# Modeling Qubits with Matlab

Aurelia Brook

Professor Charles Peskin

Special Topics: MATH-UA 395

Midterm Project

November 5, 2021

# Contents

# 1 Introduction

In regards to researching quantum computing, making education accessible can be a hard problem to solve as many would assume that knowledge of quantum mechanics is necessary in order to make meaningful contributions to the field. Companies such as IBM, Google, and Microsoft are trying to combat this by making as much information available online as possible, and creating easily digestible textbooks[6]. One key element that has been found to increase understanding is the inclusion of visual aids and interactive demonstrations of core concepts. For this project, I will be making classical simulations of quantum phenomena, namely by modeling qubits and applying different quantum logic gates so that we may qubit interactions. I will start off with some of the basics: applying Pauli matrices and graphing the change in the vector's trajectory. Using data from my own past experiments with quantum hardware, I will also classically simulate noise, which can be modeled as invertible matrices. In order to correct for the noise, I will use Gaussian elimination with scaled partial pivoting and L-U decomposition iterative techniques for matrix algebra. With Matlab, this will classically model how a single qubit can have a series of logic gates applied onto it in a quantum circuit, with the addition of some noise, then have the noise corrected for in order to create a rudimentary fault-tolerant qubit.

# 2 A Little Background Information

Some of this section can be skipped if the reader feels that they are comfortable with topics addressed in each subheading. Information covered will be very surface level to preserve time, but a little bit of knowledge on all topics listed is necessary to understand the steps taken in the project.

## 2.1 Linear Algebra

In the most elementary sense, a vector[3] $\mathbf{v}$ is a quantity that has the properties of both magnitude and direction. They are described to have a position in space relative to another. A column vector is a vector which has $m$ rows but only 1 column:

$$\begin{bmatrix} 1 \\ 2 \\ ... \\ m \end{bmatrix} \tag{1}$$

A row vector has only one 1 row but $n$ columns:

$$\begin{bmatrix} 1 & 2 & ... & n \end{bmatrix} \tag{2}$$

A matrix can be defined as a set of rows and numbers that form an array. A linear transform[3] is a map $A : \mathbb{R}^n \to \mathbb{R}^m$ that preserves vector addition and scalar multiplication, s.t. for $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^n$, and scalar $\alpha \in \mathbb{R}$:

$$\begin{aligned} A(\mathbf{x}+\mathbf{y}) &= A(\mathbf{x}) + A(\mathbf{y}) \\ A(\alpha\mathbf{x}) &= \alpha A(\mathbf{y}) \end{aligned} \tag{3}$$

As was shown in class, a matrix can be applied to a vector, and operators can be written as matrices. If a vector represents the position and trajectory of an object, one could apply a rotation matrix to change the values of the vector, and thus act as a change in magnitude and direction. This means that for some physical systems, the dynamics of some object can be modeled as a series of matrices applied to a vector that represents the object.

## 2.2 Quantum Computing - The Basics

Quantum computing is a young field in physics, only getting its start in the 1980s with the proposal of Turing machines using quantum mechanics; we are now only just approaching actual proof of quantum supremacy[1], which is the demonstration that a quantum computer can solve problems a classical computer could not within a reasonable amount of time.

When using classical computers, information is stored as bits[6], characterized as either 0s or 1s. However, with qubits (quantum bits), the particles are taken as a coherent superposition of its 0 or 1 states, assuming a continuum of values, which allows for it to store more information. While classical computers compile code in a step-by-step process, the entanglement properties of qubits allow quantum computers to consider multiple possibilities at once. Because of this, quantum computers can have a much lower time complexity for classically intractable problems such as span from protein-folding and code-cracking. This shows the importance of making advancements in quantum computing, as they would allow large projects such as processing data sets for scientific research to execute much faster.

## 2.3 Quantum Computing - The Math

Qubits can be represented as vectors[5]. As aforementioned, a qubit can exist as a superposition of its analogous 0 and 1 states, represented by:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{4}$$

These vectors exist in $L^2$ space[4], which is defined as a function space equipped with a norm for finite-dimensional vectors:

$$||\mathbf{x}||_2 = (|x_1|^2 + |x_2|^2)^{\frac{1}{2}} \tag{5}$$

This function space is called the Hilbert space[5]. Having the ability to normalize a vector is important, as finding the magnitude can be found by taking the norm.

Within quantum mechanics, one of the first things you would learn is the concept of "wave-particle duality"[5], where light can be modeled as both a wave and a particle. For the modeling purposes of this project, qubits will be shown as particles. However, they can also be represented as a wavefunction. The wavefunction $\Psi$ represents the probability amplitude for finding a particle as some position $x$ and at some time $t$. Because probability cannot exceed one (you can't have a 120% chance of something happening), functions must be normalized to one:

$$\int_{-\infty}^{\infty} \Psi^*(x,t)\Psi(x,t)\mathrm{d}x = 1 \tag{6}$$

This ensures that $|\Psi|^2$ can still be interpreted as the probability density.

