Optimising Quantum Error Correction Codes Using Projective Simulation:
Masters Report

Candidate Number:
19132099

September 10, 2020

Abstract

Noisy Intermediate-scale Quantum (NISQ) technology relies on Quantum Error Correction (QEC) to overcome its limitations. While the surface code has been excellent at this, being able to optimise it further to increase thresholds or lower the qubit overhead is a constant area of research. One approach is to train a machine learning algorithm to search the space of possible codes. In this paper we use a projective simulation agent to try to lower the error rate of Z, X and combined channels. We find that the X and combined channels can be lowered through this method, with 2 qubits added to the initial starting code.

Contents

1 Introduction

In this paper we will attempt to use machine learning techniques to optimise Quantum Error Correction (QEC) surface codes. Before we get to this we will first need to discuss any relevant understanding needed that was not covered in the literature review, which has been added as an appendix. To begin with we will describe some more relevant information on surface codes. This is necessary to better formalise the latter parts of this project. We will then discuss the specific branch of machine learning used, Projective Simulation(PS), and why it is used in this application. Lastly we will briefly outline the SQUAB software and how it simulates noise and errors in a surface code. We will also briefly discuss the merits and limitation of erasure channels.

1.1 Surface Codes

In appendix C.2.3 we discuss stabilisers and how a surface code could be set up, here we will go a bit further and discuss exactly how X and Z errors are detected, how they can be corrected, and lastly how a logical qubit can be encoded into this framework.

1.1.1 X and Z errors

[1]

As mentioned in appendix, C.3, a surface code can be thought of as a lattice superimposed onto a section of the surface area of a torus. Within the lattice exist data and measurement qubits organised into a grid, fig(19). Each data qubit is connected to 2 measure-Z and 2 measure-X qubits. Each of the measure qubits is linked to 4 data qubits in turn. These qubits are cycled through a specific sequence of gate operations. The measure-Z qubits have 4 controlled-not (CNOT) gates applied to them with the targets being themselves and the next nearest neighbour data qubits being the controls. This yields a an eigenstate of:

$$\hat{Z}_a\hat{Z}_b\hat{Z}_c\hat{Z}_d\left|\psi\right\rangle = Z_{abcd}\left|\psi\right\rangle \quad \text{with eigenstates} \quad Z_{abcd} = \pm 1 \tag{1}$$

For the measure-X qubits the setup is similar, except that there are two Hadamard(H) gates applied, one before the CNOTs and one after. The H gates are applied to the measure qubits, with the CNOTs applied to the nearest neighbour data qubits with the measure as control. This leads to eigenstates of:

$$\hat{X}_a\hat{X}_b\hat{X}_c\hat{X}_d\left|\psi\right\rangle = X_{abcd}\left|\psi\right\rangle \quad \text{with eigenstates} \quad X_{abcd} = \pm 1 \tag{2}$$

After this measurement the cycle is repeated. An important property of these stabiliser codes is that they don't operate from the ground state, but from a state $|\psi\rangle$ that is the result of all concurrent measurements called a quiescent state. This state is randomly selected after each full surface code cycle.

For single qubit errors such as bit flip or phase flip errors, the measurement at the end of the cycle will be different to its intended result. We can categorise this as:

$$\left|\psi\right\rangle \rightarrow \left|\psi'\right\rangle = (\hat{I}_a + \epsilon\hat{Z}_a)\left|\psi\right\rangle \tag{3}$$

This causes a sign change in the measurement of the two measure-X qubits adjacent to qubit a. This is categorized as a phase flip or Z error. If this phase flip is detected, one could then apply another Z operator to this qubit to fix it, but this operator could also be erroneous, thus this is not the preferred solution to such an error. In-fact, such errors are usually just noted and then fixed later in software. This is usually how both phase and bit flip errors are fixed, with software solutions rather than another application of a gate.

Errors of measurement would be demonstrated by change in sign for that measure only. That is, for the next cycle after the error it would be corrected. Thus multiple cycles must be run in order to catch the single as well as sequential measurement errors.

1.1.2 Encoding logical qubits

Being able to fix errors isn't useful without having logical qubits that can actually use the algorithms we want qubits to explore. There are ways of encoding a logical operator that just use the pre-existing surface code. However these require that the whole of the lattice service the one qubit. This works for small systems but we hope to expand to over 50 physical qubits, which would be a very inefficient use of the qubits, with codes like Shors code only needing 9 qubits, although at lower thresholds [2]. Instead of building array-crossing chains of X and Z operators, we will instead create holes in the surface code to act as new boundaries for the qubits. These are often referred to as defects in other literature.

In fig(1) we see how this can be done. By turning off a measure-Z qubit we can create what is called a single Z-cut hole. There also exists the analogous X-cut qubit, which is achieved by turning off a measure-X qubit.

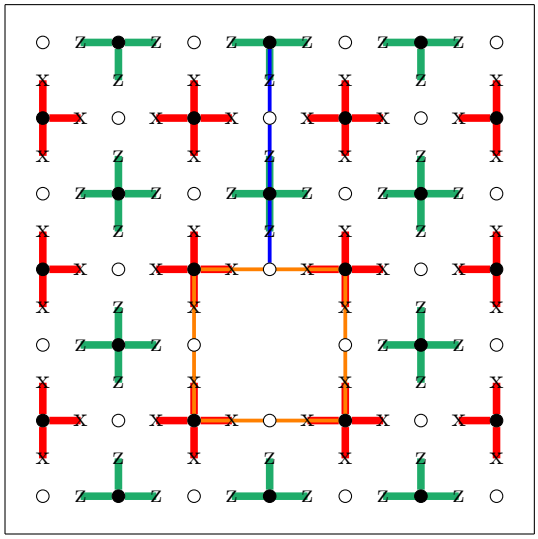


Figure 1: This is a single Z-cut qubit, which will be the type used in our experimentation. The array boundary is represented by the solid border around the Code. The Blue line represents an X logical operator and the orange line the Z logical operator, with a shared data qubit between these lines. For details on the rest of the code, see appendix C.3

1.2 Projective Simulation

[3] [4]

Here we will discuss the type of machine learning algorithm we will employ. The agent is referred to as a Projective Simulation(PS) algorithm. Like most agents it employs sensors through which it perceives its environment and actuators through which it acts on this environment. Projective simulation uses a stochastic agent, which uses information from previous percepts to determine the probabilities of what the next actuator move would be, as opposed to a deterministic one, which relies solely on its history to determine its next move.

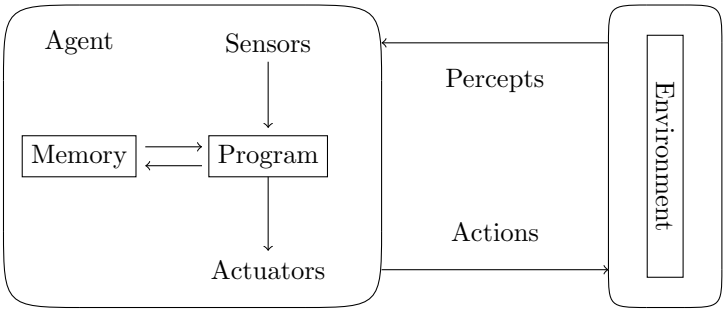


Figure 2: The agent interacts with its environment through percepts, which are a representation of the environment. These percepts are then fed back to the program through the sensors, the agent can then deliberate through memory and output an action to the environment.

In-fact this ability to learn from previous percepts extends to fictitious experience. This projective simulator, along with an episodic memory system, allowed the agent to project itself into conceivable situations. It achieves this by a random walk through past percepts, which is evaluated before leading to an action.

The episodic memory can be described as a probabilistic network of clips. Each excited clip calls its neighbouring clips with the probabilities determined by previous experience of percept history, as mentioned before. The weighting for these transition probabilities between clips is given by:

$$p_{ij}^{(t)} = \frac{e^{\beta h_{ij}^{(t)}}}{\sum_{k=1}^{M_i} e^{\beta h_{ik}^{(t)}}} \tag{4}$$

Here $p_{ij}^{(t)}$ is the transition probability between percept s_i and action a_j, β is the softmax parameter and h is a matrix element, referred to as a time dependent weight, which is associated with each edge. These h-values can also be categorized by the following equation:

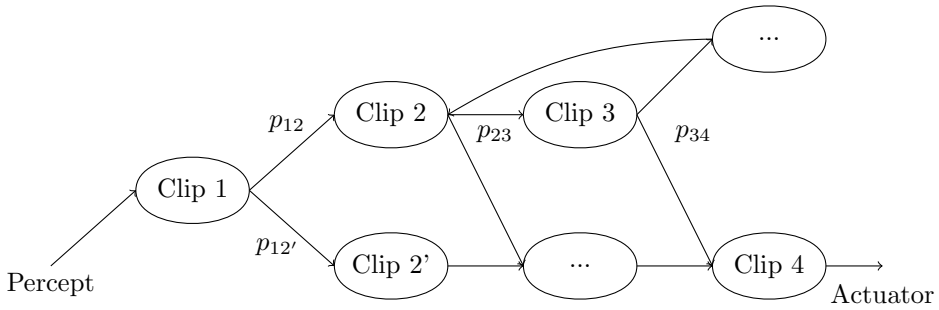


Figure 3: Diagram showing how an agent might move through clips. Note the transition probabilities from eq(4) and also the similarity to a Markov chain as fig(26)

$$h^{(t+1)} = h^{(t)} + \lambda^{(t)} g^{(t)} + \gamma(1 - h^{(t)}) \tag{5}$$

Here g is the glow matrix that redistributes rewards λ to past experiences. Experiences further in the past are awarded a decreasing fraction of this. The g matrix is initially all zero but will be updated at each interaction step by the following procedure;

$$g^{(t+1)} = (1 - \eta) g^{(t)} \tag{6}$$

This concludes our description of Projective Simulation, next we will briefly discuss the SQUAB algorithm that will be at the heart of this project.

1.3 SQUAB

[5]

SQUAB is a linear-time benchmarking tool for generalized surface codes. It tests a given surface code to see if it can correct for errors and to show how much or little overhead each logical qubit would require. Traditionally this type of software uses a depolarising channel, but SQUAB makes significant gains in performance by using an erasure channel.

Erasure channels pick a qubit and subject it to I, X, Y or Z Pauli errors. For most experiments, the location of an error can be determined and measured, so knowing which qubit is being affected is well substantiated.

Codes that work poorly on erasure channels work poorly on depolarised channels, so even if the erasure channel is a simplification, it can still be used to weed out architectures that are unlikely to perform in real world scenarios with vastly reduced computational time.

2 Setup

Here we will talk about how the software is set up with this project. We will start with how the SQUAB algorithm works in practice, and then move on to the Projective Simulation code that was used. After that we will combine the two and explain how the environment was set up and how percepts and actions are processed. This section assumes knowledge of some programming language, with extra attention being given to Python.

