3-2021

Error Detection in Quantum Algorithms

Simeon R. Hanks

QUANTUM COMPUTING USING ERROR DETECTION

THESIS

Simeon R. Hanks, Capt, USAF

AFIT-ENP-MS-21-M-120

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENP-MS-21-M-120

QUANTUM COMPUTING USING ERROR DETECTION

THESIS

Presented to the Faculty

Department of Engineering Physics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Applied Physics

Simeon R. Hanks, B.S.

Capt, USAF

March 2021

AFIT-ENP-MS-21-M-120

QUANTUM COMPUTING USING ERROR DETECTION

THESIS

Simeon R. Hanks, B.S.
Capt, USAF

Committee Membership:

Dr. D. Weeks, Ph.D.
Chair

Dr. L. Merkle, Ph.D.
Member

Maj D. Emmons, Ph.D.
Member

AFIT-ENP-MS-21-M-120

Abstract

Quantum computers need to be able to control highly entangled quantum states in the presence of environmental perturbations that lead to errors in calculations. Progress in superconducting qubits has enabled the development of computers capable of running small quantum circuits. The current era of Noise Intermediate Scale Quantum computing has a high error rate. To alleviate this error rate we apply an encoding scheme that allows us to remove results with known errors improving the quality of our results. The encoding uses multiple qubits as a single logical qubit and balances the natural tendency of state-of-the-art quantum computers to decohere towards the ground state. We use a mix of ones and zeroes in each logical qubit in such a way that we can identify and remove results that have violated our specified encoding pattern. The statistical performance of the circuits is improved by retaining the shots that maintained the encoding. Bit flip error detection is applied to the Toffoli gate and produces improved probability distribution functions as well as enhanced similarity measures when compared to its unencoded equivalent.

AFIT-ENP-MS-21-M-120

This thesis is dedicated to my wife for supporting me through school, pushing me to always move forward, and being my best friend and world-traveling partner.

Acknowledgements

I would like to acknowledge Dr. Weeks, my committee, and the faculty of the Engineering Physics department for their valuable inputs, guidance, and patience in the production of this thesis.

Simeon R. Hanks

Table of Contents

QUANTUM COMPUTING USING ERROR DETECTION

I. Introduction

1.1 Quantum Computing in the Classical Era

Classical computers are what we refer to as “computer” in day to day life. From your digital watch to the server you used to retrieve this document, we say it was done on a computer. Colloquially we say, “It’s all a bunch of ones and zeroes.” Grossly oversimplifying how a computer works, but accurately describing the language in use in all of our computers today. The computers we use today are digital and use solid state devices to achieve their switching function. These digital computers are the ancestors of analog computers. Analog computers used electricity in vacuum tubes to conduct “on/off” operations to compute calculations on room-sized machines [1]. These analog computers were in turn the ancestors of mechanical computers that used the transfer of forces through instruments to physically perform computations. If we follow this lineage far enough back we arrive at the oldest known computer, the abacus, which had a series of beads that were moved in such a way as to aid humans with arithmetic operations.

Quantum computers are now arriving and are often mistaken as the natural successor in this long line of computers. This is a flawed assumption as a quantum computer does not necessarily perform better than a classical computer in all operations [2]. While any computational operation on a classical computer can be conducted on a quantum computer, and vice versa [3], it does not mean that these systems perform those operations with the same efficiency.

A classical computer offers a deterministic solution. Meaning that we can rely on the fact that if we input one state and apply the same operation, the output will always be the same. This deterministic capability is exactly what gives computers their utility. If we wish to add two numbers, we understand that the result will be their sum. In a quantum computer, if we add two states, we will arrive at a linear superposition of their possible outcomes [2] [4]. Meaning that the result is not always state A or state B, but if we run the experiment multiple times we can get a distribution of state A or state B. This non-deterministic behavior is our first step into a quantum world. We can no longer rely on the idea that, "if A acts on B then C results," which is both intimidating and exploitable [5].

The obvious problem in this new way of thinking is that if we don't know what the answer is going to be, how do we trust the measured output? This is where we need to create tools that will help reign in quantum computers and give us confidence the the resulting solutions [6]. Namely we can use error detection, which allows us to throw away results that are known to be noise, and error correction, which enables us to correct mistakes in the calculations as they run.

1.2 Error Correction

It is of critical importance that we understand when a mistake in a calculation has taken place. This could be anything from an erroneous input to an operator not adding two inputs correctly.

In classical computers error correction at the hardware level is rarely used outside of specific sensitive hardware like those found in medical, banking, or military applications where no error can be tolerated[7]. When a hardware error occurs, the system will shutdown to protect any potential damage and give the user a message similar to a blue screen of death on a Windows operating system. Where this kind of failure is forbidden or catastrophic, redundant systems are used to check one another until the next operation takes place, greatly reducing the probability of two errors occurring simultaneously.

On a quantum computer this style of error correction is impossible due to the non-deterministic nature of the state vectors [8]. Even if two states are prepared identically and ran through the same operators, we can not verify that one is evolving correctly based on the measurement of the other. The wave forms are not entangled and any measurement of one would not give us information about the other.

What we can do is create a system that knows when an error has occurred using control gates that provide syndromes. If the syndrome on the output is found to be one way, we can implement corrective actions to undo that error [9]. We will discuss such an algorithm in the main body of work and show how we are further able to implement encoding to this corrective action to increase our confidence in a syndromes result.

1.3 Error Detection

Unlike error correction where the fault is repaired, error detection is used to identify when an error has occurred and use that information to modify or remove various data sets.

In Section 4.2 we will show how we are able to leverage two physical qubits to create an error detection scheme that greatly increases confidence in our results. We do this by running quantum computing algorithms through a physical computer and comparing those results to known solutions as a method of validating our techniques.

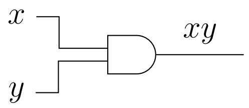
II. Logic, Bits, and Operators

2.1 Classical Logic, Bits, and Operators

Computers must follow certain rules in order for them to be useful. If a given input is entered, an expected output should result. This is done through the mechanisms inside a given machine. Whether it be two levers that force a third lever up representing a change in a variable or two nodes on a silicon chip that have their voltages changed causing a third node to modify its value, we expect a machine to follow known rules. How exactly the input translates to a given output is dependent on the logical operators inside a system, the values put into those operators, and the resulting output.

The smallest piece of computational information is called a bit. A bit is a portmanteau of the words binary and digit [10], which is exactly how a bit functions. A bit can hold either the value one or zero and represents information in a system. How a bit is physically measured or stored is left up to the human imagination and the capability of engineers. Modern computers have stored them on everything from paper punch cards to a single photon in an optical fiber [11].

Classical computer circuits are built on Boolean logic which depends upon having binary information. This meaning that they use AND, OR, NOT, NAND, NOT, XOR, and XNOR gates as operators to conduct computational operations on one and zero inputs and return one and zero outputs. We will only be discussing the AND and OR gates here as examples of classical computation operators.



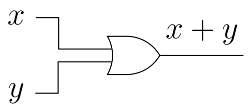
(a) AND gate with inputs x and y and outputs xy.

x	y	xy
1	1	1
1	0	0
0	1	0
0	0	0

(b) Truth table for an AND gate.

Figure 1. An AND gate with two inputs x and y and the resulting truth table for their product xy.

In the AND gate shown in Fig. 1 you can apply basic Boolean logic that would read if x and y are both 1, then xy is also equal to 1. The fact that all actions and responses are represented by the "on" or "off" state is the basis and definition of Boolean logic.



(a) OR gate with inputs x and y and outputs $x + y$.

x	y	$x + y$
1	1	1
1	0	1
0	1	1
0	0	0

(b) Truth table for an OR gate.

Figure 2. An OR gate with two inputs x and y and the resulting truth table for their sum $x + y$.

In the case of the OR gate shown in Fig. 2 we would read it as, if x or y are equal to 1, then the output will also be 1. This includes the redundant case where both values are equal to 1.

Although both these gates represent their operations in terms similar to those used in elementary algebra it is worth noting that this language is misleading. With the AND gate you appear to get the product of x and y, but this terminology is more a matter of convention than an actual multiplication of the two binary inputs. For instance, look at the OR gate where we "sum" the inputs of 1 and 1, we do not get

2, as 2 does not have a binary analog.

Using these two gates and their inverses Not AND (NAND) and Not OR (NOR) you can begin to build logical circuits that can start to implement practical computing and perform operations that we would call multiplication and summation in the traditional sense of the word.

The field surrounding how to go from bits and simple operators to the modern computer is an entire area of research in itself and well outside the scope of this thesis. It is useful to be familiar with these concepts as we get in to the strange world of quantum computers where our famous bit finds itself in both the state 1 and 0 simultaneously.

2.2 Quantum Logic, Bits, and Operators

As with classical computers, quantum computers work with information. We call this unit of information a qubit, which is a portmanteau of quantum and bit (which we already know as a binary digit). So what is a quantum binary digit? Most simply it is a two-level system that can be in a ground state 0, an excited state 1, or a linear combination of the two with some probability of existing in either state upon measurement. When we measure a state vector we will get a binary result. But if we measure that same calculation again we may get a different binary result. This requires us to take multiple measurements to find a statistical probability of a given state vector occurring and telling us the answer to our calculation.

Instead of logical gates quantum computers, at the least the IBM quantum computers used for this research, rely on five unitary operators:

$$U_3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{pmatrix}, \tag{1}$$

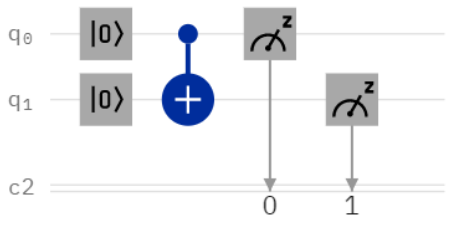
$$U_2(\phi,\lambda) = U_3(\pi/2,\phi,\lambda) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i\lambda+i\phi} \end{pmatrix}, \tag{2}$$

$$U_1(\lambda) = U_3(0,0,\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}, \tag{3}$$

$$I = U_3(0,0,0) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{4}$$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5}$$

From these operators we are able to create gates that act on an input state and place them into a linear superposition that will have a probabilistic output. The gates we will need to build our algorithms include the CNOT, Hadmard, Toffoli, NOT, T, and S.



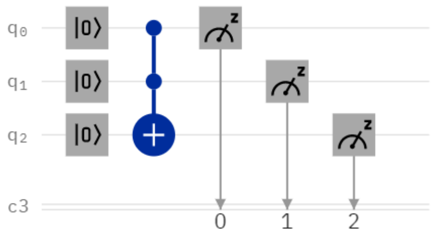
(a) CNOT gate with $q_0 = |0\rangle$ and $q_1 = |0\rangle$.

$\|q_0\rangle$	$\|q_1\rangle$	$\|z_0\rangle$	$\|z_1\rangle$
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

(b) Truth table for a CNOT gate.

Figure 3. CNOT with inputs $q_0 = |0\rangle$ and $q_1 = |0\rangle$ and the resulting truth table for the outputs associated.

A CNOT gate has a target, seen here in Fig. 3 as a circle with a plus sign on q_1 and a control shown as a single dot on q_0 both connected by a line to show a controlled gate. If the control is in an excited state $|1\rangle$ then the target will change state, otherwise both states continue unchanged.



(a) Toffoli gate with $q_0 = |0\rangle$, $q_1 = |0\rangle$, and $q_2 = |0\rangle$.

$\|q_0\rangle$	$\|q_1\rangle$	$\|q_2\rangle$	$\|z_0\rangle$	$\|z_1\rangle$	$\|z_2\rangle$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

(b) Truth table for a Toffoli gate.

Figure 4. Toffoli with inputs $q_0 = |0\rangle$, $q_1 = |0\rangle$, and $q_2 = |0\rangle$ with the resulting truth table for the outputs associated.

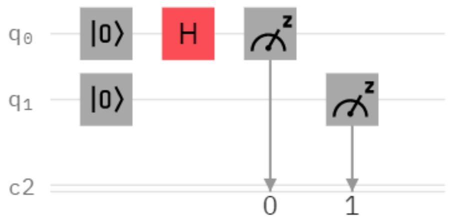
Like the CNOT gate, the Toffoli gate also has a target as seen in Fig. 4, but it has two controls.The control gate of a Toffoli must have two excited states in order to cause the target state to flip from either its excited state to ground state or ground state to the excited state.

These gates are the only two we will use for which a truth table is enlightening.