Since the probabilities are normalized to one, it makes sense that when modeling a qubit as a vector, the magnitude is also normalized to one. A way of visualizing this is on a Bloch sphere[6], otherwise known as a simple unit sphere centered at the origin in Euclidean space

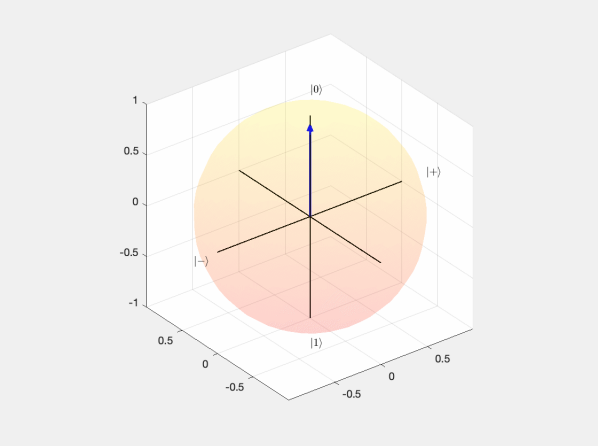


Figure 1: The initialized state, $|0\rangle$, modeled on a Bloch sphere.

This qubit represented as a vector on the Bloch sphere can then be acted on by matrices, known as quantum logic gates[6] (analogous to classical logic gates). These gates can be applied as linear operators.

Examples of such gates include the Pauli matrices:

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
$$\sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{7}$$
$$\sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

## 2.4 Error in Quantum Computers

One of the issues with current Noisy Intermediate Scale Quantum (NISQ) computers are the error levels. There are two general types of error on a quantum chip: coherent and decoherent[6]. Decoherent noise is generally up to probability, ie. the result of some random interference such as cosmic rays. Coherent noise on the other hand is caused by errors within the quantum circuit. Coherent noise can be modeled with unitary matrices, and therefore noise for a state in Quantum information takes the form of unintended unitary transforms (rotations on the Bloch sphere for a single qubit). A unitary matrix $U$ is one that is square and preserve norms, and therefore their probability amplitudes. Unitary matrices also have the property of having an inverse, and its inverse is the conjugate transpose $U^*$ of the matrix.

Common types of noise are amplitude damping error and depolarizing error. Since we can visualize qubits as wave functions, there will be visible occasional drops in amplitude that indicate a qubit resetting to $|0\rangle$, which is the initialized state of each qubit before some other operations or quantum logic gates are applied to them. This is more entropically favorable, as it "costs less" energy to just drop to the $|0\rangle$ state rather than maintain $|1\rangle$ or a superposition of the two. This can be seen as analogous to classical bits reverting to 0 when they should be 1. Amplitude damping error is then more commonly referred to as "bit reset" error. Similarly, depolarizing error is when a state suddenly becomes a completely mixed state:

$$|\Psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{8}$$

Other errors such as bit flip and phase error are a little easier to understand just from their names. While qubits can exist in a continuum of states, measurement of the qubit destroys the state and forces the qubit to then collapse into either the $|0\rangle$ or $|1\rangle$ state. Bit flip error is then named so because the bit flips to the other side, when it shouldn't. As qubit states can be modeled as vectors on a Bloch sphere, it makes sense that these states can be modeled in spherical coordinates. The phase angle $\phi$ can then rotate with respect to the x-axis incorrectly.

Another common source of error is entanglement[5]. Once famously dubbed "spooky action at a distance" by Einstein, entanglement refers to the phenomena where, once separated, two particles that were near each other will still have coupled actions. Meaning: it often appears that the particles still "know" what each other are doing, and their states cannot be separated. This can introduce noise into a physical system of more than one qubit, as one qubit's state will then mimic another's. An example of such is when modeling qubit combinations as a probability distribution:

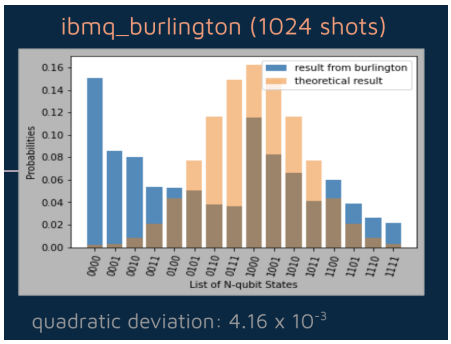


Figure 2:An experimental run on IBM's Burlington quantum chip (Brook, Tsantilas 2021).

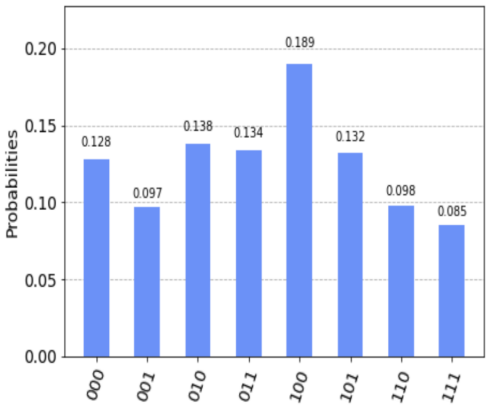


Figure 3:An experimental run on IBM's London quantum chip (Brook, Tsantilas 2021).