2.1 Working with SQUAB

The SQUAB algorithm is run through a command line interface after compilation. Commands can be passed to the program one at a time or in one long command with spaces in between. The software allows the set up of surface code quite simply. To set up the code one inputs;

```
Tiling X Y Z
```

This produces a surface code with X links in a size of Y by Z, if we take x=4, y=3 and z=3 we get outputs seen in fig(4), shown below;

This is a quick and easy setup, with there being 24 edges, see description of fig(4), there are 24 physical qubits. However the SQUAB algorithem does not allow for encoding of one logical qubit to a full surface map, but instead requires a hole to be made through an edge. We do this by opening a face, or removing a Z stabiliser, with the following codes;

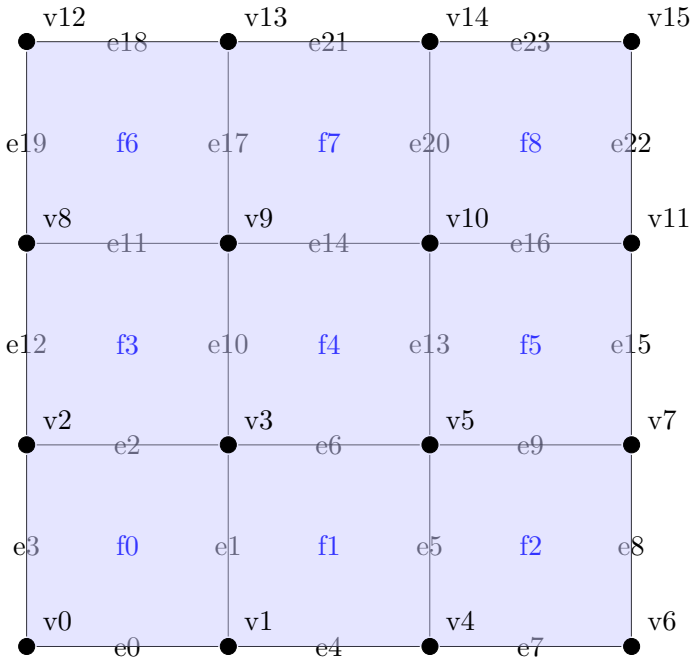


Figure 4: The simplest form of a surface code from SQUAB. Here edges represent physical qubits, denoted by e0-e23. The verticies represent the X type stabilizers and the faces represent the Z type. The faces are denoted by f0-f8. Thus in total we have 24 physical qubits with 16 X stabilisers and 9 Z stabilisers.

```
Hole f
HoleList k f1 ... fk
```

Where k is the number of holes we want to create and f1 to fk are the faces we want to remove. If we remove f4 to create a logical qubit, we have the starting point of our algorithm;

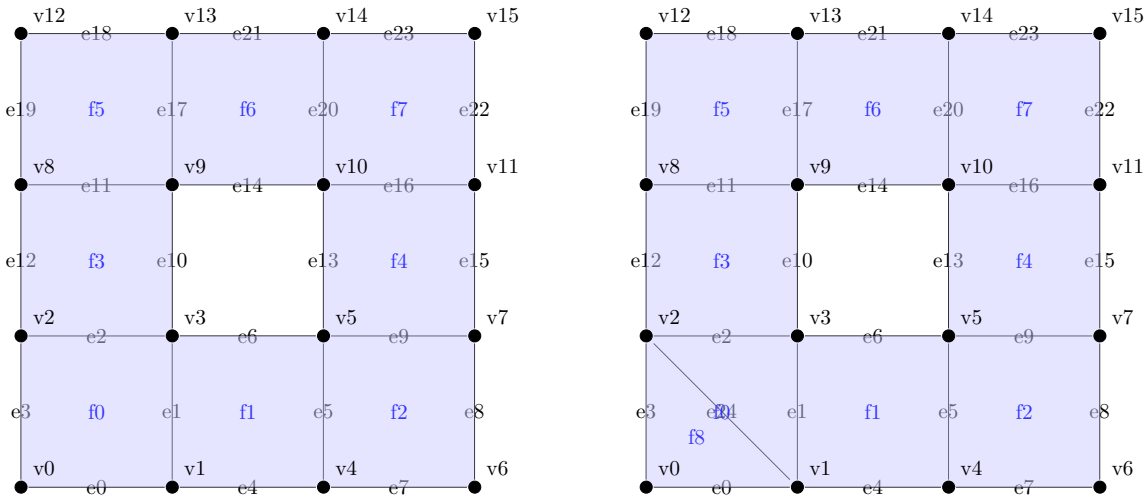


Figure 5: Our surface code with a Z stabiliser removed and a logical qubit now in the centre. This gives us 24 total physical qubits with 16 X and 8 Z stabilisers. This gives us the overhead of 24.

With this set up we can now move onto how we test a given surface code. We can do this with the following command;

```
Report pmin pmax step numtrial
```

Where pmin and pmax are the probabilities of erasure from min to max, the step is how small the increments in erasure are and the numtrials is how many attempts on each qubit the code will run. We run this with the following values;

```
Report 0 0.3 0.01 10000
```

This is the test that we will perform on each of our surface codes. The SQUAB program then outputs a helpful PDF file that contains the graphs. The results for surface code from fig(5) are shown below;

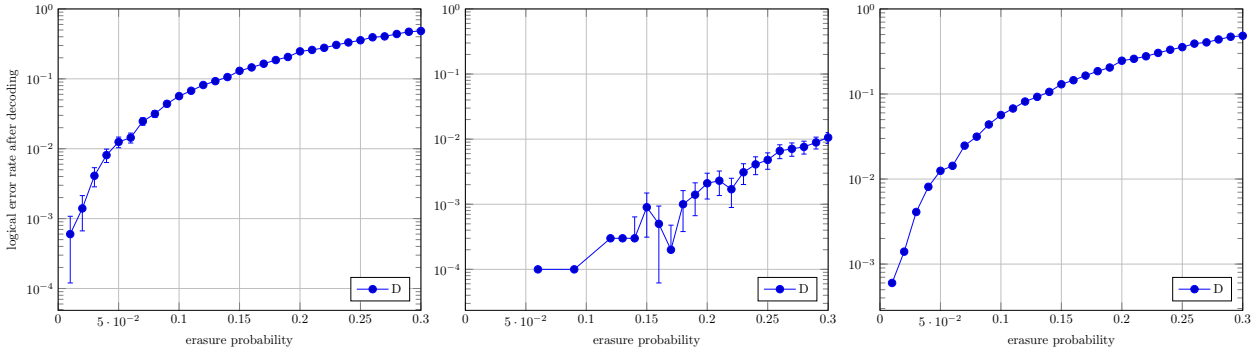


Figure 6: The Report command results for the lattice shown in fig(5)a. Left: The Combined error rate, Center: Z-logical error rate after decoding. Right: X-logical error rate after decoding. Graphs are plotted with logical error rate after decoding vs erasure probability.

These graphs wont be discussed in too much detail here, nor will the results, but its good to show how the results are collected for later use. The last action that we will need to discuss is how to add an edge, which is done by adding a face. The command here is similar to the remove face command;

AddFace k v1 . . . vk

Here k is the number of verticies that the new face will go across and v1 to vk are the verticies listed. We will be splitting a face to add a physical qubit. This is usually given by using values of k=3 and the three verticies that we want to split. If we give the command;

AddFace 3 0 2 3

We get the surface code shown in fig(5)b out. The "*Report*" results for such a code are listed in the next section, where the comparisons between the two will be discussed.

2.2 Combining all the elements

With the building blocks of the project discussed we can now turn our attention to exactly what we will be doing. First we should discuss how we will be improving the surface codes. In a paper [6] researchers outlined a method for cutting two non-neighbouring verticies and adding an extra Z stabiliser and physical qubit. This is exactly the move shown in the previous section. Changing the architecture of the surface code allows it to more easily adapt to different circumstances, ie if there are more Z or X errors. The results of running the same report as earlier are shown below;

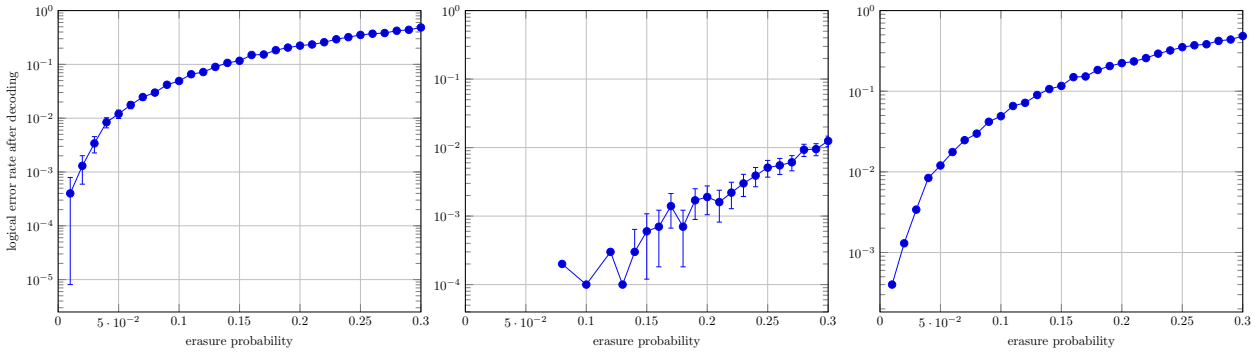


Figure 7: The Report command results for the lattice shown in fig(5)b. Left: The Combined error rate, Center: Z-logical error rate after decoding. Right: X-logical error rate after decoding. Graphs are plotted with logical error rate after decoding vs erasure probability.

The error rate we are interested is the final one with the highest erasure probability, the results are compared in a table 1.

Here we can see that the error rate has actually been raised. This is to be expected as not all combinations will be positive ones, hence the need for a comprehensive search. This leads onto why we would want to use Projective Simulation.

	Combined	±	Z errors	±	X errors	±
25 Qubits	4.8526e-01	0	1.2500e-02	2.1776e-03	4.8333e-01	0
24 Qubits	4.8357e-01	0	1.0600e-02	2.0072e-03	4.8309e-01	0

Table 1: The comparison of the initial 24 qubit code with a hole in it, with the 25 qubit code where one of the faces has been split.The results are taken from the final points in the graphs where erasure probability is highest.

We start our trial by initialising the code seen in fig(5)a. This is a 24 qubit starting point, from this we add a qubit, as in fig(5)b. A single trial of our test can include up to 40 QEC codes, the basic moves are quite generic but grow exponentially if the initial size of the code is increased. In order to search through these codes we can employ PS to search through moves that might initially increase the error rate to lower it later.

2.3 Projective Simulation Python implementation

Python was used as a coding language for this project due to its accessibility and although Python isn't the fastest language, the speed of the code wasn't a limiting factor in this case. The basis for the code used was written by Alexey Melnikov[4]. The agent used from this code is the PS_Basic. The agent required some modifications but the environmental set up was completely reworked to fit the parameters of this project. To begin with we initialise the environment with a few parameters;

```
def __init__(self):
        self.desired_outcome = 0.45
        self.current_dimensions = "D"
        self.no_moves = 0
        self.possible_moves = np.array([[0,0,2,3],[0,2,0,1]
        ,[0,1,3,5],[0,3,1,5],[0,4,5,7],[0,5,4,6],[0,5,10,11]
        ,[0,10,5,7],[0,2,8,9],[0,8,2,3],[0,8,12,13],
        ,[0,12,8,9],[0,9,13,14],[0,14,15,11],[1,0,2,3]
        ,[1,2,0,1],[1,1,3,5],[1,3,1,5],[1,4,5,7],[1,5,4,6],
        [1,5,10,11],[1,10,5,7],[1,2,8,9],[1,8,2,3]
        ,[1,8,12,13],[1,12,8,9],[1,9,13,14],[1,14,15,11]])
        self.num_actions = self.possible_moves.size
```

The moves here could have been created with a *for* loop, but for simplicity they have just been hard coded into a selection process. Here we have set the desired outcome, which is the value for each trial to try to achieve. The *self.no_moves* counts the total number of moves, and will cut out if all of them have been used up. The parameter D for the initial state of the system corresponds to the code shown in fig(5)a. Next, we come to the reset command, of which there are two:

```
def reset(self):
        """resets x and y size to initial values"""
        self.current_dimensions = "D"
        self.no_moves = 0
        return self.current_dimensions

def reset_actions(self):
        self.possible_moves = np.array([[0,0,2,3],[0,2,0,1]
        ,[0,1,3,5],[0,3,1,5],[0,4,5,7],[0,5,4,6],[0,5,10,11]
        ,[0,10,5,7],[0,2,8,9],[0,8,2,3],[0,8,12,13],
        ,[0,12,8,9],[0,9,13,14],[0,14,15,11],[1,0,2,3]
        ,[1,2,0,1],[1,1,3,5],[1,3,1,5],[1,4,5,7],[1,5,4,6],
        [1,5,10,11],[1,10,5,7],[1,2,8,9],[1,8,2,3]
        ,[1,8,12,13],[1,12,8,9],[1,9,13,14],[1,14,15,11]])
        return self.possible_moves
```

The first of thes takes the code back to its default state, with the D code structure and the number of moves made reset also. The second of these resets the actions back to all the possible moves. This is important as a little later in the program the list of possible moves is decreased after each move. This is done to avoid adding a second face to the same position, a move that would cause the SQUAB algorithem to fault. The last piece of code to be shown is the most complicated, it is the move command, which makes

a move on the part of the agent. This takes an action from the action clips, and returns a percept that is the current state of the code:

```
def move(self, move):
        print("action",move)
        action = self.possible_moves[move]
        np.delete(self.possible_moves,move,0)
        command_load = ("Load " + str(self.current_dimensions) + " ")
        if action[0] == 0:
                command_face = "Draw "
        else:
                command_face = "DrawDual "
        command_add = ("AddFace 3 " + str(action[1]) +
         " " + str(action[2]) + " " +str(action[3]) + " ")
        if self.no_moves == 20:
                save_name = str("B") + str(move)
        else:
                save_name = str(self.current_dimensions) + str(move)
        command_save = ("Save " + str(save_name) + " ")
        command_report = "Report 0 0.3 0.01 10000 "
        command_quit = "Quit"
        total_input = str(command_load) + str(command_face)
        + str(command_add) + str(command_save)
        +str(command_report) + str(command_quit)
        total_input_as_string = str.encode(total_input)
        squab = ["./squab"]
        run_squab = Popen(squab, stdin=PIPE, stdout=PIPE, stderr=PIPE)
        run_squab.stdin.write(test_input_as_string)
        std_out,std_err = run_squab.communicate()
        if run_squab.returncode != 0:
                run_squab.stdin.close()
                run_squab.stdout.close()
                run_squab.stderr.close()
                run_squab.wait()
                print("bad code")
                reward = 0
                step_finished = False
        else:
                run_squab.stdin.close()
                run_squab.stdout.close()
                run_squab.stderr.close()
                run_squab.wait()
                squab_outcome = squab_read(self.current_dimensions)
                self.current_dimensions = save_name
                print("current dim", self.current_dimensions)
                print(squab_outcome[0,1],self.desired_outcome)
                if squab_outcome[0,1] <= self.desired_outcome :
                        reward = 1
                        print("Target Reached")
                        step_finished = True
                else:
                        reward = 0
                        step_finished = False
        self.no_moves += 1
        if self.no_moves == 37:
                print("max moves reached")
                step_finished = True
        print("move number",self.no_moves)
        return self.current_dimensions, reward, step_finished
```

There is a lot to unpack here, so let us start with the inputs. The move function takes a value move. This is determined in another part of the program in the agent file, but it is a random choice over the

current size of the possible moves. This random choice is weighted through the percept deliberation process mentioned in section 1.2. This number is then input into the move command, and an action is chosen from the list of possible moves, with the action then deleted from the list of possible moves. The action is a list of 4 integers, of which the first is a choice over where the code is applied, the normal or dual lattice. The next three determine the verticies to be cut.

