NUMERICAL SIMULATION OF ONE WAY QUANTUM COMPUTATION WITH ERROR CORRECTION

JOHN CHILDREN

UNIVERSITY OF LEEDS

MSc Project Report

September 2014 – version 1.0

School of Physics and Astronomy
FACULTY OF MATHEMATICS AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Declaration of Academic Integrity

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.

I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Full Name ....JOHN CHILDREN........

Student Number ....200824907........

Signed ....[signature]........

Date ....12/9/14........

Please Note
When you become a registered student of the University at first and any subsequent registration you sign the following authorisation and declaration:
"I confirm that the information I have given on this form is correct. I agree to observe the provisions of the University's Charter, Statutes, Ordinances, Regulations and Codes of Practice for the time being in force. I know that it is my responsibility to be aware of their contents and that I can read them on the University web site http://www.leeds.ac.uk/studenthandbook. I acknowledge my obligation under the Payment of Fees Section in the Handbook to pay all charges to the University on demand.

I agree to the University processing my personal data (including sensitive data) in accordance with its Code of Practice on Data Protection http://www.leeds.ac.uk/dpa. I consent to the University making available to third parties (who may be based outside the European Economic Area see footnote one) any of my work in any form for standards and monitoring purposes including verifying the absence of plagiarised material. I agree that third parties may retain copies of my work for these purposes on the understanding that the third party will not disclose my identity."

Updated November 2012

ABSTRACT

We investigate one way quantum computation and an associated error correction scheme in order to build a numerical Monte Carlo model for fault tolerance for the model. Though the model is incomplete, we draw conclusions about difficulties in construction of the model and advise improvements that could be made to simulate the system accurately.

Word count: 11255

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION AND BACKGROUND

The very first step in development of a numerical simulation for a one way quantum computer, is to define terms so that the accuracy of the model can be checked. To this aim, we first establish a definition of cluster states as well as the features of a measurement based quantum computer so that we can examine the functionality of the one way quantum computer.

1.1 CLUSTER STATES

The main phenomenon enabling measurement based quantum computation is through the use of cluster states [3]. These states are formed of N qubits interacting with each other in arrays with a high ’persistency’ of entanglement [4]. Persistency is defined in ’Persistent entanglement in arrays of interacting particles’ as the minimum number of local measurements such that the state is completely disentangled for all measurement outcomes. This property means that measurements can be made on individual qubits in the state that will not entirely disentangle the state [4], allowing information to be passed from one qubit to another. Additionally, the states are also said to be ’maximally connected’ if they measurements on qubits in a set can project two seperate qubits into a pure Bell state [4]. This has the advantage of allowing easily teleportable states be produced through measurements on these cluster states.

These states are generally formed in either optical lattices [5] or from photons [6, 7], though other implementations are possible [3].

1.2 MEASUREMENT BASED QUANTUM COMPUTATION

The term ’measurement based quantum computer’ refers to a whole class of quantum computer architectures that use measurements instead of unitary operators to process information [8]. Unlike some other ’alternative’ methods of quantum computation, such as quantum annealing devices, Measurement based quantum computers are both universal and do not suffer from as greatly from problems with decoherence as qubits are discarded after measurement [8]. However, different challenges arise from the difficulty in forming the required states and then measuring specific qubits [3].

In the specific case of the one way quantum computer, using the two properties of cluster states mentioned earlier, it is possible to perform measurements on a ’lattice’ of entangled particles in order

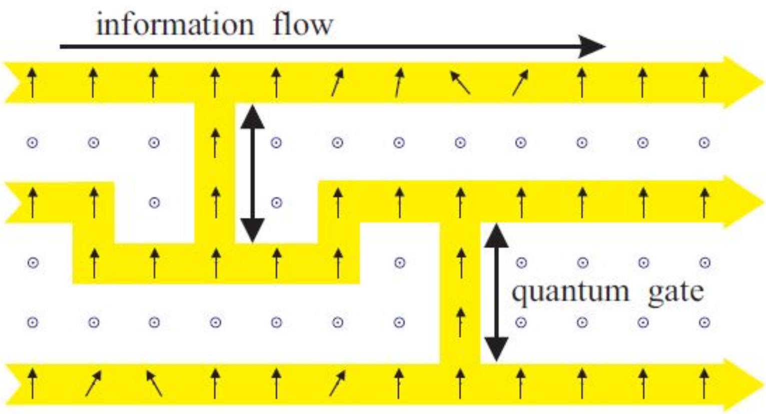

Figure 1: Sketch of information flow in one way computation [1]

to form a kind of quantum circuit that processes information [9] and this is illustrated in figure 1. The exact functionality of these circuits will be shown in the next section.

2 ONE WAY QUANTUM COMPUTATION

In order to demonstrate how numerical simulation of one way quantum computation might be achieved, the first step is to examine the operation of the one way quantum computation. This involves examination of how cluster states can be formed as well as the function performed by measurement so that these procedures can be included in the simulation. In this regard, the procedures for functionality are detailed here so they might be compared to the program in later section to verify correct functionality.

2.1 HAMILTONIAN ON INTERACTING PARTICLES

Following the work in 'Persistant Entanglement In Arrays of Interacting Particles', the Hamiltonian for a d-dimensional lattice at sites $a \in \mathbb{Z}^d$ interacting through short range interaction is [4]:

$$\hat{H}_{int} = \hbar g(t) \sum_{a,a'} f(a-a') \frac{1-\sigma_z^a}{2} \frac{1-\sigma_z^{a'}}{2} \tag{1}$$

where:

$$f(a-a') - \text{interaction range}$$
$$g(t) - \text{time dependence of interaction}$$
$$\sigma_z - \text{pauli-z operation}$$

From the time dependent Schrödinger equation we extract the time evolution of a wavefunction:

$$i\hbar \frac{\partial}{\partial t} |\Psi\rangle = \hat{H} |\Psi\rangle \tag{2}$$

$$|\Psi(t)\rangle = e^{-\frac{i\hat{H}t}{\hbar}} |\Psi(0)\rangle \tag{3}$$

From which we extract the time dependence operator:

$$\hat{U}(t) = e^{-\frac{i\hat{H}t}{\hbar}} \tag{4}$$

Substituting the interaction Hamiltonian (1) into the time dependence operator(4) gives:

$$\hat{U}(t) = exp\left(-ig(t)t \sum_{a,a'} f(a-a') \frac{1-\sigma_z^a}{2} \frac{1-\sigma_z^{a'}}{2}\right) \tag{5}$$

Considering a one-dimensional chain of N-qubits with only next neighbour interactions the interaction range can be expressed as:

$$f(a-a') = \delta_{a+1,a} \tag{6}$$

Which will prevent qubits interacting with themselves and allow nearest neighbour interactions. Now we introduce a term ϕ which represents the integration of the time dependence of interaction such that:

$$\phi = \int g(t)\,dt = Cg(t)t + D \tag{7}$$

Where C and D are constants. Therefore the time evolution operator becomes

$$\hat{U}(t) = exp\left(-i\phi\sum_a \frac{1-\sigma_z^a}{2}\frac{1-\sigma_z^{a+1}}{2}\right) \tag{8}$$

2.2 TWO QUBIT CLUSTER STATE

We can further refine this operator for the interaction between two qubits through the use of Euler's relation for operators:

$$e^{i\theta\hat{A}} = \cos(\theta)\hat{\mathbb{1}} + i\sin(\theta)\hat{A} \tag{9}$$

Now,we can expand the equation (8) as there are only two qubits to give:

$$\hat{U}(t) = exp\left(-\frac{i\phi}{4}\left(\mathbb{1}\otimes\mathbb{1} - \mathbb{1}\otimes\sigma_z - \sigma_z\otimes\mathbb{1} + \sigma_z\otimes\sigma_z\right)\right) \tag{10}$$

Typically we would need to apply the Baker-Campbell-Hausdorff formula to convert these terms into something more manageable, however the operators $\mathbb{1}$ and σ_z are commutative, as are their tensor products, so all but the first two terms of the Baker-Campbell-Hausdorff expansion can be neglected leaving:

$$\hat{U}(t) = exp(-\frac{i\phi}{4}\mathbb{1}\otimes\mathbb{1})exp(\frac{i\phi}{4}\sigma_z\otimes\mathbb{1})exp(\frac{i\phi}{4}\sigma_z\otimes\mathbb{1})exp(-\frac{i\phi}{4}\sigma_z\otimes\sigma_z) \tag{11}$$

Applying equation (9):

$$\begin{aligned}
&\hat{U}(t) = exp(-\frac{i\phi \mathbb{1}}{4}) \\
&\left(\cos(\frac{i\phi}{4})\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{4})\mathbb{1}\otimes\sigma_z\right) \\
&\left(\cos(\frac{i\phi}{4})\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{4})\sigma_z\otimes\mathbb{1}\right) \\
&\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\sigma_z\otimes\sigma_z\right)
\end{aligned} \tag{12}$$

In the case of $\phi = \pi$ this becomes:

$$\begin{aligned}
&\hat{U}(t) = \frac{1}{2\sqrt{2}}exp(-\frac{i\pi}{4})\mathbb{1} \\
&(\mathbb{1}\otimes\mathbb{1} + i\mathbb{1}\otimes\sigma_z) \\
&(\mathbb{1}\otimes\mathbb{1} + i\sigma_z\otimes\mathbb{1}) \\
&(\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z)
\end{aligned} \tag{13}$$

Now, expanding brackets and simplifying in two steps:

$$\begin{aligned}
&\hat{U}(t) = \frac{1}{2\sqrt{2}}exp(-\frac{i\pi}{4})\mathbb{1} \\
&(\mathbb{1}\otimes\mathbb{1} - \sigma_z\otimes\sigma_z + i\mathbb{1}\otimes\sigma_z + i\sigma_z\otimes\mathbb{1}) \\
&(\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z) \\
\\
&= \frac{1}{2\sqrt{2}}exp(-\frac{i\pi}{4}) \\
&(\mathbb{1}\otimes\mathbb{1} - \sigma_z\otimes\sigma_z + i\mathbb{1}\otimes\sigma_z + i\sigma_z\otimes\mathbb{1} \\
&+ i\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes \ + \mathbb{1}\otimes\sigma_z + \sigma_z\otimes\mathbb{1})
\end{aligned}$$

Factorising real and imaginary terms:

$$\begin{aligned}
&\hat{U}(t) = \frac{1}{2}exp(-\frac{i\pi}{4})\frac{1+i}{\sqrt{2}} \\
&(\mathbb{1}\otimes\mathbb{1} - \sigma_z\otimes\sigma_z + \mathbb{1}\otimes\sigma_z + \sigma_z\otimes\mathbb{1})
\end{aligned} \tag{14}$$

As $exp(-\frac{i\pi}{4}) = \frac{1-i}{\sqrt{2}}$ equation (14) simplifies to:

$$\hat{U}(t) = \frac{1}{2}\left(\mathbb{1}\otimes\mathbb{1} + \mathbb{1}\otimes\sigma_z + \sigma_z\otimes\mathbb{1} - \sigma_z\otimes\sigma_z\right) \tag{15}$$

If this operator is applied to two qubits in the $|{+}{+}\rangle$ state, for example, we get:

$$\hat{U}(t)\,|{+}{+}\rangle = \frac{1}{2}(|{+}{+}\rangle + |{+}{-}\rangle + |{-}{+}\rangle - |{-}{-}\rangle)$$
$$= \frac{1}{\sqrt{2}}(|{+}0\rangle + |{-}1\rangle) \tag{16}$$

2.3 THREE QUBIT CLUSTER STATE

By using the same method used in the previous section, we can also obtain a the time evolution operator for a chain of three qubits only interacting with their nearest neighbour. In this case the time evolution operator will be

$$\hat{U}(t) = exp\left(-i\phi\left(\frac{1-\sigma_z^1}{2}\frac{1-\sigma_z^2}{2}\mathbb{1} + \mathbb{1}\frac{1-\sigma_z^2}{2}\frac{1-\sigma_z^3}{2}\right)\right) \tag{17}$$

Expanding out this expression gives:

$$\begin{aligned}\hat{U}(t) &= exp(-\frac{i\phi}{4}(2\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - 2\mathbb{1}\otimes\sigma_z\otimes\mathbb{1}\\ &- \sigma_z\otimes\mathbb{1}\otimes\mathbb{1} - \mathbb{1}\otimes\mathbb{1}\otimes\sigma_z\\ &+ \sigma_z\otimes\sigma_z\otimes\mathbb{1} + \mathbb{1}\otimes\sigma_z\otimes\sigma_z))\end{aligned} \tag{18}$$

Applying Baker-Campbell-Hausdorff and Euler's rule:

$$\hat{U}(t) = exp(-\frac{i\phi}{2})$$
$$\left(\cos(\frac{i\phi}{2})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{2})\mathbb{1}\otimes\sigma_z\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{4})\sigma_z\otimes\mathbb{1}\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z\right)$$
$$\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\sigma_z\otimes\sigma_z\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\mathbb{1}\otimes\sigma_z\otimes\sigma_z\right)$$

Once again, using $\phi = \pi$, we expand the terms of the equation and simplify

$$\hat{U}(t) = exp(-\frac{i\pi}{2})(i\mathbb{1}\otimes\sigma_z\otimes\mathbb{1})$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sigma_z\otimes\mathbb{1}\otimes\mathbb{1})(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z)$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z\otimes\mathbb{1})(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z)$$

$$= exp(-\frac{i\pi}{2})(i\mathbb{1}\otimes\sigma_z\otimes\mathbb{1})$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sigma_z\otimes\mathbb{1}\otimes\mathbb{1} + i\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z - \sigma_z\otimes\mathbb{1}\otimes\sigma_z)$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z\otimes\mathbb{1} - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z + \sigma_z\otimes\mathbb{1}\otimes\sigma_z)$$

$$= exp(-\frac{i\pi}{2})(i\mathbb{1}\otimes\sigma_z\otimes\mathbb{1})$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z - i\sigma_z\otimes\sigma_z\otimes\mathbb{1} - \sigma_z\otimes\mathbb{1}\otimes\sigma_z + i\sigma_z\otimes\mathbb{1}\otimes\mathbb{1}$$
$$+ \sigma_z\otimes\sigma_z\otimes\sigma_z + \mathbb{1}\otimes\sigma_z\otimes\mathbb{1} - i\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z + i\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z$$
$$+ \mathbb{1}\otimes\sigma_z\otimes\mathbb{1} + \sigma_z\otimes\sigma_z\otimes\sigma_z - i\sigma_z\otimes\sigma_z\otimes\mathbb{1} - \sigma_z\otimes\mathbb{1}\otimes\sigma_z$$
$$+ i\sigma_z\otimes\sigma_z\otimes\mathbb{1} + i\mathbb{1}\otimes\sigma_z\otimes\sigma_z + \sigma_z\otimes\sigma_z\otimes\sigma_z)$$

Many of these terms will cancel and as $\frac{1}{2}exp(-\frac{i\pi}{2}) = \frac{i}{2}$ we are left with:

$$\hat{U} = \frac{1}{2}(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + \mathbb{1}\otimes\sigma_z\otimes\mathbb{1} + \sigma_z\otimes\mathbb{1}\otimes\sigma_z - \sigma_z\otimes\sigma_z\otimes\sigma_z) \quad (19)$$

Applying this to the state $|+++\rangle$:

$$\hat{U}\,|+++\rangle = \frac{1}{\sqrt{2}}(|+0+\rangle + |-1-\rangle) \quad (20)$$

However, we could have quite simply achieved the result from equation (19) by applying equation (15) twice to the three qubits. This is due to the nature of the interaction Hamiltonian being used and as such we can consider all links between two qubits to have this same property and as such we can construct time evolution operators for any system using this operator as a building block. This can be demonstrated by applying the unitary operator (15) to the state described in (16) and an addition $|+\rangle$ state.

$$
\begin{aligned}
&\frac{1}{2}\left(\mathbb{1}_2 \otimes \mathbb{1}_3 + \mathbb{1}_2 \otimes \sigma_{z3} + \sigma_{z2} \otimes \mathbb{1}_3 - \sigma_{z2} \otimes \sigma_{z3}\right) \frac{1}{\sqrt{2}}(|{+0}\rangle + |{-1}\rangle)\,|+\rangle \\
&= \frac{1}{2\sqrt{2}}(|+\rangle\,(|0+\rangle + |0-\rangle + |0+\rangle - |0-\rangle) \\
&+ |-\rangle\,(|1-\rangle + |1+\rangle - |1-\rangle + |1+\rangle)) \\
&= \frac{1}{\sqrt{2}}(|{+0+}\rangle + |{-1-}\rangle)
\end{aligned}
\tag{21}
$$

Hence CZ operators can be applied in sequence to form cluster states of any desired size or topology. This will be utilised later in the programming stage when creating subroutines to handle the initialisation of the system.