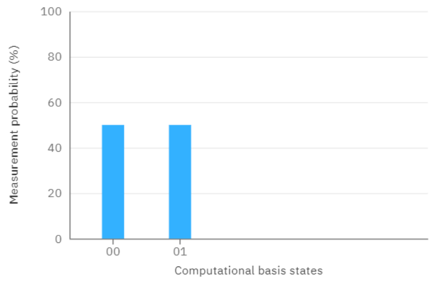
Up to this point the resulting state is labeled with 1's and 0's. But let us now take a look at the Hadamard gate, where the fun begins. Using the universal operators above we can get the matrix representation of the Hadamard gate as follows:

$$\text{Hadamard: } H = U_3(\pi/2, 0, \pi) = U_2(0, \pi) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{6}$$

This gate acting on a two-level qubit will place it into a linear superposition such that it's probability of being 0 or 1 is equal when measured [12].



(a) Hadamard gate with $q_0 = |0\rangle$ and $q_1 = |0\rangle$.



(b) Measurement probability for a Hadamard gate.

Figure 5. Hadamard with inputs $q_0 = |0\rangle$ and $q_1 = |0\rangle$ with the resulting measurement probability histogram.

When a state vector is prepared with two qubits in the ground state as seen in Fig. 5 the resulting measurement should be an equal probability of $q_0 = |0\rangle$, $q_1 = |0\rangle$, and $q_0 = |0\rangle$, $q_1 = |1\rangle$! For ease of notation we will label these state vectors as $|\psi\rangle$, because that's what they are, a single function. We will then say that $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$.

Now we have two controlled gates and a quantum operator and can begin to build our circuits. We will describe other gates as they become necessary.

III. Modern Quantum Computers

3.1 Coding On IBM Q Computers

Before we get into the circuits let's look at the tools we are using to access our physical quantum computers. We are using IBMs quantum computers due to their ease of use and state-of-the-art technology. They make many of their computers publicly accessible and through AFRL, AFIT has access to their larger systems.

IBM uses transmon qubits for their systems which are charge insensitive cooper pair boxes that behave as anharmonic oscillators that allow for pulsed microwaves to either prepare a qubit in its ground state, excite it, or act on a qubit with an arbitrary quantum operator [13] [14]. Operators and gates are achieved via waveforms that are empirically created by the engineers at IBM that provide the appropriate action of a quantum mechanical operation. These pulses must be calibrated daily as the machine falls in and out of its optimal functioning condition. This information is useful to us because the results of different runs of the same circuits may have better or worse results. One way of overcoming this uncertainty in calibration is to conduct multiple runs and collate those results into bins so that we can see how far they deviate from their theoretically expected values.

3.2 Verifying Results

In order to make use of a computation, whether classical or quantum, you must be able to both input data, operate on it, and measure the result. Classically this is achieved by measuring the voltages on the gates and using those measurements to provide output to a user. On IBM's quantum computers we achieve this by measuring the state vector at the end of a run on each qubit and analyzing the distribution of states on a given circuit. If $|\psi_f\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ we would expect to see 50% of the results in each state, but in reality we see about about the same quantity in each state with some non-zero measurements in the states $|01\rangle$ and $|10\rangle$. We use a probability distribution function to view the number of measurements in a given run and their resulting state vectors. To see if these are good runs we then compare each separate run to its expected theoretical value and use this to find a similarity measure.

3.2.1 Probability Distribution Function

The most common way the data for quantum computers is measured is with a probability distribution function (PDF). What we see on probability distribution functions is a count of how many times each vector state was measured after a set number of runs. Due to the nature of quantum mechanics and the linear superposition of multiple states we get both valid and invalid states. A state is valid when it is within the range of quantum mechanically allowed states and invalid if it is a result of noise. Unfortunately, in the current era of Noise Intermediate Scale Quantum computing we can not immediately distinguish between quantum mechanical fluctuations and the results due to noise in our system.

3.2.2 Similarity Measure

One question we have to ask is "Is this a good result?" What does a good result look like? To do this we use an analytic solution as a baseline and compare it to a measured PDF. If we expect a state vector $|\psi\rangle = |00\rangle$ and measure $|00\rangle$ 100% of the time we know that the computer is working "perfectly." This is only possible in a simulator as we know that an actual quantum computer is noisy and subject to error.

To investigate the similarity of our results to our expected results let us use the following equation:

$$\mu = 1 - \sum_{n=0}^{2^k-1} \frac{|P_n - P_n^e|}{2} \tag{7}$$

Where P_n is the measured probability and P_n^e is the expected probability of a given state. We would expect μ to be equal to 1 in a perfect simulation and closer to 1 in a machine with less noise. For k qubits we will have a distribution of states equal to 2^k and will have to measure their arithmetic average to find their similarity to the exact result.

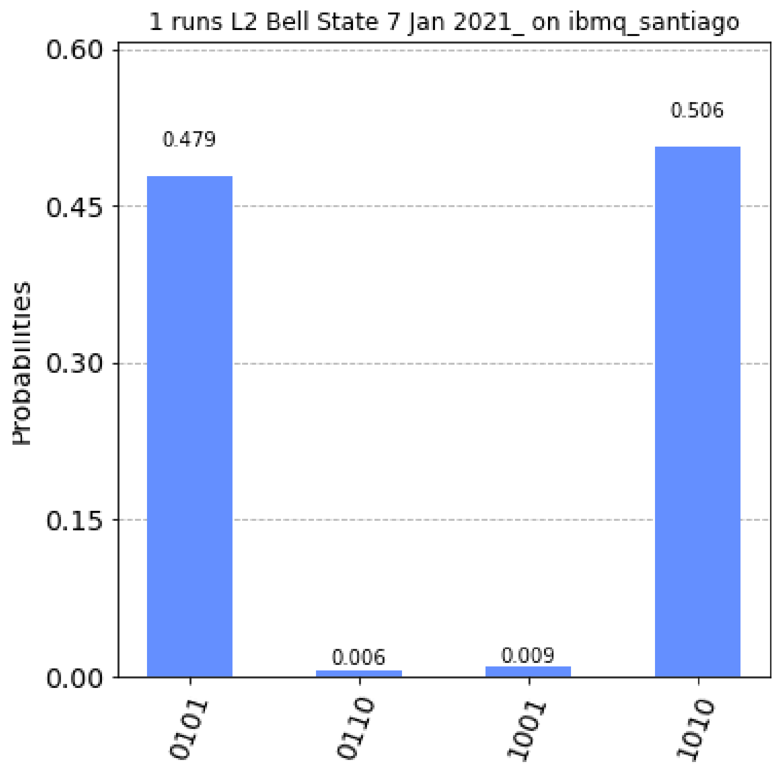


Figure 6. This probability distribution is the result of 6469 shots of an encoded Bell State recorded on the IBM Santiago quantum computer.

If we look at the probability distribution in Fig. 6 we expect a 50% chance of $|1010\rangle$ and 50% chance of $|0101\rangle$ for these results. We in fact measured some states in $|1001\rangle$ and $|0110\rangle$ that will lead to our similarity measure moving away from the theoretical value of $\mu = 1$. To find μ for this measured distribution we use Eq. 7 as follows:

$$\begin{aligned}\mu &= 1 - \frac{(|P_{0101} - P^e_{0101}| + |P_{0110} - P^e_{0110}| + |P_{1001} - P^e_{1001}| + |P_{1010} - P^e_{1010}|)}{2} \\ &= 1 - \frac{(|0.479 - 0.50| + |0.006 - 0| + |0.009 - 0| + |0.506 - 0.50|)}{2} \\ &= 0.979\end{aligned} \tag{8}$$

Which tells us that the measured results are 97.9% similar to our exact solution. We will use this method throughout the paper to compare the measured results to our expected value as a method of testing the validity of our encoding schemes against their unencoded counterparts. While knowing that a single point is 97.9% similar is useful, we are still using a relatively noisy quantum computer, and we will want to make multiple runs of each circuit and bin these results in a histogram. This will show the distribution of similarity on similar machines over many runs. This spread of similarity measures will give us a more general idea of the accuracy of our results as well as provide insightful information as to the the spread of noise between large sets of shots. Using the similarity measure in this manner for both the encoded and unecoded circuits gives us an idea of how close each is to the analytic solution but does not necessarily tell us objectively quantifiable value to our results.

IV. Analysis and Development of Quantum Circuits

4.1 State Analysis and Error Detection

To follow the logic of a quantum circuit you must first create the initial states and then look at the operators that will act on each state as it progresses through a given circuit. To better illustrate this let us look at a simple quantum circuit, the Bell State.

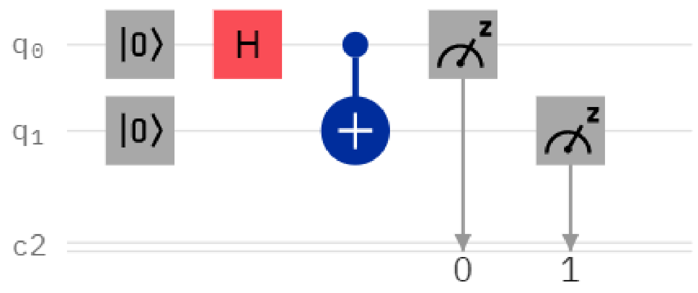


Figure 7. The Bell State. Which uses one Hadamard gate to create a linear super-position of of the $|00\rangle$ and $|11\rangle$ states each with a $\frac{1}{\sqrt{2}}$ probability .

The Hadamard gate will give us the states

$$|\psi\rangle = \frac{(|00\rangle + |10\rangle)}{\sqrt{2}} \tag{9}$$

Which is the super-position of the $|00\rangle$ input and the $|10\rangle$ excited state, each in equal probability of occurring. The Hadamard creates the two possibilities of either, $q_0 = |0\rangle$ and the CNOT control not flipping the CNOT target or $q_0 = |1\rangle$ acting on the control bit of the CNOT gate causing $q_1 = |1\rangle$ to yield $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Classically, this will result in two states each with a Probability Distribution Function giving them a 50% chance of occurring. This is where the quantum mechanics comes into play. If this circuit is ran through a simulator you get the states as expected one half in the $|00\rangle$ state and the other half in the $|11\rangle$ state. The Bell State defies these results when ran on a physical quantum computer because we get the states $|01\rangle$ and $|10\rangle$ at some non-zero probability.

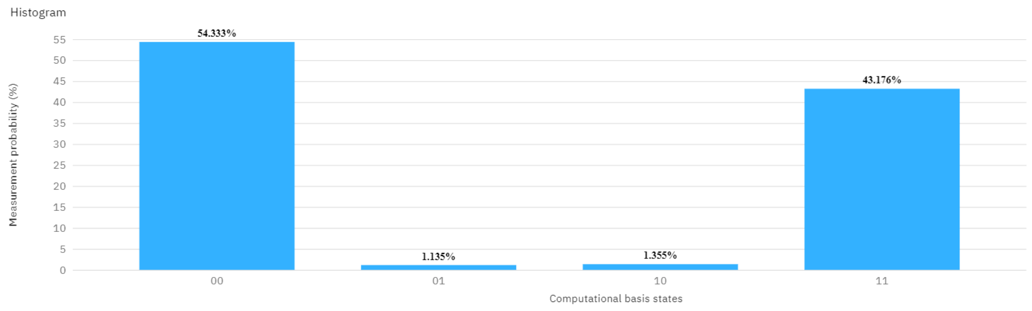


Figure 8. This probability distribution is the result of 8192 shots of the Bell State recorded on the IBM Santiago quantum computer. We in fact measure the states $|01\rangle$ and $|10\rangle$, clasically impossible, as well as the state $|00\rangle$ having a higher probability of occurring.

To analyze this circuit we must look at the wave function as it progresses through the circuit. Let us do this by drawing a line immediately after the input values.

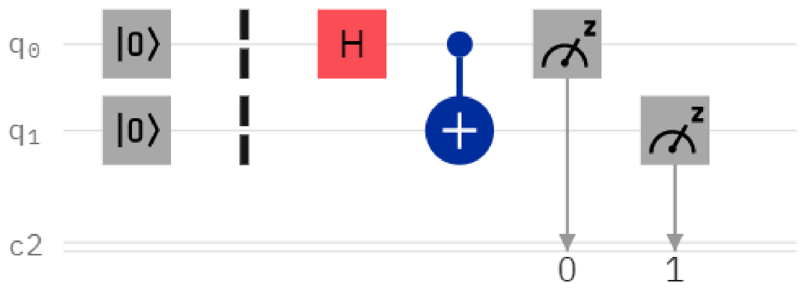


Figure 9. Define this state as $|\psi_0\rangle$ which is equal to $|00\rangle$ here as we have initialized $q_0 = |0\rangle$ and $q_1 = |0\rangle$.

In the next step of analysis $q_0 = |0\rangle$ will be operated on by the Hadamard gate which will place it in both the excited state and ground state simultaneously.