Here, the intended purpose was to model a 1D discrete Gaussian distribution on a quantum chip. Because we know what the Gaussian distribution should look like, we can benchmark the errors. While there is a peak in the middle, there are some clear signs of bit reset error since the initialized states have the highest probability of being measured. Additionally, the remaining probability amplitudes are quite close together, meaning there was likely a lot of depolarizing error and qubit coupling. This makes sense, since the gates applied were a series of Controlled-Not (CNOT) gates, which create highly entangled circuits.

Unfortunately, in order to model multiple qubit interactions on a Bloch sphere, one would need to create a $2^{n-1}$dimensional hypersphere. If this is doable, it is certainly above the scope of the project, so only single qubit dynamics will be modeled.

# 3 Applying Concepts: Modeling Qubit Interactions

Because qubits can be modeled as vectors on the Bloch sphere, one could break up the components of the vector into spherical coordinates:

$$|\Psi\rangle = e^{i\gamma}(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle) \tag{9}$$

where $\gamma$ represents global phase, $\phi$ is the angle with respect to the positive z-axis and represents phase shift, and $\theta$ is the angle formed with the positive x-axis. The magnitude of the vector is still always equal to one. When implementing this in Matlab, the Cartesian components are then resolved as:

$$\begin{aligned} x &= \sin\theta\cos\phi \\ y &= \sin\theta\cos\phi \\ z &= \cos\theta \end{aligned} \tag{10}$$

The individual angles can also be solved for some vector with $a$ as the first component, $b$ as the second:

$$\begin{aligned} \gamma &= \arctan\frac{b}{a} \\ \theta &= 2\arccos\frac{a}{\cos\gamma} \\ a &= \cos\gamma\cos\frac{\theta}{2} \\ b &= \sin\gamma\cos\frac{\theta}{2} \end{aligned} \tag{11}$$

These equations will govern how the movement of each qubit is computed. Once a qubit is initialized (at the $|0\rangle$ state), gates can be applied through the standard matrix multiplication method:

$$A\mathbf{x} = \mathbf{b} \tag{12}$$

Where $A$ is some logic gate, $\curvearrowleft$ is your qubit, and $\mathbf{b}$ is your qubit after the transformation. For example, applying the Hadamard gate to a qubit in the $0^{\text{th}}$ state:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} \tag{13}$$

This creates the completely mixed state. One could then apply the $\pi/8$ transformation:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ e^{i\pi/4}\end{bmatrix} \tag{14}$$

Now to make sure the resulting vector is still of magnitude 1:

$$\begin{gathered}\frac{1}{\sqrt{2}}\begin{bmatrix}1\\ e^{i\pi/4}\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}(\frac{\sqrt{2}}{2}(1+i))\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{2}(1+i)\end{bmatrix}\\ \left\|\begin{matrix}\frac{1}{\sqrt{2}}\\ \frac{1}{2}(1+i)\end{matrix}\right\| = \sqrt{|(\frac{1}{\sqrt{2}})|^2 + |(\frac{1}{2}(1+i))|^2} = \sqrt{\frac{1}{2}+\frac{2}{4}} = 1\end{gathered} \tag{15}$$

However, while qubit states can exist as any superposition of states, as mentioned before, measurement will force the wavefunction to collapse onto either $|0\rangle$ or $|1\rangle$, depending on whichever had the highest probability of being measured. This can be modeled as another gate transformation.

Similar to logic gates, coherent noise can be modeled as unitary matrices[5] and applied to qubits in the same matrix * qubit multiplication way as before. For example, bit flip error can be modeled as the application of a NOT gate:

$$\begin{gathered}\sigma_x\mathbf{x} = \mathbf{b}\\ \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\mathbf{x} = \mathbf{b}\end{gathered} \tag{16}$$

Very common types of error are bit reset and depolarizing error. For this, a function is created to reset the input qubit as either $= |0\rangle$ or $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

# 4 Numerically Inverting Noise Matrices

In order to correct for error, after implementing some sort of benchmarking or noise detection algorithm, it is common to model the noise as a unitary matrix or series of unitary transformations. The error matrix can then be inverted and applied to the circuit. When the code is run again, there is a decent probability than when the qubits are measured they are now providing the correct output.

One common way of implementing this is to apply a bit flip at the end before measurement. When dealing with more than one qubit, this process can become complicated due to entanglement. However, since we are only working with one qubit, much of the work instead is in creating the error matrix and then numerically inverting the noise matrix. This can be done using Gaussian elimination with scaled partial pivoting and L-U decomposition. Many other algorithms for inverting matrices as a system of equations do not work here as there are many "0" elements, such that the methods applied to this linear system would diverge.