The function then starts creating a command to be input into the SQUAB program. It first chooses to draw the dual or normal lattice, then *command_add* is the alloted face that will be split. Next it truncates the length of the name of the algorithem, this is needed for simulations of multiple codes. Then it creates a save and a command to benchmark the code. Finally, the SQUAB program is called and the command is input and left to run.

The code then checks to see if there was a simulation error, this is highly likely with these codes, as will be seen in the next section. If the code runs without a segmentation fault, then a seperate file used to read the data from the text files is called. The value of *squab_outcome* can be updated with different indicies to account for combined, Z or X errors, currently it is set to combined.

The next step decides if the code is worth of a reward, and if the code should stop and move onto the next trial. Finally the code increments the total number of moves, and if the max has been reached, finishes the trial without reward. The code returns the current dimensions of the code, which is simply the list of all the codes that have been applied in order. This is a representation of the current surface code, as dictated earlier in the paper. This is the percept that is then fed through into memory, along with the rewards and whether to continue the trial.

A word on the percept setup before we continue to results, each one must be a representation of the current enviroment, in this cast the surface code. For this paper this takes the form saving the 4 numbers of the action to the end of the previous name for the surface code. This provides the necessary information for the percept.

3 Results

With there being three separate types of error, Z, X and combined, it is tempting to try to develop a model for each of these. This is what the researchers in a recent study did [7], which is what this paper attempts to follow. We will start with the results for the combined errors. They are shown as the surface code in fig(8) and the results in table 1.

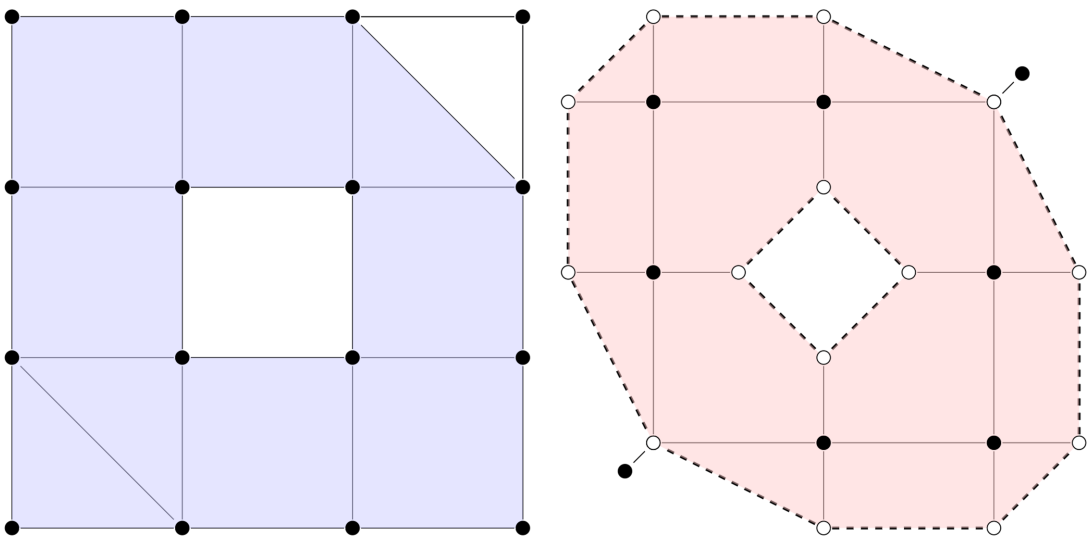


Figure 8: The code structure of our most successful agents work. 2 extra qubits have been added at the top right and bottom left. The image on the left is of the normal lattice, and the right image is its dual.

The amount of added qubits was only 2 for this trial. If we compare to the results seen earlier we do see a slight improvement over the simplest tiling.

The ideal setup from the algorithm we have discussed is given in fig(8). It adds a Z stabilising qubit at the top right and bottom left of the code. It also minimises the X-measured errors and has a much lower Z than the 25 qubit code, however still falls short of being better than just our default 24 qubit code for the measure-Z errors. The graphs of how these change with erasure probablities are shown below in fig(9)

This is in stark contrast to the results achieved in the study mentioned, which are detailed in fig(10). For the same problem researchers found that about 15 added qubits was best, with the ideal surface code

	Combined	±	Z errors	±	X errors	±
25 Qubits	4.8526e-01	0	1.2500e-02	2.1776e-03	4.8333e-01	0
24 Qubits	4.8357e-01	0	1.0600e-02	2.0072e-03	4.8309e-01	0
26 Qubits	4.6034e-01	0	1.1800e-02	2.1165e-03	4.5851e-01	0

Table 2: The comparison of all three of the tilings presented so far. The methodology is the same as from Table 1.

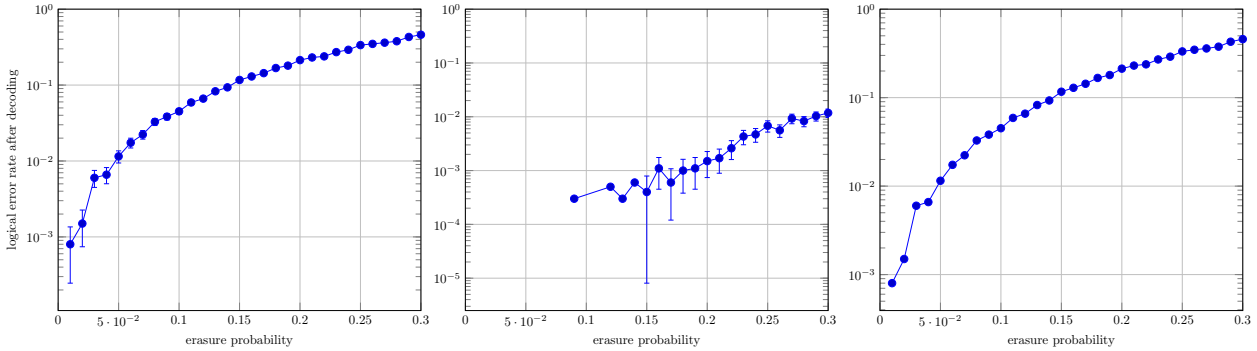


Figure 9: Left: The combined logical error rate, Center: Z-logical error rate after decoding. Right: X-logical error rate after decoding. Graphs are plotted with logical error rate after decoding vs erasure probability.

shown. One reason for this is the high probability of the SQUAB algorithm experiencing a segmentation fault, which drastically lowered the amount of actual workable moves to about 4. This problem is further personified if we look at the results of the other two possible lines on inquery. For our results we found the same code in fig(8) was the best possible code for the measure-X errors, and the base code the best for measure-Z errors. We can compare this to the results the reasearchers achieved, seen in fig(11).

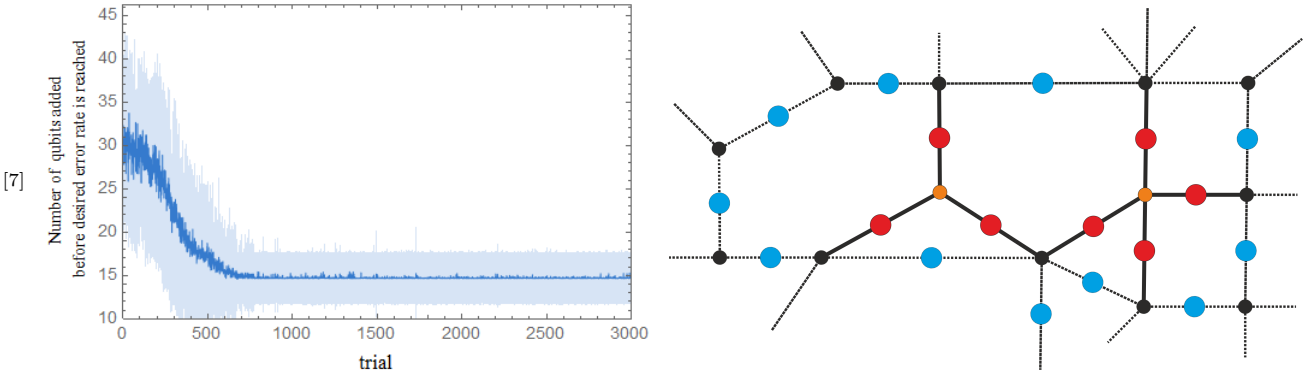


Figure 10: On the left we see the average number of added qubits from the researchers for a combined error channel, and on the right we see a part of the surface code that gave the best results, The shaded areas are the standard deviation. We can see that the flawed plaquettes in orange, with the qubits effects by high X error probabilities in red. The blue dots are physical qubits or edges in our code, and the black dots are the verticies or non errored plaquettes.

The researchers found that for an X-error based test that only 4 extra qubits would need to be added on average, and for Z based, 15 would need to be added. That is a difference of 2 and 15 qubits to our code respecitively.

We will now go into a short discussion on what has been learned and what can be improved upon.

4 Concluding remarks

While it has been shown that reinforcement learning techniques such as Projective Simulation can be used to optimise QEC surface codes [8] [7], and that these codes can be extended past simulations to real world results, this has not been extensively verified here. There were some gains from using SQUAB and PS to find lowered errors for X and Combined Errors, but none for Z.

So where did our code go wrong? It is difficult to find exact information on what commands were being

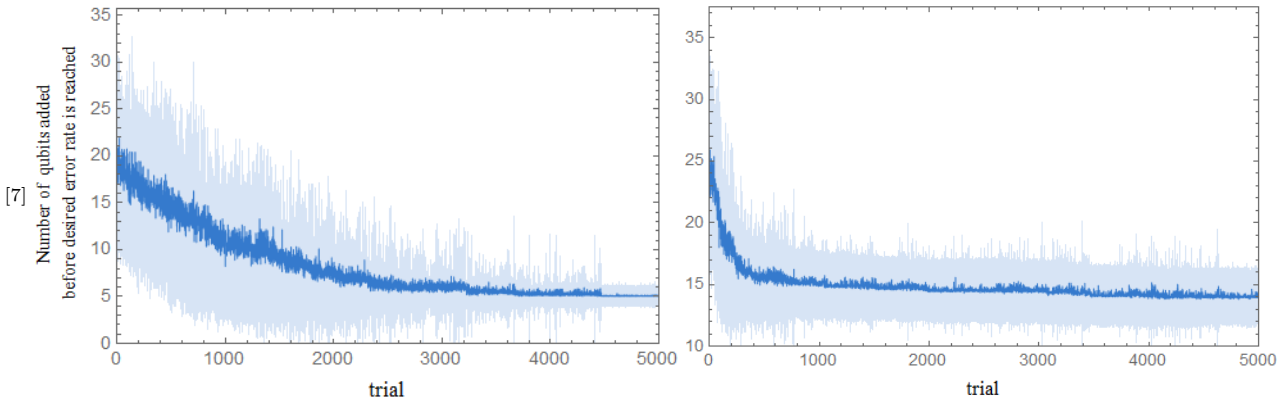


Figure 11: On the left we see the results from trying to lower the Measure-X error rate, and on the right we see the measure-Z. The shaded areas are again the standard deviation.

passed to SQUAB, infact setting up an initial 3x3 surface code like in the paper quoted was exceedingly challenging. The methodology in this paper settled at a 24 qubit solution, and while the researchers mention that the starting point is arbitrary, it might account for some of the changes to the results. For their measure X tests the researchers added 4 qubits, which would bring the total to 23, although the code structure was significantly different, as seen in fig12. Clearly there was a great difference in how the deformations were applied between this paper and the study referred.