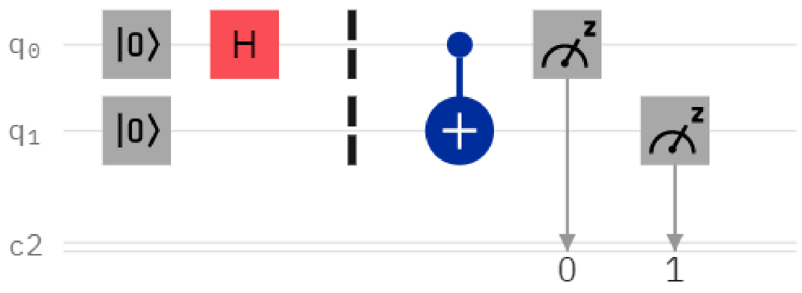


Figure 10. Define this state as $|\psi_1\rangle$ which is equal to $|00\rangle + |10\rangle$ as $q_0 = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ while $q_1 = |0\rangle$ remains true.

We will omit the $\frac{1}{\sqrt{2}}$ for the state $|\psi\rangle$ from here forwards.

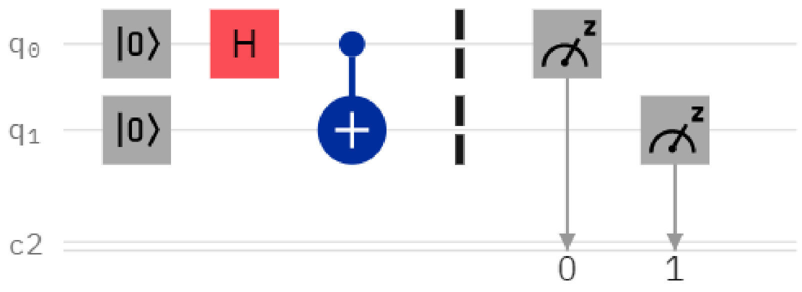


Figure 11. Define this state as $|\psi_2\rangle$ which is equal to $|00\rangle + |11\rangle$ as the CNOT gate has a control on q_0 that does nothing when $q_0 = |0\rangle$ or flips the target q_1, when $q_0 = |1\rangle$

.

This process of explicitly following the state vector as it progresses through a circuit is essential for the analysis of quantum computational circuits. It allows us to know what will happen and what we expect to measure at the end. In reality we tend to get noise from quantum phenomena or from the limitations of using physical quantum computer.

4.2 Logical Encoding

One known limitation is that the quantum computers we are using at IBM tend to decohere towards the ground state due to the nature of the transmon qubits, which are not perfectly isolated from the environment giving the false sense that the state $|00\rangle$ is more likely than $|11\rangle$ in the example of the Bell State. One method for overcoming this physical limitation is to use quantum encoding, where we use two bits to represent a single logical qubit.

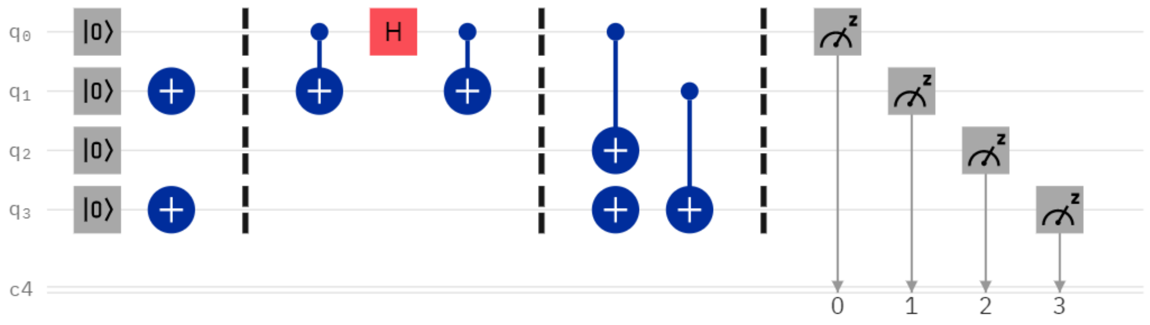


Figure 12. This is the encoded Bell State with a logical $q_{0L} = |01\rangle$ composed of a physical q_0 and q_1 and a logical $q_{1L} = |01\rangle$ composed of a physical q_2 and q_3 where we take $|01\rangle$ to be our logical zero. The circle and plus sign without a target is the NOT gate and flips the qubit from one state to the other.

This logical encoding allows us to use all the same gates as its non-encoded counter-part but divides the decoherence equally between q_{0L} and q_{1L}. The trade-off here is that we have to use twice as many qubits to construct the same function, potentially exposing us to a higher risk of adding noise to our results. This is offset by the fact that we can use the state vectors that result from these runs to see if they have fallen into an allowed state, effectively creating error detection.

With the physical Bell State we had the states $|00\rangle$ or $|11\rangle$ divided into equally probable amplitudes, with some non-zero chance of getting $|01\rangle$ and $|10\rangle$ as shown in the probability distribution function. Likewise, with the encoded version we expect $|00\rangle_L$ and $|11\rangle_L$.

Since we are dealing with a noisy machine, we will end with states that are not allowed. Meaning that some of the resulting state vectors are composed of illogical qubits such that q_{0L} or q_{1L} end up in a state that is not $|01\rangle$ or $|10\rangle$. Since we know that $|00\rangle$ and $|11\rangle$ physical do not match our encoding scheme, we can be certain an error has occurred and remove that result from our data. This is the actual method of error detection used in our results.

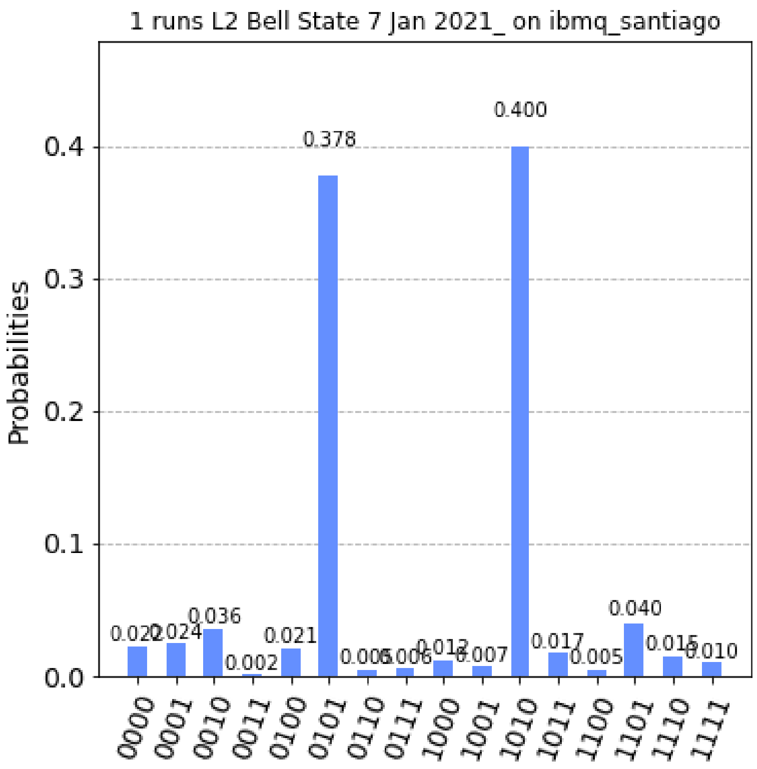


Figure 13. This probability distribution is the result of 8192 shots of the encoded Bell State recorded on the IBM Santiago quantum computer.

By looking at the histogram displayed in Fig. 13 you can see that the overall pattern agrees with what we expect, but there are 12 states that can not possibly exist. The states $|00\rangle_L$ and $|11\rangle_L$ are clearly dominant and the preference towards one or the other is no longer pronounced.

Now if we apply our error detection, meaning any logical qubit that is measured without an allowed encoding we see the power of error detection in action in Fig. 14.

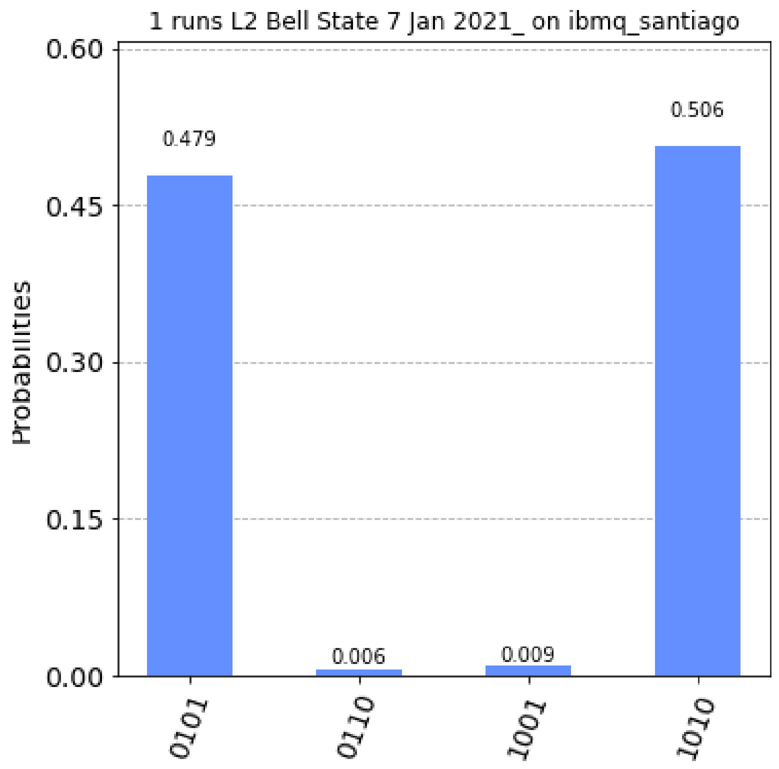


Figure 14. This probability distribution is the result of 6469 shots of the encoded Bell State recorded on the IBM Santiago quantum computer. Note that 1,723 invalid states were removed.

Note that in Fig. 8 we used all 8192 shots and had a 54% to 43% distribution of the $|00\rangle$ and $|11\rangle$ states. We can use these concepts of state analysis and error detection on any circuit that we choose and will demonstrate it's utility on a bit flip error correction circuit in the following sections.

4.2.1 Comparison of Unencoded and Encoded Bell State

Now let us look at 20 runs on the unencoded Bell State and compare the PDFs and similarity measures discussed in Section 3.2.2. We will measure the results of 8192 shots 20 times on each circuit of the Bell State, which will give us a probabilistic distribution of the various state vectors we can expect.

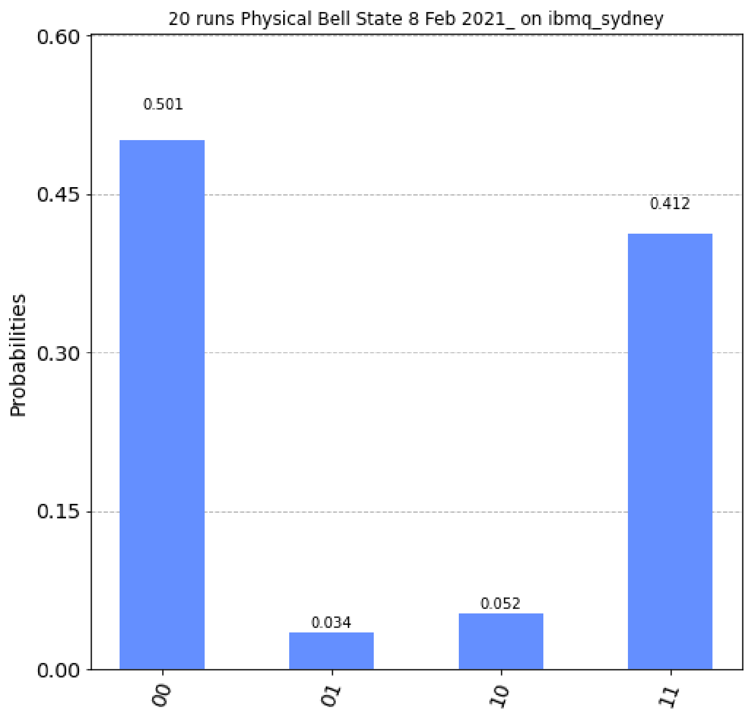


Figure 15. This probability distribution is the result of 163,840 shots of the physical Bell State recorded on the IBM Sydney quantum computer.

In Fig. 15 we again see that even after 160,000 shots are measured we have a clear bias towards the ground state $|\psi_f\rangle = |00\rangle$.