A pivot element is the first element in a matrix that is selected by an algorithm (such as the numerical method that will be applied here). Finding the pivot element is called pivoting. Scaled partial pivoting is the process of finding which pivot element is largest in each row. Typically, elements of size 0 or some small $\epsilon$ cause issues when pivoting, and this is where Gaussian elimination comes in to approximate elements as either 1 or 0.

The algorithm is as follows, borrowed from Numerical Analysis[2] (Burden, et al. 2016):

1. Check to make sure the matrix is both square and has a non-zero determinant. If the matrix is singular, the algorithm will fail.

2. **Input**: number of unknowns and equations n; augmented matrix A = $[a_{ij}]$ where $i \leq 1 \leq$ n and $1 \leq j \leq$ n+1.

3. **Output**: solution $x_1, ..., x_n$ or message that the linear system has no unique solution.

4. For $i = 1, ..., n$ set $s_i = \max_{1 \leq j \leq n} |a_{ij}|$; if $s_i = 0$, then no unique solution exists. Else: $NROW(i) = i$.

5. For $i = 1, ..., n-1$, let $p$ be the smallest integer with $i \leq p \leq n$ and $|a(NROW(p), i)| = \max_{i \leq j \leq n} |a(NROW(j), i)|$. This begins the forward elimination process.

6. If $a(NROW(p), i) = 0$ then no unique solution exists.

7. If $NROW(i) \neq NROW(p)$ then set $NCOPY = NROW(i)$; $NROW(i) = NROW(p); NROW(p) = NCOPY$. This simulates row interchange.

8. For $j = i+1, ..., n$, set $m(NROW(j), i)/a(NROW(i), i)$. Perform $(E_{NROW(j)} - m(NROW(j), i)E_{NROW(i)}) \rightarrow (E_{NROW(j)})$.

9. $\frac{|a(NROW(p),i)|}{s(NROW(p))} = \max_{i \leq j \leq n} \frac{|a(NROW(j),i)|}{s(NROW(j))}$.

10. If $a(NROW(n), n) = 0$ then no unique solution exists.

11. Set $x_n = a(NROW(n), n+1)/a(NROW(n), n)$. This begins the backwards substitution.

12. For $i = n-1, ..., 1$, set: $x_i = \frac{a(NROW(i),n+1) - \Sigma_{j=i+1}^{n} a(NROW(i),j)x_j}{a(NROW(i),i)}$.

13. Output $(x_1, ..., x_n)$, and the procedure has been completed.

# 5 Results and Discussion

In the code listed in the appendix, .gif files were made for each model of gate actions applied on qubits. For the sake of the paper, images will be shown instead, and the gifs will be accessible in a separate folder.

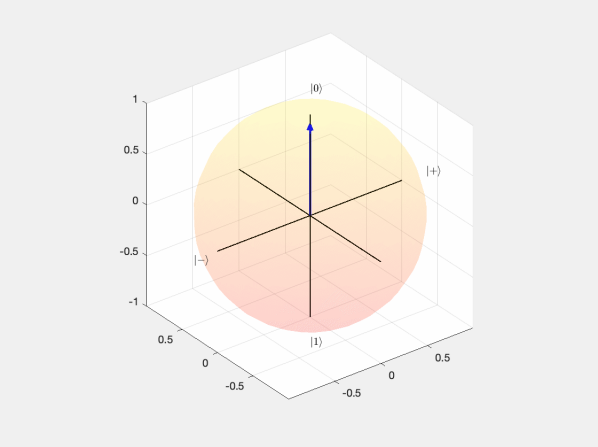


Figure 4: The initialized state, $|0\rangle$.

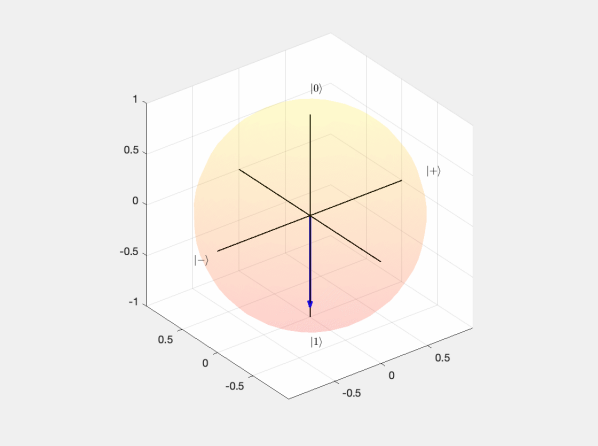


Figure 5: The initialized state after applying the Pauli X gate.

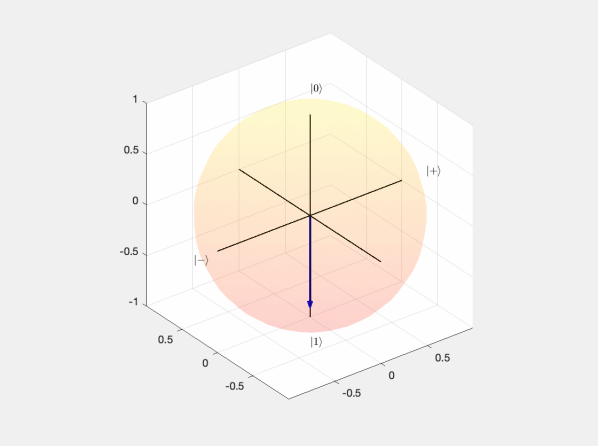


Figure 6: The initialized state after applying the Pauli Y gate.