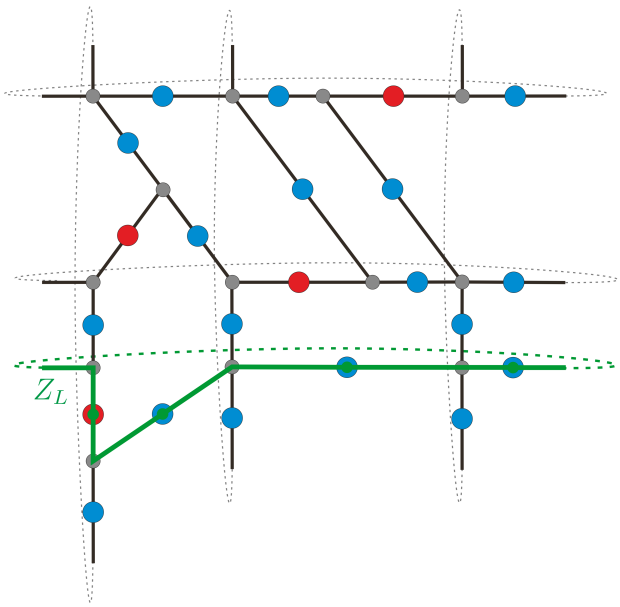


Figure 12: Here we see the researchers surface code from their best agent when searching for X errors, as in fig(11). The red dots are the added qubits here with the blue dots representing the initial 18 qubit codes. The green line represents a possible path for a Z_L string.

In order to give an output through SQUAB a logical qubit must be encoded onto the system. In our example this was done through the use of a Z-cut qubit. However there was no mention of what type of logical operator was used in the paper cited. It is possible that they omitted to encode one and let one form naturally, with those that did not produce an operator being weeded out failing to reduce a non-existant error rate. This, however, was also not mentioned.

While improvements could, and should, be made to enviromental controls to SQUAB, the part of the algorithm dealing with agent control and the execution of PS was adapted correctly. The agent has the ability to deliberate over actions, store percepts for future use and to use them with variable probabilities. If the SQUAB codes could be improved, the results could be applied to working quantum processors. It has been shown to be possible to transfer the trained agent into such a situation [7].

Were these improvements made, the extension of such a code to larger qubit structures is a natural progression. Although computing time would be increased, it would still be significantly shorter than using traditional searches without erasure channels.

Appendix

A Literature Review

Quantum computers are currently unstable and noisy.[9] Luckily we can use Quantum Error Correction(QEC) to mask these issues. This field of research is both extensive and complicated. In this review we will go through a number of ways that they have been implemented, and then asks the question, can computers do better?

Machine learning has been a hot topic in recent times, and applying it to QEC codes is not new[8;7]. However in this project we will hope to let an algorithm choose its own code to apply to the correct situation. In order to reach this lofty ideal we will have to be familiar with many different types of code, as well as types of machine learning algorithms.[3]

The aims of this literature review will be to give an underpinning to these codes and algorithms, as such we will go through;

- **Quantum Computing Physics** What is a quantum computer, and how does the physics work out? Why would we use such a device over a normal computer.
- **Quantum Error Correction** Why do we need QEC for quantum computers? how can we fix these problems?
- **Machine Learning** Why would we want machines to pick the code, and if they did, how would the go about it? We discuss types of codes and their uses here.

B Quantum computing round-up

[10]

Before we begin on the physics underpinning a quantum system, we should first discuss why we would use these systems to begin with. Traditional computers have been revolutionary in their abilities to compute complex mathematical problems vastly faster than any human could. However they are only able to solve functions classically, that is, without superpositions of states. Furthermore, large data sets are linearly difficult to input into classical computers. This means that for a larger data set, a similarly larger processor and memory are needed. While computers advance at a steady rate, and at least for large data centres, Moore's law holds true. Despite this some calculations are still far beyond them[11] and would require many years before they could be viable, if they ever were.

This is where quantum computers come in. They have the potential to scale non linearly with the number of qubits available. It has been theorised that a system of 52 qubits could hold data of 2^{52} bytes. This data could also be read and calculated with all at once. This kind of computing power is also further aided by the ability of a quantum computer to innately be able to give outputs in a superposition of states, sometimes to its detriment. This gives the tantalising possibility of being able to model quantum systems. This advantage has recently been proven by Google.[12]

With these possibilities understood, let us now begin looking into the building blocks of quantum computation, the quantum bit, or qubit.

B.1 Quantum bits

In traditional computing a bit is defined as either 1 or 0. This can take the form of a mathematical concept, or a physical bit in a computer. In much the same way we can define a quantum bit mathematically, and then show how this can apply to a real world object. However the maths behind this "qubit" is much more complicated than just being on or off.
Just as a classical bit has states 0 and 1, so too does a qubit have state $|0\rangle$ or $|1\rangle$. The notation around these states is referred to as "Dirac Notation". The difference between the classical state and this new notation is that they can also be in a superposition of both of these states. Or rather a linear combination of parts of them. This can be written as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{7}$$

Here α and β are complex numbers that can be thought of as amplitudes for the states. Together these two states form an orthonormal basis for this vector space and are known as computational basis states.

In a traditional computer we can observe whether our bit is in a 1 or a 0, and this is how we read stored data. However we cannot determine the state of our qubits by observation, that is we cannot determine α or β independently. The probability for our qubit to be in either state is 1, and the expectation value of our qubit leads to $|\alpha|^2 + |\beta|^2 = 1$.

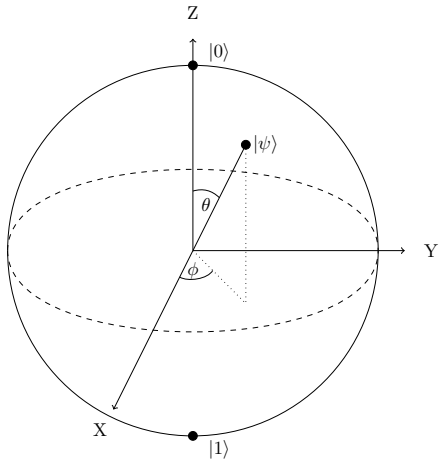
From this we can normalise our qubit to;

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{8}$$

Here we have a 50% chance to be in either states 0 or 1 when it is measured. Before being measured, it will be in a superposition of both of these states.
Another way we can rewrite eq(7) is by using $|\alpha|^2 + |\beta|^2 = 1$ to show that;

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + \exp^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{9}$$

here θ and ϕ are polar coordinates in a sphere. This is shown in fig(13) and is often referred to as a "Bloch Sphere". From it we can see that the number of possible states between $|0\rangle$ and $|1\rangle$ is infinite. However, upon measurement these states collapse into either 0 or 1.



[10]

Figure 13: Bloch Sphere representation of a qubit

This representation is useful for a single bit but sadly does not extend well to multiple bits.

B.2 Multiple Qubits

In the previous section we discussed a single qubit, with possible states $|0\rangle$ and $|1\rangle$. With two of these we now have four possible states; $|00\rangle, |01\rangle, |10\rangle$ and $|11\rangle$. However these can also exist in a superposition of any two of these states. Thus we can rewrite (7) for a two qubit system:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{10}$$

Before each state had a probability of being observed of $|\alpha|^2$ or $|\beta|^2$, which to keep with some new notation we can refer to as $|\alpha_i|^2$ where i runs from 0 to 1, as our two states do. For our current system we can say that the probability of returning a measurement of 0 would be $|\alpha_{00}|^2 + |\alpha_{01}|^2$, or $\sum_i |\alpha_{0i}|$. We can write an equation for the state of ψ after a measurement as;

$$|\psi'\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} \tag{11}$$

We can see that though an initial qubit has 2 possible states, adding a second gives us 4. This will continue and for n qubits we will have 2^n possible states, giving us one of the primary reasons for wanting to use a quantum computer in the first place.

B.3 Single Qubit Quantum gates

In traditional computation, logic gates are how we transform our 1's and 0's. They work by changing an input to a different output, much like an operator acting on a quantum state. Lets see if we can create a quantum NOT gate.
Traditional NOT gates transform a 1 to 0 and vice versa, thus we would like:

$$\alpha\,|0\rangle + \beta\,|1\rangle \rightarrow \alpha\,|1\rangle + \beta\,|0\rangle \tag{12}$$

If we treat these as the linear vectors that they are, we can describe this action through a simple matrix;

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{13}$$

The letter chosen to represent this matrix is often used to refer a NOT gate in quantum communication. We can use the vector representation of eq(7), and apply our matrix to it as such;

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \alpha\,|1\rangle + \beta\,|0\rangle \tag{14}$$

We can define two more quantum gates, the Z gate and the H or Hadamard gate:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{15}$$

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{16}$$

And they act as follows:

$$Z \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix} = \alpha\,|0\rangle - \beta\,|1\rangle \tag{17}$$

$$H \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}}(\alpha+\beta) \\ \frac{1}{\sqrt{2}}(\alpha-\beta) \end{bmatrix} = \alpha\frac{|0\rangle + |1\rangle}{\sqrt{2}} + \beta\frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{18}$$

From this we can see that the Z gate just changes the value of β negative, however the H gate is a little more complex. It is essentially an operation that ends up being halfway between sate $|0\rangle$ and $|1\rangle$. We can visualise it as through the Bloch sphere as a rotation about the y axis of 90° and then a flip along the x axis (180°).
While there are an infinite amount of possible 2x2 matrices, it can be shown that we need a much smaller set of function to map all the possible transformations. A single qubit unitary gate can be decomposed as a phase shift along with rotation along the z axis and a product of rotations, ie:

$$U = e^{i\alpha} \begin{bmatrix} e^{-i\beta/2} & 0 \\ 0 & e^{i\beta/2} \end{bmatrix} \begin{bmatrix} \cos\frac{\gamma}{2} & -\sin\frac{\gamma}{2} \\ \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} \end{bmatrix} \tag{19}$$

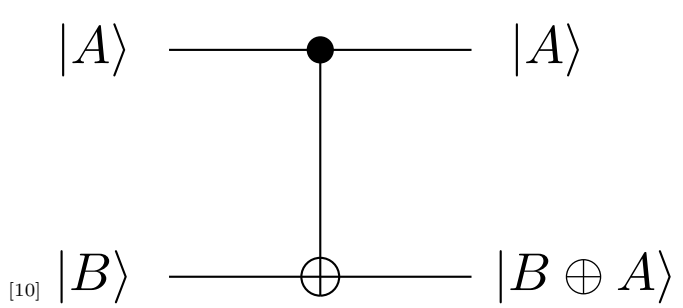
Here α, β and γ are real valued. Note that these factors don't need to be arbitrary, we will normally be working with specified fixed values of these coefficients.

The Z and X gates of course look just like the Pauli spin matrices, and are referred to as such. This should not come as a surprise as the spin matrices describe translations in Lie Algebra's. As such, there is also the less used Y gate, with the same properties as its Pauli counterpart.

B.4 Multiple Qubit Quantum Gates

While the examples in the previous system have taken one qubit as an input and produced one output result, we can now move onto the quantum NOT gate or the controlled NOT gate, CNOT. This is a two qubit logic gate where one of the inputs is a control and one is a target.

Here $|A\rangle$ is the control qubit and $|B\rangle$ is the target qubit. We can see that the output of this gate only affects the target qubit, we can also visualise this in matrix representation;



[10]

Figure 14: Demonstration of the CNOT gate, here $\oplus$ represents the tensor product of the two states.

$$\begin{aligned} U_{CNOT}|\psi\rangle &= U_{CNOT}\left(\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle\right) \\ &= \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{11} \\ \alpha_{10} \end{bmatrix} \\ &= \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|11\rangle + \alpha_{11}|10\rangle \end{aligned} \tag{20}$$

We see that the first two states remain unchanged, as our control qubit should, and that the second two act as a NOT gate, switching places. The classic NOT gate is the only gate that can be presented by a unitary matrix. All the other operations are non reversible, which requires them to be non unitary. We can, however, express any multiple qubit logic gate as a composition of a CNOT gate and a single qubit gate.