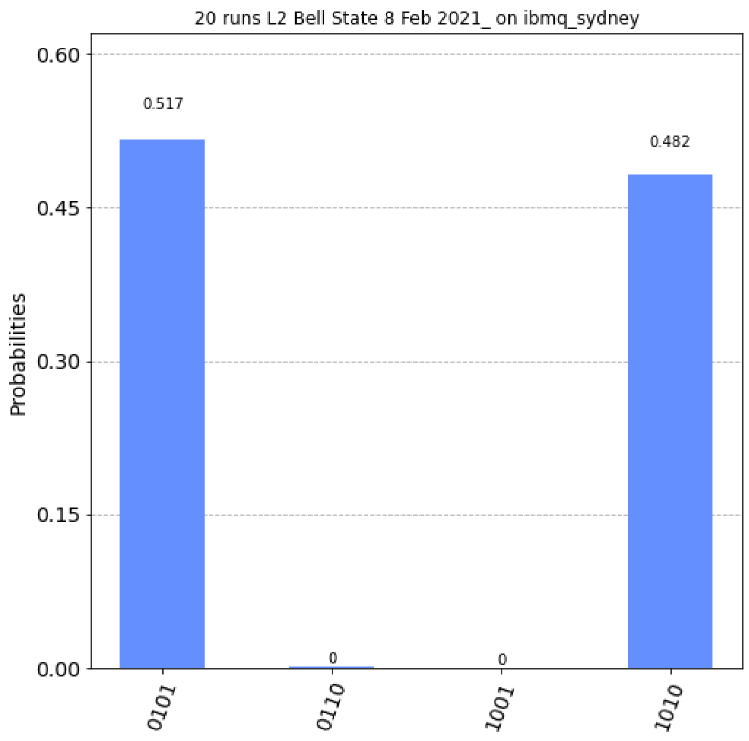


Figure 16. This probability distribution is the result of 137,204 shots of the encoded Bell State recorded on the IBM Sydney quantum computer. Note that 26,636 invalid states were removed.

By inspection we see that in Fig. 16 the distribution is much closer to being equal, where $|\psi_f\rangle_L = \frac{1}{\sqrt{2}}(|00\rangle_L + |11\rangle_L)$, but without an independent way to measure this improvement we are left with only an instinct as to what may have improved or how our encoding scheme has shaped our results. This is where we apply Eq. 7 to each of the 20 runs for the physical, encoded without error detection, and encoded with error correction to get an overlapping histogram of the respective results.

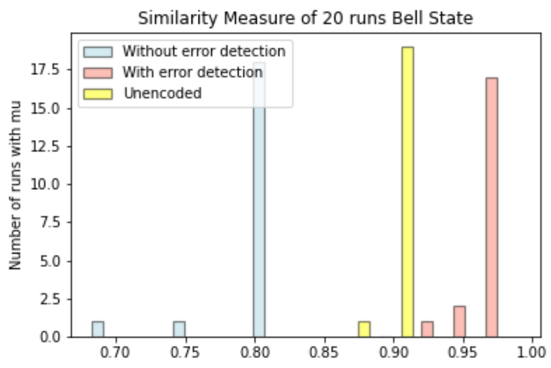


Figure 17. Similarity measure of an encoded Bell State compared to its physical equivalent.

In Fig. 17 blue represents our encoded results without error detection applied, yellow is the unencoded Bell State, and pink is the encoded circuit with error detection applied. We see that the results for the similarity measures tend to be tightly grouped around the 0.80, 0.91, and 0.97 bins respectively, where each bin represents how similar the results are to the analytic solution on a scale from 0 to 1. This shows us that after 160,000 shots that our error detection method does give repeatable and more consistent results than that of the the unencoded equivalent circuit.

V. Results

5.1 Results

We want to use or error detection encoding of logical qubits using two-level physical qubits to implement the three-bit bit flip error correction code. To do this we must first look at the physical representation of the bit flip algorithm and see its utility then address the fact that the Toffoli gate is a three-qubit gate that cannot be directly encoded using physical two-level qubits.

5.1.1 The Three-Qubit Error Correction Code

This piece of code was first proposed by Asher Peres in 1985 as a means to alleviate the susceptibility of quantum states to spontaneous flips [15].

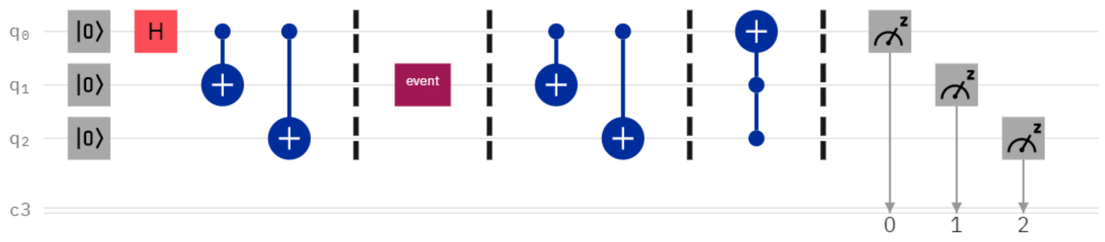


Figure 18. Thee-qubit bit flip error correction algorithm used to correct an error and identify when an error has occurred.

Here we arbitrarily choose the Hadamard gate to represent an input wave $|\psi_0\rangle$ that will run through our bit flip error correction algorithm. The purple box represents an event that could be any one of the three qubits flipping or all three qubits progressing unchanged. The Hadmard and two CNOT gates create the state $|\psi_1\rangle = |000\rangle + |111\rangle$ and this first part of the circuit is called the encoding. We have encoded our $|\psi_0\rangle$ onto the $|q_1\rangle$ and $|q_2\rangle$ qubits.

After the event, one or none of the qubits changes states, we can expect one of four possible states: no flips occurred, q_0 flipped, q_1 flipped, or q_2 flipped. Which leaves us with four possible $|\psi_2\rangle$ states immediately after the event.

The next two CNOT gates will handle the decoding of the flipped qubit and give us a state that will be operated on by the Toffoli gate and repair the flip or let us know that no flips have occurred.

Let us follow the case where no flip occurs. If we take $|\psi_2\rangle = |000\rangle + |111\rangle$ immediately before the event then assume no flips took place:

- The state $|000\rangle$ does not set the control gates high, therefore the q_1 and q_2 qubits continue unchanged.
- The alternate state $|111\rangle$ sets the control gate to high, therefore the q_1 and q_2 qubits flip.
- The result is $|\psi_3\rangle = |000\rangle + |100\rangle$ and this will be acted on by the Toffoli gate.

This leaves our original Hadamard intact on $|q_0\rangle = |0\rangle + |1\rangle$ with the $q_1 = 0$ and $q_2 = 0$. We call $|00\rangle$ the syndrome and the syndrome says no flips have occurred at the event. Finally the Toffoli gate does not have both controls set to flip so we arrive at $|\psi_f\rangle = |000\rangle + |100\rangle$.

Table 1. State analysis after the event where either q_0, q_1, or q_2 are flipped. $|\psi_2\rangle$ is immediately after the event, $|\psi_3\rangle$ is decoded by the second set of CNOTs, and $|\psi_f\rangle$ is the final state after the Toffoli.

Flipped	$\|\psi_2\rangle$	$\|\psi_3\rangle$	$\|\psi_f\rangle$
None	$\|000\rangle + \|111\rangle$	$\|000\rangle + \|100\rangle$	$\|000\rangle + \|100\rangle$
q_0	$\|100\rangle + \|011\rangle$	$\|111\rangle + \|011\rangle$	$\|011\rangle + \|111\rangle$
q_1	$\|010\rangle + \|101\rangle$	$\|010\rangle + \|110\rangle$	$\|010\rangle + \|110\rangle$
q_2	$\|001\rangle + \|110\rangle$	$\|001\rangle + \|101\rangle$	$\|001\rangle + \|101\rangle$

In every case shown in Table 1 we retain our original $|\psi_0\rangle$ in the form of a Hadamard. And the $q_1 + q_2$ return the same values for each of the respective results. This is useful because we now know exactly what happened and retained our original waveform.

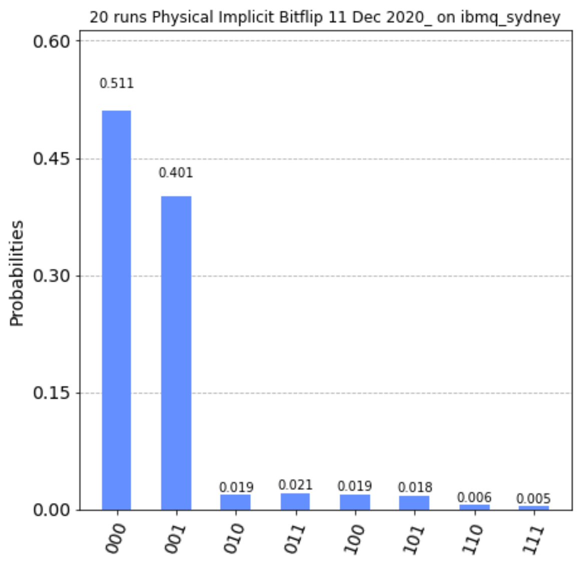
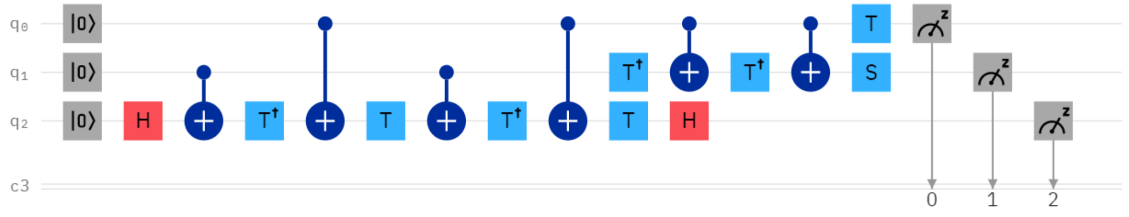


Figure 19. Probability distribution function for a bit flip error correction circuit where we expect $|\psi_f\rangle = |000\rangle + |100\rangle$.

The results in Fig. 19 show that over 90% of our measurements are in the correct states, but there appears to be a preference for the $|\psi_f\rangle = |000\rangle$ state. This is a result of having run this computation on a physical computer that tends to decohere towards the ground state.

5.1.2 Toffoli Gate

In Fig. 4b we showed how each of the input states on the multiple control gate will act on the target. To create this same truth table out of two-qubit gates we will have to engineer a gate that has the same characteristics as the Toffoli while only using one and two qubit operators.



(a) **Toffoli equivalent using two and one qubit gates with $q_0 = |0\rangle$, $q_1 = |0\rangle$, and $q_2 = |0\rangle$.**

$\lvert q_0\rangle$	$\lvert q_1\rangle$	$\lvert q_2\rangle$	$\lvert z_0\rangle$	$\lvert z_1\rangle$	$\lvert z_2\rangle$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

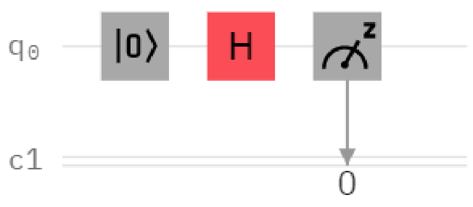
(b) **Truth table for a Toffoli equivalent using two and one qubit gates.**

Figure 20. Toffoli equivalent using two and one qubit gates where q_0 and q_1 are the controls and q_2 the target.

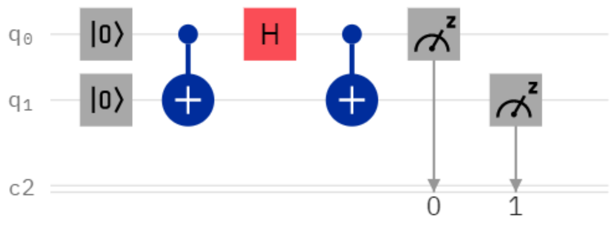
The code shown in Fig. 20 operates exactly the same as a Toffoli gate and is now suitable for use in our error detection encoding [2]. One aspect that we have neglected is that every wave form has a phase associated with it on the Bloch Sphere and up to this point we have been able to ignore it. To control for the phase shift as we use this equivalent circuit we must us the T, T†, and S operators. Where the S gate induces a $\frac{\pi}{2}$ phase shift about the z-axis and the T gate induces a $\frac{\pi}{4}$ phase shift.

5.1.3 Encoding the Toffoli Gate

In Section 4.2 we showed the encoded Bell State as having two physical qubits per a single logical qubit. Here will will further breakdown that process for application to our Toffoli equivalent code.



