to achieve two-qubit operations. Converting qubits to flying qubits is done through the CQED bad cavity limit. Faithfully transmitting desired flying qubits is achieved through coupling pulse sequences (Haeffner, 2008).

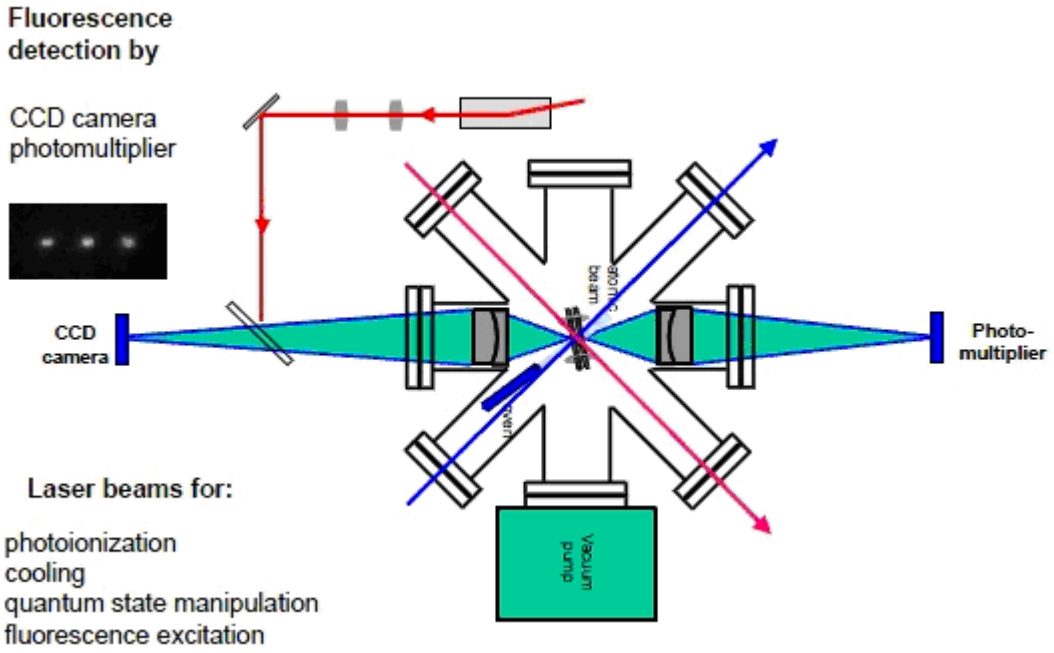

Figure 7. Experiment Set Up

We conclude this section by briefly mentioning that to make larger scale quantum circuits possible, we need to exploit quantum error correction protocol. The largest obstacle to perform a successful quantum error correction protocol seems to be the limited fidelity of the operations. In order to achieve universal quantum computing, the algorithms have to be implemented in a fault-tolerant way. It is commonly accepted that this requires quantum error correction. One of the most important goals currently is to implement quantum error correction repeatedly with high fidelity to stall decoherence and to correct for errors induced by the gate operations. The largest obstacle to perform a successful quantum error correction seems to be the limited fidelity of the operations (Haeffner 2008). In particular, the two qubit gate operations seem to be the main limiting factor. Currently, the read-out of a single qubit can be performed with a fidelity of up to

0.9999. We will skip further discussion of the fault-tolerant layer, as it is not within the scope of this paper. The different stages of ion trap research are structured so as to best assist with the overall optimization while taking into account numerous optimization criteria, including minimizing the number of expensive two-qubit gates, minimizing the number of single qubit gates, optimizing the runtime, minimizing the overall circuit error, and optimizing the classical control sequences. To map a quantum algorithm into an optimized physical-level experiment, we break down all the gates into two qubit controlled rotations of Pauli and other single-qubit gates. Then we map logical qubits into physical qubits. Next we perform further balancing of runtime errors versus other errors until the desired balance is found. So far, the Grover's algorithm, Shor's algorithm and teleportation have been successfully demonstrated on the ion trap.

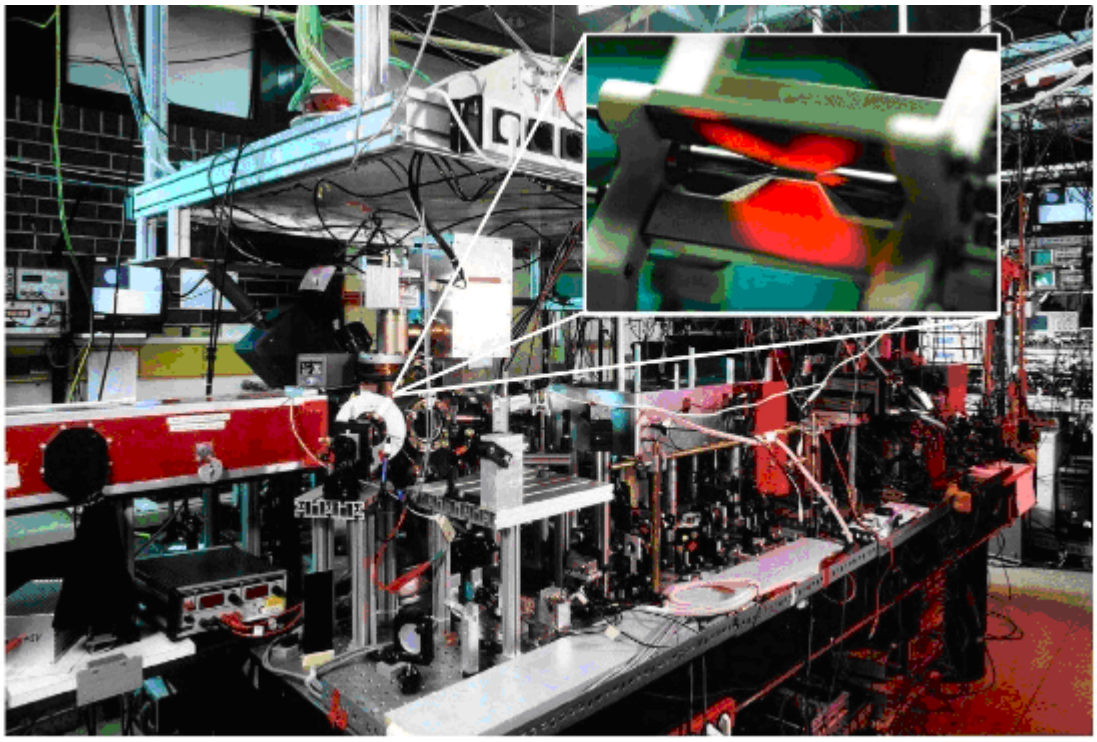

Figure 8. Laboratory

Chapter 3

Implementation of Quantum Simulators

To get an understanding of quantum computation in general and to test the capabilities of current quantum simulation tools, we implemented Shor's and Grover's algorithm using a popular quantum simulator, jQuantum. We will describe these simulator experiments and how they influenced our decisions regarding how to implement our own *k*-QSAT solver in Java. We will then describe our implementation of the SolveQ, a 2-QSAT algorithm. Finally, we describe our implementation of a general "brute force" *k*-QSAT solver.

3.1 jQuantum

jQuantum is a quantum simulator developed by de Vries (de Veries, 2004). It is written in Java using the Java Swing library. On a small scale, quantum simulators allow us to simulate the behavior of a quantum computer on a classical computer. For our own research, we built the Shor's algorithm and Grover's algorithms using this software to learn about current quantum simulators before we developed our own algorithm for solving *k*-local Hamiltonian problems. We built two versions of the Shor's algorithm for factoring, with N=15 and a=7, and with N=21 and a=5, where N is the number that we want to factor and a is an integer that is used, and correctly found the corresponding periods. For the Grover's algorithm, we were able to view how the algorithm searches the database for a given value and gradually collapsed to the correct result. We saw that after the first iteration, one state is already more preferred than all the rest; the rest of the iterations amplify the probability of getting this state, until at the end of the cycles we will have found the desired output for certain.