B.5 Density operator

The last part we will need to continue is what is called a density operator. Before we considered the state vector representation of our operators, we can also express this in another way. The density matrix can conveniently show these state when they are not fully known, i.e;

$$\rho \equiv \sum_i p_i |\psi_i\rangle \langle\psi_i| \tag{21}$$

Here we have a system where each $|\psi_i\rangle$ has a probability p_i of being in that state. Suppose that we have an operator, like before, U, acting on the state $|\psi_i\rangle$. The evolution of this density operator would then be:

$$\rho \equiv \sum_i p_i |\psi_i\rangle \langle\psi_i| \xrightarrow{U} \sum_i p_i U|\psi_i\rangle \langle\psi_i| U^\dagger = UpU^\dagger \tag{22}$$

Similarly to measure the probability of it being in state m after using the measurement operators M_m;

$$p(m|i) = \langle\psi_i| M_m^\dagger M_m |\psi_i\rangle = \mathrm{tr}(M_m^\dagger M_m |\psi_i\rangle \langle\psi_i|) \tag{23}$$

Thus the probability of obtaining result m is;

$$p(m) = \mathrm{tr}(M_m^\dagger M_m \rho) \tag{24}$$

The state of this above probability is given by;

$$|\psi_i^m\rangle = \frac{M_m|\psi_i\rangle}{\sqrt{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle}} \tag{25}$$

Furthermore, we can describe the above probability as a density operator, i.e;

$$\rho_m = \frac{M_m \rho M_m^\dagger}{\mathrm{tr}(M_m^\dagger M_m \rho)} \tag{26}$$

The last two items to mention for density operators is that;

a. They have a trace of 1

b. They are a positive operator.

This is all we will need to know to start looking into the basics of quantum error correction, the CNOT gate, the H gate, the Density operator and the Pauli spin gates.

C Correcting Quantum Circuits

[10;13;14;15]

As mentioned in the introduction, quantum circuits are not yet at their classical counterparts level of precision. That is, while in the past computers had issues with bleed from circuits and other problems in manufacturing, these days they rarely require much attention. Apart from large servers that often employ backup drives or special memory that can correct itself, the circuitry mostly just runs. However in the past designs were implemented to make this more stable, [16] [17] or fault-tolerant. This approach has transitioned to the noisy and unstable landscape of quantum computing.

In essence, fault tolerance requires that circuits can recover from an error caused from many different sources, and that this error shouldn't cascade into other parts of the machine. This requires careful planning and routing of where dependencies lie, so as not to make multiple parts of a circuit rely on just one piece that could fail.

With motivations and the notation covered in the last section we can move onto the main problems of this project, that is the physics we will be trying to "fix". We will begin with a brief description of the processes behind Quantum Error Correction (QEC) before moving onto the main type of code used in this project, Surface codes, specifically Toric code.

C.1 Sources of Errors in Quantum computers

The primary sources of errors in computers come from a few different places. Firstly, we could incorrectly apply a gate, this is known as a coherent systematic error. Other sources of error are known to be incoherent, and are environmental decoherence, loss, leakage, measurement or just failing to initialise the qubit correctly.

Coherent and incoherent errors have different methods for solution, as described below.

C.1.1 Setup and Incorrect gate applications

To begin with we will first consider a single qubit undergoing a unity transformation, I, N times;

$$|\psi\rangle_{final} = \prod_{i}^{N} I_i |0\rangle = |0\rangle \tag{27}$$

Next we perform 2 Hadamard operations and an identity transformation;

$$\begin{aligned} |\psi\rangle_{final} &= HIH|0\rangle \\ &= HI\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= H\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= |0\rangle \end{aligned} \tag{28}$$

This operation should provide a $|0\rangle$ with 1 probability, but for our first source of error we will assume a gate has been applied incorrectly. If we take (27) as a basis and assume that the error comes off with a change of a rotation around the x axis of the Bloch sphere, we get;

$$|\psi\rangle_{final} = \prod_{i}^{N} e^{i\epsilon\sigma_x} |0\rangle = \cos(N\epsilon)|0\rangle + i\sin(N\epsilon)|1\rangle \tag{29}$$

If we take the probability of the state to be in $|0\rangle$ or $|1\rangle$, we get;

$$\begin{aligned} P(|0\rangle) &= \cos^2(N\epsilon) \approxeq 1 - (N\epsilon) \\ P(|1\rangle) &= \sin^2(N\epsilon) \approxeq (N\epsilon) \end{aligned} \tag{30}$$

In this very simplistic view we can see that the chance to be in an erroneous state grows with the number of transformations, and is dependant on the error rate, ϵ.

C.1.2 Environmental Decoherence

The next type of error we want to look at is known as environmental decoherence. Consider a simple two quantum level system with two basis state, $|e_0\rangle$ and $|e_1\rangle$, which satisfy the completeness relations;

$$\langle e_i|e_j\rangle = \delta_{ij}, \quad |e_0\rangle\langle e_0| + |e_1\rangle\langle e_1| = I \tag{31}$$

A few assumptions will be made for this example, namely that;

a. The state can only go flip from $|0\rangle$ to $|1\rangle$ and not the other way around.

b. The environment only acts during a so called wait stage, or the I stage in (28).

If we assume that the state is initialised in state $|E\rangle = |e_0\rangle$ before being coupled to the system;

$$HIH\,|0\rangle\,|E\rangle = \frac{1}{2}(|0\rangle + |1\rangle)\,|e_0\rangle + \frac{1}{2}(|0\rangle - |1\rangle)\,|e_1\rangle\,S \tag{32}$$

For the environmental decoherence we will assume the pure states are to be transformed into classical mixtures, the density matrix of this would, therefore, be;

$$\begin{aligned}
\rho_f = &\frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big)\,|e_0\rangle\langle e_0| \\
&+ \frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big)\,|e_1\rangle\langle e_0| \\
&+ \frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big)\,|e_0\rangle\langle e_1| \\
&+ \frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big)\,|e_1\rangle\langle e_1|
\end{aligned} \tag{33}$$

We can then take the trace of this;

$$\begin{aligned}
\mathrm{Tr}_E(\rho_f) &= \frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big) \\
&\quad + \frac{1}{4}\Big(\,|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|\,\Big) \\
&= \frac{1}{2}\Big(\,|0\rangle\langle 0| + |1\rangle\langle 1|\,\Big)
\end{aligned} \tag{34}$$

This means that the system changes to a classical one with 50% chance of being in either $|0\rangle$ or $|1\rangle$. This means that the next transformation, of a Hadamard gate, would do nothing to the state, and mean that it had an erroneous state.

C.1.3 Loss, leakage, measurement and initialization

The last set of problems we will discuss before outlining the initial solutions available can be summarised in one way. They are errors of loss, leakage, measurement and initialisation.

The first of these we shall look at will be measurement errors. They can be modelled like environmental decoherence, as in the last section. To begin with we can either give them two positive operator value measures (POVM);

$$\begin{aligned}
F_0 &= (1 - p_M)\,|0\rangle\langle 0| + p_M\,|1\rangle\langle 1| \\
F_1 &= (1 - p_M)\,|1\rangle\langle 1| + p_M\,|0\rangle\langle 0|
\end{aligned} \tag{35}$$

Where p_M is the probability of a measurement error. Or we can map a transformation of the density matrix as follows;

$$\rho \to \rho' = (1 - p_M)\rho + p_M X\rho X \tag{36}$$

This would then be followed by a measurement of in the $(|0\rangle, |1\rangle)$ basis. It can be shown that both methods give the same probability of;

$$\rho \to \frac{M_i \rho M_i^\dagger}{\mathrm{Tr}(F_i \rho)} \quad \text{where} \quad i = 0, 1 \tag{37}$$

and,

$$\begin{aligned} M_0 &= \sqrt{1 - p_M}\,|0\rangle\langle 0| + \sqrt{p_M}\,|1\rangle\langle 1| \\ M_1 &= \sqrt{1 - p_M}\,|1\rangle\langle 1| + \sqrt{p_M}\,|0\rangle\langle 0| \end{aligned} \tag{38}$$

For both of the models the probability of measuring $|0\rangle$ is 50%, as before.

Next we should model qubit loss. This can be done by taking the trace of our density matrix, ie; $\mathrm{Tr}_i(\rho)$ where i is the index of the lost qubit. This loss cannot be directly measured or coupled to any of our other systems.

Incorrect initialisation of the qubit can be summarised by the following density matrix;

$$\rho_i = (1 - p_I)\,|0\rangle\langle 0| + p_I\,|1\rangle\langle 1| \tag{39}$$

Where p_I is the probability of an initialisation error.

The final type of error, the leakage error, can be theoretically modelled quite simply as follows. If we stop using the idealised 2 state system, and instead use one with multiple energy levels (as is the case in any real world example) then leakage can occur with improper control applied to such a system. We can define the transformation;

$$U\,|0\rangle = \alpha\,|0\rangle + \beta\,|1\rangle + \gamma\,|2\rangle \tag{40}$$

Where a third state is now available. Just entering this into our equations, which assume a two level system, will give unwanted dynamics.

If a qubit goes up to this unwanted third state, it will have to decay down to a lower energy state, presumably giving off some form of radiation. This could further upset the system.

With the basic knowledge of what we need to correct, we may now begin creating circuits to alleviate these issues.

C.2 Simple Error correction codes

[13]

In this section we will discuss codes that try to fix one or two types of errors at a time, and with what we will see to be low thresholds. After this we can move onto topological codes with higher thresholds, like the toric code.

C.2.1 3 Qubit code

The first step towards correcting the issues that crop up in the previous section is making a simple 3-qubit error correcting code. This is not a fix all remedy but merely a starting point. Simply put, we assign a logical qubit into three physical ones, where their combined output generates the signal we want for one correct qubit.

We can define a basis state for these gates as such;

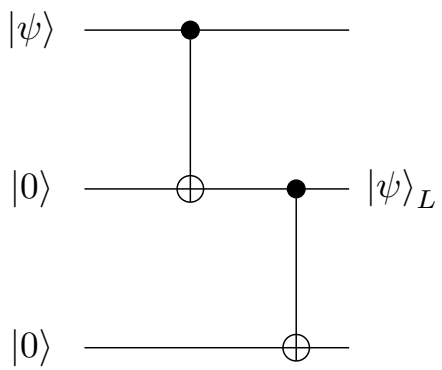
$$|0\rangle_L = |000\rangle \qquad |1\rangle_L = |111\rangle \tag{41}$$

And then map them into an arbitrary single qubit state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$;

$$\begin{aligned} \alpha\,|0\rangle + \beta\,|1\rangle &\to \alpha\,|0\rangle_L + \beta\,|1\rangle_L \\ &= \alpha\,|000\rangle + \beta\,|111\rangle \\ &= |\psi_L\rangle \end{aligned} \tag{42}$$

We can then link these together with two CNOT gates, as shown in fig(15);

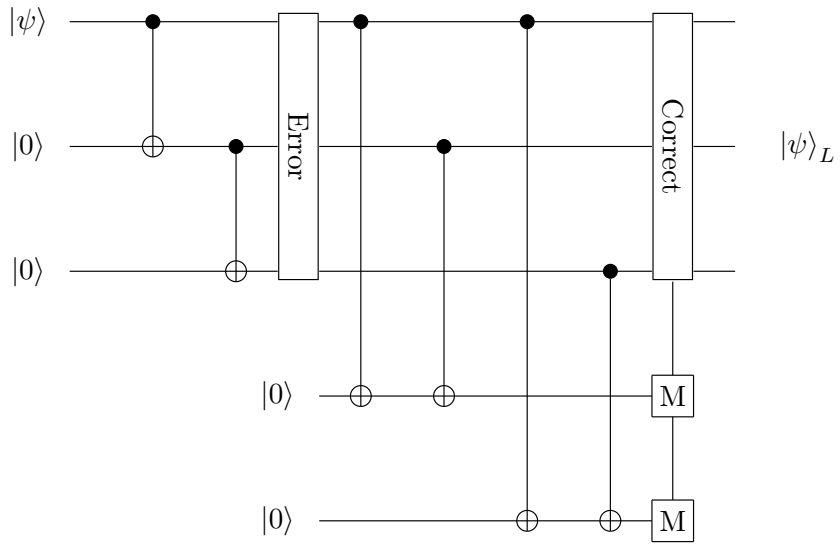
If we assume $|\psi\rangle_L$ is in the $|0\rangle$ state, then after a bit flip error, the final state is still closer to $|0\rangle_L$ than $|1\rangle_L$. It would take 2 bitflip errors to change the outcome to being closer to an incorrect state. However this



[13]

Figure 15: 3 qubit code for linking qubits together

requires measurement of $|\psi\rangle_L$ in order to determine the error. In order to extract this information without reading the qubit, 2 more physical qubits are required, as shown in fig(16);



[13]

Figure 16: A 3 qubit code that corrects a bit flip error.

