Quantum Error Correction Surface Codes

A Project Report
Submitted for the degree of

Bachelor of Technology

in

Engineering Physics

Submitted by:

Tejas Naik

Under the supervision of:

Dr. Sugata Gangopadhyay
Dr. Ajay Wasan

Department of Physics

Indian Institute of Technology, Roorkee

Roorkee-247667 (India)

April 2022

Candidate's Declaration

I hereby declare that the work, which is being presented in this report entitled **"Quantum Error Correction Surface Codes"** and submitted in partial fulfilment of the requirements of the degree of Bachelor of Technology in Engineering Physics as an authentic record of my own work carried out during the period of August 2021 to April 2022 under the supervision of **Dr. Sugata Gangopadhyay, Department of Computer Science and Engineering, IIT Roorkee**. The matter contained in this report has not been submitted by me for the award of any other degree in the institute or any other Institute/University.

[illegible]

Tejas Naik
B.Tech Engineering Physics
18311017

Certificate

This is to certify that the project entitled **"Quantum Error Correction Surface Codes"** is a bonafide work carried out by Tejas Naik, B.Tech final year student IIT Roorkee, in the partial fulfilment for the requirements of the award of the degree Bachelor of Technology in Engineering Physics under my guidance.

Dr. Sugata Gangopadhyay
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

Dr. Ajay Wasan
Professor
Department of Physics
Indian Institute of Technology Roorkee

Quantum Error Correction Surface Code

Tejas Naik
BTech. Project Report
Enrollment Number:18311017
Supervisor: Prof. Sugata Gangopadhyay
Co-Supervisor: Prof. Ajay Wasan

***Abstract*—The report starts with a brief introduction to error correction schemes such as Repetition Code and Shor's 9 qubit code. Further we discuss about surface codes, one of the most important error correction scheme in Quantum Error Correction. We discuss in depth the Blossom algorithm for finding the minimum weight perfect matching of a graph and how it can be used as a decoder for the surface code. We implement the surface code architecture in IBM Qiskit and analyze the performance of these codes as measured by logical error rates for different error models as a function of physical error rates. We also try to implement the architecture in QSim and list out the limitations of the simulator.**

I. Introduction

The key idea of any error correction scheme is to encode the message that we wish to protect against the effects of noise by adding some redundant information to the message. That way, even if some of the information in the encoded message is corrupted by noise, there will be enough redundancy in the encoded message that it is possible to recover or decode the message so that all the information in the original message is recovered. Consider the example of a classical channel through which we wish to send a bit. The channel has a bit flip probability of $p > 0$. A simple means to protect the bit would be to replace the bit with 3 copies of itself i.e.-

$$0 \rightarrow 000$$

$$1 \rightarrow 111$$

We can think of bit strings 000 and 111 as logical 0 and logical 1 respectively. When acted by the error, at the receiver end one or more bits could be flipped. The type of decoding used for this code is the majority decoding where we take majority votes of the 3 bits to decide if it is was a logical 0 or logical 1. Thus we can see that this type of encoding would failed when error occurs on 2 or more bits. The probability that two or more bits flips occur i.e. a logical error occurs is-

$$p_e = 3p^2 - 2p^3$$

Without encoding, the probability of error was p. Thus for this repetition scheme to be more reliable,

$$p_e < p$$

$$3p^2 - 2p^3 < p$$

$$p < \frac{1}{2}$$

II. Repetition Code for Qubits

These are the codes obtained by repeating qubits just like bits to protect against some noises. The key difference between Quantum Repetition Code and Classical Repetition Code is that in classical case bits are checked to take majority voting, but in quantum case syndrome measurements are taken which indirectly tells detects the error without collapsing the quantum state.

A. The 3-Qubit Bit Flip Code

We can extend a similar encoding for the qubit case. The encoding thus becomes-

$$|0\rangle \rightarrow |000\rangle$$

$$|1\rangle \rightarrow |111\rangle$$

Thus our encoding for a single logical qubit would look like $\alpha|0\rangle_L + \beta|1\rangle_L = \alpha|000\rangle + \beta|111\rangle$. As for the classical case this encoding would be able to detect bit flip errors(Pauli X) error on a single qubit and thus is called as the bit flip code. Error detection is done by a projective measurement over the four projection operators:

$$P_0 = |000\rangle\langle 000| + |111\rangle\langle 111|$$

$$P_1 = |100\rangle\langle 100| + |011\rangle\langle 011|$$

$$P_2 = |010\rangle\langle 010| + |101\rangle\langle 101|$$

$$P_3 = |001\rangle\langle 001| + |110\rangle\langle 110|$$

We can see that we would get for only 1 value of j, $\langle\psi|P_j|\psi\rangle = 1$ for $j = 0, 1, 2, 3$ would denote no X errors, X error on qubit 1, X error on qubit 2, X error on qubit 3 respectively for any quantum state $|\psi\rangle$. Here we can see that the projective measurement does not change the state of the system.

B. The 3-Qubit Phase Flip Code

Apart from the bit flip noise we can also have phase flip noise. In this error model, a qubit is left alone with probability $1-p$ and with probability p the phase flip operator Z is applied to the qubit. Thus for encoding purpose we use-

$$|0\rangle_L = |+++\rangle$$

$$|1\rangle_L = |---\rangle$$

The motive of using Hadamard basis for encoding is that in this basis, the Z operator flips $|+\rangle$ to $|-\rangle$ and vice versa. Thus it acts as a bit flip channel in hadamard basis.

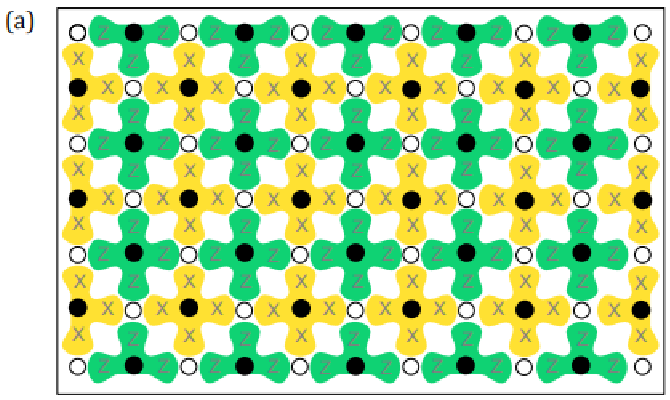


Fig. 1. Surface Code Lattice. Ref[3]

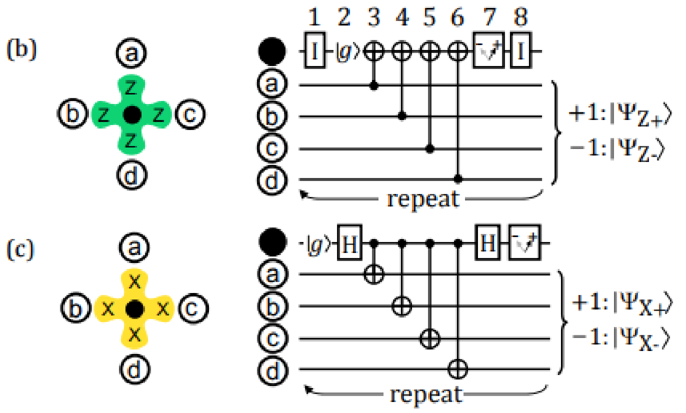


Fig. 2. Circuits for Measurement Syndromes. Ref[3]

C. Stabilizer Formalism

A quantum state $|\psi\rangle$ is said to be stabilized by a operator M if $M|\psi\rangle = |\psi\rangle$ i.e. $|\psi\rangle$ is the +1 eigenstate of the operator M. We can see that the 3-qubit bit flip codewords have stabilizers $Z \otimes Z \otimes I$, $I \otimes Z \otimes Z$ and any product of powers of these two operators. We call this two operators as the stabilizer generators of the code. Measurement of these observables gives us the error syndrome. For e.g. Consider a bit flip error acting on the first qubit of the bit flip error code. Thus the quantum state after the error would become $\alpha|100\rangle + \beta|011\rangle$. The stabilizer $Z \otimes Z \otimes I$ would give an eigenvalue of -1 and $I \otimes Z \otimes Z$ would give an eigenvalue of +1. Thus we are able to detect the location of error i.e. on qubit 1 and hence we can apply a correction X on qubit 1 so that the error is corrected.

Similarly for the phase flip code, we get the stabilizers generators as $X \otimes X \otimes I$ and $I \otimes X \otimes X$. We these two stabilizer measurements we can detect a phase flip occurring on a single qubit.

III. The Shor Code

As we can see from above, the bit flip code can only detect a bit flip error on a qubit and the phase flip code can detect only a phase flip error but not both. Thus we move to Shor's Code, which is a combination of bit flip and phase flip code. We first encode the qubit using phase flip i.e. $|0\rangle_L = |+++\rangle$ and $|1\rangle_L = |---\rangle$. Now each of these qubits are encoded using bit flip code. The resultant code we get-

$$|0\rangle_L = \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}$$

$$|1\rangle_L = \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}$$

The stabilizers for the Shor's code are-

$$Z_1Z_2, Z_2Z_3, Z_4Z_5, Z_5Z_6, Z_7Z_8, Z_8Z_9$$

$$X_1X_2X_3X_4X_5X_6, X_4X_5X_6X_7X_8X_9$$

We can see that combination of two different Z stabilizers of weight 2 (by weight, we mean the number of qubits on which the stabilizer acts non trivially) can be used to detect X errors and the weight-6 X stabilizers could a phase flip error on any of the qubit in the 3 qubit block.(i.e. qubits 1,2,3 or 4,5,6 or 7,8,9).