Notice that these two final values are the same. This is because the resulting matrix is:

$$|\Psi\rangle = \begin{bmatrix} 0 \\ i \end{bmatrix} \tag{17}$$

When taken to be real, this is identical to:

$$|\Psi\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{18}$$

Similarly, applying the Pauli-Z gate to $|0\rangle$ acts as an identity. This is because the Pauli matrices act as rotations by $\pi$ around the respective axes they're named after. Since $|0\rangle$ is already on the z-axis, rotating it by $\pi$ does nothing.

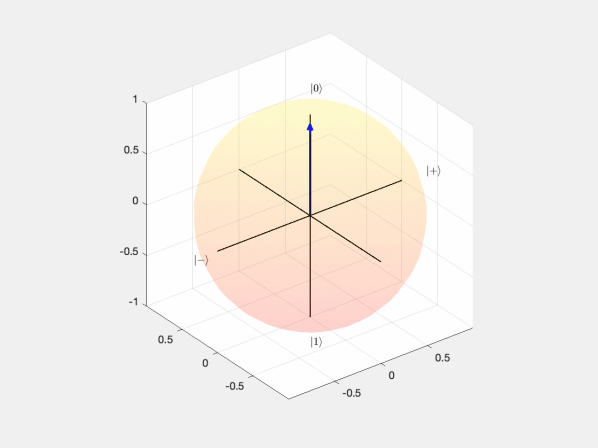


Figure 7: The initialized state after applying the Pauli Z gate.

We can also observe other gates:

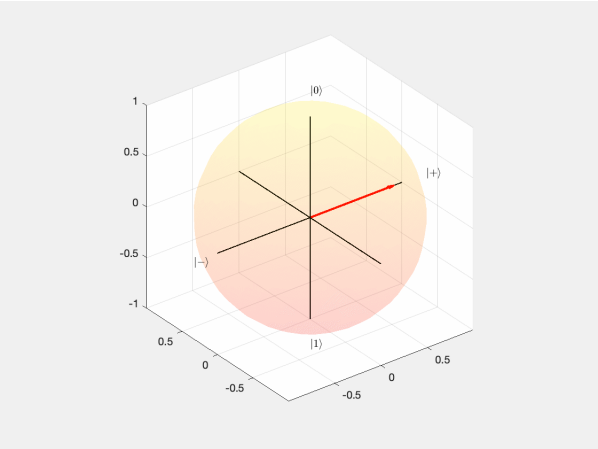


Figure 8: The initialized state after applying the Hadamard gate.

When applying gates in succession, the qubit will continue to exist in a superposition of states. The following is a simulation of a noisy qubit, that may either experience bit flip, bit reset, depolarizing, or phase errors.

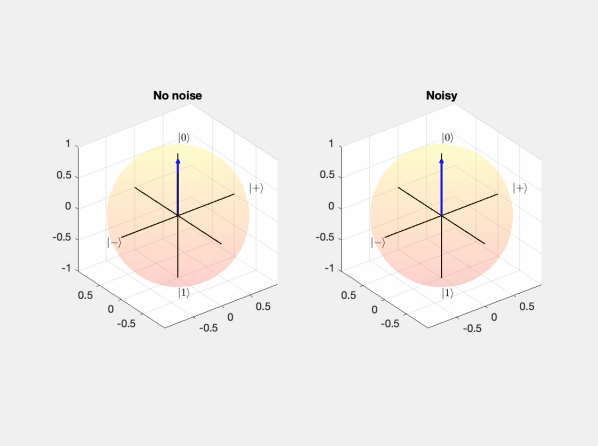


Figure 9: The initialized state, with one noisy qubit and the same qubit but in a noiseless simulation.

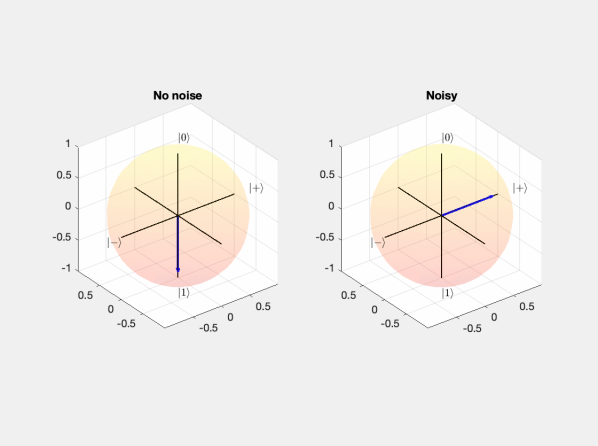


Figure 10: The same qubit, but with one simulating depolarizing error.