As the top qubit of the CNOT gate does not affect the outcome, the outcomes of the latter two qubits can be measured to dictate what is called the "syndrome" of the system. This tells us what kind of an error has occurred, so that we can fix it with an appropriate gate application. With this data, if the bit flip error has occurred and hasn't been corrected, we can correct a bit flip error after the process knowing it has occurred.

The distance between 2 codeword states, d, determines how many errors can be corrected. This is given as t, and is calculated by;

$$t = \left(\frac{d-1}{2}\right) \tag{43}$$

For this code, $d = 3$, thus $t = 1$, hence we can only correct for one bitflip error in our system.

C.2.2 9 qubit code

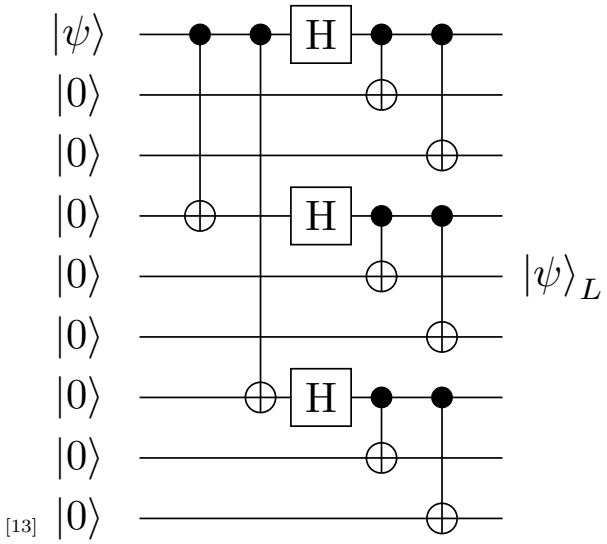
[2]

This is based on similar architecture to the 3 qubit code, but has a much higher fault tolerance. It can recover a single bit flip error, a single phase flip, or both together. It can do this on any of the 9 qubits.

We can define its basis states as;

$$\begin{aligned} |0\rangle_L &= \frac{1}{\sqrt{8}}(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle) \\ |1\rangle_L &= \frac{1}{\sqrt{8}}(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle) \end{aligned} \tag{44}$$

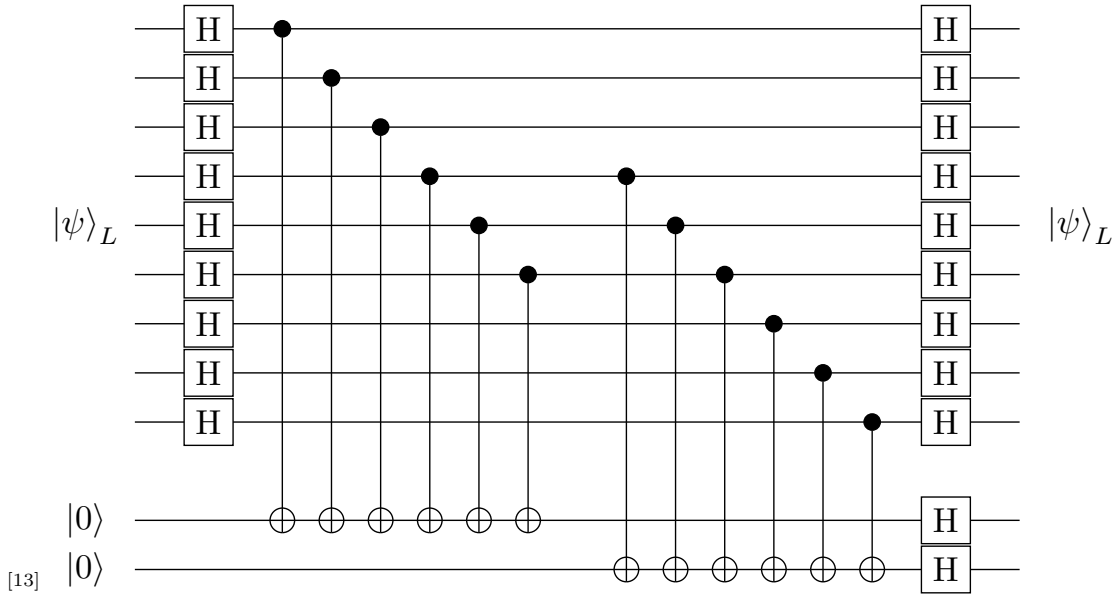
The 9 qubit circuit would then look as follows in fig(17):



[13]

Figure 17: The circuit needed for the 9 cubit code, also known as Shors algorithm [2]

As before, a measurement is needed to detect if an error has occurred. So, again, we can add on just two more qubits in order to extract syndrome information on the operation, as seen in fig(18);



[13]

Figure 18: 9 qubit code with the ability to fix the error.

Here the first set of six CNOT gates compares the sign for blocks one and two, and the later two and three. This allows us to fix phase flip errors, due to the sign difference of the out put of the bottom two qubits.

C.2.3 Stabiliser formalism

[13]

This is a way of describing errors that doesn't require a state vector representation. It generalises both circuit construction and error correction, regardless of the code used.

To begin with we name a state $|\psi\rangle$, and say that it is stabilised by some operator, K, if it is a $+1$ eigensate of K, i.e;

$$K|\psi\rangle = |\psi\rangle \quad \text{and,} \quad \sigma_z|0\rangle = |0\rangle \tag{45}$$

Where the latter equation signifies what a stabilised state would look like. We can assign a subgroup of all possible single qubit operators as the Pauli group, denoted by;

$$\mathcal{P} = \{\pm\sigma_I, \pm i\sigma_I, \pm\sigma_x, \pm i\sigma_x, \pm\sigma_y, \pm i\sigma_y, \pm\sigma_z, \pm i\sigma_z, \} \tag{46}$$

And the commutation relations between them given by;

$$[\sigma_i, \sigma_j] = 2i\epsilon_{ijk}\sigma_k, \quad \{\sigma_i, \sigma_j\} = 2\delta_{ij} \tag{47}$$

with ϵ, δ being the levi civita and chroniker delta respectively. It can be shown that with these, $\mathcal{P}$ forms a group under matrix multiplication.

The Pauli group extended over N qubits can be given as function of the tensor product;

$$\mathcal{P}_N = \mathcal{P}^{\otimes N} \tag{48}$$

Therefore, an N qubit stabiliser state, $|\psi\rangle_N$ is defined by N generators of an abelian subgroup, $\mathcal{G}$, of the N qubit Pauli group;

$$\mathcal{G} = \left\{K^i \mid K^i |\psi\rangle = |\psi\rangle, [K^i, K^j] = 0, \forall(i,j)\right\} \subset \mathcal{P}_N \tag{49}$$

We can show an example of how this works bu using a Greenberger Horne Zeilinger (GHZ) state. Specifically a three qubit GHZ state;

$$|GHZ\rangle_3 = \frac{|000\rangle + |111\rangle}{\sqrt{2}} \tag{50}$$

If we then express this linearly via any of the three generators of the group, i.e the Pauli matrices, we get;

$$\begin{aligned} K^1 &= \sigma_x \otimes \sigma_x \otimes \sigma_x \quad \equiv XXX \\ K^2 &= \sigma_z \otimes \sigma_z \otimes \sigma_I \quad \equiv ZZI \\ K^3 &= \sigma_I \otimes \sigma_z \otimes \sigma_z \quad \equiv IZZ \end{aligned} \tag{51}$$

The right hand side here is known as the short form of the stabliser formalism. There are also four orthogonal bell states'

$$\begin{aligned} \left|\Phi^{\pm}\right\rangle &= \frac{|00\rangle \pm |11\rangle}{\sqrt{2}} \\ \left|\Psi^{\pm}\right\rangle &= \frac{|01\rangle \pm |10\rangle}{\sqrt{2}} \end{aligned} \tag{52}$$

These are stabilised by the operators, $K^1 = (-1)^a XX$ and $K^2 = (-1)^a ZZ$, where $[a, b] \in \{0, 1\}$. Each of these states correspond to four ± 1 eigenstate combinations of 2 operators.

$$\begin{aligned} \Phi^+ &\equiv \begin{pmatrix} K^1 & = & XX \\ K^2 & = & ZZ \end{pmatrix} \qquad \Phi^- \equiv \begin{pmatrix} K^1 & = & -XX \\ K^2 & = & ZZ \end{pmatrix} \\ \Psi^+ &\equiv \begin{pmatrix} K^1 & = & XX \\ K^2 & = & -ZZ \end{pmatrix} \qquad \Psi^- \equiv \begin{pmatrix} K^1 & = & -XX \\ K^2 & = & -ZZ \end{pmatrix} \end{aligned} \tag{53}$$

Using this type of formalism we can define a code for our 9 qubit correction code, fig(17). Only the stabiliser qubits require this code, so for this example we would have 8, the code is;

$$\begin{array}{lccccccccc} K^1 = & Z & Z & I & I & I & I & I & I & I \\ K^2 = & Z & I & Z & I & I & I & I & I & I \\ K^3 = & I & I & I & Z & Z & I & I & I & I \\ K^4 = & I & I & I & Z & I & Z & I & I & I \\ K^5 = & I & I & I & I & I & I & Z & Z & I \\ K^6 = & I & I & I & I & I & I & Z & I & Z \\ K^7 = & X & X & X & X & X & X & I & I & I \\ K^8 = & X & X & X & I & I & I & X & X & X \end{array} \tag{54}$$

C.2.4 Threshold Theorem

[13]

This is a very important part of building a working Quantum Computer. It dictates if our system will be able to run or if it will break down due to errors. We can describe it by first taking a circuit that can experience either an X and/or a Z error independently with probability p, per gate operation, as the code in fig(18) can fix. We should also assume that the gates and components are designed with fault tolerance in mind. That is, that they do not cause cascading errors through their implementation.

If a cycle of error correction is performed after each calculation block we can call the probability of error on each logical operation, $p_L^1 = cp^2$ where p_L^1 is the failure rate per operation and c is the upper bound for the possible number of 2-error combinations.

We now concate, this is when we encode logical qubits (a group of physical qubits not necessarily using the same QEC) to a new level-2 encoded logical qubit. The formalism for doing this is given by $[[n, k, d]]$ where n is the number of qubits used, k is the final amount of output qubits and d is the same one used in eq(43), which in this case would be 3.

If the level-1 encoded qubits had a failure rate of p_L^1, then the level-2 encoded qubit would have $p_L^2 = c(p_L^1)^2 = c^3p^4$. We can keep encoding these cubits to higher level, up to g, as below;

$$p_L^g = \frac{(cp)^{2^g}}{c} \tag{55}$$

As long as $cp < 1$ this produces a arbitrarily small error rate. This also defines the threshold theorem, the physical error rate experienced by each qubit per time step must be $p_{th} < 1/c$.

From this we can see that once the error rate of a system is below the threshold, the circuit can be deemed accurate and computation can occur on it. Thus the calculation of the threshold of system and its QEC is vital before any work can be conducted on it. The thresholds for a few systems can be seen on the table below;

[13]

Code	Threshold
9 qubit	$O(10^{-5} - 10^{-9})$
surface code	$O(10^{-2} - 10^{-3})$
toric code	18%

Table 3: Note that the thresholds have been summarised from the source. This summary has a lot of great data that was too detailed to include. Note that there are many different kinds of 9 qubit code and surface codes, hence why they have a spread.

From this table we can see why we would want to move to topological codes, the increase in threshold is remarkable.

C.3 Topological codes

[18]

In this section we will go through 2 developments in QEC that are based on the arrangement of qubits on a surface.

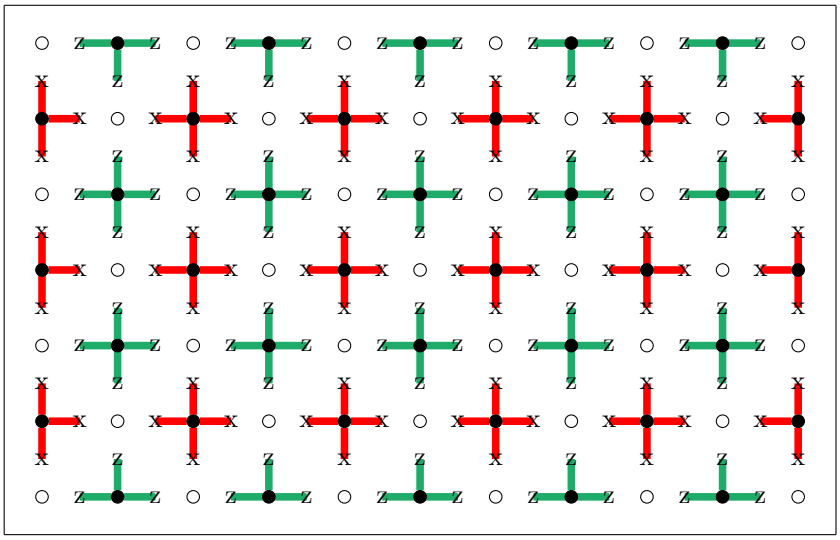
C.3.1 Surface code

[14] [1]

The arrangement of these qubits in surface code can be seen in fig(19);

There is a lot to unpack with this. Firstly, there are 2 kinds of qubits, data qubits (white circles) and measurement cubits (black circles). Data qubits are our computational qubits where data is stored for our calculations, while measurement qubits are there for error detection.