2.4 QUANTUM CONTROLLED Z OPERATOR

Consider a 2 qubit state prepared as $|++\rangle$ and subject to the unitary transform (15) for two qubit cluster states:

$$
\begin{aligned}
&\hat{U}\,|++\rangle = \frac{1}{\sqrt{2}}(|{+0}\rangle + |{-1}\rangle) \\
&= \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) \\
&= \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}
\end{aligned}
$$

As the matrix expression for $|++\rangle$ is:

$$
|++\rangle = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{22}
$$

So the transformation matrix between the two is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{23}$$

Which can be expressed as a unitary transform that forms a conditional phase gate between qubits a and b.

$$S^{ab} = |0\rangle_a \langle 0| \otimes \mathbb{1}^b + |1\rangle_a \langle 1| \otimes \sigma_z^b \tag{24}$$

Therefore the unitary operator is equivalent to the CZ operation in the circuit model. In general we can perform a controlled phase operation in the z axis in a similar form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-i\alpha} \end{pmatrix} \tag{25}$$

Where α is the phase of the operation.

2.5 MEASUREMENT BASED GATE OPERATION

The basic principle of computation in the one way quantum computer follows three simple steps. The first step involves the formation of a cluster state through the use of the CZ operation demonstrated earlier. The next step involves measurement of the state of specific qubits in a particular measurement basis depending upon the computational operation required. Finally, the outcomes of measurement are fed forward in order to determine the result of computation when the output is measured. Therefore we shall next examine the types of measurements required to achieve quantum computation through this method.

2.5.1 *One bit teleportation*

The most simple kind of operation that can be performed in one way quantum computation is through the use of teleportation to transfer the state of one qubit so that the information of the state can be projected to the other through applying measurements with a particular basis to the first qubit. To demonstrate this in general we can consider a qubit in unknown state $|\psi\rangle_1 = a|0\rangle_1 + b|1\rangle_1$ and second qubit in state $|+\rangle$.

$$|\psi\rangle_1 \otimes |+\rangle_2 = (a|0\rangle_1 + b|1\rangle_1)\frac{1}{\sqrt{2}}(|0\rangle_1 + |1\rangle_1) \tag{26}$$

A CZ operation is applied to qubit 1 and 2 such that they become entangled similarly to the two qubit + state in (16).

$$\begin{aligned} &CZ_{12}|\psi\rangle_{12} = \frac{1}{\sqrt{2}}(a|00\rangle_{12} + a|01\rangle_{12} + |10\rangle_{12} - b|11\rangle_{12}) \\ &= (a|0+\rangle_{12} + b|1-\rangle_{12}) \end{aligned} \tag{27}$$

The first qubit is then measured in a basis corresponding to it's phase angle.

$$\hat{M} = \{\ |+\psi\rangle\langle+\psi|\ ,\ |-\psi\rangle\langle-\psi|\ \} \tag{28}$$

There are two possible measurement outcomes $|+\psi\rangle_1$ and $|-\psi\rangle_2$ that can result here. In the first case of $|+\psi\rangle_1$.

$$\begin{aligned} &|+\psi\rangle_1 \frac{1}{\sqrt{2}}(\langle 0|_1 + e^{-i\psi}\langle 1|_1)(a|0+\rangle_{12} + b|1-\rangle_{12}) \\ &= |+\psi\rangle_1 [\langle 0|0\rangle_1 a|+\rangle_2 + e^{-i\psi}\langle 1|1\rangle_1 b|-\rangle_2] \\ &= \frac{1}{\sqrt{2}}|+\psi\rangle_1 (a|+\rangle_2 + e^{-i\psi} b|-\rangle_2) \end{aligned} \tag{29}$$

The information contained in the phase of the original qubit can then be obtained through the application of a series of operations which will project the information state onto the second qubit.

$$\begin{aligned} &|out^0\rangle_2 = a|+\rangle_2 + be^{-i\psi}|-\rangle_2 \\ &= H_2(a|0\rangle_2 + be^{-i\psi}|1\rangle_2) \\ &= e^{\frac{-i\psi}{2}} HR_z(-\psi)(a|0\rangle_2 + b|1\rangle_2) \end{aligned} \tag{30}$$

However, when the opposite measurement outcome occurs, the state will be projected such that an X operation is also required.

$$\begin{aligned} &|out^1\rangle_2 = X_2(a|+\rangle + be^{-i\psi}|-\rangle) \\ &= X_2|out^0\rangle_2 \end{aligned} \tag{31}$$

Thus in general we can recover the information projected onto one qubit through the measurement of two qubits entangled by a CZ operation through the following operations:

$$|out^{m_i}\rangle_2 = e^{\frac{-i\psi}{2}} X^{m_i} H R_z(-\psi)(a\,|0\rangle_2 + b\,|1\rangle_2) \tag{32}$$

Where m_i represents the number of the basis used in measurement.

2.5.2 *Controlled not gate*

In order to form a CNOT gate however, at least two qubits are required so the generalised rotation shown in the previous section is not sufficient. Therefore a new geometry is required which forms a T-shape consisting of four qubits. On the left end of longer side of the shape we have our target qubit in a particular state $|\psi_T\rangle$ and on the bottom of the shape is the control qubit $|\phi_c\rangle$. The relationship between the states of these two qubits is such that the intended output will be:

$$\begin{aligned} &CNOT_{14}\,|\psi_T\rangle\,|\phi_C\rangle = CNOT_{14}(a\,|0\rangle_1 + b\,|1\rangle_1)(c\,|0\rangle_4 + d\,|1\rangle_4) \\ &= ac\,|00\rangle_{14} + bd\,|01\rangle_{14} + bc\,|10\rangle + ad\,|11\rangle_{14} \end{aligned} \tag{33}$$

In order to prepare the qubits for measurement we prepare a state $|\psi^{in}\rangle$ that denotes the input state of the qubits formed of multiple CZ operations in order to create our T-shape of entanglement. By applying CZ operations to the known initial states of our four qubits we can determine how this state is represented to find the results of measurements.

$$\begin{aligned} &|\psi^{in}\rangle = CZ_{12}CZ_{23}CZ_{24}\,|\psi_T\rangle_1\,|+\rangle_2\,|+\rangle_3\,|\phi_4\rangle \\ &= CZ_{21}CZ_{32}CZ_{24}\,|\psi_T\rangle_1\,|+\rangle_2\,|+\rangle_3\,|\phi_4\rangle \\ &= \frac{1}{2}(|\psi_T\rangle_1\,(|0\rangle_2 + |1\rangle_2\,Z_1)(|0\rangle_3 + |1\rangle_3\,Z_2)(c\,|0\rangle_3 + d\,|1\rangle_3\,Z_3) \\ &= \frac{1}{2}((|\psi_T\rangle_1\,|0\rangle_2 + |\psi_T\rangle_1\,|1\rangle_2\,Z_1) \\ &(|0\rangle_3 + |1\rangle_3\,Z_2)(c\,|0\rangle_3 + d\,|1\rangle_3\,Z_3) \\ &= \frac{1}{2}(|\psi_T\rangle_1\,|00\rangle_{23} + Z_1\,|\psi_T\rangle_1\,|01\rangle_{23} \\ &- (Z_2\,|\psi_T\rangle_1\,|10\rangle_{23} - Z_1\,|\psi_T\rangle_1\,|11\rangle_{23})(c\,|0\rangle_4 + d\,|1\rangle_4\,Z_3) \\ &= \frac{1}{\sqrt{2}}(|\psi_T\rangle_1\,|0\rangle_2\,|+\rangle_3\,|\phi_c\rangle_4 + (Z_1\,|\psi_T\rangle_1)\,|1\rangle_2\,|-\rangle_3\,(Z_3\,|\phi_c\rangle_4) \end{aligned} \tag{34}$$

In order to perform a CNOT operation using this state between the target and control qubits we then measure qubits 1 and 2 in the basis $\{ \ |+\rangle, \ |-\rangle \ \}$. In the case of $|+\rangle$ measurement this results in:

$$
{}_1\langle +|\psi^{in}\rangle_{1234} = \frac{1}{\sqrt{2}}((a+b)\,|0\rangle_2\,|+\rangle_3\,|\phi_c\rangle_4 \\
+ (a-b)\,|1\rangle_2\,|-\rangle_4\,(Z_4\,|\phi_c\rangle_4)) \quad (35)
$$

This state then becomes:

$$
(|+\rangle \otimes |+\rangle)\,\langle\psi^{in}|_{1234}\, ac\,|00\rangle_{34} + bd\,|01\rangle_{34} + bc\,|10\rangle_{34} + ad\,|11\rangle_{34} \\
= |out\rangle_{34} \quad (36)
$$

Which from equation (33) we can see is in fact the desired CNOT with some additional operators applied. Therefore in order to recover correct output we feed forward measurement outputs for the first two qubits to obtain the output:

$$
|out^{m_1 m_2}\rangle = X_T^{n_2} Z_T^{n_1} X_C^{n_1} CNOT\,|\psi_T\rangle_3\,|\phi_C\rangle_4 \quad (37)
$$

2.5.3 *Gate universality*

The two gate operations here can be used to form a universal set of gates [10]. This set can be expressed analogously to the more commonly used universal set of five gates in the circuit model, which is demonstrated in [1] and shown in Figure 2. Therefore, if we can simulate these two gates as described here we can simulate any possible measurement based gate.

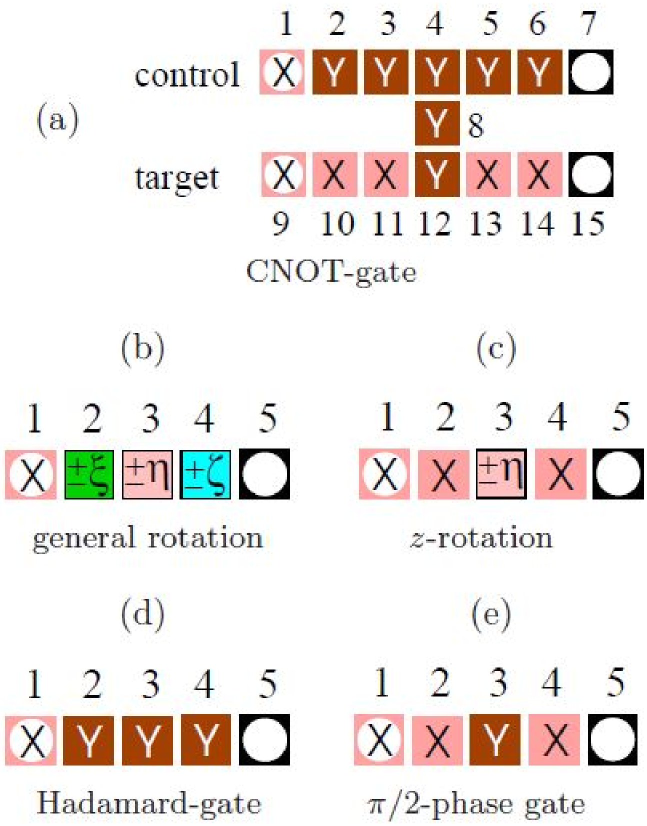

Figure 2: Universal set of quantum gates [1]

3 ERROR CORRECTION

The next part of the simulation will be the error correction scheme. In order to keep the program initially as simple as possible a variant of Jaewoo Joo's measurement based error correction scheme using two auxilary qubits to create a 'triangle state' instead of the 'pentagon state's used in Joo's paper [2]. This error correction scheme is convenient as it can be extended to larger size logical qubits easily so the correlation between fault tolerance and qubit number can be examined.

3.1 LOGICAL QUBITS

The first step towards simulation of the error correction scheme for measurement based quantum computation requires the definition of a logical state which represents a quantum state for which the information is distributed amongst multiple qubits. Following the work in 'Error - correcting one - way quantum computation with global entangling gates' [] we establish a logical state based on three qubits, rather than the five in the paper, to form a triangle state through three CZ operations. In doing so we obtain the unitary operator that will allow for conversion of three qubits in the + state to the logical + state.

$$\hat{U}(t) = exp\Bigg(- i\phi\Bigg(\frac{1-\sigma_z^1}{2}\frac{1-\sigma_z^2}{2}\mathbb{1} \\ + \frac{1-\sigma_z^1}{2}\mathbb{1}\frac{1-\sigma_z^3}{2} + \mathbb{1}\frac{1-\sigma_z^2}{2}\frac{1-\sigma_z^3}{2}\Bigg)\Bigg) \tag{38}$$

Expanding out this expression gives:

$$\hat{U}(t) = exp(-\frac{i\phi}{4}(3\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - 2\mathbb{1}\otimes\sigma_z\otimes\mathbb{1} \\ - 2\sigma_z\otimes\mathbb{1}\otimes\mathbb{1} - 2\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z \\ + \sigma_z\otimes\sigma_z\otimes\mathbb{1} + \mathbb{1}\otimes\sigma_z\otimes\sigma_z \\ + \sigma_z\otimes\mathbb{1}\otimes\sigma_z)) \tag{39}$$

Applying Baker-Campbell-Hausdorff and Euler's rule:

$$\hat{U}(t) = exp(-\frac{-3i\phi}{4})$$
$$\left(\cos(\frac{i\phi}{2})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{2})\mathbb{1}\otimes\sigma_z\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{i\phi}{2})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{2})\sigma_z\otimes\mathbb{1}\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{i\phi}{2})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{i\phi}{2})\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z\right)$$
$$\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\sigma_z\otimes\sigma_z\otimes\mathbb{1}\right)$$
$$\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\mathbb{1}\otimes\sigma_z\otimes\sigma_z\right)$$
$$\left(\cos(\frac{-i\phi}{4})\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\sin(\frac{-i\phi}{4})\sigma_z\otimes\mathbb{1}\otimes\sigma_z\right)$$

Using $\phi = \pi$, we expand the terms of the equation and simplify

$$\hat{U}(t) = \frac{1}{2\sqrt{2}}exp(-\frac{-3i\pi}{4})(i\mathbb{1}\otimes\sigma_z\otimes\mathbb{1})$$
$$(i\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z)$$
$$(i\sigma_z\otimes\mathbb{1}\otimes\mathbb{1})$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z)(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\mathbb{1}\otimes\sigma_z)$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z\otimes\mathbb{1})$$

$$= \frac{1}{2\sqrt{2}}exp(-\frac{-3i\pi}{4})(i\sigma_z\otimes\sigma_z\otimes\sigma_z)$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\mathbb{1}\otimes\sigma_z - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z - \sigma_z\otimes\sigma_z\otimes\mathbb{1})$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z\otimes\mathbb{1})$$

$$= \frac{1}{2\sqrt{2}}exp(-\frac{-3i\pi}{4})(i\sigma_z\otimes\sigma_z\otimes\sigma_z)$$
$$(\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1} + i\mathbb{1}\otimes\mathbb{1}\otimes\mathbb{1}$$
$$-\sigma_z\otimes\sigma_z\otimes\mathbb{1} - i\sigma_z\otimes\sigma_z\otimes\mathbb{1}$$
$$-\sigma_z\otimes\mathbb{1}\otimes\sigma_z - i\sigma_z\otimes\mathbb{1}\otimes\sigma_z$$
$$-\mathbb{1}\otimes\sigma_z\otimes\sigma_z - i\mathbb{1}\otimes\sigma_z\otimes\sigma_z$$

$$= \frac{1-i}{2\sqrt{2}}exp(-\frac{-3i\pi}{4})(\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z + \sigma_z\otimes\mathbb{1}\otimes\mathbb{1}$$
$$+\mathbb{1}\otimes\sigma_z\otimes\mathbb{1} - \sigma_z\otimes\sigma_z\otimes\sigma_z)$$

As $exp(-\frac{-3i\pi}{4}) = -\frac{1+i}{\sqrt{2}}$ we are left with:

$$\hat{U}\,|+++\rangle = \frac{1}{2}(\mathbb{1}\otimes\mathbb{1}\otimes\sigma_z + \sigma_z\otimes\mathbb{1}\otimes\mathbb{1} + \mathbb{1}\otimes\sigma_z\otimes\mathbb{1} - \sigma_z\otimes\sigma_z\otimes\sigma_z) \quad (40)$$

Applying this to the state $|+++\rangle$:

$$\hat{U}\,|+++\rangle = \frac{1}{2}(|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle) \quad (41)$$

This allows us to find our logical plus state by applying the operator to three qubits in the plus state to form our 'triangle state'.

