University of Cincinnati

Date: 12/14/2010

I, Stephen Shary , hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:

Java Simulator of Qubits and Quantum-Mechanical Gates Using the Bloch Sphere Representation

Student's name: **Stephen Shary**

This work and its defense approved by:

Committee chair: Marc Cahay, PhD

Committee member: Karen Davis, PhD

Committee member: Philip Wilsey, PhD

1340

Java Simulator of Qubits and Quantum-Mechanical Gates Using the Bloch Sphere Representation

A thesis submitted to the Division of Research and Advanced Studies of the University of Cincinnati in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in the Department of Computer Engineering of the College of Engineering and Applied Science

December, 2010

by

Stephen Shary

B.S., University of Cincinnati of Computer Engineering

2004

Committee Chair: Marc Cahay, Ph.D.

Abstract

One of the most promising paradigms for the development of novel high-speed and energy efficient devices is spin electronics or spintronics. It is based on the simultaneous manipulation of the electron spin and large degrees of freedom. It offers the possibility of developing electronic devices based on the control of the electron spin. The spin polarization of a single electron can exist in a coherent superposition of two orthogonal spin polarizations (i.e. mutually anti-parallel spin orientations) for a relatively long time without losing the phase coherence. The charge degree of freedom, on the other hand, loses phase coherence much faster. Therefore spin has become the preferred vehicle to host a quantum bit (or "qubit") which is a coherent superposition of two orthogonal states representing classical logic states of 0 and 1. The potential application of spin manipulation to a scalable quantum logic processor has led to the field of quantum computing.

To date, several physical quantum computers have been proposed [8, 9, 12, 21] which all require appropriate mechanisms to create, manipulate and measure individual spins. Each operation can be mapped to the action of a quantum-mechanical system acting on the spin state of the electron also known as the qubit state. The Bloch sphere is a useful tool to represent the actions of various quantum-mechanical operators on a spinor because it provides a visual representation of the qubit state evolution. Most importantly, it provides a link between the rather abstract concept of a spinor and the more intuitive way (although not rigorous) of thinking of the spinor of the electrons being associated with an intrinsic magnetic moment.

In this thesis, a simulation software is built to provide a visual representation of a quantum state or qubit based on the Bloch sphere representation. This software uses the Java language and libraries to provide a multi-platform simulator that can be quickly distributed and viewed using the Java web-start technology. This simulator shows the different qubit

states, their representation in both the two dimensional complex plan along with the bra-ket representation. It provides the ability to visualize simple basic operators representing the action of quantum gates. It allows the user to enter simple operations which can be represented by the action of 2x2 matrices. It also features some more complex functionality important in spintronics including the Larmor precession and the spin flip process under the combined action of a constant and rotating magnetic field as described by the Rabi formula.

Acknowledgments

This has been a long process for me to be able to complete this thesis. There were many times where I was not sure if I would actually have the persistence and endurance to finish this. I would like to thank my advisor, Dr. Cahay, for all of his work and patience with me to complete this project. Though this started as a class project, he encouraged me to pursue and expand this to what it is today. He has been very patient with my pace to complete this. He spent his extra time outside of his already incredibly packed schedule to pursue this with me. He encouraged me both to start this project, form it into a thesis, and now to finish it. He helped me with many of equations that govern the quantum evolution due to magnetic fields. I was honored to work under him. He literally wrote the book on this subject.

I secondly want to thank my wife, Lucia. She is a great source of encouragement and a stable person to lean against. She has been very supportive even when I am not to her. I am so glad to have her as my partner. So much of who I am has been strengthened being with her. I want to thank my parents that have understood the balance between encouragement and expectation. They are so supportive and yet have always nudged me to finish projects that I start. I want to thank so many of the guys from my home church: Rob Stower, Matt Bair and Bertholt Schroeder. They have provided great friendships that are able to distract me when I need to relax. So much of their attitude is serving and supporting others. I feel really lucky to have such a base of friends during this.

I want to thank Dr. Wilsey and Dr. Davis. I have known both of them personally and throughout my undergraduate and graduate experience. Dr. Wilsey has pushed me hard to move forward and not settle short. I would also like to thank Dr. Franco for teaching me the foundation of Java and really exposing me to security issues and principles.

Contents

Chapter 1

Introduction

1.1 Motivation

Quantum mechanical theory has been discussed for nearly a century with many additions, tests and advances. Not until about 20 years ago did the idea of applying the field to the area of computing become a possibility [16]. With technology continuing to advance, we have begun to test and apply those theories on real physical systems. The fields of spintronics and quantum computation are now rapidly expanding. The theory of solving classical problems in very efficient manners has been shown for some time [18, 10, 35]. Some classical problems with exponential complexity can be solved in polynomial time using quantum algorithms. Bacon and Van Dam describe the growth of quantum algorithms in problems such as growth in quantum random walks, NAND trees, hidden symmetries, the hidden sub-group problem, and physical simulation of quantum mechanical systems[3].

Some of the basic concepts of quantum computation can be difficult to grasp. Expensive equipment is needed to probe physical systems at the quantum level to observe their interaction and quantum states. Interaction at the quantum level can become very complex quickly with many outside factors that affect the measurement. It is only through repeating

an experiment that we can obtain a probability distribution of physical observables. The concept of a repeatable experiment with different outcomes is strange. Many times, it is not intuitively understood.

This simulator was created to provide an easy tool to visually describe simple quantum states and also apply operators to those states. This software simulator can be installed on a computer easily with little configuration needed. It is based on a web host as the distribution mechanism that can deploy off-line versions of the application. It uses the Bloch sphere theory to visually describe the state of a quantum bit or qubit. Operators can be entered or popular operators can be used to show how the state of a qubit is affected by the operators. The 3D Bloch sphere also allows an animation of the qubit state to be shown over time under the influence of constant or time-dependent, spatially uniform magnetic fields. It also provides the ability to record and playback different operators so the time evolution of the qubit can be displayed.

1.2 Survey of Current Simulators

Current simulators provide the ability to describe the state of one or more qubits and how they are affected by outside magnetic fields and various quantum gates. This section briefly describes existing simulators that describe the physical systems, networks of quantum operators (or quantum circuits) and Bloch sphere simulations of specialized states.

The first group of simulators focuses on the analysis of a real-world implementation of a quantum interaction. There are different types of physical systems that can manipulate and measure quantum interference and states. The first type uses optical lasers, beam splitters and phase shifts to investigate the quantum effects. This is achieved by introducing a single-photon representation of several quantum bits, building on the equivalence between traditional linear optics elements such as beam splitters or phase shifters and one-bit quan-

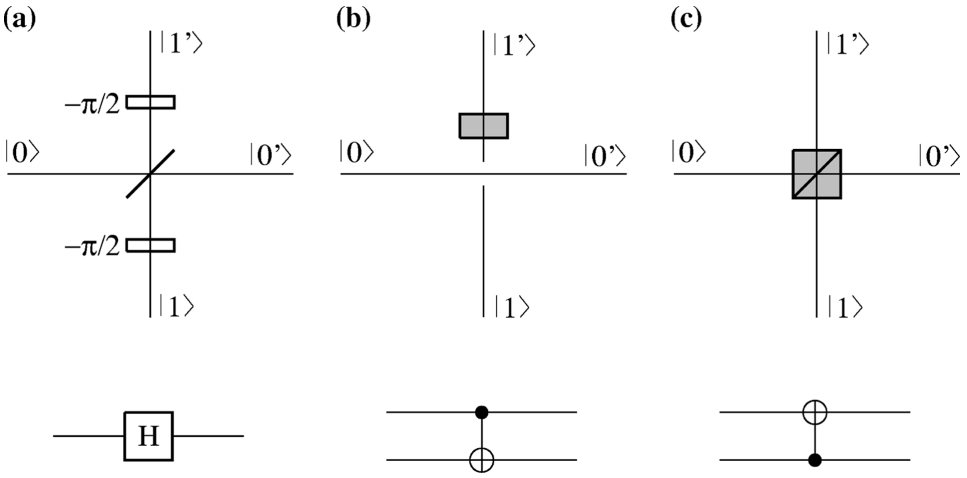

Figure 1.1: Optical configurations and their logical quantum operators described in [7]
.

tum gates[7]. Cerf et al. focuses on building the basic quantum operators from simple optical devices and completes a demonstration of how it can be used to construct different quantum circuits. In Fig. 1.1, there are three examples of how optical devices can be used to simulate quantum operators. The top half shows the optical setup and the bottom shows the corresponding quantum operator in circuit form. Part (a) shows how two beam splitters can be used to create the Hadamard gate. Part (b) shows how a C-NOT gate can be created from a polarization rotator and part (c) shows how a beam splitter can be used to create a C-NOT gate with inverted control lines.

Another interesting simulator looks at the numerical analysis of simulations using quantum dots [23]. The approach uses laterally coupled quantum dots and shows how they can be used in a semiconductor heterostructure to create a series of quantum gates that can be manipulated and measured. The authors use different modeling equations to show the effects on two quantum dots in close proximity. Rather than working on the correlation to abstract quantum operators, this work focuses on calculating expected eigenenergies due to the change of different biases. The model is able to reproduce the electron charging behavior of the quantum dots. With the rising popularity of nano structures, this shows how a

quantum gate (or a series of them) can be used to demonstrate quantum effects and how their states can be manipulated.

Poyatos et al. look at the characterization and the quality parameters of a two qubit system [32]. They describe how any physical implementation that can be fully characterized or simulated can be quantified into four different parameters to measure the action of the quantum gate. The four characteristics are the "quantum fidelity", "quantum degree", "gate purity", and "entanglement capability." As an example, they show how the simulation and characterization of the ion trap can be formalized into these parameters. This expands from beyond previous works by categorizing and comparing different physical quantum systems. It provides a benchmark to rate the usability of a quantum system.

The second group of quantum computer simulation moves from the physical implementation to an architectural and computational realm. Quantum operators are interesting in how they affect the complex state of a qubit, but the computation significance is when multiple qubits can become entangled to produce an interesting outcome. The circuit diagrams use the quantum operators with inputs and outputs on the left and right. The typical flow from the starting state to the final state is from left to right. Figure 1.2 from [15] shows some of the elementary quantum circuits. More complex diagrams can be created to implement interesting algorithms. Figure 1.3 taken from [26] displays a quantum circuit using the Hadamard, σ_x and C-NOT gates to complete Grover's search algorithm in [18].

The quantum circuit simulators that have been created run into problems of complexity. Quantum system simulators require large amounts of classical computation. In fact, a super-polynomial amount of memory and time is needed to simulate quantum systems [19]. Viamontes et al. designed their quantum, circuit simulator for optimal performance [26]. They look at the efficient implementation of Grover's search algorithm with a polynomial number of qubits. Another approach uses the statistical inference of quantum systems using a Monte Carlo algorithm to closely approximate 2D correlated systems [11]. Even then the

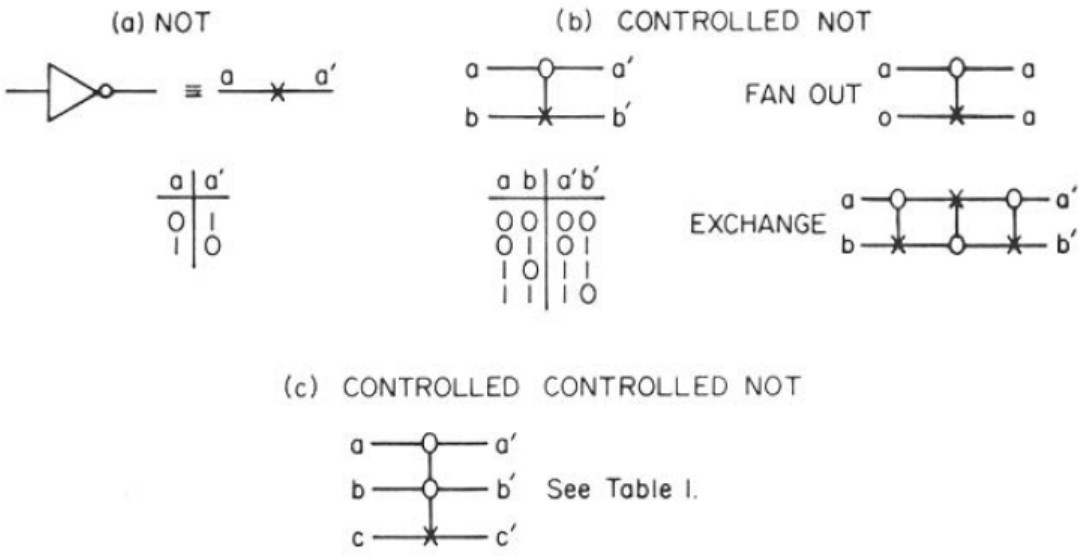

Figure 1.2: A simple diagram of reversible computer gates in the standard logical notation and also in the quantum circuit notation [15].

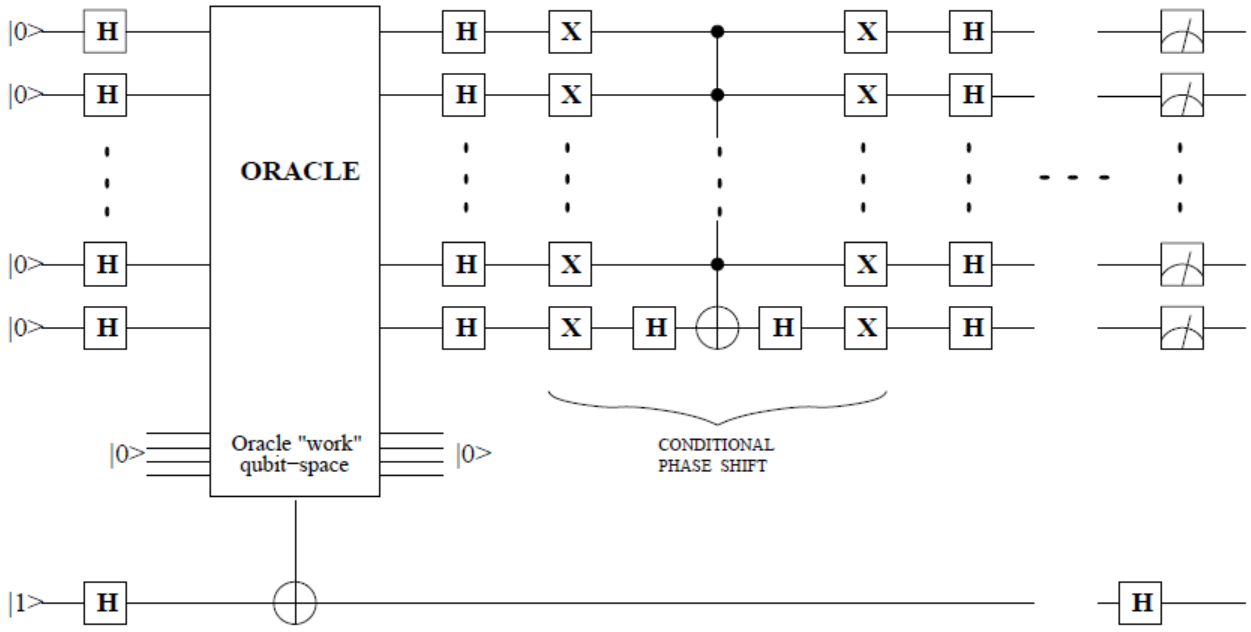

Figure 1.3: An advanced quantum circuit diagram that implements Grover's search algorithm [26].

analysis was completed on supercomputers with distributed memory and vector pipelines to allow for quick computations. More current simulators look to use distributed systems that form grid systems and super computers [6, 22]. Their focus is on the ability to simulate larger quantum systems.

Up to this point, the simulators that have been described look towards accurately calculating the physical state of a quantum implementation simulating the outcome of the operation of a series of gates set in a network. The transformation of a quantum state can be hard to grasp because there is no simple way to explain it. There have been systems developed to show the states of qubits through graphs and lines [13, 14] but they have not been very popular. There are some simulators describing the state of the quantum gates and showing the evolution of their state visually through the Bloch sphere. These simulators give an intuition of quantum transformations which can help lead to new discoveries and understandings [25]. One implementation is a package of libraries used in Matlab to visualize, measure, and transform quantum states [24]. The foundation of Matlab allows the user to manipulate the states through a large number of numerical transformations. The package allows the display of qubit states and the states of many qubits through the density matrix. The display though requires the user to understand the bra-ket notation and only it only provides the ability to display a static quantum state. In [4], the authors use the Bloch sphere to show the analysis of two photon systems (known as Raman systems) and how their state (mixed and pure) can be displayed on the Bloch sphere so that "the effects of decay, detuning, and optical pumping to be quickly intuited." Though there are simulators that use the Bloch sphere to describe states of qubits, they are limited to specific quantum scenarios or they have a high learning curve. There is a missing tool to easily manipulate qubit states to gain and understanding of quantum properties through the representation of the Bloch sphere.

Chapter 2

Background

2.1 Introduction to the Bloch sphere

The Bloch sphere is a useful tool to represent a quantum bit (or qubit for short) or spinor state and also the action of various quantum-mechanical operators acting on it. it provides a link between the rather abstract concept of the spin and the more intuitive way of thinking of a spin as a tiny magnet. This section will show that the Bloch sphere provides a very useful way to depict the action of a uniform magnetic field on a spin.

The Bloch sphere provides a visual representation of a two level system pure state. There are multi-level states that may exist in quantum theory. In the field of spintronics and quantum computation, the two level state is popular because there is a direct translation to to classical computing concept of the 0 and 1. As we will see later, the interesting aspect of quantum theory is that the two states of 0 and 1 may be represented by two states that, though they are distinct within the same system, can correlate to each other and affect each other. This phenomenon is powerful in quantum computation because it allows one operation on two or more states to be affected at the same time with only the work of the one operation. We can also observe how the state of the spinor evolves as it is affected by the

action of magnetic fields. One interesting case is when two magnetic fields, one static and one rotating perpendicular to each other, can act simultaneously to cause the probability of the spinor to "flip" or change state from the north pole to south pole (on the Bloch sphere). The probability of the spin flip was first derived by Rabi [33]. One of the more popular application of the Rabi theories is Nuclear Magnetic Resonance (NMR).

2.1.1 Quantum state notation: the "qubit"

The term "qubit" is short-hand for a quantum bit. This is the basic unit that is processed by a quantum computer. Like a classical computer that takes an input and maps to an output of classical bits (0 and 1), a quantum computer has inputs and outputs as qubits. We represent the qubit as the spin state expressed as two coherent orthogonal superpositions. Mathematically, we describe the qubit as a two dimensional complex vector that represents a pure state in the Hilbert space $\mathfrak{H}$. This state contains two complex values that can made analogous to the two states in classical computing (0,1). In the area of spintronics, we view these two states as spin up and spin down. The most commonly used basis set to describe the state of a qubit is given by

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \tag{2.1}$$

and

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.2}$$

These two basis vectors can be used to describe any general, pure state qubit as follows, $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ where α, β are complex numbers. It would seem at this point that $|\psi\rangle$ can have both values of $|0\rangle$ and $|1\rangle$. This is not entirely true. What we find is that if we take a measurement along the z-axis, we find that the measured value comes out as a $|0\rangle$ sometimes and other times as a $|1\rangle$. The values of α and β are the probability amplitude

for the qubit to be in the $|0\rangle$ or $|1\rangle$, respectively the qubit is in state $|0\rangle$ with a probability of $|\alpha|^2$ and in state $|1\rangle$ with a probability of $|\beta|^2$. Being probabilities, they must satisfy the condition $|\alpha|^2 + |\beta|^2 = 1$. This interesting constraint of the two probabilities allows us to map the state of the qubit using a sphere of radius 1. We will discuss later in this chapter how the mapping to the unit sphere or Bloch sphere is completed.

From now on, we describe state vectors using the Dirac notation. This notation allows vectors or states to be displayed as both "kets" and "bras". We have displayed the qubit states in kets, but they may also be described as a bra. The bra is vector displayed horizontally where each element in the vector is the complex conjugate of the element in the ket. For a general state give by the ket

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \tag{2.3}$$

we have the corresponding bra

$$\langle\psi| = \begin{pmatrix} \alpha^* & \beta^* \end{pmatrix}. \tag{2.4}$$

Next we discuss how the qubit is represented on the Bloch sphere.

2.1.2 Bloch Sphere

The Bloch sphere is a three dimensional representation of a qubit on a unit sphere. Since the qubit is a coherent superposition of two states of $|0\rangle$ and $|1\rangle$, we know that the qubit described as a vector will always have a length of 1. With the qubit ($|\psi\rangle$) being described as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, each α and β are complex, we can write α and β as $\alpha = \alpha_{real} + \alpha_{imag}i$ and $\beta = \beta_{real} + \beta_{imag}i$. If we convert these to polar coordinates we get

$$\alpha = r_\alpha(cos\theta_\alpha + sin\theta_\alpha i), \tag{2.5}$$

and

$$\beta = r_\beta(cos\theta_\beta + sin\theta_\beta i). \tag{2.6}$$

Using Euler's formula, we get

$$\alpha = r_\alpha e^{i\theta_\alpha}, \tag{2.7}$$

and

$$\beta = r_\beta e^{i\theta_\beta}. \tag{2.8}$$

Applying this to $|\psi\rangle$, we obtain the equation

$$|\psi\rangle = r_\alpha e^{i\theta_\alpha} |0\rangle + r_\beta e^{i\theta_\beta} |1\rangle . \tag{2.9}$$

We are now going to manipulate this equation to show that it maps to the unit sphere. We first start by multiplying the $|\psi\rangle$ by $e^{-i\theta_\alpha}$. This is possible because we know that $|\alpha|^2 + |\beta|^2 = 1$ and

$$|\alpha e^{i\theta_\alpha}|^2 = |\alpha|^2 e^{i\theta_\alpha} e^{-i\theta_\alpha} = |\alpha|^2. \tag{2.10}$$

Multiplying $|\psi\rangle$ by $e^{-i\theta_\alpha}$ gives us

$$|\psi\rangle = r_\alpha |0\rangle + r_\beta e^{i\theta_\beta - \theta_\alpha} |1\rangle . \tag{2.11}$$

We substitute $\phi = \theta_\beta - \theta_\alpha$ and get

$$|\psi\rangle = r_\alpha |0\rangle + r_\beta e^{i\phi} |1\rangle . \tag{2.12}$$

We can show that this matches the unit sphere. We convert $r_\beta e^{i\phi}$ back to Cartesian coordinates

$$r_\beta e^{i\phi} = (x + iy). \tag{2.13}$$

Since $|\alpha|^2 + |\beta|^2 = 1$, substituting the values of α and β above, we get

$$|r_\alpha|^2 + |x + iy|^2 = 1, \tag{2.14}$$

or

$$r_\alpha^* r_\alpha + (x + iy)^*(x + iy) = 1, \tag{2.15}$$

or

$$r_\alpha^2 + (x - iy)(x + iy) = 1, \tag{2.16}$$

and finally

$$r_\alpha^2 + x^2 + y^2 = 1. \tag{2.17}$$

This is the equation for the unit sphere where $r_\alpha = z$. Now we use the mapping of Cartesian coordinates to polar coordinates and we get

$$\begin{aligned}
|\psi\rangle &= z\,|0\rangle + (x + iy)\,|1\rangle\,, \\
|\psi\rangle &= cos\theta\,|0\rangle + [sin\theta cos\phi + i(sin\theta sin\phi)]\,|1\rangle\,, \\
|\psi\rangle &= cos\theta\,|0\rangle + sin\theta(cos\phi + isin\phi\,|1\rangle\,, \\
|\psi\rangle &= cos\theta\,|0\rangle + e^{i\phi} sin\theta\,|1\rangle\,.
\end{aligned}$$

At this point, we have a mapping from Cartesian coordinates to the polar coordinates. But this mapping is not a one-to-one mapping, it is actually as "a 2 to 1 homomorphism of SU(2) on SO(3)."[17] There are multiple angles that can be used to describe the Cartesian coordinates. If we start with a point with the spherical coordinates $R = (1, \theta, \phi)$, adding the

angle of π to each angle will give us the following:

$$|\psi\rangle = \cos(\theta+\pi)\,|0\rangle + e^{i(\phi+\pi)}\sin(\theta+\pi)\,|1\rangle \tag{2.18}$$

$$= -\cos(\theta)\,|0\rangle - e^{i\phi}e^{i\pi}\sin(\theta)\,|1\rangle$$

$$= -\cos(\theta)\,|0\rangle - e^{i\phi}\sin(\theta)\,|1\rangle$$

$$|\psi\rangle = -1\left(\cos(\theta)\,|0\rangle + e^{i\phi}\sin(\theta)\,|1\rangle\right). \tag{2.19}$$

The "-1" is what we can categorize as a global phase factor, removing any effect on the state of the qubit. If we choose the angle θ, which has a range of $(0 \leq \theta \leq 2\pi)$ and confine it to a range of $(0 \leq \theta' \leq \pi)$ where $\theta' = \frac{\theta}{2}$, then we effectively create a one-to-one mapping.