It happens that the Shor's code protects against not only bit flip and phase flip errors but also against any completely arbitrary error provided it affects only a single qubit. If we represent an error η in its operator-sum representation, then for any logical qubit $\alpha|0\rangle_L + \beta|1\rangle_L$, we would have,

$$\eta(\rho) = \sum_i E_i \rho E_i^\dagger$$

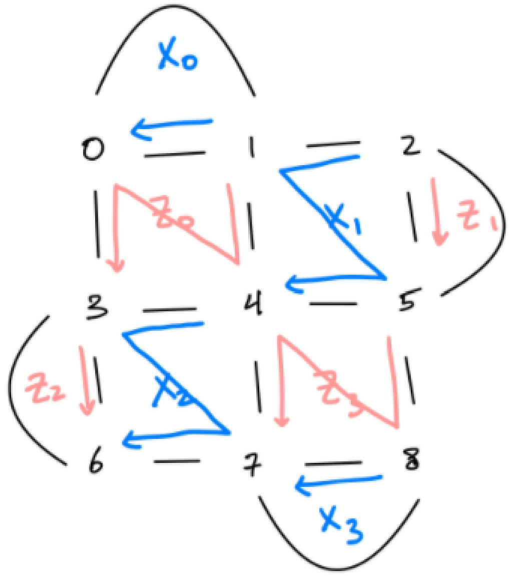
where ρ is the density matrix of the quantum state and $\{E_i\}$ are the operation elements. As this acts non-trivially only on a single qubit, we can write each of these operation elements as a linear combination of Pauli matrices i.e. $E_i = e_0 I + e_1 X_j + e_2 Z_j + e_3 X_j Z_j$ where the pauli operators act on a single qubit j. Thus we can write the quantum state after the application of E_i as a linear combination of $\psi\rangle$, $X_j\psi\rangle$, $Z_j\psi\rangle$ and $X_jZ_j\psi\rangle$. Measuring the syndrome collapses the superposition to any of the four possible states and thus we can apply suitable correction operator to get $|\psi\rangle$.

IV. Surface Codes

Surface Codes is a family of Quantum Error Correcting Codes defined on a 2D lattice of qubits. A surface code is formed by truncating a toric code and rotating the lattice of qubits by 45^o. Consider the surface code given in (fig.1). Here the open circles are the data qubits used for encoding a single logical qubit and the dark circles are the syndrome qubits used for measuring the syndrome measurements for error detection. The circuits for measuring the Z and X measurement syndromes of its corresponding data qubits neighbours are also given.(fig.2)

For our purpose we would be focusing on surface codes having an odd parameter 'd' which states the dimension of the lattice i.e. number of data qubits on any side of the lattice. Thus the number of data qubits for a surface code with distance 'd' are 'd^2' and total number of syndrome qubits are $d^2 - 1$.

Its important to note that, while taking X and Z syndrome measurements the direction of stabilizer measurement in each cell should be as directed in the graph below.



Here we consider the distance between consecutive data qubits to be of 1 unit. The data qubit indexed 0 is kept at origin and the direction to the horizontal right is taken as $+x$ axis and vertically downwards as the $+y$ axis.The coordinates of the syndrome qubits thus can be written as, for Z_0 it is (0.5.0.5) and $X_1 = (0.5, 1.5)$. The co-ordinates of other syndromes can be calculated similarly. One of the main advantages which surface codes provide is its ability to detect errors locally with the help of syndrome measurement blocks.

A. *Quiescent State*

The measure Z syndrome basically forces its neighboring data qubits a,b,c,d into an eigenstate of operator $Z_a Z_b Z_c Z_d$ measuring the Z-stabilizer. Similarly, the X syndrome measures the X-stabilizer. For a stabilizer measurement round, all the stabilizer measurements are taken, where each stabilizer block is measured in a fashion mentioned before. A concurrent measurement of all the stabilizer syndromes collapses the data qubits neighboring any syndrome block randomly into either the $+1$ or -1 eigenbasis of the stabilizer. Thus the combined state of all the data qubits results into a state known as the quiescent state. In absence of any errors, subsequent rounds of stabilizer measurements would not change the quiescent state. This is because all the X and Z syndrome measurements are commuting operators. For e.g. consider a Z stabilizer measurement block on qubits a,b,c,d and X stabilizer measurement block on qubits a,b,e,f. Thus the commutator of these 2 stabilizer becomes:

$$\begin{aligned}[Z_a Z_b Z_c Z_d, X_a X_b X_e X_f] &= (Z_a X_a Z_b X_b) Z_c Z_d X_e X_f \\ &-(X_a Z_a X_b Z_b) Z_c Z_d X_e X_f \\ &= 0\end{aligned}$$

For a given surface code, there are a large number of quiescent states. For n syndrome qubits, there are 2^n possible quiescent states (each syndrome qubit can project its neighboring data qubit into $+$ or $-$ eigenstate of the stabilizer). The stabilizer measurement randomly projects the data qubits onto one of these quiescent states.

V. Mapping the syndrome decoding to the MWPM problem

Consider a surface code with total number of data qubits equal to n. Let us consider an error model such that an error E occurs on a qubit with probability p. Then the likelihood of that error E occurring on m qubits is given by-

$$p(E) = p^m (1-p)^{n-m} = (1-p)^n \left(\frac{p}{1-p}\right)^m$$

Thus for typical scenarios when $p < 0.5$, the most probable error is the error with the minimum weight. Thus we take all the syndrome nodes which show a change in syndrome measurement with respect to the syndrome measurement of the quiescent state and make a fully connected graph known as an error graph. The weights of the edges between these nodes is the minimum weight Pauli error joining the two syndrome nodes. Now the problem is to find the minimum weight perfect matching to these graph, so that suitable corrections could be applied to the matches obtained. In order to solve this problem, we will learn about the Edmund Blossom's Algorithm for finding the minimum weight perfect matching.

VI. Edmund Blossom's Minimum Weight Perfect Matching Algorithm

Consider a graph G(V,E). A set of edges M is said to be matching of G if no node of G is incident with more than one edge in M. Given a matching M, we say that M covers a node v (or that v is M-covered) if some edge of M is incident with v. Otherwise, v is M-exposed.A perfect matching is a one that covers all the nodes. Given a matching M of G, a path P is M-alternating if its edges are alternately in and not in M. If, in addition, the end nodes of P are distinct and are both M-exposed, then P is an M-augmenting path.

Consider a graph G(V,E) and a matching M. We take an M-exposed vertex r as a root and construct an M-alternating tree T. We construct 2 new sets B(T) and A(T) such that vertices at even number of edges from root are in B(T) and at odd edges are in A(T). We define the following 2 operations:

1. Extension: If $v \epsilon B(T)$ such that there exists $vw \epsilon E$ and w is M-covered with edge $wz \epsilon M$. Extend tree T by adding w to A(T) and z to B(T).

2. Augmenting: If $v \epsilon B(T)$ such that there exists $vw \epsilon M$ and w is not in the tree. Augment the tree T by flipping the edges on the path from root to w.

A. *LPP Formulation*

: Let the weights of edges be given by c_e for $e \epsilon E$. We define a vector x whose entry $x_e = 1$ if the edge e is in the matching and 0 otherwise. We define $x(\delta v)$ as the sum of x_e of all the edges incident on the vertex v. Thus for the minimum weight perfect matching problem, the optimization problem becomes-

$$Minimize \sum(c_e x_e) : \forall e \epsilon E$$

subject to,

$$x(\delta v) = 1, \forall v \epsilon V$$

$$x_e \epsilon 0, 1, \forall e \epsilon E$$

We see that the above optimization problem is an integer programming problem. In order to convert it to a LPP, we need to relax the integer constraint. Relaxing the integer constraint, we get,

$$Minimize \sum(c_e x_e) : \forall e \epsilon E$$

subject to,

$$x(\delta v) = 1, \forall v \epsilon V$$

$$x_e \geq 0 : \forall e \epsilon E$$

The optimal value we get from LPP is only equal to the original integer programming problem if and only if the graph is a bipartite graph. For a general graph, we need to add one more constraint to the LPP. We call a cut D of G odd if it is of the form $\delta(S)$ where S is an odd-cardinality set of nodes. If D is an odd cut and M is a perfect matching, then M must contain at least one edge from D. If C is a set of all odd cuts of G with atleast 3 vertices then we get,

$$x(D) \geq 1 : D \epsilon C$$

Thus the LPP becomes,

$$Minimize \sum(c_e x_e) : \forall e \epsilon E$$

subject to,

$$x(\delta v) = 1, \forall v \epsilon V$$

$$x(D) \geq 1 : D \epsilon C$$

$$x_e \epsilon 0, 1, \forall e \epsilon E$$

The dual of this problem can be written as,

$$Maximize \sum(y_v : v \epsilon V) + (y_D : D \epsilon C)$$

subject to,

$$y_v + y_w + \sum(Y_D : e \epsilon D \epsilon C) \leq c_e, \forall e = vw \epsilon E$$

$$Y_D \geq 0, \forall D \epsilon C$$

We define the slack variable for the first constraint as $\bar{c_e}(y, Y) = c_e - (y_v + y_w + \sum(Y_D : e \epsilon D \epsilon C))$. By complementary slackness theorem we get: $x_e > 0$ implies $\bar{c_e} = 0$ and $Y_D > 0$ implies $x(D) = 1$. For a given feasible solution (y, Y), we define an Equality set of edges $E_=$ which contains all the edges who have $\bar{c_e} = 0$.