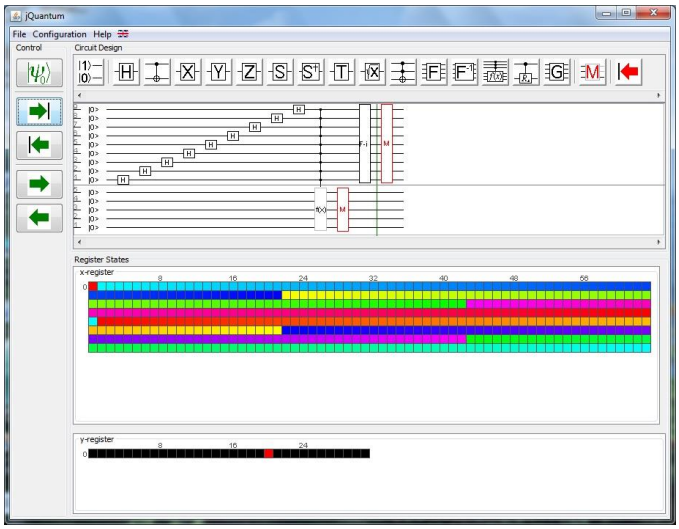

Figure 10. jQuantum Screen

How jQuantum affected us in our writing of our own quantum simulators is that we observed that jQuantum source code keeps track of its quantum states in two 1-D arrays of size 2^n. The first array is for the real coefficients, and the second one is for the imaginary coefficients. In the simulator GUI, the ratios of the imaginary components to the real components of the coefficients are depicted using bright colors, in a representation in which each quantum state is represented by a colored square, as you can see, at the lower portion of the jQuantum screen in Figure 10. In our own quantum simulators we did not use two arrays to keep track of the real and imaginary coefficients like in jQuantum. However, we did use one array to keep track of the probabilities for each of those states. In doing this, we have applied the idea that each quantum state is associated with a number containing information about that state.

jQuantum has many limitations. It does not contain enough rotation gates to build an Inverse Quantum Fourier Transform, which greatly limits its potential for doing real work. Also the fact that jQuantum can handle no more than 15 qubits total limits the kinds of problems it can be used for. Still, the use of colors to illustrate the concept of

quantum information processing was laudable. Despite its merits, jQuantum cannot be used to conduct our *k*-local Hamiltonian experiments and after failing to find such a quantum simulator, we decided to write our own quantum simulators.

3.2 SolveQ

The first quantum simulator we wrote was the algorithm SolveQ written by Bravyi. The SolveQ algorithm (Beaudrap & Gharibian, 2015) is a quantum algorithm for solving 2-QSAT problems. We implemented this algorithm in Java in about 800 lines of code. We tested how well this 2-QSAT algorithm solves 2-SAT problems, especially compared to the brute force classical algorithms of solving *k*-SAT problems. Since the quantum algorithm also has to work for non-classical inputs, it solves problems by a much more complicated method than the classical algorithm to find just a single solution let alone all of the solutions like in the classical algorithm. For the rest of our discussion, we shall call the constraint matrix, the matrix Π. Each constraint is represented by a matrix like the following: $\prod \quad = |\phi\rangle\langle\phi| = |0\rangle\langle 0| \otimes |1\rangle\langle 1|$

A 2-SAT clause or a constraint containing two variables is described using the following definitions:

For the OR clause ($x_1 \vee x_2$), the $|\phi\rangle$ is $|0\rangle \otimes |0\rangle$

For the OR clause ($\bar{x}_1 \vee x_2$), the $|\phi\rangle$ is $|1\rangle \otimes |0\rangle$

For the OR clause ($x_1 \vee \bar{x}_2$), the $|\phi\rangle$ is $|0\rangle \otimes |1\rangle$

For the OR clause ($\bar{x}_1 \vee \bar{x}_2$), the $|\phi\rangle$ is $|1\rangle \otimes |1\rangle$

The solutions to one clause include all combinations of 1 and 0 on the chosen variables that are not exactly the same as the ones in $|\phi\rangle$. Take an example, for a constraint on particles 3 and 5, if we have $|\phi\rangle = |1\rangle \otimes |0\rangle$, then we can get a solution for particles 1 through 7 as a 7 digit binary strings, as long as particle 3 is not 1 and particle 5 is not 0. In other words, the solutions to the single clause ($\bar{x}_3 \vee x_5$), can be 01, 00, 11, but not 10. Solutions 01, 11, and 00 are called null vectors of the forbidden matrix or matrix-Π, while 10 is called the forbidden vector. Similarly, to find the solution of $(x_3 \vee x_5) \wedge (x_3 \vee \bar{x}_5)$, you exclude the forbidden vectors 00 and 01, and are left with 10 and 11. As you can see, the more clauses there are, the fewer solutions, which is why a clause is also often referred to as a "constraint." It should be noted that if all four of the clauses with $|\phi\rangle$ = {00, 01, 10, 11} are in the problem, then the rank of the Π matrix will be four and there will be no solutions to the entire problem, since there is no null space to the Π matrix. In other words, the existence of solutions or null vectors depends on the existence of fewer than the maximum number of constraints. Furthermore, the solutions to two clauses, like $(x_3 \vee x_5) \wedge (x_3 \vee \bar{x}_5)$, must satisfy both clauses simultaneously. By the same logic if there are 15 clauses, the solution set is the intersection of the solution sets of all 15 clauses when considered independently. Solutions that satisfy all 15 clauses simultaneously satisfy the sum of the fifteen Π matrices. We shall make use of this fact later; this method of directly looking for all the null vectors is called the brute force way of solving a problem. But, as one can see, the algorithm we are currently talking about, SolveQ uses a more elegant method of figuring out the solutions, it uses a "chain reaction" built from "transfer matrices." The drawback of this method is that it figures

out only one solution and not all possible solutions, and like most other quantum algorithms, the probability of getting the correct answer is greater than ½ but less than one.

For 2-SAT problems, each clause or constraint operates on two variables, and each variable can be either negated or not negated. If the variable is negated, it is represented by $|1\rangle$, and if not, it is represented by $|0\rangle$ in the vector $|\phi\rangle$. The solutions to the clause lie in the null space of the matrix Π, which is found by analyzing the matrix $\Pi = |\phi\rangle\langle\phi|$. Now that we have the vector $|\phi\rangle$, we can find the transfer matrix *T* based off of this vector, which is given by the equation from the paper (Beaudrap & Gharibian, 2015), and which we will define in a moment. There are actually two such transfer matrices for each matrix described by the vector $|\phi\rangle$, one to represent the edge that goes from *i* to *j*, and the other from *j* to *i*. We use whichever one is necessary to make our chain reaction. A chain reaction is a sequence of matrix multiplications of these transfer matrices, which when applied to our starting variable state *i* allows us to obtain successive solutions *j*. This process only works if there are no "conflicts" in the chain reaction, which has two indications: First, the same variable or qubit cannot take on two different values. Second, after the chain of multiplication with the transfer matrices, the qubit cannot be the zero vector $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, which would indicate the chain reaction has ended.

The transfer matrix is used for traversing a chain reaction, starting with some arbitrary variable that has either $|0\rangle$ or $|1\rangle$ as its initial state. To determine whether the initial state is $|0\rangle$ or $|1\rangle$, we do the following: If the $|\phi\rangle$ component for variable *i* is $|0\rangle$,

then we use the state $|0\rangle$, and if the $|\phi\rangle$ component for variable *i* is $|1\rangle$, we use the state $|1\rangle$ as our input going from *i* to *j*. Given below is how to obtain the transfer matrix based on the vector $|\phi\rangle$.