$$|+^L\rangle = CZ_{12}CZ_{23}CZ_{13}\,|+++\rangle \quad (42)$$

Applying the CZ operators in sequence:

$$|+^L\rangle = \frac{1}{2}CZ_{23}CZ_{13}(|+++\rangle + |+-+\rangle + |-++\rangle - |--+\rangle)$$

$$= \frac{1}{2}CZ_{23}CZ_{13}(|+\rangle\,(|++\rangle + |-+\rangle) + |-\rangle\,(|++\rangle - |-+\rangle))$$

$$= \frac{1}{4}CZ_{13}(|+\rangle\,(|++\rangle + |+-\rangle + |-+\rangle - |--\rangle$$
$$+ |-+\rangle + |++\rangle + |--\rangle - |+-\rangle)$$
$$+ |-\rangle\,(|++\rangle + |+-\rangle + |-+\rangle - |--\rangle - |-+\rangle$$
$$- |++\rangle - |--\rangle + |+-\rangle))$$

$$= \frac{1}{2}CZ_{13}(|+\rangle\,(|++\rangle + |-+\rangle) + |-\rangle\,(|+-\rangle - |--\rangle))$$

$$= \frac{\sqrt{2}}{2}CZ_{13}(|+0+\rangle + |-1-\rangle)$$

Which is the result from (20) with an additional CZ operation. Continuing by applying the final operator:

$$|+^L\rangle = \frac{\sqrt{2}}{4}(|+0+\rangle + |+0-\rangle + |-0+\rangle - |-0-\rangle$$
$$+ |-1-\rangle + |+1-\rangle + |-1+\rangle - |+1+\rangle)$$

Which simplifies to:

$$|+^L\rangle = \frac{1}{2}(|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle) \tag{43}$$

So states (41) and (43) are identical and applying three CZ operators to form a logical qubits is valid. By a similar method we can find that:

$$|-^L\rangle = \frac{1}{2}(|+++\rangle - |--+\rangle - |-+-\rangle - |+--\rangle) \tag{44}$$

Then, by using the relations $|0^L\rangle = \frac{|+^L\rangle + |-^L\rangle}{\sqrt{2}}$ and $|1^L\rangle = \frac{|+^L\rangle - |-^L\rangle}{\sqrt{2}}$ we can find expressions for $|0^L\rangle$ and $|1^L\rangle$

$$|0^L\rangle = \frac{1}{2}(|0++\rangle - |0--\rangle + |1+-\rangle + |1-+\rangle) \tag{45}$$

$$|1^L\rangle = \frac{1}{2}(|0+-\rangle - |1++\rangle + |1--\rangle + |0-+\rangle) \tag{46}$$

But in order to use these qubits in computation we must first determine the equivalent logical gate operations.

3.2 LOGICAL OPERATIONS

Logical gate operators represent the product of standard one qubit operations that transform one logical state into the other. Now that we have the four primary logical states that will be used for the model, we can determine the logical operations required to transform between them. These logical states will be used to correct errors in the state vector dependant upon the final states of the auxiliary qubits used to detect errors.

3.2.1 *Logical Z operation*

In order to reconstruct the equivalent gate operations, we consider the logical input and output states and then determine the operation required to transform one to the other. Firstly, considering a Z operator we know that:

$$Z|+\rangle = |-\rangle \qquad Z|-\rangle = |+\rangle$$
$$Z|0\rangle = |0\rangle \qquad Z|1\rangle = -|1\rangle$$

So we should expect that:

$$Z^L\left|+^L\right\rangle = \left|-^L\right\rangle \qquad Z^L\left|-^L\right\rangle = \left|+^L\right\rangle$$
$$Z^L\left|0^L\right\rangle = \left|0^L\right\rangle \qquad Z^L\left|1^L\right\rangle = -\left|1^L\right\rangle$$

Examining the logical $+$ and $-$ states it seems that each of the four states that make up each logical state has a partner in the other logical state that is opposite in sign.

$$\left|+^L\right\rangle = \frac{1}{2}(\left|+-+\right\rangle + \left|++-\right\rangle + \left|-++\right\rangle - \left|---\right\rangle)$$
$$\left|-^L\right\rangle = \frac{1}{2}(-\left|-+-\right\rangle - \left|--+\right\rangle - \left|+--\right\rangle + \left|+++\right\rangle)$$

Therefore it seems obvious to attempt to see if a product of three Z operations will transform one logical state into another in order to find the Z logical state.

$$\begin{aligned} Z_1Z_2Z_3\left|+^L\right\rangle &= \frac{1}{2}(\left|-+-\right\rangle + \left|--+\right\rangle + \left|+--\right\rangle - \left|+++\right\rangle) \\ &= -\left|-^L\right\rangle \end{aligned} \tag{47}$$

$$\begin{aligned} Z_1Z_2Z_3\left|-^L\right\rangle &= \frac{1}{2}(-\left|+-+\right\rangle - \left|++-\right\rangle - \left|-++\right\rangle + \left|---\right\rangle) \\ &= -\left|+^L\right\rangle \end{aligned} \tag{48}$$

From equations (47) and (48) it therefore seems likely that the logical Z operation for three qubits is $-Z_1Z_2Z_3$. Testing this further with the logical 0 and 1 states:

$$-Z_1Z_2Z_3\left|0^L\right\rangle = \frac{1}{2}(-\left|0--\right\rangle + \left|0++\right\rangle + \left|1-+\right\rangle + \left|1+-\right\rangle) = \left|0^L\right\rangle \tag{49}$$

$$-Z_1Z_2Z_3\left|1^L\right\rangle = \frac{1}{2}(\left|0-+\right\rangle + \left|1--\right\rangle - \left|1++\right\rangle + \left|0+-\right\rangle)\left|1^L\right\rangle \tag{50}$$

Which further confirms that $Z^L = -Z_1Z_2Z_3$.

3.2.2 *Logical X operation*

Continuing this process for the X operation, we should expect that:

$$X^L|+^L\rangle = |+^L\rangle \qquad X^L|-^L\rangle = -|-^L\rangle$$
$$X^L|0^L\rangle = |1^L\rangle \qquad X^L|1^L\rangle = |0^L\rangle$$

So we can similarly try three single qubit Pauli X operations to determine the equivalent logical operation:

$$X_1X_2X_3|+^L\rangle = \frac{1}{2}(-|+-+\rangle - |++-\rangle - |-++\rangle - |---\rangle) \\ = -|+^L\rangle \tag{51}$$

$$X_1X_2X_3|-^L\rangle = \frac{1}{2}(-|+++\rangle - |--+\rangle - |-+-\rangle + |+--\rangle) \\ = |-^L\rangle \tag{52}$$

So can also conclude that $X^L = -X_1X_2X_3$.

3.2.3 *Logical Hadamard operation*

The Logical Hadamard operation should be such that:

$$H^L|+^L\rangle = |0^L\rangle \qquad H^L|-^L\rangle = -|1^L\rangle$$
$$H^L|0^L\rangle = |+^L\rangle \qquad H^L|1^L\rangle = |-^L\rangle$$

First let us try:

$$H_1|+^L\rangle = \frac{1}{2}(|0-+\rangle + |0+-\rangle + |1++\rangle - |1--\rangle) \tag{53}$$

However as our target is:

$$|0^L\rangle = \frac{1}{2}(|0++\rangle - |0--\rangle + |1+-\rangle + |1-+\rangle)$$

There seems to be a mismatch between the first qubit and the others if we only apply the Hadamard to the first qubit, so we can attempt to rectify this by also applying an X operation.

$$X_1 H_1 \left|+^L\right\rangle = \frac{1}{2}(|1-+\rangle + |1+-\rangle + |0++\rangle - |0--\rangle) = \left|0^L\right\rangle \quad (54)$$

Which is the intended result, but when we try the same set of operations on $\left|-^L\right\rangle$ we find that the operations are not adequate alone.

$$X_1 H_1 \left|-^L\right\rangle = \frac{1}{2}(|1++\rangle - |0-+\rangle - |0+-\rangle - |1--\rangle) = -\left|1^L\right\rangle \quad (55)$$

However, if we apply a Z^L operation to both sides this problem will be rectified.

$$\begin{aligned}
&- Z_1 Z_2 Z_3 X_1 H_1 \left|-^L\right\rangle \\
&= \frac{1}{2}(|1--\rangle + |0+-\rangle + |0-+\rangle - |1++\rangle) \\
&= \left|1^L\right\rangle
\end{aligned} \quad (56)$$

Similarly:

$$\begin{aligned}
&- Z_1 Z_2 Z_3 X_1 H_1 \left|+^L\right\rangle \\
&= \frac{1}{2}(|1+-\rangle + |1-+\rangle - |0--\rangle + |0++\rangle) \\
&= \left|0^L\right\rangle
\end{aligned} \quad (57)$$

Thus the logical Hadamard gate is:

$$H^L = -Z_1 Z_2 Z_3 X_1 H_1 \quad (58)$$

3.2.4 *Logical rotation operation*

Another gate type required for universal computation is the Z rotation operation. Repeating the process used for the other logical operations, we should expect that:

$$R^L(\xi)\left|+^L\right\rangle = \frac{1}{\sqrt{2}}\left(e^{\frac{-i\xi}{2}}\left|0^L\right\rangle + e^{\frac{i\xi}{2}}\left|1^L\right\rangle\right)$$
$$R^L(\xi)\left|-^L\right\rangle = \frac{1}{\sqrt{2}}\left(e^{\frac{-i\xi}{2}}\left|0^L\right\rangle - e^{\frac{i\xi}{2}}\left|1^L\right\rangle\right)$$
$$R^L(\xi)\left|0^L\right\rangle = e^{\frac{-i\xi}{2}}\left|0^L\right\rangle$$
$$R^L(\xi)\left|1^L\right\rangle = e^{\frac{i\xi}{2}}\left|1^L\right\rangle$$

The process of forming this operation is slightly long-winded compared to the others, however the first step is simply to apply a normal z rotation operation to the first qubit of a logical state.

$$R_z(\xi)\left|+^L\right\rangle = \frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(|0-+\rangle + |0+-\rangle + |0++\rangle - |0--\rangle$$
$$+ e^{i\xi}(|1-+\rangle + |1+-\rangle - |1++\rangle + |1--\rangle))$$

Then a Z operation is applied to this first qubit.

$$Z_1R_z(\xi)\left|+^L\right\rangle = \frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(|0-+\rangle + |0+-\rangle + |0++\rangle - |0--\rangle$$
$$- e^{i\xi}(|1-+\rangle + |1+-\rangle - |1++\rangle + |1--\rangle))$$

Next we apply three Hadamard operations, but this requires first some rearrangement:

$$Z_1R_z(\xi)\left|+^L\right\rangle = \frac{1}{2e^{\frac{i\xi}{2}}}(|0-1\rangle + |0+0\rangle - e^{i\xi}(|1-0\rangle - |1+1\rangle))$$

Now applying the operators:

$$H_1H_2H_3Z_1R_z(\xi)\left|+^L\right\rangle = \frac{1}{2e^{\frac{i\xi}{2}}}(|{+}1{-}\rangle + |{+}0{+}\rangle - e^{i\xi}(|{-}1{+}\rangle - |{-}0{-}\rangle))$$

For the next step we apply a CNOT_{12}, which is a controlled X operation between qubits 1 and 2 such that the right hand side of the equation becomes:

$$\frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(|01-\rangle + |10-\rangle + |00+\rangle + |11+\rangle$$
$$- e^{i\xi}(|01+\rangle - |10+\rangle - |00-\rangle + |11-\rangle))$$

Then CNOT_{13} is similarly applied

$$\frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(|01-\rangle - |10-\rangle + |00+\rangle + |11+\rangle$$
$$- e^{i\xi}(|01+\rangle - |10+\rangle - |00-\rangle - |11-\rangle))$$

A Hadamard is then applied again to the first qubit

$$\begin{aligned}&\frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(|+1-\rangle - |-0-\rangle + |+0+\rangle + |-1+\rangle \\ &- e^{i\xi}(|+1+\rangle - |-0+\rangle - |+0-\rangle - |-1-\rangle))\end{aligned} \tag{59}$$

Rearranging the right hand side gives:

$$\frac{1}{4e^{\frac{i\xi}{2}}}(|01-\rangle + |11-\rangle - |00-\rangle + |10-\rangle$$
$$+ |00+\rangle + |10+\rangle + |01+\rangle - |11+\rangle$$
$$- e^{i\xi}(|01+\rangle + |11+\rangle - |00+\rangle + |10+\rangle$$
$$- |00-\rangle - |10-\rangle - |01-\rangle + |11-\rangle))$$

Re-factorising this expression then allows us to obtain the states of logical 0 and logical 1, demonstrating that the rotation applied to the first qubit has been applied to all three qubits. Thus this process can be used to encode information states into the logical qubits.

$$\begin{aligned}&\frac{1}{2\sqrt{2}e^{\frac{i\xi}{2}}}(-|0--\rangle + |1+-\rangle + |0++\rangle + |1-+\rangle \\ &- e^{i\xi}(-|0-+\rangle + |1++\rangle - |0+-\rangle - |1--\rangle)) \\ &= \frac{1}{\sqrt{2}}(e^{-i\xi}|0^L\rangle + e^{i\xi}|1^L\rangle)\end{aligned} \tag{60}$$

3.2.5 *Logical controlled Z operation*

The final essential building block for logical scheme is the ability to connect two logical qubits together through a logical controlled Z operation. The desired result for the operation is:

$$CZ^L_{AB}|+^L\rangle_A|+^L\rangle_B = \frac{1}{\sqrt{2}}(|0^L\rangle_A|+^L\rangle_B + |1^L\rangle_A|-^L\rangle_B) \tag{61}$$

However, this state can be achieved with only CZ operations.

$$\prod_{i,j=1}^{3} CZ_{a_i b_j} |+^L\rangle_A |+^L\rangle_B$$
$$= \frac{1}{2}(|+^L\rangle_A |+^L\rangle_B + |+-^L\rangle_A |+^L\rangle_B$$
$$+ |+^L\rangle_A |-^L\rangle_B - |-^L\rangle_A |-^L\rangle_B)$$
$$= \frac{1}{\sqrt{2}}(|0^L\rangle_A |+^L\rangle_B + |1^L\rangle_A |-^L\rangle_B)$$

This series of operators is difficult to demonstrate explicitly, however the state can be formed from a few more simple operations as will be shown in the next section.