Now for finding the MWPM, we start by choosing a random vertex and making it the root of our tree T. Then if we find an edge $e \epsilon E_=$ such that its ends $v \epsilon B(T)$ and w is an M-exposed node outside the tree T, Use vw to augment M and choose a different M-exposed node as the root. If we find an edge $e \epsilon E_=$ such that its ends $v \epsilon B(T)$ and w is an M-covered node outside the tree T, Use vw to extend M. If there are no equality edges remaining to either extend or augment the graph, we change the value of y so that for atleast one edge we can either extend or augment the matching. The rules are as follows: Let, $\epsilon_1 = \min(\bar{c_e}$: e joins in G a node in B(T) to a node not in V(T)). $\epsilon_2 = \min(c_e/2$: e joins in G, two nodes in B(T). $\epsilon_3 = \min(y_v : v \epsilon A(T)$, v is a pseudonode of G.

$$\epsilon = min(\epsilon_1 \epsilon_2 \epsilon_3)$$

Replace y_v by, $y_v + \epsilon$ if $v \epsilon B(T)$ and by $y_v - \epsilon$ if $v \epsilon A(T)$ and don't change otherwise.

In the updation rules above for y, we defined a pseudonode. For that we first define what a blossom is. If there exists an edge vw with $v, w \epsilon B(T)$, then this edge together with the path in T from v to w forms an odd circuit known as a blossom. We can shrink this blossom to a single node called a pseudonode with all the edges from the blossom to the nodes outside of blossom, now incident on the pseudonode. Therefore for the weighted case, for an edge joining 2 nodes $v, w \epsilon B(T)$, when using vw to shrink G,M and T, we also need to update the weights c_{vw} by $c_{vw} - y_v$ for edges $v \epsilon C$ and w not belonging to C, where C is the blossom(odd circuit). Also, the pseudonode formed by shrinking the blossom should be assigned $y_C = 0$.

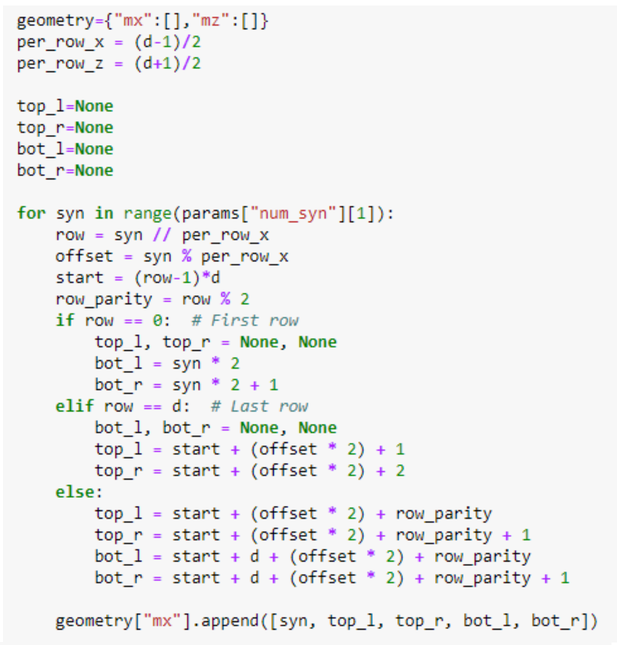
One last thing remaining is how to expand these shrinked blossom. For expanding a pseudonode, we look for pseudonodes in A(T) whose dual variable $y_v = 0$. We then expand the pseudonode and do the inverse of what we did in shrinking i.e. update the weights for edge st with $s \epsilon C$ and t not belonging to C by replacing c_{st} by $c_{st} + y_s$. A way to know that we have acheived the minimum weight perfect matching is when after augmenting an edge, there is no M-exposed node left in the graph G and also all the pseudonodes have been expanded. On the other hand for every edge e incident on $v \epsilon B(T)$, the other end is in A(T), with no pseudonodes in A(T), the graph has no perfect matching. This is because there is no way to update the tree T and the graph still has an M-exposed node which is the root of the tree.

VII. Implementation in IBM Qiskit

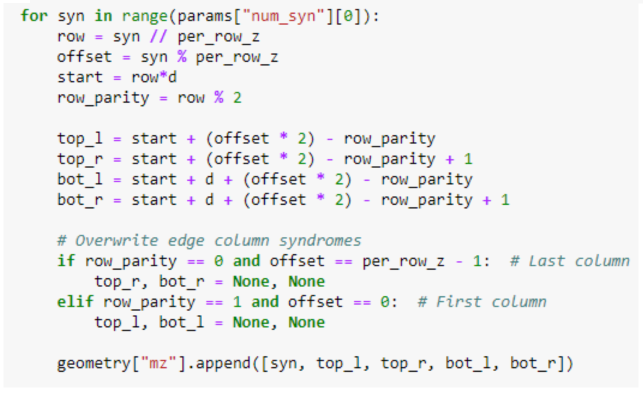
We now move to implementing this in IBM Qiskit. We will run the surface code architecture on the backend "Qasm Simulator" provided by Qiskit Aer. Qiskit doesn't have a predefined module for constructing the surface code. So, the first thing is to create the surface code architecture.

A. Constructing Surface Code Architecture

We define a function SurfaceCodeQubit which takes as inputs a dictionary of parameters. In this parameters we define the distance of surface code $'d'$, the round of measurement T, set initially to -1, the number of data qubits $'d^{2}'$, number of syndrome qubits $((d^2 - 1)/2, (d^2 - 1)/2)$ for the X and Z stabilizers respectively. Then we define quantum registers labelled as 'data qubits', 'mz' and 'mx' for the physical qubits and Z and X syndrome qubits respectively. In accordance with the geometry of the lattice we need to index the data qubits and syndrome qubits appropriately and create stabilizer blocks. For that we used the following function-

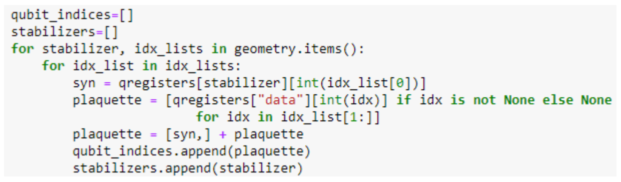


```
geometry={"mx":[],"mz":[]}
per_row_x = (d-1)/2
per_row_z = (d+1)/2

top_l=None
top_r=None
bot_l=None
bot_r=None

for syn in range(params["num_syn"][1]):
    row = syn // per_row_x
    offset = syn % per_row_x
    start = (row-1)*d
    row_parity = row % 2
    if row == 0:  # First row
        top_l, top_r = None, None
        bot_l = syn * 2
        bot_r = syn * 2 + 1
    elif row == d:  # Last row
        bot_l, bot_r = None, None
        top_l = start + (offset * 2) + 1
        top_r = start + (offset * 2) + 2
    else:
        top_l = start + (offset * 2) + row_parity
        top_r = start + (offset * 2) + row_parity + 1
        bot_l = start + d + (offset * 2) + row_parity
        bot_r = start + d + (offset * 2) + row_parity + 1

    geometry["mx"].append([syn, top_l, top_r, bot_l, bot_r])
```



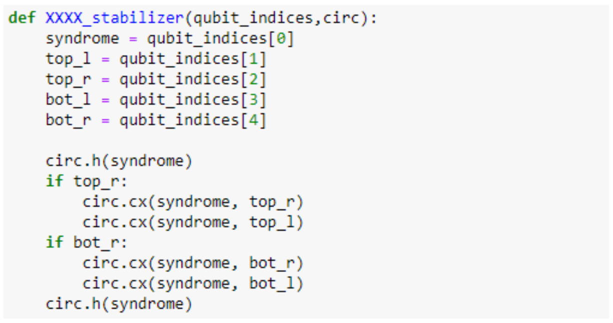
```
for syn in range(params["num_syn"][0]):
    row = syn // per_row_z
    offset = syn % per_row_z
    start = row*d
    row_parity = row % 2

    top_l = start + (offset * 2) - row_parity
    top_r = start + (offset * 2) - row_parity + 1
    bot_l = start + d + (offset * 2) - row_parity
    bot_r = start + d + (offset * 2) - row_parity + 1

    # Overwrite edge column syndromes
    if row_parity == 0 and offset == per_row_z - 1:  # Last column
        top_r, bot_r = None, None
    elif row_parity == 1 and offset == 0:  # First column
        top_l, bot_l = None, None

    geometry["mz"].append([syn, top_l, top_r, bot_l, bot_r])
```

We create two lists 'stabilizers' and 'qubit indices' of equal length, where the i^{th} element of 'stabilizers' tells whether the stabilizer syndrome to be measured on the qubits in the i^{th} list of the 'qubit indices' are X or Z syndromes.