As you will see later, our implementation of this algorithm SolveQ differs from our implementation of the brute force *k*-QSAT solvers in labeling conventions. In SolveQ the left ket is the first particle, or particle *i*, while for the *k*-QSAT solver, the leftmost ket is the last or nth particle. This is just a note about how we labeled the parameters. You can label them some other way, but you will have to stay consistent to get the correct answer. The transfer matrices derived from the phi vector, $|\phi\rangle = |i\rangle \otimes |j\rangle$, are listed below, and allow one to fix variable *j* given variable *i* as input.

$$\text{For } |\phi\rangle \text{ is } |0\rangle \otimes |0\rangle \; T_{ij} = |1\rangle\langle 0|, \; T_{ji} = |1\rangle\langle 0|$$

$$\text{For } |\phi\rangle \text{ is } |1\rangle \otimes |0\rangle, T_{ij} = |1\rangle\langle 1|, \; T_{ji} = |0\rangle\langle 0|$$

$$\text{For } |\phi\rangle \text{ is } |0\rangle \otimes |1\rangle, T_{ij} = |0\rangle\langle 0|, \; T_{ji} = |1\rangle\langle 1|$$

$$\text{For } |\phi\rangle \text{ is } |1\rangle \otimes |1\rangle, T_{ij} = |0\rangle\langle 1| \; , \; T_{ji} = |0\rangle\langle 1|$$

In the transfer matrices above, for T_{ij}, you operate on state of *i* to get output state of *j*. For the $|\phi\rangle$, the left element is *i* and the right element is *j*. Using these transfer matrices, you can either traverse forward from 2 to 3, or you can traverse backwards from 3 to 2 using the other transfer matrix. If you successfully found a chain reaction, or CR, then you have found the solutions to the variables contained in the CR. Sometimes it takes

more than one CR to find the entire solution. To know when the CR terminates, you apply the transfer matrix to your variable state ($|0\rangle$ or $|1\rangle$), if you get the zero vector $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$, then the CR has terminated, and if you do not have the entire solution, you need to start a new CR.

There are two types of CRs. The first is the normal kind with no repeating vertex. The second has a repeating vertex, and it is called a discretizing cycle. As mentioned earlier, even though you traverse a vertex twice, you are supposed to get the same state both times for a single particle. If this is not the case, then the CR is invalid and it is said to have a "conflict" (Moore & Mertens, 2011). When you detect such a conflict, you know that there is no solution to the entire problem. Allow me to illustrate how the SolveQ solves a specific problem. For the problem:

$(x_1 \vee \bar{x}_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge (x_4 \vee \bar{x}_5) \wedge (x_5 \vee \bar{x}_6)$, we have the "chain reaction" built from multiplication of 5 of these transfer matrices, $|0\rangle\langle 0|$, since there are 5 clauses in the problem. In this example, the solutions are all 0 or false for all of the six variables.

This algorithm is really nice and easy to play with on piece of scratch paper; you can find solutions easily just by avoiding conflicts in your CR. However, it is very difficult and involves knowledge of tree traversals to program. Thus we traversed the list of constraints destructively as we fixed our solutions, by removing each constraint after we are done using it. We used multiple copies of the list of constraints because more than one was needed. The CR example given at the end of the previous paragraph only uses T_{ij} and never T_{ji}, where *i* is the first variable in your OR clause, and *j* is the second variable in your OR clause. This is only a special case, most CRs use both T_{ij} and T_{ji}.

The preliminaries that happen before the body of the algorithm are also easy to understand but hard to program, in which you make sure rank-4 constraints prints "Unsolvable," and shuts down the program, and then you have to go through the long and arduous task of processing the rank-2 and 3 constraints; only rank-1 constraints do not need processing, you will take care of them in the last step of the algorithm that happens after the preliminaries.

So far, we have only considered how this 2-QSAT algorithm solve 2-SAT problems, but we have not considered how it solves 2-QSAT problems. To do this we basically add another term to the $|\phi\rangle$ vectors as well as to the transfer matrices as shown below. This extra term allows some unsolvable 2-SAT problems to become solvable, but you have to set a threshold, and so long as your current product is greater than your threshold, the solution you got is still valid, otherwise an "Unsolvable" message will be printed just like in the earlier 2-SAT version of the program.

For $|\phi\rangle = \alpha|0\rangle|0\rangle + \beta|1\rangle|1\rangle$ we have, $T_{ij} = \beta|0\rangle\langle 1| - \alpha|1\rangle\langle 0|$, $T_{ji} = \beta|0\rangle\langle 1| - \alpha|1\rangle\langle 0|$

For $|\phi\rangle = \alpha|1\rangle|0\rangle + \beta|0\rangle|1\rangle$ we have, $T_{ij} = \beta|0\rangle\langle 0| - \alpha|1\rangle\langle 1|$, $T_{ji} = \beta|1\rangle\langle 1| - \alpha|0\rangle\langle 0|$

For $|\phi\rangle = \alpha|0\rangle|1\rangle + \beta|1\rangle|0\rangle$ we have, $T_{ij} = \beta|1\rangle\langle 1| - \alpha|0\rangle\langle 0|$, $T_{ji} = \beta|0\rangle\langle 0| - \alpha|1\rangle\langle 1|$

For $|\phi\rangle = \alpha|1\rangle|1\rangle + \beta|0\rangle|0\rangle$ we have, $T_{ij} = \beta|1\rangle\langle 0| - \alpha|0\rangle\langle 1|$, $T_{ji} = \beta|1\rangle\langle 0| - \alpha|0\rangle\langle 1|$

On classical inputs, this algorithm is very poor in quality. First of all, the transfer matrices only give you one assignment out of every three possible assignments, so it ignores a large part of the possible solution set. As a result, many of the problems that

are solvable will be labeled as unsolvable by the algorithm even if you performed the correct steps. There are also many extraneous rules you have to follow to get a valid assignment. For example, the order you add your constraints to the constraints list influences what solutions you will get. So to properly use our program for your purpose, you have to set fixed rules regarding the order in which the constraints must be added to the list based on rank of constraint or the probabilities in your OR clauses. Also our convention was that the first variable number must be less than the second variable number, so that constraints on the same two variables can be easily found.

3.3 The Random Generator

The purpose of this experiment section is actually not just to solve 2-SAT or 3-SAT problems, but to see how difficult it is to simulate quantum algorithms on a classical computer; and to gauge the difference between a classical and a quantum algorithm. We want to explore the boundary between problems that are solvable and non-solvable using a classical algorithm and compare it with the boundary between solvable and non-solvable using a quantum algorithm. Our experiments involve running our *k*-QSAT solvers on random problem instances and observing the number of problems that were satisfiable; again, this number is dynamic depending on the number of clauses in the problem. We now describe how we generated random problem instances. In both the quantum and classical cases we know that as the number of clauses to number of variables ratio increases, the probability that a random instance will have a satisfying assignment decreases from 1 to 0. For the classical case, this ratio is approximately between 3.5 and 4.5. In order to conduct our experiments we did 50 trials and recorded the number of satisfied trials for a given number of clauses. We wanted to conduct our

experiment so that we could better understand this transition zone or “freezing point” zone.

To write the random instance generator, first we need a clause-size to determine how many variables we are acting on in one clause. Then we need to define the total number of variables in the problem, this number needs to be greater than or equal to the clause-size. We also need the number of clauses to determine how many constraints there are in each problem. Lastly, we need the number of instances, which determines how many problems we want in our research, so that we get a more statistically average view using an average problem. Once we know we need these four numbers, we generate all other numbers randomly using Java’s Math.random() function.

3.4 What is a *k*-SAT problem?

A *k*-SAT problem consists of a set of clauses where each clause has *k* Boolean variables or their negations joined by disjunction signs. The clauses are then further joined by conjunction signs. The problem is an AND clause of these OR clauses. In the quantum way of solving problems, each OR clause is represented by a tensor product of outer products. For *k*-SAT, there are a maximum of *k* such outer products joined in tensor product. The AND symbol joining the clauses means simultaneously satisfaction of the individual OR clauses is required.