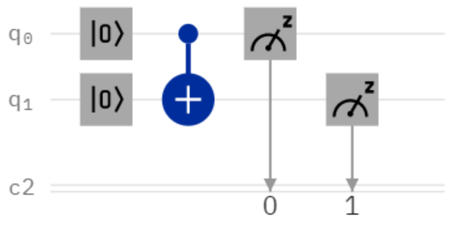
(a) Hadamard gate with $q_0 = |0\rangle$ and $q_1 = |0\rangle$.



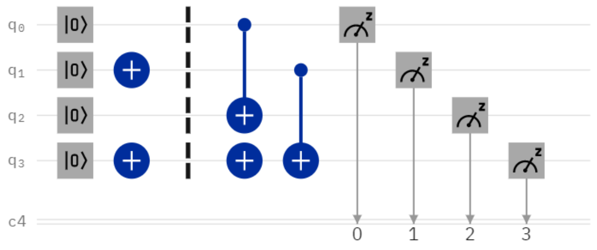
(b) Encoded Hadamard with $q_0 = |0\rangle$ and $q_1 = |0\rangle$.

Figure 21. In subfigure (a) the single qubit hadmard puts q_0 into a linear superposition but leaves q_1 unchanged yielding $|\psi\rangle = |00\rangle + |10\rangle$. In subfigure (b) the encoded Hadamard places both q_0 and q_1 into a linear superposition creating the $|\psi\rangle = |00\rangle + |11\rangle$ state.

This encoding in Fig. 21 creates a single logical qubit using the q_0 and q_1 physical qubits which will allow us to use it in a larger encoded circuit. This same encoding of two qubits for a single qubit operator applies to the T, T†, and S operators.



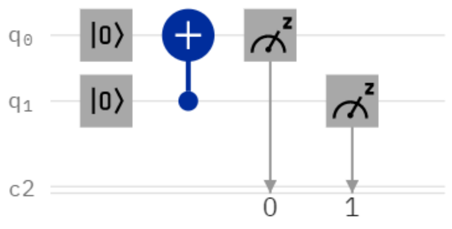
(a) CNOT with control q_0 and target q_1.



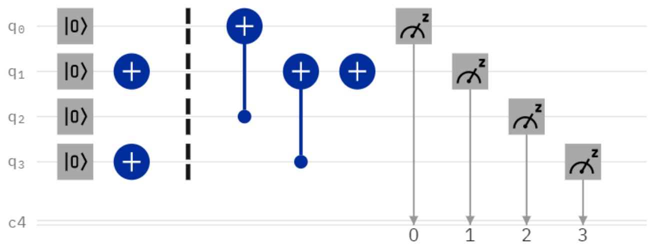
(b) Encoded CNOT control q_{0L} and target q_{1L}.

Figure 22. In subfigure (a) we have a single CNOT With two physical qubits. In subfigure (b) the encoded equivalent has inputs $q_{0L} = |0\rangle_L$ which is made with physical $q_0 = |0\rangle$ and $q_1 = |1\rangle$. While $q_{1L} = |0\rangle_L$ is composed of $q_2 = |0\rangle$, and $q_3 = |0\rangle$.

Both the physical and encoded CNOTs in Fig. 22 share the exact same truth table shown in Fig. 3 with the noted exception that each logical qubit is composed of either $|01\rangle$ or $|10\rangle$. One interesting point that should be shown before moving forward is the case where the CNOT target is on a lower qubit that the control. In the physical building of the circuit we just change which qubit functions as a control or target by turning the CNOT gate upside down. When you encode the CNOT you must take in to consideration that the appropriate matrices require some minor modifications.



(a) CNOT with target q_0 and control q_1.



(b) Encoded CNOT with target q_{0L} and control q_{1L}.

Figure 23. In subfigure (a) we have a single CNOT With two physical qubits. In subfigure (b) the encoded equivalent has inputs $q_{0L} = |0\rangle_L$ which is made with physical $q_0 = |0\rangle$ and $q_1 = |1\rangle$. While $q_{1L} = |0\rangle_L$ is composed of $q_2 = |0\rangle$, and $q_3 = |1\rangle$.

Both configurations shown in Fig. 22 and Fig. 23 will results in a valid truth table and can be used to construct an encoded circuit depending on the needs of the circuit.

Now looking at Fig. 20 let us build its coded equivalent of just the Toffoli gate.

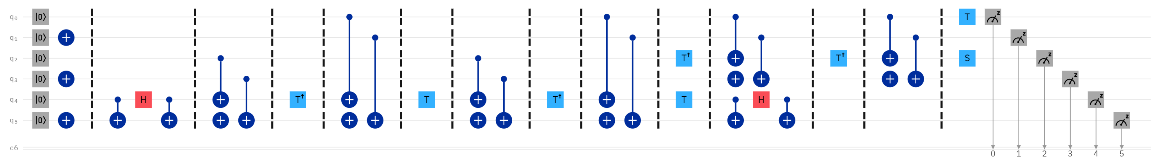


Figure 24. Encoded Toffoli with controls on q_{0L} and q_{1L} and target on q_{2L}.

Note that this encoded Toffoli in Fig. 24 is identical in its function to the non-encoded version in Fig. 20. If we run these two circuits though the IBM computers we can see that the statistical distributions agree well with the expected states found in their respective truth tables.

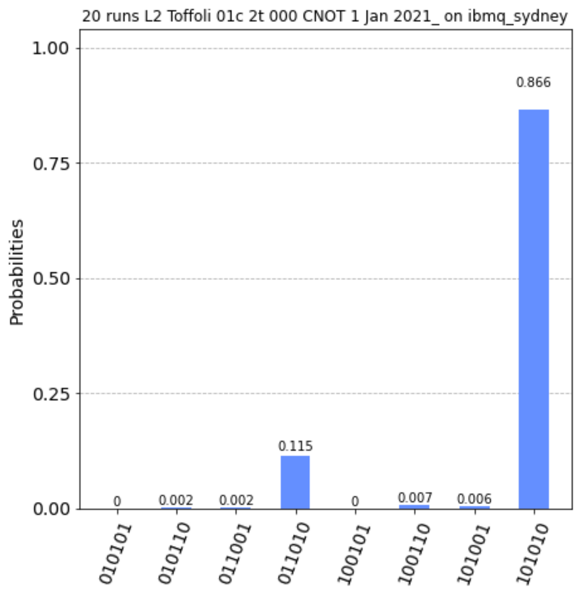


Figure 25. Probability Distribution Function of an encoded Toffoli gate with $|\psi_f\rangle_L = |000\rangle_L$ where the right-most state $|\psi_f\rangle = |010101\rangle$ as read from top to bottom.

Figure 25 is exactly what we expect to see on the truth table of a Toffoli gate. You must read the measured results from top to bottom, with the top number being the first qubit, $|q_0\rangle$ and the lowest number being $|q_5\rangle$. So the final state would be $|\psi_f\rangle = |010101\rangle$ which is our $|000\rangle_L$ equivalent. When compared to the results of our probability distribution function of a physical gate in Fig. 26 we see that there is agreement.

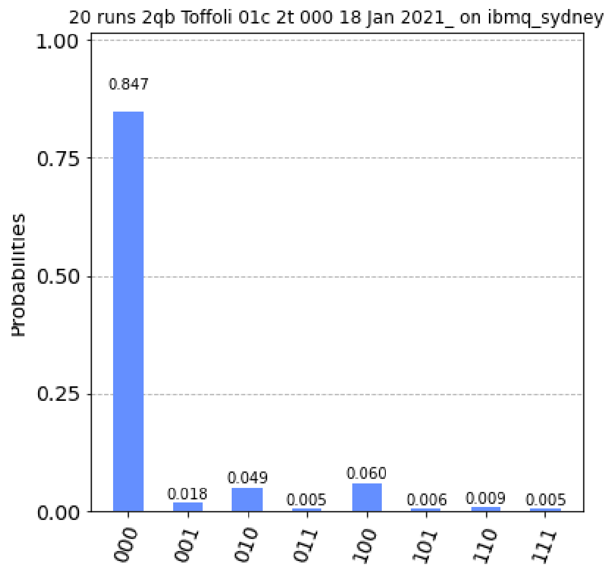


Figure 26. Probability Distribution Function of an unencoded Toffoli gate with $|\psi_f\rangle = |000\rangle$.

While we can see that they are in agreement it is important to have a way to measure this, so we want to compare each run of 8192 shots on the IBM quantum computers and use each similarity measure to see if the results are better or the same.

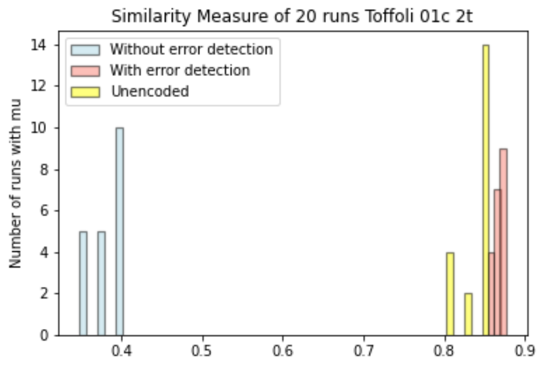


Figure 27. Similarity measure of an encoded Toffoli compared to its physical equivalent.

The yellow bins in Fig. 27 represent the similarity measure of the unencoded circuit in Fig. 20. The blue and pink are the similarity measure of the circuits in Fig. 24 where blue doesn't use error detection and pink does.

5.1.4 Encoding Bit Flip Error Correction

We have now built up a library of functions we can use to implement our error correction scheme while simultaneously encoding logical qubits to utilize error detection. Let us look at the encoded version of Fig. 18 shown here in Fig. 28.

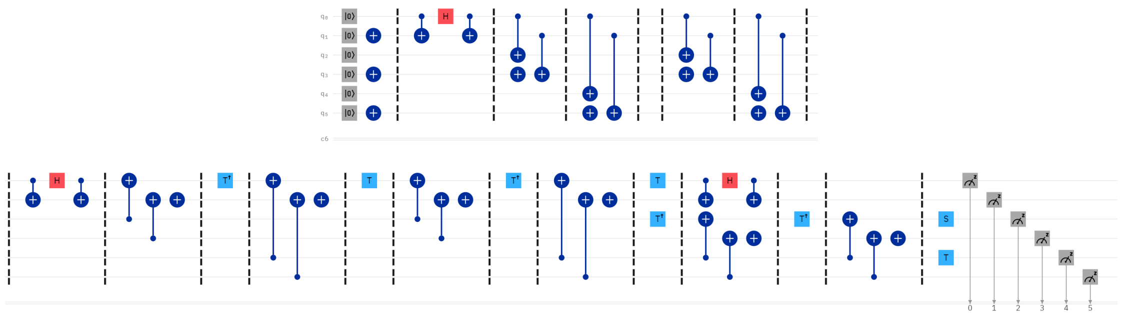


Figure 28. Fully encoded 3-qubit bit flip error correction algorithm. Where we expect $|\psi_f\rangle_L = |000\rangle_L + |100\rangle_L$

We know from previous work that error detection encoding will improve the probability distribution function towards a more reliable result when dealing with a simple Bell State. We want to apply this same method to an even larger circuit in hopes of achieving similar results and have arbitrarily chosen the bit flip error correction circuit. If we look at the results of all 64 possible states of the encoded bit flip error correction circuit it appears to be very noisy at a glance.

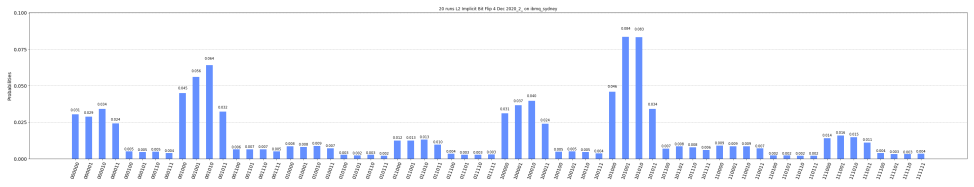


Figure 29. Probability Distribution Function of all 64 possible states on an encoded bit flip error correction circuit where we expect $|\psi_f\rangle_L = |000\rangle_L + |100\rangle_L$.

In Fig. 29 $|000\rangle_L$ and $|100\rangle_L$ are the largest peaks but without applying error detection it is difficult to see that any improvement, if any has occurred.