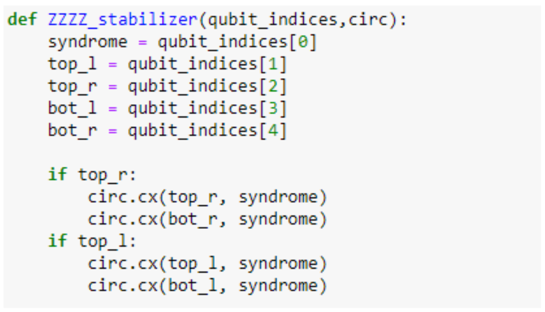


```
qubit_indices=[]
stabilizers=[]
for stabilizer, idx_lists in geometry.items():
    for idx_list in idx_lists:
        syn = qregisters[stabilizer][int(idx_list[0])]
        plaquette = [qregisters["data"][int(idx)] if idx is not None else None
                            for idx in idx_list[1:]]
        plaquette = [syn,] + plaquette
        qubit_indices.append(plaquette)
        stabilizers.append(stabilizer)
```

Now we define the X and Z stabilizer syndrome measurements. We also need to incorporate the weight-2 stabilizers on some qubits. This is given by-

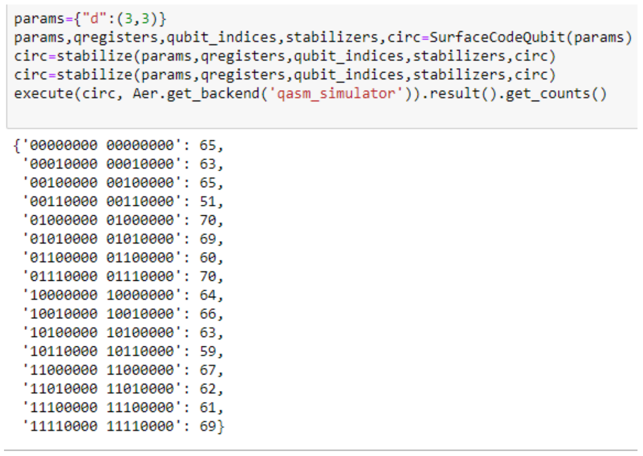


```
def XXXX_stabilizer(qubit_indices,circ):
    syndrome = qubit_indices[0]
    top_l = qubit_indices[1]
    top_r = qubit_indices[2]
    bot_l = qubit_indices[3]
    bot_r = qubit_indices[4]

    circ.h(syndrome)
    if top_r:
        circ.cx(syndrome, top_r)
        circ.cx(syndrome, top_l)
    if bot_r:
        circ.cx(syndrome, bot_r)
        circ.cx(syndrome, bot_l)
    circ.h(syndrome)
```



```
def ZZZZ_stabilizer(qubit_indices,circ):
    syndrome = qubit_indices[0]
    top_l = qubit_indices[1]
    top_r = qubit_indices[2]
    bot_l = qubit_indices[3]
    bot_r = qubit_indices[4]

    if top_r:
        circ.cx(top_r, syndrome)
        circ.cx(bot_r, syndrome)
    if top_l:
        circ.cx(top_l, syndrome)
        circ.cx(bot_l, syndrome)
```

At the start of the circuit, all the qubits will be in $|00..0\rangle$ state and thus won't be in an eigenstate of the surface code. Thus at start, we set the parameter $T = -1$ so that after one round of stabilizer, the qubits' state is transformed to the quiescent state after which in absence of errors the syndrome measurements of successive rounds would remain unchanged. To verify this let us run the syndrome measurement 2 times:



```
params={"d":(3,3)}
params,qregisters,qubit_indices,stabilizers,circ=SurfaceCodeQubit(params)
circ=stabilize(params,qregisters,qubit_indices,stabilizers,circ)
circ=stabilize(params,qregisters,qubit_indices,stabilizers,circ)
execute(circ, Aer.get_backend('qasm_simulator')).result().get_counts()
```

```
{'00000000 00000000': 65,
 '00010000 00010000': 63,
 '00100000 00100000': 65,
 '00110000 00110000': 51,
 '01000000 01000000': 70,
 '01010000 01010000': 69,
 '01100000 01100000': 60,
 '01110000 01110000': 70,
 '10000000 10000000': 64,
 '10010000 10010000': 66,
 '10100000 10100000': 63,
 '10110000 10110000': 59,
 '11000000 11000000': 67,
 '11010000 11010000': 62,
 '11100000 11100000': 61,
 '11110000 11110000': 69}
```

Here, the qiskit formatting is [syndrome1 syndrome0] where bits in syndrome0 are 'X(N-1)X(N-2)...X(0)Z(N-1)Z(N-2)....Z(0)' where N is the number of X or Z syndrome qubits. As noted while learning about quiescent state, each syndrome measurement collapses randomly to either $0(+)$ or $1(-)$ eigenstate, thus giving a probability distribution over different possible outcomes when the circuit is ran many number of times. We can see that the measurement counts for T=0 round and T=1 round are same. In presence of error, we can see that the bits change, and thus we need to XOR the syndrome measurements from the 2 rounds to extract the change in syndrome node measurement. Let's add a Pauli Y error acting on data qubit indexed 1.

```
params={"d":(3,3)}
params,qregisters,qubit_indices,stabilizers,circ=SurfaceCodeQubit(params)
circ=stabilize(params,qregisters,qubit_indices,stabilizers,circ)
circ.y(qregisters["data"][1])
circ=stabilize(params,qregisters,qubit_indices,stabilizers,circ)
execute(circ, Aer.get_backend('qasm_simulator')).result().get_counts()
```

```
{'00010001 00100000': 66,
 '00000001 00110000': 57,
 '00100001 00010000': 74,
 '00110001 00000000': 76,
 '01000001 01110000': 56,
 '01010001 01100000': 67,
 '01100001 01010000': 63,
 '01110001 01000000': 62,
 '10000001 10110000': 58,
 '10010001 10100000': 74,
 '10100001 10010000': 67,
 '10110001 10000000': 63,
 '11000001 11110000': 64,
 '11010001 11100000': 60,
 '11100001 11010000': 56,
 '11110001 11000000': 61}
```

Here we can see that the syndrome measurement of Z_0, X_0 and X_1 has flipped with respect to the quiescent state measurement thus signaling the occurrence of an error. In order to calculate the XOR of these syndrome measurements and to calculate the coordinates of syndromes which have flipped we use the parse readout function below-

```
readouts=list(counts.keys())
readouts_string=readouts[0]
readout=parse_readout(params,readouts_string)
print(readout)
```

```
{'X': [(0.0, -0.5, 0.5), (0.0, 0.5, 1.5)], 'Z': [(0.0, 0.5, 0.5)]}
```

Using these syndromes which have detected errors, we create an error graph, where the nodes are the syndrome qubits obtained from parse readout and weight of the edges connecting the nodes is the minimum weight error chain joining the two nodes. For this, we first create syndrome graphs which contains all the nodes of a particular syndrome type (X or Z) which are connected to syndrome nodes of adjacent blocks by an edge of weight 1. We do this by the function 'Syndrome-Graph' which gives the syndrome graph 'S' and the node-map which holds the co-ordinates of all the nodes.

```
def Syndrome_Graph(params):
    syndrome_graph_keys=["X","Z"]
    S={}
    node_map={}
    for syndrome_graph_key in syndrome_graph_keys:
        S[syndrome_graph_key]=nx.Graph()
        node_map[syndrome_graph_key]={}
    make_syndrome_graph(S,syndrome_graph_keys,node_map,params)
    return S,node_map

def make_syndrome_graph(S,syndrome_graph_keys,node_map,params):
    start_nodes = {"Z": (0.5, 0.5), "X": (0.5, 1.5)}
    for syndrome_graph_key in syndrome_graph_keys:
        start_node = start_nodes[syndrome_graph_key]
        node_map[syndrome_graph_key][start_node] = S[syndrome_graph_key].add_node(start_node)
        populate_syndrome_graph(start_node, [], syndrome_graph_key, 1,S,node_map,params)
```

This function uses a recursive function 'populate-syndrome-graph' where given a node in the syndrome graph, we look at the neighboring syndrome nodes and check if they are valid (syndrome nodes at boundary will not have 4 neighboring syndrome nodes) and populate the syndrome graph as given below-

```
def populate_syndrome_graph(current_node,visited_nodes,syndrome_graph_key,edge_weight:int,S,node_map,params):
    visited_nodes.append(current_node)
    neighbors = []
    i = current_node[0]  # syndrome node x coordinate
    j = current_node[1]  # syndrome node y coordinate
    neighbors.append((i - 1, j - 1))  # up left
    neighbors.append((i + 1, j - 1))  # down left
    neighbors.append((i - 1, j + 1))  # up right
    neighbors.append((i + 1, j + 1))  # down right

    normal_neighbors = [n
                        for n in neighbors
                        if valid_syndrome(n, syndrome_graph_key,params)
                        and (n[0], n[1]) not in visited_nodes]

    if not normal_neighbors:
        return

        # add neighbors
    for target_node in normal_neighbors:
        if target_node not in S[syndrome_graph_key].nodes():
            # add target_node to syndrome subgraph if it doesn't already exist
            node_map[syndrome_graph_key][target_node] = S[syndrome_graph_key].add_node(target_node)

            S[syndrome_graph_key].add_edge(node_map[syndrome_graph_key][current_node],
                                           node_map[syndrome_graph_key][target_node],
                                           weight=edge_weight)  # add edge between current_node and target_node

    for target in normal_neighbors:
        populate_syndrome_graph(target, visited_nodes, syndrome_graph_key, edge_weight,S,node_map,params)
```

In order to make the error graph, we need to calculate the weight of the edges. For that, I used the networkx library function floyd_warshall_numpy which calculates the shortest path length between two nodes. We operate the floyd_warshall_numpy function on the syndrome graph to compute the weights.The 'error-graph' thus can be made by the following function.

```
def make_error_graph(nodes,S,syndrome_graph_key,node_map,params):
    node_map_error={}
    error_graph = nx.Graph()


    # add all nodes to error_graph
    for node in nodes:
        if node not in error_graph.nodes():
            node_map_error[node] = error_graph.add_node(node,pos=node)

    shortest_distance_mat=nx.floyd_warshall_numpy(S[syndrome_graph_key])


    for source, target in combinations(nodes, 2):
        nodes_whole=list(node_map[syndrome_graph_key].keys())
        i=Index_(nodes_whole,source)
        j=Index_(nodes_whole,target)
        distance = shortest_distance_mat[i][j]
        error_graph.add_edge((source), (target), weight=float(-1*distance))
    return error_graph
```

Here we can notice that we have taken the edge weights to be the negative of the distance calculated from floyd_warshall_numpy function. This is because, while running the MWPM decoder we will use the max weight matching function from the networkx library. After running this for the problem we discussed before, the matches of nodes we get are-

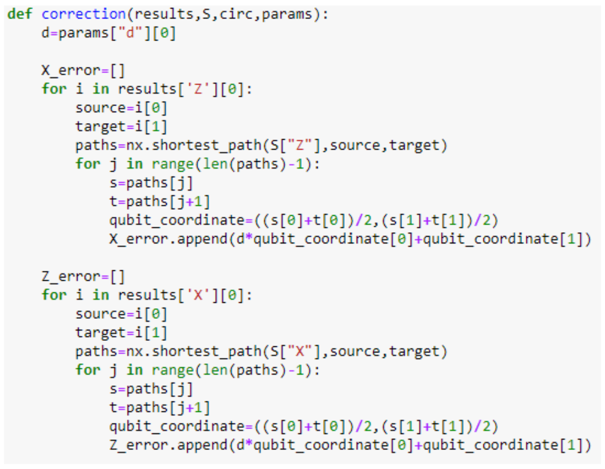
```
S,node_map=Syndrome_Graph(params)
Decoder(readout,S,node_map,params)
```

```
{'X': [{('(0.0, -0.5, 0.5)', '(0.0, 0.5, 1.5)')}], 'Z': [set()]}
```

Here we can see that the Z matches contains an empty set. This is because this method will only work for even number of syndrome nodes and thus no matching can be found for Z syndromes as there is only 1 Z-syndrome in error graph [(0.5,0.5)]. In order to overcome this, we move to the concept of virtual nodes.