Any *k*-SAT formula can be converted to a 3-SAT formula that is satisfiable only if the original formula is. If a clause has just one or two variables, we can add dummy variables, padding it to a set of 3-variable clauses. A k-SAT clause becomes *k-2*, 3-SAT clauses linked together by *k-3* dummy variables *z*. The following is an example of how a

5-SAT clause can be reduced to 3-SAT form:

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \Leftrightarrow (x_1 \vee x_2 \vee z_1) \wedge (\bar{z}_1 \vee x_3 \vee x_2) \wedge (\bar{z}_2 \vee x_4 \vee x_5)$$

This is the argument that all *k*-SAT problems can be efficiently reduced to 3-SAT problems.

3.5 Random *k*-SAT solver: Version 1

(Classical Brute Force Solver, Repeating Variables)

The second program we implemented was a brute force solver for classical *k*-SAT problems with *n*=7 variables. Below is the pseudo-code for what was implemented.

For each problem instance: (For example, each problem contains 15 clauses)

I. Make each clause:

1. Generate *k* non-repeating particles numbers for each *k*-SAT clause.
2. For a 3-SAT clause, generate an integer 0-7 that when converted to binary represent the forbidden vector, phi or $|\phi\rangle$.

II. Sort each clause by ascending variable numbers, the matching bits in the OR clause are attached to the variable numbers and must be swapped as well during the sort so that it remains matched up with particle numbers. Example: 2,1,3 is sorted to 1,2,3. If the phi vector was originally 011, it is now 101.

III. Generate binary strings that represent the integers 0-127, and add them to a solutions list.

IV. Take one clause. Cycle through the recent solutions list to find the elements that needs to be deleted. The elements that needs to be deleted are the n-digit strings that match the k-digit forbidden vector at all the *k* given particle number locations.

V. Repeat IV for all the rest of the clauses in a problem.

END

Allow us to illustrate the example when each constraint operates on *k*=3 variables. A fourth parameter is needed that is a decimal number equivalent of one of the eight binary strings, 000, 001, 010, 011, 100, 101, 110, 111. Thus, the fourth parameter is an integer from 0 to7. This parameter together with the *k* variables make up each constraint or clause. Again, like mentioned in the SolveQ section, if the variable is negated, it is

represented by 1 and if not, it is represented by 0. So, for example, the clause (1,2,3,7) is a constraint operating on variables 1, 2, and 3, and the phi vector is represented by 111 or $|1\rangle \otimes |1\rangle \otimes |1\rangle$. Again, our convention was the leftmost ket corresponds to variable 3.

First, we generated all the possible assignments for a 7-variable problem. How we did this was by taking all the integers counting from 0 to 2^7-1 , and converting them each to a seven digit binary strings. Because it is a 3-SAT problem, each constraint operates on 3 of the 7 variables. Furthermore, the 3-SAT clauses are OR clauses, so for the example given at the end of the last paragraph, 000, 001, or 110, matched to (1, 2, 3), so all would solve the clause perfectly. But 111 matches to (1, 2, 3) is definitely wrong, because $|1\rangle \otimes |1\rangle \otimes |1\rangle$ is a forbidden vector and cannot be a null vector. Notice that we are given a 7-variable problem and we only consider 3-variable clauses, the rest of the variables—4 of them— can take on either 0 or 1, and would not influence decision of whether that particular string is a solution.

Continuing with the same example, we next eliminate the wrong solutions from the original set containing 0 to $2^7 - 1$ in binary. We cannot have 7 variable combinations, where variable 1 is 1, variable 2 is 1, and variable 3 is 1, simultaneously. That is why we need to remove these possibilities from our solution set. After we successfully find a solution to our problem, we can interpret the 0 and 1 to mean the Booleans: false and true.

After processing one clause we repeat what we just did for the other 14 clauses, or however many we had in our problem. A solution to our problem must satisfy all 15 clauses at once. It is obvious that some clauses are tautologies and do not constrain the assignments of the variables, which is why we wrote Version 1 so as to not allow repeating particles, to prevent ever generating these tautologies in the first place. Also

because of the nature of random number generation in Java, it was much too easy to get problem instances with only one or two solution strings out of the original 128 possible strings. To ignore these cases, we set 6 or less solution strings as being equal to no solution strings. Doing this allowed us to get a freezing point ratio within the expected range based on article (Bravyi, Moore & Russel, 2014).

3.6 Random *k*-QSAT Solver: Version 2

(Π ArrayList and $2^{\text{number of variables}}$ -1 null vectors)

For each problem: (For example, each problem contains 15 clauses)

I. Make each clause:

1. Generate *k* non-repeating particles numbers for each *k*-QSAT clause.
2. Generate 2^k coefficients for a *k*-QSAT clause. For 3-QSAT, generate 8 coefficients.
3. Normalize the *k*-QSAT clause, so that the sum of the coefficients adds up to 1.
4. Make forbidden matrix:
    1. Generate the k-bit and n-k bit strings. Insert the *k* bits into the *n*-*k* bits to get a row in the forbidden matrix after. First sort the *k* variables and bits by ascending variable number which is necessary for inserting correctly. After this insertion, we have a row. Coefficients go along with each row. This is accomplished through an inner class: State, which is a package of a binary string and a coefficient.
    2. Sort the States by converting each n-bit String to its decimal number. This is the number of leading zeros in each row of the forbidden matrix.
    3. The resulting forbidden matrix should be a diagonal matrix.

II. Add up all the forbidden matrices in one problem

III. Generate $2^n - 1$d null vectors then orthogonalize and normalize them.

IV. Find Solution and verify Solution.

1. Find the solution by taking the term in the sum of null vectors with largest probability.
2. Find the term in the 2^k term clause with largest probability and add to a list.
3. Verify that the solution satisfy each of the *k*-SAT clause in the list. Each clause is a forbidden vector; the solution mustn't match exactly with the forbidden vector.

END

We now look at different variants of the *k*-QSAT solver. Version 2 is our first quantum algorithm for solving *k*-QSAT problems. It was not used for our freezing point experiment because it used a faulty way of quantum information processing but it had some usable sections of code that were reused later, in Version 5. Unlike in the classical way of solving problems where the fourth constraint parameter is just one term such as 010, the quantum constraint is composed of all 2^k (for 3-SAT it is 8) such terms with a complex coefficient multiplied to each term. For the two-particle, or 2-SAT case, we have $\Pi = a|0\rangle\langle 0|\otimes|0\rangle\langle 0| + b|0\rangle\langle 0|\otimes|1\rangle\langle 1| + c|1\rangle\langle 1|\otimes|0\rangle\langle 0| + d|1\rangle\langle 1|\otimes|1\rangle\langle 1|$ (taking the outer product of phi with itself will give us the constraint matrix-Π). Also note that the vector (a,b,c,d) must be normalized by a+b+c+d=1. This is one constraint on the two particles. To get another constraint on these same two particles, we have to get another such Π like this one and add the two Π matrices. We must be careful that if the two constraints are not operating on the same list of particles, we cannot add them. Because we cannot add together clauses that operate on different particles, we first have to convert each *k*-SAT constraint to a n-variable matrix-Π, so that they can be added. Version 2 does this through the cut and paste of strings, and then sorting to get the net seven-particle matrix-Π at the end. When we calculate the solutions, we have to find the vectors that are in the null space of all of these constraints. The coefficients a, b, c, d are decimals that are randomly generated. The solution to the problem lies in the null space of the net Π matrix. For 15 different constraints, the solutions need to lie in the null space of all 15 constraints simultaneously. To make sure of this, we added up all 15 n-particle Π matrices together, then we found the null space of the sum of all 15 of them.