Shown on the next page is the image of the Bloch sphere from [30].

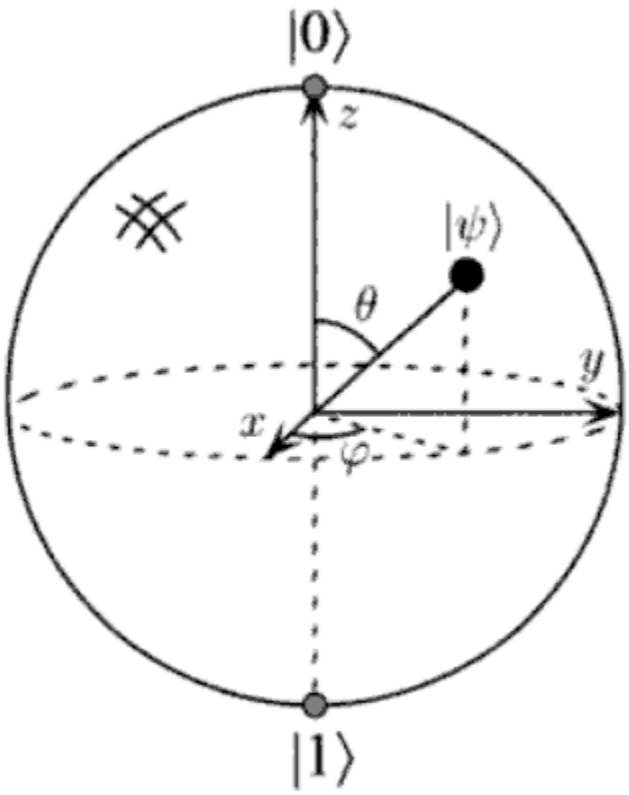

Figure 2.1: The Bloch sphere showing the θ and ϕ angles to place the qubit [30].

2.2 Operators

Operators are mappings from one Hilbert space to another Hilbert space. All quantum operators representing the action of quantum gates must be unitary, i.e.,

$$MM^{\dagger} = I, \tag{2.20}$$

where I is the 2x2 identity matrix and the dagger represents the action of the transpose operation and the complex conjugation operation. $M^{\star}$ is the complex conjugate operation on the operator where each element multiplies the imaginary part of the element by -1. This is important because it shows that the operator is a rotation of unit modulus. Operators representing physical observables must also be Hermitian. An operator is Hermitian if after taking the complex conjugate and transpose of that operator, we get back the original operator. We represent the adjoint operation as a dagger. If an operator is equal to its adjoint, then it is Hermitian. An operator is Hermitian if $M = M^{\dagger}$. Here we show the mapping of the operator elements in relation to their original value:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{\dagger} = \begin{pmatrix} a^{*} & c^{*} \\ b^{*} & d^{*} \end{pmatrix}. \tag{2.21}$$

We can see that if an operator is both Hermitian and unitary, then applying the operator twice will map the qubit back to its original state. Additionally, if an operator is unitary, Shankar states that "if one reads the columns of an n x n unitary matrix as components of n vectors, these vectors are orthonormal. In the same way, the rows may be interpreted as components of n orthonormal vectors"[34]. A simple example of this is the 2x2 identity matrix. This matrix is both Hermitian and unitary. The following sections will show the set of Hermitian operators that form the rotation matrices around the three axes of the Bloch

sphere. These three rotation matrices are known as the Pauli matrices.

2.2.1 X operator

The X operator is one of the three Pauli matrices. It is also known as the NOT operator because it flips the $|0\rangle$ qubit to the $|1\rangle$ qubit and vice versa. It is Hermitian and unitary. It is also referred to as the σ_x operator. On the Bloch sphere, it corresponds to a π rotation around the x-axis. Knowing its mapping, we can derive the operator from the sum of the outer products that represent the matrix. The outer products

$$\langle 0| \, |1\rangle = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \tag{2.22}$$

and

$$\langle 1| \, |0\rangle = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \tag{2.23}$$

form the full expression of the σ_x operator as

$$\sigma_x = \langle 0| \, |1\rangle + \langle 1| \, |0\rangle = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \tag{2.24}$$

This is Hermitian, i.e. $\sigma_x = \sigma_x^\dagger$, as easily checked

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^\dagger = \left(\left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^* \right)^T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \right. \tag{2.25}$$

2.2.2 Y operator

The Y operator is another of the three Pauli matrices. Like the X operator, it is both Hermitian and unitary. This is also referred to as the σ_y operator. On the Bloch sphere, it is the operator that rotates the qubit around the y-axis by an angle of π. The 2x2 matrix form of the σ_y operator is given as:

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \tag{2.26}$$

which is Hermitian since $\sigma_y = \sigma_y^\dagger$. Indeed

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}^\dagger \tag{2.27}$$

$$= \left(\left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}^* \right) \right)^T \tag{2.28}$$

$$= \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}^T \tag{2.29}$$

$$= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}. \tag{2.30}$$

2.2.3 Z operator

The last of the three Pauli matrices is the Z operator which is also Hermitian and unitary. This is commonly referred to as the σ_z operator. On the Bloch sphere, it is equivalent to a rotation around the z-axis by an angle of π. The 2x2 matrix form of the operator is given

as:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{2.31}$$

which is indeed Hermitian since $\sigma_z = \sigma_z^\dagger$

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}^\dagger \tag{2.32}$$

$$= \left(\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}^* \right)^T \tag{2.33}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}^T \tag{2.34}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{2.35}$$

2.2.4 Properties of the Pauli matrices

Since all Pauli matrices are both unitary and Hermitian, applying the operator twice on a qubit ($MM|\psi\rangle$) will give back the original qubit, i.e., $M^2 = I$ where M is either σ_x, σ_y, or σ_z.

Another interesting property about the Pauli matrices involves their product. Taking the product of σ_x, and σ_y, we get

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} \tag{2.36}$$

$$= i \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.37}$$

$$= i\sigma_z. \tag{2.38}$$

This is equivalent to the qubit $|\psi\rangle$ being rotated around the x-axis by an angle of π and then around the y-axis by and angle of π. If we were to apply the σ_z operator to the original state of $|\psi\rangle$, we would see that it moves it to the same spot. There is the interesting "i" factor that exists in the equation. However, this has no measurable effect on the state of the qubit.

Similarly, we can calculate all the different products of the Pauli matrices.

$$\sigma_x\sigma_y = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} = i\sigma_z, \tag{2.39}$$

$$\sigma_y\sigma_x = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix} = -i\sigma_z, \tag{2.40}$$

$$\sigma_y\sigma_z = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} = i\sigma_x, \tag{2.41}$$

$$\sigma_z\sigma_y = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix} - i\sigma_x, \tag{2.42}$$

$$\sigma_x\sigma_z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = i\sigma_y, \tag{2.43}$$

$$\sigma_z\sigma_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = -i\sigma_y. \tag{2.44}$$

The product of any two Pauli matrices is the third Pauli matrix multiplied by the phase factor of i and $-i$. This does not change the probability distribution associated to the α and β values. Additionally, since the phase factors are i and $-i$, we can rewrite them as $e^{\frac{i\pi}{2}}$ and $e^{i\frac{3\pi}{2}}$. With the standard description of qubit as $|\psi\rangle = e^{i\phi}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\gamma}sin\frac{\theta}{2}|1\rangle\right)$, the constant associated with the operator will only affect the phase factor on the qubit, $e^{i\phi}$.

2.2.5 H operator

The Hadamard operator is the also known as the $\sqrt{NOT}$ or the square root of the NOT operator. This is because visually the operator rotates the qubit around the y-axis by $\pi/2$ followed by a rotation along the x-axis by π. The H operator is defined as

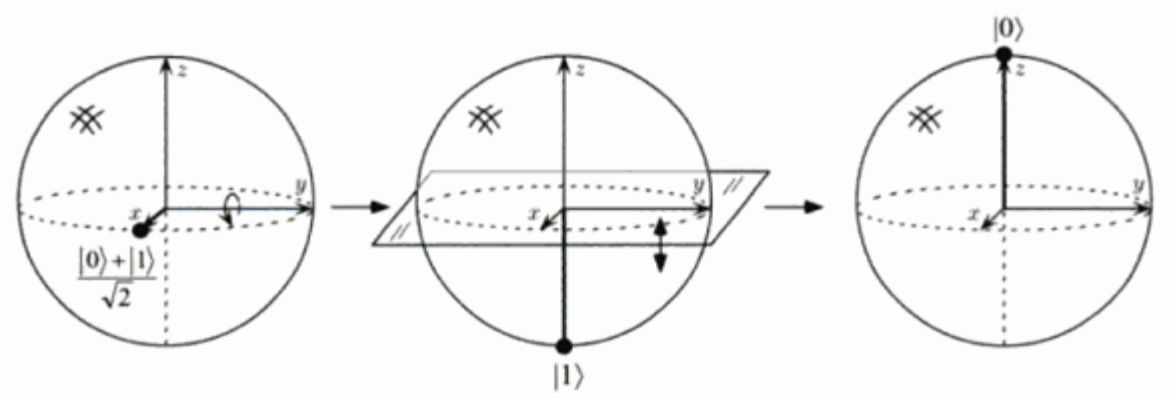

Figure 2.2: The visual effect of the Hadamard operator on the qubit $\psi = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ [30]

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{2.45}$$

which is unitary and Hermitian. Even though it is called the square root of NOT operator, applying it twice does not create the NOT operator seen in classical computation. Since the operator is both unitary and Hermitian, applying the operator twice is the same as applying the identity operator and does not affect the state of the qubit. Figure 2.2 is an illustration of the Hadamard operator [30].

2.2.6 S and T operator

The S and T operators are two operators that define a partial spin around the Z-axis. The S operator is defined as

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \tag{2.46}$$

and the T operator is defined as

$$T = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i \end{pmatrix}. \tag{2.47}$$

We can make the angle of rotation appear explicitly if we redefine S and T using Euler's relation. In that case

$$S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{pmatrix}, \tag{2.48}$$

and

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}i} \end{pmatrix}. \tag{2.49}$$

Using Euler's relation $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, we can conclude that the rotation for the S operator of $\frac{\pi}{2}$ is

$$e^{i\frac{\pi}{2}} = \cos(\frac{\pi}{2}) + i\sin(\frac{\pi}{2}) = i, \tag{2.50}$$

and for the T operator with a rotation of $\frac{\pi}{4}$ is

$$e^{i\frac{\pi}{4}} = \cos(\frac{\pi}{4}) + i\sin(\frac{\pi}{4}) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i. \tag{2.51}$$

These operators are unitary but not Hermitian.

2.2.7 Rotation matrices

Next, we seek the matrices associated to rotations on the Bloch sphere around the three axes by any arbitrary angle, not just π. To show the rotation in each axis, we will first derive the general equation that can be applied to each rotation. We define an arbitrary rotation by representing it by the operator that rotates around that axis. We define this operator as A.

We will start by describing the rotation around an axis through the basic equation

$$rot(|\psi\rangle) = e^{i\frac{\theta}{2}A}\,|\psi\rangle\,, \tag{2.52}$$

where $e^{i\frac{\theta}{2}A}$ is a rotation around that axis by an angle of θ. Using the Taylor series expansion

$$e^k = \sum_{k=0}^{\infty} \frac{x^k}{k!}, \tag{2.53}$$

we get the following rotation operator expansion

$$e^{i\frac{\theta}{2}A} = I + i\frac{\theta}{2}A + \frac{(i\frac{\theta}{2}A)^2}{2!} + \frac{(i\frac{\theta}{2}A)^3}{3!} + \frac{(i\frac{\theta}{2}A)^4}{4!} + \ldots. \tag{2.54}$$

Next, we regroup the terms into even and odd powers

$$e^{i\frac{\theta}{2}A} = \left(I + \frac{(i\frac{\theta}{2}A)^2}{2!} + \frac{(i\frac{\theta}{2}A)^4}{4!} + ...\right) + \left(i\frac{\theta}{2}A + \frac{(i\frac{\theta}{2}A)^3}{3!} + \frac{(i\frac{\theta}{2}A)^5}{5!} + ...\right). \tag{2.55}$$

Since we know that the A operator is one of the Pauli matrices $(\sigma_x, \sigma_y, \sigma_z)$, we have $A^2 = I$. Using this property, we then obtain

$$e^{i\frac{\theta}{2}A} = \left(I + \frac{(i\frac{\theta}{2})^2 I}{2!} + \frac{(i\frac{\theta}{2})^4 I^2}{4!} + ...\right) + \left(i\frac{\theta}{2}A + \frac{(i\frac{\theta}{2})^3 IA}{3!} + \frac{(i\frac{\theta}{2})^5 I^2 A}{5!} + ...\right) \tag{2.56}$$

or

$$e^{i\frac{\theta}{2}A} = \left(1 - \frac{(\frac{\theta}{2})^2}{2!} + \frac{(\frac{\theta}{2})^4}{4!} - ...\right) I + i\left(\frac{\theta}{2} - \frac{(\frac{\theta}{2})^3}{3!} + \frac{(\frac{\theta}{2})^5}{5!} - ...\right) A. \tag{2.57}$$

Using the Taylor expansion series for sin and cos, i.e.,

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}, \tag{2.58}$$

and

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}, \tag{2.59}$$

we finally get

$$e^{i\frac{\theta}{2}A} = \cos(\frac{\theta}{2})I - i\sin(\frac{\theta}{2})A. \tag{2.60}$$

For the special case where A is the Pauli matrix σ_x, we get

$$\begin{aligned} e^{i\frac{\theta}{2}\sigma_x} &= \cos(\frac{\theta}{2})I - i\sin(\frac{\theta}{2})\sigma_x, \\ &= \begin{pmatrix} \cos(\frac{\theta}{2}) & 0 \\ 0 & \cos(\frac{\theta}{2}) \end{pmatrix} - \begin{pmatrix} 0 & i\sin(\frac{\theta}{2}) \\ i\sin(\frac{\theta}{2}) & 0 \end{pmatrix}, \\ &= \begin{pmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}. \end{aligned} \tag{2.61}$$

This matrix corresponds to a rotation around the x-axis by and angle of θ. Using the same approach starting with the y and z Pauli matrices, we get

$$R_y(\theta) = e^{\frac{i\theta\sigma_y}{2}} = \begin{pmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}, \tag{2.62}$$

and

$$R_z(\theta) = e^{\frac{i\theta\sigma_z}{2}} = \begin{pmatrix} -\cos(\frac{\theta}{2}) + \sin(\frac{\theta}{2}) & 0 \\ 0 & \cos(\frac{\theta}{2}) - \sin(\frac{\theta}{2}) \end{pmatrix}. \tag{2.63}$$

Thus, we can describe any rotation around the three axes. An arbitrary rotation is merely the combination of rotations of one or more rotations around the x, y, and z axes.

2.3 Time evolution of qubit

The simulator also provides an animation which describes the evolution of a qubit in the presence of a DC electromagnetic field applied along the z-axis. A second simulation allows the consideration of another DC magnetic field rotating with uniform angular velocity in the x-y plane. This is referred to as the Rabi Field. The simulator shows how the different strengths of the applied electromagnetic fields affect the state of one or more qubits. The Rabi field simulation also allows for the simulation to show interesting scenarios where the state of the qubit will "flip" back and forth between the $|1\rangle$ and $|0\rangle$ states as an illustration of the Rabi formula.

2.3.1 Larmor precession

The simulator performs a time evolution of a qubit known as the Larmor precession. The precession is the spin of an electron due to a constant magnetic field, as shown in Fig. 2.3.

The earliest observations of this effect was the Stern-Gerlach experiment [29]. The equations governing this spin evolution are derived from the Ehrenfest theorem that describes the time evolution of the expectation value of a quantum-mechanical operator. The Ehrenfest equation is given by

$$\frac{d}{dt} < A >= \frac{1}{i\hbar} < [A, H(t)] > + < \frac{dA}{dt} > . \tag{2.64}$$

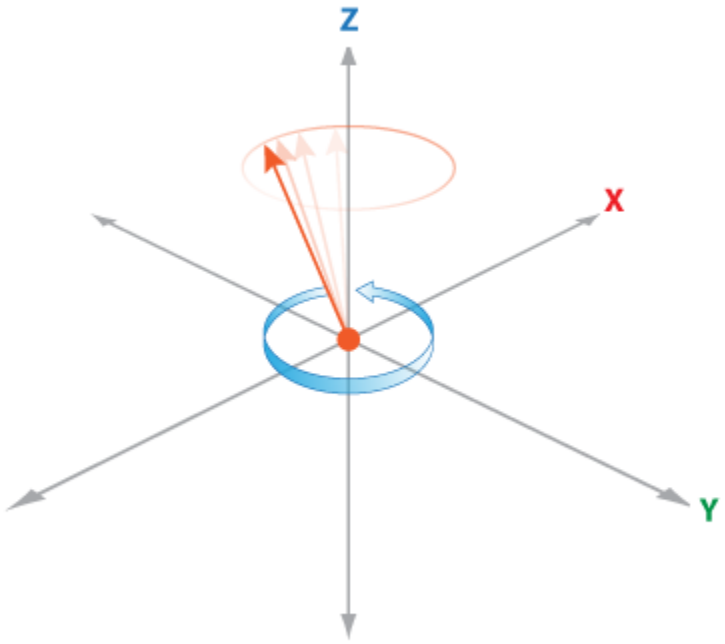

Figure 2.3: The rotation path of a qubit under the influence of a constant a magnetic field applied along the z-axis. The figure illustrates the Larmor precession at a constant angular velocity on a cone whose principle axis in along the z-axis.

Following ref. [5], we derive the equations that describe the Larmor precession by using the Pauli matrices: σ_x, σ_y, and σ_z for the operator A in Eq. (2.64):

$$\frac{d}{dt} < \sigma_x >= \frac{g\mu_B}{\hbar}(B_y < \sigma_z > -B_z < \sigma_y >), \tag{2.65}$$

$$\frac{d}{dt} < \sigma_y >= \frac{g\mu_B}{\hbar}(B_x < \sigma_x > -B_x < \sigma_z >), \tag{2.66}$$

$$\frac{d}{dt} < \sigma_z >= \frac{g\mu_B}{\hbar}(B_z < \sigma_y > -B_y < \sigma_x >). \tag{2.67}$$

We can write these three equations in a more compact notation, i.e.,

$$\frac{d < \vec{\sigma} >}{dt} = \frac{g\mu_B}{\hbar}(\vec{B} \times < \vec{\sigma} >) = \vec{\Omega} \times < \vec{\sigma} >, \tag{2.68}$$

where $\vec{\Omega} = \frac{g\mu_B}{\hbar}\vec{B}$ and $\mu_B = \frac{e\hbar}{2m_0}$ is the Bohr magnetron. Since $\vec{S} = \frac{\hbar}{2}\vec{\sigma}$, Eq. (2.68) can be rewritten as follows

$$\frac{d < \vec{S} >}{dt} = \frac{g\mu_B}{\hbar}(\vec{B} \times < \vec{S} >) = \vec{\Omega} \times < \vec{S} >, \tag{2.69}$$

where $< \vec{S} >$ is the expected value of the spin angular momentum. Equation (2.69) gives us the standard Larmor equation describing the Larmor precession. Starting with Eq. (2.69), we easily get

$$\frac{d(\vec{S} \cdot \vec{B})}{dt} = 0, \tag{2.70}$$

which means that the angle between the two vectors time independent. This fixes the apex angle, θ of the qubit. Furthermore, using Eq. (2.69), it can be shown that the azimuthal angle, ϕ increases at a constant angular velocity given by

$$\frac{d\phi}{dt} = \frac{g\mu_B B_z}{\hbar}, \tag{2.71}$$

which is defined as the Larmour frequency

$$\omega_l \stackrel{def}{=} \frac{egB}{2m_0}, \tag{2.72}$$

where e is the energy of the charge, g is the "g-factor" or gyro-magnetic ratio ("the ratio of the magnitude of the magnetic moment to that of the angular momentum"[29]) and m_0 is the free electron mass. In the simulator, we set $m_0 = 9.109 \times 10^{-31}$kilograms [28], $e = -1.26 \times 10^{-19}$C [28] and $g = 2.002319$[31]. Thus, we finally describe the two characteristic factors of the qubit as:

$$\theta = c, \tag{2.73}$$

and

$$\phi = \phi_0 + \omega_l t \tag{2.74}$$

.

This is the description of the qubit rotation around the z-axis, like a cone, at a constant angular velocity due to the influence of a constant magnetic field.

2.3.2 Rabi field

IN this section, we look at the action of two magnetic fields on a qubit. The first field is applied along the z-axis as in the case of the Larmor precession studied in the previous section. We use the notation B_z to describe this magnetic field. The second magnetic will be a DC field that rotates around the origin in the x-y plane at a constant angular velocity.

The direction of the magnetic field is away from the origin. We denote this second magnetic field B_{xy}. The qubit will not rotate on a cone, but will experience an additional magnetic force duo to B_{xy}. The angular velocity and the strength of the B_{xy} will have an effect on the qubit. A question of interest is: "What if the force of the magnetic field B_z were to push the qubit around the z-axis as the same rate as B_{xy} rotates in the xy-plane?" As we will show, this can lead to a spin flip from the north to Sought pole on the Bloch sphere. I.I. Rabi was the first to calculate the probability of the spin flip between the two poles. Here we will present a different derivation that focuses on the two-state qubit to show the effect of the Rabi field.