B. *Correction Operator*

Before moving on to virtual nodes, we will write a function to apply correction operators on appropriate data qubits according to the matches obtained of syndrome nodes from the MWPM decoder. For each syndrome graph key (X and Z), we find the shortest path in the syndrome graph between the nodes in a match using the shortest_path function provided by networkx. Once we get the shortest path we apply a suitable correction operator (X for Z-syndrome nodes) on the data qubits in between the syndrome nodes of the shortest path. For a surface code of parameter 'd', the index of a data qubit with coordinates (x, y) is $d.x + y$. The function to add the correction operator to the circuit can thus be written as-



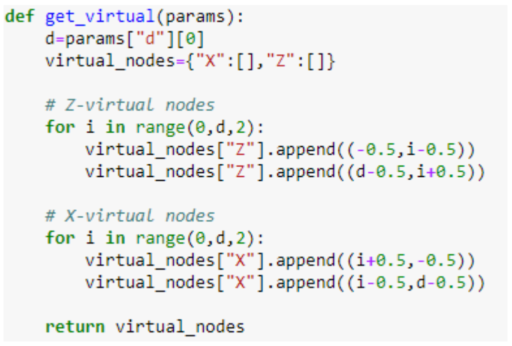
```
def correction(results,S,circ,params):
    d=params["d"][0]

    X_error=[]
    for i in results['Z'][0]:
        source=i[0]
        target=i[1]
        paths=nx.shortest_path(S["Z"],source,target)
        for j in range(len(paths)-1):
            s=paths[j]
            t=paths[j+1]
            qubit_coordinate=((s[0]+t[0])/2,(s[1]+t[1])/2)
            X_error.append(d*qubit_coordinate[0]+qubit_coordinate[1])

    Z_error=[]
    for i in results['X'][0]:
        source=i[0]
        target=i[1]
        paths=nx.shortest_path(S["X"],source,target)
        for j in range(len(paths)-1):
            s=paths[j]
            t=paths[j+1]
            qubit_coordinate=((s[0]+t[0])/2,(s[1]+t[1])/2)
            Z_error.append(d*qubit_coordinate[0]+qubit_coordinate[1])
```

The X_error and Z_error lists contains the indices of data qubits on which the respective correction operator has to be applied.

C. *Virtual Nodes*

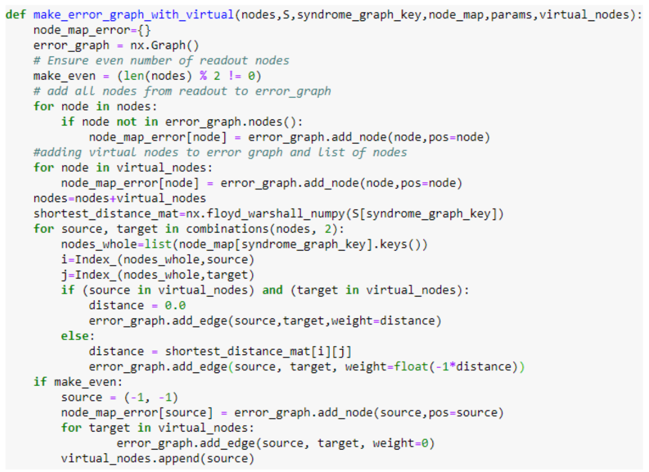
Consider a surface code with $d = 5$ of fig.11 at the end of the document. Let us consider a case where after performing stabilizer measurements, we get the triggered error syndromes as Z_0 and Z_{11}. If we ran our MWPM decoder, it will match the syndromes Z_0 and Z_{11}. The correction operator would then apply correction operators on the shortest path connecting the two nodes i.e. on data qubits 6,12 and 18 and thus would apply a Pauli-X correction $X_6X_{12}X_{18}$ which is a weight-3 operator. For this case, we can figure out that the most probable error in reality would have been X_0X_{24} which is a weight-2 Pauli error. Even after using the minimum weight perfect matching decoder, we are not getting the most probable matching.

The second issue is when we get an odd number of triggered syndromes as we discussed in previous section, for which a perfect matching could not be found. In order to overcome these two difficulties we introduce virtual nodes for each stabilizer syndrome type. These nodes are not physical qubits and we add them to our graphs while solving for the MWPM problem. As seen in the fig.? the virtual Z nodes are kept at the top and bottom of the surface code lattice, while the virtual X nodes are kept at the left and right of the surface code lattice. In order to get the coordinates of the virtual nodes, we can run the following code:



```
def get_virtual(params):
    d=params["d"][0]
    virtual_nodes={"X":[],"Z":[]}

    # Z-virtual nodes
    for i in range(0,d,2):
        virtual_nodes["Z"].append((-0.5,i-0.5))
        virtual_nodes["Z"].append((d-0.5,i+0.5))

    # X-virtual nodes
    for i in range(0,d,2):
        virtual_nodes["X"].append((i+0.5,-0.5))
        virtual_nodes["X"].append((i-0.5,d-0.5))

    return virtual_nodes
```

The procedure for creating the syndrome graphs remains the same with the only changes that in populate_syndrome_graph function, while adding the edges between neighboring nodes, we also consider these virtual nodes. For making the error graphs, we will take the edge weight between any two virtual nodes as 0 as we want these virtual nodes to get connected only after a minimum weight perfect matching is found for the normal syndrome nodes. One last thing to consider is when the number of triggered syndrome nodes are not even. As the number of virtual nodes are even, we need to add one more virtual node to the error graph, so that the total number of nodes are even and thus the decoder can find a perfect matching. The function to calculate the error graph with virtual nodes can be written as-



```
def make_error_graph_with_virtual(nodes,S,syndrome_graph_key,node_map,params,virtual_nodes):
    node_map_error={}
    error_graph = nx.Graph()
    # Ensure even number of readout nodes
    make_even = (len(nodes) % 2 != 0)
    # add all nodes from readout to error_graph
    for node in nodes:
        if node not in error_graph.nodes():
            node_map_error[node] = error_graph.add_node(node,pos=node)
    #adding virtual nodes to error graph and list of nodes
    for node in virtual_nodes:
        node_map_error[node] = error_graph.add_node(node,pos=node)
    nodes=nodes+virtual_nodes
    shortest_distance_mat=nx.floyd_warshall_numpy(S[syndrome_graph_key])
    for source, target in combinations(nodes, 2):
        nodes_whole=list(node_map[syndrome_graph_key].keys())
        i=Index_(nodes_whole,source)
        j=Index_(nodes_whole,target)
        if (source in virtual_nodes) and (target in virtual_nodes):
            distance = 0.0
            error_graph.add_edge(source,target,weight=distance)
        else:
            distance = shortest_distance_mat[i][j]
            error_graph.add_edge(source, target, weight=float(-1*distance))
    if make_even:
        source = (-1, -1)
        node_map_error[source] = error_graph.add_node(source,pos=source)
        for target in virtual_nodes:
                error_graph.add_edge(source, target, weight=0)
        virtual_nodes.append(source)
```

Let us consider a surface code of distance d=3 and errors Pauli Y acting on data qubit indexed 1 and Pauli Z acting on data qubit indexed 7. The syndrome graphs for the X and Z syndrome nodes are represented in fig.10 and fig.12 respectively at the end of document. The error graphs for the triggered syndromes for X and Z syndromes are shown in fig.3 and fig.4 respectively. After running the decoder we get the matches as:

$$X : [((0.5, 1.5), (-0.5, 0.5)), ((1.5, 0.5), (2.5, 1.5))]$$

$$Z : [((-0.5, -0.5), (0.5, 0.5))]$$

VIII. Logical Operators

Now that we have implemented the surface code architecture, we need to understand how to use the logical qubit which

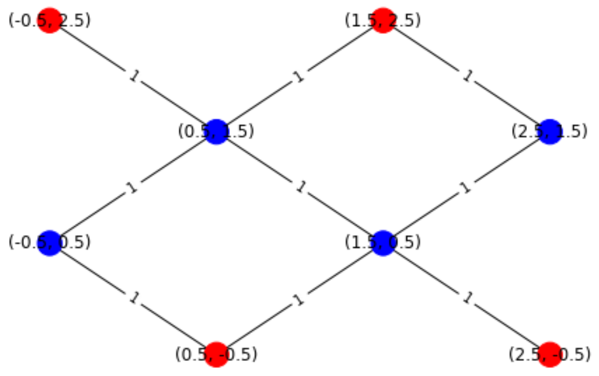


Fig. 3. X-Syndrome Graph. Blue nodes denotes normal syndromes, red denotes virtual.