3.3 ENTANGLED THREE QUBIT STATES

In order to demonstrate how a logical CZ state can be achieved through the a more basic method, we consider six qubits each in the + state separated onto two groups of three, a and b, which will each represent a logical qubit in the final state.

$$|+\rangle_{a_1} |+\rangle_{a_2} |+\rangle_{a_3} |+\rangle_{b_1} |+\rangle_{b_2} |+\rangle_{b_3}$$

First, CZ operations are applied in each section to form the (20) state labelled $|ghz_+\rangle$ states.

$$CZ_{a_1a_2}CZ_{a_1a_3}CZ_{b_1b_2}CZ_{b_1b_3} |+\rangle_{a_1} |+\rangle_{a_2} |+\rangle_{a_3} |+\rangle_{b_1} |+\rangle_{b_2} |+\rangle_{b_3}$$
$$= \frac{1}{2}[|0\rangle_{a_1} |+\rangle_{a_2} |+\rangle_{a_3} + |1\rangle_{a_1} |-\rangle_{a_2} |-\rangle_{a_3}]$$
$$[|0\rangle_{b_1} |+\rangle_{b_2} |+\rangle_{b_3} + |1\rangle_{b_1} |-\rangle_{b_2} |-\rangle_{b_3}]$$
$$= |ghz_+\rangle_{a_1a_2a_3} |ghz_+\rangle_{b_1b_2b_3} \tag{62}$$

Then the qubits denoted as 1 in both sections are entangled by a CZ operation to form a state denoted as $|G_2^+\rangle$.

$$CZ_{a_1b_1} |ghz_+\rangle_{a_1a_2a_3} |ghz_+\rangle_{b_1b_2b_3}$$
$$= \frac{1}{2}[|0\rangle_{a_1} |+\rangle_{a_2} |+\rangle_{a_3} [|0\rangle_{b_1} |+\rangle_{b_2} |+\rangle_{b_3} + |1\rangle_{b_1} |-\rangle_{b_2} |-\rangle_{b_3}]$$
$$+ |1\rangle_{a_1} |-\rangle_{a_2} |-\rangle_{a_3} [|0\rangle_{b_1} |+\rangle_{b_2} |+\rangle_{b_3} + |1\rangle_{b_1} |-\rangle_{b_2} |-\rangle_{b_3}]]$$
$$= |G_2^+\rangle \tag{63}$$

Next, Hadamard operations are applied to the first qubits in each section which produces an entangled state that is equivalent to applying CZ operations between each qubit in opposite sections.

$$\begin{aligned}
&|G_2^H\rangle = (H_{a_1} \otimes H_{b_1})\,|G_2^+\rangle \\
&= \frac{1}{2}[|+\rangle_{a_1}|+\rangle_{a_2}|+\rangle_{a_3}|+\rangle_{b_1}|+\rangle_{b_2}|+\rangle_{b_3} \\
&+ |+\rangle_{a_1}|+\rangle_{a_2}|+\rangle_{a_3}|-\rangle_{b_1}|-\rangle_{b_2}|-\rangle_{b_3} \\
&+ |-\rangle_{a_1}|-\rangle_{a_2}|-\rangle_{a_3}|+\rangle_{b_1}|+\rangle_{b_2}|+\rangle_{b_3} \\
&- |-\rangle_{a_1}|-\rangle_{a_2}|-\rangle_{a_3}|-\rangle_{b_1}|-\rangle_{b_2}|-\rangle_{b_3}]
\end{aligned} \tag{64}$$

Finally, CZ operations are applied between each qubit in their respective sections similarly to having the operator (40) applied. This will produce the logical CZ state specified in the expression (61).

$$\begin{aligned}
&CZ_{a_1a_2}CZ_{a_2a_3}CZ_{a_3a_1}CZ_{b_1b_2}CZ_{b_2b_3}CZ_{b_3b_1}\,|G_2^H\rangle \\
&= \frac{1}{2}[(|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle)_{a_1a_2a_3} \\
&(|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle)_{b_1b_2b_3} \\
&+ (|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle)_{a_1a_2a_3} \\
&(|+++\rangle - |--+\rangle - |-+-\rangle - |+--\rangle)_{b_1b_2b_3} \\
&+ (|+-+\rangle + |++-\rangle + |-++\rangle - |---\rangle)_{a_1a_2a_3} \\
&(|+++\rangle - |--+\rangle - |-+-\rangle - |+--\rangle)_{b_1b_2b_3} \\
&- (|+++\rangle - |--+\rangle - |-+-\rangle - |+--\rangle)_{a_1a_2a_3} \\
&(|+++\rangle - |--+\rangle - |-+-\rangle - |+--\rangle)_{b_1b_2b_3}] \\
&= \frac{1}{2}[|+^L\rangle_{a_1a_2a_3}|+^L\rangle_{b_1b_2b_3} + |+^L\rangle_{a_1a_2a_3}|-^L\rangle_{b_1b_2b_3} \\
&+ |-^L\rangle_{a_1a_2a_3}|+^L\rangle_{b_1b_2b_3} - |-^L\rangle_{a_1a_2a_3}|-^L\rangle_{b_1b_2b_3}]
\end{aligned} \tag{65}$$

This expression can be simplified to:

$$= \frac{1}{\sqrt{2}}(|0^L\rangle_{a_1a_2a_3}|+^L\rangle_{b_1b_2b_3} + |1^L\rangle_{a_1a_2a_3}|-^L\rangle_{b_1b_2b_3}) \tag{66}$$

Now that we have an expression for the logical CZ state we have the first step towards demonstration of the logical qubit system as a means towards error correction.

3.3.1 *Demonstration of the validity of simpler operations*

So far the validity of equation (64) has not been demonstrated. In order display its correct, we can perform CZ operations on six plus states.

$$
\begin{aligned}
&CZ_{a_1b_1}CZ_{a_1b_2}CZ_{a_1b_3}\\
&CZ_{a_2b_1}CZ_{a_2b_2}CZ_{a_2b_3}\\
&CZ_{a_3b_1}CZ_{a_3b_2}CZ_{a_3b_3}\\
&|+\rangle_{a_1}|+\rangle_{a_2}|+\rangle_{a_3}\\
&|+\rangle_{b_1}|+\rangle_{b_2}|+\rangle_{b_3}
\end{aligned}
$$

Applying the C-Z operations of the same index numbers yields

$$
\begin{aligned}
&CZ_{a_1b_2}CZ_{a_1b_3}CZ_{a_2b_1}\\
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{1}{8}[|{++}\rangle+|{-+}\rangle+|{+-}\rangle-|{--}\rangle]_{a_1b_1}\\
&[|{++}\rangle+|{-+}\rangle+|{+-}\rangle-|{--}\rangle]_{a_2b_2}\\
&[|{++}\rangle+|{-+}\rangle+|{+-}\rangle-|{--}\rangle]_{a_3b_3}
\end{aligned}
$$

Now applying the first 'diagonal' CZ operations between the qubits indexed as a_1 and b_2 and expanding the brackets:

$$
\begin{aligned}
& CZ_{a_1b_3}CZ_{a_2b_1} \\
& CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2} \\
& \frac{1}{16}[[|++++\rangle + |-+++\rangle + |+++-\rangle - |-++-\rangle] \\
& + [|+++-\rangle + |-++-\rangle + |++++\rangle - |-+++\rangle] \\
& + [|++-+\rangle + |-+-+\rangle + |++--\rangle - |-+--\rangle] \\
& - [|++--\rangle + |-+--\rangle + |++-+\rangle - |-+-+\rangle] \\
& \\
& + [|+-++\rangle + |--++\rangle + |+-+-\rangle - |--+-\rangle] \\
& + [|+-+-\rangle + |--+-\rangle + |+-++\rangle - |--++\rangle] \\
& + [|+--+\rangle + |---+\rangle + |+---\rangle - |----\rangle] \\
& - [|+---\rangle + |-+--\rangle + |+--+\rangle - |---+\rangle] \\
& \\
& + [|-+++\rangle + |++++\rangle + |-++-\rangle - |+++-\rangle] \\
& + [|-++-\rangle + |+++-\rangle + |-+++\rangle - |++++\rangle] \\
& + [|-+-+\rangle + |++-+\rangle + |-+--\rangle - |++-+\rangle] \\
& - [|-+--\rangle + |++--\rangle + |-+-+\rangle - |++-+\rangle] \\
& \\
& - [|--++\rangle + |+-++\rangle + |--+-\rangle - |+-+-\rangle] \\
& - [|--+-\rangle + |+-+-\rangle + |--++\rangle - |+-++\rangle] \\
& - [|---+\rangle + |+--+\rangle + |----\rangle + |+---\rangle] \\
& + [|----\rangle + |+---\rangle + |---+\rangle - |+--+\rangle]]_{a_1b_1a_2b_2} \\
& [|++\rangle + |-+\rangle + |+-\rangle - |--\rangle]_{a_3b_3}
\end{aligned}
$$

Half of the terms here will cancel leaving:

$$
\begin{aligned}
& CZ_{a_1b_3}CZ_{a_2b_1} \\
& CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2} \\
& \frac{1}{8}[[|++++\rangle + |+++-\rangle + |-+-+\rangle - |-+--\rangle] \\
& + [|+-++\rangle + |+-+-\rangle + |---+\rangle - |----\rangle] \\
& + [|-+++\rangle + |-++-\rangle + |++-+\rangle - |++--\rangle] \\
& + [|+---\rangle - |+--+\rangle - |--++\rangle - |--+-\rangle]]_{a_1b_1a_2b_2} \\
& [|++\rangle + |-+\rangle + |+-\rangle - |--\rangle]_{a_3b_3}
\end{aligned}
$$

Using the relations $|0\rangle = \sqrt{2}(|+\rangle + |-\rangle)$ and $|0\rangle = \sqrt{2}(|+\rangle + |-\rangle)$ this expression can be further simplified:

$$
\begin{aligned}
&CZ_{a_1b_3}CZ_{a_2b_1}\\
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{8}[[|0++\,+\rangle+|0++\,-\rangle+|0+-\,+\rangle-|0+-\,-\rangle]\\
&+[|1-+\,+\rangle+|1-+\,-\rangle-|1--\,+\rangle-|1--\,-\rangle]]_{a_1b_1a_2b_2}\\
&[|{+}{+}\rangle+|{-}{+}\rangle+|{+}{-}\rangle-|{-}{-}\rangle]_{a_3b_3}
\end{aligned}
$$

and:

$$
\begin{aligned}
&CZ_{a_1b_3}CZ_{a_2b_1}\\
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{1}{4}[|0++0\rangle+|0+-1\rangle+|1-+0\rangle-|1--1\rangle]_{a_1b_1a_2b_2}\\
&[|{+}{+}\rangle+|{-}{+}\rangle+|{+}{-}\rangle-|{-}{-}\rangle]_{a_3b_3}
\end{aligned}
$$

Applying the second "diagonal" between the qubits indexed as a_2 and b_1 now gives

$$
\begin{aligned}
&CZ_{a_1b_3}CZ_{a_2b_3}\\
&CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{1}{8}[|0\rangle\,[|{+}{+}\rangle+|{-}{+}\rangle+|{+}{-}\rangle-|{-}{-}\rangle]\,|0\rangle\\
&+|0\rangle\,[|{+}{-}\rangle+|{-}{-}\rangle+|{+}{+}\rangle-|{-}{+}\rangle]\,|1\rangle\\
&+|1\rangle\,[|{-}{+}\rangle+|{+}{+}\rangle+|{-}{-}\rangle-|{+}{-}\rangle]\,|0\rangle\\
&-|1\rangle\,[|{-}{-}\rangle+|{+}{-}\rangle+|{-}{+}\rangle-|{+}{+}\rangle]\,|1\rangle]_{a_1b_1a_2b_2}\\
&[|{+}{+}\rangle+|{-}{+}\rangle+|{+}{-}\rangle-|{-}{-}\rangle]_{a_3b_3}
\end{aligned}
$$

Expanding these brackets produces a large number of terms (64), however most terms will cancel yielding:

$$
\begin{aligned}
&CZ_{a_1b_3}CZ_{a_2b_3}\\
&CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{1}{4}[|{+}{+}{+}{+}\rangle+|{+}{-}{+}{-}\rangle+|{-}{+}{-}{+}\rangle-|{-}{-}{-}{-}\rangle]_{a_1b_1a_2b_2}\\
&[|{+}{+}\rangle+|{-}{+}\rangle+|{+}{-}\rangle-|{-}{-}\rangle]_{a_3b_3}
\end{aligned}
$$

3.3.2 *Second set of diagonal operations*

Next we apply the CZ operator between qubits a_1 and b_3

$$
\begin{aligned}
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{1}{4}[\left|+++\right\rangle_{b_1a_2b_2}[CZ\left|++\right\rangle(\left|+\right\rangle\\
&+\left|-\right\rangle)+CZ\left|+-\right\rangle(\left|+\right\rangle--)]_{a_1b_3a_3}\\
&+\left|-+-\right\rangle_{b_1a_2b_2}[CZ\left|++\right\rangle(\left|+\right\rangle\\
&+\left|-\right\rangle)+CZ\left|+-\right\rangle(\left|+\right\rangle--)]_{a_1b_3a_3}\\
&+\left|+-+\right\rangle_{b_1a_2b_2}[CZ\left|-+\right\rangle(\left|+\right\rangle\\
&+\left|-\right\rangle)+CZ\left|--\right\rangle(\left|+\right\rangle--)]_{a_1b_3a_3}\\
&-\left|---\right\rangle_{b_1a_2b_2}[CZ\left|-+\right\rangle(\left|+\right\rangle\\
&+\left|-\right\rangle)+CZ\left|--\right\rangle(\left|+\right\rangle--)]_{a_1b_3a_3}]
\end{aligned}
$$

$$
\begin{aligned}
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{4}[\left|+++\right\rangle+\left|-+-\right\rangle]_{b_1a_2b_2}[\left|+0+\right\rangle+\left|-1-\right\rangle]_{a_1b_3a_3}\\
&+[\left|+-+\right\rangle-\left|---\right\rangle]_{b_1a_2b_2}[\left|+1-\right\rangle+\left|-0+\right\rangle]_{a_1b_3a_3}
\end{aligned}
$$

Expanding these brackets and rearranging the order of the qubits in the kets gives:

$$
\begin{aligned}
&CZ_{a_2b_3}CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{4}[\left|+++++0\right\rangle+\left|-+-++1\right\rangle\\
&+\left|+++--0\right\rangle+\left|-+---1\right\rangle\\
&+\left|+--++1\right\rangle+\left|--+++0\right\rangle\\
&-\left|+----1\right\rangle-\left|--+--0\right\rangle]_{a_1a_2a_3b_1b_2b_3}
\end{aligned}
$$

Using the relations:

$$
\begin{aligned}
&CZ\left|+0\right\rangle=\left|+0\right\rangle\\
&CZ\left|-0\right\rangle=\left|-0\right\rangle\\
&CZ\left|+1\right\rangle=\left|-1\right\rangle\\
&CZ\left|-1\right\rangle=\left|+1\right\rangle
\end{aligned}
$$

We can now apply the CZ operator between qubits a_2 and b_3

$$
\begin{aligned}
&CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{4}[|+++++0\rangle+|---++1\rangle\\
&+|+++--0\rangle+|-----1\rangle\\
&+|++-++1\rangle+|--+++0\rangle\\
&-|++---1\rangle-|--+--0\rangle]_{a_1a_2a_3b_1b_2b_3}
\end{aligned}
$$

3.3.3 *Final diagonals*

By re-factorising the expression such that $|0\rangle$ and $|1\rangle$ terms are being operated on we can further mitigate the need for complex expressions. Hence we rearrange to get:

$$
\begin{aligned}
&CZ_{a_3b_1}CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{4}[|++0+++\rangle+|--0+++\rangle\\
&+|++1--+\rangle-|--1--+\rangle\\
&+|++1++-\rangle+|--1++-\rangle\\
&+|++0---\rangle-|--0---\rangle]_{a_1a_2a_3b_1b_2b_3}
\end{aligned}
$$

Now the CZ operator between qubits a_3 and b_1 can be easily applied:

$$
\begin{aligned}
&CZ_{a_3b_2}\\
&\frac{\sqrt{2}}{4}[|++0+++\rangle+|--0+++\rangle\\
&+|++1+-+\rangle-|--1+-+\rangle\\
&+|++1-+-\rangle+|--1-+-\rangle\\
&+|++0---\rangle-|--0---\rangle]_{a_1a_2a_3b_1b_2b_3}
\end{aligned}
$$

Finally, applying the CZ operator between qubits a_3 and b_2

$$\frac{\sqrt{2}}{4}[|{+}{+}0{+}{+}{+}\rangle + |{-}{-}0{+}{+}{+}\rangle$$
$$+ |{+}{+}1{+}{+}{+}\rangle - |{-}{-}1{+}{+}{+}\rangle$$
$$+ |{+}{+}1{-}{-}{-}\rangle + |{-}{-}1{-}{-}{-}\rangle$$
$$+ |{+}{+}0{-}{-}{-}\rangle - |{-}{-}0{-}{-}{-}\rangle]_{a_1a_2a_3b_1b_2b_3}$$

and this expression is nothing more than:

$$\frac{1}{2}[|{+}{+}{+}{+}{+}{+}\rangle + |{-}{-}{-}{+}{+}{+}\rangle$$
$$+ |{+}{+}{+}{-}{-}{-}\rangle - |{-}{-}{-}{-}{-}{-}\rangle]_{a_1a_2a_3b_1b_2b_3}$$

Thus demonstrating the validity of equations (63) and (64) in the formation of a logical CZ state.

3.4 General Encoding

The operation to initialise a CZ state in the previous section is, however, specific for an initial state in the first qubit consisting of three $|+\rangle$ state qubits. In order to realise a generalised logical state and perform a CZ operation, we require usage of the logical rotation operator and the the CZ operations required to transform three $|+\rangle$ state qubits into the $|+^L\rangle$ state.

This process will have an identical effect to the encoding and decoding circuits for initialising the error correction described in [2]. By combining this with the procedures described in equations (63) and (64), we can effectively perform the full quantum error correction circuit described in [2].

3.5 Error

The functionality of this error correction scheme comes from the detection of states in the auxiliary qubits. After the formation of the logical cluster state, there is an additional decoding step consisting of another set of CZ operators and then the conjugate of the part of the rotation operator. The measured states of the auxiliary qubits will then be in the form of a binary value which indicates the kind of logical X and logical Z operations required to correct the fault. For the case of 5 qubits, the table from Joo's paper [2] is shown in figure 3.

This process is also represented with a corresponding circuit model in Joo's paper [2], included here in figure 4 to facilitate understanding of the table.