Of these measurement qubits there are also two kinds, measure z and measure x, denoted by each name and the colours of green and red respectively. These refer to the Z and X syndrome of the surface code. Each data qubit is couple to two of each type of measurement qubit



[1]

Figure 19: An initial representation of the surface code.

Thus each measurement qubit is forced into an eigenstate of Z or X, and from its position we can calculate which qubit has erred.

This is as far as we will go with this for now, we are mostly interested in Toric code, which is a subset of surface code.

C.3.2 Toric code

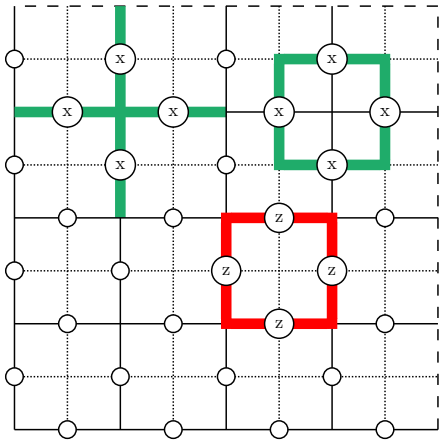
[14] [19]

This is a specific type of surface code that has had a very high threshold, about 18% [13]. It uses the surface of a toroid as its basis, hence its name. There are quite a few parts taken from the surface code. Firstly, it is also encoded on a lattice of $n = 2L^2$ physical qubits. Where L is the length of the square lattice.

As before, we have 2 sets of stabilisers, the x and the z. Here they are called vertex and plaquette operators, respectively. ;

$$A_\nu = \otimes_{i\in\nu} X_i \qquad B_p = \otimes_{i\in p} Z_i \tag{56}$$

In addition to the standard cross links seen above in the surface code, called the primal lattice, in a Toric lattice we can also construct a dual lattice. This is done by replacing the plaquettes of the primal lattice by vertices and vice versa, this is shown in fig(20);



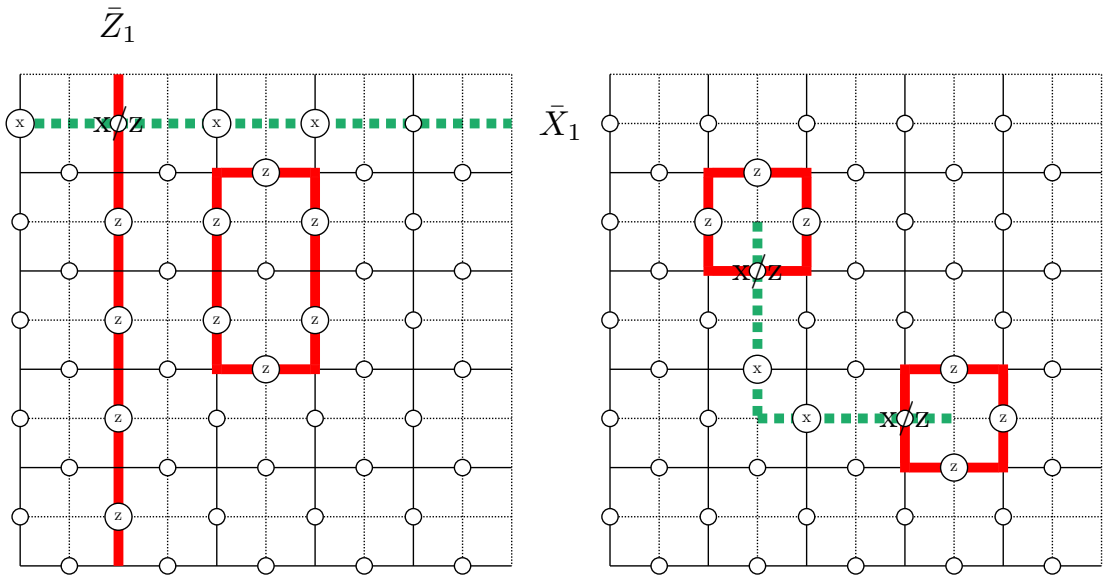
[19]

Figure 20: The Toric code for which there are periodic boundary conditions imposed on the lattice. It can be thought of as a section of a toroid (doughnut).

By considering both of these lattices we can see that all stabilizers are closed loops. This means that all plaquette operators have an analogous vertex type on the secondary type of lattice. Thus all results calculated for either bit flip or phase flip errors can be interchanged for the other type.

All of the stabilisers correspond to homologically trivial cycles. In fig(21)we can see and example of this. Logical operators are denoted by an overbar as in $\bar{Z}, \bar{X}$. These logical operators are also represented by Pauli operator cycles. But these wrap around the torus and are not homologically equivalent to stabilisers. They, in fact, correspond to homologically non trivial cycles, and have a non trivial effect on the code space, with weight L.

Errors can be detected if they anti commute with at least one element of the stabiliser generators S. We can call the set of all errors on the lattice a chain, E. These errors commute with the stabiliser except on ∂E, or on the boundary. Specifically it is where the eigenvalues of this chain is non zero. Theses eigenvalues in turn provide us with the syndrome of our system.



[19]

Figure 21: On the left we have an examples of non trivial cycles for $\bar{X}_1$ and $\bar{Z}_1$ acting on the encoded data. Note how they anticommute and thus are perpendicular to each other. The circle is representative of a trivial cycle of two adjacent plaquette operators.
On the right we instead have the dashed line representing X errors. Measuring the two generators, cycles, gives us an eigenvalue of -1, showing anti commutation.

D Machine learning

[20]

The above processes use computer codes that are static, what we mean by this is that they have a fixed algorithm thought up by a human and implemented to a computer. This algorithm can iterate but its core structure is a single mathematical equation. The best of these so far is the Minimum Weight Perfect Matching [21] (MWPM) algorithm. However, these codes always encounter scalability issues. With machine learning we could let computers use the codes best suited to each situation, or make up a new set based on their own weighting.

However, machine learning is a very broad term. Any algorithm that uses a previous result and then iterates on that result to make a prediction, can be classed as machine learning. Therefore in this section we will discuss and define a few initial algorithms and them move onto reinforcement and then deep learning.

D.1 Terminology

Here we will briefly discuss the classifications of machine learning. This is important to do to understand where our chosen form of machine learning fits, and why we chose to use it in for this project. Firstly we

will make a distinction based on how the system makes its predictions, then on how the data is supplied, and lastly we will discuss how it generalises data.

D.1.1 Supervised vs Non supervised

For supervised systems, you will require training data to feed to your algorithm in order to "train" it. Typically these systems are used for classifications, i.e. determining if there is a face in a picture or not. Or for more mathematical models this could be deciding prices of items in the future based on historical trends.

These systems decide on a value for a "target" given data on it called "predictors". This type of algorithm is often refereed to as regression. Some common types of supervised algorithms are;

- Linear Regression.
- Logistic Regression.
- Support Vector Machines (SVMs).
- Decision Tree and Random Forests
- Neural networks

For unsupervised systems there are no input parameters. The system must make its own data sets. A simple and popular version of this is with clustering. Given a random number of points, centres for these points can be found. This type of result can be useful for detecting trends, for instance where people would like to shop, or for astronomical research based on star locations. Important examples of unsupervised systems are;

- Clustering
- Visualisation and dimensionality reductions
- Association rule learning

There is a third variant of these called semi supervised learning, which takes data as an input, makes guesses, then asks for input on these results to label the rest of the data. This is particularly useful for determining identities of faces in groups of photos.

Lastly there is reinforcement learning. This will be covered later on in more detail, but it is a separate form of learning style in its own right and so should be covered here. In essence this learning model uses an in-between agent. This agent is given rewards or punishments for getting correct or incorrect guesses to known results. It then can adjust its code to account for these rewards, but essentially teaches itself how to best function.

D.1.2 Batch or Online

This has to do with how the algorithm accesses its data. Firstly we will look at batch learning. This is probably the more classical way you would train an algorithm. To begin with, we can take all the data and input it into our system. This is the simplest solution, however, often we are dealing with thousands, or even millions, of data points. This is difficult in traditional computers, as mentioned in the introduction, so we need a workaround. A solution to this is to have the algorithm go through the data in batches. This can then be parallelised if needed. Often this type of data feed is called offline learning, and typically all the data is available from the start.

The other side of this is called online learning. Here we feed the data incrementally, and sequentially. This system is good for systems where fresh data is always being added, like stock markets.

D.1.3 Instance vs model based learning.

The last set of categorisations we will discuss is based on how the models generalise the data. This refers to how the algorithm will fit its predictions to new data having been trained. The first way this can be done is called "instance-based" learning.

This is where each type of outcome is "instanced" as a response. I.e, if this happens again in the same way, it will be categorised as such. This is useful for a spam filter, or for classifications of stars, they

both have similarities between each instance of email or star that we can hope to use for the prediction of properties for future entries.

If there aren't similarities between entries in a set, we can instead use "model-based" learning. This is where we pick a model to fit to our data. If the data looks like it could fit a linear line we fit it linearly, if it looks asymptotic we could use an exponential function and so on.

D.2 Common problems of machine learning

Broadly speaking, there are two types of problems that can occur with a machine learning algorithm that are common to all classifications listed above. The first is problems with the training data, and the second is problems with the fit. In this section we will go over these issues.

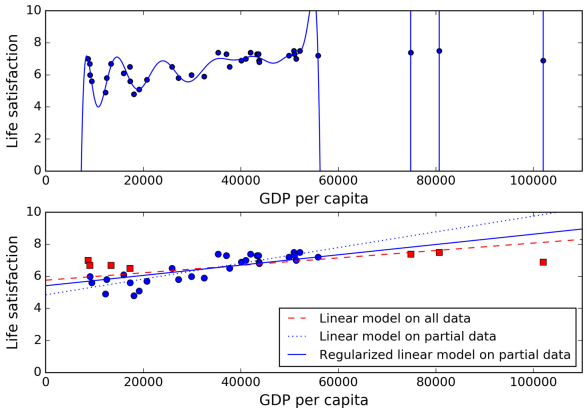
D.2.1 Training data

Training data can come in may different forms, and with a multitude of problems. The most prominent are;

- **Non representative training data** - This can include data that was missing key points or data with inherent sampling bias.
- **Irrelevant features** - Data that includes values that are irrelevant to a fit.
- **Poor quality data** - Usually data that is too noisy or with a systematic error.
- **Poor quantity of data** - Simply not enough points to form a valid hypothesis from the results.

D.2.2 Over and Under-fitting

This is perhaps where the largest complexity in machine learning comes from. Knowing when you fit the points instead of fit the model is a fine line, that takes multiple runs to get right. Take for example the data in fig(22);



[20]

Figure 22: The above image gives an example of data that is over-fitted while the graph below shows a more reasonable set of lines.

In the first image we see an example of over-fitting the data. This model most probably used an algorithm that reduced the root mean squared (RMS) above all other features, and resulted in a model that fit the data instead of producing a reasonable trend. A model based on these results will have extremely skewed conclusions, and values that are between some of these lines will be placed into incorrect categories. The second graph gives a much more reasonable conclusion to our results, even with a larger RMS.

Similarly, we can try to fit a linear line to a set of data that should have been polynomial. This would be an example of under fitting.

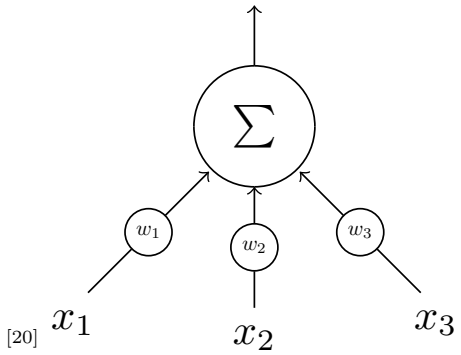
D.3 Neural nets

[22]

Before we can discuss reinforcement learning we need to underpin some of the information that will be presented. Firstly we need to define an artificial neuron, then we need to create a neural net from this neuron. After this we can discuss how learning and deep learning can be done on these nets. We briefly talked about this in types of learning from the last section, and we will delve deeper into the theory behind this. Then finally we can bring it all together with reinforcement learning.