Let us next discuss how this algorithm is special. Basically, when you measure, each of the 127 null vectors has same chance of being measured. Each null vector is further composed of 128 variable assignments. The 127 null vectors are found as follows:

$$|\phi\rangle = (a_0, a_1, a_2, a_3, a_4, a_5 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots a_{127})$$
$$|n_1\rangle = (a_1, -a_0, 0,0,0,0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 0)$$
$$|n_2\rangle = (a_2, 0, -a_0, 0,0,0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 0)$$
$$|n_3\rangle = (a_{3,} 0,0, -a_0, 0,0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 0)$$
$$|n_4\rangle = (a_4, 0,0,0 - a_0, 0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 0)$$
$$|n_5\rangle = (a_5, 0,0,0,0, -a_0, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 0)$$
$$\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots$$
$$|n_{127}\rangle = (a_{127}, 0,0,0,0 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots - a_0)$$

Each of the single assignment states has a coefficient that represents its probability. Using nested for-loops, we cycle through the null vector stack, and through each null vector single assignment state, to find the single assignment state with the net largest probability, after we summed together the probabilities of the 127 null vectors. We took our solutions to be the ones with variable assignments out of 128 possible variable assignments with the largest coefficient after summing the 127 normalized null vectors. Next, we retrieved our clauses list. Each clause used to be a superposition of 8 (3-SAT) possible forbidden vectors; now we reduce it to just one possibility—the one with the largest probability. Finally, we checked if our solution satisfies all the single-term clauses. And it is rarely ever satisfied. So we never got this algorithm, Version 2, to work, because our solution never satisfies the all the list of single-term clauses. From our

mistake we have learned that the important lesson that the solution does not have to satisfy a *k*-SAT problem, to satisfy a *k*-QSAT problem.

3.7 Random *k*-QSAT Solver: Version 3

(Permutator Matrix P, Threshold, Non-repeating variables)

For each problem: (For example, each problem contains 15 clauses)

I. Make each clause:

1. Generate *k* non-repeating particles numbers for each *k*-QSAT clause.
2. Generate coefficients for a *k*-QSAT clause. For 3-QSAT, generate 8 coefficients, and add them to a coefficients list.
3. Normalize the *k*-QSAT clause, so that the sum of the coefficients adds up to 1.
4. Make forbidden matrix:
    a. Make a 1-D array, with the first *k* terms containing the particle numbers for that clause, and the rest of *n-k* terms containing the remaining particle numbers.
    b. Use this 1-D array to make the permutator matrix, and add to permutator list.
    c. Use this 1-D array and the *k* coefficients list to make each constraint matrix, and add to constraints list.

II. Use a loop to cycle through the permutator list and the constraints list, and find the matrix product PAP^{-1}, where *P* is a permutator and *A* is a constraint matrix.

III. Sum together all the constraint matrices in one problem

IV. Do Gaussian elimination on this sum, to get the net forbidden matrix.

V. Look on the diagonal of the net forbidden matrix to check for values below 0.0055, the threshold for null vector.

END

First we generate three random particle numbers, and we use these three integers to make the permutator matrix *P*. The matrix *P* is a permutation matrix that permutes the rows of the identity matrix, the size of which is 2^n. Next we use the three particle numbers for 3-SAT clause to construct the forbidden matrix. For a clause of size *k*, we generate 2^k random decimals for the probability of each term in the forbidden matrix. We normalize the 2^k numbers to total length of one; then we use these values to generate the

forbidden matrix. The forbidden matrix is a tensor product of the particle numbers we are acting on with the particles we are leaving alone. For the particles we are leaving alone, we simply use the two-by-two identity matrix. Next, we multiply them together: P times the forbidden matrix times the inverse of P. So far what we have described is one k-QSAT clause. If a single problem has more than one clause, we simply add together each element of the matrices for all the clauses, we find $\sum_i P_i A_i P_i^{-1}$. Next we do a Gaussian elimination routine on this matrix, the net forbidden matrix to get it into diagonal form. In the last step we see if there are any null vectors by checking the diagonal of the matrix for numbers equal to 0 or below a threshold of say 0.0055. If there are any such numbers, then we have successfully solved the problem, but if there are not, the problem is unsolvable.

3.8 Random *k*-QSAT Solver: Version 4 (Permutator Matrix P, Fewer than 2^k terms, Non-repeating variables)

For each problem: (For example each problem contains 15 clauses)

I. Make each clause:

1. Generate k non-repeating particles numbers for each k-QSAT clause.
2. Generate coefficients for a k-QSAT clause. For 3-QSAT, generate 8 coefficients. Do a coin flipping using Math.random() to determine that each of the 8 coefficients is either 0 or a decimal.
3. Normalize the k-QSAT clause, so that the sum of the coefficients adds up to 1.
4. Make forbidden matrix for each clause, the way it was done in Version 3.

II. Add up all the forbidden matrices in one problem, to get the net forbidden matrix.

III. Do Gaussain elimination on the net forbidden matrix.

IV. Find solutions by looking for 0 on the diagonal of the net forbidden matrix.

END

Why we wrote this version is that we realized that Version 3 will never have true null vectors. Version 3 can never have true null vectors, since it is mathematically predictable

as having a full diagonal. To fix this problem, we wrote Version 4 so that the only difference between Version 4 and Version 3 is that Version 4 uses single clauses with a variable number of terms with non-zero coefficients— ranging anywhere between 0 and 8 inclusive for a 3-QSAT case, while for Version 3 we used a linear superposition of all 8 terms with all non-zero probabilities. In other words, Version 3 uses a threshold; whereas, Version 4 only uses true null vectors with 0-energy.

3.9 Random *k*-QSAT Solver: Version 5 (Π ArrayList, Fewer than 2^k terms, Non-repeating variables)

For each problem: (For example, each problem contains 15 clauses)

I. Make each clause:

1. Generate *k* non-repeating particles numbers for each *k*-QSAT clause.
2. Generate coefficients for a *k*-QSAT clause. For 3-QSAT, generate 8 coefficients. The coefficients are either 0 or a decimal.
3. Normalize the *k*-QSAT clause, so that the sum of the coefficients adds up to 1.
4. Make forbidden matrix for each clause, the way it was done in Version 2, using the cut and paste of strings.

II. Add up all the forbidden matrices in one problem, to get the net forbidden matrix.

III. Find solutions by looking for 0 on the diagonal of the net forbidden matrix.

END

Although Version 2 did not work, the cut and paste of strings idea can be readapted to a working version. The 127 null vectors idea is discarded because we do not need null vectors with more than one term out of the 128 single assignment states; we want a single assignment written in the 7 qubits that satisfy our 3-SAT problem, not a superposition of more than one assignment. However using the cut-and-paste of strings to build the forbidden matrix is still a valid idea. Version 4 and Version 5 works approximately the same because both involve forbidden matrices built by zeroing some of the total of 2^k

terms. The only difference between Version 4 and Version 5, is that Version 5 uses a cut-and-paste of strings technique; whereas Version 4 uses 2-D arrays and permutators. The cut-and-paste of strings is a method for converting a 3-SAT clause, to a 7-particle forbidden matrix; we take the 3 bits in the binary string representing the phi vector, for example 011, and insert these 3 bits into the other 4 bits at the correct locations, determined by the *k* particle numbers that we have set for that 3-SAT clause.

3.10 Random *k*-QSAT Solver: Version 6 and 7 (Permutator Matrix P, Threshold, Repeating variables)

For each problem: (For example, each problem contains 15 clauses)

I. Make each clause:

1. Generate *k* particles numbers for each *k*-QSAT clause that allows for repeating particles.
2. Find the number of unique elements, *u*, in a *k*-QSAT clause, and generate 2^u coefficients.
3. Normalize the *k*-QSAT clause, so that the sum of the coefficients adds up to 1.
4. Make forbidden matrix:
    1. Make a 1-D array, with the first *k* terms containing the particle numbers for that clause, and the rest of n-k terms containing the remaining particle numbers.
    2. Use this 1-D array to make the permutator matrix, and add to permutator list.
    3. Use this 1-D array and the *k* coefficients list to make each constraint matrix, and add to constraints list.