Error type	Syndrome ($\lvert a\,b\,c\,d\rangle_{2345}$)	Outcome
None	0000	
Z_2	1000	
Z_3	0100	$\lvert\psi\rangle$
Z_4	0010	
Z_5	0001	
X_1	1001	
X_3	1010	
X_4	0101	$X\lvert\psi\rangle$
X_3Z_3	1110	
X_4Z_4	0111	
X_1Z_1	0110	
X_2	1011	
X_5	1101	$XZ\lvert\psi\rangle$
X_2Z_2	0011	
X_5Z_5	1100	
Z_1	1111	$Z\lvert\psi\rangle$

Figure 3: Error correction table based on outcomes in auxiliary qubits [2]

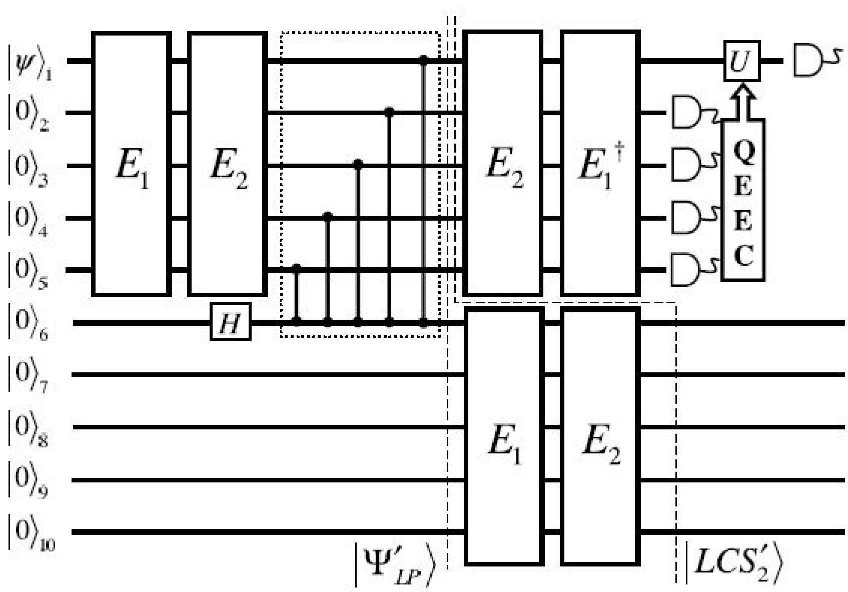

Figure 4: Full error correction circuit [2]

3.6 higher complexities

One of the advantages of this error correction scheme is that it can be easily scaled for the encoding of higher numbers of qubits in each logical state. The conversion for this is simply to scale all operators that act on all three qubits to operate on the number of qubits in the new logical state. The effect of this should be a higher degree of fault tolerance when the error correction scheme is applied.

4 SIMULATION OF ONE WAY QUANTUM COMPUTATION

For the next part of the project, a simulation was to be created that would allow for investigation of levels of fault tolerance for different logical qubits in the error correction scheme through Monte Carlo methods. The program would provide simulations of the single gate operations with randomised errors being introduced to the state vector of the system between the encoding and measurement steps. With these randomised errors and by comparing the outputs of the program with expected results, the fault tolerance of each set-up could be determined through repeated running of the program and collection of data. By then comparing the fault tolerance between various sizes of logical qubit in 'triangle' or 'pentagon' states etc, a correlation between fault tolerance and logical qubit size could be established to aid in choice of error correction scheme based on physical parameters of a system.

4.1 SET-UP

The first step of any simulation is, of course, to establish how the problem will be solved. In this regard, the simulation was first planned in pseudo code and the relevant components sketched out. This plan was somewhat ambitious as it encompassed a scheme for universal applications of one way quantum computation with the described error correction scheme. Though much of this plan was not completed, its contents are described in this section.

4.1.1 *Resources used*

For the simulation fortran 95 was chosen as an appropriate language due to it's scalability and ability to be broken up into modules. Details of the compilers, libraries and system used for simulation will be included in Appendix B.

Fortran 95 also contained standard functions to call for random numbers that were required for both measurement results and the addition of random errors as well as the ability to determine array sizes intrinsically, allowing for a tidier coding of subroutines.

4.1.2 *Subroutines*

A key requirement for the structure of any program was that it would be comprised of subroutines that handled the various steps of the operation of the modelled computer such that these subroutines could be easily rearranged in the main program to simulate any required topological structure. For example, the various single qubit operations required to satisfy universal computation in [1] can be simulated using the same physical system of a short chain of qubits but with different eigenbases for measurements. Hence in the simulation of a chain of spins the program was designed such that the required measurement operations could be read from a file allowing for any single qubit gate to be simulated with the same program.

Additionally, the two qubit CZ operation was written such that it could be performed between any two qubits in the state vector of the system, allowing it to be used in more complicated programs that would simulate the systems from the triangle states required for the basic implementation of the error correction code to the 8-qubit controlled phase gates.

As such every initialisation and measurement operation was set up to be self contained into one of three modules. The CZ operation module, the measurement module and an additional module that contained several linear algebra procedures required for the program to function. The modular structure also allowed for dynamic array allocation to occur at the top level of the program as array sizes could be passed to the various modules without need to include array sizes as extra arguments in calls to subroutines. This meant that all calls to internal subroutines were reasonably understandable and concise in the coding.

4.1.3 *Libraries*

As this program was primarily structured around matrices and linear algebra, extensive use of the BLAS external library as well as the LAPACK library [11]. Not only would this make some calculation easier to code, but also meant that the program had some capacity for scalability onto parallel architectures given the structure of the routines in the library. Whenever possible BLAS was used in addition to intrinsic Fortran calls for dot products or matrix multiplication with preprocessor statements allowing the option of compiling without external libraries. However, as the program developed it became more dependant on these libraries and so is no longer functional without them.

4.1.4 *Precision*

As the program relied on numerical solutions to systems of linear equations, precision in variables was tantamount to functionality. As such IEEE 754 double precision [12] was used in the program as standard in all real and complex variables, arrays and conversion functions. Because of this machine error was extremely low for all runs of the program, manifesting itself only in null values. While these null values may look untidy in current outputs for the program, they current provide a reasonable guarantee of numerical accuracy and can be easily cleared up in later versions through a rounding step before writing to file.

However, this would mean that each double precision value in the state vectors of the subroutine or in operation matrices would be using 64 bits for each value stored. Hence the size of the system would quickly cause the memory usage to expand drastically. For a one dimensional chain of qubits, this is not a problem as qubits can be removed from the state vector after measurement, resulting in maximum memory usages of 128×2^2 for the state vector. In the case of simulation of the quantum Fourier transform described in [1], at least 20 qubits would be required, causing memory requirements of 128×2^{20} bits, which would be around 17 megabytes. While this is easily manageable for single qubits, incorporation of the triangle states as an error correction code would require around 18,000 petabytes, a value clearly infeasible for computation as even the top supercomputers have access to even one petabyte [13] TOP 500 super computers.

4.2 CZ OPERATION

The first building block subroutine implemented into the program was a CZ operation subroutine. This subroutine would take a state vector and apply a CZ operation based on integer values of control and target bits. In this way, CZ operations could be achieved between any two qubits in the state vector, satisfying the requirements for initialisation of both measurement based single qubit gates and triangle states for error correction.

4.2.1 *Kronecker Product routine*

In order to generate the matrix describing the CZ operation between two qubits in the state vector, a subroutine was adapted from previous work [14], which was updated for Fortran 95's ability to intrinsically determine the size of an array. This subroutine for a Kronecker product was necessary to code as an equivalent was not found in the common external Fortran libraries.

Using this subroutine CZ matrices were generated from 2×2 identity and Pauli-z matrices based on the the unitary operator (19).

Listing 1: CZ generation main algorithm

```
!Loops over values 1 to 4 with J
!Reflective of the unitary operator which has 4 terms
do j = 1, 4

  !Allocates the secondary work matrix into the initial size
  !for Kronecker product multiplication
  Allocate(out_matrix(2, 2))

  !Performs a check to see if the first matrix in the order
      of
  !multiplication corresponds to a control or target
  !if they do, assigns the appropriate matrix dependant
  !upon the value of j.
  !Otherwise assigns the identity matrix
  if((trgt.eq.1).and.((j.eq.2).or.(j.eq.4))) then
    out_matrix = z_matrix
  elseif((ctrl.eq.1).and.((j.eq.3).or.(j.eq.4))) then
    out_matrix = z_matrix
  else
    out_matrix = identity
  end if

  !Loops over each individual qubit
  !This will produce a matrix of appropriate size for each
  !term of the unitary operator
  do i = 2, n

    !Assigns a size value to the work matrix dependent
    !on the step in the loop, thus allowing it
    !to contain the appropriate size of matrix at this step
    Allocate(work_matrix(2**(i-1), 2**(i-1)))

    !Assigns the work matrix the value of the output matrix
    !Takes the value from the output of last step
    !Allowing output matrix to be deallocated
    work_matrix = out_matrix

    !Deallocate output matrix
    Deallocate(out_matrix)

    !Allocates new size to the output matrix
    !New size is appropriate for storage of
    !Kronecker product between work matrix and a 2x2 matrix
    Allocate(out_matrix(2**i, 2**i))

    !Determines whether the next operator of the
        multiplication
```

```
          !Will be a control or target qubit, depending on the
              term of U
          !Assigns the appropriate value if so for kronecker
              products
          !Otherwise uses the identity matrix for the kronecker
              product
          if((ctrl.eq.i).and.((j.eq.3).or.(j.eq.4))) then
                call kronecker_product_complex(work_matrix,
                    z_matrix, out_matrix)
          elseif((trgt.eq.i).and.((j.eq.2).or.(j.eq.4))) then
                call kronecker_product_complex(work_matrix,
                    z_matrix, out_matrix)
          else
                call kronecker_product_complex(work_matrix,
                    identity, out_matrix)
          end if

          !Deallocates the work matrix ready
          !For allocation in next loop
          Deallocate(work_matrix)

        !Ends do loop for the jth term of the Operator
        end do

        !Determines which term of operator the loop is on
        !If j is four, removes output from cz_matrix
        !otherwise adds output to cz_matrix
        !Reflective of signs of unitary operator
        if(j.eq.4) then
          cz_matrix = cz_matrix - out_matrix
        else
          cz_matrix = cz_matrix + out_matrix
        end if

        !Deallocates out matrix for next loop of j
        Deallocate(out_matrix)

      end do
```

4.2.2 *Improvements*

As the CZ operation matrix is always diagonal, the subroutine could be easily improved to calculate and store the operation matrix as a one dimensional array. This would allow the use of a simpler Kronecker product routine designed only for vectors, but could also make calls to other subroutines unnecessary as the calculation of the operator would be reasonably trivial with a standard formula for generation that required only multiplication of negative numbers into the operator at the appropriate places.

Additionally, the array could be downscaled to integer values and then converted to double precision only when directly applied to the state vector, making savings in computation time and memory usage.

It would also be possible to generate CZ operators for the formation of triangle states similar to the unitary operator in (41). By generating these through a separate routine, the application of triangular CZ states shown in the equation (65) for the encoding of the logical cluster state, would be optimized further. This operation could even be generalised to satisfy pentagonal, heptagonal or any other alternative formation of logical qubits in the error correction scheme.

4.3 OTHER QUBIT OPERATIONS

One requirement for the modules of the program was that circuit model style operations would need to be performed in order to encode the error correction scheme into the system.

4.3.1 *Pauli Z Operation*

While a Pauli Z operation was not coded into the program for single qubits, such a routine would be reasonably simple to implement using a simple modification of the CZ operation subroutine. Fundamentally the CZ operation includes two single qubit Z operations so by removing one of the nested loops the desired effect could be achieved. This subroutine could also be optimised as a diagonal matrix or as integer values in a similar way to the CZ operation, minimising memory usage and computational time.

4.3.2 *Hadamard Operation*

Similarly to the Pauli Z operation, the error correction scheme relies on the application of multiple Hadamard operations. Unlike the Z operation, this operator can be required to be applied to multiple qubits at the same time and as such requires a more complicated subroutine. Using the same Kronecker product subroutine it would be possible to form an operator that acts on an array of integers representing qubits to be operated upon. This way any possible error protection configuration could easily be handled by the same subroutine.

While this subroutine could not be held as a single dimensional array like the CZ and Z operations, it could still be stored as an integer, provided a real value be held as an additional variable to reflect the constant value multiplied by the matrix.

4.3.3 *Logical Operations*

In order to properly simulate the error correction protocols, the logical operators that are applied for both encoding and error correction need to be incorporated. These operators would be easily handled in the same way as the other operators described earlier as they can be seen to be components of the logical operators. As a result, the logical operator subroutines would operate at a higher level in the program, making use of the basic operators as resources.

The key condition in designing a subroutine to function such that it can be used for larger logical qubits in order to compare the fault tolerance correlation between logical qubit sizes. This would be trivial to code however as it would just require the number of loops used to apply operations to all of the qubits comprising the logical qubits to be set in the argument of the subroutine rather than hard coded into the program.

4.4 MEASUREMENT

Obviously the most vital subroutine for the simulation of the operation of a measurement based quantum computation scheme is the act of measurement itself. Fundamentally, this subroutine must handle the functionality of any quantum algorithms whilst also supporting certain metaphysical principles and assumptions [15]. The program was designed to reflect the same metaphysical assumptions used in the algebraic model earlier and hence the model for measurement used earlier was also applied in the program.

4.4.1 *Random measurement in basis*

To reflect the random nature of measurement outcomes in each basis, the intrinsic random number generator in Fortran 95 was employed. This provided a random double precision number between 0 and 1 which was then rounded to the nearest integer value with the intrinsic function NINT(). This provided a simple way to deal with random measurement outcomes whilst also allowing for fixed outcomes in debugging as the default seed for the random number function is consistent, providing identical outcomes on each calling. Whilst seed generation for proper running of the program was explored, the time limited nature of programming meant that such features were never incorporated into the program at large, but it is likely that a seed generation library from the operating system would have been used or a custom one created based on system time.

The integer values representing the outcomes of the measurements were then written to file where they could be retrieved by the subroutine handling the feed-forward of measurement outcomes at the end of the program.

4.4.2 *Pauli bases measurement*

In order to easily model CNOT, Hadamard and $\pi/2$ phase gates, a subroutine for measurement in the three Pauli bases which correspond to the eigenvectors of the 2×2 Pauli matrices was incorporated into the program. As these measurement bases were commonly used in almost all qubit gates it made sense to have them hard coded into the program instead of using a single generalised measurement routine

$$\begin{aligned}
\sigma_{x+} &= \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}, \quad \sigma_{x-} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}, \\
\sigma_{y+} &= \frac{1}{\sqrt{2}}\begin{pmatrix}1\\i\end{pmatrix}, \quad \sigma_{y-} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-i\end{pmatrix}, \\
\sigma_{z+} &= \begin{pmatrix}1\\0\end{pmatrix}, \quad \sigma_{z-} = \begin{pmatrix}0\\1\end{pmatrix}.
\end{aligned} \tag{67}$$

In order to keep the programming concise, all three kinds of basis were included into the same subroutine which was controlled by a single character in the argument of the subroutine. This character would then be recorded in a file containing data about the measurements that could be used to calculate the final states in the feed forward subroutine. Currently however the program uses the phase directly, but this could be optimised in future iterations by hard coding the phase amounts of the Pauli bases into the feed forward routine. Unfortunately the Pauli Y and Pauli Z basis measurements do not work in the program when it comes to the final feed forward of measurement because of an error with the phase of the basis when it comes to feed forward of measurements to obtain the final state, so this change might even help alleviate that problem.

4.4.3 *Arbitrary measurement*

In addition the the Pauli bases measurements, a generalised measurement of arbitrary phase was also required for the program to implement the general rotation and z rotation single qubit gates. The subroutine for this process was very similar to the Pauli measurement subroutine aside from the inclusion of a variable measurement phase in the arguments of the subroutine and a general rotation vector replacing the eigenvectors of the Pauli bases.

The subroutine, like the Pauli measurement subroutine, recorded the variable phase amount to a file so that it could be used in the feed forward subroutine as well as multiplying the result of the inner product of the measurement vector and the measured qubit state vector by the global state vector. Potentially this subroutine could be consolidated into the Pauli measurement subroutine through the use of the control character, likely with 'R' signifying a custom phase, but this is yet to be performed.

4.4.4 *Feed Forward*

The final necessary component of the measurement scheme consisted of a subroutine to feed forward measurement outcomes to recover the desired state at the end of the measurements. This subroutine performed the functionality of equation (32) for the single qubit states.

In order to achieve this result, the subroutine read the measurements performed over the course of the program in order and then applied (32) for each measurement, leading to accurate results.

The subroutine also has the functionality of being able to decompose larger state vectors in order to apply the feed forward results to specific qubits, but this is feature is currently unfinished.

4.4.5 *Multiple Outputs*

One of the weaknesses of the current subroutine to handle the feed forward of measurements is that it cannot deal with output state vectors consisting of multiple qubits simultaneously. For example, in the output state of the CNOT gate in equation (37), a series of decoding operations are required to be performed on two qubits at once, which the subroutine would not be able to handle.