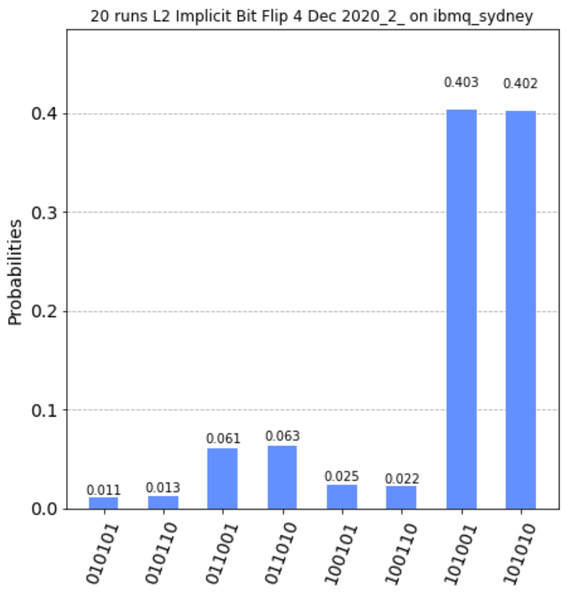


Figure 30. Probability Distribution Function after applying error detection on an encoded bit flip error correction circuit where we expect $|\psi_f\rangle_L = |000\rangle_L + |100\rangle_L$.

If we compare Fig. 30 to the probability distribution function of the unencoded version of the bit flip seen in Fig. 29, two critical observations are made. We have reduced the number of expected $|000\rangle$ and $|100\rangle$ measurements, but have also balanced the probability of either state being measured effectively eliminating any bias towards the ground state!

Using our similarity measure in Fig. 31 which simply returns the deviation from the expected probability distribution this circuit appears to have decreased the overall reliability of our measurements.

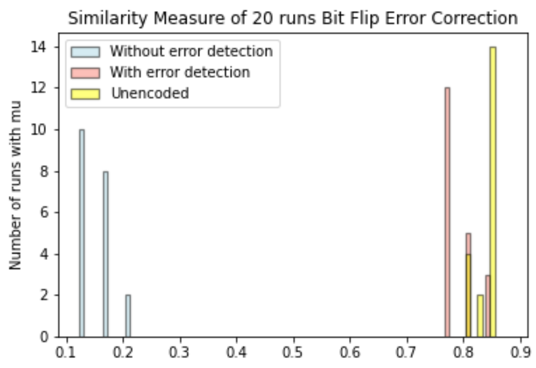


Figure 31. Similarity measure of 20 run each using 8192 shots of an encoded and unencoded bit flip correction circuit.

The trade off of balancing the distribution of expected states of a linear superposition with the cost of having fewer measurements in those states is the final result of this work. It is unclear if balancing the distributions is better than having more measurements in the expected states for every application.

If we break down the process in the same way as section 5.1.1 we can see how the depth of the circuit causes some level of deterioration in our improvement using the logical encoded qubits.

We established the intial state as $|\psi_0\rangle = |010101\rangle + |100101\rangle$ which is the logical $|\psi_0\rangle_L = |000\rangle_L + |100\rangle_L$.

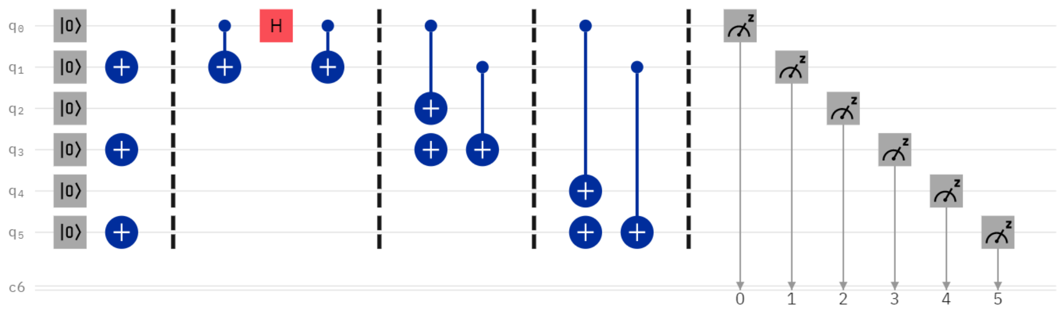


Figure 32. Encoded bit flip error correction circuit immediately after the first two CNOTs.

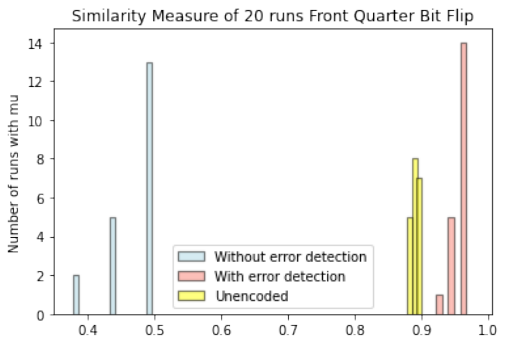


Figure 33. Similarity measure of 20 runs each using 8192 shots of an encoded and unencoded bit flip error correction circuit immediately after the first two CNOTs.

From Fig. 32 and the similarity measure shown in 33 a marked improvement over the unencoded portion of this circuit that is even stronger than the improvement of the Toffoli gate. Next we decode our state vector and look at the new $|\psi_3\rangle_L = |000\rangle_L + |100\rangle_L$.

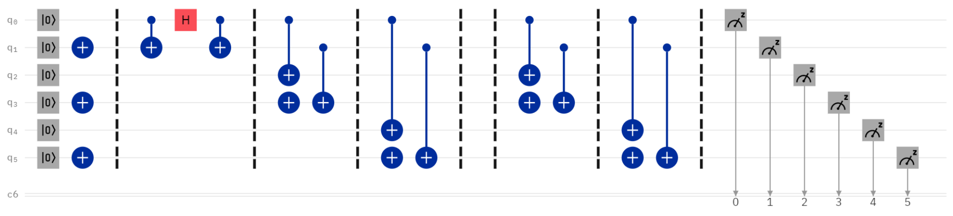


Figure 34. Encoded bit flip error correction circuit immediately after the second two CNOTs.

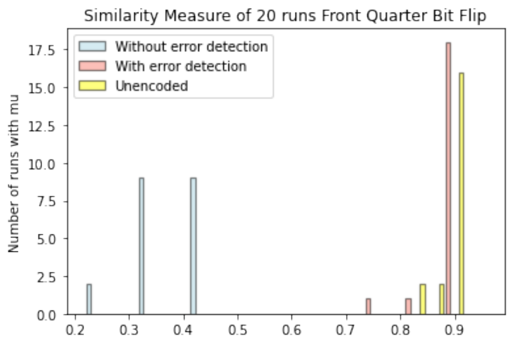
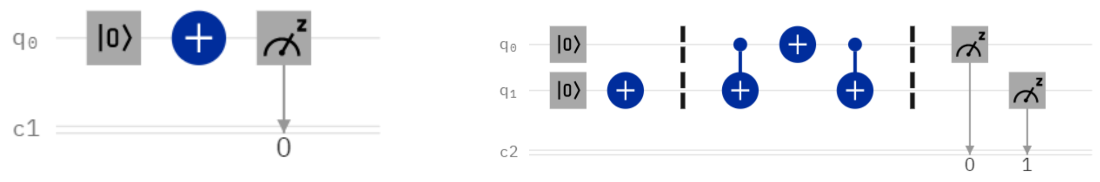


Figure 35. Similarity measure of 20 runs each using 8192 shots of an encoded and unencoded bit flip error correction circuit immediately after the first two CNOTs.

It appears that the depth of the circuit shown in Fig. 34 with similarity measures in 35 creates some net neutral return when it comes to using only the similarity measure, but the end results is a well balanced probability distribution function. There are other similarity measures that could weigh the delta from the expected values more heavily that could show that the results of the encoded circuits is even more beneficial than we have already demonstrated. The similarity measure chosen shows only the raw difference in the expectation values from those measured.

5.1.5 Logical Bit Flip

To ensure the robustness of our circuit and its ability to send an error syndrome for a flip on the $|q_0\rangle_L$, $|q_1\rangle_L$, or $|q_2\rangle_L$ we must run the circuit using a logical bit flip. Much like the CNOT, Hadamard, or Toffoli, a logical bit flip is not as simple as a single physical bit flipping inside of our encoded circuit. This would not represent a logical flip and would ultimately just lead to an invalid state that our error detection would reject. To get a valid bit flip we must use a logical Pauli X.



(a) Pauli X with output $q_0 = |1\rangle$. **(b) Logical Pauli with output $q_0 = |1\rangle_L$.**

Figure 36. In subfigure (a) Pauli X (NOT) gate with input $q_0 = |0\rangle$ and output $q_0 = |1\rangle$ representing a induced flip. (b) the encoded Pauli X forces both q_0 and q_1 to flip using the two CNOT gates taking our input $|\psi\rangle = |01\rangle$ and flippling it to $|10\rangle$, the logical equivalent to subfigure (a).

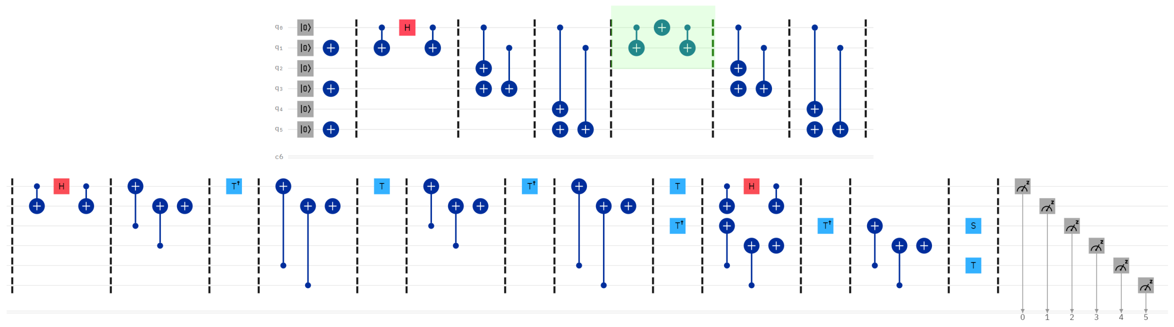


Figure 37. Fully encoded 3-qubit bit flip error correction algorithm. Where we expect $|\psi_f\rangle_L = |000\rangle_L + |100\rangle_L$, but have introduced a logical bit flip on the q_{0L} qubit shown highlighted in green

Using the logical Pauli X shown in Fig. 36 in Fig. 37 we would expect this flip to give us the identical syndrome for a flip on the q_0 qubit but in its logical form resulting in $|\psi_f\rangle_L = |011\rangle_L + |111\rangle_L$ as shown on the truth table in Fig. 1.

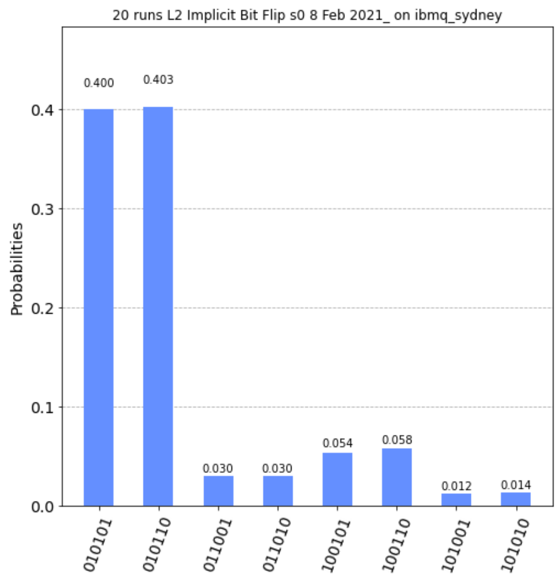


Figure 38. Probability distribution function a bit flip error correction circuit where we expect $|\psi_f\rangle_L = |011\rangle_L + |111\rangle_L$.

In Fig. 38 we prove that the logic of our encoded bit flip error correction circuit holds true even under the improbable conditions that would induce a logical Pauli X flip. We measured $|\psi_f\rangle = |011010\rangle + |101010\rangle$, again read from top to bottom and written left to right, which is exactly the logical equivalent under our encoding expected from the truth table. Furthermore, the truth table holds for all four logical equivalents of flips on q_{0L}, q_{1L}, q_{2L}, or no flips.

VI. Conclusion

6.1 Future Work

There are many different gates and circuits that were used throughout this research and each one is fundamental to the entire body of quantum computational research. We are limited by only having access to one type of quantum computer, those made using transmons at IBM, and the unitary gates provided to create our gates. Using these available gates and operators we can continue to explore more complicated and complex circuits to investigate the benefit of error detection or error correction further. Another avenue of potential research would be to see if these encodings improve results on other quantum computers using alternate infrastructures.

The results of our similarity measurements in the encoded bit flip error correction circuit is likely due to phase flips that occur as the computation progresses. We did not focus on or measure phase flips in this work. It is likely that if we used error detection and phase flip detection we could improve these results.