The inspiration for artifical neurons comes from their biological counterparts. Much like the brain reinforces certain pathways between neurons in our brains depending on use (the basis on how we learn) so too do neural nets develop links between artificial neurons with rewards. To begin with, though, we must define one such net.

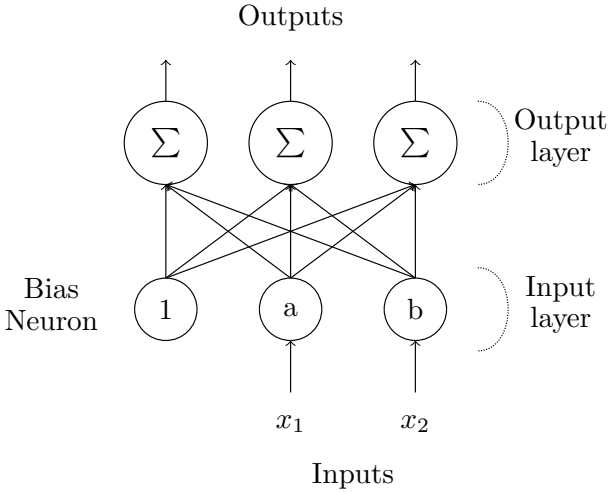
We will skip straight to the type of net we will be using (and there are other kinds), the Perceptron. We begin with a single neuron which takes a set of inputs, numbers, takes an average and outputs a step function, as shown in the diagram below;



[20]

Figure 23: Figure demonstrating how data inputs can be summed then exported through an LTU.

Here we take an input to a central weighted sum in the form of $\sum = \vec{w}^T \cdot \vec{x}$. The output then uses a step function of $h_w(\vec{x}) = \text{step}(z) = \text{step}(\vec{w}^T \cdot \vec{x})$. The most common of these step functions is a Heaviside step function. This way if the weighted sum meets a threshold, a simple $+1, -1$ or 0 can be outputted and added to the next link in the chain. Because of this threshold, we can refer to this neuron as a linear threshold unit (LTU). We can link together such LTU's to form a chain, such as the net in fig(24);



[20]

Figure 24: Here we have added a bias neuron and a pass through input layer. This is the simplest form of neural net.

Here the perceptron is composed of a single layer of LTU's, with each neuron connected to each of the inputs. The neurons a and b are special pass through neurons, that just take the inputs and pass them to all out the output layers. The above configuration has specifically 2 inputs and three outputs, but this need

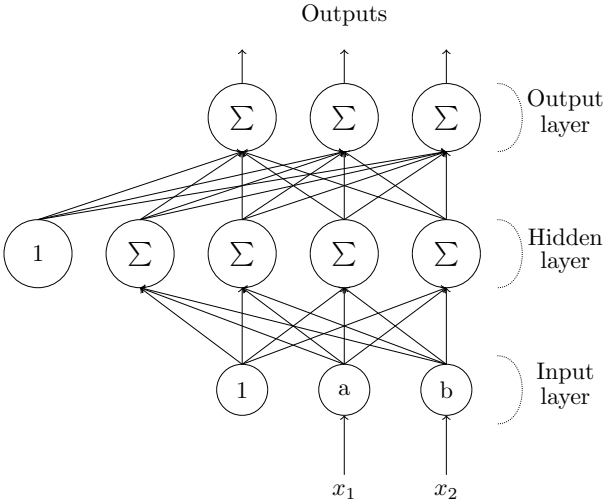
not always be the case. There is also an extra bias neuron added to the input layer that always outputs 1.

To train the sytem we give it data and ask it to make a prediction. Each time the output neuron gives an incorrect answer, it reinforces the neuron that gave the correct answer, giving it a higher weighting in the weighted mean. This can be summarised by the following equation:

$$w_{ij+1} = w_{ij} + \eta(y - y_j)x_i \tag{57}$$

Where w_{ij} is the connection weight between neurons i and j, x_i is the training input, y and y_j are the target outcome and the actual outcome and η is the learning rate.

This is a simple linear example of a neural net, to make a more complex one we need to add another layer of neurons, called multi-layer perceptron (MLP). One of these is outlined below in fig(25).



[20]

Figure 25: An MLP with a single layer of hidden neurons. If it had more than one layer, it would be referred to as a Deep Neural Net or DNN.

This operates much like the single layered version, but to train it we need to employ a gradient descent algorithm. Each time a training instance gives a prediction, the difference between the desired result and the result given is calculated. The algorithm then calculates how much each neuron in the last hidden layer contributed to the error. It then does it for the next hidden layer, and continues to do it until no more hidden layers remain. This negative error gradient is then input into our gradient descent algorithm and the weights adjusted accordingly. This type of training is called Backpropagation. Note that in order for this to work, the step function had to be changed out for a logistic function.

D.4 Reinforcement Learning

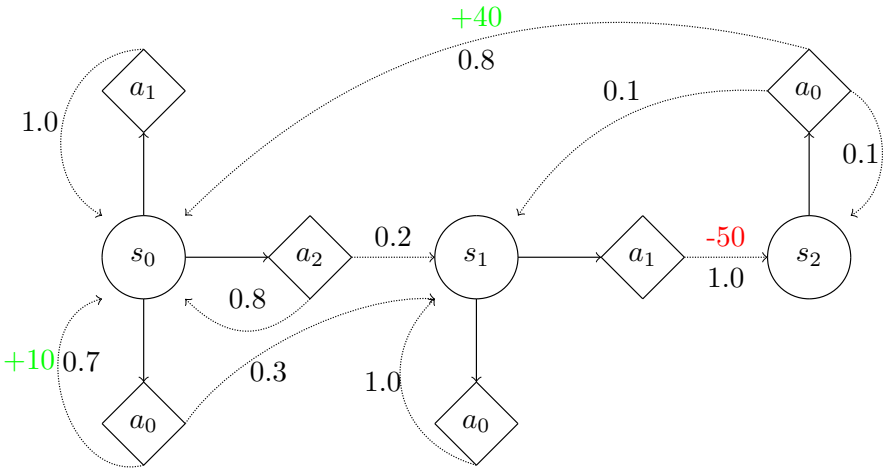
The next logical step, after creating a DNN that can give predictions, is to hand control of its own parameters completely to an outside agent, that is given real world data and asked to figure out the best course of action. The algorithm used by the agent to determine its actions is called its policy. The agent makes observations and takes actions within an environment, and is given rewards for the actions.

In order to assign an appropriate policy for our agent to use, we must first create an appropriate environment for it. A place it can be trained with data before being given control of a real world system. After creating an appropriate environment and assigning the correct policy for an agent to use, we must then choose how to assign the agent credit for its tasks. If we give them too frequently we essentially just create a supervised learning situation, similar to over fitting data. If we create too few, the agent will be unable to piece together a solution.

If we assign a value to an action based on the sum of all rewards that come after it, while applying a discount at each step, we can get around this problem somewhat. That is, some actions that act towards the correct action gives more than others. This method requires many runs through these simulations in

order to normalise the results and to determine the best reward values.

Next we can choose how we train our agent. For this we will start with a Markov chain and then move onto Q-learning. Below is an example of a Markov decision process;



[20]

Figure 26: An example of a Markov chain. Here the positive reinforcement bonuses are marked in green and the negative in red. The fractions by each line are the chance of each direction being taken. Furthermore move from an action is represented by a dotted line while a move from a state is a solid line.

Here the example has three states, (s_0, s_1, s_2), and three actions, (a_0, a_1, a_2), at each step. If we start in state s_0 then the agent can move to actions a_0, a_1 or a_2. The probability of getting the rewards is listed as a decimal, the path to maximise points is not initially obvious. The bellman optimality equation can help. It can calculate the optimal state value for each state s_i, it is given below;

$$V^\star(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^\star(s')\right] \qquad \text{for all } s \tag{58}$$

Where $T(s,a,s')$ is the transition probability from s to s' given the agent chose action a, $R(s,a,s')$ is the rewards given the same parameters and γ is the discount rate. This equation leads to an iterative form. To begin with we can initialize all the sate value estimates to zero, and then use the following formula to update them;

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right] \qquad \text{for all } s \tag{59}$$

The new addition, $V_k(s)$. is the estimated value of the state s at the k^{th} iteration. This equation doesn't tell the agent what to do, this is given by the optimal state-action values, often called Q-Values. The equation is very similar;

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right] \qquad \text{for all } s \tag{60}$$

Once you have this, the optimal policy is given the me maximum of this argument, $Q(s,a)$. Finally, Q-learning can be summarised as a slight change to the above equation, still with initially unknown rewards;

$$Q_{k+1}(s,a) \leftarrow (1-\alpha) Q_k(s,a) + \alpha \left[r + \gamma \max_{a'} Q_k(s',a')\right] \tag{61}$$

Where α is the learning rate.

This is where we can leave the theory behind and begin moving onto implementing an initial code through Tensorflow. This, however, is not necessary to include in a literature review.

References

[1] John M. Martinis Andrew N. Cleland Austing G.Fowler, Matteo Mariantoni. Surface codes: Towards practical large-scale quantum computation. https://arxiv.org/ftp/arxiv/papers/1208/1208.0928.pdf. 16/3/2020.

[2] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995.

[3] Gemma De las Cuevas Hans J. Briegel. for artificial intelligence. https://arxiv.org/abs/1104.3787. 24/3/2020.

[4] Vedran Dunjko Hans J. Briege Alexey A. Melnikov, Adi Makmal. Projective simulation with generalization. https://arxiv.org/pdf/1504.02247.pdf. 09/09/2020.

[5] David Poulin Nicolas Delfosse, Pavithran Iyer. A linear-time benchmarking tool for generalized surface codes. https://arxiv.org/abs/1611.04256. 09/09/2020.

[6] Sergey B. Bravyiy and Alexei Yu. Kitaev. Quantum codes on a lattice with boundary. http://cds.cern.ch/record/371927/files/9811052.pdf. 09/09/2020.

[7] Nicolas Delfosse Vedran Dunjko Hans J. Briegel Nicolai Friis Hendrik P. Nautrup. Optimizing quantum error correction codes with reinforce-ment learning. https://arxiv.org/pdf/1812.08451.pdf. 24/3/2020.

[8] Joel Johansson Simon Liljestrand Mats Granath Philip Andreasson. Quantum error correction for the toric code using deep reinforcement learning. https://arxiv.org/abs/1811.12338. 24/3/2020.

[9] John Preskill. Quantum computing in the nisq era and beyond. https://arxiv.org/abs/1801.00862. 10/9/2020.

[10] Isaac L.Chuang Michael A.Neilson. *Quantum Computation and Quantum Informtaion*.

[11] Ivan Kassal, James D. Whitfield, Alejandro Perdomo-Ortiz, Man-Hong Yung, and AlÃ¡n Aspuru-Guzik. Simulating chemistry using quantum computers. *Annual Review of Physical Chemistry*, 62(1):185–207, 2011. PMID: 21166541.

[12] Arya K. Babbush R. et al Arute, F. Quantum supremacy using a programmable superconducting processor. https://doi.org/10.1038/s41586-019-1666-5. 7/4/2020.

[13] William J.Munro Kae Nemoto Simon J.Devitt. Quantum error correction for beginners. https://arxiv.org/pdf/0905.2794.pdf. 17/11/2019.

[14] A. Yu. Kitaev. Fault-tolerant quantum computation by anyons. https://arxiv.org/abs/quant-ph/9707021. 27/11/2019.

[15] M. F. Araujo de Resende. A pedagogical overview on 2d and 3d toric codes and the origin of their topological orders. https://arxiv.org/abs/1712.01258. 27/11/2019.

[16] Peter Gács. Reliable computation with cellular automata. *J. Comput. Syst. Sci.*, 32(1):15–78, 1986.

[17] J. von Neumann. Lectures on probabilistic logics and the synthesis of reliable organisms from unreliable components. https://www.sns.ias.edu/pitp2/2012files/Probabilistic_Logics.pdf. 1/4/2020.

[18] Dan Browne. Lectures on topological codes and quantum computation. https://sites.google.com/site/danbrowneucl/teaching/lectures-on-topological-codes-and-quantum-computation. 24/3/2020.

[19] Sean D Barret Fern H E Watson. Logical error rate scaling of the toric code. https://iopscience.iop.org/article/10.1088/1367-2630/16/9/093045/pdf. 19/3/2020.

[20] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*.

[21] Austin G.Fowler. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $o(1)$ parallel time. https://arxiv.org/abs/1307.1740. 24/3/2020.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning:An Introduction*.