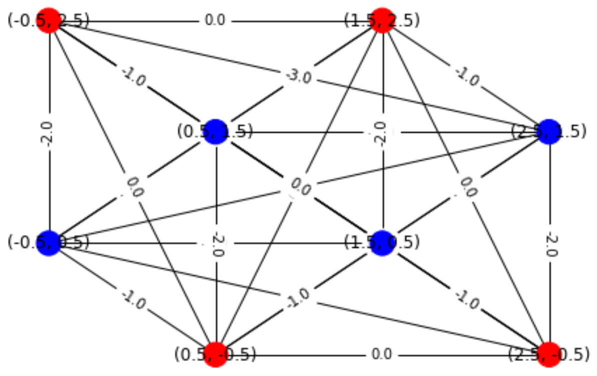


Fig. 4. X syndrome error graph. Blue nodes denotes normal syndromes, red denotes virtual.

we have created. What logical operations can we perform on this qubit and how can those be performed? How can we take out measurement readouts of the qubits? What are logical errors and how do they occur?

We start by addressing the question of what logical errors are. We know that the syndrome qubits detect chains of Pauli errors acting on neighboring data qubits. In the event of a Pauli error of weight-2 acting on a stabilizer block, that physical error would not be detected by the syndrome qubit. Consider a 4-data qubit block with data qubits a,b,c and d which is acted upon X stabilizer $X_aX_bX_cX_d$. Now consider a Z Pauli error Z_aZ_b of weight-2 acting on qubits a and b. If the initial quiscent state was $|\psi\rangle$, then we get:

$$\begin{aligned} X_aX_bX_cX_d(Z_aZ_b|\psi\rangle) &= Z_aZ_b(X_aX_bX_cX_d|\psi\rangle) \\ &= Z_aZ_b(X_{abcd}|\psi\rangle) \\ &= X_{abcd}(Z_aZ_b|\psi\rangle) \end{aligned}$$

Here X_{abcd} is the eigenvalue of the X stabilizer on the data qubit block a,b,c,d which remains unchanged even after a weight-2 Pauli Z error. But this error would be detected by the other 2 X syndrome blocks which contains data qubits a and b respectively. What would then be the length of the shortest Pauli Z error chain which would go unnoticed by all the X syndrome measurement qubits? If we look at the surface code lattice, any Pauli Z error chain with its ends at the left and right sides of the lattice would go unnoticed and would thus result in a logical error.

An application of logical error would change the state of data qubits to a different quiescent state without any change in syndrome measurements. Similarly consider a chain of Pauli X error with its ends at the top and the bottom side of the surface code lattice. Refer to fig.5 . We define the former as the logical Z_L error and the latter as logical X_L error. Thus if we apply a logical X_L to the quiescent state $|\psi\rangle$ we get,

$$|\psi\rangle_X = X_L|\psi\rangle$$

Here $|\psi\rangle_X$ would be trivially related to only if $|\psi\rangle$ is an eigenstate of X_L i.e. $|+\rangle_L$ or $|-\rangle_L$. Similarly the eigenstates of logical Z_L operator would be $|0\rangle_L$ and $|1\rangle_L$.

Now we can also figure out how to take logical readouts. As a convention we will take Z logical readout on the top most horizontal line of data qubits, whereas we will take X logical readout on the left most vertical line of data qubits. We readout these logical operators on the ancilla qubits with the following code:

```
# Logical Z readout
def readout_z_into_ancilla(params,qregisters,circ):
    d=params["d"][0]
    circ.reset(qregisters["ancilla"])
    for i in range(d):
        circ.cx(qregisters["data"][i],qregisters["ancilla"])

# Logical X readout
def readout_x_into_ancilla(params,qregisters,circ):
    d=params["d"][0]
    circ.reset(qregisters["ancilla"])
    circ.h(qregisters["ancilla"])
    for i in range(0,d*d,d):
        circ.cx(qregisters["ancilla"],qregisters["data"][i])
    circ.h(qregisters["ancilla"])
```

IX. Multi-Path Summation

In one of the previous sections, we tried to maximize the likelihood of a weight-m error E occurring on the data qubits and found the following relationship:

$$p(E) = p^m(1-p)^{n-m} = (1-p)^n \left(\frac{p}{1-p}\right)^m$$

When we map the triggered error syndromes to nodes in an error graph, we can express total weight of error connecting two matched syndrome nodes as a sum of weights of edges of the path connecting these nodes in the syndrome graph. Thus the overall likelihood can be expressed as a product over edge weights:

$$p(E) = (1-p)^n \prod_{e \in edges(E)} \left(\frac{p}{1-p}\right)^{|e|}$$

$$p(E)\alpha \prod_{e \in edges(E)} \left(\frac{p}{1-p}\right)^{|e|}$$

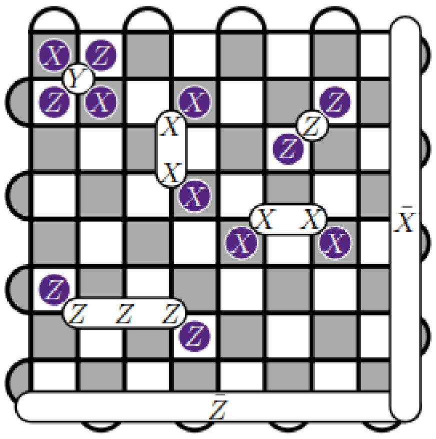


Fig. 5. Logical Operators. Ref[7]

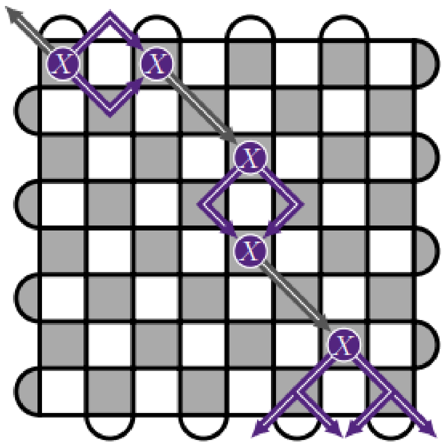


Fig. 6. Degeneracy.Ref[7]

In order to maximize the likelihood we take the log of the right hand side:

$$p(E) \sim \sum_{e \in edges(E)} |e| ln \left(\frac{p}{1-p} \right)$$

$$p(E) \sim - \sum_{e \in edges(E)} |e|$$

The negative sign comes from the assumption that,

$$\frac{p}{1-p} < 1$$

$$p < 0.5$$

For this assumption $ln(p/1-p)$ is negative. This thus means that maximizing likelihood is equivalent to minimizing the weights of the edges in the perfectly matched error graphs.

Now consider a case as given in fig.6 . We can see that for this specific error case the minimum weight perfect matching of the error graph would be of a Pauli error of weight 5 as marked by the grey lines. Here the top left syndrome node is connected to a virtual node. Our MWPM decoder would predict this as the most probable error event. But if we look at the fig. and consider the Pauli error of weight 6 connected by purple lines, we can see that for a matched syndrome nodes there are multiple paths which give the same weight of the error. In the fig we can see that the total number of degenerate paths having the same weight 6 Pauli errors connecting the syndrome nodes are 16 ($2 \times 2 \times 4$).

In order to account for degeneracy, the likelihood function would change to:

$$p(E) = \Omega(E)(1-p)^n \left(\frac{p}{1-p} \right)^m$$

where $\Omega(E)$ are the total number of degenerate paths for the error E of weight m. Let us calculate under what circumstances would the likelihood of weight-6 error be greater than a weight-5 error.

$$P(E_6) > P(E_5)$$

$$16 \times \left(\frac{p}{1-p} \right)^6 > \left(\frac{p}{1-p} \right)^5$$

$$\frac{p}{1-p} > \frac{1}{16}$$

Thus we can see that under these conditions, the likelihood of weight-6 error would be greater than that of weight-5 error.

In order to incorporate this effect into our MWPM decoder, we change weights of the edges in accordance with the degeneracy of the edges. We can write:

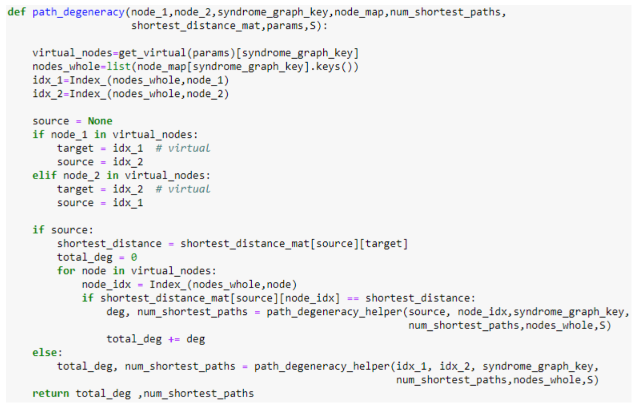
$$p(E) \alpha \prod_{e \in edges(E)} \Omega(e) \left(\frac{p}{1-p} \right)^{|e|}$$

Taking log on the right hand side we get,

$$p(E) \sim \sum_{e \in edges(E)} ln(\Omega(e)) + |e| ln \left(\frac{p}{1-p} \right)$$

$$p(E) \sim - \sum_{e \in edges(E)} |e| - \frac{ln(\Omega(e))}{ln \left(\frac{p}{1-p} \right)}$$