The Hamiltonian of a qubit submitted to the superposition of the two magnetic fields described above is given by

$$\begin{aligned} H(t) &= B_z S_z + B_{xy}\left[\cos(\omega t)S_x + \sin(\omega t)S_y\right] \qquad (2.75)\\ &= \frac{\hbar}{2}B_z\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} + \frac{\hbar}{2}B_{xy}[\cos(\omega t)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + \sin(\omega t)\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}]\\ &= \frac{\hbar}{2}\begin{pmatrix} B_z & 0 \\ 0 & -B_z \end{pmatrix} + \frac{\hbar}{2}\left[\begin{pmatrix} 0 & B_{xy}e^{-i\omega t} \\ B_{xy}e^{i\omega t} & 0 \end{pmatrix}\right]\\ &= \frac{\hbar}{2}\begin{pmatrix} B_z & B_{xy}e^{-i\omega t} \\ B_{xy}e^{i\omega t} & -B_z \end{pmatrix} \qquad (2.76) \end{aligned}$$

where

$$S_y = \frac{\hbar}{2}\sigma_y, \qquad (2.77)$$

and

$$S_z = \frac{\hbar}{2}\sigma_z. \qquad (2.78)$$

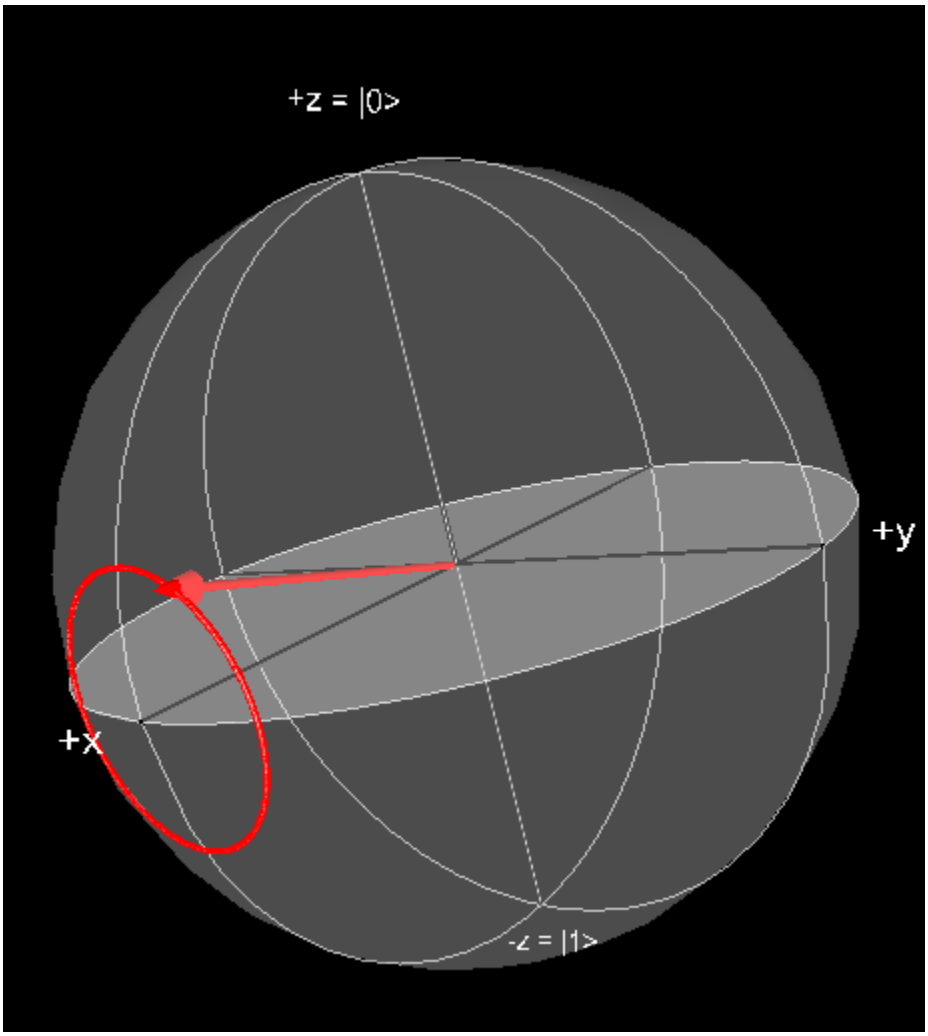

Figure 2.4: The rotation path of a qubit with a magnetic field applied along the x-axis.

We describe the current state of qubit $|\psi\rangle$ at time t as $|\psi(t)\rangle$. If we use the basis set ($|0\rangle$, $|1\rangle$), the state of the qubit at any given time t is given by

$$|\psi(t)\rangle = \alpha(t)\,|0\rangle + \beta(t)\,|1\rangle\,. \tag{2.79}$$

The latter must satisfy the Schroedinger equation

$$i\hbar\frac{d}{dt}\,|\psi(t)\rangle = H(t)\,|\psi(t)\rangle\,. \tag{2.80}$$

Substituting the expression $|\psi(t)\rangle$ above in the last equation, we find that $\alpha(t)$ and $\beta(t)$ must satisfy the following coupled differential equations

$$i\frac{d\alpha(t)}{dt} = \frac{B_z}{2}\alpha(t) + \frac{B_{xy}}{2}e^{-i\omega t}\beta(t), \tag{2.81}$$

and

$$i\frac{d\beta(t)}{dt} = \frac{B_{xy}}{2}e^{i\omega t}\alpha(t) - \frac{B_z}{2}\beta(t). \tag{2.82}$$

Now, we will change the reference frame to describe the qubit evolution. Currently, we view the magnetic fields and the state of the qubit from a fixed perspective. But if we change that perspective to a rotating reference, we then solve this equation in a more simple manner. The frame of reference is the rotation of the magnetic field that is rotating around the z-axis along the x-y axis. Here we define the rotational reference components $\hat{\alpha}(t)$ and $\hat{\beta}(t)$ as

$$\hat{\alpha}(t) = e^{\frac{i\omega t}{2}}\alpha(t), \tag{2.83}$$

and

$$\hat{\beta}(t) = e^{\frac{i\omega t}{2}}\beta(t). \tag{2.84}$$

It is easily shown that these new variables satisfy the following equations

$$i\frac{d\hat{\alpha}(t)}{dt} = -\frac{\triangle\omega}{2}\hat{\alpha}(t) + \frac{B_{xy}}{2}\hat{\beta}(t), \tag{2.85}$$

and

$$i\frac{d\hat{\beta}(t)}{dt} = \frac{B_{xy}}{2}\hat{\alpha}(t) + \frac{\triangle\omega}{2}\hat{\beta}(t), \tag{2.86}$$

where $\triangle\omega = \omega^2 - B_z$. The derivatives of the components $\hat{\alpha}(t)$ and $\hat{\beta}(t)$ can be written in the qubit and operator form

$$i\hbar\frac{d\left|\hat{\psi}\right\rangle}{dt} = \hat{H}\left|\hat{\psi}\right\rangle, \tag{2.87}$$

where

$$\left|\hat{\psi}\right\rangle = \begin{pmatrix} \hat{\alpha}(t) \\ \hat{\beta}(t) \end{pmatrix} = \hat{\alpha}(t)\left|0\right\rangle + \hat{\beta}(t)\left|1\right\rangle, \tag{2.88}$$

and

$$\hat{H} = \frac{\hbar}{2}\begin{pmatrix} -\triangle\omega & B_{xy} \\ B_{xy} & \triangle\omega \end{pmatrix}. \tag{2.89}$$

Now we will show the mapping from $|\psi(t)\rangle$ to $|\hat{\psi}(t)\rangle$. We define the mapping of the $|\psi(t)\rangle$ to the $|\hat{\psi}(t)\rangle$ through the rotation operator $R(t)$. We define $R(t)$ as

$$R(t) = e^{\frac{i\omega t}{\hbar}S_z}, \tag{2.90}$$

such that

$$\left|\hat{\psi}(t)\right\rangle = R(t)\left|\psi(t)\right\rangle, \tag{2.91}$$

and

$$|\psi(t)\rangle = R(t)^{-1} |\hat{\psi}(t)\rangle \,. \tag{2.92}$$

Using the definitions of $\hat{\alpha}(t)$ and $\hat{\beta}(t)$ we get the function $R(t)$, a rotation around the z-axis

$$R(t) = e^{\frac{-i\omega t}{\hbar} S_z}, \tag{2.93}$$

and so the inverse is

$$R^{-1}(t) = e^{\frac{-i\omega t}{2} \sigma_z}. \tag{2.94}$$

These last two equations can be rewritten in the form of a Schroedinger equation by introducing the column vector

$$|\hat{\psi}(t)\rangle = \begin{pmatrix} \hat{\alpha}(t) \\ \hat{\beta}(t) \end{pmatrix}. \tag{2.95}$$

The two equations for $\hat{\alpha}(t)$ and $\hat{\beta}(t)$ can then be recast as follows

$$\frac{d}{dt} |\hat{\psi}(t)\rangle = \frac{\hat{H}}{i\hbar} |\hat{\psi}\rangle = \hat{X} |\hat{\psi}\rangle \,, \tag{2.96}$$

where the matrix $\hat{X}$ is given by

$$\hat{X} = \frac{-i}{2} \begin{pmatrix} -\triangle\omega & B_{xy} \\ B_{xy} & \triangle\omega \end{pmatrix} = i \begin{pmatrix} \frac{\triangle\omega}{2} & \frac{-B_{xy}}{2} \\ \frac{-B_{xy}}{2} & \frac{-\triangle\omega}{2} \end{pmatrix}. \tag{2.97}$$

We seek a solution of this last equation as follows

$$|\hat{\psi}(t)\rangle = e^{Q(t)} |\hat{\psi}(0)\rangle \,. \tag{2.98}$$

Since $\hat{X}$ is time independent, it is easy to show by simple substitution that $Q(t)$ is given

by

$$Q(t) = \int_0^t \hat{X}(t)\, dt = i \begin{pmatrix} \frac{\triangle\omega t}{2} & \frac{-B_{xy}t}{2} \\ \frac{-B_{xy}t}{2} & \frac{-\triangle\omega t}{2} \end{pmatrix}. \tag{2.99}$$

Next, we introduce γ and δ to simplify the equations below

$$\gamma = \frac{\triangle\omega}{2}t, \tag{2.100}$$

and

$$\delta = -\frac{B_{xy}}{2}t. \tag{2.101}$$

In this case

$$Q(t) = \begin{pmatrix} \gamma & \delta \\ \delta & -\gamma \end{pmatrix}. \tag{2.102}$$

To calculate $e^{Q(t)}$ we first diagonalize $Q(t)$. Calling S the matrix which diagonalizes $Q(t)$, then

$$\Lambda(t) = S^{-1}Q(t)S. \tag{2.103}$$

where $\Lambda(t)$ is the column vector containing the two eigenvalues of $Q(t)$. Taking the exponential on both sides of this equation, we obtain

$$e^{\Lambda(t)} = S^{-1}e^{Q(t)}S, \tag{2.104}$$

which leads to the following expression for $e^{Q(t)}$

$$e^{Q(t)} = Se^{\Lambda(t)}S^{-1}. \tag{2.105}$$

The latter can be calculated explicitly once we have the eigenvalues of $Q(t)$. The eigenvalues are

$$\lambda_1 = i\sqrt{\gamma^2 + \delta^2}, \tag{2.106}$$

and

$$\lambda_2 = -i\sqrt{\gamma^2 + \delta^2}, \tag{2.107}$$

and therefore $e^{\Lambda(t)}$ is given by

$$e^{\Lambda(t)} = \begin{pmatrix} e^{\lambda_1} & 0 \\ 0 & e^{\lambda_2} \end{pmatrix} = \begin{pmatrix} e^{i\sqrt{\gamma^2+\delta^2}} & 0 \\ 0 & e^{-i\sqrt{\gamma^2+\delta^2}} \end{pmatrix}. \tag{2.108}$$

The matrix S which diagonalizes $Q(t)$ is formed by the two column vectors which are the corresponding eigenvectors to λ_1 and λ_2. These eigenvectors can be found easily and S is given explicitly as follows

$$S = \begin{pmatrix} \frac{-\delta}{\gamma-\sqrt{\gamma^2+\delta^2}} & \frac{-\delta}{\gamma+\sqrt{\gamma^2+\delta^2}} \\ 1 & 1 \end{pmatrix}. \tag{2.109}$$

The inverse of the matrix is found to be

$$S^{-1} = \frac{\delta}{2\sqrt{\gamma^2+\delta^2}} \begin{pmatrix} 1 & \frac{\delta}{\gamma+\sqrt{\gamma^2+\delta^2}} \\ -1 & \frac{\delta}{\gamma-\sqrt{\gamma^2+\delta^2}} \end{pmatrix}. \tag{2.110}$$

This allows us to write the equation that will provide a mapping from that state of the qubit at time t, though this is from the rotating perspective around the z-axis at the rate of ω radians per second.

$$|\hat{\psi}(t)\rangle = Se^{\Lambda(t)}S^{-1}\,|\hat{\psi}(0)\rangle \tag{2.111}$$

Since we also know the mapping from $|\hat{\psi}\rangle$ to $|\psi\rangle$, we can finally determine the equation that describes $|\psi\rangle$ at time t:

$$\begin{aligned}
|\psi\rangle &= R^{-1}(t)\,|\hat{\psi}\rangle\,, \\
&= R^{-1}(t)\left(Se^{\Lambda(t)}S^{-1}\right)|\hat{\psi}(0)\rangle\,, \\
&= R^{-1}(t)\left(Se^{\Lambda(t)}S^{-1}\right)R^{-1}(0)\,|\psi(0)\rangle\,, \\
|\psi\rangle &= R^{-1}(t)\left(Se^{\Lambda(t)}S^{-1}\right)|\psi(0)\rangle\,. \qquad (2.112)
\end{aligned}$$

Now we combine all of the operators found above to get an analytic expression for the operator on the right hand side of Eq. (2.112)

$$\begin{aligned}
R^{-1}(t) &= e^{\frac{-i\omega t}{2}\sigma_z} \\
&= \begin{pmatrix} \cos\left(\frac{\omega t}{2}\right) - i\sin\left(\frac{\omega t}{2}\right) & 0 \\ 0 & \cos\left(\frac{\omega t}{2}\right) + i\sin\left(\frac{\omega t}{2}\right) \end{pmatrix}
\end{aligned}$$

$$Se^{\Lambda(t)}S^{-1} = \frac{\delta}{2\sqrt{\gamma^2+\delta^2}} \begin{pmatrix} \frac{-\delta}{\gamma-\sqrt{\gamma^2+\delta^2}} & \frac{-\delta}{\gamma+\sqrt{\gamma^2+\delta^2}} \\ 1 & 1 \end{pmatrix} \begin{pmatrix} e^{i\sqrt{\gamma^2+\delta^2}} & 0 \\ 0 & e^{-i\sqrt{\gamma^2+\delta^2}} \end{pmatrix} \begin{pmatrix} 1 & \frac{\delta}{\gamma+\sqrt{\gamma^2+\delta^2}} \\ -1 & \frac{\delta}{\gamma-\sqrt{\gamma^2+\delta^2}} \end{pmatrix} \qquad (2.113)$$

We condense this by defining

$$\rho = \sqrt{\gamma^2+\delta^2}, \qquad (2.114)$$

which finally leads to

$$Se^{\Lambda(t)}S^{-1} = \frac{\delta}{2\rho} \begin{pmatrix} \frac{-\gamma e^{i\rho}}{\gamma-\rho} - \frac{-\gamma e^{-i\rho}}{\gamma+\rho} & e^{i\rho} + e^{-i\rho} \\ e^{i\rho} - e^{-i\rho} & \frac{\gamma e^{i\rho}}{\gamma+\rho} + \frac{\gamma e^{-i\rho}}{\gamma-\rho} \end{pmatrix}. \qquad (2.115)$$

This is the explicit form of the matrix needed to find $|\psi\rangle$ at all time t as a result of the two magnetic fields.

Chapter 3

Bloch Sphere Simulator

3.1 Simulator description

3.1.1 Running the application

The Bloch sphere simulator has been written so that it can be launched in three different ways: as an application, as an applet on a web page or as an application launched from a web page. Java has three different launch configurations. The first is the Java application. In this case, there is either a script that runs the Java virtual machine directly using the compiled Java classes as an argument, or the launch parameters can be embedded in a compressed file containing the Java classes. Our implementation uses a script. This is because we wanted to be able to support several different operating systems and processor configurations. Our application supports three different operating systems and two different architectures. This simulator has the launch configurations for Windows (XP, Vista, 7) in the x86-32 and x86-64 architectures, Linux in the x86-32 and x86-64 architectures, and Mac OSX in the x86 architecture. The deployment, or list of files that are used when running the application are: the scripts to launch the application, the Java libraries that contain the classes to run the application, and the architecture/operating system specific libraries to run the 3D part of

the application.

To run the application in Windows, one must either run the runSimulatorWindows32.bat or the runSimulatorWindows64.bat. The 32 is for the 32bit version of the Java Virtual Machine and the 64 is for the 64 bit version of the Java Virtual Machine. Windows does allow 32 bit programs to be run on its 64 bit version of the operating system. To run the application on Linux, you should run either the runSimulatorLinux32.sh or run the runSimulatorLInux64.sh. Like the windows version, the JVM determines which version (32 or 64 bit) as opposed to the OS version. Additionally, the script assumes that the bash shell script is installed in the Linux environment. Running the application in MacOSX requires that the user run the shell script runSimulatorOSX.sh. The bash shell is installed by default with all default Mac OSX installations.

The second method to launch the application is by launching the application through Java Native Launch Protocol (JNLP). The JNLP technology works by clicking on the JNLP link on the bloch3d.html web page. This will prompt a security warning. This is because the files that are used to run the application need to be signed. The current version of the application does not have a certificate authority (CA) to provide the ability to sign the jar files [27]. One of the advantages of running the application through JNLP is that is can be cached on the local computer. After running the simulator once, if the computer does not have in Internet connection, the simulator can be launched through a local cached version. To run the locally cached version of the application, one must first open the Java Control Panel. In Windows, this can be accomplished by opening the control panel and then opening "Java". In Linux, you can do this by running the "javacpl" located in the "JRE HOME bin" directory where "JRE HOME" is the path the Java installation. Figure 3.1 shows the Java control panel.

Clicking on the "View..." button under the section "Temporary Internet Files" will bring up the Java Cache Viewer. This is a collection of previously run JNLP programs. An

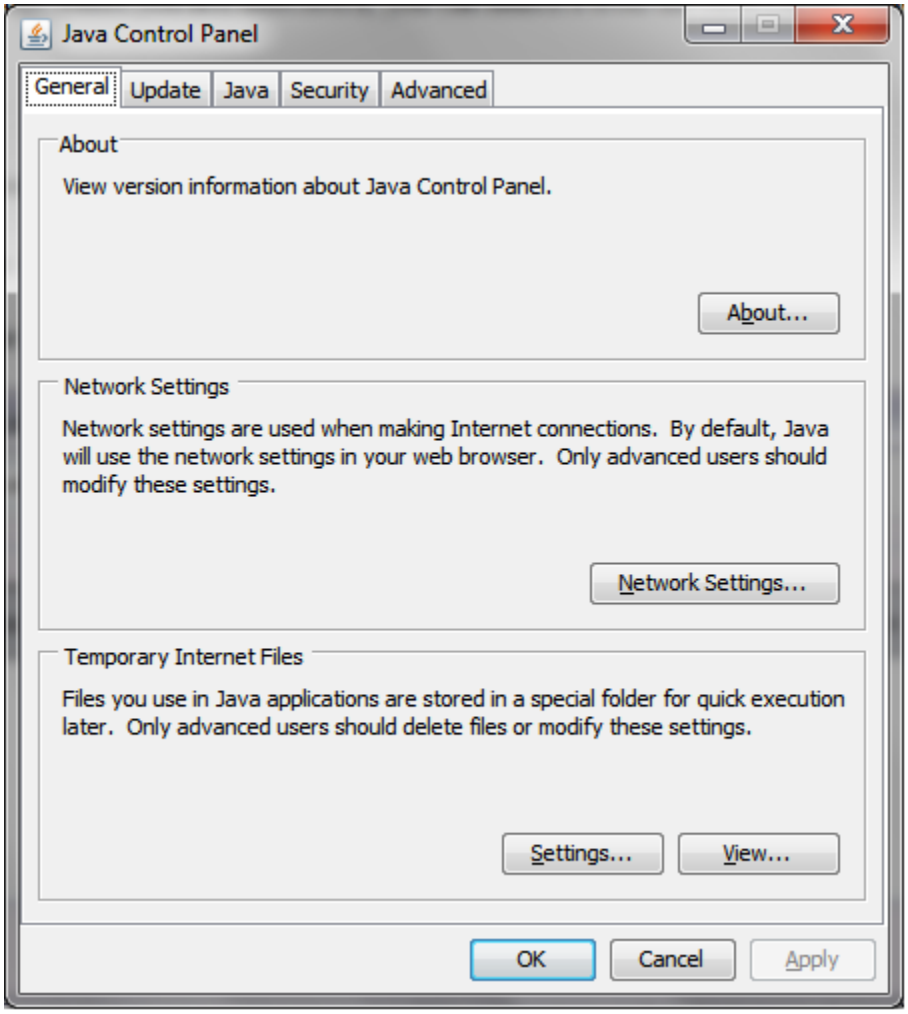

Figure 3.1: The Java Control Panel window

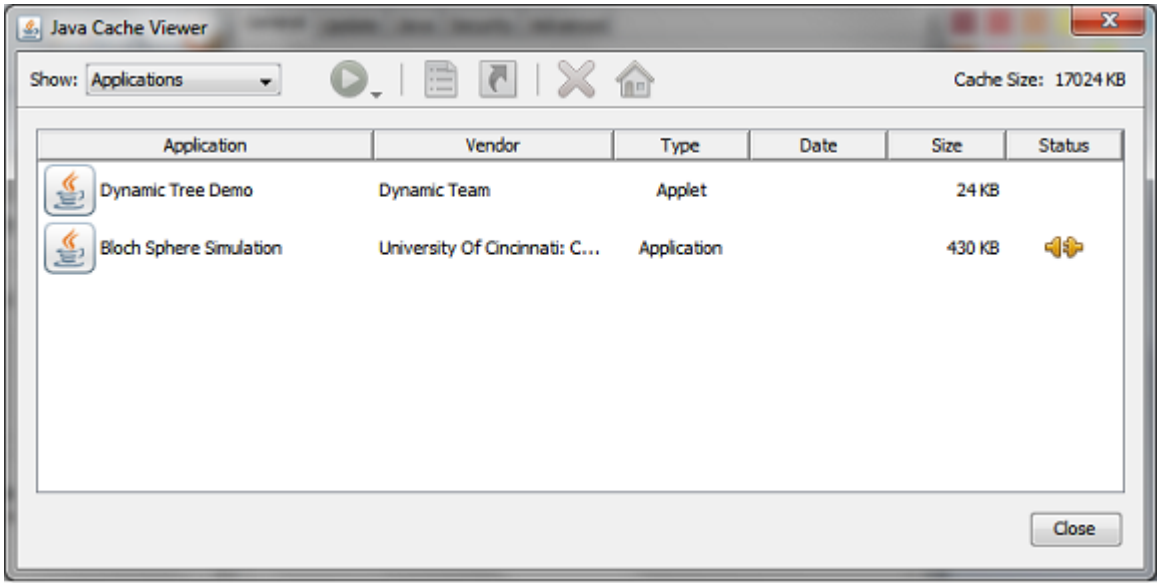

Figure 3.2: The Java Cache Viewer window

example of this can be seen in fig. 3.2 There are many options to run the program offline. One option is to create a shortcut and other options as seen in fig. 3.3.