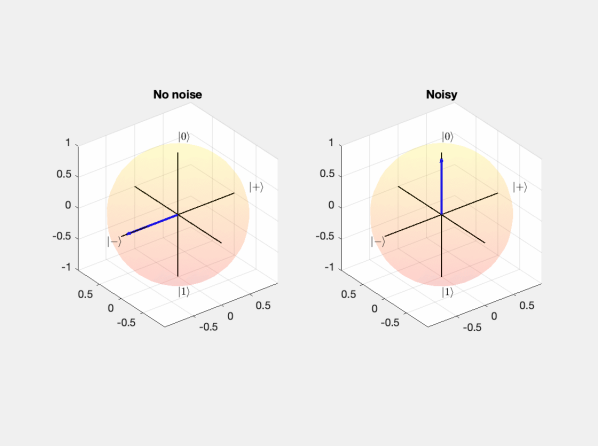


Figure 11: The result of applying the same rotation to qubits after one experiences depolarizing error and the other does not.

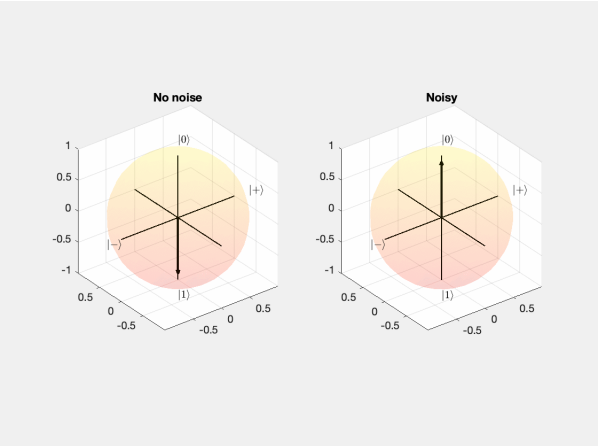


Figure 12: The same qubit, but one with noise and one without noise after measurement.

As can be seen, they have collapsed onto opposite states. The noise is then modeled as a series of applied unitary matrices, and the matrices are numerically inverted. The inverted noise matrix now serves as the error correction matrix, and is added as an additional step. The same code is run, and different errors may be encountered, but as a final step: the identity matrix is applied to the noiseless qubit, and the error correction matrix is applied to the noisy qubit. Then, both versions of the qubit are measured, such that they will collapse onto either the $|0\rangle$ or $|1\rangle$ state.

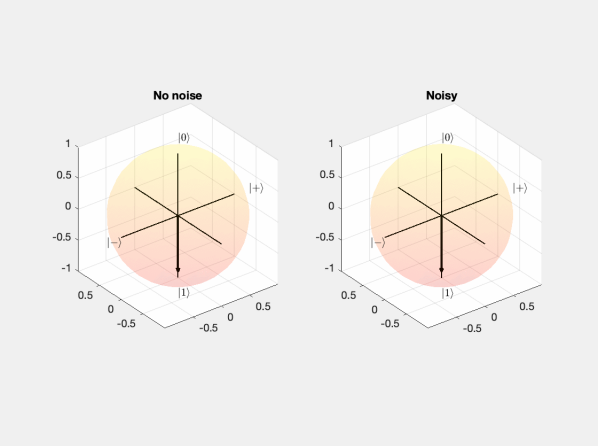


Figure 13: The same qubit after measurement, but one with noise and one without noise. This is after an error correction matrix was applied to the noisy qubit and then measured.

The error correction matrix worked during this run!

# 6 Conclusion

The application of gates to a single qubit has been modeled, such that it will be easier to view how linear operators can govern qubit dynamics. Having a visual aid in processes that are too small to witness in real life often makes understanding easier.

This method of error correcting qubits is not always guaranteed to work, as qubit states are probabilistic and so are their errors. However, it is a popular method[1] and still has a comparatively chance of working. Performing tests such as these where one knows what the output should look like is what allows for the creation of good error correction algorithms. These can then be applied to runs on real hardware where the answer is unknown, so the numerical method of finding and inverting a error model is one that can carry on to any number of noisy qubits.

# 7 Bibliography


1. AI, Google Quantum. "Exponential suppression of bit or phase errors with cyclic error correction." Nature 595.7867 (2021): 383.

2. Burden, Richard L., et al. Numerical Analysis. Cengage Learning, 2016.

3. Judson, Thomas W., and Robert A. Beezer. Abstract Algebra: Theory and Applications. Orthogonal Publishing, 2015.

4. Sakurai, Jun J., and San Fu Tuan. Modern Quantum Mechanics. Addison-Wesley Longman, 1993.

5. Shankar, Ramamurti. Principles of Quantum Mechanics. Springer Science+Business Media, LLC, 2013.

6. The Qiskit Team. "Learn Quantum Computation Using Qiskit." Learn Quantum Computation Using Qiskit, Data 100 at UC Berkeley, 18 Oct. 2021, https://qiskit.org/textbook/preface.html.