Thus we can see that updating the weights of the edges with the above formula would incorporate for the degeneracy in the error paths. For this, I have defined a function path_degeneracy which calculates the number of shortest paths between two nodes. Here we note that, if one of the nodes is a virtual node, we name it as target and then while calculating degeneracy we take all the virtual nodes whose distance from the other node is same as that of the target and add all the degeneracy together. For calculating the number of shortest paths, I used the function all_shortest_paths from networkx library which gives the list of all shortest paths between a pair source and target nodes. For calculating the number of shortest paths we count the number of shortest paths given by the all_shortest_paths function. The code for calculating can we written as below:



```python
def path_degeneracy(node_1,node_2,syndrome_graph_key,node_map,num_shortest_paths,
                    shortest_distance_mat,params,S):

    virtual_nodes=get_virtual(params)[syndrome_graph_key]
    nodes_whole=list(node_map[syndrome_graph_key].keys())
    idx_1=Index_(nodes_whole,node_1)
    idx_2=Index_(nodes_whole,node_2)

    source = None
    if node_1 in virtual_nodes:
        target = idx_1  # virtual
        source = idx_2
    elif node_2 in virtual_nodes:
        target = idx_2  # virtual
        source = idx_1

    if source:
        shortest_distance = shortest_distance_mat[source][target]
        total_deg = 0
        for node in virtual_nodes:
            node_idx = Index_(nodes_whole,node)
            if shortest_distance_mat[source][node_idx] == shortest_distance:
                deg, num_shortest_paths = path_degeneracy_helper(source, node_idx,syndrome_graph_key,
                                                                 num_shortest_paths,nodes_whole,S)
                total_deg += deg
    else:
        total_deg, num_shortest_paths = path_degeneracy_helper(idx_1, idx_2, syndrome_graph_key,
                                                                num_shortest_paths,nodes_whole,S)
    return total_deg ,num_shortest_paths
```

Once the degeneracy is calculated we can make appropriate changes to the weights while making the error graphs.

X. Error Analysis in IBM Qiskit

Now that we have defined a method to take logical readout, let us check how well the surface code is performing for error models. For a given physical error rate, what logical error rate does we achieve and how does this change with the surface code parameter d.

We will start by considering a IID (Independently Identical Distribution) error model where with probability $1-p$ an Identity operation occurs and with probability $p/2$ each Pauli errors X and Z occurs on all the data qubits. We apply this as an Identity block of gates acting on all the data qubits after the preparation of quiescent state and add the noise model with the help of Qiskit Aer library to only the 'id gates'. The procedure to find the logical error rates follows as below:

1) We first run the stabilizer round to achieve a quiescent state. After that noisy Identity gates are applied and one more round of stabilizer measurements are taken. We also take logical Z readout of the surface code at this point.

2) We run the decoder over the XORed syndrome nodes detected by the parse readout function and find the matches.

3) If any of these matches crosses the logical readout line, we flip the logical readout value obtained for that count. Do this until we consider all the matches obtained through the decoder. The final logical readout we get for that count will be the corrected logical value.

4) As we started with the logical $|0\rangle_L$ to begin with, the logical readout values that still remains 1 after step 3 would constitute of the logical errors.

5) Thus the logical error rate would simply be number of counts with logical readout value equal to 1 divided by total counts.

A. Pauli Noise

We start by considering the noise model of Pauli error on data qubits given by-

$$E(\rho) = (1-p)\rho + \frac{p}{2}X\rho X^{\dagger} + \frac{p}{2}Z\rho Z^{\dagger}$$

where ρ is a density matrix corresponding to a single qubit. All the data qubits are acted upon identical error model after the quiescent state is formed. We plot the logical error rate vs. the physical error rate (fig. 7) for different surface code

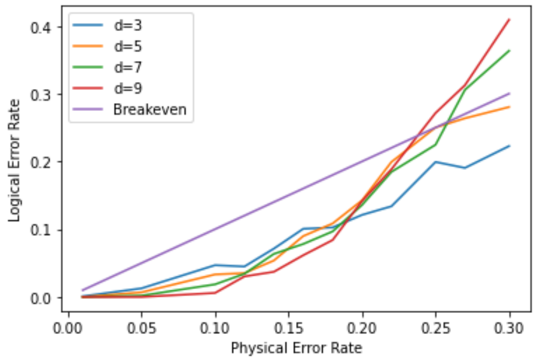


Fig. 7. Logical error rate plot for Pauli noise.

parameter d. We take the physical error rates as $[1e-2, 5*1e-2, 1e-1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.22, 0.25, 0.27, 0.3]$

The breakeven line denotes the physical error rate and thus we desire to have our logical error rates below the breakeven line. Here we can see that for lower values of physical error rates, the logical error rate decreases as we increase the value of d. But there seems to be some point near p=0.19 where all the curves for different values of d intersect. This point is known as the threshold error rate p_{th}. Beyond the p_{th} the trend gets flipped i.e. the logical error rate increases as we increase the value of d. Reference [3] provided an empirical formula for the relationship between logical error rate and physical error rate. If we denote P_L as the logical error rate and p as the physical error rate, the empirical formula says,

$$P_L \simeq 0.03\left(\frac{p}{p_{th}}\right)^{d_e}$$

where $d_e = (d+1)/2$. Taking $p_{th} = 0.19$ let us plot a graph using this empirical relation (fig. 8).

We see that both the graphs follows the same shape and trend. There are some irregularities in the surface code graph compared to the empirical one. This maybe due to the limited number of data points taken for plotting the graph as the computation becomes very expensive at large values of d and takes a lot of time to run. Moreover increasing the number of shots might also able to increase the smoothness of the graph.

B. Depolarizing Noise

A depolarizing noise is a completely positive trace preserving map and can be written as:

$$E(\rho) = (1-p)\rho + p\frac{I}{2}$$

This means that with probability $1-p$ the channel leaves the state of the qubit unaltered and with probability p it converts the qubit to a maximally mixed state. We plot the logical error rate vs physical error rate (fig. 9) for the physical error rate values equal to- $[1e-2, 5*1e-2, 1e-1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.25, 0.3]$

Here we can see that we get the same trend as followed for the Pauli error model and the error threshold rate $p_th = 0.2$.

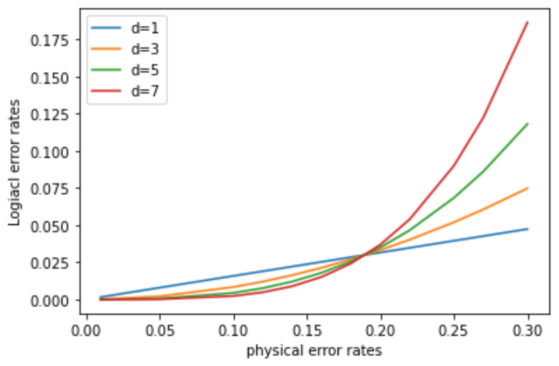


Fig. 8. Graph obtained by empirical formula taking $p_{th} = 0.19$.

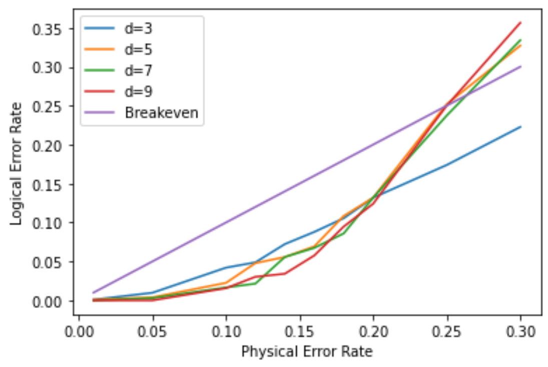


Fig. 9. Logical error rate plot for depolarizing noise.

We are thus able to reproduce the shapes and trends of logical error rate vs physical error rate as given in reference for the error models Pauli error and depolarizing channel.

XI. Implementation in QSim

QSim is a classical simulator framework made for performing quantum simulations on a density matrix based backend. The current version of QSim supports simulations containing upto 12 qubits. For the implementation of surface code, we know that even the smallest surface code with $d = 3$ requires 9 data qubits and 8 syndrome qubits which makes a total of 17 qubits. Thus in order to run the surface code architecture, we will reuse the same syndrome qubit for different stabilizer measurements by resetting the qubit to 0 after every stabilizer block measurement. Thus we will be using 9 qubits to encode data and 2 qubits for syndrome measurements (1 for each X and Z syndromes). The code for constructing the surface code architecture is the same as the one we followed for Qiskit.

For calculating syndrome measurement outcomes, we extract the probabilities of the syndrome qubit being 0 after each stabilizer measurement. This is because the density matrix simulator doesn't provide an option to obtain the counts as bit strings on a classical register as offered by Qasm simulator.