The third method to launch the application is similar to the JNLP application launch method. This method allows the application to be run within the browser. Instead of clicking on a link, the Bloch sphere simulator is automatically started when the web page is loaded.

3.1.2 Requirements to run the simulator

To run the Bloch sphere simulator, there are some software and hardware requirements for the application to run. The program was written in Java 1.6.0. This is a recent version of Java. The requirements for Java 1.6.0 are available on the Internet [2]. Java supports the Solaris, Windows, Linux and Mac OSX operating systems. The Java virtual machine (or JVM) needs to be installed on the computer before the application can be run [4]. If the user has an older version of java (at least JVM 1.3), then the JVM should be able to detect that a newer version is required. It will automatically download the JVM from java.sun.com.

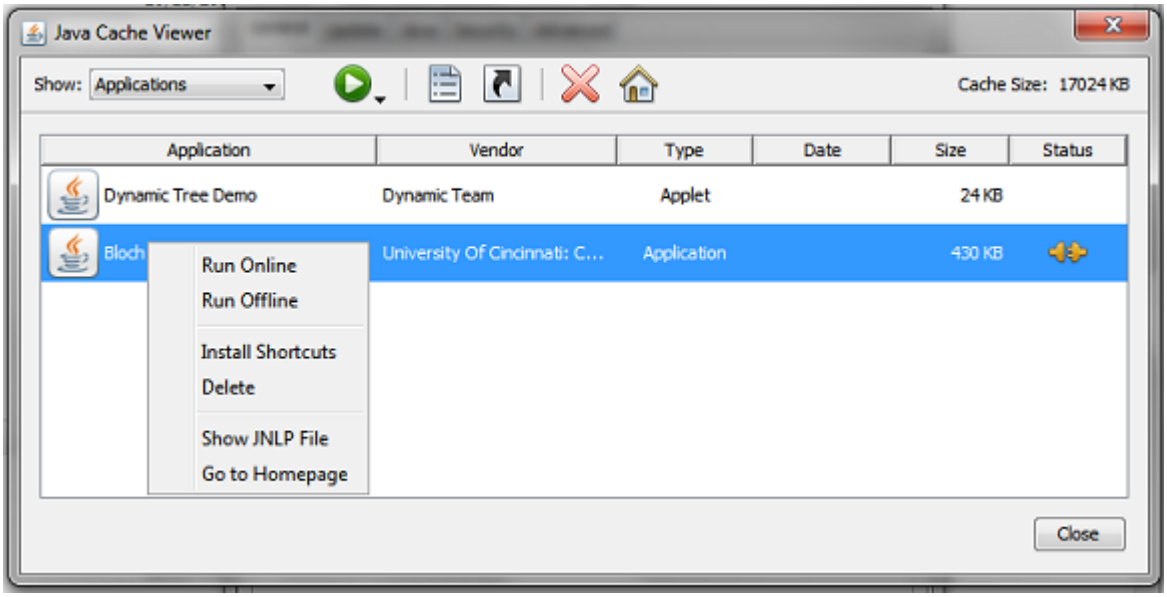

Figure 3.3: The selection options in the Java Cache Viewer window

When running the application, it will automatically download the required 3D libraries used to run the application.

Windows, Mac OSX, and Linux require that OpenGL version 1.3 (or greater) be installed on the operating system. All version of Windows comes with OpenGL already installed with the operating system[1].

3.1.3 An overview of the simulator user interface

The application has four major parts to it: The toolbar, the 3D qubit viewer, the qubit state control panel, and the external magnetic force control panel. Figure 3.4 displays those four parts. The top section, labeled "A", is the toolbar. Below the toolbar is the 3D representation of the Bloch sphere, labeled "B". The third is the qubit state control panel, labeled "C". Finally, the bottom right part, labeled "D", is the series of tabs for the custom operators, Larmor precession, record and playback, and the Rabi Field controls.

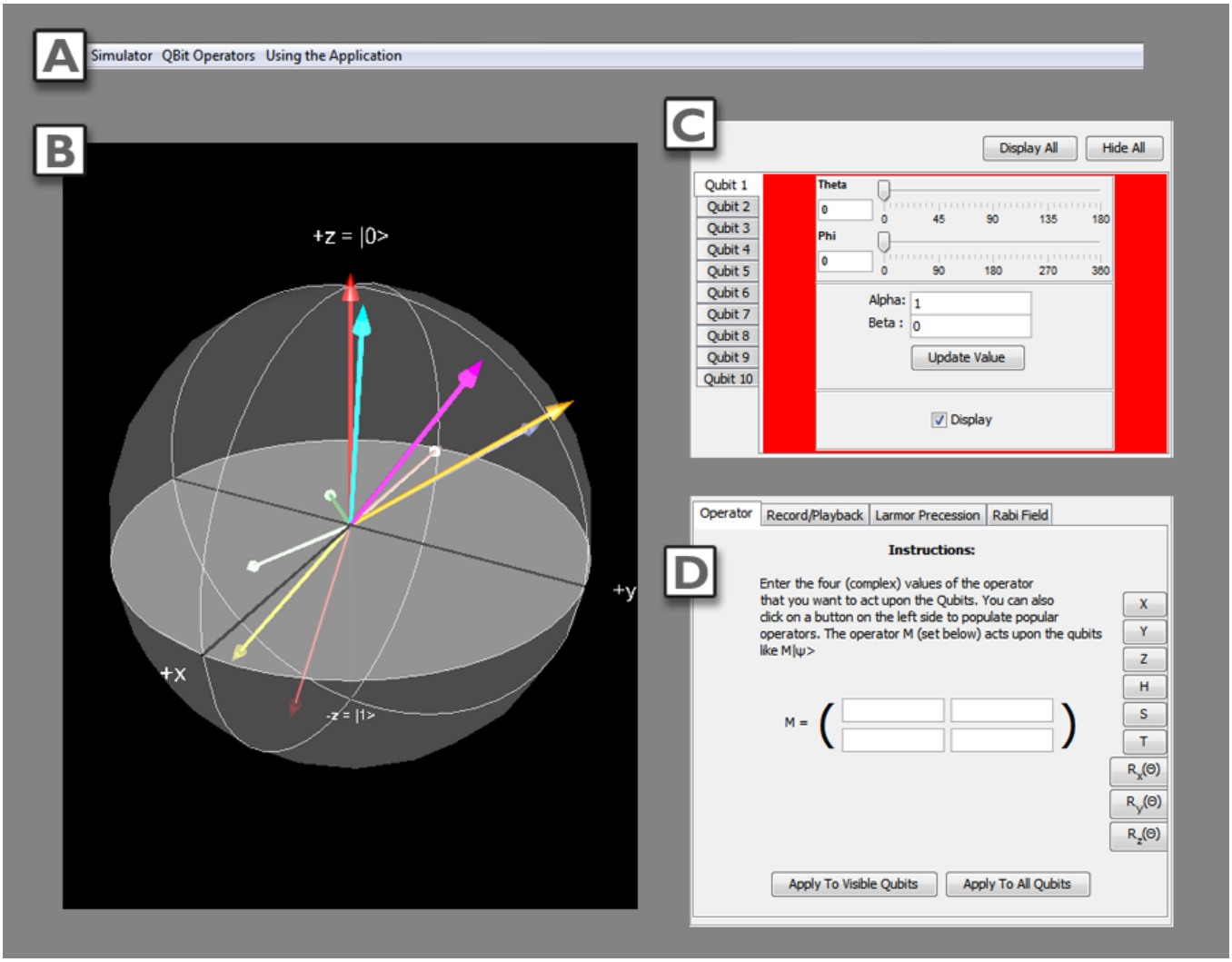

Figure 3.4: The Bloch sphere simulator with the parts highlighted in different colors.

3.1.4 Simulator menu bar

The menu bar, located at the top has three menu selections: "Simulator", "Qubit Operators", and "Using the Application". The simulator menu option has three selections: "Reset View", "Save Current State", and "Load State From File". The Reset view function allows the

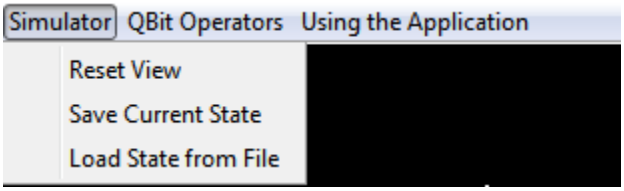

Figure 3.5: The Bloch sphere simulator menu selection.

application to reset the point-of view of the 3D Bloch sphere. As the Bloch sphere can be rotated, this resets the view. The save and load current state allows the user to save the current state of each of the qubits. This includes the visibility of the qubit. The qubit state files are saved as ".bss" files. The ".bss" is a Bloch sphere simulation file. This is a text file that is created with the phi, theta and visibility property for each of the qubits. Below is an example files saving the default state of the qubits. The file has the phi and theta values as floats that represent the phi and theta angle of the qubit. The angle is a float value in the unit of degrees. The visible property associated with the qubit is true if the qubit is currently visible and false if it is hidden.

The next menu option is the "Qubit Operators". This provides information about some of the operators that are available in the simulator. These are the σ_x, σ_y, σ_z, the Hadamard operator and the phase shift operator.

Each of these provides a dialog that shows an animation of the qubit to demonstrate the rotation effect of each operator. Along with it on the right side is information about the operator and some of the properties of the operator. Each of the operators will show a rotation of the qubit based on the effect of the operator. Since the operators are rotations,

text 1 An example file showing the saved state of the qubits in the simulator.

```
qbit.0.phi=0.0
qbit.0.theta=0.0
qbit.0.visible=true
qbit.1.phi=36.0
qbit.1.theta=18.0
qbit.1.visible=true
qbit.2.phi=72.0
qbit.2.theta=36.0
qbit.2.visible=true
qbit.3.phi=108.0
qbit.3.theta=54.0
qbit.3.visible=true
qbit.4.phi=144.0
qbit.4.theta=72.0
qbit.4.visible=true
qbit.5.phi=180.0
qbit.5.theta=90.0
qbit.5.visible=true
qbit.6.phi=216.0
qbit.6.theta=108.0
qbit.6.visible=true
qbit.7.phi=252.0
qbit.7.theta=126.0
qbit.7.visible=true
qbit.8.phi=288.0
qbit.8.theta=144.0
qbit.8.visible=true
qbit.9.phi=324.0
qbit.9.theta=162.0
qbit.9.visible=true
```

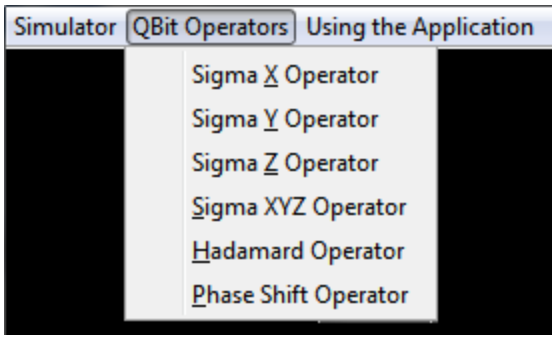

Figure 3.6: The Bloch sphere qubit operators menu selection.

the animation shows the rotation. Though the operator does not slowly move the state of the qubit, this is done so the rotation is apparent. Figure 3.7 shows the help information for the σ_x operator.

The last menu selection item is the "Using the Application" menu. This allows the user to see information about the advanced features of the application. This shows the information to set the qubit state, use the custom operator, use the record/playback functionality, the Larmor precession, and the Rabi field. Figure 3.8 shows the menu options available.

Selecting each one provides an understanding of the controls along with some of the theory behind the interaction. Figure 3.9 displays the example dialog of the information provided for the Larmor precession.

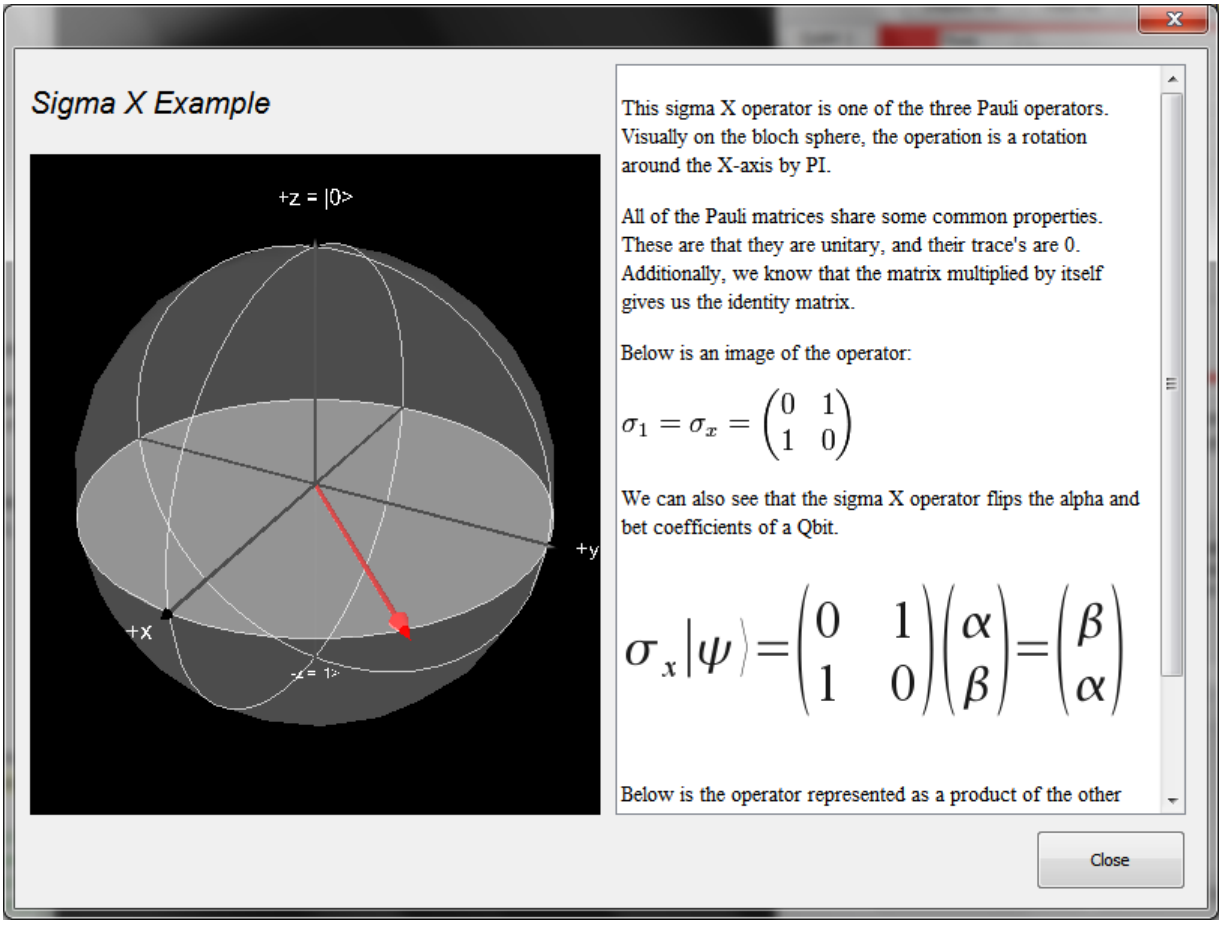

Figure 3.7: The Bloch sphere example help that shows the qubit starting before the σ_x operator is applied.

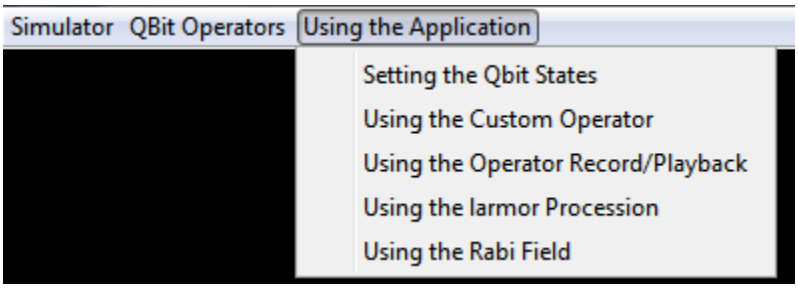

Figure 3.8: The Bloch sphere qubit operators menu selection.

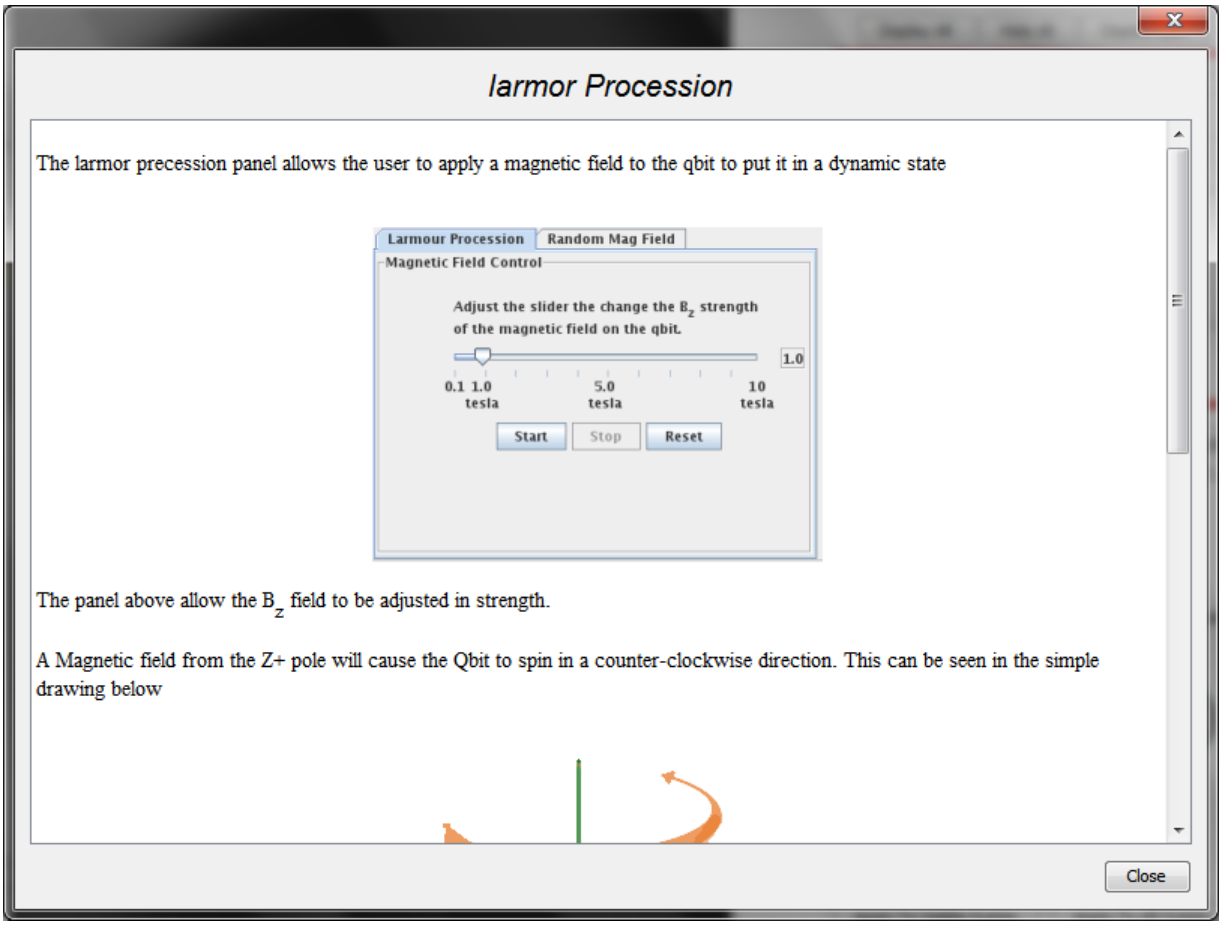

Figure 3.9: An example dialog showing how to use the Larmor precession and its basic theory.

3.1.5 Viewing the qubit state on the Bloch sphere

The 3D qubit viewer is the visual representation of the qubits. They show the representation of the qubit using the Bloch sphere theory. As discussed in Chapter 3, the Bloch sphere is a spherical representation of the qubit for the alpha and beta values describing the qubit states. The point of view can be manipulated by dragging the cursor in the viewer. Dragging of the mouse with the left mouse button horizontally will cause the sphere to rotate around the X-axis. Dragging the mouse using the left mouse button horizontally will cause the sphere to rotate around the Y axis. The changing of the point of view on the Bloch sphere in the viewer will have no effect on the position or motion of the qubits with respect to the axis of the Bloch sphere. As stated before, the point of view can be reset using the "Reset View" option under the "Simulator" menu.

3.1.6 Changing qubit state

When using the simulator, you have the ability to measure the current state of the qubit (without destroying it)! You also have the ability to manipulate the state to see how an operator, or series of operators will affect the state of the qubit. Figure 3.10 highlights the control panel for the qubit states in the top left. There are three ways that the qubit state can be change: by manually typing in the theta and phi angles (in degrees), adjusting the slider values of the angles, or entering the α and β values. Whenever the state of the qubit is modified by using one control, it will update the values in the other controls. Changing the angle using the slider will update the text value of the angle and also the α and β values.