Fortunately this is only a minor problem as the error correction scheme only requires a single output qubit and with sufficient time the subroutine could be updated to deal with such problems if deemed necessary.

4.5 DECOMPOSITION

One of the critical elements to the functionality of the program was the ability to decompose the global state vector into individual qubit states in order for both the simulated measurement to be processed and the final state of the qubit to be determined. This required a reversing of the Kronecker product routine used to generate the state vector which presented a formidable challenge computationally.

While Kronecker products are not directly reversible, it is possible to determine a 'canonical' result for certain matrices. The principle behind this relies upon the relation between the Kronecker product and the vectorisation operator. Considering the vector W which is a Kronecker product of two real vectors U and V:

$$W = V \otimes U = vec(UV^T) \tag{68}$$

Vectorisation is a process which is easily reversed, so the vectors U and V can be recovered by decomposition of the resulting matrix. Note that for complex vectors V^T becomes V^H which represents the conjugate transpose of V.

$$UV^T = vec^{-1}(W) \tag{69}$$

Unfortunately, there are a wide range of possible decompositions, so this is where a canonical decomposition must be decided upon in order to progress further.

4.5.1 *Rank decomposition*

The first subroutine developed to perform this decomposition was a simple rank 1 decomposition. It was originally thought that this would be sufficient in all cases as the Kronecker product of two vectors would always be rank one. However this was short-sighted as the application of the CZ operation modified the rank of the state vector, a phenomenon which seems to be representative of the entanglement induced in the system.

The algorithm worked by pulling the first non-zero column of the matrix formed from the inverse vectorisation of W and normalising it. Then the algorithm divided each value of the first non-zero row by the new normalised values to obtain the other vector.

For a rank 1 state vector of pure such as $|00\rangle$ the algorithm was successful, but when encountering mixed states, the algorithm could not handle the input and returned null values. This was obviously unacceptable as the CZ operation will always make mixed states out of pure input states.

Listing 2: Rank decomposition algorithm

```
!Perform an inverse of the vec() operator by reshaping
!vector A into a matrix of dimensions n x o
B = TRANSPOSE(RESHAPE(A, (/ o, n /)))


do i = 1, o
  do j = 1, n
    if(B(j, i) /= 0.0_dp) then
      C(j) = B(j, i)
```

```
      exit
    else
      continue
    end if
  end do
end do

!Normalise

!Calculate length
do i = 1, n
 magnitude = magnitude + C(i)*C(i)
end do

magnitude = sqrt(magnitude)


!Divide components by length
C = C / magnitude

do i = 1, o
  do j = 1, n
    if(C(j) /= 0.0_dp) then
      D(i) = (1.0_dp/(C(j))) * B(j, i)
      exit
    else
      continue
    end if
  end do
end do
```

4.5.2 *Single value decomposition*

As the rank decomposition was not a sufficient way to canonically decompose the matrix, the single value decomposition (SVD) was chosen as an alternative. As this was an $M \times N$ decomposition it would allow for particular qubits to be determined, compared to something like the eigen-decomposition which would only work with square matrices and thus be only able to break the state vector in half.

Fortunately, unlike the rank decomposition there are plenty of available libraries for single value decomposition. For the simulation the double precision complex general decomposition routine zgesvd was chosen from the LAPACK library as it both complied with the established IEEE 754 precision standard and allowed for decomposition of a general matrix.

However, the SVD is a little more complicated than the rank decomposition as the matrix is broken into three parts. In the case of a complex matrix these consist of a diagonal matrix $\mathbf{\Sigma}$ of the singular values σ_i of the matrix and two matrices **U** and **V** composed of rows and columns of vectors such that:

$$\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H \tag{70}$$

When performing this decomposition in the program the resulting vectors U and V^H were assumed to be the summations of the rows or columns of matrices **U** and **V** respectively, while the i-th rows of **U** were multiplied by the corresponding single values σ_i.

The routine was tested with the application of a pauli X basis measurement on a pair of two qubits initialised in the $|+\rangle$ state before having the CZ operation applied. The resulting state vector of this initialisation was:

$$W = \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{pmatrix} \tag{71}$$

So when inverse vectorisation was applied this became:

$$\mathbf{M} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \tag{72}$$

When entered into the subroutine the output was:

$$\mathbf{U} = \begin{pmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \quad \mathbf{\Sigma} = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \quad \mathbf{V}^H = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \tag{73}$$

So the vectors U and V became:

$$U = \begin{pmatrix} \frac{2}{\sqrt{2}} \\ 0 \end{pmatrix} V = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \tag{74}$$

When the measurement in the Pauli X basis was applied, in case one:

$$\begin{aligned} &|+\rangle \langle +|U\rangle \otimes |V\rangle \\ &= |+\rangle \left(\frac{1}{\sqrt{2}}\frac{2}{\sqrt{2}} \langle 0|0\rangle\right) \otimes |V\rangle \\ &= |+\rangle \otimes |V\rangle \end{aligned} \tag{75}$$

Feeding the measurement result forward to determine the output state then gives:

$$e^0 X^0 H R_0 |V\rangle = H \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = |0\rangle \tag{76}$$

Similarly for the other measurement outcome:

$$e^0 X^1 H R_0 |V\rangle = XH \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = |1\rangle \tag{77}$$

These outcomes match the expected outcomes for measurements of the qubit in the pauli X basis which is easily seen by looking measurement of the state expressed in the form described in equation (16). In case 1:

$$\langle +| \frac{1}{\sqrt{2}}(|{+0}\rangle + |{-1}\rangle) = |0\rangle = H|+\rangle \tag{78}$$

And in case 2:

$$\langle -| \frac{1}{\sqrt{2}}(|{+0}\rangle + |{-1}\rangle) = |1\rangle = XH|+\rangle \tag{79}$$

Unfortunately the decomposition has not been tested for larger or more complex state vectors due to time constraints. However, it seems like there will be certain matrices which will throw up errors due to the multiplication of zero values or through the summation of rows and columns being zero, this could be fixed by performing multiple decompositions until a satisfactory decomposition for which zero values are minimised is achieved, but this would take extra computational resources and be mathematically untidy. In this regard, it seems that there must be a better way to reconstruct the vectors from the decomposition matrices but so far this remains unclear.

4.6 ERROR

One of the crucial functionalities a completed program would be the ability to add randomised errors to a single or multiple qubits in a state vector of any size in order for the fault tolerance of each system to be analysed. Such a subroutine was only in its early stages in the program as it had not yet advanced to the stage of implementation of the error correction scheme necessary for the subroutine to be meaningful, but a clear plan had been drawn up as to how errors would be implemented numerically into the system. Ideally error would be induced between the CZ and measurement operations, as this would be consistent with the state vector that is resilient to errors in the error correction scheme, but error could be easily added to the state vector of the system at any time.

4.6.1 *Flip error*

Flip errors are reasonably easy to simulate in the state vector in any size as they can be effectively expressed as the application of a Pauli X operation to a qubit. In order to operate on a single qubit, or multiple, the Kronecker product subroutine would be employed to form operator matrices of appropriate size in a similar fashion to the way operators would be used to initialise the system with the error correction scheme. For example the operator operation applied to flip qubit i in a state vector of N qubits would be:

$$\mathbb{1}_1 \otimes \mathbb{1}_2 \otimes \cdots \sigma_X \cdots \otimes \mathbb{1}_N \tag{80}$$

It is therefore clear that such an operator could easily be produced with nested loops of the Kronecker product subroutine, what remains unclear however is as which qubits the flips would be applied and at what rate.

The simple way to decide which qubits to flip would be to have a universal error rate for bit flips and then include either Pauli operators or identity operators depending on whether a random number meets the threshold of the error rate. This would have the bonus effect of saving memory and calculation time compared to performing each flip individually as well as simplifying the programming of any subroutine handling errors.

The alternative to this would be to have separate error rates for each number of errors so that the number of errors are calculated and then assigned to random qubits in the following step. This would be particularly useful when testing whether the error correction scheme protects against particular numbers of phase flips, though would obviously cause a slight interest in computational requirements.

One advantage to only modelling phase flips however is that any operator matrix can be stored in integer form and only converted to double precision when multiplying out with the state vector. This would save large amounts of memory compared to a phase error matrix which would require double precision storage and thus have memory requirements equal to the square of the memory requirements of the state vector.

4.6.2 *Phase error*

Similarly to the flip error, phase errors can be simulated through the inclusion of a phase operation on a qubit in the state vector. However, the phase of the error present provides another variable for the model as phase errors are obviously continuous variables rather than the discrete flip errors. Hence not only would a random number of qubits need to be exposed to the error, the exact value of the error would also need to be determined. While this could be done with a fixed error, something like a Gaussian distribution would work reasonably well for modelling. Indeed, if a a Gaussian distribution with a particularly high standard deviation were used, Phase errors could be applied to every qubit, reducing the number of steps in calculation of the operator.

4.7 DATA ANALYSIS

Another key component for the program is the ability to analyse cohesive and accurate output data. This puts requirements on the format of output data as well as the method of storage. For the program in its current form, data is output formally from the fidelity function, the value of the state vectors is printed directly to standard output, which was typically piped into a separate file.

4.7.1 *Fidelity*

In order to determine if the output state and expected output state correspond to one another, a fidelity subroutine was also included in the program. Like the Kronecker product subroutine, this subroutine was adapted from earlier work and simply performed the basic calculation:

$$\text{Fidelity} = |\,\langle \text{Expected State}|\text{Final State}\rangle\,|^2 \tag{81}$$

Like the other subroutines in the program, this calculation could be performed with or without the BLAS library's complex double precision dot product routine (ZDOTU) and as such has potential for scalability into parallel processing.

4.7.2 *Bulk data*

The ideal end result for a complete program would involve a large data set of fidelity data from multiple runs of the program. As a result some changes would have to be made so that fidelity data was instead appended to the file. Once this had been done, a scatter plot between the number of gates performed on the data and the fidelity of the result would be plotted for each size of logical qubit.

This data would then be fitted to a curve using whatever method best worked for the output in order to establish the correlation between the two axes depending on the fixed error rate from the program. This would hopefully lead to a better understanding of the way in which fault tolerance was affected by the size of logical qubits for this measurement scheme.

4.8 ERROR CORRECTION SCHEME

The final part of the program is the incorporation of the error correction scheme. This involves two main subroutines, the encoding subroutine and the detection/correction subroutine.

4.8.1 *Encoding*

The encoding subroutines main function would be reflective rotational operation combined with the method for forming logical CZ operators described in section 3.3. This would be challenging to perform and would require a great deal of coding in order for the operations to be scalable to any size logical qubit.

4.8.2 *Correction*

Correction, in comparison, would be a little easier to implement as the measurement routine currently in use for the program could be recycled for the bulk of the functionality. The main challenge would then be the conditional statements to implement the scheme and apply the appropriate logical operations depending upon the measurement outcomes. There would also need to be a small decoding step also, but this could be adapted from the encoding step which would need to be implemented first for correction to be possible in the first place.

4.9 OTHER STRUCTURES

In addition to testing the error correction scheme on single qubit gates, there are some other gates and set-ups that would be both interesting and useful to simulate with the program. These structures are of interest in particular as they represent the key building blocks of common quantum algorithms.

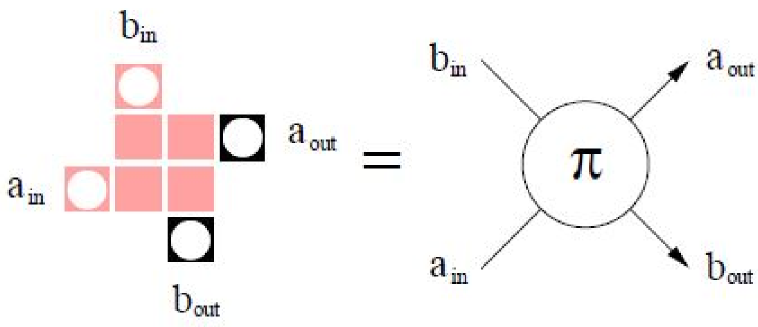

Figure 5: Controlled phase gate structure and functionality [1]

4.9.1 *Controlled not gate*

The first interesting structure that would make a good extension to the program would be a controlled not gate, though implementation would require the problems with the feed forward subroutine being unable to deal with multiple qubit state vectors to be rectified.

Obviously one of the advantages of demonstrating that this structure can be fault tolerant is that, when combined with the single qubit gates, forms a universal set of quantum gates. This means that such a demonstration would be necessary to prove the validity of the error correction scheme as universal.

Unlike the single qubit gates, it would not be possible to perform this operation repetitively without the introduction of new control qubits. While this is not too difficult a problem to overcome, it would make comparison of final results with expected outcomes more challenging as the expected outcomes would require far more calculation than repetitive use of a $\pi/2$ gate, for example.

4.9.2 *Controlled phase gate*

A very interesting structure for simulation by the program, would be a controlled phase gate. This structure is interesting both because not only can it be represented without the decompositions into CNOTs and rotations required for the circuit model, but it is also a crucial building block for the Quantum Fourier transform algorithm [1].

This structure, show in figure 5, consists of 8 qubits in a spiral pattern with measurements performed on the central square, far simpler than the chain of gates required in the circuit model. However, like the CNOT gate, implementation of a measurement feed forward subroutine that could handle multiple outputs would be required. Implementation of this structure into the model with the error correction code would be also quite interesting given the entanglement of four qubits simultaneously, which would present an interesting programming challenge.

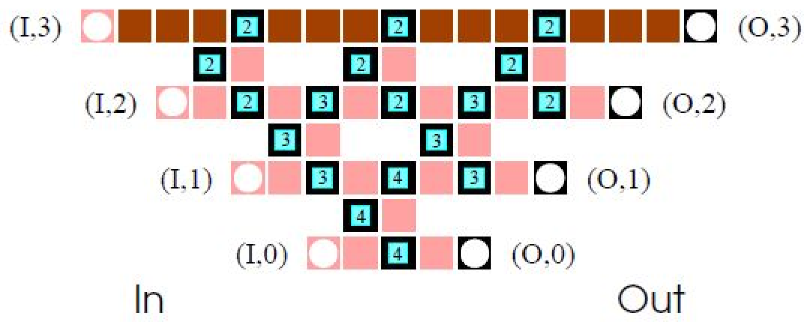

Figure 6: Quantum Fourier transform schema [1]

4.9.3 *Quantum Fourier transform*

If the program can simulate controlled phase gate, the Fourier transform is also possible with sufficient computational resources. As shown in 'Measurement-based quantum computation on cluster states', this structure can be constructed from six controlled phase gates and three Hadamard gates [1]. This particular structure is of great interest as it forms a crucial part of Shor's algorithm, making implementation highly desirable. However, memory usage of implementation for this structure in conjunction with an error correction scheme would be problematic given its high requirements on the number of qubits that are simultaneously active. It is possible that the components of the structure could be broken down in order to minimise the required computational resources through the inclusion of unitary gates between controlled phase gates and the simulation of each phase gate individually, but as yet this seems difficult to envision.

5 CONCLUSIONS

Due to the very unfinished nature of the program, few conclusions can be drawn about the fault tolerance of the error correction scheme, however things can be learned from the challenges encountered while programming so that such problems can be averted in later versions of the program as well as in other programming based projects.

5.1 CHOICE OF RESOURCES

It was found that Fortran is very much an appropriate language of choice for this kind of program given its access to a wide range of linear algebra libraries and the ease of constructing modular programs. If the program were to be adapted for parallel processing, this language would continue to work well give its access to these libraries, likely in conjunction with openMPI. Otherwise, it might be better if it were converted to python due to python's better handling of complex numbers and the removal of the need to compile the program, something which would be of benefit while many values in the program are still hard coded.

5.2 MATRIX DECOMPOSITION

As a way of recovering the individual state vectors of particular qubits from the larger state of the system, matrix decomposition is incredibly unreliable. While this can be mitigated to a certain extent by intelligent use of algorithms that form canonical decompositions such as the single value decomposition or eigen-decomposition, there are likely to always be problems with certain values as the complexity of superposition for the state vector increases. Fortunately this is not too large a problem for they systems in this project as they would rarely exceed six qubits or so, but it seems like a big issue for larger systems.

It seems possible that rather than just being a mathematical challenge, the difficulty in decomposing a matrix to accurately represent individual state vectors is more of problem resulting from the foundations of quantum mechanics. The difficulty in taking apart a state vector for mixed states is a mathematical reflection of the tricky nature of superposition and is unlikely to ever be resolved easily.