II. Use a loop to cycle through the permutator list and the constraints list, and find the matrix product PAP^{-1}, where *P* is a permutator and *A* is the constraint matrix.

III. Sum together all the constraint matrices in one problem

IV. Do Gaussian elimination on this sum, to get the net forbidden matrix.

V. Look on the diagonal of the net forbidden matrix to check for values below 0.0055, the threshold for null vector.

END

Before this version of the *k*-QSAT solver, the best version was Version 3. Originally the idea of Version 6 was motivated by the fact that Version 3 was not working—it had

no freezing point because it was coded wrong. But we did not find this out until much later; at first we thought that Version 3 did not work, because it did not use repeating variables. So we made the appropriate changes to Version 3 to make a new version, Version 6, and concluded that Version 6 was best for solving *k*-QSAT problems. Then we went through a brief phase of doubt and tried to write a Version 7, because Version 6 does not factor in "empty" clauses that do not put any new constraints on a repeated variable because it is of the form: $(x_1 \vee \bar{x}_1 \vee ....)$. Later we abandoned this idea because these tautologies lead to inflated number of clauses when we calculate the freezing point ratio and should therefore be omitted. So for a long time the “best” version was Version 6. In the end, Version 3 and Version 6 work approximately the same, as we expected, because the matrix with or without repeating particle should have a full diagonal and therefore operates about equally well.

3.11 The Antiferromagnetic Heisenberg Model and the Ising Model

The matrices used to solve *k*-SAT problems are only one type of *k*-Local Hamiltonians, there are other existing Hamiltonians such as the Ising Model and the Aniferromagnetic Heisenberg Model, shown below.

Anitferromagnetic Heisenberg Model: $H = \sum_{i,j} \left(\sigma_i^x \sigma_j^x + \sigma_i^y \sigma_j^y + \sigma_i^z \sigma_j^z \right)$

Ising Model: $H = -J \sum_{i,j} \left(\sigma_i^z \sigma_j^z \right) - g \sum_i \sigma_i^x$

To understand where the Ising Model comes from, consider how a block of iron put in a magnetic field becomes a magnet; it can even magnetize itself spontaneously without an external magnetic field. However, memory of the magnetic field decreases as its temperature increases, and at the critical temperature it loses its memory completely. The

model for this type of magnetism is the Ising Model, in which neighboring spins interact. The first term in the Ising Model is called the "energy" term. If the J constant is greater than 0, the model is ferromagnetic and the spins of the particles are in the same direction. If the J constant is less than 0, the model is antiferromagnetic and the spins of the particle tend to align in opposite directions. If the J constant is 0, the spins are randomly up or down. The second term in the Ising Model is the magnetization term, and it is affected by temperature change. At low temperature, energy tends to be minimized, and magnetization is high. At high temperature, entropy tends to be maximized, so spins tend to disalign and the magnetization tends to be low. The critical temperature for determining whether the temperature is high or low is T=2.269 J/k-boltzman. If g is negative, spin aligns in positive direction. If g is positive, spin align in negative direction. If g is 0, there is no external influence on the spin site. For 9 spin sites, the ground state is $\frac{E_0}{N} = -(12/9)J$. When there are a very large number for spin sites, this value gets closer and closer to $\frac{E_0}{N} = -2J$. The next step in exploring with these models is to wrap the sheet of spin sites into exotic shapes such as tubes, toruses and twisted toruses. We actually attempted to program a nine spin-site tube, torus and twisted torus, but unfortunately, when we ran the program the computer crashed, due to large matrix size. A maximum of 6 particles in the tube configuration was workable.

As the last part of research, we experimented with these models in a random way so we get a more statistically averaged view of how they behave. First we built the Hamiltonian matrices for the models. We do this according to the following plan:

1. Specify how many particles are there in one interaction.

2. Specify how many neighbors the central atom has.

3. Specify how many interactions there are in one problem.

4. For each interaction, generate random values for the neighbors that are currently interacting with the central atom.

5. Build the Hamiltonian for the interaction using tensor product.

The Hamiltonian we get is for one clause or one interaction. If we have more than one interaction, we simply sum up the matrices for all the interactions. Next we see if there are any null vectors. It turns out that there are not. So next we want to see if there are any low energy solutions or solutions with small eigenvalues. Such eigenvalues need to be below a certain threshold. One way to do this is checking a determinant to see if it is close enough to zero, or a better way is to use the Gaussian elimination.

3.12 Freezing Point Experiment

After having written the code discussed above, I used these codes to conduct a "freezing point" experiment. The theory behind this experiment is that when there are too many constraints in a problem, the problem will have no solution. Usually, this occurs when the number of clauses to number of variables is around 3.5 to 4.5. As you will see in the data tables below, if the problem is solvable, the number of successes is incremented by one. If the problem is unsolvable, the number of successes will not be incremented. Most of my experiments on *k*-SAT problems used 50 instances; while for the Ising and Heisenberg models, I used 3 to 5 instances, because any higher will cause the computer to crash.

Chapter 4

Experiments and Results

4.1 Data and Analysis

In this chapter, we describe the freezing point ratio experiments for our quantum and classical algorithms when running on a classical computer and compare them. The freezing point ratio is a ratio between the maximum number of clauses for each instance of *k*-SAT problem to be solvable, and the total number of particles or variables in the problem. The reason that an instance is likely to become unsolvable is that as the number of clauses increases, and more terms are added together is that, it becomes more and more likely that the resulting matrix will not have null vectors. Null vectors exist because there are "empty slots" in the matrix diagonal; when there are too many matrices being added, the empty slots get filled up, so there are fewer and fewer null vectors. In other words, this is because the more terms there are, the more forbidden vectors or non-zero-energy eigenvectors there are, leaving no room for zero-energy eigenvectors. For quantum algorithms, the freezing point ratio should be between 0.818 and 3.594. For classical algorithms, the freezing point ratio should be between 3.52 and 4.490 (Bravyi, Moore & Russel, 2014). The data that we collected were slightly different from our expectations. First of all, there was no sudden drop to 0 in the null vector count—the drop is very slow and gradual instead. Also, it takes more clauses for the freezing point to decrease to zero for the classical algorithm than for the quantum algorithm. Secondly, we had some difficulty fitting our data to the required ranges from paper (Bravyi, Moore & Russel, 2014); it appears that our result always falls on the boundaries of these ranges.

Clausesize =3
# of variable=7

# of clauses=5-31
# of instances=50

Table 1.

Version 1 Success Count

#Clauses	**5**	**10**	**15**	**20**	**25**	**31**
#Success	**50**	**50**	**49**	**36**	**13**	**1**