As stated before, one the ways to change the qubit state is to use the sliders on the qubit control panel. The theta value can vary from 0 to 180 degrees. The theta angle is the angle between the positive z axis or the $|0\rangle$ state and the qubit. When the user enters a new value and then changes the focus of the text field by hitting the tab key, or by using the mouse

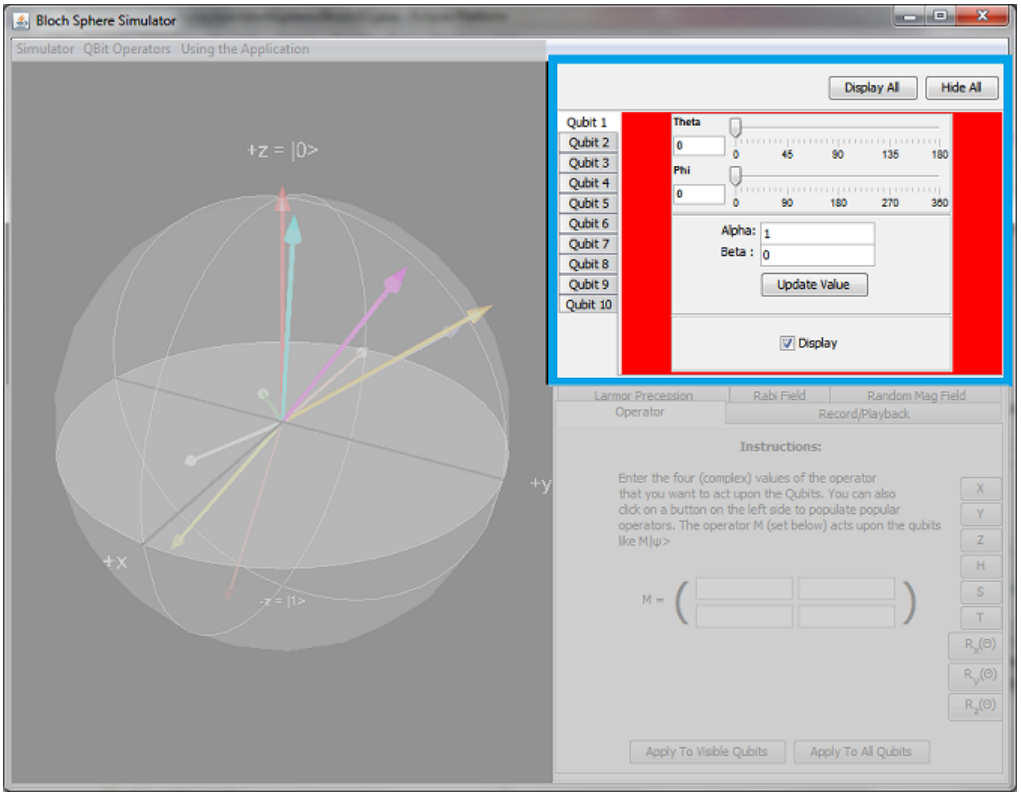

Figure 3.10: The Bloch sphere simulator highlighting the qubit control panel in the top right.

to change the focus by clicking on another component, the application will update the qubit state using the new angle to set the qubit state. If the user enters a value that is outside of the range of 0 to 180, then the application will reset the value back to the previous value. The user can also change the phi angle. The phi angle is the angle between the positive x-axis and the projections of the qubit in the (x, y) plane. Phi varies between 0 to 360 degrees. Again, to change the values by manually entering the numeric value, the tab key must be hit, or transfer focus to another part of the application to have the value take effect.

The second method to change the qubit state is by using the sliders. A slider works by having the user select the slider, by tabbing to it, or by clicking on it. The value can be changed by dragging the slider marker left and right using the mouse, or by using the left and right arrow keys to adjust the value by one increment. As the slider changes value, the qubit state will be immediately modified and displayed on the Bloch sphere.

The third method to change the qubit value is by modifying the alpha and beta values of the qubit. The alpha and beta fields can have complex numbers entered into either field. If a field is left blank, the application will assume that it has a value of zero. When both values have been changed in the two text fields, the value can be updated by pressing the "Update Value" button. Setting the qubit value through this control can be the most precise, but also the most tricky. The simulator will adjust the value if it is not normalized. Remember that the α and the β values must adhere to the equation: $|\alpha|^2 + |\beta|^2 = 1$. Secondly, if a complex number is entered to the alpha value, it will be changed to a real number by multiplying both the α and the β values by the complex conjugate of α.

3.1.7 Using the custom operator

The custom operator allows the user to select an operator, or to enter a custom operator. The custom operator tab has three parts to the screen: the operator values, the preset operators, and the action buttons. Figure 3.11 shows the three different parts.

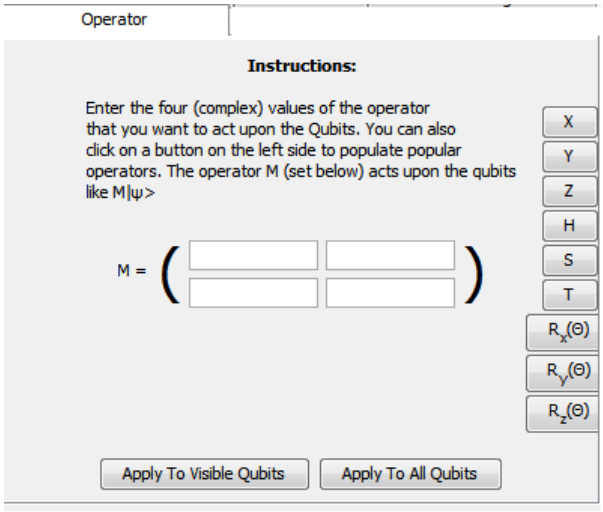

Figure 3.11: The custom operator tab.

The first part contains the operator values. When an operation is applied to the current state of the qubits, the values that are in the two by two matrix, consisting of complex numbers. The simulator checks that the operator is unitary. The basic definition is that an operator is unitary if when multiplied by its adjoint, that we get the identity operator. The simulator will check that the operator is correct by multiplying the elements and verifying that they are equal to the elements in the identity matrix. The operation performs the multiplication

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a^* & b^* \\ c^* & d^* \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.1}$$

i.e., it checks if the following equalities are satisfied

$$aa^* + bc^* = 1, \tag{3.2}$$

$$ab^* + bd^* = 0, \tag{3.3}$$

$$ca^* + dc^* = 0, \tag{3.4}$$

and

$$cb^* + dd^* = 1. \tag{3.5}$$

If the operator values do not pass this check, then the simulator will not apply the operator. The action of the operator is done when the user clicks on either the "Apply To Visible Qubits" or "Apply To All Qubits" button. The "Apply To Visible Qubits" button will have the operator act on all the qubits that are visible on the 3D Bloch sphere or if their "Display" check box is checked in the qubit state tab. The other button will have the operator act on the qubit regardless of its visibility on the Bloch sphere. Each time the button is clicked, the operator will act on the qubits. There are several functions that are both unitary and Hermitian, so having the operator act on the qubits twice will have a null effect. The qubits will return to their original state.

The operators can be difficult to understand and grasp. For this, the simulator provides the ability to load popular operators. The user can click on the left side of the operator tab to load the operator values for the σ_x, σ_y, σ_z, H, R, S and T operators. This will not have the operators act on the qubits, but it will load the values of the operators into the operator values. The user can have the operator then act on the qubits by clicking on the action buttons. The bottom three operators are the generic rotation operators. The operators can rotate around the X, Y, or Z axis. When the user clicks on any of these, a dialog appears allowing them to specify the angle of the desired rotation. Figure 3.12 shows this dialog. The user must enter the angle rotation in radians. Unfortunately, the application does not

allow the user to put in "2pi" or "pi/2". The value must be a decimal that is greater than zero. Instead of π, the user should type in 3.141. After a valid angle has been entered, the simulator willl calculate the rotation values and enter them into the operator values.

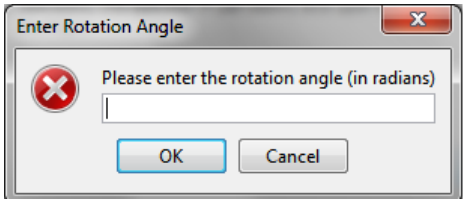

Figure 3.12: The custom operator tab.

The custom operator gives the user to create an infinite number of operations to apply to the qubit. The effect can be seen immediately on the state of one or more qubits. The next section will discuss how a series of these operators can be recorded to show the evolution of the state of a qubit as it experience multiple operators acting upon it in a sequential manner.

3.1.8 Using the record/playback functionality

The record/playback functionality allows users to record a sequence of operators that act upon the qubit states. The recording function allows the user to use the custom operator to create operators and have them act on the qubits. As they are applied to the qubits, they are recorded in the order that they are applied. The recorded operators can then be saved to a file for playback later. The playback functionality allows the user to load a saved file and then "play" or have the operators applied to the qubit state in the same order that they were recorded. The speed at which the operators are applied can be varied and the user can choose to individually step through each operator.

When the user clicks on the "Record/Playback" tab, the first screen that comes up allows the user to decide to record a series of operators, or play them back. This is seen in Fig. 3.13.

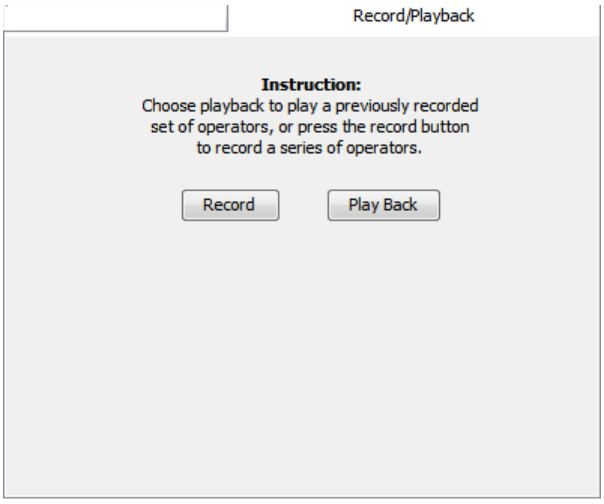

Figure 3.13: The record/playback tab.

When the user clicks on the record screen, the qubit state panel is replaced with the record panel and the only tab that is available on the bottom left is the "Custom Operator" tab. At this point, the simulator is in recording mode and any operator that is applied to the qubits will be recorded on the top left record panel. Figure 3.13 shows the default state with the recording panel on the top and the custom operator panel on the bottom.

As the operators are applied to the qubits, the record panel will show the value of the operator. The operator at the top of the list is the first operator that is applied. Each operator applied after that will appear below in the order that they are applied to the qubits. The Custom operator tab at the bottom works in the same manner when applying an operator to a qubit state. When all the desired operators have been recorded, the user can save the recorded operators by clicking on the "Save" button in the record panel. This will bring up a file explorer dialog to choose the file name and location.

The file that stores the operators is a plain text file that contains the operator number and then the four elements of the operator, deliminated by a comma. File 1 shows an example file that contains the σ_x, σ_y, and σ_z operators recorded (in that order).

The contents of the file do not need to be created by the simulator, they may be created

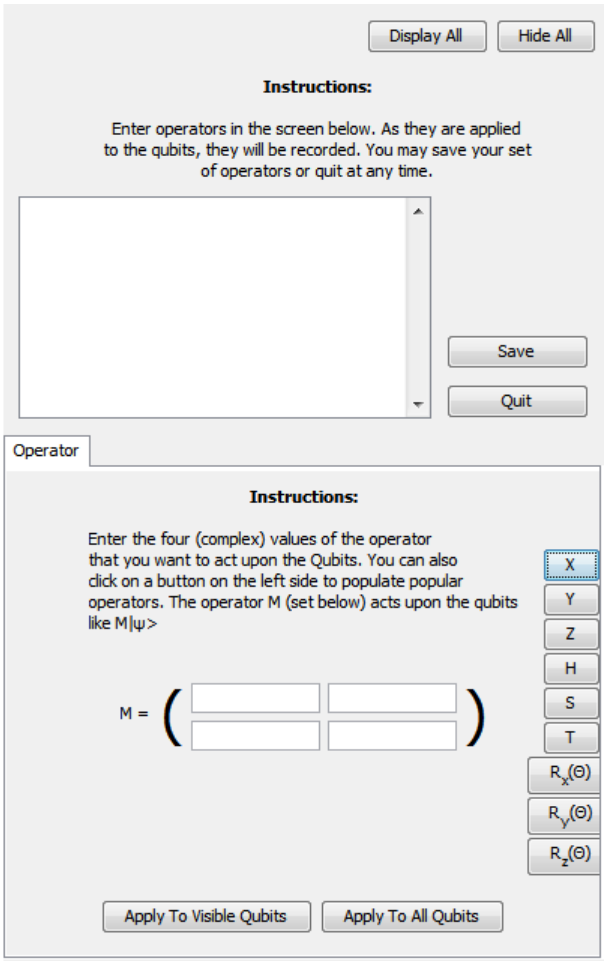

Figure 3.14: The right hand side of the simulator when in recording mode.

manually by typing it into a text editor, or by creating a custom program that creates the operators. There is no theoretical limit to the number of operators that may be recorded. In initial development, several hundred were tested and performed well.

The second part of functionality described is the playback function. When the user clicks on "Playback", they will have to select a file to playback. Currently, the user must select a file that has the extension ".bso". This stands for Bloch Sphere operators. When a file has been selected, the playback tab will be displayed in the bottom right. Figure 3.15 shows the state of the playback when File 1 is loaded.

File 1 An example of recorded operators saved to a file.

```
operator.0=1,0,0,1
operator.1=-i,0,0,i
operator.2=0,1,-1,0
```

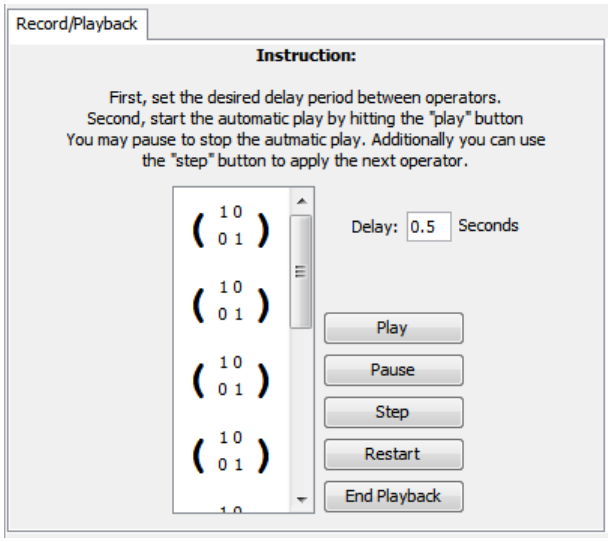

Figure 3.15: The right hand side of the simulator when in recording mode.

After the recorded operators have been loaded, there are several things that can be done to play back the operators. The first is the delay. The user can set the delay between the acting on the qubits. The standard delay is half a second. That means that the application will wait one half of a second between applying the current operator and the next operator. The delay can be changed as long as the application is not playing back the operators. To start the playback, the user should click the "Play" button. This will apply the first operator and then wait the delay before applying the next operator and continuing on. At any time, the user can pause the playback by hitting the "Pause" button. The pause button is only enabled though when the operators are being played back. Additionally, if the playback is paused, then the user can manually step through each of the operators by hitting the "Step" button. This will apply the next operator in the playback sequence. The operator

application sequence can be set back to the first operator by hitting the "Restart" button. When playback is finished, the user should click the "End Playback" to return to the normal simulator functionality.

The record playback function allows users to view visually the evolution of the qubits as quantum operators are acted on them. The record and playback functionality gives the user the flexibility to apply any number of operators without restriction on the operators themselves. The simple format of the saved file also allows the user to translate the output of other applications to the Bloch sphere simulator to view that effect.

3.1.9 Using the Larmor precession tab

The Larmor precession allows the user to see how the state of a qubit is affected by a constant magnetic field. The magnetic field in this simulator points from the origin, along the Z axis, to Z+. The user has the ability to alter the field and view in real time the effects of that field. Figure 3.16 shows the tab that displayed on the simulator when the user selects the "Larmor Precession" tab. There are four controls that are available. The first is the magnetic force strength slider. This slider allows the user to slide the scale from 0.1 to 10 Teslas. As the strength of the magnetic field is increases, the angular velocity of the qubits will increase.

The precession does not start until the user hits the "Start" button. At this point, the effect of the constant magnetic field will be seen on the qubits. The user may continue to use the slider to change the strength of the magnetic field. When the user is finished applying the magnetic field to the qubits, the "Stop" button should be clicked. The stop button will immediately end the effect of the Larmor precession, but will keep the qubits in their last state. Clicking on the "Restart" button will bring the qubits back to their original state when the precession was last started.

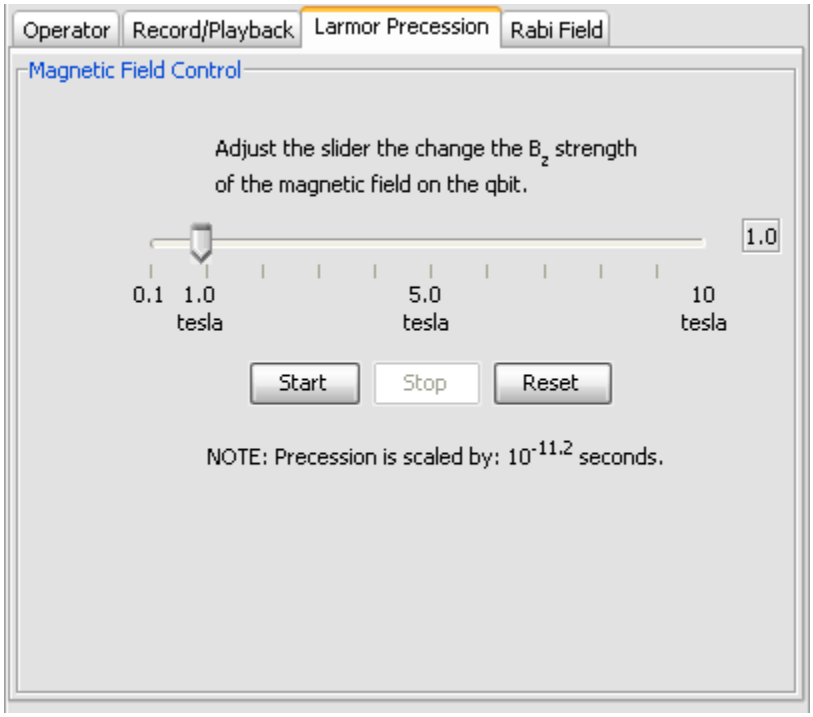

Figure 3.16: The Larmor Precession tab.

3.1.10 Using the Rabi field tab

The Rabi Field tab allows the user to see how the state of qubit is affected by two magnetic fields. The one rotates around the z-axis, on the x-y plane pointing towards the origin, and the second is a field the magnetic field along the z-axis that points from toward to the origin from Z+. There are three sliders on the tab when selecting the Rabi Field. Figure 3.17 shows the control tab on the simulator. The first slider is the B_z value which is the strength of the magnetic field that is along the z-axis. Adjusting this from low to high will make the qubit rotate around the z-axis at a faster pace. The second slider controls the strength of the perpendicular magnetic field that is rotating around the z-axis, along the x-y plane. Increasing the value of the slider will cause the qubit to rotate at a higher speed around the rotating magnetic field. This can be difficult to see if the magnetic field is rotating at a higher speed and also if the strength of the B_z is high (relatively speaking). The final

slider controls the angular velocity (ω) for the B_{perp} magnetic field around the z-axis. This is measured in radians/second.

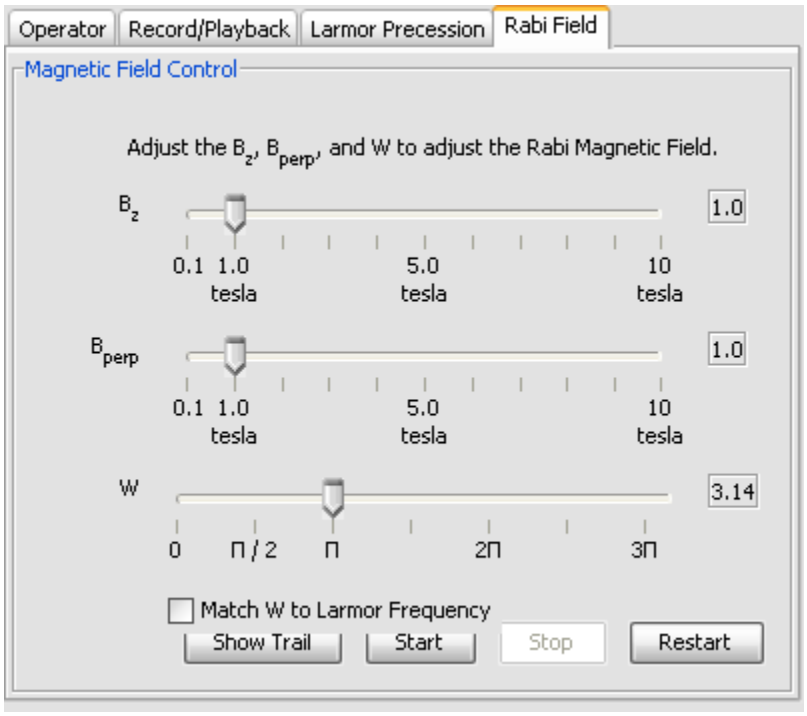

Figure 3.17: The Rabi Field tab.

There is a check box that allows the simulator to match the speed of the rotation of the B_{perp} to the Larmor frequency. This is the frequency at which the combined force of B_{perp} and B_z are matched in such a way that the B_z pushes the qubit around the z-axis at the same speed of the B_{perp} magnetic field causing it to "flip" or move from the Z+ and Z- poles. When the user checks this check box, the simulator will calculate the correct value of ω and will change it correctly. If the B_z value is changed, the ω value will be updated accordingly.

The last control does not affect the simulation of the Rabi Field, but it allows the user to view the path of the qubit, affected by the two magnetic fields. When the user clicks on the button "Show Trail", it will become depressed and the length of the trail, or previous states of the qubit will be shown. Because of computing constraints, the trail is limited and

will continually follow the path of the qubit. Figure 3.18 shows the evolution of the qubit where B_z, B_{perp} have the value of 1 Tesla, and ω with the value of 1 rad/sec.

To start the effects of the Rabi field, click on the "Start" button. This will apply the two magnetic fields on only the visible qubits in the simulator. If the slider values are changed, they will take effect immediately. One this to note is that the effect of the magnetic fields is the change in the qubit from its original state. Changing the slider may cause the qubit(s) to move in a sudden or jerky motion. This is the because the the calculation of the current qubit state is the effect of the magnetic field with the spinor values applied from the start time to the current time. If there are problems viewing the trail of the spinor, it may be best to "hide" the trail of the qubit and then "show" it later after the spinor values have been set.

Lastly, the probability that the qubit will "flip" from the north and south, along the z-axis pole when matching ω to the Larmor frequency is greatest when the qubit starts at the state of $|1\rangle$ or $|0\rangle$. The simulator will assume that the starting state of the qubit is its current state when the "Start" button is hit. If the Rabi Field simulation is stopped and started multiple times, each time it is started, the simulator will assume that the original state of the qubit is the last place that the Rabi Field stopped it. This means that the Rabi field cannot be paused. Since the starting state affects the overall path of the qubit, the qubit must be put at the original state to replay the same evolution.