5.3 FURTHER WORK

While this work is unfinished, it also has a lot of potential for extension, this includes both the planned features that were not included to the program and the inclusion of the more complex structures that have interesting implications. As a result the work for this project will be continued independently of the end of the course in order to resolve the outstanding questions.

If the program can be rebuilt into a more concise package with better modules, it may even serve as a useful tool for students interested in one way quantum computation as it provides a clear outline of the process in several easy to understand modules.

APPENDIX A: CODE

A

A.1 SINGLE CHAIN PROGRAM

Listing 3: Single qubit gate program

```
!**********************************************
!Finds "fidelity" of two states
!
!
!Can be compiled with or without BLAS
!
!
!**********************************************
program single_chain
use measurement_module
use kronecker_module
use cz_op_module
implicit none

!Precision of numerical values
integer,           parameter :: dp=selected_real_kind(15,
    300)            !IEEE 754 Double Precision

integer :: i, chain_length
complex(kind=dp), dimension(:), Allocatable :: state_in,
    state_out
complex(kind=dp), dimension(:), Allocatable :: next_qbit
complex(kind=dp), dimension(2) :: init_state

 character(len=1) :: m_type !measurement time string

real(kind=dp) :: fid_out
real(kind=dp) :: measurement_phase



open(100, file='measurements.dat', status='replace', iostat=
    ierr)
  if (ierr/=0) stop 'Error in opening file measurements.dat'
 close(100)

open(200, file='m_instructions.dat', iostat=ierr)
 if (ierr/=0) stop 'Error in opening file m_instructions.dat
    '
```

```
 chain_length = 4

!call init_random_seed


Allocate(wf_in(2))

wf_in = (1.0_dp / SQRT(2.0_dp)) * (/ 1.0_dp, 1.0_dp/)
!wf_in = (/ 1.0_dp, 0.0_dp/)

init_wf = wf_in


do i = 1, chain_length - 1

  Allocate(next_qbit(2))

  next_qbit = (1.0_dp / SQRT(2.0_dp)) * (/ 1.0_dp, 1.0_dp/)


  Allocate(wf_out(4))

  call kronecker_product_complex_vector(wf_in, next_qbit,
      wf_out)

  Deallocate(next_qbit)

  Deallocate(wf_in)
  Allocate(wf_in(4))

  wf_in = wf_out

  call cz_operation(wf_in, 2, 1, 2)

  !call add_error()

  Deallocate(wf_out)
  Allocate(wf_out(2))

  READ(200, *) m_type!, measurement_phase

  if(m_type == 'R') then
    call phi_measurement(wf_out, wf_in, 2, measurement_phase
        , 1)
  else
    call pauli_measurement(wf_out, wf_in, 2, m_type, 1)
  end if

  Deallocate(wf_in)
  Allocate(wf_in(2))

  wf_in = wf_out
```

```
  Deallocate(wf_out)

end do

 close(200)

 call feed_forward(wf_in, 1, i-1, 1)

print *, wf_in

fid_out = fidelity(wf_in, init_wf)

open(100, file="fideity.dat",iostat=ierr)
  if (ierr/=0) stop 'Error in opening file fidelity.dat'

write(100, *) i, fid_out

 close(100)

Deallocate(wf_in)

end program single_chain
```

A.2 CZ OPERATION MODULE

Listing 4: CZ Operation Module

```
!**********************************************
!
!Module containing the variable cz_operation
!creation and application subroutine
!
!In modular form to make use of multi-level allocatable
    arrays
!that are standard in Fortran 95
!
!**********************************************

module cz_op_module
use kronecker_module

contains

!**********************************************
!
!Forms the cz_operator between control and target qubits
!in state vector of n qubits
!
!Applies cz_operation to state vector and returns output
```

```
!
!**********************************************

subroutine cz_operation(state_vector, n, ctrl, trgt)
implicit none

!Precision of numerical values
integer,          parameter :: dp=selected_real_kind(15,
    300)           !IEEE 754 Double Precision

!Loop integers
integer :: i, j

!Qubit number, input from function
integer :: n

!Control and target qubits, input from function
!Determines matrix multiplication proceedure for final
    cz_matrix
integer :: ctrl, trgt

!Constant size matrices
complex(kind=dp), dimension(2, 2) :: identity
                                    !2x2 identity matrix
complex(kind=dp), dimension(2, 2) :: z_matrix
                                    !2x2 z pauli matrix

!Matrices dependant upon integer n for sizing
!Two dimensional array of size 2**n by 2**n
complex(kind=dp), dimension(2**n, 2**n) :: cz_matrix
                !Final cz_matrix

!State vector array for which size is dependant upon the
    array input in function
complex(kind=dp), dimension(:), Allocatable :: state_vector
                !wavefunction arrays

!Work matrices used in calculation of cz_matrix
!Allocatable as constantly resized in loops for kronecker
    products
complex(kind=dp), dimension(:, :), Allocatable ::
    work_matrix       !Primary variable work matrix
complex(kind=dp), dimension(:, :), Allocatable :: out_matrix
            !Loop output variable work matrix

!Typical 2x2 matrix initialisation
identity(:,:)  = 0.0_dp         !Identity matrix values
identity(1, 1) = 1.0_dp         !Identity matrix values
identity(2, 2) = 1.0_dp         !Identity matrix values

z_matrix(:,:)  = 0.0_dp         !z matrix values
z_matrix(1, 1) = 1.0_dp         !z matrix values
```

```
z_matrix(2, 2) = -1.0_dp                !z matrix values


!Initialises all values of CZ matrix to double precision
    zero
!As this matrix is made of several additions this step is
!necessary to minmise error in calculation
 cz_matrix(:,:) = 0.0_dp

!Loops over values 1 to 4 with J
!Reflective of the unitary operator which has 4 terms
do j = 1, 4

  !Allocates the secondary work matrix into the initial size
  !for Kronecker product multiplication
  Allocate(out_matrix(2, 2))

  !Performs a check to see if the first matrix in the order
      of
  !multiplication corresponds to a control or target
  !if they do, assigns the appropriate matrix dependant
  !upon the value of j.
  !Otherwise assigns the identity matrix
  if((trgt.eq.1).and.((j.eq.2).or.(j.eq.4))) then
    out_matrix = z_matrix
  elseif((ctrl.eq.1).and.((j.eq.3).or.(j.eq.4))) then
    out_matrix = z_matrix
  else
    out_matrix = identity
  end if

  !Loops over each individual qubit
  !This will produce a matrix of appropriate size for each
  !term of the unitary operator
  do i = 2, n

    !Assigns a size value to the work matrix dependent
    !on the step in the loop, thus allowing it
    !to contain the appropriate size of matrix at this step
    Allocate(work_matrix(2**(i-1), 2**(i-1)))

    !Assigns the work matrix the value of the output matrix
    !Takes the value from the output of last step
    !Allowing output matrix to be deallocated
    work_matrix = out_matrix

    !Deallocate output matrix
    Deallocate(out_matrix)

    !Allocates new size to the output matrix
    !New size is appropriate for storage of
    !Kronecker product between work matrix and a 2x2 matrix
```

```
    Allocate(out_matrix(2**i, 2**i))

    !Determines whether the next operator of the
        multiplication
    !Will be a control or target qubit, depending on the
        term of U
    !Assigns the appropriate value if so for kronecker
        products
    !Otherwise uses the identity matrix for the kronecker
        product
    if((ctrl.eq.i).and.((j.eq.3).or.(j.eq.4))) then
          call kronecker_product_complex(work_matrix,
              z_matrix, out_matrix)
    elseif((trgt.eq.i).and.((j.eq.2).or.(j.eq.4))) then
          call kronecker_product_complex(work_matrix,
              z_matrix, out_matrix)
    else
          call kronecker_product_complex(work_matrix,
              identity, out_matrix)
    end if

    !Deallocates the work matrix ready
    !For allocation in next loop
    Deallocate(work_matrix)

  !Ends do loop for the jth term of the Operator
  end do

  !Determines which term of operator the loop is on
  !If j is four, removes output from cz_matrix
  !otherwise adds output to cz_matrix
  !Reflective of signs of unitary operator
  if(j.eq.4) then
    cz_matrix = cz_matrix - out_matrix
  else
    cz_matrix = cz_matrix + out_matrix
  end if

  !Deallocates out matrix for next loop of j
  Deallocate(out_matrix)

end do

!Multiplies every term of the cz_matrix by half
!This is the first constant of the operator
 cz_matrix = 0.5_dp * cz_matrix


!If BLAS is enabled when compiling (-lblas), this code
    segment
!will be compiled into final program using BLAS functions
#ifdef lblas
```

```
 !call for double precision complex matrix vector multiply
     from BLAS
 !Applies cz_matrix operator to state vector and outputs it
 call ZGEMV( 'N', 2**n, 2**n, 1.0_dp, cz_matrix, 2**n, &
 state_vector, 1, 0.0_dp, state_vector, 1)

!If BLAS is not enabled when compiling, this section of
    intrinsic functions
!will be used instead.
#else

 !call for intrinsic matrix vector multiply
 !Multiplies Hamiltonian matrix with basis(i) to form vector
      Hi
 state_vector = MATMUL(cz_matrix, state_vector)

!ends the preprocessor if statement
#endif

!Deallocates cz_matrix as it is no longer needed
Deallocate(cz_matrix)

return

end subroutine cz_operation

end module
```

A.3 KRONECKER PRODUCT MODULE

Listing 5: Kronecker Product Module

```
!**********************************************
!
!Performs Kronecker product of two matrices A and B
    producing matrix P
!n and m correspond to the dimensions of A and x and y
    correspond to the dimensions of B
!dimensions of P are assumed to be n*x and m*y
!
!**********************************************
module kronecker_module
implicit none

!Global variables
!Precision of numerical values
integer,                        parameter :: dp=
    selected_real_kind(15, 300)             !IEEE 754 Double
     Precision
```

```
!Error reporting
integer,                          save :: ierr

            !Error integer

contains


  !**********************************************
  !
  !Performs Kronecker product of two matrices A and B
      producing matrix P
  !n and m correspond to the dimensions of A and x and y
      correspond to the dimensions of B
  !dimensions of P are assumed to be n*x and m*y
  !
  !**********************************************

  subroutine kronecker_product_complex(A, B, P)
  implicit none
  integer :: i, j, k, l
  integer :: n, m, x, y
  complex(kind=dp), dimension(:, :), Allocatable :: A
  complex(kind=dp), dimension(:, :), Allocatable :: B
    complex(kind=dp), dimension(:, :), Allocatable :: P

    n = SIZE(A, 1)
    m = SIZE(A, 2)

    x = SIZE(B, 1)
    y = SIZE(B, 2)

    do i = 1, n
      do j = 1, m

            do k = 1, x
              do l = 1, y

                P((i-1)*x + k, (j-1)*y + l) = A(i, j) * B(k,
                     l)

              end do
             end do

      end do
    end do

    return

  end subroutine kronecker_product_complex
```

```
!**********************************************
!
!Performs Kronecker product of two matrices A and B
    producing matrix P
!n and m correspond to the dimensions of A and x and y
    correspond to the dimensions of B
!dimensions of P are assumed to be n*x and m*y
!
!**********************************************

subroutine kronecker_product_complex_vector(A, B, P)
  implicit none
  integer :: i, j
  integer :: n, m, x, y
  complex(kind=dp), dimension(:), Allocatable :: A
  complex(kind=dp), dimension(:), Allocatable :: B
  complex(kind=dp), dimension(:), Allocatable :: P

  n = SIZE(A)

  x = SIZE(B)


  do i = 1, n
    do j = 1, x

           P((i-1)*x + j) = A(i) * B(j)

    end do
  end do


  return

end subroutine kronecker_product_complex_vector

!**********************************************
!
!Performs Kronecker product of two matrices A and B
    producing matrix P
!n and m correspond to the dimensions of A and x and y
    correspond to the dimensions of B
!dimensions of P are assumed to be n*x and m*y
!
!**********************************************

subroutine rank_decomposition_complex(A, C, D)
  implicit none
  integer :: i, j
  integer :: m, n, o
  real(kind=dp) :: magnitude
  complex(kind=dp), dimension(:), Allocatable :: A
```

```
complex(kind=dp), dimension(:), Allocatable :: C
complex(kind=dp), dimension(:), Allocatable :: D
complex(kind=dp), dimension(:,:), Allocatable :: B

m = SIZE(A)
n = SIZE(C)
o = SIZE(D)

Allocate(B(n, o))

magnitude = 0.0_dp
C(:) = 0.0_dp
D(:) = 0.0_dp
B(:,:) = 0.0_dp

!Perform an inverse of the vec() operator by reshaping
!vector A into a matrix of dimensions n x o
B = TRANSPOSE(RESHAPE(A, (/ o, n /)))


do i = 1, o
  do j = 1, n
    if(B(j, i) /= 0.0_dp) then
      C(j) = B(j, i)
      exit
    else
      continue
    end if
  end do
end do

!Normalise

!Calculate length
do i = 1, n
  magnitude = magnitude + C(i)*C(i)
end do

magnitude = sqrt(magnitude)


!Divide components by length
C = C / magnitude

do i = 1, o
  do j = 1, n
    if(C(j) /= 0.0_dp) then
      D(i) = (1.0_dp/(C(j))) * B(j, i)
      exit
    else
      continue
    end if
```

```
    end do
  end do

  Deallocate(B)

  return


end subroutine

!***********************************************
!
!Performs Kronecker product of two matrices A and B
    producing matrix P
!n and m correspond to the dimensions of A and x and y
    correspond to the dimensions of B
!dimensions of P are assumed to be n*x and m*y
!
!***********************************************

subroutine sv_decomposition_complex(A, C, D)
  implicit none
  integer :: i, j
  integer :: m, n, o
  complex(kind=dp), dimension(:), Allocatable :: A
  complex(kind=dp), dimension(:), Allocatable :: C
  complex(kind=dp), dimension(:), Allocatable :: D
  complex(kind=dp), dimension(:,:), Allocatable :: B

  real(kind=dp), dimension(:), Allocatable :: S
  complex(kind=dp), dimension(:,:), Allocatable :: U
  complex(kind=dp), dimension(:,:), Allocatable :: VT

  complex(kind=dp), dimension(:), Allocatable :: WORK
  real(kind=dp), dimension (:), Allocatable :: RWORK
  integer :: LWORK, LDA, LDU, LDVT

  m = SIZE(A)
  n = SIZE(C)
  o = SIZE(D)

  LWORK = MAX(1, 4*(2*MIN(n, o) + MAX(n, o)))
  LDA = MAX(1, n)
  LDU = MAX(1, n)
  LDVT = MAX(1, o)

  Allocate(S(MIN(n, o)))
  Allocate(B(n, o))

  Allocate(U(LDU, n))
  Allocate(VT(LDVT, o))
```

```
  Allocate(WORK(MAX(1, LWORK)))
  Allocate(RWORK(5*MIN(n, o)))

  C(:) = 0.0_dp
  D(:) = 0.0_dp
  B(:,:) = 0.0_dp

  S(:) = 0.0_dp
  U(:, :) = 0.0_dp
  VT(:, :) = 0.0_dp
  WORK(:) = 0.0_dp
  RWORK(:) = 0.0_dp


  !Perform an inverse of the vec() operator by reshaping
  !vector A into a matrix of dimensions n x o
  B = TRANSPOSE(RESHAPE(A, (/ o, n /)))

  call zgesvd('A', 'A', n, o, B, LDA, S, U, LDU, VT, LDVT,
       WORK, LWORK, RWORK, ierr)
    if(ierr.eq.0) then
      continue
    elseif(ierr.gt.0) then
      stop 'ZBDSQR did not converge'
    elseif(ierr.lt.0) then
      stop 'argument had an illegal value'
    end if

  do i = 1, n
    C(i) = S(i)*SUM(U(:,i))
  end do


  do i = 1, o
    D(i) = SUM(VT(i,:))
  end do

  Deallocate(S)
  Deallocate(B)
  Deallocate(U)
  Deallocate(WORK)
  Deallocate(RWORK)
  Deallocate(VT)

  return


end subroutine

!**********************************************
!
```

```
!Performs Kronecker product of two matrices A and B
    producing matrix P
!n and m correspond to the dimensions of A and x and y
    correspond to the dimensions of B
!dimensions of P are assumed to be n*x and m*y
!
!*********************************************

subroutine get_vector_complex(A, E, G, m, trgt)
  implicit none
  integer :: trgt, i
  integer :: m, n, o, p
  complex(kind=dp), dimension(:), Allocatable :: A
  complex(kind=dp), dimension(:), Allocatable :: C
  complex(kind=dp), dimension(:), Allocatable :: D
  complex(kind=dp), dimension(:), Allocatable :: E
  complex(kind=dp), dimension(:), Allocatable :: F
  complex(kind=dp), dimension(:), Allocatable :: G


  if(trgt.eq.1) then
    n = 2
    o = 2**(m - 1)
  elseif(trgt.eq.m) then
    o = 2
    n = 2**(m - 1)
  else
    o = 2**(m - trgt)
    n = 2**(trgt)
    p = 2**(trgt-1)
  end if

  Allocate(D(o))
  Allocate(C(n))

  call sv_decomposition_complex(A, C, D)


  if(trgt.eq.1) then
    E = C
    G = D
  elseif(trgt.eq.m) then
    G = C
    E = D
  else
    Allocate(F(p))

    call sv_decomposition_complex(D, E, F)

    call kronecker_product_complex_vector(C, F, G)

    Deallocate(F)
```

```
      end if




      Deallocate(D)
      Deallocate(C)


      return

    end subroutine

  end module
```