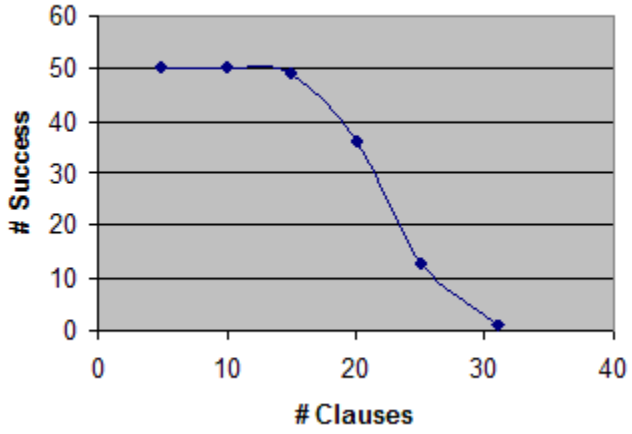

Figure 11. Version 1 Freezing Zone

Version 1 is our classical algorithm for solving *k*-SAT problems. The data is collected using quantum clauses that are determined by Java's random number-generating feature, Math.random(). Therefore, there exists a fluctuation of about 1 to 5 counts in the number of satisfiable problems each time the program is run. Sometimes the curve for the freezing zone will reflect data that are above or below the statistical average portrayed by the steady decline like the one shown in Figure 4.1. Initially, in our results, the freezing point was found to be 10.0 for 7-variable problems, while the freezing point ratio is supposed to be between 3.52 and 4.490

according to paper (Bravyi, Moore & Russel, 2014). So the correct result was apparently not supported by my data, in which the satisfaction curve declines very gradually to 0, at a much slower rate. At first we did not know why this happened, but after a while, we realized that we should use non-repeating particles, because a large percentage of repeating particle constraints are tautologies. By "non-repeating" we mean that the quantum clause can act on variables 1, 2, 3 but not 2, 2, 3, or 3, 3, 3; so all three variable have to be different for the 3-SAT clause the program generated. We also know that Java uses pseudorandomness rather than true randomness, and it eventually repeats itself. Both of these factors combined cause the program to be more likely to get solvable instances than if we use repeating particles. There are, after all, 128 choices for our solutions to begin with, and to have them all crossed out after the algorithm, is a very difficult task that truly depends on coincidence. There are different possible ways we could define the threshold from our data. The way we described above is to use the perfect 0 success out of 50. Another way is to use the beginning of the decline from 50, when we have 49 or 48 out of 50; in this case the freezing point ratio would be 2.857. To summarize, at first we were using repeating particles, and it took 70 clauses to zero our success count, which was too high, so to decrease the success count, we then used non-repeating particles only, which took 55 clauses to zero the success count for certain. Then because 55 is still too high, we decided to set the rule that 6 or less solutions count as no solutions. We did this so that we could further decrease the number of clauses to reach zero success count at 31 clauses, which barely satisfies the bounds given in paper (Bravyi, Moore & Russel, 2014). Since we could find no other errors in my program, we were satisfied that 6

was still a reasonably small number compared to 128. In doing this, the freezing point ratio we ended up with was 4.429. The freezing zone is from 20 clauses to 31 clauses corresponding to the ratio range 2.857 to 4.429.

Clausesize =3
# of variable=7
# of clauses=1-5
# of instances=50
Threshold=0.0065

Table 2.

Version 3 Success Count

# of Clause	**1**	**2**	**3**	**4**	**5**
# Satisfied	**50**	**32**	**10**	**1**	**0**

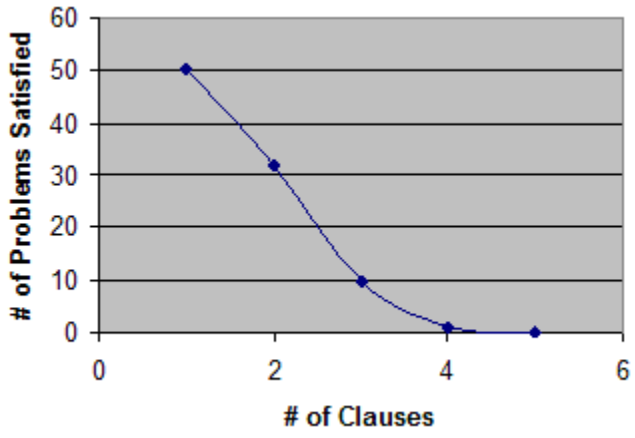

Figure 12. Version 3 Freezing Zone

Version 3 is our second quantum algorithm for solving *k*-QSAT problems—it uses permutators. It also has no null vectors, because it only has potential for low energy solutions, which are also of interest to us because they are approximately null vectors. Version 3 was actually the most correct version, until its twin version, Version 6. This is

because, as we found out later, we were actually supposed to keep all 8 terms, rather than throw out some terms just to get null vectors, when we generated the forbidden matrix. We picked the 0.0055 threshold by trial runs, since all 50 one-clause problems are suppose to be all solvable. As mentioned earlier, our original algorithm was wrong, and we were really surprised because the number of solvable instances never dropped to 0, indicating that there was no freezing point. But in the end we found that this error was due to faulty implementation of the permutators. After fixing this error and conducting new experiment, the resulting data no longer oscillates in a sinusoid that refuses to decline. Version 3 turns out to work approximately as well as Version 6, even though at first we thought Version 6 was much better. At first we were concerned that the random number generating feature of Java might depend on disk space, but actually the data used in this experiment does not depend on the memory space of the computer used. We tried running our programs on both desktop and laptop computers and we obtained pretty much the same results. Our desktop is just slower than our laptop, because it has less memory, but it produces the same data. Our ratio for Version 3 was found to be 0.714.

Table 3.

Version 4 Success Count

# of Clause	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
# Satisfied	**49**	**49**	**41**	**39**	**23**	**17**	**8**	**7**	**7**	**4**
# of Clause	**11**	**12**								
# Satisfied	**1**	**0**								

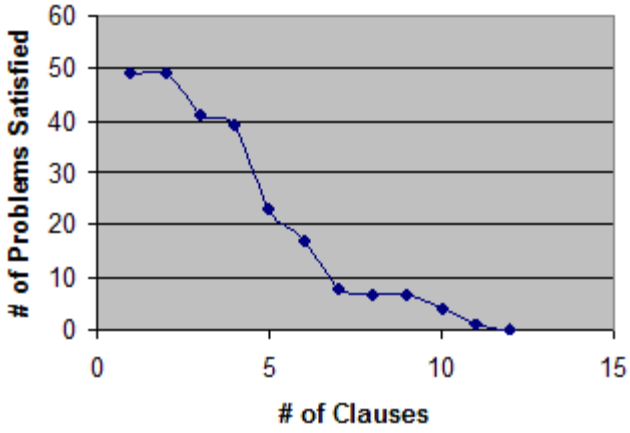

Figure 13. Version 4 Freezing Zone

Table 4.

Version 5 Success Count

# of Clause	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
# Satisfied	**50**	**50**	**48**	**41**	**32**	**28**	**21**	**13**	**7**	**3**
# of Clause	**11**	**12**								
# Satisfied	**1**	**0**								

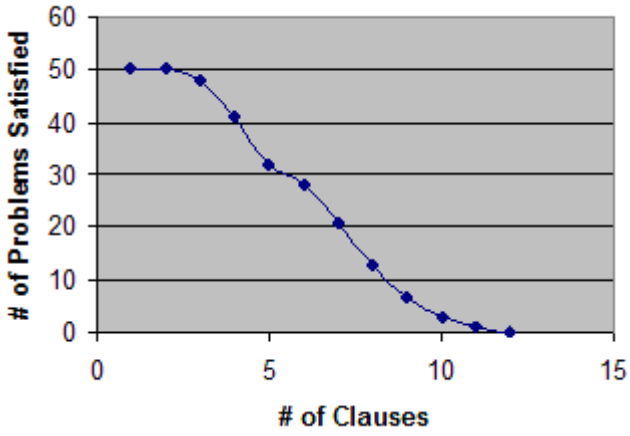

Figure 14. Version 5 Freezing Zone

Version 4 is a quantum algorithm for finding null vectors; it is adapted from Version 3 as Version 5 is a quantum algorithm adapted from Version 2. Version 4 and 5 both seem to work about the same for finding null vectors, after fixing the permutators and division by zero problems. The bumpiness of the Version 4 graph is due to randomness of the algorithm and data instability. We know for a fact that Version 4 has potential to generate a smooth-looking curve just like Version 5, and we believe that it is pure coincidence that it turned out not smooth-looking instead in Figure 13. As one would expect, Version 4 should ideally give the same exact data as Version 5, since there is no difference in the forbidden matrices that were used for the two versions. Based on the freezing point ratio that was observed, Version 4 and 5 are legitimate quantum algorithms according to (Bravyi, Moore & Russel, 2014). At the time of writing these two algorithms, we did not realize that we were supposed to keep all 8 terms, and not randomly throw out a few of the terms; we should use a threshold to get approximate null vectors, rather than try to find true mathematical null vectors. The point of these two versions is to compare the function of the permutators to the function of the "cut-and-paste of strings method" (See sections 3.6 and 3.9) because both methods should give us the same results. It is also good for comparing the freezing point with true 0-energy solutions, as in Version 4 and 5, with low energy threshold solutions as in Version 3 and 6 which had ratios 0.714 and 0.571 respectively. For both Version 4 and Version 5, the ratio was 1.714.