# 8 Appendix

11/5/21 7:14 PM /Users/aureliabrook/Docum.../blochSphere.m 1 of 4

```
% Define the Pauli matrices
X = [0 1; 1 0]; % Quantum equivalent of a NOT gate
Y = [0 -1i; 1i 0];
Z = [1 0; 0 -1];

% Other quantum logic gates
I = [1 0; 0 1]; % Identity matrix
H = (1/sqrt(2) * (X + Z)); % Hadamard Gate
CNOT = [1 0 0 0; 0 1 0 0; 0 0 0 1; 0 0 1 0]; % Controlled NOT gate

C = [0 1; 1 0]; %Correction gate

y = makeGIF({{X, H, I}, {X, H, C}}, {[1, 0], [1, 0]}, 20, ["No noise", "Noisy"], "blah",↙
[false, true]);

function P = P(p)
    P = [1 0; 0 cos(p)+1i*sin(p)];
end % Phase shift with regards to some angle phi

function [euc, ang] = QubitToEuclidean(q)
    a = real(q(1));
    b = imag(q(1));
    if a == 0
        gamma = 0;
    else
        gamma = atan(b / a);
    end
    theta = 2*acos(a / cos(gamma));
    if theta == 0
        phi = 0;
    else
        phi = acos(real(q(2)) / sin(theta/2)) - gamma;
    end

    % Convert Spherical to Euclidean
    euc = [sin(theta)*cos(phi), sin(theta)*sin(phi), cos(theta)];
    ang = [theta, phi];
end

function q1 = noise(q0)
    n = randi(6);
    if n < 4
        q1 = [1, 0];
        disp("ket0");
    elseif n < 6
        q1 = [1/sqrt(2), 1/sqrt(2)];
        disp("ket+");
     else
         q1 = q0;
         disp("nothing");
    end
end

function b = measure(euc)
    b = [0, 0];
    if euc(3) > 0
        b(3) = 1;
    else
```

```
        b(3) = -1;
    end
end

function  q1 = makeGIF(gates, q0, n, ti, filename, err)
    q1 = {};
    vecs = {{}};
    d = length(q0);
    for i = 1:d
        for k = 1:length(gates{i})
            [euc0, ~] = QubitToEuclidean(q0{i});
            q1{i} = gates{i}{k} * q0{i}';
            [euc1, ~] = QubitToEuclidean(q1{i});
            if k == 1
                vecs{i} = {euc0};
            else
                vecs{i}{end+1} = euc0;
            end
            vec_diff = (euc1 - euc0) / n;
            if euc0*euc1' == -1
                for j = 1:n-1
                    vecs{i}{end+1} = (euc0+j*vec_diff);
                end
            else
                for j = 1:n-1
                    vecs{i}{end+1} = (euc0+j*vec_diff)/norm(euc0+j*vec_diff);
                end
            end
            vecs{i}{end+1}=euc1;
            q0{i} = q1{i}';
            if err(i)
                q1{i} = noise(q0{i})';
                euc0 = euc1;
                [euc1, ~] = QubitToEuclidean(q1{i});
                vecs{i}{end+1} = euc0;
                vec_diff = (euc1 - euc0) / n;
                if euc0*euc1' == -1
                    for j = 1:n-1
                        vecs{i}{end+1} = (euc0+j*vec_diff);
                    end
                else
                    for j = 1:n-1
                        vecs{i}{end+1} = (euc0+j*vec_diff)/norm(euc0+j*vec_diff);
                    end
                end
                vecs{i}{end+1}=euc1;
                q0{i} = q1{i}';
            elseif ~err(i) && ~all(~err)
                for j = 1:n+1
                    vecs{i}{end+1} = euc1;
                end
            end
        end
    end
    for i = 1:d
        vecs{i}{end+1} = vecs{i}{end};
        euc0 = vecs{i}{end};
        disp(euc0);
```

```
        euc1 = measure(euc0);
        vec_diff = (euc1 - euc0) / n;
        if euc0*euc1' == -1
            for j = 1:n-1
                vecs{i}{end+1} = (euc0+j*vec_diff);
            end
        else
            for j = 1:n-1
                vecs{i}{end+1} = (euc0+j*vec_diff)/norm(euc0+j*vec_diff);
            end
        end
        vecs{i}{end+1}=euc1;
    end
    [x, y, z] = sphere(25);
    h = figure('visible', 'off');
    for i = 1:length(vecs{1})
        % Create the Bloch sphere, and customize how it looks
        for j = 1:d
            subplot(1, d, j);
            bloch = surf(x, y, z);
            title(ti(j));
            axis equal manual
            colormap autumn
            shading interp
            bloch.FaceAlpha = 0.1;
            if ~err(j) && rem(i, 2*n+2) > n+1 && ~all(~err)
                color = "Green";
            elseif 1 < i && i < length(vecs{1}) && rem(i, n+1) ~= 0
                color = "Red";
            else
                color = "Blue";
            end
            if (i > 2 * d * (n + 1) && ~all(~err)) || (i > d * (n + 1) && all(~err))