A.4 MEASUREMENT MODULE

Listing 6: Measurement Module

```
module measurement_module
  use kronecker_module
  implicit none

  !Precision of numerical values
  integer,          parameter :: dp=selected_real_kind(15,
      300)          !IEEE 754 Double Precision
  character(len=70), save :: data_format = '(A4, 2I4 , 3E25
      .16E3)'

  real(kind=dp), parameter :: pi =
      3.14159265358979323846264338327950

contains

  subroutine phi_measurement(wf_out, wf_in, n, m_phase, trgt
      )
    implicit none

    integer :: trgt !target qubit
    integer :: n

    real(kind=dp) :: rand_num !scalar variable of
    integer :: m_result
    complex(kind=dp) :: ZDOTU

    complex(kind=dp), dimension(:), Allocatable :: wf_in
    complex(kind=dp), dimension(:), Allocatable :: wf_out

    complex(kind=dp), dimension(:), Allocatable ::
        work_vector
```

```
    complex(kind=dp), dimension(:), Allocatable :: m_vector
                                        !2x2 measurement
         matrix

    complex(kind=dp) :: m_value
    real(kind=dp) :: m_phase

    complex(kind=dp), dimension(2, 2) :: x_matrix

    x_matrix(1, 2) = 1.0_dp             !Pauli x
    x_matrix(1, 1) = 0.0_dp             !Pauli x
    x_matrix(2, 2) = 0.0_dp             !Pauli x
    x_matrix(2, 1) = 1.0_dp             !Pauli x

    Allocate(m_vector(2))

    m_vector(1) = EXP(CMPLX(0.0_dp, (- m_phase) / 2.0_dp,
        kind=dp))
    m_vector(2) = EXP(CMPLX(0.0_dp, (m_phase) / 2.0_dp, kind
        =dp))

    call RANDOM_NUMBER(rand_num)
    m_result = NINT(rand_num)

    if(m_result.eq.0) then
      continue
    elseif(m_result.eq.1) then
      m_vector = MATMUL(x_matrix, m_vector)
    else
      print *, 'Error with random measurement results'
      stop
    end if

    Allocate(work_vector(2))

    call get_vector_complex(wf_in, work_vector, wf_out, n,
        trgt)


#ifdef lblas

    m_value = ZDOTU(2**n, m_vector, 1, work_vector, 1)
#else

    m_value = DOT_PRODUCT(m_vector, work_vector)
#endif

    wf_out = m_value * wf_out

    Deallocate(m_vector)
    Deallocate(work_vector)
```

```
  open(100, file='measurements.dat', access='append',
      iostat=ierr)
    if (ierr/=0) stop 'Error in opening file measurements.
        dat'

  write(100, data_format) 'R', trgt, m_result, m_phase,
      REALPART(m_value), IMAGPART(m_value)

  close(100, iostat=ierr)
    if (ierr/=0) stop 'Error in closing file measurements.
        dat'

  return

end subroutine phi_measurement

subroutine pauli_measurement(wf_out, wf_in, n, m_type,
    trgt)
  implicit none

  integer :: trgt !target qubit
  integer :: n

  real(kind=dp) :: rand_num !scalar variable of
  integer :: m_result
  character(len=1) :: m_type !measurement time string

  complex(kind=dp), dimension(:), Allocatable :: wf_in
  complex(kind=dp), dimension(:), Allocatable :: wf_out

  complex(kind=dp), dimension(:), Allocatable ::
      work_vector
  complex(kind=dp), dimension(:), Allocatable :: m_vector
                                              !2x2 measurement
       matrix

  complex(kind=dp) :: m_value
  real(kind=dp) :: m_phase
   complex(kind=dp) :: ZDOTU

  !common matrices
  complex(kind=dp), dimension(2) :: x_evector
                              !2x2 x pauli matrix
  complex(kind=dp), dimension(2) :: z_evector
                              !2x2 z pauli matrix
  complex(kind=dp), dimension(2) :: y_evector
                      !2x2 y pauli matrix
  complex(kind=dp), dimension(2, 2) :: x_matrix
  complex(kind=dp), dimension(2, 2) :: z_matrix

  x_evector(1) = (1.0_dp / SQRT(2.0_dp)) * 1.0_dp
              !Pauli x
```

```
x_evector(2) = (1.0_dp / SQRT(2.0_dp)) * 1.0_dp
            !Pauli x

z_evector(1) = (1.0_dp / SQRT(2.0_dp)) * 1.0_dp
            !Pauli z
z_evector(2) = (1.0_dp / SQRT(2.0_dp)) * 0.0_dp
            !Pauli z

y_evector(1) = (1.0_dp / SQRT(2.0_dp)) * (1.0_dp, 0.0_dp
    ) !Pauli y
y_evector(2) = (1.0_dp / SQRT(2.0_dp)) * (0.0_dp, 1.0_dp
    )  !Pauli y

x_matrix(1, 2) = 1.0_dp             !Pauli x
x_matrix(1, 1) = 0.0_dp             !Pauli x
x_matrix(2, 2) = 0.0_dp             !Pauli x
x_matrix(2, 1) = 1.0_dp             !Pauli x

z_matrix(1, 1) = 1.0_dp             !Pauli z
z_matrix(1, 2) = 0.0_dp             !Pauli z
z_matrix(2, 1) = 0.0_dp             !Pauli z
z_matrix(2, 2) = -1.0_dp            !Pauli z

open(100, file='measurements.dat', access='append',
    iostat=ierr)
  if (ierr/=0) stop 'Error in opening file measurements.
      dat'

if(m_type == 'X') then
  m_vector = x_evector
  m_phase = 0.0_dp
elseif(m_type == 'Z') then
  m_vector = z_evector
  m_phase = 0.0_dp
elseif(m_type == 'Y') then
  m_vector = y_evector
  m_phase = pi / 2.0_dp
else
  print *, 'Error selecting measurement matrix check
      subroutine input'
  stop
end if

call RANDOM_NUMBER(rand_num)
m_result = 0!NINT(rand_num)


if(m_result.eq.0) then
  continue
elseif((m_result.eq.1).and.(m_type.eq.'Z')) then
  m_vector = MATMUL(x_matrix, m_vector)
elseif((m_result.eq.1).and.(m_type.eq.'X')) then
```

```
        m_vector = MATMUL(z_matrix, m_vector)
      elseif((m_result.eq.1).and.(m_type.eq.'Y')) then
        m_vector = MATMUL(z_matrix, m_vector)
      else
        print *, 'Error with random measurement results'
        stop
      end if

      Allocate(work_vector(2))

      call get_vector_complex(wf_in, work_vector, wf_out, n,
          trgt)

      print *, m_vector, work_vector

#ifdef lblas

      m_value = ZDOTU(2**n, m_vector, 1, work_vector, 1)
#else

      m_value = DOT_PRODUCT(m_vector, work_vector)

#endif


      wf_out = m_value * wf_out

      Deallocate(m_vector)
      Deallocate(work_vector)

      open(100, file='measurements.dat', access='append',
          iostat=ierr)
        if (ierr/=0) stop 'Error in opening file measurements.
            dat'

      write(100, data_format) m_type, trgt, m_result, m_phase,
           REALPART(m_value), IMAGPART(m_value)

      close(100, iostat=ierr)
        if (ierr/=0) stop 'Error in closing file measurements.
            dat'

      return

    end subroutine pauli_measurement


    subroutine feed_forward(state_vector, n, m_number, trgt)
      implicit none

      integer :: trgt, n, m_number
      integer :: i
```

```
integer :: m_result

character(len=1) :: m_type

integer :: m_target
real(kind=dp) :: m_phase, m_value_re, m_value_im
complex(kind=dp) :: m_value

complex(kind=dp), dimension(:), Allocatable ::
    state_vector
complex(kind=dp), dimension(:), Allocatable :: trgt_wf
complex(kind=dp), dimension(2, 2) :: x_matrix
complex(kind=dp), dimension(2, 2) :: h_matrix
complex(kind=dp), dimension(2, 2) :: rz_matrix
complex(kind=dp), dimension(2, 2) :: h_rz_matrix

complex(kind=dp), dimension(:), Allocatable ::
    work_vector
complex(kind=dp), dimension(:), Allocatable :: top_wf,
    bot_wf

h_matrix(1, 1) = (1.0_dp / SQRT(2.0_dp))
h_matrix(1, 2) = (1.0_dp / SQRT(2.0_dp))
h_matrix(2, 1) = (1.0_dp / SQRT(2.0_dp))
h_matrix(2, 2) = -(1.0_dp / SQRT(2.0_dp))

x_matrix(1, 2) = 1.0_dp             !Pauli x
x_matrix(1, 1) = 0.0_dp             !Pauli x
x_matrix(2, 2) = 0.0_dp             !Pauli x
x_matrix(2, 1) = 1.0_dp             !Pauli x

Allocate(trgt_wf(2))


if(n.eq.1) then
  trgt_wf = state_vector
else
  Allocate(work_vector(2**(n-1)))
  call get_vector_complex(state_vector, trgt_wf,
      work_vector, n, trgt)
end if

open(100, file='measurements.dat', access='sequential',
    iostat=ierr)
  if (ierr/=0) stop 'Error in opening file measurements.
      dat'


do i = 1, m_number
  read(100, data_format) m_type, m_target, m_result,
      m_phase, m_value_re, m_value_im
```

```
      rz_matrix(:,:) = 0.0_dp
      rz_matrix(1,1) = EXP(CMPLX(0.0_dp, (- m_phase) / 2.0
          _dp, kind=dp))
      rz_matrix(2,2) = EXP(CMPLX(0.0_dp, (m_phase) / 2.0_dp,
           kind=dp))

      h_rz_matrix = MATMUL(h_matrix, rz_matrix)

      if(m_result.eq.0) then
        trgt_wf = EXP(CMPLX(0.0_dp, (- m_phase) / 2.0_dp,
            kind=dp)) * MATMUL(h_rz_matrix, trgt_wf)
      elseif(m_result.eq.1) then
        trgt_wf = EXP(CMPLX(0.0_dp, (- m_phase) / 2.0_dp,
            kind=dp)) * MATMUL(MATMUL(x_matrix, h_rz_matrix)
            , trgt_wf)
      endif

    end do


    close(100, iostat=ierr)
      if (ierr/=0) stop 'Error in closing file measurements.
          dat'


    if(n.eq.1) then
      state_vector = trgt_wf

    else
      Allocate(top_wf(2**trgt))
      Allocate(bot_wf(2**(n-trgt)))

      call sv_decomposition_complex(work_vector, top_wf,
          bot_wf)


      Deallocate(work_vector)
      Allocate(work_vector(2**trgt))

      call kronecker_product_complex_vector(top_wf, trgt_wf,
           work_vector)

      call kronecker_product_complex_vector(work_vector,
          bot_wf, state_vector)

      Deallocate(top_wf)
      Deallocate(bot_wf)
      Deallocate(work_vector)

    end if

    Deallocate(trgt_wf)
```

```
    return

  end subroutine

end module measurement_module
```

A.5 FIDELITY FUNCTION

Listing 7: Fidelity Function

```
!**********************************************
!Finds "fidelity" of two states
!
!
!Can be compiled with or without BLAS
!
!
!**********************************************
function fidelity(state_one, state_two)
implicit none

!Precision of numerical values
integer,          parameter :: dp=selected_real_kind(15,
    300)           !IEEE 754 Double Precision

!Function variables
integer :: n
                                                  !integer
     size of state vectors
real(kind=dp) :: fidelity
                                                  !
    Fidelity variable for function output
complex(kind=dp) :: ZDOTU

    !ZDOT variable for blas library function

!Input state vectors
complex(kind=dp), dimension(:), Allocatable :: state_one
                    !First input state vector
complex(kind=dp), dimension(:), Allocatable :: state_two
                    !Second input state vector

!Checks to see if array sizes of two state vectors match
!If they do assigns value to n for inner product calculation
!Otherwise stops program and prints error message
if(SIZE(state_one).eq.SIZE(state_two)) then
    n = SIZE(state_one)
else
        stop 'Array size mismatch in fidelity function'
```

```
end if

!If BLAS is enabled when compiling (-lblas), this code
    segment
!will be compiled into final program using BLAS functions
#ifdef lblas

fidelity = (abs(ZDOTU(n, state_one,1, state_two, 1)))**2

!If BLAS is not enabled when compiling, this section of
    intrinsic functions
!will be used instead.
#else

fidelity = (abs(DOT_PRODUCT(state_one, state_two)))**2

!ends the preprocessor if statement
#endif

!returns value of fidelity from subroutine to main program
return

end function fidelity
```

B

APPENDIX B: RESOURCES USED

The following contains details about computational resources used for the creation and running of the program.

B.1 COMPUTER USED

Processor	Intel Core i7 2600K @ 3.40GHz
Memory	8.00GB Dual-Channel DDR3 @ 802MHz
Motherboard	ASUSTeK Computer INC. P8P67 PRO
Disk Drive	931GB SAMSUNG HD103SJ
Host Operating System	Windows 8.1 Pro 64-bit
Guest Operating System	Linux Mint 13 Maya

B.2 COMPILER SETTINGS

The code was compiled using the gfortran compiler using flags -O3 -lblas and -llapack. -g was used for debugging. BLAS was used for matrix vector multiplication and inner product calculation.LAPACK was used for single value decomposition.

[1] R. Raussendorf, D. E. Browne, and H. J. Briegel, "Measurement-based quantum computation with cluster states," arXiv e-print quant-ph/0301052, Jan. 2003. Phys. Rev. A 68, 022312 (2003).

[2] J. Joo and D. L. Feder, "Error-correcting one-way quantum computation with global entangling gates," arXiv e-print 0908.0768, Aug. 2009. PHYSICAL REVIEW A 80, 032312 (2009).

[3] H. J. Briegel, D. E. Browne, W. DÃŒr, R. Raussendorf, and M. V. d. Nest, "Measurement-based quantum computation," arXiv e-print 0910.1116, Oct. 2009. Nature Physics 5 1, 19-26 (2009).

[4] H. J. Briegel and R. Raussendorf, "Persistent entanglement in arrays of interacting particles," arXiv e-print quant-ph/0004051, Apr. 2000.

[5] J. Joo, E. Alba, J. J. GarcÃa-Ripoll, and T. P. Spiller, "Generating and verifying graph states for fault-tolerant topological measurement-based quantum computing in 2d optical lattices," arXiv e-print 1207.0253, July 2012.

[6] N. Kiesel, C. Schmid, U. Weber, O. Guehne, G. Toth, R. Ursin, and H. Weinfurter, "Experimental analysis of a 4-qubit cluster state," arXiv e-print quant-ph/0508128, Aug. 2005. Phys. Rev. Lett. 95, 210502 (2005).

[7] P. Walther, K. J. Resch, T. Rudolph, E. Schenck, H. Weinfurter, V. Vedral, M. Aspelmeyer, and A. Zeilinger, "Experimental one-way quantum computing," *Nature*, vol. 434, pp. 169–176, Mar. 2005.

[8] R. Jozsa, "An introduction to measurement based quantum computation," arXiv e-print quant-ph/0508124, Aug. 2005.

[9] R. Raussendorf and H. J. Briegel, "A one-way quantum computer," *Physical Review Letters*, vol. 86, pp. 5188–5191, May 2001.

[10] R. Raussendorf and H. J. Briegel, "Quantum computing via measurements only," arXiv e-print quant-ph/0010033, Oct. 2000.

[11] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[12] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.

[13] "Top500 list - june 2014 | TOP500 supercomputer sites."

[14] J. Children, "Spin chains, hyper-fine spin interactions and magnetic spin coupling in quantum dots," Master's thesis, University of York, York, May 2013.

[15] A. Hagar, "Quantum computing," in *The Stanford Encyclopedia of Philosophy* (E. N. Zalta, ed.), spring 2011 ed., 2011.