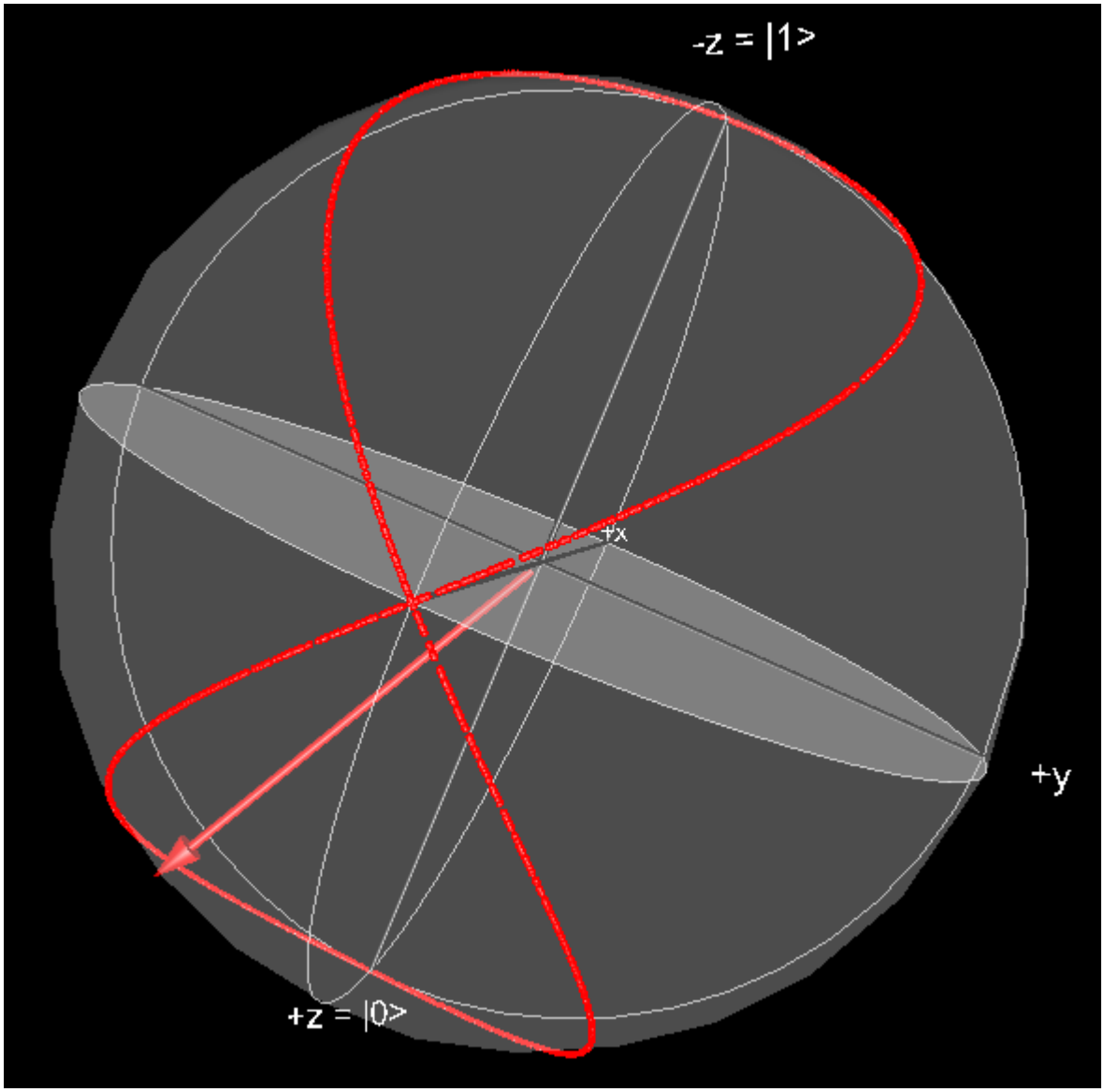

Figure 3.18: The Rabi Field shown with a trail where B_z and B_{perp} have the value of 1 Tesla, and ω with the value of 1 rad/sec. The trail of red dots show the previous path of the qubit.

3.2 Technologies Used

3.2.1 Java

The simulator is written in is Java. This is a popular programming language that is used in application and server environments. It has a philosophy of "compile once, run everywhere." This means that the code can be compiled in Java byte codes and then run on many different operator systems. All Java programs are pre-compiled in the the byte codes and then the byte codes are interpreted by a "virtual machine" to execute the code in a native environment.

The simulator consists of many class files. Each of the class files represent a class (or object in object oriented language terms) and is be grouped together into one file that becomes more easy to manage. The Java archive file (JAR) is a compressed archive of the the class files that are used to execute the simulator in the Java virtual machine (JVM). The simulator has all of the images, HTML help files, and Java class files into one jar: `Bloch3d.jar`. The simulator relies on other technologies such as Java3D (discussed in the next section) to correctly display the Bloch sphere in a 3D environment.

When running a Java program in the JVM, a list of locations is used to allow the program to fetch classes and resources. The path or listing of available files is called the class path and the JVM will look for the first class file or resource on the list that will satisfy the request. If there are multiple copies available then the first one specified on the class path will be used. If any of the classpath entries are in JARs or folders, then the JVM will search inside of those archives.

3.2.2 Java3D

Java 3D is the second technology that was used in the Bloch sphere simulator. The simulator uses Java 3D to display a representation of the Bloch sphere and the qubits that lie in its domain. Java3D is an extension of Java that links the 3D hardware accelerated functions of

each platform to Java functions so 3D programs may be created. Each implementation of Java3D is platform specific. The libraries link to OpenGL and DirectX available on different platforms. The Bloch sphere simulator uses the different launch scripts to allow operating system (OS) specific launch files to be used. The application contains all of the different libraries and the user must choose the correct launch script. They are seen in the table

$Windows32$	$runSimulatorWindows32.bat$
$Windows64$	$runSimulatorWindows64.bat$
$Linux32$	$runSimulatorLinux32.sh$
$Linux64$	$runSimulatorLinux64.sh$
OSX	$runSimulatorOSX.sh$

Figure 3.19: The launch files for each OS. The 32 and 64 versions of Linux and Windows are the bit widths of the word length defined by the OS.

Each launch file uses the platform specific libraries. File 2 is an example launch script to use the libraries in Windows. File 3 lists the Java3D jars to launch the simulator in the Linux OS.

File 2 The Windows (32 bit JVM) launch script to start the application.

```
@echo off
rem The path to the Java 3D installation where the lib/ext jars are located.
set JAVA_3D_PATH=./windows-32
rem  The path the OS specific binary files are located.
set JAVA_3D_LIBRARY_PATH=./%JAVA_3D_PATH%/bin

set CLASSPATH=./lib/bloch3d.jar
set CLASSPATH=%CLASSPATH%;%JAVA_3D_PATH%/lib/ext/j3dcore.jar
set CLASSPATH=%CLASSPATH%;%JAVA_3D_PATH%/lib/ext/j3dutils.jar
set CLASSPATH=%CLASSPATH%;%JAVA_3D_PATH%/lib/ext/vecmath.jar

java -Djava.library.path=%JAVA_3D_LIBRARY_PATH% edu.uc.ece.blochSphere.BlochApplication
```

The Java3D has a well defined application program interface (API) that defines how the libraries can be used. Their functionality and use does not change even though the underlying

File 3 The Linux (32 bit JVM) launch script to start the application.

```
#!/bin/bash
setenv J3D_HOME=/linux-32

export LD_LIBRARY_PATH=${J3D_HOME}/lib/i386

export CLASSPATH=./lib/bloch3d.jar
export CLASSPATH=${CLASSPATH}:${J3D_HOME}/lib/ext/j3dcore.jar
export CLASSPATH=${CLASSPATH}:${J3D_HOME}/lib/ext/j3dutils.jar
export CLASSPATH=${CLASSPATH}:${J3D_HOME}/lib/ext/vecmath.jar

java edu.uc.ece.blochSphere.BlochApplication
```

operating system does change. The 3D implementation in the native environments uses OpenGL, an established 3D display API and programming platform.

3.2.3 JNLP

Java has an architecture that allows applications to be launched through the Internet by using an XML specification that copies files from central locations (typically HTTP web servers) to client computers before they are executed on the client machine. The architecture is known as the Java Native Launch Protocol (JNLP). The other common name of the utility provided to the client is Java Web Start. One of the great features of JNLP is the extension tag that allows a set of jars and native libraries to be added to classpath. The JNLP launch program can then choose the correct set of JARs and libraries based on the client architecture and operating system.

File 4 The JNLP launch script to load the Bloch Sphere simulator.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://daffy-duck.rhod.uc.edu/" href="bloch3d.jnlp">
  <information>
    <title>Bloch Sphere Simulation</title>
    <vendor>University Of Cincinnati: College of Engineering</vendor>
    <homepage href="http://daffy-duck.rhod.uc.edu"/>
    <description>Bloch Sphere Simulation</description>
    <icon href="viewersplash.jpg" kind="splash"/>
    <offline-allowed/>
  </information>

  <security>
    <all-permissions/>
  </security>

  <resources>
    <j2se version="1.5+"/>
    <jar href="bloch3d.jar" main="true"/>
    <extension href="http://download.java.net/media/java3d/webstart/release/java3d-
    latest.jnlp"/>
  </resources>

  <application-desc main-class="edu.uc.ece.blochSphere.BlochApplication"/>
</jnlp>
```

3.3 Simulator Implementation

3.3.1 Previous work

The Bloch sphere simulator has gone through two different development phases. The first phase was done by Nick Vatatumic, David Kesler and Brian Jauch, form UC students who took the class on quantum computation taught by Professor Cahay. This was the first version of the simulator that displayed the 3D Bloch sphere. Operator were computed by rotating the qubit in the 3D space and then determines the qubit state. This strategy hard-coded many special cases and was limited to the specific operators implemented. The simulator was written as a Java application that worked only in Windows. The 3D Bloch sphere was very well developed, but the UI interface lacked advanced functionality and usability. Additional work was done by Changming Huo [20] to display the spin flip using the Rabi Field and the effect of random magnetic pulses. Unfortunately, this work was not combined with the current simulator work and did not provide documentation on using the advanced features. The previous work set a foundation of code that was moved to more advanced functionality, help and flexibility. There originally were tabs for the Larmor precession and for the Rabi Field, but both controls relied on the user to have mastered the theoretical concepts and equations. The earlier versions of the Rabi Field simulation and the Larmor Precession were therefore removed and rewritten.

3.3.2 Overview of current work

The rework of the simulator consisted of eight major changes: refactoring the code to be launched as an application, through JNLP and also as an Applet, removing out-dated third party libraries, adding save and load functionality, generalizing the qubit operators, rewriting the Larmor precession and the Rabi field, adding record and playback functionality, and adding help menus. There were also numerous bug fixes, memory leak fixes and other

changes that improved the functionality and usability of the simulator.

The Bloch simulator consists of 40 Java files containing 5,992 lines of code to create the simulator. There are additional HTML and configuration files that are used to display help text in the application. One of the features of the language is that the compiled byte codes can be run on different platforms. Sun Microsystems (now a subsidiary of Oracle), provides JVMs for Windows(7/XP/Vista/2000/2003/2008), Solaris (32 and 64 bit), Linux (32 and 64 bit), and Mac OSX. There are different versions of Java. The Bloch sphere supports the Java 2 Standard Edition (J2SE).

3.3.3 Foundational code

The changes that were made to the simulator are non-functional but it was done to correct the implementation of the qubit operators. The main move was to correct the simple operators that acted upon the qubits. The original operators were actual rotations of the 3D qubit. This does achieve the function of the operator visually but this is more difficult to translate to all operators. The first task was to focus on the manipulation of the α and β complex values that define a qubit state. This more accurately represents the qubit. In addition, the state of the qubit can be modified in a more consistent manner when affecting the values that represent the state of the qubit rather than the approximation of the qubit on a sphere.

The first step to correctly setting the state of the qubit was to use complex numbers. Appendix A shows the interface that defines the interaction of the `ComplexNumber`class. Currently there are no classes in the J2SE library that support complex numbers. Because of this, a complex number interface and class was created to complete the simple operations of addition, subtraction, multiplication, and division. Additionally, the complex conjugate operation is available. The other interesting feature of this class is the `parseString()`method as seen in the listing 3.1. This allows a string the be parsed as a complex number. The regular expression

```
[+-]?([\d]+[iI]?|[iI]?[\d]+)[+-]?([\d]+[iI]?|[iI]?[\d]+)
```

defines the allowed string. Before parsing the string, the code will change all letters to lower case and will remove all whitespace in the string. It will then try to extract the first number.

Listing 3.1: The function parseString() in the ComplexNumber class

```
/**
 * Parses the input string to set the values to a complex number. All whitespace
 * will be removed before the numberis parsed. The valid values are set in this
 * regular expression:
 *      "[+-]?([\d]+[iI]?|[iI]?[\d]+)[+-]?([\d]+[iI]?|[iI]?[\d]+)"
 *
 * @param complexNumber
 *              The string to parse. All whitespace will be ignored and the
 *              values of "i" will be case insensitive.
 * @return
 *              A complex number with the real and imaginary values set. If
 *              there are two imaginary, or two real parts to the String, then
 *              this will use the last value of the value and the other will be
 *              ignored.
 * @throws NumberFormatException
 *              If the string does not contain a valid format of the number.
 */
public static ComplexNumber parseString(String complexNumber)
                throws NumberFormatException
{
        double real = 0.0f;
        double imag = 0.0f;
        complexNumber = complexNumber.toLowerCase();
        complexNumber = complexNumber.replaceAll(" ", "");
        while(complexNumber.length() > 0)
        {
                if(isNextNumberImag(complexNumber))
                {
                        imag = parseNextNumber(complexNumber);
                }
                else
                {
                        real = parseNextNumber(complexNumber);
                }
                if(complexNumber.charAt(0) == '-' ||
                        complexNumber.charAt(0) == '+'){
                        complexNumber = complexNumber.substring(1);
                }
                // find the next occurance of '-' or '+'
                if(complexNumber.indexOf('+') != -1){
                        complexNumber = complexNumber.substring(
                                                complexNumber.indexOf('+'));
                }
```

```
                else if(complexNumber.indexOf('-') != -1){
                        complexNumber = complexNumber.substring(
                                                complexNumber.indexOf('-'));
                }
                else {
                        complexNumber = "";
                }
        }
        return new ComplexNumber(real, imag);
}
```

In Listing 3.1, `isNextNumberImag()` determines if there is an "i" in the next sequence of the string before a sign character ("-" or "+"). It will return true if there is an "i". The other function, `parseNextNumber()`,will parse the string the next sign character and return back the decimal value of the number. It will include any leading sign characters into the conversion of the number and will exclude all values of "i".

The next important foundation class is the `Operator`class. This class represents a 2x2 matrix where the four elements are complex numbers. It can be applied to other operators and it can be applied to qubits. The important uses of this class is with the customer operator and the Rabi field simulation. The Rabi Field uses four different operators from equation 2.30. This is easily accomplished using the `Operator`class. The function `multiply()` allows one operator to be multiplied by another. The listing 3.2 shows the action of the multiply operator. The operator class defines the complex numbers in the matrix as

$$\Omega = \begin{pmatrix} topLeft & topRight \\ bottomLeft & bottomRight \end{pmatrix}$$

Listing 3.2: The multiply() function in the Operator class

```
/**
 * Multiplies the current operator by the input
 * operator.
 *
 * @param operator
 *                The operator that this operator is multiplied
 *                 by.
```

```
 * @return
 *              A new Operator that is the product of this
 *          operator and the input operator.
 */
public Operator multiply(Operator operator){
  return new Operator(
        topLeft.multiply(operator.getTopLeft())
                .add(topRight.multiply(operator.getBottomLeft())),
        topLeft.multiply(operator.getTopRight())
             .add(topRight.multiply(operator.getBottomRight())),
        bottomLeft.multiply(operator.getTopLeft())
             .add(bottomRight.multiply(operator.getBottomLeft())),
        bottomLeft.multiply(operator.getTopRight())
             .add(bottomRight.multiply(operator.getBottomRight())));
}
```

Another important function is the ability to quickly test if the current operator is unitary. This is done by multiplying the operator by the complex conjugate of itself. This verifies that the user is creating a valid operator in the custom operator tab. The user can enter any complex values and this quickly checks to verify that the operation will not move any of the qubits in to an unknown or unusable state. The function `isOperatorUnitary()`in Listing 3.3 shows the function that checks if the operator is unitary.

Listing 3.3: The isOperatorUnitary() function available in the Operator.java class

```
/**
 * Checks if the current operator is a unitary operator.  It does this
 * by checking if the operator (A) is equal to I when A*A = I
 *
 * @return
 *              If the operator multiplied by the complex conjugate is equivalent
 *      to I.  It will check that each multiplied value is equivalent to
 *      expected value of I.  This does rounding due to the rounding problems
 *      and precision of the ComplexNumber
 */
public boolean isOperatorUnitary()
{
  return (
      topLeft.getComplexConjugate().multiply(topLeft)
         .add(bottomLeft.getComplexConjugate().multiply(bottomLeft))
           .equalsRounded(new ComplexNumber(1,0), 2) &&
      topLeft.getComplexConjugate().multiply(topRight)
         .add(bottomLeft.getComplexConjugate().multiply(bottomRight))
            .equalsRounded(new ComplexNumber(0,0), 2) &&
      topRight.getComplexConjugate().multiply(topLeft)
         .add(bottomRight.getComplexConjugate().multiply(bottomLeft))
```

```
            .equalsRounded(new ComplexNumber(0,0), 2) &&
        topRight.getComplexConjugate().multiply(topRight)
            .add(bottomRight.getComplexConjugate().multiply(bottomRight))
            .equalsRounded(new ComplexNumber(1,0), 2)
    );
}
```

Another useful function that is used for checking the validity of an operator is the ability to generate the adjoint of an operator. The adjoint is defined in equation 2.21. Here in listing 3.4, the function will create the adjoint of itself.

Listing 3.4: The getAdjoint() function available in the Operator.java class

```
/**
 * Creates the adjoint of the current opetor.  The current
 * value of the operator are:
 *
 *      { topLeft      topRight    }
 *      { bottomLeft   bottomRight }
 *
 *
 * @return
 *      The adjoint of the operator where the values are defined
 *  from the operator values above:
 *
 *     { topLeft.getComplexConjugate()    bottomLeft.getComplexConjugate()   }
 *     { topRight..getComplexConjugate()  bottomRight.getComplexConjugate()  }
 */
public Operator getAdjoint()
{
    return new Operator(topLeft.getComplexConjugate(),
                                        bottomLeft.getComplexConjugate(),
                                    topRight.getComplexConjugate(),
                                    bottomRight.getComplexConjugate());
}
```

With the adjoint defined, the operator class can quickly determine if the value of an operator is a Hermitian operator by checking that it is both unitary and equal to its adjoint. The simulator will use this verify that the operators used are Hermitian to verify that they are valid operators. The function `isOperatorHermitian()`provides the simple way to leverage the `getAdjoint()`and `isUnitary()`functions in the class. Listing 3.5 shows the code to check if the current operator is Hermitian.

Listing 3.5: The isOperatorHermitian() function available in the Operator.java class

```
/**
 * Determines if the current operator is Hermitian.  If an operator
 * is Hermitian, then it is both unitary and the operator is equal
 * to it's adjoint.
 *
 * @return
 *          true if the current values of the operator make the operator
 *      both unitary and Hermitian (the operator is equal to it's adjoint).
 */
public boolean isOperatorHermitian(){
        return isOperatorUnitary() &&
               this.equals(getAdjoint());
}
```

One of the new features of the simulator is that it can be run as an application and also as an applet. Originally, the simulator was created as an applet. One of the disadvantages to this approach was that there was no ability to run the simulator without Internet connection or if the server providing simulator was down. Additionally, applets tend to be used more for small applications and can take longer to load when running from a remote server. The improvement was to provide an applet and a stand alone application that would use the same code. This was done by moving majority of the code that controls the UI interactions to the `Bloch3D.java` class. Once that was completed, the `Bloch3D` class can be reused in both the application and the applet. The application uses the Java `JFrame` class object to display the simulator and all of the UI components. The applet uses the `JApplet` class object to display the simulator in a web browser. The code for the two methods becomes smaller, allowing the vast majority of the simulator code to work on both. Listing 3.6 shows the modified JApplet container class that displays the Bloch sphere simulator. Listing 3.7 shows the coding for displaying the Bloch sphere simulator as a stand-alone application.

Listing 3.6: The BlochApplet class that contains the code to display the simulator in an applet

```
public class BlochApplet extends JApplet {

        private static final long serialVersionUID = -4523676628636348991L;
        Bloch3D bloch3d;
        /**
```

```
 * A one-time initialization that will
 * create the bloch sphere simulator and
 * the menu bar associated with it.
 */
public void init() {
        bloch3d = new Bloch3D(this.getContentPane());
        setJMenuBar(new BlochMenuBar(bloch3d).getMenuBar());
}
/**
 * This is run when the applet has finished
 * initalizing and is read to start.
 */
public void start() {
        getContentPane().setVisible(true);
}
/**
 * Run when the applet is no longer to be
 * run.
 */
        public void stop() {
                getContentPane().setVisible(false);
                bloch3d = null;
        }
}
```

Listing 3.7: The BlochApplication class that displays the Bloch sphere simulation in an stand-along application

```
public class BlochApplication {

private static final long serialVersionUID = 5384905380645652345L;
/**
 * The constructor that will create the visual
 * window (JFrame) object and display it.
 *
 */
public BlochApplication() {
        try {
                UIManager.setLookAndFeel(UIManager
                                        .getSystemLookAndFeelClassName());
        } catch (Exception e) {
                e.printStackTrace();
        }
        // run this code no matter what.
        finally {
                JFrame blochFrame = new JFrame();
                Bloch3D blochSim = new Bloch3D(blochFrame.getContentPane());
                blochFrame.setTitle("Bloch Sphere Simulator");
                blochFrame.setSize(900, 700);
                blochFrame.setLocation(0, 0);
                blochFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
                BlochMenuBar menuBar = new BlochMenuBar(blochSim);
                blochFrame.setJMenuBar(menuBar.getMenuBar());
                blochFrame.setVisible(true);
        }
}