Table 5

Version 6 Success Count

# of Clause	**1**	**2**	**3**	**4**	**5**

# Satisfied	**48**	**34**	**6**	**0**	**0**

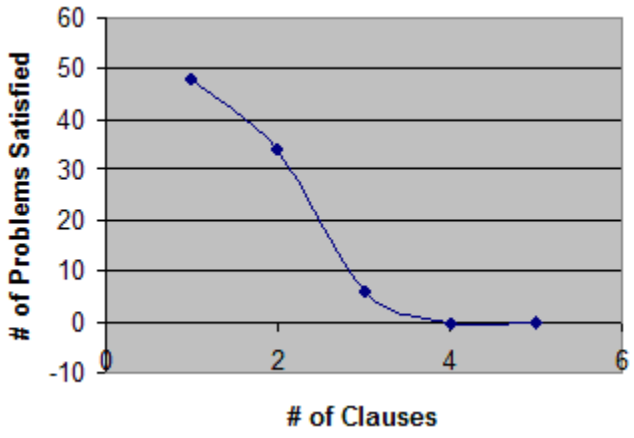

Figure 15. Version 6 Freezing Zone

Version 6 is adapted from Version 3; the only difference is that it allows repeating variables in the random generation process of clauses. Version 6 has a freezing point of 0.571, which is actually slightly less than 0.818, but if we set a slightly higher threshold, we can probably get the ratio to be 0.818. As we can see by testing out different thresholds, the larger the threshold, the more null vectors there will be, so to get the freezing point, you should use as small a threshold as possible. But 0.003 was too small because some one-clause problems were unsolvable. We used a threshold of 0.0055 instead, which was found using the idea that all one-clause problems are barely all solvable. To conclude, our Version 6 is the closest to our classical algorithm for solving k-SAT problems. In comparing Version 6 and Version 3, the only difference is that Version 6 uses repeating particles and Version 3 does not. We were satisfied to note that both versions work approximately equally well, because quantum clauses with repeating particles have same type of matrix representations as quantum clauses without repeating particles. Both types of clauses do not have any

true null vectors in the forbidden matrix and only have potential for low-energy solutions. The slight difference in the freezing point ratio of Versions 3 and 6, given in the data is likely due to randomness of the algorithm, and data instability.

Table 6.

Models Success Count

Type of Physical Quantum Model	**Data**
Quantum Ising Model	Clausesize =2 # of variable=5 (crashes for >5) # of clauses=3(crashes for >3) # of instances=5(crashes for >5) J = 0.5 G= 0.35 Threshold=0.06 (0.66) # success=0-3
Quantum Antiferromagnetic Heisenberg Model	Clausesize =2 # of variable=6 (crashes for >6) # of clauses=4 (crashes for >4) # of instances=3 (crashes for >3) Threshold=1.5 # success=2 or 3

We now describe our Ising and Antiferomagnetic Heisenberg Models experiments documented in Table 6. For the first of these experiments we used a very rough and inaccurate way to determine whether there are low energy eigenvalues. We just used the determinant, because the determinant is the product of all the eigenvalues. This is

imprecise at treating the case where some elements in diagonal are much larger than the threshold while others are much smaller than the threshold; it results often in the answer that there are no null vectors, when there are null vectors. So we gave up on this method, and used a threshold, and a Gaussian elimination algorithm instead. As we noted in our data table for the two physical models, above a certain number in number of variables, number of clauses, and number of instances, the computer used to conduct the experiment always crashed. As a result, our experiments on the tube, torus and twisted torus configurations were mostly unsuccessful. We are able to get it to work for 6-particle tube configuration but not for 9-particle tube, torus and twisted torus configurations.

For Versions 3 and 6, Ising Model, and Antiferromagnetic Heisenberg Model, we used thresholds. The key to setting the appropriate threshold is: the bigger the threshold, the more null vectors we get, which is confirmed by our data. For the Ising Model and Antiferromagnetic Heisenberg Model, when there is only one clause, there are no null vectors, but when two or more clauses add together, it is possible to enhance and degrade some of the terms so that there are null vectors, because there are negative elements in the Aniferromagnetic Heisenberg Model forbidden matrix. This basically explains why more clauses sometimes do not always mean less success count at finding a null vector for the Antiferromagnetic Heisenberg Model.

Chapter 5

Conclusion

Based on this experiment, we have seen that it takes much fewer clauses to get no solution to a *k*-QSAT problem on *n* variables than it is to get no solution for a *k*-SAT problem on *n* variables. For the classical algorithm, it took about 70 clauses to freeze, while for the quantum algorithm it took about 4 clauses to freeze (ratio 0.571). We have also seen that this kind of simulation, using the classical computer to simulate a quantum system is inefficient and can only be done for small register sizes. It is open to future work to find a better way to understand the quantum computer with or without the classical computer.

We had to overcome a lot of errors in our thinking to develop our quantum simulator codes. We determined a freezing point value for all of our classical and quantum algorithms for solving SAT problems. We discovered that the oscillation and lack of freezing point in our original data were due to faulty permutator matrices. We confirmed the fact that Version 4 and Version 5 work in exactly the same manner. Lastly, we verified that repeating particle (allow clauses on 2,2,3 or 3, 3, 3) clauses are equally constraining as non-repeating particle clauses (only clauses where all *k* variables are different). We found out why the freezing point for the classical algorithm, Version 1, did not fall between 3.52 and 4.49. We found out that the Ising and Antiferromagnetic Heisenberg Models are hard to simulate well on classical computers. Based on our experiments, the boundary between almost surely satisfiable *k*-QSAT instances and almost surely unsatisfiable *k*-QSAT instances in terms of clause to variable numbers ratio seems more complicated than in the

classical case. Of course, simulating what is supposed to happen is different than getting correct results in an actual lab. We expect Quantum Hamiltonian Complexity to be a fruitful area of future research.

References

Aaronson, S. (2013). Quantum Computing Since Democritus. New York, USA: Cambridge University Press.

Beaudrap N. & Gharibian, S. (2015). A linear time algorithm for quantum 2- SAT. USA: 31st Conference on Computational Complexity.

Bravyi, S., Moore C. & Russel, A. (2014). Bounds on the quantum satisfiablility threshold. Santa Fe Institute.

Gharibian, S., Huang, Y., Landau, Z., & Shin S.W. (2014) Quantum Hamiltonian complexity USA: Foundations and Trends in Theoretical Computer Science.

Wikipedia. (2017, August 10) Grover's algorithm. Retrieved from https://en.wikipedia.org/wiki/Grover%27s_algorithm

Haeffner, H. Ion Trap. Retrieved from https://qudev.phys.ethz.ch/content/courses/QSIT07/qsit21_v1_2page.pdf

Haeffner, H. (September 25, 2008). Quantum computing with trapped ions. University of California Berkeley: Physics Reports 469.

deVries.(2004) jQuantum-Quantum Computer Simulator. Retrieved from http://jquantum.sourceforge.net/

Maslov, D. (2017). Basic circuit compilation techniques for an ion-trap quantum machine. IOP Publishing Ltd.

Moore, C., & Mertens, S. (2011) The Nature of Computation. Oxford University Press.

Nielsen, M., & Chuang, I. (2010) Quantum Computation and Quantum Information. New York, USA: Cambridge University Press.

Wikipedia. (2017, August 10) Shor's algorithm. Retrieved from https://en.wikipedia.org/wiki/Shor%27s_algorithm