                color = "Black";
            end

            % Create axes and labels
            line([-1, 1], [0, 0], [0, 0], 'LineWidth', 1, 'Color', [0 0 0]);
            line([0, 0], [-1, 1], [0, 0], 'LineWidth', 1, 'Color', [0 0 0]);
            line([0, 0], [0, 0], [-1, 1], 'LineWidth', 1, 'Color', [0 0 0]);
            text(0, 0, 1.25, '$\left| 0 \right>$', 'Interpreter', 'latex');
            text(0, 0, -1.25, '$\left| 1 \right>$', 'Interpreter', 'latex');
            text(1.25, 0, 0, '$\left| + \right>$', 'Interpreter', 'latex');
            text(-1.25, 0, 0, '$\left| - \right>$', 'Interpreter', 'latex');
            hold on
            quiver3(0, 0, 0, vecs{j}{i}(1), vecs{j}{i}(2), vecs{j}{i}(3), 'LineWidth', 2,↙
'Color', color);
            hold off
        end
        drawnow;

        % Capture the plot as an image
        frame = getframe(h);
        im = frame2im(frame);
        [imind,cm] = rgb2ind(im,256);

        % Write to the GIF File
        if i == 1
```

```
            imwrite(imind,cm,strcat(filename, ".gif"),'gif', 'DelayTime',0.025,↙
'Loopcount',inf);
        else
            imwrite(imind,cm,strcat(filename, ".gif"),'gif','DelayTime',0.025,↙
'WriteMode','append');
        end

        if i == 1 || i == length(vecs{1}) || rem(i, n+1) == 0
            for k = 1:9
                imwrite(imind,cm,strcat(filename, ".gif"),'gif','DelayTime',0.025,↙
'WriteMode','append');
            end
        end
    end
end
```

```
function y = forward_sub (L, b)
  [m,n] = size(L);
  y = zeros(n,1);        % initialize x to be a column vector

  % The first variable calculated directly
  y(1) = b(1)/L(1,1);

  % Back-substitution for remaining variables.
  % ---This could also be accomplished without loop by using .*
  for i = 1:n
    y(i) = b(i); % Remember forward-sub: y_i = (b_i - sum of A_ij*x_j )/A_ii
    for j = 1:i-1
      y(i) = y(i)-L(i,j)*y(j);
    end
    y(i) = y(i)/L(i,i);
  end

end
```

```
% Perform gaussian elimination.
% In place of changing elements of A to zero. Use them to save entries of L
% Returns the permutation vector and matrix A after transformation.

function [A, p] = gauss_eli_srpp (A)
  [n,m] = size(A);
  if n~=m
    error('This function requires a square matrix as an input!')
    return;
  end
  p = (1:n)';    % the permutation vector
  s = max(abs(A'));    % scale for each row

  for k = 1:(n-1) % k-th step of Gaussian Elimination
    d_s = abs(A(p(k),k))/s(p(k));
    pivot_index = k;

    % Find next Pivot row
    for i = k+1:n
      z = abs(A(p(i),k))/s(p(i));
      if z > d_s,
        d_s = z;
        pivot_index = i;
      end
    end

    % Change the permutation vector indicating row-swaps
    curr_index = p(k);
    p(k)= p(pivot_index);
    p(pivot_index) = curr_index;

    % Gaussian Elimination for step-k.
    for i = (k+1):n
      m = A(p(i),k)/A(p(k),k); % The multiplier for row k+1, ...

      %change elements of every column in row i
      for j = (k+1):n
        A(p(i),j) = A(p(i),j)- m*A(p(k),j);
      end

      A(p(i),k) = m; % We can save the multiplier right in A in place of L
    end

  end
end
```

```
function x = backward_sub (U, b)
  [m,n] = size(U);
  x = zeros(n,1);        % initialize x to be a column vector

  % The last variable calculated directly
  x(n) = b(n)/U(n,n);

  % Back-substitution for remaining variables.
  % ---This could alsobe accomplished without loop by using .*
  for i = (n-1):-1:1
    x(i) = b(i); % Remember back-sub: x_i = (b_i - sum of A_ij*x_j )/A_ii
    for j = (i+1):n
      x(i) = x(i)-U(i,j)*x(j);
    end
    x(i) = x(i)/U(i,i);
  end
end
```

```
function x = MatrixSystem(M,b)
    [A, p] = gauss_eli_srpp(M);
    [n,m] = size(A);
    if n~=m
        error('This function requires a square matrix as an input!')
        return;
    end
    P = zeros(n);
    for i=1:n
        P(i, p(i)) = 1;
    end
    A = P * A;
    U = triu(A);
    L = (tril(A)+diag(ones([n, 1])-diag(tril(A))));
    y = forward_sub(L, P*b);
    x = backward_sub(U, y);
end
```

```
function I = MatrixInverse(M)
    [n, m] = size(M);
    I = zeros(n);
    for i = 1:n
        curr = zeros([n, 1]);
        curr(i) = 1;
        b=MatrixSystem(M, curr);
        for j=1:n
            I(j, i) = b(j);
        end
    end
    disp(M*I);
end
```