/**
 * The standard main function used to launch
 * a stand-alone application.
 *
 * @param args
 *                The arguments passed into the application.
 *    There are no used arguments for this application.
 */
public static void main(String[] args) {
        new BlochApplication();
}
}
```

Both listings show the use of the Bloch3D class which controls the UI of the 3D Bloch sphere and also the controls on the right side of the application.

3.3.4 Help dialogs

One of the important aspects of the simulator is to help others understand how quantum gates and operators affect the state of the qubit. This being said, the help dialogs were created to further provide instruction and demonstration on the function of the operators. The help dialogs show the rotations of the qubits based on the operator that affects it. Though the operators do not put the qubits into partial rotation states as they are applied, the help dialogs show the path of rotation, further aiding in the idea of the rotation.

The help dialogs were created using a base template and all implementing classes of that template would only need to provide the rotation animation of the qubit state, and a reference to the HTML that displays the information about the quantum operator. The operator template is setup into two different pieces, the rotation animation, and the HTML help information. Figure 3.20 shows the two different sections. The rotation animation is a small 3D Bloch sphere that displays a single qubit and shows the action of the operator as

a slow motion rotation (or series of rotations).

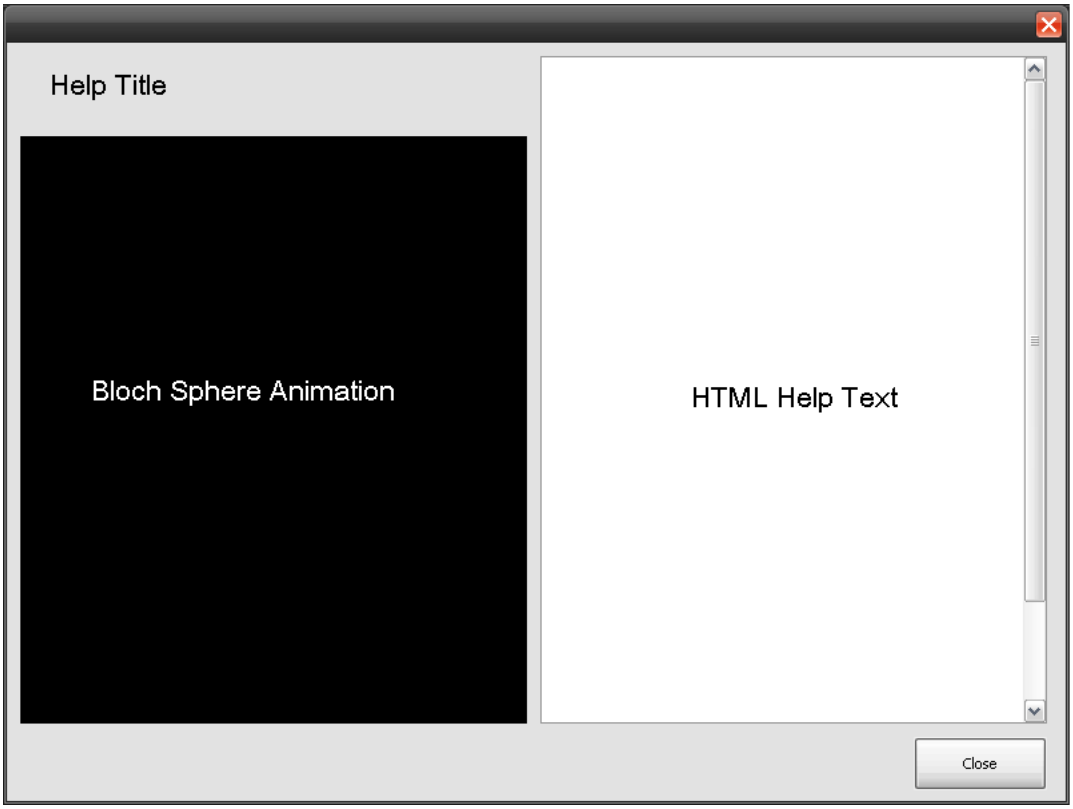

Figure 3.20: The template of the help dialog.

The code design of help dialogs is displayed in the UML diagram referenced in Fig. 3.21. This shows the package structure, the base class (the `ExampleDialog.java` class) and the child classes.

The parent, or base class, that creates the dialog and all of the code associated with the UI interaction is described in Appendix B. This code defines the dialog, the placement of the title, the 3D Bloch sphere, and the scrolling text pane. There are three abstract methods associated with this class. The abstract method defines that any concrete, or class that can be instantiated with a `new` operator in Java must have these methods in that class. The abstract keyword allows the abstract class to define the function signature (inputs, outputs, name). Using this, the abstract class can use these functions assuming that they will be

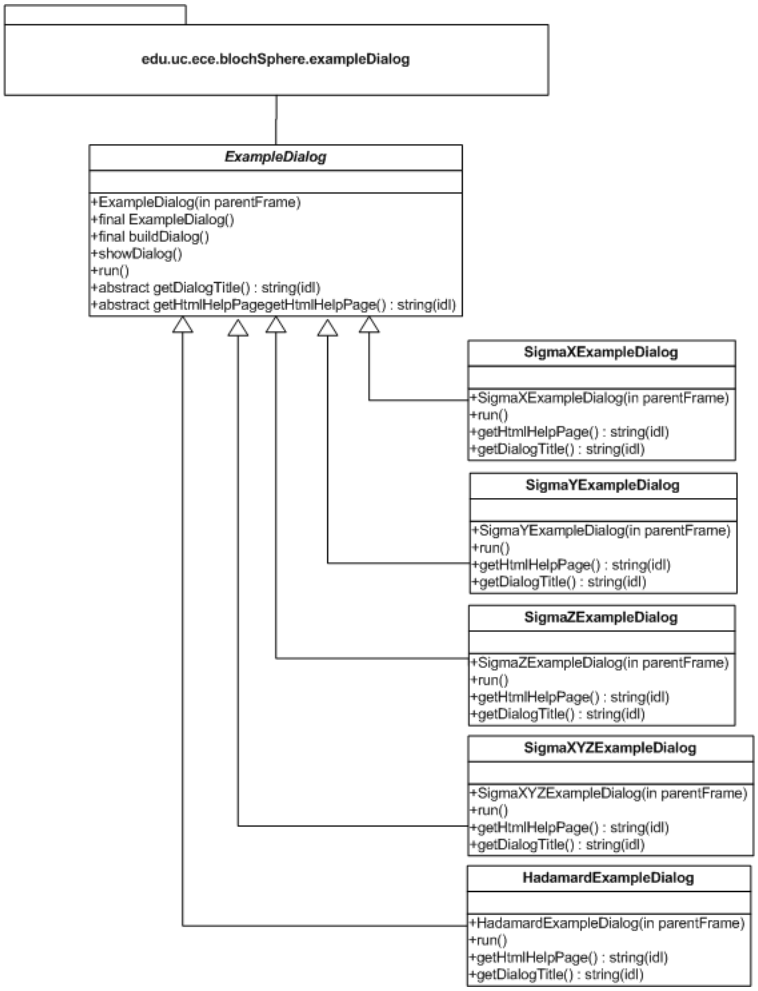

Figure 3.21: The UML diagram of the example help template and child classes that were used.

implemented by a concrete, child class. Listing 3.8 shows the function signatures for the `ExampleDialog` class.

Listing 3.8: The abstract methods associated with the ExampleDialog.java class

```
public abstract class ExampleDialog
                implements ActionListener,
                Runnable, IBlochDialog {

...

/**
 * This function should be implemented to show the animation of the example.
```

```
 * It should continue and stop when m_dialog.isVisible() is false.
 */
public abstract void run();
/**
 * This function will return the title of the help dialog that should be
 * displayed
 *
 * @return The title of the help dialog.
 */
public abstract String getDialogTitle();
/**
 * Defines the html page that should be used in the display of the help
 * dialog. All dialogs should have the html page displayed in the
 * edu/uc/ece/blochSphere/docs folder of the bloch3d.jar
 *
 * @return The name of the HTML file (including the file extension). The
 *         folder is assumed.
 */
public abstract String getHtmlHelpPage();
```

The abstract functions from are then implemented in a class that inherits from the `ExampleDialog` class. The listing 3.9 shows the small amount of code that is required to implement the σ_y operator animation. The animation shows a rotation of the qubit around the Y-axis by an angle of $\frac{\pi}{2}$.

Listing 3.9: The σ_y help dialog that extends from the ExampleDialog class

```
/**
 * The help dialog that displays the Sigma Y operator.
 * This will show the information about the operator and
 * will also demonstrate a rotation around the Y axis by
 * PI/2, or 180 degrees.
 *
 * @author Stephen Shary
 *
 */
public class SigmaYExampleDialog extendsExampleDialog {

protected static boolean m_animationRun = false;

public SigmaYExampleDialog(
                Frame parentFrame) {
        super(parentFrame);
}

@Override
public String getDialogTitle() {
```

```
        return "Sigma Y Example";
}

@Override
public String getHtmlHelpPage() {
        return "sigmay.html";
}

public void run() {
        try {
                // Wait for the dialog to be displayed initially.
                while (!dialog.isVisible()) {
                        Thread.sleep(100);
                }
                qubitModel.setVisible(true);
                while (dialog.isVisible()) {
                        qubitModel.resetAxisRotations();
                        Thread.sleep(1000);
                        // Go through 180 degrees.
                        for (int i = 0; i < 180; i++) {
                                qubitModel.doYAxisRotation(i);
                                qubitModel.finishRotation();
                                Thread.sleep(20);
                        }
                        Thread.sleep(1000);
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}
}
```

3.3.5 Qubit operator

Using the `Operator` and the `ComplexNumber` classes, we have the code foundation to quickly and accurately define and modify the qubit states. This is done in the custom operator tab of the simulator. The simulator provides the ability for the user to enter in the complex number values into the four fields for the operator. The class that provides the UI and the functions to apply common operators is the qubit `Operator` class. This has a 2x2 grid of text fields that allows the user to enter in custom operators that are applied to the qubits. Since the `Operator` class is available, there are convenient functions to allow the verification of a valid operator and its application to the qubits in the simulator.

To apply the common operators that are available, the simulator will simply fill out the 2x2 grid of values that define the operator. Each of the buttons that are on the form are registered with the `QubitOperator` class so that the class can handle the click action and fill in the correct values. Listing 3.10 shows the `actionPerformed()` function that fills out the values in the grid when the button has been pressed. The function contains the hard-coded values of each of the operators.

Listing 3.10: The actionPerformed() function in the QubitOperator class.

```
/**
 * This is called when a click event is fired on one of the custom operator.
 */
public void actionPerformed(ActionEvent e)
{
        ...

        else if(e.getSource().equals(this.xOperatorButton)){
                setOperatorvalues("0","1","1","0");
        }
        else if(e.getSource().equals(this.yOperatorButton)){
                setOperatorvalues("0","-i","i","0");
        }
        else if(e.getSource().equals(this.zOperatorButton)){
                setOperatorvalues("1","0","0","-1");
        }
        else if(e.getSource().equals(this.hOperatorButton)){
                setOperatorvalues("0.7071","0.7071","0.7071","-0.7071");
        }
```

```
        else if(e.getSource().equals(this.sOperatorButton)){
                setOperatorvalues("1","0","0","i");
        }
        else if(e.getSource().equals(this.tOperatorButton)){
                setOperatorvalues("1","0","0","0.707+0.707i");
        }

        ...
}
```

There are three available operators that are general rotation operations. The user can choose between a rotation around the X, Y, or Z axis. When the user selects one of the rotation operators, the simulator then prompts the user to enter the magnitude of the rotation. This is entered in radians. Listing 3.11 shows the code to enter the custom angle and listing 3.12 shows the section of the `actionPerformed()` that handles the input of the angle and then the calculates the operator values.

Listing 3.11: The getRotationAngle() function in the QubitOperator class.

```
/**
 * Prompts the user to enter the rotation angle (in radians). This will
 * then parse and re-prompt the user to enter and angle if there are
 * any problems.
 *
 * @return
 *              A double value of the rotation angle. If the user has canceled
 *      from the process, then this indicates that the user wants to
 *      cancel the operator action.
 */
private Double getRotationAngle(){
        Double angle = null;
        while(angle == null){
                String angleString = (String)JOptionPane.showInputDialog(
                this.getControlPanel(),
                "Please enter the rotation angle (in radians)",
                "Enter Rotation Angle",
                JOptionPane.OK_OPTION);
                if(angleString == null){
                        return null;
                }
                try{
                        angle = Double.parseDouble(angleString);
                }
                catch(NumberFormatException nfe){
                        JOptionPane.showMessageDialog(this.getControlPanel(),
                        "Please enter a number to represent the angle (in radians).");
```

```
                }
        }
        return angle.doubleValue();
}
```

Listing 3.12: The actionPerformed() function in the QubitOperator class showing the action of the custom rotation operators.

```
/**
 * This is called when a click event is fired on one of the custom operator.
 */
public void actionPerformed(ActionEvent e)
{
        ...
        else if(e.getSource().equals(this.rXOperatorButton)){
                Double angle = getRotationAngle();
                if(angle == null){
                        return;
                }
                ComplexNumber topLeft =
                        new ComplexNumber(Math.cos(angle / 2), 0);
                ComplexNumber topRight =
                        new ComplexNumber(0, -1 * Math.sin(angle / 2 ));
                ComplexNumber bottomLeft =
                        new ComplexNumber(0, -1 * Math.sin(angle / 2 ));
                ComplexNumber bottomRight =
                        new ComplexNumber(Math.cos(angle / 2), 0);
                setOperatorvalues(topLeft.toString(3),
                                  topRight.toString(3),
                                  bottomLeft.toString(3),
                                  bottomRight.toString(3));
        }
        else if(e.getSource().equals(this.rYOperatorButton)){
                Double angle = getRotationAngle();
                if(angle == null){
                        return;
                }
                ComplexNumber topLeft =
                        new ComplexNumber(Math.cos(angle / 2), 0);
                ComplexNumber topRight =
                        new ComplexNumber( -1 * Math.sin(angle / 2 ), 0);
                ComplexNumber bottomLeft =
                        new ComplexNumber(Math.sin(angle / 2 ), 0);
                ComplexNumber bottomRight =
                        new ComplexNumber(Math.cos(angle / 2), 0);
                setOperatorvalues(topLeft.toString(3),
                                  topRight.toString(3),
                                  bottomLeft.toString(3),
                                  bottomRight.toString(3));
        }
        else if(e.getSource().equals(this.rZOperatorButton)){
```

```
                Double angle = getRotationAngle();
                if(angle == null){
                        return;
                }
                ComplexNumber topLeft =
                        new ComplexNumber(Math.cos(angle / 2), -1 * Math.sin(angle / 2));
                ComplexNumber topRight =
                        new ComplexNumber(0, 0);
                ComplexNumber bottomLeft =
                        new ComplexNumber(0, 0);
                ComplexNumber bottomRight =
                        new ComplexNumber(Math.cos(angle / 2), Math.sin(angle / 2));
                setOperatorvalues(topLeft.toString(3),
                                  topRight.toString(3),
                                  bottomLeft.toString(3),
                                  bottomRight.toString(3));
        }

        ...
}
```

All of the coding above has shown how the custom operator values are set into the UI and allow them to be applied to the qubits. The code below will show how the qubit operators are applied to the change the qubit values. There are three functions that are used to apply an operator. There is the `getOperatorFromUI()`function, shown in listing 3.15 that fetches the complex values from text fields on the UI. This does a complex number validation making sure that all of the four fields are valid complex numbers. This uses the functions in the `ComplexNumber` class to parse the strings and make a determination if the String values are valid. The second function is the `checkOperator()`function. This function, shown in listing 3.16, will look at the four complex numbers fetched from the UI and determine if together, they form a valid quantum operator can be created. The validation that is done uses the `Operator` class to verify that the operator is unitary. The last is the `applyOperation()` function takes the list of qubits and applies the operator that was created. Listing 3.14 shows the code that used to complete the `applyOperation()` function. When the code has arrived at the `applyOperation()` function, it is assumed that all applicable checks and validations have been completed. Listing 3.13 shows the code the orchestrates all of these

functions.

Listing 3.13: The actionPerformed() that shows the application of the qubit operations.

```
/**
 * This is called when a click event is fired on one of the custom operator.
 */
public void actionPerformed(ActionEvent e)
{
        if(e.getSource().equals(applyToVisible)){
                if(checkOperator()){
                        applyOperation(getApplicableQubits(true));
                }
        }
        else if(e.getSource().equals(this.applyToAll)){
                if(checkOperator()){
                        applyOperation(getApplicableQubits(false));
                }
        }

        ...
}
```

Listing 3.14: The function applyOperation()

```
/**
 * This will apply the operator on the each of the qubits that
 * are supplied in the Vector.  If the Vector is empty, then the
 * operation will not be done.
 *
 * @param qubits
 *              The qubits that should have the operator applied to them.
 */
private void applyOperation(Vector<Qubit>qubits){
        Operator operator = getOperatorFromUI();
        for(Qubit qubit : qubits){
                ComplexNumber qubitAlphaNum =
                        new ComplexNumber(qubit.m_alphaValue,0);
                ComplexNumber qubitBetaNum =
                        new ComplexNumber(qubit.m_betaRealValue,
                                        qubit.m_betaImaginaryValue);
                ComplexNumber productAlpha =
                        operator.applyOperatorToAlpha(qubitAlphaNum,
                                                qubitBetaNum);
                ComplexNumber productBeta =
                        operator.applyOperatorToBeta(qubitAlphaNum,
                                                qubitBetaNum);
                qubit.setAlphaBetaFullValues(productAlpha.getRealPart(),
                                        productAlpha.getImaginaryPart(),
                                        productBeta.getRealPart(),
                                        productBeta.getImaginaryPart());
```

```
        }
}
```

Listing 3.15: The function getOperatorFromUI()

```
/**
 * Fetches the operator from the values that are set on the UI.
 * If the values are not well formed, then this will show a pop-up
 * dialog with the value that is malformed.
 *
 * @return
 *              The operator represented by the values set on the UI.  If any
 *      of the values are not valid complex numbers then this will
 *      return null.
 */
public Operator getOperatorFromUI()
{
        String errorValue = null;
        try     {
                errorValue = "Alpha";
                ComplexNumber alphaNum = new ComplexNumber(alphaTextField.getText());
                errorValue = "Beta";
                ComplexNumber betaNum = new ComplexNumber(betaTextField.getText());
                errorValue = "Gamma";
                ComplexNumber gammaNum = new ComplexNumber(gammaTextField.getText());
                errorValue = "Delta";
                ComplexNumber deltaNum = new ComplexNumber(deltaTextField.getText());
                return new Operator(alphaNum, betaNum, deltaNum, gammaNum);

        }
        catch(NumberFormatException ne)
        {
                JOptionPane.showMessageDialog(this.mainControlPanel,
                    errorValue + " value must be in the form A+Bi or A-Bi",
                    "Alpha Number Error",
                    JOptionPane.ERROR_MESSAGE);
                return null;
        }
}
```

Listing 3.16: The checkOperator() function.

```
/**
 * An internal check to verify that the operator is non-null, all
 * entries are well formed complex numbers and that the operator
 * is unitary.
 *
 * @return
 *              true if the operator is non-null, well-formed, and that the
 *      operator values make the operator unitary.
 *      otherwise false.
```

```
 */
private boolean checkOperator() {
        Operator operator = getOperatorFromUI();
        if(operator == null){
                return false;
        }
        else if(!operator.isOperatorUnitary()){
                JOptionPane.showMessageDialog(this.mainControlPanel,
                                              "The operator must be Unitary.",
                                              "Operator Value Error",
                                              JOptionPane.ERROR_MESSAGE);
                return false;
        }
        return true;
}
```

3.3.6 Larmor precession code

The Larmor Precession is the first of the two time evolution simulations that were implemented. This simulation shows the evolution of the qubit state as a constant magnetic force is applied along the Z-axis. The UI for this simulation is fairly simple: a single slider to determine the strength of the magnetic field and a start, stop and restart button to control the simulator. The class `LarmorPrecession` contains the code for the UI and the qubit interaction when the precession is started. The class contains two main functions that handle the control of the simulation and the calculation of the qubit states during the simulation. The `LarmorPrecession` implements the `Runnable` interface. Implementing this interface means that this class will have a `run()` function. This allows a new thread to be created in the JVM that will execute the `run()` function. The `run()` function will continuous run until the function `getApplyField()` returns false. This function simply stores the boolean variable that defines if the precession should be running. When the simulation is started, then the `setApplyField(true)` function is called. When the stop button is hit, then the `setApplyField(false)` is called. The run() method will then detect the change in the value after a single simulation step and will stop. The `actionPerformed()` function, shown

in listing 3.17, shows the actions of the different button actions. ‘

Listing 3.17: The actionPerformed() function in the LarmorPrecession class

```
/**
 * Receives all events from the registered components on the panel and
 * processes their actions.
 *
 * @param e
 */
public void actionPerformed(ActionEvent e) {
        Object component = e.getSource();
        // If we hit start, then only start when it isn't already running.
        if (component.equals(m_startButton)) {
                if (!getApplyField()) {
                        setApplyField(true);
                        new Thread(this).start();
                }
        }
        // If we hit stop, then set the ApplyField to false.
        if (component.equals(m_stopButton))
                setApplyField(false);
        // If we hit the reset button, then stop the field, and set the value to
        // 1.0 tesla (10 on the JSlider).
        if (component.equals(m_resetButton)) {
                setApplyField(false);
                m_teslaSlider.setValue(10);
                for (int i = 0; i < qbitVector.size(); i++) {
                        Qubit qbit = (Qubit) qbitVector.get(i);
                        // Set the qubit phi value back to the original.
                        qbit.setPhi(Math.toDegrees(((Double) m_originalPhi.get(i))
                                        .doubleValue()));
                }
        }
}
```

The `run()` function applies the new ϕ state to each of the qubit based on the strength of the magnetic field. The precession is calculated through several constants and through the magnetic field strength. The spin of the qubit is then scaled down to approximately to 10 radians per second. Listing 3.18 shows the constants in the class and the `run()` function that updates the qubit values as the precession occurs.

Listing 3.18: The code in the LarmorPrecession class to apply magnetic field to the qubit(s)

```
public static final double G_STAR = 2.002319;
public static final double M_0 = 9.1 * Math.pow(10, -31);
public static final double E = 1.6 * Math.pow(10, -19);
```

```
public static final double H_BAR = 6.582 * Math.pow(10, -16);
public static final double W_L = (E * G_STAR) / (2 * M_0) ;
public static final double TIME_SCALE_FACTOR = Math.log10(W_L);

...

/**
 * The thread spawned function that will run to apply the magnetic field.
 */
public void run() {
        long startTime = Calendar.getInstance().getTimeInMillis();
        m_originalPhi = new Vector<Double>();
        for (int i = 0; i < qbitVector.size(); i++)
                m_originalPhi.add(new Double(Math.toRadians(
                                ((Qubit) qbitVector.get(i)).getPhi())));
        while (getApplyField() && qbitVector != null
                        && qbitVector.size() > 0) {
                double deltaTime = new Long(Calendar.getInstance()
                                .getTimeInMillis()
                                - startTime).doubleValue() / 1000;
                double deltaPhi = deltaTime * W_L
                                * Double.parseDouble(
                                        m_BzSliderValueLabel.getText())
                                * Math.pow(10, -TIME_SCALE_FACTOR);
                for (int i = 0; i < qbitVector.size(); i++) {
                        Qubit qbit = (Qubit) qbitVector.get(i);
                        // Only apply to visible qbits.
                        if (!qbit.isVisible())
                                continue;
                        // Set the new value and remember to convertto degrees.
                        qbit.setPhi(Math.toDegrees(
                                        m_originalPhi.get(i) - deltaPhi));
                        // We sleep for 1 millisecond to give the CPU a break.
                        // also improves the UI responsiveness.
                        try {
                                Thread.sleep(1);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

3.3.7 Rabi field code

The Rabi Field is the second of the two time evolution simulations that were implemented in this simulator. This is similar in implementation to the Larmor Precession, using similar

functions, but the implementation uses more foundation classes as the simulation is much more complex. The Rabi field contains three sliders and a check box to vary the different parameters of the simulation. The most complicated part of the Rabi Field simulation is the calculation of the correct operator. This is done by taking the slider values and then creating equation 2.112. The code in listing 3.19 shows the creation of the `Operator` object that can be used to set the correct qubit state at Δt. The input parameters are for the operator are the three slider values ω, β_z, and β_{perp}. Along with Δt, the inputs can create the operator that will calculate the correct state of the original qubit.