Now let us add a Pauli Y error on data qubit indexed 4 in fig. A Y error can be written as a combination of XZ errors (ignoring global phase). Thus due to this error we shall expect to see Z syndromes Z_0 and Z_3 to be triggered by the X error and similarly X syndromes X_1 and X_2 to be triggered by the Z error. Let us see the results of this computation in QSim.

```
Syndrome round 0
{'mz': [1.0, 1.0, 1.0, 1.0], 'mx': [0.49999999999999994, 0.49999999999999994, 0.5, 0.5]}
Syndrome round 1
{'mz': [0.0, 1.0, 1.0, 0.0], 'mx': [0.5, 0.5, 0.5, 0.5]}
```

Here we can see that the probabilities of syndromes Z_0 and Z_3 which were 1 for the quiescent state, has changed now to 0, signaling the occurrence of Pauli X error. On the other hand, the probabilities for X syndromes were 0.5 for the quiescent state and even after the application of Y error and next syndrome measurement round, the probabilities remains unchanged for the X syndromes. The reason for this anomalous behaviour is that when a stabilizer measurement round is applied to $|00..0\rangle$ input data qubits, we get a quiescent state where each stabilizer block randomly collapses into either the 0 (+) or 1 (-) eigenstate of the stabilizer. Thus if we look at the counts we get from Qiskit, each of these bit strings counts represent a different quiescent state and for the case when the number of shots=1024, we get the statistics of counts as mentioned above. Here, we can see that the X syndrome qubits show a measurement value of 1 in some cases and 0 in other. If we look at the probability of X syndromes being 0 it would come nearly to 0.5. Thus even when the bits for each counts get flipped due to the error, the probability of the X syndrome qubits being 0 still remains 0.5.

```
Syndrome round 0
{'mz': [0.5, 0.5, 0.5, 0.5], 'mx': [1.0, 1.0, 1.0, 1.0]}
Syndrome round 1
{'mz': [0.5, 0.5, 0.5, 0.5], 'mx': [1.0, 0.0, 0.0, 1.0]}
```

I further tried to check what would happen if we initialize all the data qubits to $|++..+\rangle$ i.e. the X-basis. For this case, the X syndromes probability of being in 0 state after quiescent state is achieved is 1. Here we can see that after next round of syndrome measurements, the probabilities are flipped for X_1 and X_3 syndrome qubits thus signaling the occurrence of Z error. But for the case of Z syndromes, we are faced with the same problem we had with X syndromes for the $|00..0\rangle$ initialization of data qubits. Thus we can see that the graph analysis of error syndromes can be done individually for these syndromes but not together. A state initialized with $|00..\rangle$ would only be able to detect X errors and the state initialized with $|++..+\rangle$ would only be able to detect Z errors.

Moreover, we would run into an issue when the errors are probabilistic. As we do not have a measurement option to extract counts, such errors would not be possible to detect. One of the disadvantages of the density matrix simulator is its increased computational complexity for doing simulations as compared to a statevector based simulator. This was evident from the time taken to run the stabilizer codes which was much higher than that of Qasm simulator. The complexity would also have increased largely due to the availability of limited number of qubits due to which we had to perform stabilizer measurements sequentially. While in case of Qasm simulator, these stabilizer measurements can be carried out parallelly thus saving a lot of computation. The GPU access did improve the speed of the quantum simulation but it was only marginal. This could be due to the sequential stabilizer measurements.

A. Limitations of QSim

Upon the analysis written above, I faced the following limitations while working on QSim:

1) The absence of a measurement method to extract counts from the quantum state. In real world we know that, any quantum mechanical system when measured, collapses to a classical state and thus we repeat the experiment many times to obtain statistics about the state. A density matrix approach while doing any quantum simulation maybe a more robust method than the statevector simulator as we know that the statevector representation is not a complete representation of a quantum state. Thus the density matrix simulator might be able to perform better for mixed states and and for errors which does not preserve the purity of the states (incoherent errors). Thus in order to mimic the real quantum devices, along with the other measurement options such as ensemble measure, a probabilistic measurement which collapses the state according to the probability distribution obtained from the density matrix should be added to the simulator.

2) The second limitation is the absence of ability to selectively apply error models on certain types of gates as offered by Qiskit. QSim offers a wide range of error model options but these errors are applied automatically to all the gates. While this maybe a more realistic scenario, many quantum algorithms and noise analysis make assumptions about noise acting on only certain operations. For the case of surface codes, one of the assumptions made is that the gates applied for stabilizers are noise free and noise occurs between two syndrome measurements. Even though a naive assumption, most of the research in this field is based on this assumption as we are still far from achieving fault tolerant quantum computation. Thus a feature to selectively apply noise models to certain gates might help to catch up with the current algorithms.

3) Currently the number of qubits offered by QSim for simulation is also an issue as for algorithms and applications which required more than 12 qubits those cannot be implemented or in case such as surface code can be implemented at the expense of increased computation due to the sequential modelling.

As of now these are the 3 primary limitations I faced while working on QSim. Due to this limitations, a more rigorous analysis of surface codes such as logical error rate vs physical error rate could not be done at this point.

XII. Conclusion

We started by learning different quantum error correction codes and developed the notion of surface codes. We learned about how a problem of error finding can be mapped to a graph theoretic problem of minimum weight perfect matching which we learned through the Edmund Blossom's algorithm. We then implemented the surface code architecture in IBM Qiskit and simulated the surface code for basic Pauli errors. We discussed the notion and need for the virtual nodes and implemented the same in Qiskit. We also implemented correction operators for the detected errors. Then we discussed about the degeneracy in error paths which happen and compensated for the issue by altering the edge weights. We also implemented two error models on the surface code in Qiskit and plotted the performance of the surface code in terms of logical error rate vs physical error rate. We matched our shape and trends of the plot with the empirical relation derived in Reference[3]. Finally we tried to implement the surface code architecture in QSim, and we came across 3 major limitations due to which a rigorous analysis was not possible. We noted the 3 limitations and suggestions for improvements in the next iteration.

XIII. Future Outlook

A future scope for this project may include a rigorous analysis of how surface code behaves for different errors. for e.g. a comparison between coherent vs incoherent noise. Complexity analysis of the surface code method and the decoder can also be done. A possible direction also include trying for other options for decoder other than the MWPM decoder which have been proposed in the literature. Once the QSim version gets a count-based measurement option, these models can also be tested on QSim. A possible direction also can be to try to simulate some simple quantum algorithm requiring very few qubits on the surface code and test its viability.

XIV. Acknowledgements

I would like to thank my supervisor Prof. Sugata Gangopadhyay and my co-supervisor Prof. Ajay Wasan for guiding me throughout this project and motivating me to explore this nascent field of Quantum Error Correction and giving me a chance to work on QSim and give my feedback on it. I would also like to thank Jothishwaran for being a mentor for my project and addressing any doubts and difficulties.

XV. References

[1] Qiskit Documentation
[2] Networkx Documentation
[3] Surface codes: Towards practical large-scale quantum computation; Austin G. Fowler, Matteo Mariantoni, John M. Martinis, Andrew N. Cleland
[4] Tomita, Y. Svore, K. M. Low-distance surface codes under realistic quantum noise. Phys. Rev. A 90, 062320 (2014).
[5] https://github.com/yaleqc/qtcodes.
[6] Combinatorial Optimization; William J Cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver
[7] Multi-path Summation for Decoding 2D Topological Codes. Ben Criger and Imran Ashraf.

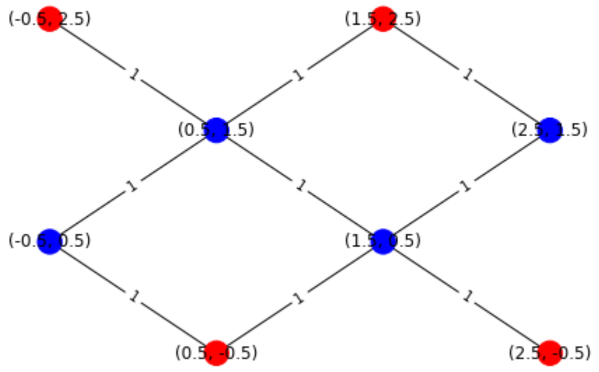


Fig. 10. X-Syndrome Graph. Blue nodes denotes normal syndromes, red denotes virtual.

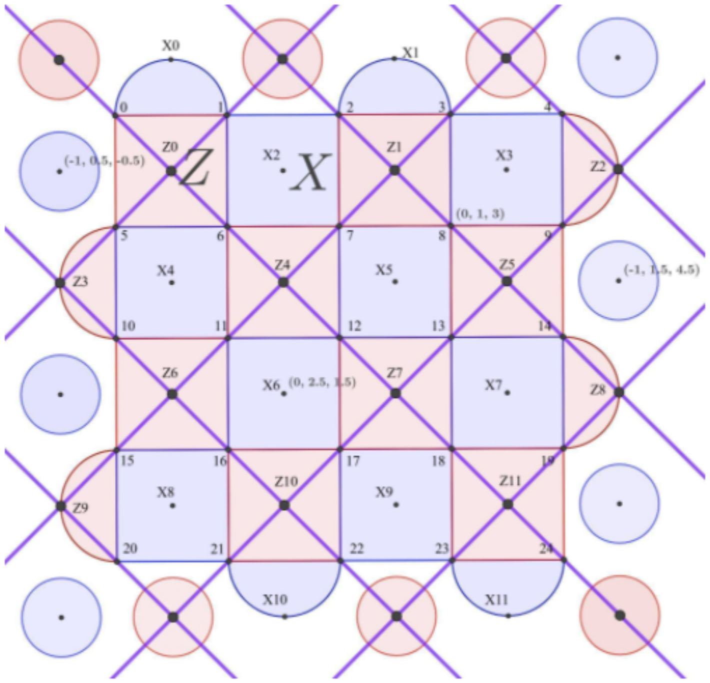


Fig. 11. $d = 5$ surface code with virtual nodes. Ref[5]

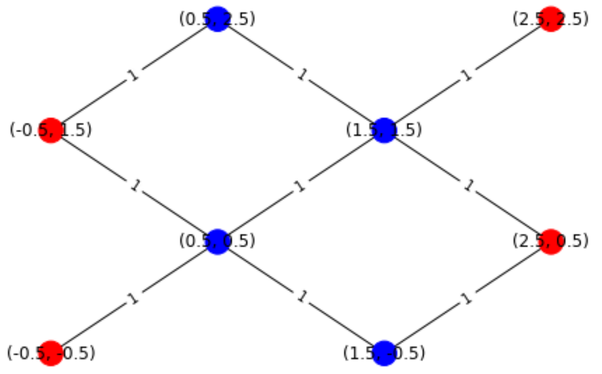


Fig. 12. Z-Syndrome Graph. Blue nodes denotes normal syndromes, red denotes virtual.