An alternate decomposition of the Toffoli gate could be created using fewer phase gates and could reduce the number of unitary gates required to run the circuit decreasing the depth of the circuit and allowing us to reduce the time risk has to occur.

6.2 Final Thoughts

We have created a useful tool to use unitary operators and error detection implemented in other circuits requiring a Toffli gate that would allow for efficient error detection. The similarity measures and probability distributions showed superior results to the unencoded equivalent gate while reducing bias towards the ground state resulting from the engineering limitations of a physical quantum computer. These results did not hold through a larger circuit, but with the addition of phase flip detection and correction there are multiple avenues to continue working around the technical limitation in the current era of modern quantum computing. We have used what is available to us to take one small step forward for creating viable quantum computational algorithms.

Appendix A. Physical and Logical Circuits Implemented in Qiskit

```

'''CNOT'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.cx(qreg_q[1], qreg_q[0])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
```

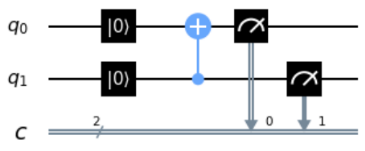


Figure 39. The above source code will provide the unencoded CNOT gate in qiskit for use on IBM quantum computer.

```

'''Toffoli'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(3, 'q')
creg_c = ClassicalRegister(3, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[2])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
```

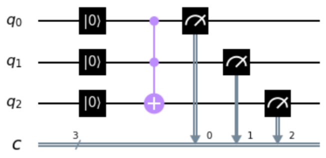


Figure 40. The above source code will provide the unencoded Toffoli gate in qiskit for use on IBM quantum computer.

```

'''Hadamard'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.h(qreg_q[0])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
```

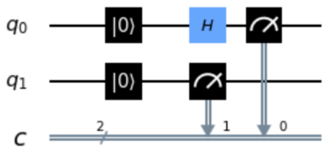


Figure 41. The above source code will provide the unencoded Hadamard gate in qiskit for use on IBM quantum computer.

```
'''Bell State'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.barrier(qreg_q[0], qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
```

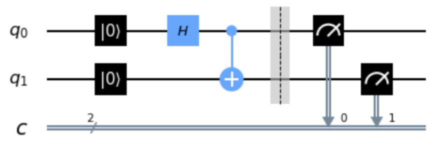


Figure 42. The above source code will provide the unencoded Bell State in qiskit for use on IBM quantum computer.

```
'''Encoded Bell State'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(4, 'q')
creg_c = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.reset(qreg_q[3])
circuit.x(qreg_q[1])
circuit.x(qreg_q[3])
circuit.barrier(qreg_q[1], qreg_q[0], qreg_q[2], qreg_q[3])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.barrier(qreg_q[3], qreg_q[0], qreg_q[1], qreg_q[2])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.x(qreg_q[3])
circuit.cx(qreg_q[1], qreg_q[3])
circuit.barrier(qreg_q[0], qreg_q[1], qreg_q[2], qreg_q[3])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
circuit.measure(qreg_q[3], creg_c[3])
```

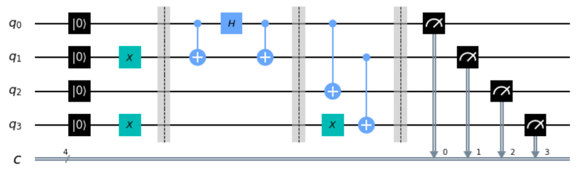


Figure 43. The above source code will provide the encoded Bell State in qiskit for use on IBM quantum computer.

```
'''Bit Flip Error Correction'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(3, 'q')
creg_c = ClassicalRegister(3, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.barrier(qreg_q[1], qreg_q[0], qreg_q[2])
circuit.barrier(qreg_q[2], qreg_q[1], qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.barrier(qreg_q[2], qreg_q[1], qreg_q[0])
circuit.ccx(qreg_q[2], qreg_q[1], qreg_q[0])
circuit.barrier(qreg_q[0], qreg_q[1], qreg_q[2])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
```

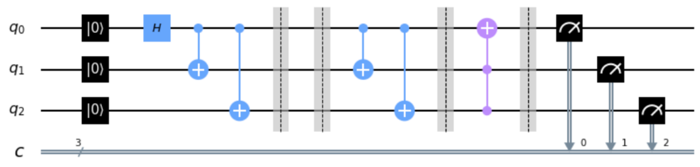


Figure 44. The above source code will provide the unencoded bit flip error correction circuit in qiskit for use on IBM quantum computer.

```
'''Toffoli Equivalent'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(3, 'q')
creg_c = ClassicalRegister(3, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.h(qreg_q[2])
circuit.cx(qreg_q[1], qreg_q[2])
circuit.tdg(qreg_q[2])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.t(qreg_q[2])
circuit.cx(qreg_q[1], qreg_q[2])
circuit.tdg(qreg_q[2])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.tdg(qreg_q[1])
circuit.t(qreg_q[2])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.h(qreg_q[2])
circuit.tdg(qreg_q[1])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.t(qreg_q[0])
circuit.s(qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
```

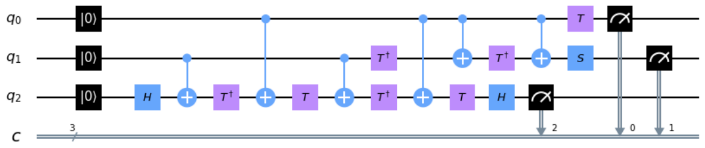


Figure 45. The above source code will provide the unencoded Toffoli equivalent circuit in qiskit for use on IBM quantum computer.

```
'''Encoded Hadmard'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
```

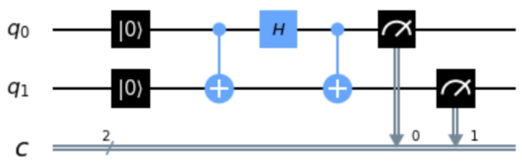


Figure 46. The above source code will provide the encoded Hadamard gate in qiskit for use on IBM quantum computer.

```
'''Encoded CNOT'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(4, 'q')
creg_c = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.reset(qreg_q[3])
circuit.x(qreg_q[1])
circuit.x(qreg_q[3])
circuit.barrier(qreg_q[3], qreg_q[2], qreg_q[1], qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[2])
circuit.x(qreg_q[3])
circuit.cx(qreg_q[1], qreg_q[3])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
circuit.measure(qreg_q[3], creg_c[3])
```

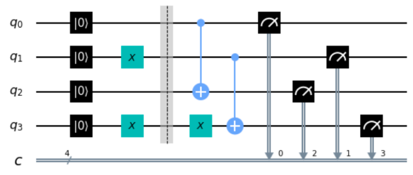


Figure 47. The above source code will provide the encoded CNOT gate in qiskit for use on IBM quantum computer.

```python

'''Encoded NOTC'''

from qiskit import QuantumRegister, ClassicalRegister,
    ↪ QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(4, 'q')
creg_c = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.reset(qreg_q[0])
circuit.reset(qreg_q[1])
circuit.reset(qreg_q[2])
circuit.reset(qreg_q[3])
circuit.x(qreg_q[1])
circuit.x(qreg_q[3])
circuit.barrier(qreg_q[3], qreg_q[2], qreg_q[1], qreg_q[0])
circuit.cx(qreg_q[2], qreg_q[0])
circuit.cx(qreg_q[3], qreg_q[1])
circuit.x(qreg_q[1])
circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])
circuit.measure(qreg_q[2], creg_c[2])
circuit.measure(qreg_q[3], creg_c[3])
```

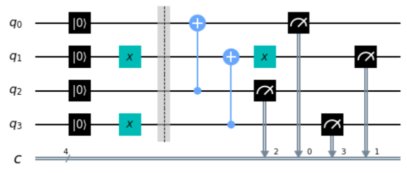


Figure 48. The above source code will provide the encoded inverted CNOT gate in qiskit for use on IBM quantum computer.