Listing 3.19: The calculateOperator() function in the Rabi Field class

```
/**
 * Calculates the correct operator that will move the qubit from the original
 * state to the state at deltaTime based on the input parameters.
 *
 * @param omega
 *          The rotational speed of the b_perp magnetic field along the
 *          XY-axis.
 * @param beta_perp
 *          The strength (in Telsas) of the magnetic field that is rotated
 *          around the XY-axis.
 * @param beta_z
 *          The strength (in Telsas) of the magnetic field that is run along
 *          the Z-axis.
 * @param deltaTime
 *          The change in time from the original state of the qubit(s).
 *
 * @return The 2x2 Operator object that represents the rotation of the
 *         original qubit states to their correct placement.
 */
private Operator calculateOperator(double omega,
                                   double beta_perp,
                                   double beta_z,
                                   double deltaTime) {
  double W = omega;
  double W1 = beta_perp;
  double W0 = beta_z;
  if (matchFrequencies.isSelected()) {
    W = Math.sqrt(W0);
  }
  double dW = (W * W) - W0;
  Operator rOperator =
    new Operator(
        new ComplexNumber(Math.cos(W * deltaTime/ 2),
```

```
                                -1 * Math.sin(W * deltaTime / 2)),
            new ComplexNumber(0, 0),
            new ComplexNumber(0, 0),
            new ComplexNumber(Math.cos(W * deltaTime / 2),
                              Math.sin(W * deltaTime / 2)));
    double b = (-1 * W1 * deltaTime) / 2;
    double a = (dW * deltaTime) / 2;
    double q = Math.sqrt((a * a) + (b * b));
    double bOverAMinusQ = (a - q) == 0 ? 0 : b / (a - q);
    double bOverAPlusQ = (a + q) == 0 ? 0 : b / (a + q);
    Operator sOperator =
      new Operator(
            new ComplexNumber(-1 * bOverAMinusQ, 0),
            new ComplexNumber(-1 * bOverAPlusQ, 0),
            new ComplexNumber(1, 0),
            new ComplexNumber(1, 0));
    Operator eLambdaOperator =
      new Operator(
            new ComplexNumber(Math.cos(q), Math.sin(q)),
            new ComplexNumber(0, 0),
            new ComplexNumber(0, 0),
            new ComplexNumber(Math.cos(q), -1 * Math.sin(q)));
    Operator sInverseOperator =
      new Operator(
            new ComplexNumber(1, 0),
            new ComplexNumber(bOverAPlusQ, 0),
            new ComplexNumber(-1, 0),
            new ComplexNumber(-1 * bOverAMinusQ, 0));
    return rOperator.multiply(
                sOperator.multiply(
                  eLambdaOperator.multiply(sInverseOperator))));
}
```

One of the interesting parts of the `calculateOperator()` function is the part that checks if the value of ω (W) is set to the square root of ω_0 (W_0). This setting will match ω to the Larmor frequency.

The animation that shows the movement of the Rabi Field is similar to the Larmor Precession implementation in that it runs on a separate thread and will continue until the boolean function `getApplyField()` return false. Since the operator that is created for the Rabi Field calculates the rotation of the qubit(s) from their original position, the original α and β values need to be saved because the values stored in the `Qubit` objects are the last state of the simulation at $\Delta t - 1$. The code function in listing 3.20 shows the `run()` function that

loops through and updates the qubit values. This calls the `calculateOperator()` function (as seen in figure 3.19) at each Δt step to determine the operator that will move each qubit from its original state to its state at time t.

Listing 3.20: The run() function in the RabiField class.

```
/*
   * Created to be run when the start button has been selected. This should
   * apply the animation to the visible qubits.
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    long startTime = Calendar.getInstance().getTimeInMillis();
    alphaValues = new Vector<ComplexNumber>();
    betaValues = new Vector<ComplexNumber>();
    for (Qubit qbit : m_qbits) {
      alphaValues.add(qbit.getAlphaValue());
      betaValues.add(qbit.getBetaValue());
    }
    double deltaTime = 0;
    while (getApplyField() && m_qbits != null && m_qbits.size() > 0) {
      deltaTime = new Long(
                Calendar.getInstance().getTimeInMillis() - startTime)
                .doubleValue() / 1000;
      double beta_z = Double.parseDouble(m_magFieldValueLabel.getText());
      double beta_perp = Double.parseDouble(m_magPerpValueLabel.getText());
      double omega = Double.parseDouble(m_omegaValueLabel.getText());

      Operator finalOperator = calculateOperator(omega,
                                                 beta_perp,
                                                 beta_z,
                                                 deltaTime);
      int counter = 0;
      for (Qubit qbit : m_qbits) {
        // Only apply to visible qbits.
        if (qbit.isVisible()) {
          ComplexNumber betaValue = alphaValues.get(counter);
          ComplexNumber alphaValue = betaValues.get(counter);
          qbit.setAlphaBetaValues(
                finalOperator.applyOperatorToAlpha(alphaValue, betaValue),
                finalOperator.applyOperatorToBeta(alphaValue, betaValue));
          if (isShowPreviousStates()) {
            addTrackDot(qbit);
          }
        }
        counter++;
      }
      // This will keep the CPU from spiking.
```

```
      try {
        Thread.sleep(2);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
```

The run function has a reference to the `TrackDot` class. This class is used to store the previous values of the qubits as they rotate around the origin. The `JCheckBox` class used in the UI allows the user to show the previous value of each qubit. The function `addTrackDot()` allows the current value of the qubit to be saved and kept in a `Vector` or list of previous values. When the the list has exceeded a certain size (in the current implementation 1600), then the list will remove the oldest point and then add the newest value to the end. This limiting is done to maintain the performance of the simulator.

Listing 3.21: The addTrackDot() function for the RabiField class

```
public static final int MAX_NUM_PREV_STATES = 1600;

...

/**
 * Adds a new "track dot" or small position dot representing
 * the previous value of the qubit.
 *
 * @param qubit
 *        The current qubit to create a new position dot for.
 */
private void addTrackDot(Qubit qubit) {
  Vector<TrackDot> trackDots = allTracks.get(qubit);
  if (trackDots == null) {
    trackDots = new Vector<TrackDot>();
    trackDots.add(
          new TrackDot(
                new Double(0.8f).doubleValue(),
                qubit.getTh(),
                qubit.getPhi(),
                this.parentBG,
                qubit.getQubitColor(),
                false));
    allTracks.put(qubit, trackDots);
  } else {
    if (trackDots.size() > MAX_NUM_PREV_STATES) {
```

```
            trackDots.get(0).kill();
            trackDots.remove(0);
        }
        Color3f color = qubit.getQubitColor();
        double magnitude = 0.8f;
        boolean big = false;
        trackDots.add(
            new TrackDot(
                    magnitude,
                    qubit.getTh(),
                    qubit.getPhi(),
                    this.parentBG,
                    color,
                    big));
    }
}
```

Chapter 4

Conclusion

The Bloch sphere simulator provides the ability for those that are studying the field of quantum mechanics and quantum computation to better understand the effects of quantum mechanical gates and the evolution of the qubit with magnetic fields. The simulator does have the ability to store the current qubit state, and also record and playback a series of quantum operations. This allows for the simulator to be expanded beyond just the simple execution of single operators. Many times a tool can best be understood an applied when used in the field. The simulator may provide additional benefits or may be able to describe qubit state changes that were not intended as part of its original design. This in no way should be discouraged, but rather encouraged.

Further expansions of this simulator can be explored down two different paths. The first is the expanded path of magnetic field interactions. The simulator can be extended to simulate the effect of random magnetic fields, magnetic field pulses. Though the Rabi field and the spin flip effect is very important and the simulator provides controls that set the strength and rotation, expanded flexibility of the magnetic fields would allow further simulations.

The second avenue to expand the work on the simulator is to integrate the visualization of

qubit states with a quantum circuit simulator. There are different quantum circuit simulators as described in Section 1.2 that can be used in conjunction with this simulator. The Bloch sphere provides a way for the state of one or more qubits to be visualized and can help others see how the state of a qubit evolves over time.

Additional enhancements can be made through integration with other tools. Like the suggestion to integrate the simulator with a quantum circuit simulator, export and import functions can be very useful. The transfer of a qubit state, or time evolution to Matlab and other mathematical programs provides the most flexible expansion of the capabilities of the simulator.

The Bloch sphere simulator provides a good basis to understand and explore quantum mechanical gates and interactions. The hope and intent is to provide a solid step for others to advance the field.

Bibliography

[1] Java 3d 1.5.2 release notes. `https://j3d-core.dev.java.net/j3d1_5_2/RELEASE-NOTES.html`.

[2] Java.com: Java + you. `http://java.com`.

[3] Dave Bacon and Wim van Dam. Recent progress in quantum algorithms. *Commun. ACM*, 53(2):84–93, 2010.

[4] Ryan S. Bennink, Carlos R. Stroud, and Robert W. Boyd. Graphical solution of coherent raman systems using the bloch sphere. In *Conference on Lasers and Electro-Optics/Quantum Electronics and Laser Science Conference*, page QTuG17. Optical Society of America, 2003.

[5] Marc Cahay and Supriyo Bandyopadhyay. *Introduction to Spintronics*. CRC Press, 2008.

[6] Simona Caraiman, Alexandru Archip, and Vasile Manta. A grid enabled quantum computer simulator. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0:189–196, 2009.

[7] N. J. Cerf, C. Adami, and P. G. Kwiat. Optical simulation of quantum logic. *Phys. Rev. A*, 57(3):R1477–R1480, Mar 1998.

[8] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74(20):4091–4094, May 1995.

[9] H. K. Cummins and J. A. Jones. Nuclear magnetic resonance: a quantum technology for computation and spectroscopy, 2000.

[10] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. 400:97–117, 1985.

[11] H. Q. Ding. Monte carlo simulations of quantum systems on massively parallel computers. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 34–43, New York, NY, USA, 1993. ACM.

[12] David P. Divincenzo. The physical implementation of quantum computation. *Fortschr. Phys*, 48:2000, 2000.

[13] Sara Felloni, Alberto Leporati, and Giuliano Strini. Diagrams of states in quantum information: an illustrative tutorial. Technical Report arXiv:0904.2656, Apr 2009. Comments: 29 pages, 25 figures, to be published in IJUC - International Journal of Unconventional Computing.

[14] Sara Felloni, Alberto Leporati, and Giuliano Strini. Evolution of quantum systems by diagrams of states. Technical Report arXiv:0912.0026, Dec 2009. Comments: 20 pages, 12 figures; to be published in IJUC - International Journal of Unconventional Computing.

[15] Richard P. Feynman. Quantum mechanical computers. *Optics News*, 11(2):11–20, 1985.

[16] Richard Phillips Feynman. *Feynman Lectures on Computation*. Perseus Books, Cambridge, MA, USA, 2000.

[17] Ian Glendinning. The bloch sphere. `http://www.vcpc.univie.ac.at/~ian/hotlist/qc/talks/bloch-sphere.pdf`, February 2005.

[18] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING*, pages 212–219. ACM, 1996.

[19] Anthony J. G. Hey, editor. *Feynman and computation: exploring the limits of computers*. Perseus Books, Cambridge, MA, USA, 1999.

[20] Changming Huo. A Bloch Sphere Animation Software using a Three Dimensional Java Simulator. Master's thesis, University of Cincinnati, Cincinnati, OH, 2010.

[21] J. A. Jones and M. Mosca. Implementation of a quantum algorithm on a nuclear magnetic resonance quantum computer. *The Journal of Chemical Physics*, 109(5):1648–1653, 1998.

[22] Hideaki Kikuchi, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Hiroshi Iyetomi, Shuji Ogata, Takahisa Kouno, Fuyuki Shimojo, Kenji Tsuruta, and Subhash Saini. Collaborative simulation grid: multiscale quantum-mechanical/classical atomistic simulations on distributed pc clusters in the us and japan. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–8, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[23] D. V. Melnikov L. X. Zhang and Jean-Pierre Leburton. Simulation of spin-qubit quantum dot circuit with integrated quantum point contact read-out. *Journal of Computational Electronics*, 4(1-2):111–114, Aug 2005.

[24] Shai Machnes. Qlib - a matlab package for quantum information theory calculations with applications. 2007.

[25] A Mandilara, J W Clark, and M S Byrd. Elliptical orbits in the bloch sphere. *Journal of Optics B: Quantum and Semiclassical Optics*, 7(10):S277, 2005.

[26] George Viamontes Manoj, Manoj Rajagopalan, Igor L. Markov, and John P. Hayes. Gate-level simulation of quantum circuits. In *Los Alamos Quantum Physics Archive, Aug. 2002 http://xxx.lanl.gov/abs/quant-ph/0208003*, pages 295–301, 2003.

[27] Mauro Marinilli. *Java Deployment: with JNLP and WebStart*. Sams, Indianapolis, IN, USA, 2001.

[28] Peter J. Mohr, Barry N. Taylor, and David B. Newell. Codata recommended values of the fundamental physical constants: 2006. *Rev. Mod. Phys.*, 80(2):633–730, Jun 2008.

[29] Roger G. Newton. *Quantum Physics: A Text for Graduate Students*. Springer, 2002.

[30] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[31] B. Odom, D. Hanneke, B. D'Urso, and G. Gabrielse. New measurement of the electron magnetic moment using a one-electron quantum cyclotron. *Phys. Rev. Lett.*, 97(3):030801, Jul 2006.

[32] J. F. Poyatos, J. I. Cirac, and P. Zoller. Complete characterization of a quantum process: The two-bit quantum gate. *Phys. Rev. Lett.*, 78(2):390–393, Jan 1997.

[33] I. I. Rabi. On the process of space quantization. *Phys. Rev.*, 49(4):324–328, Feb 1936.

[34] Ramamurti Shankar. *Principles of Quantum Mechanics*. Springer, 1994.

[35] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

Appendices

Appendix A

Interface for IComplexNumber.java

IComplexNumber.java

```
package edu.uc.ece.blochSphere;

/**
 * The representation of complex numbers. This provides the ability to add,
 * subtract, multiply, and divide by other complex numbers. There is also some
 * robust parsing functionality to import a string and convert it to a Complex
 * Number.
 *
 * @author Stephen Shary
 *
 */
public interface IComplexNumber {

        /**
         * Allows the real part of the Complex Number to be defined.
         *
         * @param real
         *             The real part of the complex number. If there is no real part,
         *             then supply 0.0f.
         */
        public void setRealPart(double real);
        /**
         * Allows the imaginary part of the complex number to be defined.
         *
         * @param imaginary
         *             The imaginary part of the complex number.
         */
        public void setImaginaryPart(
```

```
                double imaginary);
/**
 * Gets the real part of the Complex Number.
 *
 * @return The real part of the complex number. If there is not defined
 *         value, then the value will be 0.0f.
 */
public double getRealPart();
/**
 * Gets the imaginary part of the Complex Number.
 *
 * @return The imaginary part of the complex number. If there is not defined
 *         value, then the value will be 0.0f.
 */
public double getImaginaryPart();
/**
 * Returns a new ComplexNumber object where the real part of the number is
 * unaffected and the imaginary part is multiplied by "-1", effectively
 * changing the sign of the imaginary part of the complex number.
 *
 * @return A new ComplexNumber object where the new object is the complex
 *         conjugate of the current value of this object.
 */
public ComplexNumber getComplexConjugate();
/**
 * Returns a new complex number where the value of this complex number is
 * added to the input parameter.
 *
 * @param compNum
 *            The complex number that should be added to this number.
 * @return The sum of the two complex numbers. If the input number is null,
 *         then this will return the a new complex number object with the
 *         values of this.
 */
public ComplexNumber add(
                ComplexNumber compNum);
/**
 * Returns a new complex number where the value of this complex number is
 * subtracted by the input parameter.
 *
 * @param compNum
 *            The complex number that should be subtracted from this number.
 * @return The difference of the two complex numbers. If the input number is
 *         null, then this will return the a new complex number object with
 *         the values of this.
 */
public ComplexNumber subtract(
                ComplexNumber compNum);
/**
 * Returns a new complex number where the value of this complex number
 * multiplied by the input parameter.
```

```
     *
     * @param compNum
     *            The complex number that should be multiplied by this number.
     * @return The product of the two complex numbers. If the input number is
     *         null, then this will return the a new complex number with a value
     *         of zero (for real and imaginary parts).
     */
    public ComplexNumber multiply(
                    ComplexNumber compNum);
    /**
     * Returns a new complex number where the value of this complex number is
     * divided by the input parameter.
     *
     * @param compNum
     *            The complex number that this number will be divided by.
     * @return The result of dividing this by the input complex number. If the
     *         input number is null, then this will return the a new complex
     *         number with a value of zero (for real and imaginary parts).
     */
    public ComplexNumber dividedBy(
                    ComplexNumber compNum);
    /**
     * Determines if the input number is the complex conjugate of this.
     *
     * @param conjugate
     *            The input number to check if it is the complex conjugate.
     * @return true if the input complex number is non-null and it is the
     *         complex conjugate of the current value. Otherwise, false.
     */
    public boolean isComplexConjugateOf(
                    ComplexNumber conjugate);
    /**
     * Determines if the input number is equal to the current values.
     *
     * @return true if the input object is a complex number with the real and
     *         imaginary parts that are equal. Otherwise, false.
     */
    public boolean equals(
                    Object equalsComplex);
    /**
     * Determines if the number are equal to a certain degree of accuracy.
     *
     * @param equalsComplex
     *            The complex number to compare.
     * @param decimalPlaces
     *            The number of decimal places to check to in the comparison of
     *            the imaginary and real values.
     * @return true if the values are non-null and they are equal to the number
     *         of decimal places as specified above.
     */
    public boolean equalsRounded(
```

```
                    Object equalsComplex,
                    int decimalPlaces);
}
```

Appendix B

The base class for the Help dialog:

ExampleDialog.java

ExampleDialog.java

```
package edu.uc.ece.blochSphere.exampleDialog;

import java.awt.Color;
import java.awt.Container;
import java.awt.Font;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JEditorPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import edu.uc.ece.blochSphere.Bloch3DCanvas;
import edu.uc.ece.blochSphere.IBlochDialog;
import edu.uc.ece.blochSphere.Qubit3DModel;

/**
 * A base class that was created that shows the Bloch Sphere on the left and has
 * a rich text window on the right. The Examples will have an animation in the
 * Bloch sphere and must implement the {@link #run()} method.
 *
```

```
 * @author Stephen Shary
 *
 */
public abstract class ExampleDialog
                implements ActionListener,
                Runnable, IBlochDialog {

protected JDialog dialog = null;
protected JLabel titleLabel = null;
protected JButton closeButton = null;
protected Bloch3DCanvas blochCanvas = null;
protected JEditorPane exampleEditorPane = null;
protected Qubit3DModel qubitModel = null;

/**
 * This was made private so all other child classes must use the constructor
 * below that defines the parent frame.
 */
private ExampleDialog() {
}

public ExampleDialog(Frame parentFrame) {
        dialog = new JDialog(parentFrame, true);
        dialog.setSize(DIALOG_WIDTH, DIALOG_HEIGHT);
        dialog.setResizable(false);
}

/**
 * Initializes all of the components in the Dialog.
 *
 */
public final void buildDialog() {
        Container contentPane = dialog
                        .getContentPane();
        JPanel basePanel = new JPanel();
        basePanel.setLayout(null);
        basePanel.setSize(DIALOG_WIDTH, DIALOG_HEIGHT);
        // Add title label at the top.
        titleLabel = new JLabel();
        titleLabel.setBounds(PADDING,
                                PADDING,
                                200,
                                50);
        titleLabel.setText("Abstract Example");
        titleLabel.setFont(new Font(
                        Font.SANS_SERIF,
                        Font.ITALIC, 20));
        titleLabel.setText(getDialogTitle());
        // Add close button at bottom right
        closeButton = new JButton();
        closeButton.setText("Close");
```

```
        int buttonWidth = 100;
        int buttonHeight = 40;
        closeButton.setBounds(DIALOG_WIDTH - (PADDING * 2) - buttonWidth,
                              DIALOG_HEIGHT - (PADDING * 4) - buttonHeight,
                              buttonWidth,
                              buttonHeight);
        closeButton.addActionListener(this);
        // Add panel for Demonstration Text.
        exampleEditorPane = new JEditorPane();
        exampleEditorPane.setEditable(false);
        setPage(getHtmlHelpPage());
        JScrollPane scrollPane = new JScrollPane(exampleEditorPane);
        scrollPane.setBackground(Color.gray);
        int exampleWidth = 380;
        int exampleHeight = closeButton.getY() - (PADDING * 2);
        scrollPane.setBounds(    400,
                              closeButton.getY() - PADDING - exampleHeight,
                              exampleWidth,
                              exampleHeight);
        scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
        // Add the BlochSphere
        blochCanvas = new Bloch3DCanvas();
        blochCanvas.buildCanvas();
        blochCanvas.getCanvas().setBounds(PADDING,
                                          (PADDING * 2) + titleLabel.getHeight(),
                                          380,
                                          440);
        // Add the QBit. We don't set it to visible, but rely on the animation
        // to do that when it is ready.
        qubitModel = new Qubit3DModel(
                        45,
                        0,
                        Bloch3DCanvas.RED,
                        blochCanvas.getQubitsGroup());
        qubitModel.setInitalXAngle(0);
        qubitModel.setInitalYAngle(0);
        qubitModel.setInitalZAngle(45);
        basePanel.add(titleLabel);
        basePanel.add(closeButton);
        basePanel.add(scrollPane);
        basePanel.add(blochCanvas.getCanvas());
        contentPane.add(basePanel);
}

/**
 * Displays the dialog and starts the animation. The animation code should
 * be implmented in the public void run() function.
 */
public final void showDialog() {
        // Start the animation. We do this first because the setVisible()
```

```
        // function is blocking.
        new Thread(this).start();
        dialog.setVisible(true);
}

/**
 * This function should be implemented to show the animation of the example.
 * It should continue and stop when m_dialog.isVisible() is false.
 */
public abstract void run();

/**
 * This function will return the title of the help dialog that should be
 * displayed
 *
 * @return The title of the help dialog.
 */
public abstract String getDialogTitle();

/**
 * Defines the html page that should be used in the display of the help
 * dialog. All dialogs should have the html page displayed in the
 * edu/uc/ece/blochSphere/docs folder of the bloch3d.jar
 *
 * @return The name of the HTML file (including the file extension). The
 *         folder is assumed.
 */
public abstract String getHtmlHelpPage();

/**
 * Handles action events from the close button on the dialog that will close
 * the dialog.
 */
public void actionPerformed(
                ActionEvent e) {
        // This will end the animation as well.
        dialog.dispose();
}

/**
 * Sets the html documentation from in the edu/uc/ece/blochSphere/docs in
 * the bloch.jar file.
 *
 * All images that are referenced relatively should be placed in the
 * edu.uc.ece.blochSphere.docs.images package.
 *
 * @param htmlPageName
 *            the name of the html file.
 *
 */
public void setPage(
```

```
                String htmlPageName) {
        try {
                exampleEditorPane.setPage(ExampleDialog.class
                                .getResource("../docs/" + htmlPageName));
        } catch (IOException e) {
                e.printStackTrace();
                return;
        }
}
}
```