Appendix B. Analysis and Parsing of IBM Q Data

```python
'''Established backends associated with an account'''

from qiskit import IBMQ
import numpy as np
# API_TOKEN = '[Token]'
# IBMQ.save_account(API_TOKEN)

Public_Provider = IBMQ.load_account()
AFRL_Provider = IBMQ.get_provider(hub = 'ibm-q-afrl')
AFRL_Provider.backends()
Sim_Backend = AFRL_Provider.get_backend(name = '
    ↪ ibmq_qasm_simulator')
Toronto_Backend = AFRL_Provider.get_backend(name = '
    ↪ ibmq_toronto')
Sydney_Backend = AFRL_Provider.get_backend(name = 'ibmq_sydney
    ↪ ')
```

```python

'''Establishes which backend to use for following code'''

# backend = Sim_Backend
# backend = Toronto_Backend
backend = Sydney_Backend
```

```python

'''Runs the circuit on the assigned backend
   sending multiple runs with named jobs'''

from qiskit import IBMQ, execute

for i in range(20): # Set range as number of runs required

    JobName = 'Physical Implicit Bitflip 11 Dec 2020_' + str(i
        ↪ ) # Creates an iterated number to uniquely tag each
        ↪ job for retrieval
    execute(circuit, backend, optimization_level=2, shots =
        ↪ 2**13, job_name=JobName)
```

```python

'''Establishes JobsRan as a list of all the jobs fitting the
    ↪ filters below for a given backend'''

JobSearch = 'Physical Implicit Bitflip 11 Dec 2020_'
data_dump = JobSearch + str(backend) + '.txt' # Unique name to
    ↪  write out counts and related data

JobsRan = backend.jobs(limit=20, job_name=JobSearch, status='
    ↪ DONE')
# job_name can be found on list of jobs on IBMQ
# job_name = 'bar' will return jobs with 'bar' in them
# such as 'fubar', 'crowbar_01', ect

print(len(JobsRan))
print(type(JobsRan))
```

```

'''Writes a file of all the counts and related data associated
   ↪  with each run into the unique file name below'''

f1 = open(data_dump,'w') # Date_backend_circuit

for i in range(len(JobsRan)): # Iterates through each job in
   ↪ JobsRan to output desired information

    JobId = str(JobsRan[i]._job_id)
    f1.write('#' + str(i)+'\n')
    f1.write('Date job created: ' + str(JobsRan[i].
       ↪ creation_date()) +'\n')
    f1.write('Job ID: ' + JobId +'\n')
    f1.write('Job Name: ' + str(JobsRan[i]._name) +'\n')

    old_job = backend.retrieve_job(JobId)
    f1.write('Backend: ' + str(old_job._backend) + '\n')
    old_result = old_job.result()
    old_counts = old_result.get_counts()
    f1.write('Results: \n' + str(old_counts) + '\n\n')

f1.close()
```

```

'''Processses the data_dump.txt file into dictionaries of
   ↪ results and uses the parse_data() function
   to seperate valid and invalid entries'''

def parse_data(newcounts, allcounts):

    qb = 6 # Qubits used on physical computer
    lb = 3 # Number of logical bits total
    zero = '01'
    one = '10'

    values = []
    binary = []
    sigfig = []

    keys = range(2**lb) # Assigns the total number of possible
       ↪  states 2^n, where n is the number of logical qubits

    dicts = {}
    new_counts = {}


    for i in keys:
        bitcountflag = 0 # Flag is used to reset counting from
           ↪  000 to 111 in binary
        values.append('placeholder') # Appends a dummy
           ↪ variable to be overwritten into the values[] list
        binary.append(f'{i:06b}') # Expresses i in binary form
           ↪  from 000000 to 111111

        for j in range(lb): # Iterates the total length of
           ↪ logical bits
```

```
            sigfig.append('sf'+str(j)) # Sets the sigfig right
               ↪  to left to be converted, so in 001, j == 0
               ↪ selects "1"
            sigfig[j] = binary[i][qb-j-1] # Pulls integer
               ↪ values of the binary bit from right to left

            if sigfig[j] == '0': # If the value is 0, the
               ↪ logical zero replaces the respective sigfig
               ↪ integer
                sigfig[j] = zero
            else:
                sigfig[j] = one  # If the value is 1, the
                   ↪ logical zero replaces the respective
                   ↪ sigfig integer
            bitcountflag = bitcountflag + 1 # Counts up to the
               ↪  number of physical bits (qb) minus logical
               ↪ bits (lb)

            if bitcountflag == (qb - lb): # When the logical
               ↪ representation of each bit is set the values
               ↪ are concatenated
                values[i] = sigfig[lb-lb]+sigfig[lb-(lb-1)]+
                   ↪ sigfig[lb-(lb-2)] # In our example above
                   ↪ 001 becomes 010110

    for j in keys: # Now the key:value pairs in each
       ↪ dicationary are exchanged
        try:
            new_counts[values[j]] = older_counts[values[j]] #
               ↪ Logical bit values are replaced by their
               ↪ physical counts
# If a key is not a valid logical bitsequence, it is not
   ↪ copied
        except KeyError:
            new_counts[values[j]] = 0 # If a dictionary value
               ↪ is blank it is replaced with a zero


    return new_counts, older_counts # Returns new_counts as a
       ↪ dictionary with keys of valid logical sequences
                                    # and old_counts with keys
                                       ↪  of all possible
                                       ↪ sequences

import csv
n = 0
m = 0
d = []
ind_counts = []
ind_valid_counts = []
new_counts = {}

with open(data_dump, 'r') as f: # data_dump is created in the
   ↪ proceess that parses counts and related data
    reader = csv.reader(f, delimiter='\n')
    for row in reader:
        try:
            if row[n].startswith('{'): # Grabs all
```

```
               ↪ dictionaries of counts as strings from
               ↪ data_dump and places them in d[]
                d.append(row[0])
            else:
                pass
        except IndexError:
            pass


older_counts = eval(d[0]) # Places the string key:values of d
   ↪ [0] as a dictionary
parsed_data2 = parse_data(new_counts, older_counts) # Uses the
   ↪  parse_data function to only select the valid counts
# parsed_data2[0] are valid counts parsed_data2[1] are all
   ↪ counts
# parsed_data2 is a list that holds the dummy values to add up
   ↪  counts

for i in range(len(JobsRan)):

    new_counts = {}

    older_counts = eval(d[i]) # See comments above for details
    parsed_data = parse_data(new_counts, older_counts) # See
       ↪ comments above for breakdown of parse_data function
    ind_counts.append(parsed_data[1])
    ind_valid_counts.append(parsed_data[0])


    if parsed_data[0] != parsed_data2[0]: # Adds up all the
       ↪ counts from the JobsRan in the allowed states
        for key in parsed_data[0]: # parsed_data is a tuple
           ↪ with the processed dictionary in the 0th position

            if key in parsed_data2[0]: # Ensures given key is
               ↪ in both dictionaries
                parsed_data2[0][key] = str(int(parsed_data2
                   ↪ [0][key]) + int(parsed_data[0][key])) #
                   ↪ Sums up values of dictionaries
            else:
                pass
    else:
        pass


    if parsed_data[1] != parsed_data2[1]: # Adds up all the
       ↪ counts from the JobsRan in all states
        for key in parsed_data[1]: # parsed_data is a tuple
           ↪ with the unprocessed dictionary in the 1st
           ↪ position

            if key in parsed_data2[1]: # Ensures given key is
               ↪ in both dictionaries
                parsed_data2[1][key] = str(int(parsed_data2
                   ↪ [1][key]) + int(parsed_data[1][key])) #
                   ↪ Sums up values of dictionaries
            else:
                pass
```

```python
    else:
        pass

sum_all_counts = parsed_data2[1]
sum_all_valid_counts = parsed_data2[0]
# ind_counts are all individual counts
# ind_valid_counts are all individual valid counts

theoretical_all_counts = {'110110': 0, '100011': 0, '010110': 0, '100101': 0, '010000': 0, '001101': 0, '100100': 0, '110011': 0, '100001': 0, '000111': 0, '110000': 0, '101100': 0, '011101': 0, '001100': 0, '011011': 0, '010011': 0, '001010': 0, '110111': 0, '101111': 0, '111110': 0, '011001': 0, '110101': 0, '010001': 0, '001011': 0, '001001': 0, '101001': int(2**13/2), '111101': 0, '010010': 0, '011111': 0, '011010': 0, '101000': 0, '001110': 0, '100110': 0, '110010': 0, '000100': 0, '001111': 0, '010111': 0, '011000': 0, '100111': 0, '000010': 0, '111111': 0, '101010': int(2**13/2), '111011': 0, '111010': 0, '000001': 0, '101110': 0, '000011': 0, '100010': 0, '000110': 0, '000101': 0, '101101': 0, '001000': 0, '100000': 0, '011100': 0, '000000': 0, '010100': 0, '010101': 0, '111001': 0, '111000': 0, '011110': 0, '111100': 0, '101011': 0, '110100': 0, '110001': 0}

print("sum of all_counts")
print(sum_all_counts)

print("Sum of all valid_counts")
print(sum_all_valid_counts)

print("theoretical_all_counts")
print(theoretical_all_counts)

print("ind_counts")
print(ind_counts)

print("ind_valid_counts")
print(ind_valid_counts)
```

```python

'''Sum of all runs and calculates similarity measures'''

def shot_count(counts):
    shots = 0

    for key in counts:
        shots = shots + int(counts[key])

    return shots

def pdf(counts, theory):

    probability_difference = theory.copy()

    shots = 0

    for key in counts:
        shots = shots + int(counts[key])

    shots = float(shots)



    for key in probability_difference: # parsed_data is a tuple with the processed dictionary in the 0th position

        if key in counts: # Ensures given key is in both dictionaries

            probability_difference[key] = abs(float(int(counts[key])/(shots) - float(theoretical_all_counts[key])/8192.0)) #Finds difference between theoretical value and measured values

        else:
            pass

    sum = 0.0

    for key in probability_difference:
        sum = sum + probability_difference[key]

    mu = 1 - (sum/2)


    return mu

print('Average of 20 runs in all 64 possible states have a similarity measure of:')
mu_sum_all_counts = pdf(sum_all_counts, theoretical_all_counts )

shots = shot_count(sum_all_counts)

print('This run used ' + str(shots) + ' shots')
print(mu_sum_all_counts)


print('All 8 valid states have a similarity measure of:')
mu_sum_all_valid_counts = pdf(sum_all_valid_counts, theoretical_all_counts)



shots = shot_count(sum_all_valid_counts)

print('This run used ' + str(shots) + ' shots')
print(mu_sum_all_valid_counts)

'''Individual runs'''

mu_ind_counts = []
```

```
mu_ind_counts_shots = []

print('Each run of all 64 states have a similarity measure of:
    ↪ ')

for i in range(len(ind_counts)):
    mu = pdf(ind_counts[i], theoretical_all_counts)
    mu_ind_counts.append(mu)
    mu_ind_counts_shots.append(shot_count(ind_counts[i]))

print(mu_ind_counts)
print(mu_ind_counts_shots)

mu_ind_valid_counts = []
mu_ind_valid_counts_shots = []

print('Each run of all 8 valid states have a similarity
    ↪ measure of:')

for i in range(len(ind_valid_counts)):
    mu = pdf(ind_valid_counts[i], theoretical_all_counts)
    mu_ind_valid_counts.append(mu)
    mu_ind_valid_counts_shots.append(shot_count(
        ↪ ind_valid_counts[i]))


print(mu_ind_valid_counts)
print(mu_ind_valid_counts_shots)
```

```

'''Plots similarity measures as histograms'''

import matplotlib.pyplot as plt
import numpy as np
np.random.seed(1)

n_bins = 3

x = mu_ind_counts
y = mu_ind_valid_counts
z = mus_saved #from physical

fig, ax = plt.subplots()
ax.hist(x, n_bins, color='lightblue', alpha=0.5,label='Without
    ↪  error detection',
         edgecolor='black', linewidth=1.2, width=0.008)
ax.hist(y, n_bins, color='salmon', alpha=0.5, label='With
    ↪ error detection',
        edgecolor='black', linewidth=1.2, width=0.008)
ax.hist(z, n_bins, color='yellow', alpha=0.5, label='Unencoded
    ↪ ',
        edgecolor='black', linewidth=1.2, width=0.008)


ax.set(title='Similarity Measure of 20 runs Bit Flip Error
    ↪ Correction', ylabel='Number of runs with mu')
ax.legend()
ax.margins(0.05)
ax.set_ylim(bottom=0)
plt.savefig('Similarity Measure of 20 runs Bit Flip Error
    ↪ Correction.png')
plt.show()
```

```

'''Plots probability distribution function as histogram'''

from qiskit.tools.visualization import plot_histogram

plot_histogram(sum_all_valid_counts,
               title=str(len(JobsRan)) + ' runs ' + JobSearch
                  ↪ + ' on ' + str(backend), figsize=(7,7)).
                  ↪ savefig('Logical ' + str(len(JobsRan)) + '
                  ↪  runs ' + JobSearch + ' on ' + str(backend
                  ↪ ))
plot_histogram(sum_all_valid_counts,
               title=str(len(JobsRan)) + ' runs ' + JobSearch
                  ↪ + ' on ' + str(backend), figsize=(7,7))
```

Bibliography

1. S. McCartney, *Eniac: The Triumphs and Tragedies of the World's First Computer.* Berkley Publishing Group, 2001.

2. M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information.* Cambridge University Press, 2019.

3. M. Kliesch, T. Barthel, C. Gogolin, M. Kastoryano, and J. Eisert, "Dissipative quantum church-turing theorem," *Phys. Rev. Lett.*, vol. 107, p. 120501, Sep 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.107.120501

4. E. h. Shaik and N. Rangaswamy, "Implementation of quantum gates based logic circuits using ibm qiskit." *2020 5th International Conference on Computing, Communication and Security (ICCCS), Computing, Communication and Security (ICCCS), 2020 5th International Conference on*, pp. 1 – 6, 2020. [Online]. Available: https://afit.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.9277010&site=eds-live

5. R. P. Feynman, "Quantum mechanical computers," *Optics News*, vol. 11, no. 2, pp. 11–20, Feb 1985. [Online]. Available: http://www.osa-opn.org/abstract.cfm?URI=on-11-2-11

6. M. Grassl, T. Beth, and T. Pellizzari, "Codes for the quantum erasure channel." 1996. [Online]. Available: https://afit.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edsarx&AN=edsarx.quant-ph%2f9610042&site=eds-live

7. R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

8. S. J. Devitt, W. J. Munro, and K. Nemoto, "Quantum error correction for beginners." *Reports on Progress in Physics*, vol. 76, no. 7, pp. 1 – 35, 2013. [Online]. Available: https://afit.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=asn&AN=90437493&site=eds-live

9. E. Knill, "Quantum computing with realistically noisy devices." *Nature*, vol. 434, no. 7029, pp. 39 – 44, 2005. [Online]. Available: https://afit.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=asn&AN=16272453&site=eds-live

10. C. E. Mackenzie, *Coded character sets, history and development.* Addison-Wesley, 1980.

11. M. Hilbert and P. Lopez, "The worlds technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, p. 60–65, 2011.

12. A. Barenco, C. H. Bennett, R. Cleve, D. P. Divincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Physical Review A*, vol. 52, no. 5, p. 3457–3467, 1995.

13. J. Koch, T. M. Yu, J. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, "Charge-insensitive qubit design derived from the cooper pair box," *Phys. Rev. A*, vol. 76, p. 042319, Oct 2007. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.76.042319

14. J. M. Chow, J. M. Gambetta, A. D. Córcoles, S. T. Merkel, J. A. Smolin, C. Rigetti, S. Poletto, G. A. Keefe, M. B. Rothwell, J. R. Rozen, M. B. Ketchen, and M. Steffen, "Universal quantum gate set approaching fault-tolerant thresholds with superconducting qubits," *Phys.*

Rev. Lett., vol. 109, p. 060501, Aug 2012. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.109.060501

15. A. Peres, "Reversible logic and quantum computers," *Phys. Rev. A*, vol. 32, pp. 3266–3276, Dec 1985. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.32.3266

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 21–03–2021	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2019 — Mar 2021

4. TITLE AND SUBTITLE Quantum Computing Using Error Detection	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Hanks, Simeon R., Capt, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering an Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENP-MS-21-M-120
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory 525 Brooks Road Rome, NY 13441	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RITQ
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Quantum computers need to be able to control highly entangled quantum states in the presence of environmental perturbations that lead to errors in calculations. Progress in superconducting qubits has enabled the development of computers capable of running small quantum circuits. The current era of Noise Intermediate Scale Quantum computing has a high error rate. To alleviate this error rate we apply an encoding scheme that allows us to remove results with known errors improving the quality of our results. The encoding uses multiple qubits as a single logical qubit and balances the natural tendency of state-of-the-art quantum computers to decohere towards the ground state. We use a mix of ones and zeroes in each logical qubit in such a way that we can identify and remove results that have violated our specified encoding pattern. The statistical performance of the circuits is improved by retaining the shots that maintained the encoding. Bit flip error detection is applied to the Toffoli gate and produces improved probability distribution functions as well as enhanced similarity measures when compared to its unencoded equivalent.

15. SUBJECT TERMS

Quantum Computing, Error Correction, Error Detection

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Weeks David, AFIT/ENP
U	U	U	UU	55	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636 x4561; david.weeks@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18